From Byzantine Consensus to BFT State Machine Replication: A Latency-Optimal Transformation

João Sousa and Alysson Bessani
University of Lisbon, Faculty of Sciences, LaSIGE
Lisbon, Portugal

***Abstract*—We present a new algorithm for state machine replication that is built around a leader-driven consensus primitive. This algorithm requires only two additional communication steps between replicas and clients if the consensus leader is correct and the system is synchronous, being thus the first latency-optimal transformation from Byzantine consensus to BFT state machine replication. We also discuss how to make simple modifications to leader-driven consensus algorithms in order to make them compatible with our transformation.**

***Keywords*-Byzantine fault tolerance; state machine replication; consensus; modularization.**

I. Introduction

Replication is a fundamental technique for implementing dependable services that are able to ensure integrity and availability despite the occurrence of faults and intrusions. State Machine Replication (SMR) [18], [26] is a popular replication method that enables a set of replicas (state machines) to execute the same sequence of operations for a service even if a fraction of the them are faulty.

A fundamental requirement of SMR is to make all client-issued requests to be totally ordered across replicas. Such requirement demands the implementation of a total order broadcast protocol, which is known to be equivalent the consensus problem [9], [16], [23]. Therefore, a solution to the consensus problem is in the core of any distributed SMR protocol.

In the last decade, many practical SMR protocols for the Byzantine fault model were published (e.g., [2], [7], [10], [17], [27], [28]). However, despite their efficiency, such protocols are *monolithic*: they do not separate clearly the consensus primitive from the remaining protocol.

From a theoretical point of view, many Byzantine fault-tolerant (BFT) total order broadcast protocols (the main component of a BFT SMR implementation) were built using black-box Byzantine consensus primitives (e.g., [6], [9], [16], [23]). This modularity simplifies the protocols, making them both easy to reason about and to implement. Unfortunately, these modular transformations plus the underlying consensus they use always require more communication steps than the aforementioned monolithic solutions.

Figure 1 presents the typical message pattern of modular BFT total order broadcast protocols when used to implement SMR. The key point of most of these transformations is the use of BFT reliable broadcast protocol [4] to disseminate client requests among replicas, ensuring they will be eventually proposed (and decided) in some consensus instance that defines the order of messages to be executed. As illustrated in Figure 1, the usual BFT reliable broadcast requires three communication steps [4].

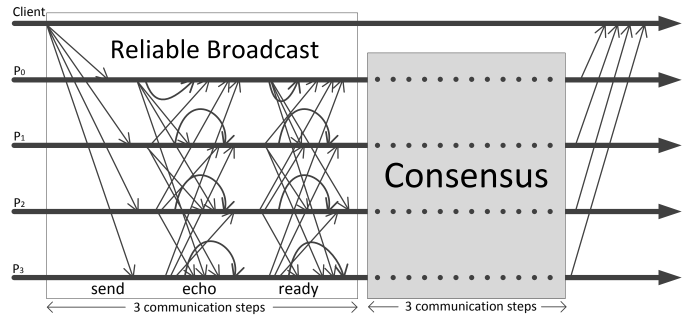

Figure 1. Modular BFT state machine replication message pattern for a protocol that uses reliable broadcast and a consensus primitives. This protocol is adapted from [23], when tolerating a single fault.

It is known that optimally resilient Byzantine consensus protocols cannot safely decide a value in two or less communication steps [13], [22]. This means that latency-optimal protocols for BFT SMR that use only $3f + 1$ replicas to tolerate f Byzantine faults (e.g., PBFT [7]) requires at least three communication steps for the consensus plus two extra steps to receive the request from the client and send a reply[1]. By the other hand, the protocol of Figure 1 requires at least six communication steps to totally order a message in the best-case, plus one more to send a reply to the client, making a total of seven steps.

Considering this gap, in this paper we investigate the following question: *Is it possible to obtain a BFT state machine replication protocol with an optimal number of communications steps (similar to PBFT), while explicitly using a consensus primitive at its core?* The main contribution of this work is a new transformation from Byzantine consensus to BFT state machine replication dubbed *Modular State Machine Replication* (Mod-SMaRt), which answers this question affirmatively. Mod-SMaRt implements SMR

[1]This excludes optimistic protocols that are very efficient in contention-free executions [2], [10], speculative protocols [17], protocols that rely on trusted components [28], and fast protocols that require more than $3f + 1$ replicas [22].

using a special Byzantine consensus primitive called *Validated and Provable Consensus* (VP-Consensus), which can be easily obtained by modifying existing leader-driven consensus algorithms (e.g., [5], [20], [22], [25], [29]). To our knowledge, MOD-SMART is the first modular BFT SMR protocol built over a well-defined consensus module which requires only the optimal number of communication steps, i.e., the number of communication steps of consensus plus two.

The core of our solution is the definition and use of the VP-Consensus as a *"grey-box" abstraction* that allows the modular implementation of SMR without using reliable broadcast, thus avoiding the extra communication steps required to safely guarantee that all requests arrive at all correct replicas. The monolithic protocols, on the other hand, avoid those extra steps by merging the reliable broadcast with the consensus protocol, being thus more complex. MOD-SMART avoids mixing protocols by using the rich interface exported by VP-Consensus, that allows it to handle request timeouts and, if needed, triggers internal consensus timeouts. The use of a VP-Consensus is a good compromise between modularity and efficiency, specially because this primitive can be easily implemented with simple modifications on several leader-driven partially-synchronous Byzantine consensus protocols [5], [20]–[22], [25], [29].

Although this work main contribution is theoretical, our motivation is very practical. MOD-SMART is implemented as one of the core modules of BFT-SMART [1], an open-source Java-based BFT SMR library in which modularity is treated as a first-class property.

The paper is organized in the following way. We first describe our system model and the problem we want to address in Sections II and III. The Validated and Provable Consensus primitive is discussed in Section IV. Next, Section V present the the MOD-SMART algorithms. Possible optimizations and additional considerations are discussed in Section VI. In Sections VII and VIII we put the related work in context and present our conclusions. Finally, all proofs that MOD-SMART implements SMR are described in the Appendix.

II. SYSTEM MODEL

We consider a system composed by a set of $n \geq 3f + 1$ replicas R, where a maximum of f replicas may be subject to *Byzantine faults*, and a set C with an unbounded (but finite) number of clients, which can also suffer Byzantine faults. A process (client or replica) is considered correct if it never deviates from its specification; otherwise, it is considered faulty.

Like in PBFT and similar protocols [7], [10], [17], [27], MOD-SMART does not require synchrony to assure *safety*. However, it requires synchrony to provide *liveness*. This means that, even in the presence of faults, correct replicas will never evolve into an inconsistent state; but the execution of the protocol is guaranteed to terminate only when the system becomes synchronous. Due to this, we assume an *eventually synchronous* system model [14]. In such model, the system operates asynchronously until some unknown instant, at which it will become synchronous. At this point, unknown time bounds for computation and communication will be respected by the system.

We further assume that all processes communicate through *reliable and authenticated point-to-point channels*, that can be easily implemented over fair links using retransmission and message authentication codes.

Finally, we assume the existence of cryptographic functions that provide digital signatures, message digests, and message authentication codes (MAC).

III. STATE MACHINE REPLICATION

The state machine replication model was first proposed in [18], and later generalized in [26]. In this model, an arbitrary number of client processes issue commands to a set of replica processes. These replicas implement a stateful service that changes its state after processing client commands, and sends replies to the clients that issued them. The goal of this technique is to make the state at each replica evolve in a consistent way, thus making the service completely and accurately replicated at each replica. In order to achieve this behavior, it is necessary to satisfy four properties:

1) If any two correct replicas r and r' apply operation o to state s, both r and r' will reach state s';
2) Any two correct replicas r and r' start with state s_0;
3) Any two correct replicas r and r' execute the same sequence of operations $o_0, ..., o_i$;
4) Operations from correct clients are always executed.

The first two requirements can be fulfilled without any distributed protocol, but the following two directly translates to the implementation of a total order broadcast protocol – which is equivalent to solving the consensus problem. MOD-SMART satisfy properties 3 and 4, assuming the existence of a VP-Consensus primitive and that the service being replicated respects properties 1 and 2.

IV. VALIDATED AND PROVABLE CONSENSUS

In this section we introduce the concept of *Validated and Provable Consensus* (VP-Consensus). By 'Validated', we mean the protocol receives a predicate γ together with the proposed value – which any decided value must satisfy. By 'Provable', we mean that the protocol generates a cryptographic proof Γ that certifies that a value v was decided in a consensus instance i. More precisely, a VP-Consensus implementation offers the following interface:

- *VP-Propose*(i, l, γ, v): proposes a value v in consensus instance i, with initial leader l and predicate γ;
- *VP-Decide*(i, v, Γ): triggered when value v with proof Γ is decided in consensus instance i;

- *VP-Timeout(i, l)*: used to trigger a timeout in the consensus instance i, and appoint a new leader process l.

Three important things should be noted about this interface. First, VP-Consensus assumes a leader-driven protocol, similar to any Byzantine Paxos consensus. Second, the interface assumes the VP-Consensus implementation can handle timeouts to change leaders, and a new leader is (locally) chosen after a timeout. Finally, we implicitly assume that all correct processes will invoke *VP-Propose* for an instance i using the same predicate γ.

Just like usual definitions of consensus [5], [9], [16], VP-Consensus respects the following properties:

- *Termination*: Every correct process eventually decides;
- *Integrity*: No correct process decides twice;
- *Agreement*: No two correct processes decide differently.

Moreover, two additional properties are also required:

- *External Validity*: If a correct process decides v, then $\gamma(v)$ is true;
- *External Provability*: If some correct process decides v with proof Γ in a consensus instance i, all correct process can verify that v is the decision of i using Γ.

External Validity was originally proposed by Cachin et al. [6], but we use a slightly modified definition. In particular, *External Validity* no longer explicitly demands validation data for proposing v, because such data is already included in the proposed value, as will be clear in Section V.

A. Implementation requirements

Even though our primitive offers the classical properties of consensus, the interface imposes some changes in its implementation. Notice that we are not trying to specify a new consensus algorithm; we are only specifying a primitive that can be obtained by making simple modifications to existing ones [5], [20]–[22], [25]. However, as described before, our interface assumes that such algorithms are leader-driven and assume the partially synchronous system model. Most Paxos-based protocols satisfy these conditions [5], [22], [25], [29], and thus can be used with MOD-SMART. In this section we present an overview of the required modifications on consensus protocols, without providing explanations for it. We will come back to the modifications in Section V-E, when it will become clear why they are required.

The first change is related to the timers needed in the presence of partial synchrony. To our knowledge, all published algorithms for such system model requires a timer to ensure liveness despite leader failures [5], [19], [22]. The primitive still needs such timer; but it will not be its responsibility to manage it. Instead, we invoke *VP-Timeout* to indicate to the consensus that a timeout has occurred, and it needs to handle it.

The second change is related to the assumption of a leader-driven consensus. To our knowledge, all the leader-driven algorithms in literature have deterministic mechanisms to select a new leader when sufficiently many of them suspect the current one. These suspicions are triggered by a timeout. A VP-Consensus implementation still requires the election of a new leader upon a timeout. However, the next leader will be defined by MOD-SMART, and is passed as an argument in the *VP-Propose* and *VP-Timeout* calls. Notice that these two requirements are equivalent to assuming the consensus protocol requires a leader election module, just like Ω failure detector, which is already used in some algorithms [5], [22].

The third change imposes the consensus algorithm to generate the cryptographic proof Γ to fulfill the *External Provability* property. This proof can be generated by signing the messages that can trigger a decision of the consensus[2]. An example of proofs would be a set of $2f + 1$ signed COMMIT messages in PBFT [7] or $\lceil (n + f + 1)/2 \rceil$ signed COMMITPROOF messages in Parametrized FaB [22].

Finally, we require each correct process running the consensus algorithm to verify if the value being proposed by the leader satisfies γ before it is accepted. Correct processes must only accept values that satisfy such predicate and discard others – thus fulfilling the *External Validity* property.

V. THE MOD-SMART ALGORITHM

In this section we describe MOD-SMART, our modular BFT state machine replication algorithm. The protocol is divided into three sub-algorithms: client operation, normal phase, and synchronization phase. The proofs that MOD-SMART satisfies the BFT state machine replication properties under our system model are presented in the Appendix.

A. Overview

The general architecture of a replica is described in Figure 2. MOD-SMART is built on top of a reliable and authenticated point-to-point communication substrate and a VP-Consensus implementation. Such module may also use the same communication support to exchange messages among processes. MOD-SMART uses VP-Consensus to execute a sequence of consensus instances, where in each instance i a batch of operations are proposed for execution, and the same proposed batch is decided on each correct replica. This is the mechanism by which we are able to achieve total order across correct replicas.

During normal phase, a log of the decided values is constructed based on the sequence of VP-Consensus executions. Each log entry contains the decided value, the *id* of the consensus instance where it was decided, and its associated proof. To simplify our design, MOD-SMART assumes each correct replica can execute concurrently only the current instance i and previous consensus instance $i-1$. All correct

[2]Due to the cost of producing digital signatures, the cryptographic proof can be generated with MAC vectors instead of digital signatures, just like in PBFT [7].

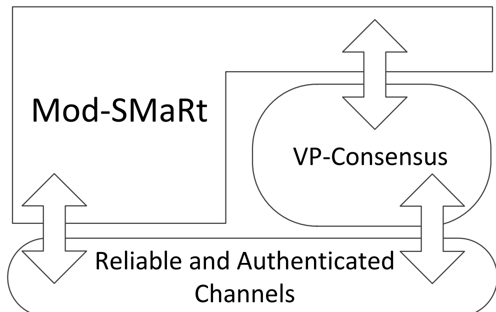

Figure 2. MOD-SMART replica architecture. The reliable and authenticated channels layer guarantee the delivery of point-to-point messages, while the VP-Consensus module is used to establish agreement on the message(s) to be delivered in an consensus instance.

replicas remain available to participate in consensus instance $i-1$, even if they are already executing i. This is required to ensure that if there is one correct replica running consensus $i-1$ but not i, there will be at least $n-f$ correct replicas executing $i-1$, which ensures the delayed replica will be able to finish $i-1$.

Due to the asynchrony of the system, it is possible that a replica receives messages for a consensus instance j such that $j > i$ (early message) or $j < i-1$ (outdated message). Early messages are stored in an out-of-context buffer for future processing while outdated messages are discarded. We do not provide pseudo-code for this mechanism, relying on our communication layer to deliver messages in accordance with the consensus instances being executed.

This pretty much describes the *normal phase* of the protocol, which is executed in the absence of faults and in the presence of synchrony. When these conditions are not satisfied, the *synchronization phase* might be triggered.

MOD-SMART makes use of the concept of *regencies*. This is equivalent to the *view* mechanism employed by PBFT and ViewStamped Replication [7], [24], where a single replica will be assigned as the leader for each regency. Such leader will be needed both in MOD-SMART, and in the VP-Consensus module. During each regency, the normal case operation can be repeated infinitely; during a synchronization phase, an unbounded (but finite) number of regency changes can take place, since the system will eventually become synchronous.

The avoidance of executing a reliable multicast before starting the Byzantine consensus may lead to two problems. First, a faulty leader may not propose messages from some client for ordering, making it starve. Second, a faulty client can send messages to all replicas but to the current (correct) leader, making other replicas suspect it for not ordering messages from this client. The solution for these problems is to suspect the leader only if the timer associated with a message expires twice, making processes forward the pending message to the leader upon the first expiration.

In case a regency change is needed (i.e., the leader is suspected), timeouts will be triggered at all replicas and the synchronization phase will take place. During this phase, MOD-SMART must ensure three properties: (1) a quorum of $n-f$ replicas must have the pending messages that caused the timeouts; (2) correct replicas must exchange logs to jump to the same consensus instance; and (3) a timeout is triggered in this consensus, proposing the same leader at all correct replicas (the one chosen during the regency change). Notice that MOD-SMART does not verify consensus values to ensure consistency: all these checks are done inside of the VP-Consensus module, after its timeout is triggered. This substantially simplifies faulty leader recovery by breaking the problem in two self-contained blocks: the state machine replication layer ensures all processes are executing the same consensus with the same leader while VP-Consensus deals with the inconsistencies within a consensus.

B. Client Operation

Algorithm 1 describes how the client invokes an operation in MOD-SMART. When a client wants to issue a request to the replicas, it sends a REQUEST message in the format specified (line 6). This message contains the sequence number for the request and the command issued by the client. The inclusion of a sequence number is meant to uniquely identify the command (together with the client id), and prevent replay attacks made by an adversary that might be sniffing the communication channel. A digital signature α_c is appended to the message to prove that such message was produced by client c. Although this signature is not required, its use makes the system resilient against certain attacks [3], [8].

The client waits for at least $f+1$ matching replies from different replicas, for the same sequence number (lines 9–11), and return the operation result.

Algorithm 1: Client-side protocol for client c.

```
1  Upon Init do
2      nextSeq = 0
3      Replies ← ∅

4  Upon Invoke(op) do
5      nextSeq = nextSeq + 1
6      send ⟨REQUEST, nextSeq, op⟩_{α_c} to R

7  Upon reception of ⟨REPLY, seq, rep⟩ from r ∈ R do
8      Replies ← Replies ∪ {⟨r, seq, rep⟩}
9      if ∃ seq, rep : |{⟨∗, seq, rep⟩ ∈ Replies}| > f
10         Replies ← Replies\{⟨∗, seq, rep⟩}
11         return rep
```

C. Normal Phase

The normal phase is described in Algorithm 2, and its message pattern is illustrated in Figure 3. The goal of this phase is to execute a sequence of consensus instances in each replica. The values proposed by each replica will be

a batch of operations issued by the clients. Because each correct replica executes the same sequence of consensus instances, the values decided in each instance will be the same in all correct replicas, and since they are batches of operations, they will be *totally ordered* across correct replicas. All variables and functions used by the replicas in Algorithms 2 and 3 are described in Table I.

Reception of client requests are processed in line 1-2 through procedure $RequestReceived$ (lines 20–24). Requests are only considered by correct replicas if the message contains a valid signature and the sequence number expected from this client (to avoid replay attacks), as checked in line 21. If a replica accepts an operation issued by a client, it stores it in the $ToOrder$ set, activating a timer associated with the request (lines 22–24). Notice that a message is also accepted if it is forwarded by other replicas (lines 18-19).

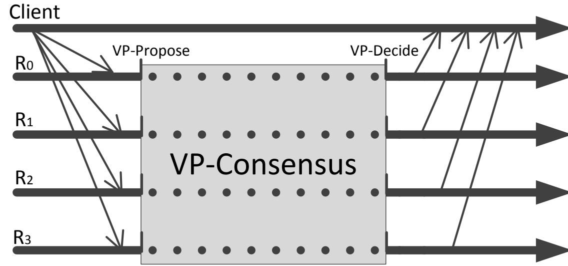

Figure 3. Communication pattern of MOD-SMART normal phase for $f = 1$. A correct client send an operation to all replicas, a consensus instance is executed to establish total order, the operation is executed, and a reply is sent to the client.

When the $ToOrder$ set contains some request to be ordered, there is no consensus being executed and the ordering of messages is not stopped (see next section), a sub-set of operations $Batch$ from $ToOrder$ is selected to be ordered (lines 3 and 4). The predicate *fair* ensures that all clients with pending requests will have approximately the same number of operations in a batch to avoid starvation. The replica will then create a consensus instance, using $Batch$ as the proposed value (lines 5 and 6). The predicate γ given as an argument in *VP-Propose* should return TRUE for a proposed value V if the following three conditions are met:

1) $fair(V)$ is TRUE (thus V is not an empty set);
2) Each message in V is either in the $ToOrder$ set of the replica or is correctly signed and contains the next sequence number expected from the client that issued the operation;
3) Each message in V contains a valid command with respect to the service implemented by MOD-SMART.

When a consensus instance decides a value (i.e., a batch of operations) and produces its corresponding proof (line 7), MOD-SMART will: store the batch of operations and its cryptographic proof in each replica log (line 11); cancel the timers associated with each decided request (line 14); deterministically deliver each operation contained in the batch to the application (line 16); and send a reply to the client that requested the operation with the corresponding response (line 17). Notice that if the algorithm is stopped (possibly because the replica is running a synchronization phase, see next section), decided messages are stored in a $Decided$ set (lines 8 and 9), instead of being executed.

Algorithm 2: Normal phase at replica r.

```
1  Upon reception of m = ⟨REQUEST, seq, op⟩α_c from c ∈ C do
2  |  RequestReceived (m)

3  Upon (toOrder ≠ ∅) ∧ (currentCons = −1) ∧ (¬stopped) do
4  |  Batch ← X ⊆ ToOrder : |X| ≤ maxBatch ∧ fair(X)
5  |  currentCons ← hCons(DecLog).i + 1
6  |  VP-Propose (currentCons, creg mod R, γ, Batch)

7  Upon VP-Decide⟨i, Batch, Proof⟩ do
8  |  if stopped
9  |  |  Decided ← Decided ∪ {⟨i, Batch, Proof⟩}
10 |  else
11 |  |  DecLog ← DecLog ∪ {⟨i, Batch, Proof⟩}
12 |  |  if currentCons = i then currentCons ← −1
   |  |  // Deterministic cycle
13 |  |  foreach m = ⟨REQUEST, seq, op⟩α_c ∈ Batch do
14 |  |  |  cancelTimers ({m})
15 |  |  |  ToOrder ← ToOrder\{m}
16 |  |  |  rep ← execute(op)
17 |  |  |  send ⟨REPLY, seq, rep⟩ to c

18 Upon reception of ⟨FORWARDED, M⟩ from r' ∈ R do
19 |  ∀m ∈ M : RequestReceived (m)

20 Procedure RequestReceived(m)
21 |  if lastSeq[c] + 1 = m.seq ∧ validSig(m)
22 |  |  ToOrder ← ToOrder ∪ {m}
23 |  |  if ¬stopped then activateTimers({m}, timeout)
24 |  |  lastSeq[c] ← m.seq
```

D. Synchronization Phase

The synchronization phase is described in Algorithm 3, and its message pattern is illustrated in Figure 4. This phase aims to perform a regency change and force correct replicas to synchronize their states and go to the same consensus instance. It occurs when the system is passing through a period of asynchrony, or there is a faulty leader that does not deliver client requests before their associated timers expire. This phase is started when a *timeout* event is triggered for a sub-set M of pending messages in $ToOrder$ (line 1).

When the timers associated with a set of requests M are triggered for the first time, the requests are forwarded to all replicas (lines 2 and 3). This is done because a faulty client may have sent its operation only to some of the replicas, therefore starting a consensus in less than $n - f$ of them (which is not sufficient to ensure progress, and therefore will cause a timeout in these replicas). This step forces such requests to reach all correct replicas, without forcing a leader change.

If there is a second timeout for the same request, the replica starts a regency change (line 4). When a regency

Table I
VARIABLES AND FUNCTIONS USED IN ALGORITHMS 2 AND 3.

Variables		
Name	Initial Value	Description
$timeout$	INITIAL_TIMEOUT	Timeout for a message to be ordered.
$maxBatch$	MAX_BATCH	Maximum number of operations that a batch may contain.
$creg$	0	Replica current regency.
$nreg$	0	Replica next regency.
$currentCons$	-1	Current consensus being executed.
$DecLog$	$\emptyset$	Log of all decided consensus instances and their proofs.
$ToOrder$	$\emptyset$	Pending messages to be ordered.
Tmp	$\emptyset$	Messages collected in a STOP messages.
$Decided$	$\emptyset$	Decision values obtained during the synchronization phase.
$stopped$	FALSE	Indicates if the synchronization phase is activated.
$lastSeq[1..\infty]$	$\forall c \in C : lastSeq[c] \leftarrow 0$	Last request sequence number used by each client c.
$ChangeReg[1..\infty]$	$\forall g \in N : ChangeReg[g] \leftarrow \emptyset$	Replicas that want a change to regency g.
$Data[1..\infty]$	$\forall g \in N : Data[g] \leftarrow \emptyset$	Signed STOPDATA messages collected by the leader during change to regency g.
$Sync[1..\infty]$	$\forall g \in N : Sync[g] \leftarrow \emptyset$	Set of Logs sent by the leader to all replicas during regency change g.

Functions	
Interface	Description
`activateTimers(Reqs, timeout)`	Creates a timer for each request in *Reqs* with value *timeout*.
`cancelTimers(Reqs)`	Cancels the timer associated with each request in *Reqs*.
`execute(op)`	Makes the application execute operation *op*, returning the result.
`validSig(req)`	Returns TRUE if request *eq* is correctly signed.
`noGaps(Log)`	Returns TRUE if sequence of consensus *Log* does not contain any gaps.
`validDec(decision)`	Returns TRUE if *decision* contains a valid proof.
`hCons(Log)`	Returns the consensus instance from *Log* with highest id.
`hLog(Logs)`	Returns the largest log contained in *Logs*.

change begins in a replica, the processing of decisions is stopped (line 7), the timers for all pending requests are canceled (line 9) and a STOP message is sent to all replicas (line 10). This message informs other replicas that a timeout for a given set of requests has occurred. When a replica receives more than f STOP messages requesting the next regency to be started (line 15), it begins to change its current regency using the valid messages in Tmp (line 16). This procedure ensures that a correct replica starts a view change as soon as it knows that at least one correct replica started it, even if no timeout was triggered locally.

When a replica receives more than $2f$ STOP messages, it will install the next regency (lines 19 and 20). It is necessary to wait at least $2f+1$ messages to make sure that eventually all correct replicas will install the next regency. Following this, the timers for all operations in the $ToOrder$ set will be re-activated and a new leader will be elected (lines 21–23).

After the next regency is installed, it is necessary to force all replicas to go to the same state (i.e., synchronize their logs and execute the logged requests) and, if necessary, start the consensus instance. To accomplish this, all replicas send a STOPDATA message to the new regency leader, providing it with their decision log (line 23). As long as the proof associated with each decided value is valid and there is no consensus instance missing, the leader will collect these messages (lines 26 and 27). This is necessary because it proves that each consensus instances has decided some batch of operations (which will be important later). When at least $n-f$ valid STOPDATA messages are received by the leader, it will send a SYNC message to all replicas, containing all the information gathered about their decided instances in at least $n-f$ replicas (lines 28 and 29).

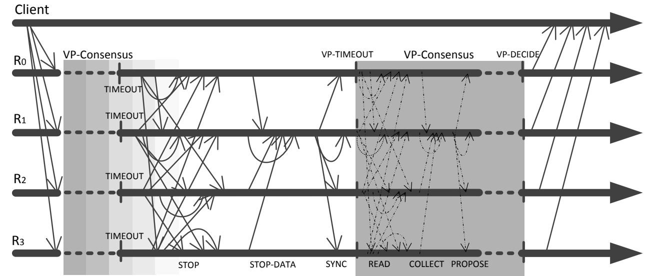

Figure 4. Communication steps of synchronization phase for $f = 1$. This phase is started when the timeout for a message is triggered for a second time, and can run simultaneously with VP-Consensus. Dashed arrows correspond to messages of the VP-Consensus protocol.

When a replica receives a SYNC message, it executes the same computations performed by the leader (lines 31–35) to verify if the leader has gathered and sent valid information. If the leader is correct, after receiving the same SYNC message, all correct replicas will choose the same highest log (line 36) and resume decision processing (line 37). All correct replicas will evolve into the same state as they deliver the value of each consensus instance that was already decided in other replicas (lines 40 and 41) and either trigger a timeout in the VP-Consensus being executed (line 42 and 43) or make everything ready to start a new consensus instance (line 44).

E. Reasoning about the Consensus Modifications

As we mentioned in Section IV-A, the VP-Consensus primitive does not need to start and stop timers, since our state machine algorithm already does that. Due to this, the VP-Consensus module only needs to be notified by the state

Algorithm 3: Synchronization phase at replica r.

```
1  Upon timeout for requests M do
2      M_first ← {m ∈ M : first timeout of m}
3      if M_first ≠ ∅ then send ⟨FORWARDED, M⟩ to R
4      else if M\M_first ≠ ∅ then StartRegChange (M\M_first)

5  Procedure StartRegChange(M)
6      if nreg = creg
7          stopped ← TRUE
8          nreg ← creg + 1
9          cancelTimers(ToOrder) // Cancel all timers
10         send ⟨STOP, nreg, M⟩ to R

11 Upon reception of ⟨STOP, reg, M⟩ from r' ∈ R do
12     if reg = creg + 1
13         Tmp ← Tmp ∪ M
14         ChangeReg[reg] ← ChangeReg[reg] ∪ {r'}
15         if |ChangeReg[reg]| > f
16             M' ← {m ∈ Tmp : m.seq > lastSeq[m.c]∧
                       validSig(m)}
17             StartRegChange (M')
18             ToOrder ← ToOrder ∪ M'
19             if |ChangeReg[reg]| > 2f ∧ nreg > creg
20                 creg ← nreg
21                 activateTimers (ToOrder, timeout)
22                 leader ← creg mod n
23                 send ⟨STOPDATA, reg, DecLog⟩_{α_r} to leader

24 Upon recept. of m = ⟨STOPDATA, creg, Log⟩_{α_{r'}} from r' ∈ R do
25     if creg mod n = r
26         if (noGaps(Log)) ∧ (∀d ∈ Log : validDec(d))
27             Data[creg] ← Data[creg] ∪ {m}
28         if |Data[creg]| ≥ n − f
29             send ⟨SYNC, creg, Data[creg]⟩ to R

30 Upon reception of ⟨SYNC, creg, Proofs⟩ from r' ∈ R do
31     if (nreg = creg) ∧ (creg mod n = r') ∧ ProofCons[creg] = ∅
32         foreach ⟨STOPDATA, creg, Log⟩_{α_{r''}} ∈ Proofs do
33             if (noGaps(Log)) ∧ (∀d ∈ Log : validDec(d))
34                 Sync[creg] ← Sync[creg] ∪ {⟨r'', Log⟩}
35         if |Sync[creg]| ≥ n − f
36             Log ← hLog(Sync[creg] ∪ {⟨r, DecLog⟩}) ∪ Decided
37             stopped ← FALSE
38             Decided ← ∅
39             Tmp ← ∅
               // Deterministic cycle
40             foreach ⟨i', B, P⟩ ∈ Log : i' > hCons(DecLog).i do
41                 Trigger VP-Decide⟨i', B, P⟩
42             if currentCons = hCons(Log).i + 1
43                 VP-Timeout (currentCons, creg mod R)
44             else currentCons = −1
```

machine algorithm when it needs to handle a timeout. This is done by invoking *VP-Timeout* for a consensus i, at the end of a synchronization phase (line 43 of Algorithm 3). The *VP-Timeout* operation also receives as an argument the new leader the replica should rely on. This is needed because we assume a leader-driven consensus, and such algorithms tend to elect the leader in a coordinated manner. But when a delayed replica jumps from an old consensus to a consensus i during the synchronization phase, it will be out-of-sync with respect to the current regency, when compared with the majority of replicas that have already started consensus i during the normal phase. For this reason, we need to explicitly inform VP-Consensus about the new leader.

Let us now discuss why the *External Validity* is required for MOD-SMART. The classic *Validity* property would be sufficient in the crash fault model, because processes are assumed to fail only by stopping, and will not propose invalid values; however, in the Byzantine fault model such behavior is permitted. A faulty process may propose an invalid value, and such value might be decided. An example of such value can be an empty batch. This is a case that can prevent progress within the algorithm. By forcing the consensus primitive to decide a value that is useful for the algorithm to keep making progress, we can prevent such scenario from occurring, and guarantee liveness as long as the execution is synchronous.

Finally, it should now be clear why the *External Provability* property is necessary: in the Byzantine fault model, replicas can lie about which consensus instance they have actually finished executing, and also provide a fake/corrupted decision value if a synchronization phase is triggered. By forcing the consensus primitive to provide a proof, we can prevent faulty replicas from lying. The worst thing a faulty replica can do is to send old proofs from previous consensus. However, since MOD-SMART requires at least $n - f$ logs from different replicas, there will be always more than f up-to-date correct replicas that will provide their most recent consensus decision.

VI. OPTIMIZATIONS

In this section we discuss a set of optimizations for efficient MOD-SMART implementation. The first important optimization is related with bounding the size of the decision log. In MOD-SMART, such log can grow indefinitely, making it inappropriate for real systems. To avoid this behavior we propose the use of checkpoints and state transfer. Checkpoints would be performed periodically in each replica: after some number *D* of decisions are delivered, the replica request the state from the application, save it in memory or disk, and clear the log up to this point[3]. If in the end of a synchronization phase a replica detects a gap between the latest decision of its own log, and the latest decision of the log it chose, it invokes a state transfer protocol. Such protocol would request from the other replicas the state that was saved in their latest checkpoint. Upon the reception of $f + 1$ matching states from different replicas, the protocol would force the application to install the new state, and resume execution.

The second optimization aims to avoid the computational cost of generating and verifying digital signatures in the protocol critical path: client requests and VP-Consensus proofs (to satisfy External Provability) can be signed using MAC vectors instead of digital signatures, as done in PBFT.

[3]Notice that, differently from the PBFT checkpoint protocol [7], MOD-SMART checkpoints are local operations.

However, in the case of client requests, this results in a less robust state machine implementation vulnerable to certain performance degradation attacks [3], [8].

If we use VP-Consensus based on a Byzantine consensus algorithm matching the generalization given in [20], and employ the optimizations just described, MOD-SMART matches the message pattern of PBFT in synchronous executions with correct leaders, requiring thus same number of communication steps and cryptographic operations. This is exactly what was done in BFT-SMART [1], an implementation of optimized MOD-SMART using the Byzantine consensus protocol described in [5].

VII. RELATED WORK

Byzantine Fault Tolerance has gained wide-spread interest among the research community ever since Castro and Liskov showed that state machine replication can be practically accomplished for such fault model [7]. Their algorithm, best known as PBFT (Practical Byzantine Fault Tolerance) requires $3f + 1$ replicas to tolerate f Byzantine faults and is live under the partial synchronous system model [14] (no synchrony is needed for safety). PBFT is considered the baseline for all BFT protocols published afterwards.

One of the protocols published following PBFT was Query/Update (Q/U) [2], an optimistic quorum-based protocol that presents better throughput with larger number of replicas than other agreement-based protocols. However, given its optimistic nature, Q/U performs poorly under contention, and requires $5f + 1$ replicas. To overcome these drawbacks, Cowling et al. proposed HQ [10], a hybrid Byzantine fault-tolerant SMR protocol similar to Q/U in the absence of contention. However, unlike Q/U, HQ only requires $3f+1$ replicas and relies on PBFT to resolve conflicts when contention among clients is detected. Following Q/U and HQ, Kotla et al. proposed Zyzzyva [17], a speculative Byzantine fault tolerant protocol, which is considered to be one of the fastest BFT protocol up to date. It is worth noticing that all these protocols tend to be more efficient than PBFT because they avoid the complete execution of a consensus protocol in the expected normal case, relying on it only to solve exceptional cases.

Guerraoui et al. [15] proposed a well-defined modular abstraction unifying the optimizations proposed by previous protocols through composition, making it easy to design new protocols that are optimal in well-behaved executions (e.g., synchrony, absence of contention, no faults), but revert to PBFT if such nice behavior does not hold. However, the modularity proposed is at state machine replication level, in the sense that each module provides a way to totally order client requests under certain conditions, and does not suggest any clear separation between total order broadcast and consensus.

The relationship between total order broadcast and consensus for the Byzantine fault model is studied in many papers. Cachin et al. [6] show how to obtain total order broadcast from consensus provided that the latter satisfy the *External Validity* property, as needed with MOD-SMART. Their transformation requires an echo broadcast plus public-key signature, adding thus at least two communication steps (plus the cryptography delay) to the consensus protocol. Correia et al. [9] proposed a similar reduction without relying on public-key signatures, but using a reliable broadcast and a multi-valued consensus that satisfies a validity property different from Cachin's. The resulting transformation adds at least three communication steps to the consensus protocol in the best case. In a very recent paper, Milosevic et al. [23] take in consideration many variants of the Byzantine consensus *Validity* property proposed in the literature, and show which of them are sufficient to implement total order broadcast. They also prove that if a consensus primitive satisfy the *Validity* property proposed in [11], then it is possible to obtain a reduction of total order broadcast to consensus with constant time complexity – which is not the case of the previous reductions in [6], [9]. However, their transformation still requires a reliable broadcast, and thus adds at least three communication steps to the consensus protocol. Doudou et al. [12] show how to implement BFT total order broadcast with a weak interactive consistency (WIC) primitive, in which the decision comprises a vector of proposed values, in a similar way to a vector consensus (see, e.g., [9]). They argue that the WIC primitive offers better guarantees than a Byzantine consensus primitive, eliminating the issue of the *Validity* property of consensus. The overhead of this transformation is similar to [6]: echo broadcast plus public-key signature.

All these works provide reductions from total order broadcast to Byzantine consensus by constructing a protocol stack that does not take into account the implementation of the consensus primitive; they only specify which properties such primitive should offer — in particular, they require some strong variant of the *Validity* property. MOD-SMART requires both a specific kind of *Validity* property, as well as a richer interface, as defined by our VP-Consensus abstraction. The result is a transformation that adds at most one communication step to implement total order broadcast, thus matching the number of communication steps of PBFT at the cost of using such gray-box consensus abstraction.

There are many works dedicated to generalize the algorithms of consensus. Lampson proposed an abstract Paxos algorithm, from which several other versions of Paxos can be derived (e.g., Byzantine, classic, and disk paxos) [20]. Another generalization of Paxos-style protocols is presented in [21], where the protocol is reduced to a write-once register satisfying a special set properties. Implementations of such register are given for different system and failures models. Rütti et al. extends these works in [25], where they propose a more generic construction than in [20], and identify three classes of consensus algorithms. Finally, Cachin proposes

a simple and elegant modular decomposition of Paxos-like protocols [5] and show how to obtain implementations of consensus tolerating crash or Byzantine faults based in the factored modules. All these works aim to modularize Paxos either for implementing consensus [5], [21], [25] or state machine replication [20] under different assumptions; our work, on the other hand, aims at using a special kind of consensus to obtain a BFT state machine replication.

VIII. Conclusion

Despite the existence of several works providing efficient BFT state machine replication, none of them encapsulate the agreement within a consensus primitive, being thus monolithic. On the other hand, all published modular protocol stacks implementing BFT total order broadcast from Byzantine consensus require a number of communication steps greater than all practical BFT SMR. We bridge this gap by presenting MOD-SMART, a latency- and resiliency-optimal BFT state machine replication algorithm that achieves modularity using a well-defined consensus primitive. To achieve such optimality, we introduce the *Validated and Provable Consensus* abstraction, which can be implemented by making simple modifications on existing consensus protocols. The protocol here presented is currently in use in BFT-SMART, an open-source BFT SMR library [1].

Acknowledgments: We warmly thank Bruno Vavala, João Craveiro, Pedro Costa and the EDCC'12 anonymous reviewers for their feedback on early versions of this paper. This work was partially supported by the EC through project TCLOUDS (FP7/2007-2013, ICT-257243) and also by FCT through project ReD (PTDC/EIA-EIA/109044/2008) and the Multiannual (LaSIGE) and CMU-Portugal Programmes.

References

[1] BFT-SMaRt: High-performance byzantine fault-tolerant state machine replication. http://code.google.com/p/bft-smart/.

[2] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proc. of the 20th ACM Symp. on Operating Systems Principles - SOSP'05*, 2005.

[3] Y. Amir, B. Coan, J. Kirsch, and J. Lane. Byzantine replication under attack. In *Proc. of the Int. Conf. on Dependable Systems and Networks – DSN'08*, 2008.

[4] G. Bracha. An asynchronous $\lfloor (n-1)/3 \rfloor$-resilient consensus protocol. In *Proc. of the 3rd Annual ACM Symp. on Principles of Distributed Computing - PODC'84*, 1984.

[5] C. Cachin. Yet another visit to Paxos. Technical report, IBM Research Zurich, 2009.

[6] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and efficient asynchronous broadcast protocols. In *Proc. of the 21st Annual Int. Cryptology Conf. on Advances in Cryptology – CRYPTO'01*, 2001.

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

[8] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Proc. of the 6th USENIX Symp. on Networked Systems Design & Implementation – NSDI'09*, 2009.

[9] M. Correia, N. F. Neves, and P. Veríssimo. From consensus to atomic broadcast: Time-free Byzantine-resistant protocols without signatures. *The Computer Journal*, 49(1):82–96, 2006.

[10] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ-Replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Proc. of 7th Symp. on Operating Systems Design and Implementation - OSDI'06*, 2006.

[11] D. Dolev and E. N. Hoch. Constant-space localized Byzantine consensus. In *Proc. of the 22nd Int. Symp. on Distributed Computing – DISC '08*, 2008.

[12] A. Doudou, B. Garbinato, and R. Guerraoui. *Dependable Computing Systems Paradigms, Performance Issues, and Applications*, chapter Tolerating Arbitrary Failures with State Machine Replication, pages 27–56. Wiley, 2005.

[13] P. Dutta, R. Guerraoui, and M. Vukolić. Best-case complexity of asynchronous Byzantine consensus. Technical report, School of Computer and Comm. Sciences, EPFL, 2005.

[14] C. Dwork, N. A. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–322, 1988.

[15] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić. The next 700 BFT protocols. In *Proc. of the 5th European Conf. on Computer systems – EuroSys'10*, 2010.

[16] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Cornell University, 1994.

[17] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *Proc. of the 21st ACM Symp. on Operating Systems Principles - SOSP'07*, 2007.

[18] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[19] L. Lamport. The part-time parliament. *ACM Transactions Computer Systems*, 16(2):133–169, 1998.

[20] B. Lampson. The ABCD's of Paxos. In *Proc. of the 20th Annual ACM Symp. on Principles of Distributed Computing – PODC'01*, 2001.

[21] H. C. Li, A. Clement, A. S. Aiyer, and L. Alvisi. The Paxos register. In *Proc. of the 26th IEEE Int. Symp. on Reliable Distributed Systems - SRDS'07*, 2007.

[22] J.-P. Martin and L. Alvisi. Fast Byzantine consensus. *IEEE Trans. on Dependable and Secure Computing*, 3(3):202–215, 2006.

[23] Z. Milosevic, M. Hutle, and A. Schiper. On the reduction of atomic broadcast to consensus with Byzantine faults. In *Proc. of the 30th IEEE Int. Symp. on Reliable Distributed Systems – SRDS'11*, 2011.

[24] B. M. Oki and B. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proc. of the 7th Annual ACM Symp. on Principles of Distributed Computing – PODC'88*, 1988.

[25] O. Rütti, Z. Milosevic, and A. Schiper. Generic construction of consensus algorithms for benign and Byzantine faults. In *Proc. of the Int. Conf. on Dependable Systems and Networks - DSN'10*, 2010.

[26] F. B. Schneider. Implementing fault-tolerant service using the state machine aproach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[27] G. Veronese, M. Correia, A. Bessani, and L. Lung. Spin one's wheels? Byzantine fault tolerance with a spinning primary. In *Proc. of the 28th IEEE Int. Symp. on Reliable Distributed Systems – SRDS'09*, 2009.

[28] G. Veronese, M. Correia, A. Bessani, L. Lung, and P. Verissimo. Efficient Byzantine fault tolerance. *IEEE Transactions on Computers*, 2011.

[29] P. Zielinski. Paxos at war. Technical report, University of Cambridge Computer Laboratory, 2004.

APPENDIX

In this Appendix we prove the correctness of MOD-SMART. The first theorem proves the safety of the protocol, i.e., that all correct replicas process the same sequence of operations.

Theorem 1 *Let p be the correct replica that executed the highest number of operations up to a certain instant. If p executed the sequence of operations $o_1, ..., o_i$, then all other correct replicas executed the same sequence of operations or a prefix of it.*

Proof: Assume that r and r' are two distinct correct replicas and o and o' are two distinct operations issued by correct client(s). Assume also that b and b' are the batches of operations were o and o' were proposed, respectively. For r and r' to be able to execute different sequences of operations that are not prefix-related, at least one of three scenarios described below needs to happen.

(1) *VP-Consensus instance i decides b in replica r, and decides b' in r'*. Since in this scenario the same sequence number can be assigned to 2 different batches, this will cause o and o' to be executed in different order by r and r'. But by the *Agreement* and *Integrity* properties of VP-Consensus, such behavior is impossible; *Agreement* forbids two correct processes to decide differently, and *Integrity* prevents any correct process from deciding more than once.

(2) *b is a batch decided at VP-Consensus instance i in both r and r', but the operations in b are executed in different orders at r and r'*. This behavior can never happen because Algorithm 2 (line 13) forces the operations to be ordered deterministically for execution, making these operations be executed in the same order by these replicas.

(3) *Replica r executes sequence of operations $S = o_0, ..., o_s$ and r' executes a subset of operations in S (but not all of them), preserving their relative order*. This will result in a gap in the sequence of operations executed by r'. From Algorithm 2, we can see that any operation is executed only after the *VP-Decide* event is triggered. This event is triggered either when a consensus instance decides a batch in line 7 — which occurs during the normal phase — or when invoked by Algorithm 3 in line 41. For simplicity, let us assume that each batch of messages contains a single operation. In the absence of a synchronization phase, lines 3-6 of Algorithm 2 ensure that any consensus instance i is only started after instance $i-1$ is decided. This forces any correct process to execute the same sequence of operations.

Lets now reason about the occurrence of a synchronization phase. In such case, r' will create the *Log* set at Algorithm 3, and then trigger the *Decide* event for each decision contained in *Log* (lines 36-41). *Log* is created using operations from both the most up-to-date log contained in the SYNC message or from the replica's $DecLog$ (line 36). Let us assume that r' did not execute S before entering the synchronization phase. Let us further consider $T = \{o_{s+1}, ..., o_t\}$ with $t \geq s+1$ to be a sub-sequence of operations that have been executed by r. For r' to skip S in this situation, it is necessary that *Log* contains U (such that U is a prefix of T) but does not contain S, and that r' triggers *VP-Decide* at Algorithm 3 (line 41) for each operation in U. This situation can never happen since r' is correct, and the algorithm ensures *Log* is constructed using valid operations (satisfying `validDec`) from decision logs that contain no gaps, i.e., satisfy the `noGaps` predicate. Furthermore, each decision in *Log* also satisfied the `validDec` predicate, so r' will not pick a sequence of operations with invalid decisions. Finally, since r' is correct, *DecLog* will already satisfy these predicates. This means that either: (1) both S and U are in *Log*; (2) only S is in *Log*, (3) neither sequence is in *Log*. Therefore, if *Log* contains U, then it must also contain S, and both sequences will be executed in r'. ■

Next lemmata prove several MOD-SMART properties. These lemmata use the following additional definitions. We say that an operation issued by a client c *completes* when c receives the same response for the operation from at least $f+1$ different replicas. We also consider that an operation sent by a client is *valid* if it is correctly signed and if its

sequence number is greater than the sequence number of the last operation sent by that client.

Lemma 1 *If a correct replica receives a valid operation o, eventually all correct replicas receive o.*

Proof: We have to consider four possibilities concerning the client behavior and the system synchrony.

(1) *Correct client and synchronous system.* In this case, the client will send its operation to all replicas, and all correct ones will receive the operation and store it in the $ToOrder$ set before a timeout occurs (Algorithm 2, line 1-2 plus procedure *RequestReceived*).

(2) *Faulty client and synchronous system.* Assume a faulty client sends a valid operation o to at least one correct replica r. Such replica will initiate a timer t and start a consensus instance i (Algorithm 2, lines 1 and 2 plus procedure *RequestReceived*). However, not enough replicas (less than $n - f$) will initialize a consensus instance i. Because of this, the timeout for t will eventually be triggered on the correct replicas that received it (Algorithm 3, line 1), and o will be propagated to all other replicas (lines 2 and 3). From here, all correct ones will store the operation in the $ToOrder$ set (Algorithm 2, lines 18 and 19 plus procedure *RequestReceived*).

(3) *Correct client and asynchronous system.* In this case, a correct replica might receive an operation, but due to delays in the network, it will trigger its timeout before the client request reaches all other replicas. Such timeout may be triggered in a correct replica and the message will be forwarded to other replicas. Moreover, since the client is correct, the operation will eventually be delivered to all correct replicas and they will store it in their $ToOrder$ set.

(4) *Faulty client and asynchronous system.* This case is similar to 3), with the addition that the client may send the request to as few as one correct replica. But like it was explained in 2), the replica will send the operation to all other replicas upon the first timeout. This ensures that eventually the operation will be delivered to all correct replicas and each one will store it in the $ToOrder$ set.

Therefore, if a correct replica receives a valid operation o, them all correct replicas eventually receive o. ■

Lemma 2 *If a synchronization phase for regency g starts with a faulty leader l, then eventually synchronization phase for regency $g' > g$ starts with correct leader $l' \neq l$.*

Proof: Each synchronization phase uses a special replica called 'leader', that receives at least $n - f$ STOPDATA messages and sends a single SYNC message to all replicas in the system (Algorithm 3, lines 24-29). If such leader is faulty, it can deviate from the protocol during this phase. However, its behavior is severely constrained since it can not create fake logs (such logs are signed by the replicas that sent them in the STOPDATA messages). Additionally, each entry in the log contains the proof associated with each value decided in a consensus instance, which in turn prevents the replicas from providing incorrect decision values. Because of this, the worst a faulty leader can do, is:

(1) *Not send the* SYNC *message to a correct replica.* In this case, the timers associated with the operations waiting to be ordered will eventually be triggered at all correct replicas - which will result in a new iteration of the synchronization phase.

(2) *Send two different* SYNC *messages to two different sets of replicas.* This situation can happen if the faulty leader waits for more than $n - f$ STOPDATA messages from replicas. The leader will then create sets of logs L and L', such that each set has exactly $n - f$ valid logs, and sends L to a set of replicas Q, and L' to another set of replicas Q'. In this scenario, Q and Q' may create different logs at line 36 of Algorithm 3, and resume normal phase at different consensus instances. But in order to ensure progress, at least $n - f$ replicas need to start the same consensus instance (because the consensus primitive needs these minimum amount of correct processes). Therefore, if the faulty leader does not send the same set of logs to a set Q_{n-f} with at least $n - f$ replicas that will follow the protocol (be them either all correct or not), the primitive will not make progress. Hence, if the faulty leader wants to make progress, it has to send the same set of logs to at least $n - f$ replicas. Otherwise, timeouts will occur, and a new synchronization phase will take place.

Finally, in each synchronization phase a new leader is elected. The new leader may be faulty again, but in that case, the same constraints explained previously will also apply to such leader. Because of this, when the system reaches a period of synchrony, after at most f regency changes, there is going to be a new leader that is correct, and progress will be ensured. ■

Lemma 3 *If one correct replica r starts consensus i, eventually $n - f$ replicas start i.*

Proof: We need to consider the behavior of the clients that issue the operations that are ordered by the consensus instance (correct or faulty), the replicas that start such instance (correct or faulty), and the state of the system (synchronous or asynchronous).

We can observe from Algorithm 2 that an instance is started after selecting a batch of operations from the $ToOrder$ set (lines 4-6). This set stores valid operations issued by clients. From Lemma 1, we know that a valid operation will eventually be received by all correct replicas, as long as at least one of those replicas receives it. Therefore, it is not necessary to consider faulty clients in this lemma.

From the protocol, it can be seen that a consensus instance is started during the normal phase (Algorithm 2, line 6). Following this, there are two possibilities:

(1) *r decides a value for i before a timeout is triggered.* For this scenario to happen, it is necessary that at least $n-f$ processes participated in the consensus instance without deviating from the protocol. Therefore, $n-f$ replicas had to start instance i.

(2) *A timeout is triggered before r is able to decide a value for i.* This situation can happen either because the system is passing through a period of asynchrony, or because the current leader is faulty. Let us consider a consensus instance j such that j is the highest instance started by a correct replica, say r'. Let us now consider the following possibilities:

2-a) r started i and $i < j$. Remember that our algorithm executes a sequence of consensus instance, and no correct replica starts an instance without first deciding the previous one (Algorithm 2, lines 3-6). If $i < j$, j had to be started after i was decided in r'. But if i was decided, at least $n-f$ processes participated in this consensus instance. Therefore, $n-f$ replicas had to start instance i.

2-b) r started i and $i > j$. This situation is impossible, because if j is the highest instance started, and both r and r' are correct, i cannot be higher than j.

2-c) r started i and $i = j$. In this case, the synchronization phase might be initialized before all correct replicas start i. Because only a single correct replica might have started i, the log which goes from instance 0 to instance $i-1$ might not be present in the SYNC message (sent by Algorithm 3, lines 28-29), even if all replicas are correct (because the leader can only safely wait for $n-f$ correct STOPDATA messages). This means that an instance h such that $h \leq i$ will be selected by all correct replicas upon the reception of the SYNC message from the leader.

If the system is asynchronous, multiple synchronization phases might occur, where in each one a new leader will be elected. In each iteration, a faulty replica may be elected as leader; but from Lemma 2, we know that a faulty leader cannot prevent progress. Therefore, when the system finally becomes synchronous, eventually a correct leader will be elected, and h will eventually be started by $n-f$ replicas.

Finally, let us consider the case where $h < i$. In this case, a total of $n-f$ replicas may start h instead of i. But by the *Termination* property of our primitive, h will eventually decide, and all correct replicas will start the next instance. Because of this, eventually $n-f$ replicas will start i, even if more synchronization phases take place. ■

Using Lemmata 1-3 we can prove that MOD-SMART satisfies the SMR Liveness with the following theorem.

Theorem 2 *A valid operation requested by a client eventually completes.*

Proof: Let o be a valid operation which is sent by a client, and I the finite set of consensus instance where o is proposed. Due to Lemma 1, we know that o will eventually be received by all correct replicas, and at least one of them will propose o in at least one instance of I (the *fair* predicate ensures this). By Lemma 3, we also know that such instances will eventually start in $n-f$ replicas.

Furthermore, let us show that there must be a consensus instance $i \in I$ where o will be part of the batch that is decided in i. As already proven in Lemma 1, all correct replicas will eventually receive o. Second, we use the *fair* predicate to avoid starvation, which means that any operation that is yet to be ordered, will be proposed. Because of this, all correct replicas will eventually include o in a batch of operations for the same consensus instance i. Furthermore, the γ predicate used in the VP-Consensus ensures that (1) the operations in the batch sent by the consensus leader is not empty; (2) it is correctly signed; and (3) the sequence number of each operation is the next sequence number expected from the client that requested it.

Since there are enough replicas starting i (due to Lemma 3), the *Termination* property of consensus will hold, and the consensus instance will eventually decide a batch containing o in at least $n-f$ replicas. Because out of this set of replicas there must be $f+1$ correct ones, o will be correctly ordered and executed in such replicas. Finally, these same replicas will send a REPLY message to the client (line 17, Algorithm 2), notifying it that the operation o was ordered and executed. Therefore, a valid operation requested by a client eventually completes. ■