Bounded Delay in Byzantine Tolerant State Machine Replication (Full Paper)

Zarko Milosevic, Martin Biely, André Schiper
Ecole Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland
Email: firstname.lastname@epfl.ch

***Abstract*—The paper proposes a new state machine replication protocol for the partially synchronous system with Byzantine faults. The algorithm, called *BFT-Mencius*, guarantees that the latency of updates initiated by correct processes is eventually upper-bounded, even in the presence of Byzantine processes. BFT-Mencius is based on a new communication primitive, *Abortable Timely Announced Broadcast* (ATAB), and does not use signatures. We evaluated the performance of BFT-Mencius in the cluster settings, and showed that it provides bounded latency and good throughput, being comparable to the state-of-the-art algorithms such as PBFT and Spinning in fault-free configurations and outperforming them under performance attacks by Byzantine processes.**

I. Introduction

We become increasingly dependent on online services; therefore, their availability and correct behavior become increasingly important. In order to guarantee that services are available in spite of failures one basic strategy is to use replication: By replicating a service on multiple servers, clients are guaranteed that even if some replica fails, the service is still available. State machine replication (SMR) is a general approach for replicating services that can be modeled as a deterministic state machine [1], [2]. The key idea of this approach is to guarantee that all replicas start in the same state and then apply requests from clients in the same order, thereby guaranteeing that the replicas' states will not diverge.

Current deployments of SMR in industry handle benign failures only, with all major players employing some sort of replication in their infrastructure (e.g., Zookeeper [3], Chubby [4], Dynamo [5]). Indeed, as Barroso and Hölzle point out [6, Chapter 7] in a warehouse-scale data-center of 100'000 servers built with hardware with a mean time between failure of 30 years, one server can be expected to crash every day. Byzantine failures are until now considered to occur with only negligible probability, but given growth of data-centers it seems only a matter of time until a similar argument holds for Byzantine failures as well. Moreover, as Driscoll et al. [7] point out—albeit in the context of safety critical systems—the assumption that Byzantine failures are not relevant in practice might lead to actual Byzantine failures to be overlooked, and miss-categorized, for instance as software bugs.

Byzantine fault-tolerant replication algorithms allow computer systems to continue to provide a correct service even when some of their components behave in an arbitrary way, either due to faults or due to a malicious intruder. Although Byzantine failures have already been introduced in 1980 [8], Byzantine fault tolerant (BFT) replication protocols were considered too expensive to be practical [9]. This changed when Castro and Liskov introduced "Practical Byzantine faulty tolerance" (PBFT) [10]. They showed that in the fault-free settings BFT replication protocols can achieve performance that is close to that of non-replicated systems. The key observation that made PBFT practical was using MACs instead of signatures, which were the main performance bottleneck in previous systems [10]. Indeed, while our implementation of PBFT (cf. Section VII for details) achieves peak throughput of 52K requests per second with average client-latency of 4.5 ms, once we introduced RSA signatures to PBFT, peak throughput drops to 6K requests per second with 20ms latency.

After PBFT, several similar approaches continued to improve performance in the fault-free case (e.g., Zyzzyva [11]). However, like PBFT, most of these protocols are fixed sequencer protocols [12], i.e., a single server has a special role to propose order of requests. Amir et al. showed [13] that this class of protocols is vulnerable to *performance attacks*. The key observation is that a malicious sequencer can delay the ordering of requests, causing a considerable increase in latency and a great reduction in throughput. Performance failures were defined as Byzantine servers behaving as a "correct but very slow" server. More precisely, a Byzantine server exhibiting performance failures sends messages according to the protocol, but delayed—typically just in time to avoid triggering protocol timeouts that will get them demoted. This makes it very hard to detect a faulty server and apply some kind of reconfiguration mechanism, in order to reduce the impact on the protocol. Bounding the service response time (or having other performance guarantees) is not only of theoretical interests. For instance, in Amazon's Dynamo [5], there is a formal Service Level Agreement (SLA) where a client and a service agree on the client's expected request rate distribution and the expected service latency under those conditions.

Amir et al. [13] also proposed a new performance-oriented criterion, called *bounded-delay*, that requires that, given a system that is not overloaded and where servers have sufficient bandwidth to communicate, in a (long enough) period of synchrony the latency of updates initiated by correct servers is eventually upper-bounded, even in the presence of Byzantine servers. Thus ensuring bounded-delay could be considered as capturing what one would informally describe as tolerating

performance failures.

Unfortunately, this is not entirely true, because the definition of *bounded-delay* does not tell us how big the upper-bound should be. Consider for example protocols that continuously rotate the leader, such as Aardvark [14] or Spinning [15] (which do not consider bounded-delay as a criterion). It can be easily seen that such protocols are able to guarantee an upper-bound of $f \cdot T$, where f is the number of faulty servers and T is a protocol timeout whose expiration triggers reconfiguration mechanism such as view change. The intuition behind such claim is simple: in the worst case, from the time a correct server p has a request to propose (e.g., because it received it from a client), until it can propose the request, the server needs to wait until $n-1$ instances of other servers have terminated. Even if p broadcasts the request when its not its turn, the next f instances may be coordinated by faulty servers. As the timeout values are normally chosen rather conservatively[1], the guaranteed bound that is roughly $f \cdot T$ can be significantly higher than the latency in the failure-free case. Thus, providing such bound does not necessarily match the intuition one has for *tolerating performance attacks*. One would rather expect that algorithms that tolerate performance attacks achieve the same order of performance as in the failure-free case, where it is in the order of the real communication delay among servers.

Apart from defining new performance criteria, Amir et al. also proposed [13] a new BFT algorithm called *Prime*. Note that what Prime provides is in accordance with our arguments above, as its upper bound on latency is in the order of the real communication delay. Prime is derived from PBFT by adding a pre-agreement phase. That is, servers exchange requests using a reliable broadcast protocol before they are actually ordered by what is essentially PBFT. While increasing the message complexity and the number of communication steps on the critical path, having this pre-agreement phase is what allows servers to compute a threshold of acceptable performance, which is then used to judge if the sequencer is faulty. More precisely, executing reliable broadcast allows correct servers to come to a consistent view of what the current sequencer should do, for example, when the next instance should start and what requests should be proposed in a next instance. Without a pre-agreement phase, i.e., in PBFT and similar protocols, this knowledge is only available at the sequencer.[2] Therefore, it is very hard (if not impossible) for a correct server to determine a bound that defines an acceptable level of performance in such centralized protocols. In a sense, adding a pre-agreement phase makes Prime a less centralized protocol, as in the pre-agreement phase all servers have the same role. So although still based on the fixed-sequencer scheme, being more decentralized allows Prime to reduce the impact a faulty sequencer can have on the protocol, by demoting the sequencer if it does not provide the acceptable level of performance that is in the order of real communication delay among correct servers. When comparing Prime with PBFT we see that ensuring the stronger performance criteria comes at a price: reintroducing signatures makes it costly for cluster settings, and adding the pre-agreement phase strongly affects performance in the failure-free case.

In this paper, we propose a new BFT SMR protocol, BFT-Mencius, that ensures bounded-delay that is in the order of real communication delay among correct servers, but does not incur additional costs (as Prime). The key to do so is to go one step further: Instead of adding a decentralized pre-agreement phase to PBFT as in Prime, BFT-Mencius is a fully decentralized protocol where all servers propose in different instances of a sub-protocol called ATAB (explained below) concurrently. Because these instances are tightly coupled, servers can use their own progress in proposing requests to estimate the progress others should make.

In more detail, BFT-Mencius is inspired by Mencius [16], an efficient multi-leader SMR protocol that tolerates (only) crash faults, originally designed for WAN settings. Mencius uses an (infinite) sequence of instances of a subprotocol referred to as *simple consensus*, where only a pre-determined initial leader can propose any value; the others can only propose a special *no-op* value. A basic idea of Mencius is to partition the sequence of instances among the servers such that each server is the coordinator in an infinite number of instances. For instance, servers can take the role of coordinating instances in round-robin fashion. As servers take turns proposing values, Mencius itself is a moving sequencer atomic broadcast protocol. As such, the difficult part of Mencius is (i) preventing servers that do not have requests to propose from blocking the protocol and (ii) dealing with instances where the coordinator is a faulty server. While the latter is handled inside *simple consensus*, the former is handled by servers allowing to skip their turns by proposing a special *no-op* request. The key to Mencius' performance is that simple consensus allows servers to skip their turns without having to execute the full agreement protocol, thereby requiring to wait for messages from a majority of servers. In fact, Mencius even allows servers to skip implicitly by participating in higher numbered instances started by "faster" servers.

As Mao et al. [16] pointed out, such a mechanism does not work in the Byzantine tolerant systems, because not all decisions are communicated through a quorum[3]. Therefore, we designed BFT-Mencius using an abstraction that we call *Abortable Timely Announced Broadcast (ATAB)*. ATAB is a new broadcast primitive, that is similar to *Timely Announced Broadcast* [18] and *Terminating Reliable Broadcast* (TRB). In contrast to these two, it is specified such that it can be implemented in the partially synchronous system model. Like these two primitives (owing to the fact that it is a broadcast abstraction) each instance of ATAB has a dedicated sender server. BFT-Mencius let servers skip their turns by proposing *no-op* requests, and relies on ATAB to terminate instances with faulty dedicated sender within bounded time. Furthermore, its announcement property allows to tie the start times of different instances together thereby enabling correct servers to compute during the synchronous period a threshold for "acceptable speed" with which servers should start and terminate its ATAB instances during synchronous period. This knowledge is used in the blacklisting mechanism that ensures that faulty servers behave according to this bound; otherwise they get blacklisted and their subsequent ATAB instances ignored.

[1]For example, the timeout in Aardvark is set to 40ms, which is order of magnitude bigger than a duration of a single instance during synchronous period.

[2]Note that rotating the sequencer is of no help as a correct server has a complete picture only when it is its turn.

[3]Other option would be relying on trusted components as proposed in [17]

As we mentioned before, although BFT-Mencius provides strong performance guarantees, this does not penalize performance in the failure-free case, where its latency and throughput are comparable to state-of-the-art algorithms such as PBFT and Spinning [15]. We implemented the prototype of BFT-Mencius and evaluated its performance in cluster settings. We show that it provides bounded delay and good throughput, both in fault-free configurations and under performance attacks by Byzantine servers. For example, BFT-Mencius achieves a throughput of 45K requests per second with latency always below 5ms even under performance attacks.

Contribution: We propose a new BFT state machine replication protocol, BFT-Mencius, that guarantees a bounded-delay in the order of PBFT's latency in the failure-free case. Thus it *tolerates performance attacks*. Key to achieving this is by concurrently running multiple interlocked instances of new broadcast-abstraction called ATAB, which *does not use signatures*. This makes BFT-Mencius a *modular* protocol, which, we think, makes it simpler to understand, implement and test than other BFT SMR protocols, which are in general monolithic and often rather complex.

The remainder of the paper is as follows: We discuss related work in Section II. Section III defines a system model considered and gives the problem definitions. ATAB is introduced in Section IV, and the total order broadcast algorithm based on ATAB in Section V. The complete BFT-Mencius algorithm is given in Section VI. We evaluate performance of BFT-Mencius in Section VII and conclude in Section VIII.

II. Related work

Byzantine fault tolerant state-machine replication is a well established replication technique, which has been extensively discussed in the literature. Amir et al. [13] showed that malicious processes can significantly reduce throughput and increase the service latency in previous BFT protocols, by sending valid messages but as slow as possible without triggering timeouts. In the following we focus on work that considers such performance attacks. As we have explained before, Amir et al. propose a protocol called Prime, in which a pre-ordering phase is used to distribute client requests to all replicas. This is followed by a PBFT based global ordering phase, where among other information, the set of yet unproposed client requests is used to detect that the malicious leader is doing a performance attack.

Although Prime is the only protocol (before BFT-Mencius) that considers bounded-delay as performance criteria, it is not the only work that tolerates performance failures by Byzantine processes. In a similar spirit, Clement et al. [14] have advocated what they called robust BFT. That is, they propose to shift the focus from algorithms that optimize only best case performance, and to design algorithms that can offer predictable performance under the broadest possible set of circumstances—including when faults occur. To this end they propose a set of mechanisms to increase robustness of PBFT, in a system called Aardvark. Aardvark decreases the impact of slow leader by constantly monitoring the throughput sustained in the current view and by regularly performing view-changes. The idea of always rotating the sequencer was also used in BAR-B [19], but with different purpose. BAR-B is a cooperative backup system designed for the Byzantine-Altruistic-Rational model, where the leader is always changed so that every node has the equal opportunity to submit proposals to the system. Continuously changing the leader is also used in Spinning [15], with the goal to reduce the effect of performance attacks by Byzantine processes. In Spinning, the leader is changed after it defined the order of a single batch of requests, making leader change more efficient under performance failures than with Aardvark since there is no need for a complex view change protocol[4]. Spinning is also more efficient than BAR-B because it does not use signatures and ordering a request takes three communication steps compared to six with BAR-B. In order to prevent a faulty process from periodically impairing the protocol performance by requiring merge phases, Spinning introduces a blacklisting mechanism. Processes that are in the blacklist loose the privilege to propose, i.e., instances where they are primary are skipped. After a successful merge phase in view v, the primary of the view $v-1$ is added to the blacklist. As the merge phase is triggered once the timeout expires in view $v-1$, this blacklisting mechanism is still vulnerable to performance failures. BFT-Mencius also contains a blacklisting mechanism, but it uses a different detection mechanism which makes it very effective in detecting performance failures.

The common property of all three protocols, Aardvark, BAR-B and Spinning, is that all servers are allowed to propose requests only once it is their turn. As already explained in Section I, these algorithms ensure bounded-delay, but with an upper-bound in the order of $f \cdot T$. As we show in Section VI, BFT-Mencius does not have such limitation: its latency does not depend on number of faulty processes, and is in the order of the actual communication delay among correct processes.

Although most of the deterministic BFT protocols have a process with a special role (leader, coordinator, sequencer), there exists a deterministic consensus protocol that is fully decentralized (leader-free) [20]. However, the algorithm terminates in the order of $f \cdot T$ even in the failure-free case.

Contrary to deterministic BFT protocols, most randomized BFT protocols (e.g., [21], [22], [23]) are decentralized (there are no processes with special role) and work in the asynchronous systems; therefore faulty processes can not prevent correct processes from moving forward by delaying messages. Although, the intuition suggests that randomized protocols are less vulnerable to performance failures, we are not aware of a study that validates such claim. As these protocols are normally considered more costly in the failure-free case than deterministic protocols, we believe that making a detailed comparison study of these two groups of protocols is an interesting research topic.

III. Definitions

A. Model

We consider a system composed of n server processes $\Pi = \{1, \ldots, n\}$ and a finite number of clients processes connected by point-to-point channels. We assume integrity of

[4]The equivalent of the view change protocol in Spinning is called *merge* and it is executed when a sufficient number of correct processes suspect the current leader as faulty.

channels, that is if a process p received a message m from process q, then q sent message m to p before. We consider a partially synchronous system model: in all executions of the system, there is a bound Δ and an instant GST (Global Stabilization Time) such that all communication among correct processes after GST is reliable and Δ-timely, i.e., if a correct process p sends message m at time $t \geq GST$ to correct process q, then q will receive m before $t+\Delta$. The bound Δ and GST are system parameters whose values are not required to be known for the safety of our algorithms. However, the timing bounds derived for our algorithm, and thus the guaranteed latency require knowledge of Δ.

We do not make any assumption before GST. For example, messages among correct processes can be delayed, dropped or duplicated before GST. Spoofing/impersonation attacks are assumed to be impossible also before GST.

We assume that process steps (which might include sending and receiving messages) take zero time. Processes are equipped with clocks able to measure local timeouts.

At most f processes may fail in an arbitrary way, i.e., we consider Byzantine faults. We use the set $\mathcal{F}$ to denote the set of processes that are faulty in an execution, and we assume that the size of $\mathcal{F}$ is at most f. Processes in the set $\mathcal{C} = \Pi \setminus \mathcal{F}$ are called correct.

B. Replicated State Machines

Following Schneider [2], we note that the following is key for implementing a replicated state machine tolerant to f (Byzantine) faults:

Replica Coordination. All [non-faulty] replicas receive and process the same sequence of requests.

Moreover, as Schneider also notes this property can be decomposed into two parts, *Agreement* and *Order*: Agreement requires all (non-faulty) replicas to receive all requests, and Order requires that the order of received requests is the same at all replicas. In SMR protocols, Agreement and Order are often ensured by servers proposing client requests using a communication primitive known as atomic broadcast or total-order broadcast [24]. We also follow this approach.

C. Total-Order Broadcast

Total Order Broadcast is defined in terms of two primitives, to-broadcast and to-deliver. A process p that wishes to broadcast a message m from the set of messages $\mathcal{M}$ invokes to-broadcast(m). A message m is delivered by process q by executing to-deliver(m). We assume that the sender of a message can be determined from the message (denoted by $sender(m)$) and that all messages are unique. Both can be easily achieved by adding process identifiers and sequence-numbers to messages. Total order broadcast fulfills the following properties [25]:

- *TO-Validity*: If a correct process p invokes to-broadcast(m), then p eventually executes to-deliver(m).
- *TO-Agreement*: If a correct process p executes to-deliver(m), then every correct process q eventually executes to-deliver(m).
- *TO-Integrity*: For any message m, every correct process p executes to-deliver(m) at most once. Moreover, if $sender(m)$ is correct, then it previously invoked to-broadcast(m).
- *TO-Order*: If correct processes p and q execute to-deliver(m) and to-deliver(m'), then p delivers m before m' if and only if q delivers m before m'.

IV. Abortable Timely Announced Broadcast

In this section we introduce a new broadcasting primitive called *abortable timely announced broadcast* (ATAB), that we will use later to solve Total-Order broadcast. ATAB is defined in terms of four primitives: $atab\text{-}cast(m)$, $atab\text{-}abort$, $atab\text{-}announce$, $atab\text{-}deliver(m)$. The first two primitives are invoked by processes, while the latter two are triggered by ATAB. Moreover, there is a dedicated sending process s and no correct process $p \neq s$ executes $atab\text{-}cast$. If s is correct and it has a message m to broadcast it executes $atab\text{-}cast(m)$. Note that while we define ATAB as a one-shot problem with a fixed sender, we will use multiple instances of ATAB, with different sending processes for different ATAB instances. When a process delivers a message m it executes $atab\text{-}deliver(m)$. Like with *timely announced broadcast* (TAB) [18] a process is notified of an ongoing broadcast by $atab\text{-}announce$ before a message is actually delivered. In contrast to TAB we require all correct processes to eventually execute $atab\text{-}deliver$, much like *terminating reliable broadcast* (TRB). Like TRB we also allow delivery of a special value $\perp$. Unlike TRB which is designed for synchronous systems[5] and benign faults, ATAB is designed to tolerate Byzantine faults in partially synchronous systems. Therefore, we allow $\perp$ to be delivered only if some correct process aborts the broadcast by invoking $atab\text{-}abort$. Typically, this occurs when the protocol using ATAB suspects the sending process s. It is known that in the presence of Byzantine faults failure detection requires application knowledge [26]. The idea of $atab\text{-}abort$ is to make this knowledge explicit, and the idea of $atab\text{-}announce$ is to help with failure detection.

An algorithm solves ATAB with parameters d_1, and d_2, such that $d_1 \geq d_2$, if the following properties hold:

- *ATAB-Agreement*: If a correct process executes $atab\text{-}deliver(m)$, then every correct process eventually executes $atab\text{-}deliver(m)$.
- *ATAB-Integrity*: A correct process executes $atab\text{-}deliver(m)$ at most once. Furthermore, if s is a correct process and s executed $atab\text{-}cast(b)$ then $m \in \{\perp, b\}$.
- *ATAB-Termination*: If all correct processes execute either $atab\text{-}cast$, or $atab\text{-}announce$, or $atab\text{-}abort$ then every correct process eventually executes $atab\text{-}deliver$.
- *ATAB-Validity*: If a correct process s executes $atab\text{-}cast(m)$ at time $T \geq GST$ and no correct process calls $atab\text{-}abort$ before $T+d_1$, then all correct processes execute $atab\text{-}deliver(m)$ before $T + d_1$.
- *ATAB-Announcement*: If a correct process p executes $atab\text{-}deliver(m)$ at time T, then it executed

[5] TRB has been shown to require synchronous systems or an asynchronous system with a perfect failure detector.

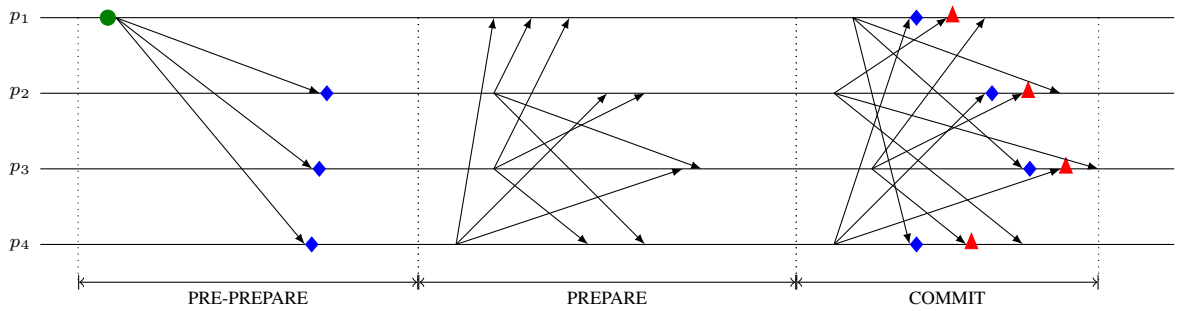


Fig. 1: Solving ATAB: message pattern of initial view. Process p_1 is the correct sender; its sending event ($atab\text{-}cast$) is indicated by the circle. Delivery of messages ($atab\text{-}deliver$) is indicated by triangles, while diamonds indicate $atab\text{-}announce$ events.

$atab\text{-}announce$ before T. Furthermore, if at time T, p executes $atab\text{-}deliver(m)$ or $atab\text{-}cast(m')$, then every correct process executes $atab\text{-}announce$ before $\max\{T, GST\} + d_2$.

A. Solving ATAB

Algorithms that solve ATAB are similar to algorithms that solve consensus. In fact, we can solve ATAB by using a coordinator based consensus algorithm that tolerates Byzantine faults [27] and adding the $atab\text{-}announce$ up-call and the $atab\text{-}abort$ down-call. The sender process execute $atab\text{-}cast(m)$ where it would execute $propose(m)$ in the consensus protocol, while $atab\text{-}deliver(m)$ is triggered at the point the consensus protocol invokes $decide(m)$. As consensus protocols normally proceeds in a sequence of views (sometime also called rounds, ballots or phases), the protocol would normally react on $atab\text{-}abort$ call by changing the view.

We now present an algorithm that solves ATAB in the partially synchronous system model with Byzantine faults. The algorithm requires $n > 3f$ processes to tolerate at most f Byzantine faults. The code of the algorithm is given as Algorithms 1 and 2. The *upon rules* of Algorithm 1 and Algorithm 2 are executed atomically. Since we allow messages-loss before GST, we use the p-send primitive which denotes periodic broadcast to all processes with Δ as an interval between two broadcast events. Processes can stop re-broadcasting once they have decided.

The algorithm is inspired by the PBFT SMR algorithm by Castro and Liskov [10], therefore we call it CL-ATAB. Contrary to PBFT which solves SMR (multiple instances problem), CL-ATAB solves the single instance problem ATAB.[6]

The algorithm proceeds in views, such that in every view there is a single process that is the coordinator (of the view). The assignment scheme of views to coordinators is known to all processes and is given as a function $\mathsf{coord}(v)$ returning the coordinator for view v. The sender s is the coordinator of the initial view ($view = 1$).

B. Normal case

The initial view is very similar to the "normal case" protocol of PBFT with addition of $atab\text{-}announce$ upcalls as shown in Figure 1. All subsequent views of CL-ATAB are very similar to the "view change protocol" of PBFT. We now explain the protocol of initial view (Algorithm 1), and the procedure for changing view is discussed for Section IV-C.

Once a process p wants to broadcast a message m using ATAB, it executes $atab\text{-}cast(m)$ (line 10 of Algorithm 1). Upon $atab\text{-}cast(m)$, a process sends $\langle \mathsf{INIT}, 1, m\rangle$ message to all processes (line 11 of Algorithm 1). Once a process receives $\langle \mathsf{INIT}, 1, m\rangle$ from the sender for the first time, and it is in the view 1, it sends $\langle \mathsf{ECHO}, 1, m\rangle$ message to all processes (line 14).

Once a process p receives $\langle \mathsf{ECHO}, 1, m\rangle$ message from $\lceil (n + f + 1)/2 \rceil$ processes, and it is in the view 1, it updates $vote_p$ and ts_p (lines 17-18) and sends $\langle \mathsf{COMMIT}, 1, m\rangle$ message to all processes (line 19).

A process receiving $\langle \mathsf{COMMIT}, 1, m\rangle$ message from $\lceil (n + f + 1)/2 \rceil$ processes, while being in the view 1, it executes $atab\text{-}deliver(m)$ (line 23). A process can also $atab\text{-}deliver(m)$ by receiving $f + 1$ $\langle \mathsf{DEC}, m\rangle$ messages (line 20).

The algorithm also needs to execute $atab\text{-}announce$ such that the *ATAB-Announcement* property of ATAB is fulfilled. The correct process always executes $atab\text{-}announce$ before it executes $atab\text{-}deliver$ (line 21). Furthermore, the algorithm need to ensure that once a correct process executes $atab\text{-}deliver$ all correct processes will eventually announce. This is ensured by the second part of the rule at line 24. Informally speaking, the idea is the following. Once a correct process executes $atab\text{-}deliver$, then at least single correct process received $\langle \mathsf{COMMIT}, v, m\rangle$ messages from $\lceil (n+f+1)/2 \rceil$ processes. Since messages are sent using $p - send$, and we assume that $n > 3f$, at least $f + 1$ correct processes sent $\langle \mathsf{COMMIT}, v, m\rangle$ message to all. Therefore, all correct processes will eventually also receive this messages and execute $atab\text{-}announce$ by the rule at line 24.

C. Changing view

In this section we explain the part of the protocol that processes execute when changing view. The protocol is given as Algorithm 2. There are several ways why a process can enter a new view, making execution of the view-change sub-protocol necessary:

[6]Thus, the relation between ATAB and PBFT can be considered similar to that of the Synod and Parliament protocols of [28].

Algorithm 1 CL-ATAB (part A)

```
1: Initialization:
2:    vote_p := noop
3:    ts_p := 0
4:    history_p := ∅
5:    view_p := 1
6:    state_p ∈ {init = 1, echoed = 2, changingView = 3}, initially init
7:    decision_p = null
8:    timeout1 := 3Δ
9:    timeout2 := 6Δ

10: upon atab-cast(m) do
11:    p-send ⟨INIT, 1, m⟩ to all

12: upon receiving ⟨INIT, view_p, m⟩ from coord(1) while state_p = init do
13:    history_p ← {(m, view_p)}
14:    p-send ⟨ECHO, view_p, m⟩ to all
15:    state_p ← echoed

16: upon receiving ⟨ECHO, view_p, m⟩ from ⌈(n + f + 1)/2⌉ processes while
    state_p ≤ echoed do
17:    vote_p ← m
18:    ts_p ← view_p
19:    p-send ⟨COMMIT, view_p, m⟩ to all

20: upon receiving (⟨COMMIT, view, m⟩ from ⌈(n + f + 1)/2⌉ processes or
    ⟨DEC, m⟩ from f + 1 processes) while decision_p ≠ null do
21:    Announce()
22:    decision_p ← m
23:    atab-deliver(m)

24: upon receiving ⟨INIT, 1, v⟩ from coord(1) or ⟨COMMIT, v, m⟩ from f + 1
    processes do
25:    Announce()

26: upon receiving any message ≠ ⟨DEC, −⟩ from q while decision_p ≠ null do
27:    send ⟨DEC, decision_p⟩ to q

28: Function Announce :
29:    atab-announce()
30:    if view_p = 1 then
31:       after timeout1 execute OnTimeout(1)

32: Function OnTimeout(v) :
33:    if decision_p = ⊥ then
34:       ProgressToView(v + 1)
```

Algorithm 2 CL-ATAB (part B) — Changing view

```
35: upon atab-abort() do
36:    ProgressToView(2)

37: upon receiving ⟨VC, view, −, −, −⟩ with view > view_p from f + 1
    processes do
38:    ProgressToView(view)

39: upon receiving ⟨VC, view_p, −, −, −⟩ from ⌈(n + f + 1)/2⌉ processes do
40:    after timeout2 execute OnTimeout(view_p)

41: upon receiving ⟨VC, view_p, v, ts, history⟩ from q do
42:    if viewChange_p[q] = ⊥ then
43:       viewChange[q] ← (view_p, v, ts, history)
44:       p-send ⟨VC-ACK, viewChange[q], q⟩
45:       CheckFLV

46: upon receiving ⟨VC-ACK, view_p, v, ts, history, q⟩ from r do
47:    if viewChangeAck_p[q][r] = ⊥ then
48:       viewChangeAck[q][r] ← (view, v, ts, history)
49:       CheckFLV

50: upon receiving ⟨VC-INIT, view_p, v, VC[]⟩ from coord(view_p) do
51:    if state_p = changingView ∧ newPrePrepare_p = ⊥ then
52:       newPrePrepare_p ← (view_p, v, VC[])
53:       CheckFLV

54: Function ProgressToView(v) :
55:    if view_p < v then
56:       view_p ← v
57:       viewChange[] := ⊥; VC[] := ⊥; viewChangeAck[][] := ⊥;
          newPrePrepare_p := ⊥
58:       state_p ← changingView
59:       p-send ⟨VC, view_p, v_p, ts_p, history_p⟩

60: Function CheckFLV :
61:    for i = 1 to n do
62:       if |{j : viewChangeAck[i][j] = viewChange[i]}| > 2f + 1 then
63:          VC[i] = viewChange[i]
64:    if p = coord(view_p) then
65:       select ← FLV(VC[])
66:       if select ≠ null then
67:          history_p ← history_p ∪ {(select, view_p)}
68:          state_p ← prePrepared
69:          p-send ⟨VC-INIT, view_p, select, VC[]⟩
70:          p-send ⟨ECHO, view_p, select⟩
71:    else if newPrepare_p ≠ ⊥ and isValid(newPrepare_p) then
72:       history_p ← history_p ∪ {(newPrepare_p, view_p)}
73:       p-send ⟨ECHO, view_p, newPrepare_p⟩
74:       state_p ← echoed

75: Function isValid(m) :
76:    for i = 1 to n do
77:       if m.VC[i] ≠ ⊥∧ ≠ viewChange_p[i] or
          |{q : viewChangeAck[i][q] = m.VC[i]}| < f + 1 then
78:          return false
79:    if FLV(m.VC) = m.v then
80:       return true
81:    else
82:       return false

83: Function FLV(V[]) :
84:    possibleVotes_p ← {(vote, ts, −) ∈ V :
       |{(vote', ts', −) ∈ V : vote = vote' ∨ ts > ts'}| ≥ ⌈(n + f + 1)/2⌉
85:    correctVotes_p ← {v : (v, ts, −) ∈ possibleVotes_p ∧
          |{(vote', ts', history') ∈ V : (v, ts) ∈ history'}| > f }
86:    if |correctVotes_p| > 0 then
87:       return min{v s.t. (v, −, −) ∈ correctVotes_p}
88:    else if |{(vote, ts, −) ∈ V : ts = 0}| ≥ ⌈(n + f + 1)/2⌉ then
89:       return ⊥
90:    else
91:       return null
```

- The process leaves the initial view in case *atab-abort* is executed (line 35).
- The process enters a new view after the timeout expires, and it has not yet learned what message should be delivered (lines 32-34). The timeout is triggered after process executes *Announce* function (line 28), or when process receives $\langle \mathsf{VC}, -, -, -, - \rangle$ message for the current view from $\lceil (n + f + 1)/2 \rceil$ processes (line 39).
- Finally, the process can move to the higher view if it receives $\langle \mathsf{VC}, -, -, -, - \rangle$ message from a correct process that is in a higher view (line 37).

As mentioned above changing the view is handled in a similar way as in PBFT. The processes exchange its state by sending $\langle \mathsf{VC}, view, vote, ts, history \rangle$ messages upon entering view. When a process p receives $\langle \mathsf{VC}, -, -, -, - \rangle$ message from a process q for the first time in the current view, it acknowledges its receipt by sending $\langle \mathsf{VC\text{-}ACK}, -, q \rangle$ message to all (lines 41-44). The coordinator of the new view will consider only $\langle \mathsf{VC}, -, -, -, - \rangle$ messages for which it is received acknowledgments by at least $2f + 1$ processes. This ensures that at least $f + 1$ correct processes received the same $\langle \mathsf{VC}, -, -, -, - \rangle$ message from a process q (that is potentially Byzantine process). The array of messages that satisfies this condition is passed to the FLV function (line 83) that is responsible for selecting a value that the new coordinator will propose such that *ATAB-Agreement* and *ATAB-Integrity* properties are not violated.

The FLV function ensures that in case some correct process $atab\text{-}deliver(m)$ in the previous views, it can select only m or $null$. The value $null$ is returned to indicate that not enough information was provided to the FLV function. As processes receive more information, i.e., more view-change messages, they will retry to obtain a previous decision value by calling FLV again. If there was a decision, FLV is guaranteed to eventually return the value decided. In case no correct process decided in the previous views, FLV returns $\perp$.

Once the FLV function returns the value that is not $null$, the coordinator of the new view sends $\langle \mathsf{VC\text{-}INIT}, -, -, - \rangle$ message to all processes (line 69). The other processes verify if $\langle \mathsf{VC\text{-}INIT}, -, -, - \rangle$ message sent by the new coordinator is valid using $isValid$ function (line 75). This mechanism is needed because the new coordinator can be faulty process. In case $isValid$ function returns `true`, the process sends $\langle \mathsf{ECHO}, -, - \rangle$ message (line 70) and the protocol continues as in view 1.

D. Proof of correctness

Lemma 1. *Let v_m be the highest view entered by some correct process up to time t. If $n > 3f$, then at least $f + 1$ correct processes are either in the view v or in the view $v - 1$ at time t.*

Proof: Let denote with p a first correct process that started view v_m. There are two cases to consider: (i) $v_m = 2$ or (ii) $v_m > 2$. In case (i), the lemma trivially follows since all correct processes are in at least view 1. In case (ii), since v_m is the highest view started by some correct process, the process p entered view v_m upon timeout expiration. Since $v_m > 2$, the timeout is set at line 40. By line 39, p received $\langle \mathsf{VC}, v_m - 1, -, -, - \rangle$ from $\lceil (n + f + 1)/2 \rceil$ processes. Since $n > 3f$, $\lceil (n+f+1)/2 \rceil > f$. Therefore at least $f+1$ correct processes sent $\langle \mathsf{VC}, v_m - 1, -, -, - \rangle$ message, i.e., at least $f+1$ correct processes are at least in the view $v_m - 1$ at time t. ■

Lemma 2. *If $n > 3f$, and $f + 1$ correct processes start view $v \geq 2$ at time $t > GST$ such that no correct process is in the higher view and $owner(v)$ is a correct process, then all correct processes will decide in view v the latest at time $t+6\Delta$.*

Proof: Let denote with C_f the set of $f + 1$ correct processes that start view v, and let assume that p is the last among correct processes from C_f that starts view v. Furthermore, let assume that it starts view v at time $t > GST$. Then the latest at time $t + \Delta$ all correct processes receive $\langle \mathsf{VC}, v, -, -, - \rangle$ message from processes C_f. Because of rule at line 37, all correct processes then start view v at time $t+\Delta$, and send $\langle \mathsf{VC}, v, -, -, - \rangle$ message. Note that process start timeout for view v once it receives $\langle \mathsf{VC}, v, -, -, - \rangle$ messages from $\lceil (n+f+1)/2 \rceil$ processes (line 39). Therefore, the earliest time when timeout for view v is started at some correct process is t.

At time $t + 2\Delta$, all correct processes are in view v and receive $\langle \mathsf{VC}, v, -, -, - \rangle$ message from all correct processes. Upon receipt of $\langle \mathsf{VC}, v, -, -, - \rangle$ message, $\langle \mathsf{VC\text{-}ACK}, -, - \rangle$ message is sent (line 44). Therefore, at time $t + 2\Delta$ all correct processes acknowledge receipt of $\langle \mathsf{VC}, v, -, -, - \rangle$ messages from correct processes by sending the corresponding $\langle \mathsf{VC\text{-}ACK}, -, - \rangle$ message. This mean that $owner(v)$ send $\langle \mathsf{VC\text{-}INIT}, v, m, - \rangle$ message before $t + 3\Delta$ (line 69) and all correct processes receive it and have the condition at line 71 of Algorithm 2 evaluates to `true` at latest at $t+4\Delta$ (at processes that are not $owner(v)$). Then they send $\langle \mathsf{ECHO}, v, m \rangle$ message (see line 73 of Algorithm 2) that is received before $t + 5\Delta$. Since $n > 3f$, $n - f \geq \lceil (n + f + 1)/2 \rceil$, so once a correct process receives $\langle \mathsf{ECHO}, v, m \rangle$ message from all correct processes, it sends $\langle \mathsf{COMMIT}, v, m \rangle$ message (line 19). All correct processes then decide before $t + 6\Delta$. Since the value of $timeout$ is 6Δ, no correct process will leave view v before $t + 6\Delta$ and therefore all correct processes will decide in view v. ■

Lemma 3. *If up to time T, all correct processes executed either $atab\text{-}announce$ or $atab\text{-}abort$, then all correct processes are in view $v \geq 2$ the latest at time $T + 3\Delta$.*

Proof: The correct process that executed $atab\text{-}abort$ move in view 2 the latest at time T (line 36). After correct process executes $atab\text{-}announce$, the timer is started (line 31). Once timeout expires, the process move in view 2 (if it is not already in view higher than 1). Therefore, all correct processes are in view 2 the latest at time $T + 3\Delta$. ■

Lemma 4. *If at least $f + 1$ correct processes are in view $v \geq 2$ at time $t > GST$, then all correct processes will start the same view v, such that no correct process is in the higher view, the latest at time $t + 16\Delta$.*

Proof: Let denote with p the correct process that is in the highest view v_m at time t. We have two cases to consider: (i) there are at most $f - 1$ other correct processes p in view v at time t or (ii) there are more than f other correct processes in view v at time t.

In case (i), at time $t + \Delta$ (due to retransmission that takes place every Δ time) all correct processes resend their $\langle \mathsf{VC}, -, -, -, - \rangle$ messages, so all correct processes receive those messages before $t + 2\Delta$. Because of rule at line 37, all correct processes enter at least view $v - 1$ at time $t + 2\Delta$ and send their $\langle \mathsf{VC}, -, -, -, - \rangle$ messages for view $v - 1$ at that point. They start timer for view $v - 1$ at time $t + 3\Delta$ (line 39). Therefore, they will enter view v the latest at time $t + 9\Delta$ since timeout value is 6Δ. Note however, that at time $t + \epsilon$ a correct process from view $v - 1$ can move to view v. Therefore, it can happen that some correct process receives $\langle \mathsf{VC}, v, -, -, - \rangle$ message from $\lceil (n + f + 1)/2 \rceil$ processes at time $t+\epsilon$ and start timer for view v. Therefore, at time $t+6\Delta$ he will leave view v and start view $v + 1$. Therefore, we need to calculate the point in time when other correct processes will start view $v + 1$. Since all correct processes start view v the latest at time $t + 9\Delta$, they will receive $\langle \mathsf{VC}, v, -, -, - \rangle$ message from $\lceil (n+f+1)/2 \rceil$ processes before time $t+10\Delta$. Therefore, they will start timer for view v the latest at $t+10\Delta$, and therefore start view $v + 1$ the latest at time $t + 16\Delta$.

In case (ii), at time $t + \Delta$ (due to retransmission that takes place every Δ time) $f+1$ correct processes resend their $\langle \mathsf{VC}, v, -, -, - \rangle$ messages, so all correct processes will receive them before time $t + 2\Delta$ and enter view v. They start timer for view v the latest at time $t + 3\Delta$. Since at any time after t there can be correct process that leaves view v and start view $v + 1$ we need to calculate what is the latest point when other

correct processes will enter view $v+1$. Since they start timer for view v the latest at time $t+3\Delta$, they will start view $v+1$ the latest at time $t+9\Delta$.

Therefore, all correct processes will start the same view v, such that no correct process is in the higher view, the latest at time $t+16\Delta$. ■

Lemma 5. *If at least $f+1$ correct processes are in view $v \geq 2$ at time $t > GST$, then all correct processes decide the latest at time $t+16\Delta+f\cdot 7\Delta+6\Delta$.*

Proof: By Lemma 4, all correct processes start the same view v, such that no process is in the higher view, the latest at time $t+16\Delta$. If $owner(v)$ is a correct process, then by Lemma 2 all correct processes decide the latest at time $t+16\Delta+6\Delta$. In case $owner(v)$ is a faulty process, then all correct processes will start timeout for view v the latest at time $t+17\Delta$. Therefore, they will start the view $v+1$ the latest at $t+17\Delta+6\Delta$. Since we apply rotating coordinator strategy for $owner(v)$ function, we will have correct process being owner of the view $v+f$. The correct processes start view $v+f$ the latest at time $t+16\Delta+f\cdot 7\Delta$. According to Lemma 2, all correct processes decide in the view $v+f$ the latest at time $t+16\Delta+f\cdot 7\Delta+6\Delta$. ■

Lemma 6. *For all $f \geq 0$, any two sets of size $\lceil (n+f+1)/2 \rceil$ have at least one correct process in common.*

Proof: We have $2\lceil (n+f+1)/2 \rceil \geq n+f+1$. This means that the intersection of two sets of size $\lceil (n+f+1)/2 \rceil$ contains at least $f+1$ processes, *i.e.*, at least one correct process. The result follows directly from this. ■

Lemma 7. *If $m \neq null$ is the only value that can returned by FLV function at correct processes in view v, then a correct process p can set $vote_p$ only to m in view v.*

Proof: If m is the only not-$null$ value that can be that can returned by FLV function at correct processes in view v, then if a correct process sends $\langle \mathsf{ECHO}, v, value \rangle$, $value = m$. Because there are at most f Byzantine processes, and $f < \lceil (n+t+1)/2 \rceil$, for all correct processes holds that if exists some value that satisfies the condition at line 16, then it must be m. So if a correct process p set $vote_p$ in view v, it set it to m. ■

Lemma 8. *If some correct process q $atab\text{-}deliver(m)$ in view v_0, then in all views $v > v_0$, FLV function at all correct processes can return either m or $null$.*

Proof: We prove the result by induction on v.

Base step $v = v_0 + 1$: Assume by contradiction that p is some correct process where FLV function returns $m' \neq m \wedge m' \neq null$ in view v_0+1. This implies that either (i) line 87 or (ii) line 89 was executed by p in phase v_0+1.

For (ii), the condition of line 88 has to be true. If the condition of line 88 is true, this implies that either (i) $|correctVotes_p| > 1$ or (ii) there are at least $\lceil (n+f+1)/2 \rceil$ messages with $ts = 0$ in the array V. Since q has decided in view v_0, by LemmaX at least one correct process received at least $\lceil (n+f+1)/2 \rceil$ messages $\langle \mathsf{COMMIT}, v_0, m \rangle$ at line 20. All correct processes c who sent a message $\langle \mathsf{COMMIT}, v_0, m \rangle$ have set $vote_c = v$ and $ts_c = v_0$ in view v_0. Let us denote this set of correct processes with Q_c. By Lemma 6 the intersection of two sets of size $\lceil (n+f+1)/2 \rceil$ contains at least one correct process. Therefore, in the $\lceil (n+f+1)/2 \rceil$ messages received there is at least one message sent by process from Q_c, *i.e.*, the second part of the condition at line 88 cannot be true. So the case (i) was executed by p.

For (i), the condition at line 86 have to be true, i.e., the $|correctVotes_p| > 0$, there exists a $(m', ts', history')$ such that m' is in $correctVotes_p$ and because of line 85 $(m', ts', history')$ in $possibleVotes_p$. We now show that if $(m', ts', history') \in possibleVotes_p$, then m' does not satisfy the condition at line 85 to be added to the $correctVotes_p$. This establishes the contradiction.

Since the parameter passed to the FLV function consists of $\langle \mathsf{VC}, -, -, - \rangle$ messages, by Lemma 6, the array V contains at least one message $\langle \mathsf{VC}, v_0+1, m, v_0, - \rangle$ sent by a process in Q_c. So the $(m', ts', history')$ can only be added to the set $possibleVotes_p$ if $ts' > v_0$.

In order to have m' in the set $correctVotes_p$ it is necessary to have at least $f+1$ messages $\langle \mathsf{VC}, v_0+1, -, -, history \rangle$ in V such that $\exists (m', ts') \in history$, i.e., that there is a correct process c_1 that sends such a message. A contradiction with the assumption that c_1 is a correct process.

Induction step from ϕ to $\phi+1$: Lemma 7 and the arguments similar to the base step can be used to prove the induction step. ■

Lemma 9. *If $n > 3f$, Algorithm 1 satisfies* ATAB-Agreement.

Proof: Let view v_0 be the first view in which some correct process p executes $atab\text{-}deliver(m)$. By Lemma 14, there exists a correct process c that received $\langle \mathsf{COMMIT}, v, m \rangle$ message from $\lceil (n+f+1)/2 \rceil$ processes in some view $v \leq v_0$. Since $n > 3f$, $\lceil (n+f+1)/2 \rceil > f$, so at least one correct process c_1 sent $\langle \mathsf{COMMIT}, v, m \rangle$ message in the view v. Therefore c_1 received at least $\lceil (n+f+1)/2 \rceil$ messages $\langle \mathsf{ECHO}, v, m \rangle$ (*). We prove now that if some correct process q executes $atab\text{-}deliver(m')$ in some view $v \geq v_0$, then $m = m'$. In case $v = v_0$, by Lemma 14, there exists a correct process c' that that received $\langle \mathsf{COMMIT}, v, m' \rangle$ message from $\lceil (n+f+1)/2 \rceil$ processes. Since $n > 3f$, $\lceil (n+f+1)/2 \rceil > f$, so at least one correct process sent $\langle \mathsf{COMMIT}, v, m' \rangle$ message in the view v. Therefore it received least $\lceil (n+f+1)/2 \rceil$ messages $\langle \mathsf{ECHO}, v, m' \rangle$. By Lemma 6, any two sets of messages of size $\lceil (n+f+1)/2 \rceil$, contains at least one correct process in intersection. Therefore, there exists a correct process c_1 that sends $\langle \mathsf{ECHO}, v, m \rangle$ and $\langle \mathsf{ECHO}, v, m' \rangle$ message. A contradiction with the assumption that c_1 is a correct process and the rule at line 12 and lines 73-74.

In case $v > v_0$, by Lemma 8 and Lemma 7, all correct processes can only set $vote$ to m in views bigger than v_0. Since $n > 3f$, $f < \lceil (n+f+1)/2 \rceil$, so the first part of the condition at line 20 can be `true` only for m. This is in contradiction with Lemma 14, so correct process q cannot $atab\text{-}deliver$ message different than m. ■

Lemma 10. *If s is a correct process and executes $atab\text{-}cast(m)$, then FLV function at all correct processes can return only b such that $b \in \{m, \perp, null\}$.*

Proof: Assume by contradiction that view v is the first view where FLV function at a correct process q returns m' such that $m' \notin \{m, \perp, null\}$. This implies that line 87 is executed by q. By assumption we have that for all messages $\langle \mathsf{VC}, v, -, -, history\rangle$ sent by correct processes, $history = \{(v,t) : v = m \vee v = \perp\}$ or $history = \emptyset$.

In order to have the condition at line 86 to be true, i.e., the $|correctVotes_q| > 0$, there exists m' in $correctVotes_q$.

In order to have m' in the set $correctVotes_p$ it is necessary to have at least $f+1$ messages $\langle \mathsf{VC}, v, -, -, history\rangle$ in V such that $\exists(m', ts') \in history$, i.e., that there is a correct process c_1 that sends such a message, i.e., FLV returns m' at c_1 in the view $v' < v$. A contradiction. ■

Lemma 11. *If $n > 3f$, Algorithm 1 satisfies* ATAB-Integrity.

Proof: A correct process p executes $atab\text{-}deliver$ only once because the rule in which $atab\text{-}deliver$ is executed is triggered only if $decision_p = null$ (line 20). After process p executes line $atab\text{-}deliver$ for the first time, $decision_p$ is set to some value $m \neq null$.

Now assume that s is a correct process and executes $atab\text{-}cast(m)$ and there is a correct process q that executes $atab\text{-}deliver(m')$ such that $m' \notin \{m, \perp\}$. By Lemma 14, there is a correct process c that received $\langle \mathsf{COMMIT}, v', m'\rangle$ messages from $\lceil (n+f+1)/2 \rceil$ in some view v' when $view_c = v'$. Since $n > 3f$, $\lceil (n+f+1)/2 \rceil > f$, there is at least one correct process c_1 that sends $\langle \mathsf{COMMIT}, v', m'\rangle$ message in view v'. Therefore, c_1 received $\lceil (n+f+1)/2 \rceil > f$ messages $\langle \mathsf{ECHO}, v', m'\rangle$. Since $n > 3f$, $\lceil (n+f+1)/2 \rceil > f$, i.e., there is at least single correct process c_2 that sent $\langle \mathsf{ECHO}, v', m'\rangle$ in the view v'. There are two cases to consider: (i) $v' = 1$ and (ii) $v' > 1$. In case (i), the process c_2 received $\langle \mathsf{INIT}, v', m'\rangle$ message from s. A contradiction with the assumption that s is a correct process that executed $atab\text{-}cast(m)$. In case (ii), the process c_2 received $\langle \mathsf{VC\text{-}INIT}, v', m', -\rangle$ from some process that is $owner(v')$.

Since process c_2 sent $\langle \mathsf{ECHO}, v', m'\rangle$ after receiving $\langle \mathsf{VC\text{-}INIT}, v', m', -\rangle$, this implies that the function $isValid$ at process c_2 returns `true` for message $\langle \mathsf{VC\text{-}INIT}, v', m', -\rangle$ (line 73). By line 79, the function FLV returns m' at process c_2 in view v'. A contradiction with Lemma 10. ■

Lemma 12. *If $n > 3f$, Algorithm 1 satisfies* ATAB-Validity *with $d_1 = 3\Delta$.*

Proof: If s executes $atab\text{-}cast(m)$ at time T, it sends $\langle \mathsf{INIT}, 1, m\rangle$ message to all (line 11), and all correct processes receive it before time $T + \Delta$. Since no correct process calls $atab\text{-}abort$ before $T + 3\Delta$, $state$ variable at all correct processes is $init$ and $view = 1$, so the condition at line 12 evaluates to `true`, and all correct processes send $\langle \mathsf{ECHO}, 1, m\rangle$ to all (line 14 and set $state$ to $echoed$ (line 15). Before time $T + 2\Delta$, all correct processes receive $\langle \mathsf{ECHO}, 1, m\rangle$ message from all correct processes. Since $n > 3f$ and no correct process $atab\text{-}abort$ before $T+3\Delta$, all correct processes receive $\langle \mathsf{COMMIT}, 1, m\rangle$ message from $\lceil (n+f+1)/2 \rceil$ processes, so the condition at line 16 evaluates to `true`, and every correct process sends $\langle \mathsf{COMMIT}, 1, m\rangle$ message to all (line 19) the latest at time $T + 2\Delta$. If a correct process has not decided up to time $T+3\Delta$, it will decide upon receiving $\langle \mathsf{COMMIT}, 1, m\rangle$ message from all correct processes since the condition at line 20 evaluates to `true`. Therefore, if no correct process $atab\text{-}abort$ before time $T + 3\Delta$, all correct processes will $atab\text{-}deliver(m)$ the latest at time $T + 3\Delta$. ■

Lemma 13. *Algorithm 1 satisfies* ATAB-Termination.

Proof: If all correct processes execute $atab\text{-}announce$ or $atab\text{-}abort$ up to time T, by Lemma 3 all correct processes enter view 2 the latest at time $T+3\Delta$. By Lemma 5 all correct processes will decide the latest at time $\max\{T+3\Delta, GST\} + 16\Delta + f \cdot 7\Delta + 6\Delta$. ■

Lemma 14. *If a correct process p executes $atab\text{-}deliver(m)$ at time T, then at least one correct process q received $\langle \mathsf{COMMIT}, v, m\rangle$ message from $\lceil (n+f+1)/2 \rceil$ processes in view v at time $t \leq T$.*

Proof: Assume by contradiction that a correct process p executes $atab\text{-}deliver(m)$ at time T, and that no correct process received $\langle \mathsf{COMMIT}, v, m\rangle$ message from $\lceil (n+f+1)/2 \rceil$ processes at time $t \leq T$. Furthermore, assume that a process p is a first correct process that executed $atab\text{-}deliver(m)$. This implies that a correct process that executes $atab\text{-}deliver(m)$ execute it after time T.

Since a correct process p executes $atab\text{-}deliver(m)$ at time T, this mean that the condition at line 20 evaluates to `true`. There are two cases to consider: (i) p received $\langle \mathsf{COMMIT}, view_p, m\rangle$ message from $\lceil (n+f+1)/2 \rceil$ processes, or (ii) p received $\langle \mathsf{DEC}, m\rangle$ message from $f+1$ processes.

In case (ii), p received $\langle \mathsf{DEC}, m\rangle$ message from $f+1$ processes. This mean that there is at least one correct process c that sent $\langle \mathsf{DEC}, m\rangle$ message before time T. A contradiction with the assumption that p is the first correct process that executed $atab\text{-}deliver(m)$. Therefore by (i), p received $\langle \mathsf{COMMIT}, view_p, m\rangle$ message from $\lceil (n+f+1)/2 \rceil$ processes. A contradiction. ■

Lemma 15. *If $n > 3f$, Algorithm 1 satisfies* ATAB-Announcement *with $d_2 = 2\Delta$.*

Proof: The first part of *ATAB-Announcement* property is trivially ensured by lines 21 and 23. We now prove the second part, i.e., (i) if a correct process p executes $atab\text{-}deliver(m)$ at time T or (ii) a correct process s executes $atab\text{-}cast(m')$ at time T, then every correct process executes $atab\text{-}announce$ before $\max\{T, GST\} + d_2$, where $d_2 = 2\Delta$.

In case (i), if a correct process p executes $atab\text{-}deliver(m)$ at time T, then by Lemma 14, there is at least one correct process c that received $\langle \mathsf{COMMIT}, view_c, m\rangle$ message from $\lceil (n+f+1)/2 \rceil$ processes at time $t \leq T$. Since $n > 3f$, $\lceil (n+f+1)/2 \rceil - f > f$, therefore at least $f+1$ correct processes sent $\langle \mathsf{COMMIT}, view_c, m\rangle$ message. Since messages are sent using $p{-}send$ primitive (and therefore resent every Δ time units), all correct processes will receive $\langle \mathsf{COMMIT}, view_c, m\rangle$ message from at least $f+1$ process the latest at time $\max\{T, GST\} + 2\Delta$, and $atab\text{-}announce$ because of the rule at line 24).

In case (ii), a correct process s executes $atab\text{-}cast(m')$ at time T. By line 11, s sends $\langle \mathsf{INIT}, 1, m'\rangle$ message to all at time T. Since the message is sent using $p - send$ primitive,

it is retransmitted every Δ time units, so all correct processes receive $\langle \mathsf{INIT}, 1, m' \rangle$ the latest at time $\max\{T, GST\} + 2\Delta$. By line 24 all correct processes execute $atab\text{-}announce$ the latest at time $\max\{T, GST\} + 2\Delta$. ■

Theorem 1. *If $n > 3f$, then the CL-ATAB algorithm solves ATAB in the partially synchronous system model with known Δ, $d_1 = 3\Delta$ and $d_2 = 2\Delta$.*

Proof: Follows from Lemma 9, Lemma 11, Lemma 13, Lemma 17 and Lemma 15. ■

V. Solving Total-Order Broadcast with ATAB

In this section, we present the central part of our BFT SMR protocol, which is a protocol that solves total-order broadcast, see Algorithm 3. It is inspired by Mencius [16], an efficient multi-leader SMR. The algorithm relies on the ATAB primitive for sending messages and refers to the values d_1 and d_2.

A. Basic idea

Algorithm 3 runs an infinite sequence of ATAB instances. We add a number i to ATAB calls to refer to ATAB instance i. These instances are evenly partitioned among the servers. Function $owner(i)$, known to all processes, returns the sender for instance i.

For every process p, $index_p$ is the next ATAB instance in which p is the sender. In order to to-broadcast a message m a process p executes $atab\text{-}cast(index_p, m)$ in the ATAB instance $index_p$ (line 9) and then updates $index_p$ (line 10).

Once a process p learns that ATAB instance i terminated (execution of $atab\text{-}deliver(i, m)$, line 11), it executes the following steps:

- If p was the sender in instance i, i.e., $owner(i) = p$, p has sent m and p learns that $m' \neq m$ is delivered (necessarily $m' = \perp$), then p to-broadcasts m again (line 17).
- Process p executes the $CheckCommit$ procedure (lines 24–30), which uses the $expected_p$ variable to keep track of the lowest yet undecided ATAB instance. Inside $CheckCommit$, p increases $expected_p$ as far as possible, executing to-deliver for all messages that are not $noop$.

Note that according to the $CheckCommit$ procedure, p executes to-deliver(m) in the order of ATAB instances. Therefore the message delivered in instance i cannot be adelivered before all instances $j < i$ have terminated. Since processes might to-broadcast at different rates, we need a mechanism to allow processes to fill the gaps so that the message delivered by ATAB instance i is not delayed because another process has nothing to broadcast in instance $j < i$. This is discussed in the two next paragraphs.

B. Process p skipping its own instances

In order to fill these gaps, a correct process will *skip* its instance j by broadcasting a special message $noop$. A process could execute $atab\text{-}cast(j, noop)$ when it sees that there is a decision in some instance $i > j$. However, skipping instance only once a higher number instance i terminates is unnecessarily late: as processes know that instance i is in progress already before decision. More precisely, in case a correct process p sees that some other process, say q, broadcasted in instance $i > index_p$ (by executing $atab\text{-}announce$ in instance i, line 20), then p skips all instances j with $index_p \leq j < i$ where p is the sender (lines 21-23). Here, the other correct processes handle instance j like any other instance owned by p. This is not the case in the next paragraph.

C. Process p skipping instances of other processes

The skipping mechanism of the previous paragraph is able to fill only those gaps caused by correct processes not broadcasting. Indeed, we cannot require a (Byzantine) faulty process to skip its instances. Therefore, a different mechanism is needed for the instances owned by a faulty process. Put differently, correct processes need a means to ensure that instances owned by faulty processes will terminate. ATAB provides the $atab\text{-}abort$ primitive to this end. However, executing $atab\text{-}abort$ too early might lead to deliver $\perp$ in ATAB instances owned by correct processes. This can be avoided during a synchronous period, i.e., after GST, with a timeout of $d_2 + d_1$, see lines 6 and 19. Consider some process p that terminates instance i at time T. By *ATAB-Announcement*, all correct processes execute $atab\text{-}announce$ for instance i at latest at $T + d_2$. By line 22, these processes $atab\text{-}cast$ in all instances $j < i$ for which they are sender (and which they did not $atab\text{-}cast$ before). By *ATAB-Validity*, these instances will all terminate within d_1, that is before $T' = T + d_2 + d_1$. Since process p does not execute $atab\text{-}abort_p(j)$ for the yet undecided instances j before T', instances owned by correct processes are indeed not aborted too early (see lines 19 and 31–33).

D. Correctness proof

Lemma 16. *Assume a correct process s calls $atab\text{-}cast(i, m)$ (in line 9) at time σ. If $\sigma > GST + d_2$ then no correct process aborts before deciding.*

Proof: Assume by contradiction that some correct processes aborted before deciding and let q be the first to abort at time t_q (line 33) before it decided. Since the process q aborted in instance i, this means that q decided in some instance $j > i$ at time $t_q - timeout$. By *ATAB-Announcement*, all correct processes (including s) announced in instance $j > i$ the latest at time $\max\{t_q - timeout, GST\} + d_2$. When s executes $atab\text{-}announce(j)$ (lines 20ff.) it updates its $index$ until it reaches some $k > j$ (lines 21 and 23). If $t_q - timeout < GST$, then s announced at $GST + d_2$, which means that at time $\sigma > GST + d_2$ it $index_s$ is at least k, such that $i \geq k > j > i$. A contradiction. Otherwise, we have two cases, either s announced before executing $atab\text{-}cast(i, m)$, then $i \geq k > j > i$ as above and we have reached a contradiction again; or s announced after executing $atab\text{-}cast(i, m)$, then as s announced before $t_q - timeout + d_2 = t_q - d_1$, it also called $atab\text{-}cast(i, m)$ before this point in time, i.e., $\sigma < t_q - d_1$ Since q is the first to abort no process aborts before t_q, that is no process aborts before $\sigma + d_1$. Therefore by *ATAB-Validity* all correct processes decide before t_q. Also a contradiction. ■

Algorithm 3 Total Order Broadcast with ATAB

```
1: Initialization:
2:   proposed_p[] := ⊥                    /* initially, for all i, proposed_p[i] = ⊥ */
3:   decided_p[] := ⊥                     /* initially, for all i, decided_p[i] = ⊥ */
4:   expected_p := 0                      /* lowest undecided ATAB instance */
5:   index_p := min {i : owner(i) = p}    /* next instance owned by p */
6:   timeout_p := d_1 + d_2

7: upon to-broadcast(m) do
8:   proposed_p[index_p] = m
9:   atab-cast(index_p, m)
10:   index_p ← min {i : owner(i) = p ∧ i > index_p}

11: upon atab-deliver(i, m) do
12:   if m ≠ ⊥ ∧ owner(i) = sender(m) then
13:     decided_p[i] ← m
14:   else
15:     decided_p[i] ← noop
16:   if p = owner(i) ∧ proposed_p[i] ∉ {m, noop} then
17:     to-broadcast(proposed_p[i])
18:   CheckCommit
19:   after timeout execute OnTimeout(i)

20: upon atab-announce(i) do
21:   while index_p ≤ i do
22:     atab-cast(index_p, noop)
23:     index_p ← min {i : owner(i, 1) = p ∧ i > index_p}

24: Function CheckCommit :
25:   while decided_p[expected_p] ≠ ⊥ do
26:     m ← decided_p[expected_p]
27:     o ← owner(expected_p)
28:     if m ∉ {noop} ∪ {decided_p[i] : i < expected_p} then
29:       adeliver(m)
30:     expected_p ← expected_p + 1

31: Function OnTimeout(i) :
32:   for each k ∈ {j ∈ [expected_p, i] : decided_p[j] = ⊥ ∧ owner(j) ≠ p}
      do
33:     atab-abort_p(k)
```

Lemma 17. *If a correct process s calls* to-broadcast(m)*, then all correct processes eventually* to-deliver *m from s.*

Proof: When s calls to-broadcast(m), it executes $atab\text{-}cast(i, m)$ for some i. If it does so after $GST + d_2$, then from Lemma 16 it follows that instance i will not be aborted, and therefore *ATAB-Validity*, ensures that all processes will execute $atab\text{-}deliver(i, m)$ and thus set $decided[i] \leftarrow m$. Eventually, all instances $j < i$ will terminate as well, so processes will execute $CheckCommit$ with $decided[j] \neq \perp$ for all $j \leq i$, and will thus to-deliver m.

Otherwise, if s calls $atab\text{-}cast(i, m)$ before $GST + d_2$, instance i might get aborted and $\perp$ may be delivered. If this is the case, processes will set $decided[i] \leftarrow noop$. Upon doing so, s will by line 17,re-call to-broadcast(m) implying some $i' > i$ for which s executes $atab\text{-}cast(i', m)$. What, therefore, remains to be shown is that there is some k such that s executes $atab\text{-}cast(k, m)$ and $atab\text{-}deliver(k, m)$. (*ATAB-Agreement* implies that all others will also execute $atab\text{-}deliver(k, m)$.)

We show this by contradiction and assume that there is no k such that s executes $atab\text{-}deliver(k, m)$. Then line 17 entails that there is an infinite sequence of instances that all decide $\perp$ although m was proposed. Clearly, one of them must start at some time $\sigma > GST + d_2$, which by Lemma 16 implies that s will $atab\text{-}deliver$ within d_1. A contradiction. ■

Proposition 1. *Given a solution to ATAB, Algorithm 3 solves Total-Order Broadcast.*

Proof: TO-Validity follows from Lemma17.

From the *ATAB-Agreement* and *ATAB-Integrity* properties it follows that if p executes 13 for some i and $v \in \mathcal{M}$ then so will any correct q, and both will do so exactly once. Since these non-$\perp$ values of $decided_p$ determine for which messages m and processes s that p executes to-deliver$_p(m)$ for, *TO-Agreement*follows.

We now turn to the first part of *TO-Integrity*: The while-loop of $CheckCommit$ ensures that p executes to-deliver(m) for every value of expected at most once. The condition of line 28 then entails that m was not adelivered for a previous value of $expected$. Thus every to-deliver$_p(m)$ is executed exactly once. Since we have already shown *TO-Agreement*, it suffices to show for the second part of *TO-Integrity*, that the correct process $s = sender(m)$ will only to-deliver$_s(m)$ if it previously executed to-broadcast$_s(m)$. Since s executes to-deliver(m) with $m = decided_s[expected]$ it follows that $m \neq noop$. Further, since $decided_s[i]$ with $i = expected$ can have been set to a non-$noop$ message only in line 13 it follows that s executed $atab\text{-}deliver(i, m)$ such that $owner(i) = s$ before. Since s is correct and $m \neq noop$, it follows from *ATAB-Integrity*of that s executed $atab\text{-}cast(i, m)$, which it must have done in line 9, that is in a call of to-broadcast(m).

Total Order follows from the fact that if p executes to-deliver$_p(m, s)$ before to-deliver$_p(m', s')$ then there are j and j' such that $j < j'$, $decided_p[j] = m$, $decided[j'] = m'$, $s = owner(j)$ and $s' = owner(j')$. Now when q executes to-deliver$_q(m', s')$ it does so when $expected = j'$, since $j < j'$ it follows that it executed line 29 with $expected = j$ before. From the argument on *TO-Agreement* above, it is clear that at this point $decided_p[j] = m$, thus q executed to-deliver$_q(m, s)$ before. Now *TO-Order* follows from *TO-Integrity*. ■

VI. BFT-Mencius

In this section we describe the complete BFT-Mencius protocol for SMR that is based on the Total Order Broadcast Algorithm 3. In BFT-Mencius clients send requests to servers (details below), which use total-order broadcast to order requests. After a request is executed by some server, the server sends the reply to the corresponding client. A client accepts a response only once it received $f + 1$ identical responses from $f + 1$ servers.

In Byzantine fault tolerant state machine replication only requests proposed by clients should be executed. This requirement is trivially ensured by using cryptographic signatures to sign client requests. Request authentication can also be achieved using MACs [10], [29]. In this paper we assume that request authentication is done using MACs as in [10] and in other protocols that will be compared experimentally with BFT-Mentius.[7]

In BFT-Mencius every server is able to propose requests, thus different variants of load balancing of client requests can be used. However, finding the optimal load balancing scheme is outside the scope of this paper. For simplicity, we assume here a static assignment of client ids to server. Servers propose requests they receive from client assigned by this scheme. In

[7]BFT-Mencius can also be used with other variants of request authentication.

the presence of faulty servers some clients will be assigned to faulty servers. In order to ensure that requests from such clients will be ordered and executed, an additional mechanism is necessary: clients send each request to all servers,[8] and servers keep track of requests not assigned to them. They propose any requests that are not executed within some time. For instance, if a server finds a request req that is not executed after the server has terminated k of its own ATAB instances, the server proposes req.[9] In our experiments we have set $k = 3$. Thus faulty servers cannot starve clients by ignoring their requests.

Moreover, since we use a (numbered) sequence of instances we have to prevent a faulty server from exhausting the space of sequence numbers by starting ATAB instance with a very large instance number. To this end, every server can have at most one outstanding non-decided instance. Put differently, a server will not react on messages received for some instance j owned by a process q if it has not terminated in all instance $k < j$ owned by q.

BFT-Mencius uses batching of client requests, which is well known to be essential for good performance. In the rest of this section we explain additional mechanisms that are employed in BFT-Mencius in order to minimize the negative impact a faulty process can have on the performance of the protocol.

A. Dealing with slow servers

Algorithm 4 Blacklisting mechanism

```
1:  Initialization:
2:    blacklist_p := ∅
3:    ∀q ∈ Π : suspects[q]_p := ∅
4:    /* see also Algorithm 3 */

5:  upon suspect(q) do
6:    if q ∉ blacklist_p then
7:      abcast(⟨SUSPECT, q⟩)

8:  Function CheckCommit :
9:    while decided_p[expected_p] ≠ ⊥ or owner(expected_p) ∈ blacklist_p
      do
10:     m ← decided_p[expected_p]
11:     o ← owner(expected_p)
12:     if m = ⟨SUSPECT, q⟩ then
13:       UpdateBlackList(q, o)
14:     else if m ∉ {noop} ∪ {decided[i] : i < expected_p} then
15:       adeliver(m, owner(expected_p))
16:     expected_p ← expected_p + 1

17: Function UpdateBlackList(q, o) :
18:   if q ∉ blacklist_p then
19:     add o to suspects_p[q]
20:     if |blacklist_p[q]| ≥ f + 1 then
21:       add q to blacklist_p
22:       suspects_p[q] ← ∅
```

The mechanism just presented addresses the problem of requests sent by clients assigned to faulty servers. Here we address the problem of faulty servers slowing down the ordering of requests assigned to *other* servers. In the total order broadcast algorithm (Algorithm 3) a faulty server can do a performance attack by delaying its own instances. This is due to the fact that the message of ATAB instance i is not delivered until all instances $j < i$ have terminated. We address this issue by introducing a *blacklisting mechanism* used as follows (we discuss when to suspect servers in Section VI-B): ATAB instance i waits for the termination of only those instances $j < i$ that are not owned by servers on the blacklist. That is, instances whose owners are in the blacklist are skipped, i.e., the effect is equivalent to the case where $noop$ was decided. Therefore, it is important that the blacklist is kept consistently on all servers. Because the blacklist may be seen as a state machine, we can use our SMR protocol to ensure consistency (similar to how reconfiguration in benign systems can be done [28]).

The blacklist is implemented as a circular buffer of size f, thus adding the $(f + 1)$-st server will rehabilitate the server that is longest in the list. A server p adds a server q to its (server of the) blacklist once $f + 1$ servers suspected q.

In order for server p to consistently inform other servers that it suspects q to be faulty, p to-broadcasts the special request $\langle \mathsf{SUSPECT}, q \rangle$. This value will then be broadcasted using p's next ATAB instance as a normal request, and thus stored in all correct servers $decided$ list at the same position. In order to avoid that p uses all its ATAB instances for SUSPECT messages, one can either limit the rate of such messages or piggy-back SUSPECT messages on the normal messages that are to-broadcast.

We show pseudocode of the blacklisting mechanism in Algorithm 4. It includes: (i) the $suspect(q)$ function, used by server p to locally trigger blacklisting mechanism once it suspects some server q, (ii) a modified version of the $CheckCommit$ function of Algorithm 3, and (iii) function $UpdateBlackList(q, o)$ called by $CheckCommit$. The function $UpdateBlackList(q, o)$ (line 17) maintains the blacklist at server p: it is executed whenever p learns that server o suspects server q. The function first checks if q is not already in the blacklist. If this is true, o is added to $suspects[q]$. If at this point there are $f+1$ different servers that suspect q, server q is added to the $blacklist$ and $suspects[q]$ is cleared. Note that, since $UpdateBlacklist(q, o)$ is called by $CheckCommit$ when $\langle \mathsf{SUSPECT}, q \rangle$ is decided, the correct servers always have a consistent view of the blacklist. That is, for each value of $expected$, the blacklist is the same at all servers.

Since ATAB instances of server p are ignored while p is on the blacklist, messages to-broadcast by p cannot be delivered. Thus once p is added to the blacklist, p's clients are reassigned to servers not in the blacklist. At this point all requests from reassigned clients not executed will be proposed by the newly assigned server.[10]

B. When to suspect a server

The blacklisting mechanism is very general, i.e., it can be used to report any suspicious behaviour. As in this paper we are interested in ensuring bounded latency, we use it to report when a server is slow. For this we rely on the properties of ATAB that hold during a synchronous period, i.e., after GST. More precisely, once a correct server executes

[8] Since messages might be lost before GST, we actually assume clients periodically retransmit.

[9] In fact it is sufficient if requests assigned to a certain server are tracked only by f other servers (instead of all).

[10] This mechanism is different from the delayed re-proposing of requests mentioned at the beginning of Section VI: the reassignment mentioned here causes instantaneous re-proposing when a server is added to the blacklist.

$atab\text{-}cast(i, m)$ at time t in line 9 of Algorithm 3, by the *ATAB-Announcement* property we know that all correct servers execute $atab\text{-}announce(i)$ the latest at time $t + d_2$. Therefore, all correct servers start their instances j, with $j < i$, the latest at time $t+d_2$. By *ATAB-Validity* all such instances terminate the latest at time $t + d_2 + d_1$. Therefore, if at time $t' > t + d_2 + d_1$, some instance $j < i$ has not terminated, the owner of instance j is suspected and $suspect(owner(j))$ is executed. The owner of an instance is also suspected if a server executes $atab\text{-}abort$ (line 33 of Algorithm 3). This solution is able to guarantee bounded delay that does not depend on the number of faulty servers.

However, d_1 and d_2 are worst case bounds. We would like to replace the d_1+d_2 timeout by a smaller value, assuming that after GST the duration of ATAB instances running concurrently do not differ substantially between owning servers. Let d_{ATAB} be the duration of ATAB instances measured by server p. This leads us to estimate d_1 as d_{ATAB}, and since $d_2 < d_1$, we conservatively also estimate d_2 as d_{ATAB}. This leads us to use $2 \cdot K_{lat} \cdot d_{ATAB}$ as a timeout to suspect servers[11], where K_{lat} is a system parameter that accounts for variability in latencies on the network. As we will show in Section VII, in the cluster settings, $K_{lat} = 1$ is sufficient to reliably detect a faulty server doing a performance attack. In less homogeneous settings, it could be preferable to use a more complex way to determine this timeout, for example by taking the median of recent durations of ATAB instances owned by different servers, by adding additional weight factors, or just by using a bigger value for K_{lat}.

In addition to suspecting servers based on their latency, it is possible to use additional strategies for detecting faulty servers. For example, one could measure the number of requests executed in the last n instances owned by each server, and suspect servers whose number is less than 50 percent of the average number of requests executed by servers. Aardvark [14] uses a similar idea to suspect the current leader. While this and other techniques of suspecting the current leader based on performance or fairness criteria easily carry over from Aardvark and other protocols, we did not consider them for BFT-Mencius, because our main goal is the ability to ensure bounded delay. In any case, if the suspect mechanism used leads to frequent changes of the blacklist, this can only lead to requests being proposed by multiple servers thereby causing performance degradation.

VII. Evaluation

We have implemented a prototype of BFT-Mencius in Scala using the Distal framework [30]. Distal is a new framework that allows writing code in a domain specific language (DSL) that is close to the protocol description. Therefore, it leads to the implementations that really reflects the protocol specification on paper. Finally, it leads to the efficient protocol implementation. As Distal currently does not provide authenticated channels, we extended Distal's messaging layer to support message authentication based on SHA-1 HMACs. In order to compare BFT-Mencius with the state-of-the-art protocols based on the same code base, we have implemented the normal case of PBFT and Spinning using Distal. We do not experimentally compare BFT-Mencius with Prime because—as mentioned in Section I—adding signatures to PBFT has already lead to a significant drop in performance even without adding the pre-agreement phase.

We make the following two points with the experimental evaluation of BFT-Mencius. First, it shows that the modular BFT-Mencius protocol has performance comparable to PBFT and Spinning in the failure-free case. Second, it shows that BFT-Mencius is able to sustain good performance (similar to the failure-free case) even under performance attack.

A. Experimental setup and methodology

The experiments were run in the *Suno* cluster of the Grid5000 testbed. This cluster consists of nodes with dual 2.26GHz Intel Xeon E5520 processors, 32GB of memory, and 1Gb/s Ethernet connections. Nodes were running Linux, kernel version 2.6.32-5, and Oracles Java 64-Bit Server VM version 1.6.0_26.

The workload was generated by nodes located in the same cluster as the servers. Clients send requests (20B of payload) in a closed loop, waiting for the answer to the current request before sending the next one. We consider a setup with $n = 4$ servers that can tolerate one faulty server ($f = 1$). Each experiment was run for 3 minutes, with the first minute ignored in the calculation of the results. The service is a simple (stateless) echoing service that sends back the request as its response.

We use as metrics (i) the throughput in requests per second and (ii) the client response time. The *client response time* is the time from the point the client sends a request until it receives the corresponding reply from $f + 1$ servers. Note that client response time includes delays incurred by queuing of requests at servers. As mentioned in Section I, guarantees for client latency assume a maximum client load.

B. Failure-free executions

In the failure-free executions, we are interested in the maximum throughput and the corresponding response time under different load (number of clients). We vary the number of clients from 30 to 500, which were evenly distributed over 15 nodes. The the number of concurrent instances was set to 4 for PBFT, and to 1 for BFT-Mencius and Spinning. These values led to the best results.

As we can see on Figure 2, BFT-Mencius has slightly lower throughput and higher latency until 120 clients. This is due to the batching policy used, as the number of clients is not enough to fill the batch. We set the batch size to 1350B so that the size of the frame matches the natural limits of the underlying Ethernet network. The server timeout for proposing the batch was set to 3ms. The same batching policy is used with PBFT, while for Spinning we used the adaptive batching strategy used also in the original Spinning implementation [15].[12] With BFT-Mencius 120 clients or less is not enough to fill the batch, i.e., the latency is dominated by the batch timeout (3ms). With more than 120 clients, BFT-Mencius is performing comparably to PBFT and Spinning.

[11] Note that we use this estimate only for suspecting in the blacklisting mechanism; Algorithm 3 uses worst case bounds for its $timeout$.

[12] Using adaptive batching with BFT-Mencius and PBFT led to the similar results.

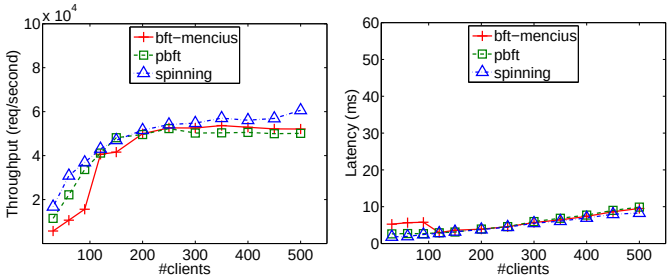
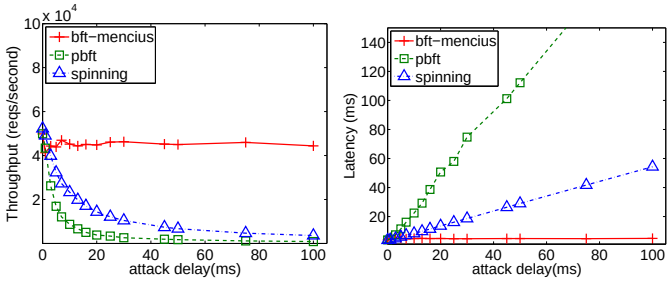


Fig. 2: Throughput and latency of BFT-Mencius, PBFT and Spinning for different client load in the failure-free case.



(a) Throughput for different client load (b) Latency for different client load

Fig. 3: Throughput and latency of BFT-Mencius, PBFT and Spinning under performance failures.

C. Executions with performance failures by faulty servers

In order to measure performance of BFT-Mencius under performance failures, we run experiments where the faulty server delays sending of the PRE−PREPARE message in instances it owns. Similarly, for Spinning and PBFT, the faulty server delays sending PRE−PREPARE whenever it is coordinator. Otherwise, the server normally participates in all algorithms. The number of clients for this experiment was set to 200 for all algorithms. We have chosen number of clients based on failure-free case where performance started to level off for all algorithms. For BFT-Mencius, we set K_{lat} (see Section VI-B) to 1. Thus we use $2 \cdot d_{ATAB}$ as suspect timeout.

We have measured performance of the three protocols by varying values for the attack delay. As we can observe on Figure 3, the performance of PBFT and Spinning, compared to the failure-free case, can be significantly degraded by a faulty replica doing a performance attack. We can also observe that always rotating the primary makes Spinning more robust to performance attacks than PBFT: its average latency is significantly better, albeit still dependent on the attack delay.

On the other hand, once the attack delay is above the suspicion timeout, the blacklisting mechanism of BFT-Mencius allows us to skip instances of blacklisted servers. Setting the suspicion timeout as explained in Section VI-B, faulty servers are blacklisted already with an attack-delay of 3ms. This allows BFT-Mencius to achieve latency always below 5ms, and throughput close to the peak throughput achieved in the failure-free case.

VIII. Conclusion

We have proposed a new state machine replication protocol for partially synchronous systems with Byzantine faults. The algorithm, called *BFT-Mencius*, is a modular, signature-free SMR protocol that ensures bounded-delay, i.e., eventual bounded latency during periods of synchrony, even in the presence of Byzantine processes. BFT-Mencius is based on a new communication primitive, *Abortable Timely Announced Broadcast* (ATAB). In cluster settings, BFT-Mencius achieves bounded latency *and* throughput comparable to state-of-the-art algorithms such as PBFT and Spinning in fault-free configurations. In other words, contrary to these protocols, BFT-Mencius is able to maintain the same performance under performance attacks.

Acknowledgement

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

References

[1] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, 1978.

[2] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, Dec. 1990.

[3] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: wait-free coordination for internet-scale systems," in *USENIXATC*, 2010, p. 11.

[4] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *OSDI*, 2006, pp. 335–350.

[5] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 205–220, Oct. 2007.

[6] L. A. Barroso and U. Hoelzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, 1st ed. Morgan and Claypool Publishers, 2009.

[7] K. Driscoll, B. Hall, H. Sivencrona, and P. Zumsteg, "Byzantine fault tolerance, from theory to reality," in *Computer Safety, Reliability, and Security*, 2003, vol. 2788, pp. 235–248.

[8] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *Journal of the ACM*, vol. 27, no. 2, pp. 228–234, 1980.

[9] M. K. Reiter, "Secure agreement protocols: reliable and atomic group multicast in rampart," in *CCS*, 1994, pp. 68–80.

[10] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACMTCS*, 2002.

[11] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," *ACM Trans. Comput. Syst.*, vol. 27, no. 4, pp. 7:1–7:39, Jan. 2010.

[12] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Comput. Surv.*, vol. 36, no. 4, pp. 372–421, Dec. 2004.

[13] Y. Amir, B. Coan, J. Kirsch, and J. Lane, "Prime: Byzantine replication under attack," *IEEE Trans. Dependable Secur. Comput.*, vol. 8, no. 4, pp. 564–577, Jul. 2011.

[14] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making byzantine fault tolerant systems tolerate byzantine faults," in *NSDI*, 2009, pp. 153–168.

[15] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung, "Spin one's wheels? byzantine fault tolerance with a spinning primary," in *SRDS*, 2009.

[16] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: building efficient replicated state machines for wans," in *OSDI*, 2008, pp. 369–384.

[17] ——, "Towards low latency state machine replication for uncivil wide-area networks," in *HotDep*, 2009.

[18] H. Attiya, F. Borran, M. Hutle, Z. Milosevic, and A. Schiper, "Structured derivation of semi-synchronous algorithms," in *DISC*, 2011, pp. 374–388.

[19] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth, "Bar fault tolerance for cooperative services," in *SOSP*, 2005, pp. 45–58.

[20] F. Borran and A. Schiper, "A leader-free byzantine consensus algorithm," in *ICDCN*, 2010, pp. 67–78.

[21] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup, "Secure and efficient asynchronous broadcast protocols," in *CRYPTO*, 2001, pp. 524–541.

[22] H. Moniz, N. F. Neves, M. Correia, and P. Verissimo, "Ritas: Services for randomized intrusion tolerance," *IEEE Trans. Dependable Secur. Comput.*, vol. 8, no. 1, pp. 122–136, Jan. 2011.

[23] Z. Milosevic, M. Hutle, and A. Schiper, "On the reduction of atomic broadcast to consensus with byzantine faults," in *SRDS*, 2011, pp. 235–244.

[24] V. Hadzilacos and S. Toueg, "Fault-tolerant broadcasts and related problems," in *Distributed systems (2nd Ed.)*, S. Mullender, Ed., 1993, pp. 97–145.

[25] ——, "A modular approach to fault-tolerant broadcasts and related problems," Ithaca, NY, USA, Tech. Rep., 1994.

[26] A. Doudou and A. Schiper, "Muteness detectors for consensus with Byzantine processes (Brief Announcement)," in *PODC*, Jul. 1998.

[27] O. Rütti, Z. Milosevic, and A. Schiper, "Generic construction of consensus algorithms for benign and byzantine faults," in *DSN*, 2010, pp. 343–352.

[28] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, May 1998.

[29] A. S. Aiyer, L. Alvisi, R. A. Bazzi, and A. Clement, "Matrix signatures: From macs to digital signatures in distributed systems," in *DISC*, 2008, pp. 16–31.

[30] M. Biely, P. Delgado, Z. Milosevic, and A. Schiper, "Distal: A framework for implementing fault-tolerant distributed algorithms," to appear in DSN, 2013.