ACE: Abstract Consensus Encapsulation for Liveness Boosting of State Machine Replication

Alexander Spiegelman
Novi Research, Menlo Park, USA

Arik Rinberg[1]
Technion, Haifa, Israel

Dahlia Malkhi
Novi Research, Menlo Park, USA

Abstract

With the emergence of attack-prone cross-organization systems, providing asynchronous state machine replication (SMR) solutions is no longer a theoretical concern. This paper presents *ACE*, a framework for the design of such fault tolerant systems. Leveraging a known paradigm for randomized consensus solutions, ACE wraps existing practical solutions and real-life systems, boosting their liveness under adversarial conditions and, at the same time, promoting load balancing and fairness. Boosting is achieved without modifying the overall design or the engineering of these solutions.

ACE is aimed at boosting the prevailing approach for practical fault tolerance. This approach, often named *partial synchrony*, is based on a leader-based paradigm: a good leader makes progress and a bad leader does no harm. The partial synchrony approach focuses on safety and forgoes liveness under targeted and dynamic attacks. Specifically, an attacker might block specific leaders, e.g., through a denial of service, to prevent progress. ACE provides boosting by running *waves* of parallel leaders and selecting a *winning* leader only retroactively, achieving boosting at a linear communication cost increase.

ACE is agnostic to the fault model, inheriting it s failure model from the wrapped solution assumptions. As our evaluation shows, an asynchronous Byzantine fault tolerance (BFT) replication system built with ACE around an existing partially synchronous BFT protocol demonstrates reasonable slow-down compared with the base BFT protocol during faultless synchronous scenarios, yet exhibits significant speedup while the system is under attack.

2012 ACM Subject Classification Security and privacy → Distributed systems security; Software and its engineering → Abstraction, modeling and modularity

Keywords and phrases Framework, Asynchronous, Consensus boosting, State Machine Replication

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2020.9

Related Version A full version of the paper is available at [49], `https://arxiv.org/abs/1911.10486`.

1 Introduction

Building reliable systems via state machine replication (SMR) requires resilience against all network conditions, including malicious attacks. The best way to model such settings is by assuming asynchronous communication links. However, as shown in the FLP result [27], deterministic asynchronous SMR solutions are impossible.

Two principal approaches are used to circumvent this result. The first is by assuming *partial synchrony* [25], in which protocols are designed to guarantee safety under worst case network conditions, but are able to satisfy progress only during "long enough" periods of network synchrony. Protocols in this model normally follow the leader-based view-by-view

[1] Part of this work has been done while Arik Rinberg was an intern at VMware Research Group.

paradigm due to its speed during synchronous attack-free periods and relative simplicity. In fact, most deployed systems, several of which have become the de facto standards for building reliable systems (e.g., Paxos [35], PBFT [19], Zyzzyva [34], Zookeeper [2] Raft [45] and others [45, 4, 51, 13]), adopt this approach. The drawback of the partial synchrony model is that it fails to capture adaptive network attacks [48], leaving the leader-based view-by-view algorithms vulnerable. For example, an attacker can prevent progress by adaptively blocking the communication of the leader of every view.

The second approach to circumventing the FLP impossibility is by employing randomization [12, 47, 21]. Randomized algorithms typically satisfy safety properties, but ensure liveness only with probability approaching 1, albeit operate at network speed under all network conditions. There are many theoretical works on asynchronous consensus (also called agreement) in the literature, but since they are typically very complex or inefficient, only a few of them were used in academic asynchronous SMR systems [5, 24] and we are not aware of any deployed in practice.

Main contribution. This paper presents *ACE*, a framework for *asynchronous boosting* that converts consensus algorithms designed according to the *leader-based view-by-view paradigm* in the partial synchrony model into randomized fully asynchronous SMR solutions. ACE provides boosting by running *waves* of parallel leaders and selecting a *winning* leader only retroactively. As a result, with ACE, a system designer can benefit twofold: (1) from the experience gained in decades of leader-based view-by-view algorithm design and system engineering, and (2) from a robust asynchronous solution that is live under attacks.

An additional feature of ACE is the following notion of *fairness*. Due to the unpredictability of the election mechanism, ACE guarantees that for each slot the probability of parties to agree on a value proposed by an honest party is at least 1/2. Another important feature of ACE is that it is model agnostic and can be applied to any leader-based protocol in the Byzantine or crash failure model. As a result, when instantiated with a BFT protocol such as PBFT [19] we get asynchronous byzantine state machine replication, and when instantiated with a crash-failure solution like Paxos [35] or Raft [45] we get the first asynchronous SMR system tolerating any minority of failures.

1.1 Technical Approach

View-by-view paradigm. Leader-based view-by-view protocols divide executions into a sequence of views, each with a designated leader. Every view is then further divided into two phases. First, the *leader-based* phase in which the designated leader tries to drive progress by getting all parties to commit its value, and all other parties start a timer to monitor the progress. If the timer expires before a decision is reached (due to a faulty leader or long network delays), the parties switch to the *view-change* phase. In this phase the parties *exchange* information to safely *wedge* the current view and proceed to the next one, restarting the process with the new designated leader.

Wave-by-Wave. Our solution boosts asynchronous liveness by eliminating the need in internal timers. To thus end, ACE leverages a known theoretical paradigm [31, 9] of running n leaders in parallel and retrospectively choosing one. In particular, ACE provides boosting by running a *wave* of n leader-base phases in parallel, waiting for a quorum of leaders to progress, and then randomly electing one leader to proceed to its *view-change* phase. Importantly, no timers are used. Rather, the trigger to move to the *view-change* phase is external and happens upon the completion of “enough” leaders.

To be able to externally switch between the phases, ACE provides a formal characterization of the leader-based view-by-view protocols by defining a *leader-based view (LBV)* abstraction, which encapsulates the main properties of a single view. Its API de-couples the *leader-based* phase from the *view-change* phase and allows each of them to be separately invoked – engage triggers the leader-based phase, and wedge&exchange triggers the view-change phase.

We define the properties of LBV, and conjecture that existing view-by-view algorithms implicitly satisfy them. Moreover, decomposing such algorithms according to the LBV abstraction yields a sequence of LBV's with an external timer triggering phase transitions, as depicted in Figure 1.

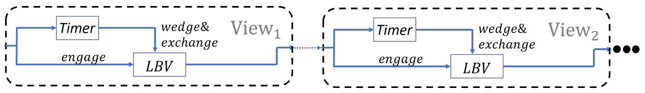


Figure 1 Using a sequence of LBV instances to reconstruct a partially synchronous leader-based view-by-view protocol.

More specifically, in a single-shot agreement protocol, an ACE wave operates as follows: Instead of running one LBV instance (as view-by-view protocols do), a wave runs n LBV instances (the leader-based phase) simultaneously, each with a distinct leader. Then, the wave performs a barrier synchronization in which parties wait until a quorum of the instances have completed. The barrier is eventually reached due to a key property of the LBV abstraction, which guarantees that if the leader is correct and no correct party invokes *view-change*, then all correct parties eventually commit a value.

After the barrier is reached, one LBV instance is selected unpredictably and uniformly at random. The chosen instance "wins", and all other instances are ignored. Then, parties use the LBV's wedge&exchange API to invoke the view-change phase in the chosen instance (only). The view-change phase here has two purposes. First, it *boosts termination*. If the chosen LBV instance has reached a decision, meaning that a significantly large quorum of parties have decided in its leader-based phase, then all correct parties learn this decision during the view-change phase. Second, as in every view-by-view protocol, the view-change phase *ensures safety* by forcing the leaders of the next wave to propose safe values.

The next wave enacts n new LBV instances, each with a different leader that proposes a value according to the state returned from the view-change phase of the chosen instance of the previous wave. Note that since parties wait for a large quorum of LBV instances to reach a decision in each wave before randomly choosing one, the chosen LBV has a constant probability of having a decision, hence, together with the termination boosting provided by the view-change phase, we get progress, in expectation, in a constant number of waves.

As to SMR, ACE implements a variant in which parties do not proceed to the next slot before they learn the decision value of the current one, but once they move to the next one they stop participating in the current slot and garbage collect all the associated resources. Deferring next slots until the current decision is known is essential for systems in which the validity of a value for a certain slot depends on all previous decision values (e.g., Blockchains). ACE's SMR solution uses an instance of the single-shot protocol for every slot together with a forwarding mechanism to help slow parties catch-up.

Applicability. ACE can take any view-by-view consensus protocol designed for the partially synchronous model and transform it into an asynchronous SMR solution. In order to instantiate ACE with a specific algorithm, e.g., PBFT [19] or Paxos [35], one only needs to take a single view of the algorithm's logic and wrap it with the LBV API. Therefore, instantiating ACE does not require new logic implementation beyond the engineering effort of providing the API. Furthermore, ACE's modularity provides a clean separation of concerns between safety (provided by the LBV properties) and asynchronous liveness (provided by the framework).

Evaluation. To demonstrate ACE, we choose to focus on the byzantine model as this is the model considered by Blockchain systems and we believe that, due to their high stakes and public infrastructures, Blockchain systems will benefit the most from a generic asynchronous SMR solution that can tolerate network attacks. We implement ACE's algorithms in C++ and instantiate the LBV abstraction with a variant of HotStuff [51] – a state of the art BFT solution, which is currently being implemented in several commercial Blockchain systems [6]. To compare the ACE instantiation to the base (raw) HotStuff implementation, we emulate different adversarial scenarios and generate networks attacks. Our evaluation shows that while base HotStuff outperforms ACE (instantiated with HotStuff) in the synchronous failure-free case, ACE has absolute superiority during asynchronous periods and network attacks. For example, we show that byzantine parties can hinder progress in base HotStuff by targeting leaders with a DDoS attack, whereas ACE manages to commit values at network speed.

Roadmap. The rest of the paper is organized as follows: Section 2 describes the model and formalizes the agreement and SMR problems. Section 3 gives an overview of the leader-based view-by-view paradigm, capturing its main properties and vulnerabilities. Section 4 defines ACE's abstractions, and its algorithms are given in Section 5. Section 6 instantiates ACE and evaluates its performance. Finally, Section 7 discusses related work and Section 8 concludes.

2 Model and Problem Definitions

2.1 System Model

We consider a peer to peer system with n parties, $f < n$ of which may fail. We say that a party is *faulty* if it fails at any time during an execution of a protocol. Otherwise, we say it is *correct*. In a peer to peer system every pair of parties is connected with a *communication link*. A message sent on a link between two correct parties is guaranteed to be delivered, whereas a message to or from a faulty party might be lost. A link between two correct parties is *asynchronous* if the delivery of a message may take arbitrary long time, whereas a link between two correct parties is *synchronous* if there is a bound Δ for message deliveries. In *asynchronous network periods* all links among correct parties are asynchronous, whereas during *synchronous network periods* all such links are synchronous.

A standard communication model assumed by algorithms that follow the view-by-view paradigm is the *partially synchronous* model (also called eventual synchrony [25]). In this model, there is an unknown point in every execution, called *global stabilization time (GST)*, which divides the execution into two network periods: before GST the network is asynchronous and after GST the network is synchronous. The partially synchronous model was defined to capture spontaneous network disconnections in wide-area networks, in which case it is reasonable to assume that asynchronous periods are short and synchronous periods are long enough for the protocols to make progress.

However, the partially synchronous model fails to capture malicious attacks that intentionally try to sabotage progress, and thus are not suitable for many current use cases (e.g., Blockchains). For example, one possible attack is the *weakly adaptive asynchronous* in which an attacker adaptively blocks one party at a time from sending or receiving messages (e.g., via DDOS). This results in a *mobile* asynchrony that moves from party to party, violating the GST assumption made by the partially synchronous model, and thus prevents progress from all leader-based view-by-view algorithms.

ACE, in contrast, assumes the fully asynchronous communication model, and thus progress in network speed under all network conditions and attacks as long as messages among correct parties are eventually delivered.

As mentioned in the Introduction and explained in more detail below, ACE abstracts away specific model assumptions and implementation details into three primitives: *Leader based view (LBV)*, *leader-election*, and *barrier*. In Section 4, we define the properties of these primitives and require that any leader-based view-by-view protocol that is instantiated into our framework satisfies them. To satisfy these properties, each protocol may have different model assumptions: for example, the relation between f and n, the failure types (e.g., crash and byzantine), and cryptographic assumptions. ACE inherits the specific assumptions made by each of the protocols it is instantiated with, and adds nothing to them. In other words, whatever assumptions are made by the instantiated protocol in order to satisfy the abstractions' properties, are exactly the assumptions under which ACE operates.

2.2 Problem Definition

We define the fair validated single-shot agreement problem below, and for space limitation defer the definition of the generalized SMR problem to to the full paper [49].

The *fair validated agreement* [9, 16, 15] is a single-shot problem in which correct parties propose externally valid values and agree on one unique such value. The formal properties are given below:

- Agreement: All correct parties that decide, decide on the same value.
- Termination: If all correct parties propose valid values, then all correct parties decide with probability 1.
- Validity: If a correct party decides an a value v, then v is externally valid.

Note that the agreement and termination properties are not enough by to guarantee real progress of any multi-shot agreement system (e.g., Blockchain) that is built on top of the single-shot problem. Without external validity, parties are allowed to agree on some pre-defined value (i.e., $\perp$) [43], which is basically an agreement not to agree. Moreover, as long as a value satisfies the system's external validity condition (e.g., no contradicting transactions in a blockchain system), parties may decide on this value even if it was proposed by a byzantine party. However, since high stake is involved and byzantine parties may try to increase the ratio of decision values proposed by them, we require an additional fairness property that is a generalization of the *quality* property defined in [9]:

- Fairness: The probability for a correct party to decide on a value proposed by a correct party is at least $1/2$. Moreover, during synchronous periods, all correct parties have an equal probability of $1/n$ for their values to be chosen.

Intuitively, note that by simply following the protocol byzantine parties can have a probability of $1/3$ (recall that $1/3$ of the parties are byzantine) for their value to be chosen in every protocols even during synchronous periods. And since during asynchronous periods the adversary can, in addition, block $1/3$ of the correct parties, we get that byzantine parties can increase their probability to $1/2$. Meaning that the fairness property we require is optimal.

3 The View-by-View Paradigm

Many (if not all) practical agreement and consensus algorithms operate a leader-based view-by-view paradigm, designed for partially synchronous models, including the seminal work of Dwork et al. [25] pioneering the approach, and underlying classical algorithms like Paxos [35], Viewstamped-Replication [44], PBFT [19], and others [34, 45].

Protocols designed according to the view-by-view paradigm advance in views. Each view has a designated leader that proposes a value and tries to convince other parties to decide on it. To tolerate faulty leaders from halting progress forever, parties use timers to measure leader progress; if no progress is made they demote the leader, abandoning the current view and proceeding to the next one.

The main problem with this approach is that a faulty leader that does not send any messages is indistinguishable from a correct leader with asynchronous links. Therefore, protocols implementing this approach are not able to guarantee progress during asynchronous periods or weakly adaptive asynchronous attacks since parties advance views before correct leaders are able to drive decisions. Below we discuss the main properties of algorithms designed according to the view-by-view paradigm:

3.1 Main properties

Safety. Perhaps the most important property of such algorithms is their ability to satisfy safety during arbitrary long asynchronous periods. This is achieved via a careful *view-change* mechanism that governs the transition between views. View-change consists of parties *wedging* the current view by abandoning the current leader, and *exchanging* information about what might have committed in the view (the closing state of the view). In the new view, parties participate in the new leader's phase only if it proposes a value that is safe in accordance with the closing state.

Liveness. Algorithms that rely on leaders to drive progress cannot guarantee progress during asynchronous periods since they cannot distinguish between faulty leaders and correct ones with asynchronous links. During asynchronous periods, messages from the current leader may be delivered only after parties timeout and move to the next view regardless of how conservative the timeouts are.

However, all these algorithms share an important property that our framework utilizes: for every view, if the leader of the view is correct and no correct party times out and abandons this view, then all correct parties decide in this view.

3.2 Practical Vulnerabilities

Deploying view-by-view algorithms requires tuning the leader timeouts. On the one hand, aggressive timeouts set close to the common network delay might cause correct leaders to be demoted due to spurious delays, and destabilize the system. On the other, conservative timeouts implies delayed actions in case of faulty leaders. It further opens the system to possible attacks by byzantine leaders that slow system progress to the maximum possible without triggering a timeout.

Another attack on the progress of leader-based protocols is the weak adaptive asynchrony in which an attacker blocks communication with the leader of each view until the view expires, e.g., via distributed denial-of-service attack. Last, a carefully executed adaptive asynchrony attack can cause a fairness bias. Some leaders (possibly byzantine) may be allowed to progress and commit their values, whereas an attacker blocks communication with other designated (possibly all correct) leaders. In Section 6, we demonstrate the above attacks, and show that ACE is resilient against them.

4 Framework abstractions

ACE provides "asynchronous boosting" for partially synchronous protocols designed according to the leader-based view-by-view paradigm. In a nutshell, ACE takes such a protocol, encapsulates a single view of the protocol into a *leader-based view (LBV)* abstraction that provides API to avoid timeouts, composes LBVs into a wave of n instances running in parallel, interjects auxiliary actions in between successive waves, and chooses one LBV instance retrospectively at random. Detailed description is given in the next section. Section 4.1 defines the *Leader based view (LBV)* abstraction and the auxiliary abstractions utilized by ACE, Barrier and Leader-election, are given in the full paper [49].

4.1 Encapsulating view-based agreement protocols

As explained above, each view in a leader-based view-by-view algorithm consists of two phases: First, all parties wait for the leader to perform the *leader-based* phase to drive decision on some value v, and then, if the leader fails to do it fast enough, parties switch to the *view-change* phase in which they *wedge* the current leader and *exchange* information in order to get the closing state of the view. To decide when to switch between the phases, existing algorithms use timeouts, which prevent them from guaranteeing progress during asynchronous periods. Therefore, in order to boost asynchronous liveness, ACE replaces the timeout mechanism with a different strategy to switch between the phases. To this end, the LBV abstraction exposes an API with two methods, engage and wedge&exchange, where engage starts the first phase of the view (leader-based), and wedge&exchange switches to the second (view-change). By exposing API with these two methods, we remove the responsibility of deciding when to switch between the phases from the view (e.g., no more timeouts inside a view) and give it to the framework, while still preserving all safety guarantees provided by each view in a leader-based view-by-view protocol.

Every instance of the LBV abstraction is parametrized with the leader's name and with an identification id, which contains information used by the high-level agreement algorithm built (by the framework) on top of a composition of LBV instances. The wedge&exchange method gets no parameters and returns a tuple $\langle s, v\rangle$, where v is either a value or $\bot$; and s is the closing state of the instance, consisting of all necessary information required by the specific implementation of the abstraction (e.g., a safe value for a leader to propose). The engage method gets the "closing state" s that was returned from wedge&exchange in the preceding LBV instance (or the initial state in case this is the first one), and outputs a value v. Intuitively, the returned value from both methods is the "decision" that was made in the LBV instance, but as we explain below, the high-level agreement algorithm might choose to ignore this value.

The safety of view-by-view algorithms strongly relies on the fact that correct parties start a new view with the closing state of the previous one. Otherwise, they cannot guarantee that correct parties that decide in different views decide on the same value. Therefore, when we encapsulate a single view in our LBV abstraction and define its properties, we consider only executions in which the LBV instances are composed one after another. Formally, we say that the LBV abstractions are *properly composed by a party* p_i in an execution if p_i invokes the engage of the first instance with some fixed initial state (which depends on the instantiated protocol), and for every instance $k > 1$, p_i invokes its engage with the state output of wedge&exchange of instance $k - 1$. In addition, we say that the LBV abstractions are *properly composed* in an execution if they are properly composed by all correct parties. Figure 2 illustrates LBV's API and its properly composed execution.

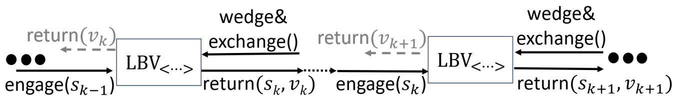


Figure 2 A properly composed execution: The engage method of instance $k > 1$ gets the state output of the wedge&exchange method of instance $k - 1$.

The formal definition of the LBV abstraction is as follows:

▶ **Definition 1.** *A protocol implements an LBV abstraction if the following properties are satisfied in every properly composed execution that consists of a sequence of LBV instances:*
Liveness.

- ***Engage-Termination:*** *For every instance with a correct leader, if all correct parties invoke* engage *and no correct party invokes* wedge&exchange*, then* engage *invocations by all correct parties eventually return.*
- ***Wedge&Exchange-Termination:*** *For every instance, if all correct parties invoke* wedge&exchange *then all* wedge&exchange *by correct parties eventually return.*

Safety.

- ***Validity:*** *For every instance, if an* engage *or* wedge&exchange *invocation by a correct party returns a value v, then v is externally valid.*
- ***Completeness:*** *For every instance, if $f + 1$* engage *invocations by correct parties return, then no* wedge&exchange *invocation by a correct party returns a value $v = \bot$.*
- ***Agreement:*** *If an* engage *or* wedge&exchange *invoked in some instance by a correct party returns a value $v \neq \bot$ and some other* engage *or* wedge&exchange *invoked in some instance by a correct party returns $v' \neq \bot$ then $v = v'$.*

Note that during the view-change phase in most leader-based protocols, parties send the closing state only to the leader of the next view. However, in ACE, since we run n concurrent LBV instances, each with a different leader, we need all parties to learn the closing state after wedge&exchange returns. Moreover, as mentioned above and captured by the Completeness property, we use wedge&exchange to also boost decisions in order to guarantee that if the retrospectively chosen LBV instance successfully completed the first (leader-based) phase, than all correct parties decide at the end of its second phase. Therefore, when encapsulating the view-change mechanism of a leader-based protocol into the wedge&exchange method, a small change has to be made in order to satisfy the above properties. Instead of sending the closing state only to the next leader, parties need to exchange information by sending the closing state to all parties and wait to receive $n - f$ such messages. No change is needed to the first phase of the encapsulated leader-based protocol since all the required properties for engage are implicitly satisfied.

5 Framework algorithms

In this section we present ACE's asynchronous boosting algorithms, which are built on top of the abstractions defined above. The algorithm for an asynchronous single-shot agreement is given below, and for space limitation, we show how to turn it into an asynchronous SMR in the full paper [49]. For completeness, the full paper [49], we show how to use the LBV abstraction to reconstruct the base partially synchronous view-by-view algorithm the LBV is instantiated with.

The pseudocode for the asynchronous single-shot agreement protocol appears in Algorithm 1 and a formal correctness proof can be found in the full paper [49]. An invocation of the protocol ($SS\text{-}propose(id, S)$) gets an initial state S and identification id, where the initial state S contains all the initial specific information (including the proposed value) required by the leader-based view-by-view protocol instantiated in the LBV abstraction.

Algorithm 1 Asynchronous single-shot agreement.

```
upon  SS-propose(id,S) do
    state ← S ; wave ← 1
    while true do
        ID ← ⟨id, wave⟩
        ⟨state′, value⟩ ← WAVE(ID, state)
        if value ≠ ⊥ and did not decide before then
            decide ⟨id, value⟩
        state ← state′
        wave ← wave + 1
upon  engage ⟨ID,p_j⟩ returns v do
    send "ID, ENGAGE-DONE" to party p_j

procedure WAVE(ID, state)
    for all p_j = p_1, . . . , p_n do
        invoke engage ⟨ID,p_j⟩(state)
        //non-blocking
    barrier-sync ID()
    leader ← elect_ID()
    return wedge&exchange ⟨ID,leader⟩()
upon  receiving n − f "ID,ENGAGE-DONE" do
    invoke barrier-ready ID()
```

The protocol proceeds in a *wave-by-wave* manner. The state is updated at the and of every wave and a decision is made the first time a wave returns a non-empty value. In every wave, each party first invokes the engage operation in n LBV instances, each with a different leader. Each invocation gets the state obtained at the end of the previous wave or the initial state if this is the first wave.

Then, parties invoke barrier-sync and wait for it to return. Recall that by the B-Coordination property, barrier-sync returns only after $f+1$ correct parties invoke barrier-ready. When an engage invocation in an LBV instance with leader p_j returns, a correct party sends an "ENGAGE-DONE" message to party p_j, and whenever a party gets $n - f$ such messages it invokes barrier-ready. Denote an LBV instance as *successfully completed* when $f + 1$ correct parties completed the first phase, i.e., their engage returned, and note, therefore, that a correct party invokes barrier-ready only after the LBV instance in which it acts as the leader was successfully completed. Thus, a barrier-sync invocation by a correct party returns only after $f + 1$ LBV instances successfully completed.

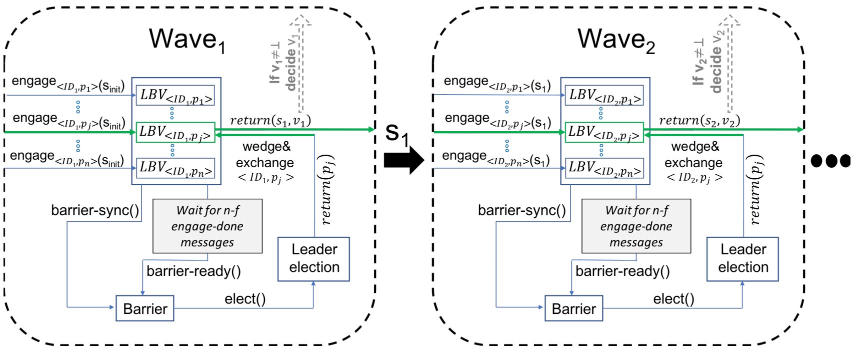

Wave$_1$
If $v_1 \neq \perp$ decide v_1
engage$_{<ID_1,p_1>}$(s$_{init}$)
engage$_{<ID_1,p_j>}$(s$_{init}$)
engage$_{<ID_1,p_n>}$(s$_{init}$)
$LBV_{<ID_1,p_1>}$
$LBV_{<ID_1,p_j>}$
$LBV_{<ID_1,p_n>}$
$return(s_1, v_1)$
wedge& exchange $<ID_1, p_j>$
$return(p_j)$
barrier-sync()
Wait for n-f engage-done messages
barrier-ready()
Leader election
Barrier
elect()
S$_1$
Wave$_2$
If $v_2 \neq \perp$ decide v_2
engage$_{<ID_2,p_1>}$(s$_1$)
engage$_{<ID_2,p_j>}$(s$_1$)
engage$_{<ID_2,p_n>}$(s$_1$)
$LBV_{<ID_2,p_1>}$
$LBV_{<ID_2,p_j>}$
$LBV_{<ID_2,p_n>}$
$return(s_2, v_2)$
wedge& exchange $<ID_2, p_j>$
$return(p_j)$
barrier-sync()
Wait for n-f engage-done messages
barrier-ready()
Leader election
Barrier
elect()

Figure 3 Asynchronous single-shot algorithm. The chosen LBVs, marked in green, are properly composed.

Next, when the barrier-sync returns, parties elect a unique leader via the leader-election abstraction, and further consider only its LBV instance. Note that since parties wait until $f+1$ LBV instances have successfully completed before electing the leader, with a constant probability of $\frac{f+1}{n}$ the parties elect a successfully completed instance (can be improved to $\frac{2f+1}{n}$ in the byzantine case with $n = 3f+1$), and even an adaptive adversary has no power to prevent it.

Finally, all parties invoke wedge&exchange in the elected LBV instance to wedge and find out what happened in its first phase, using the returned state for the next wave and possibly receiving a decision value. By the Completeness property of LBV, if a successfully completed LBV instance is elected, then all wedge&exchange invocations by correct parties return $v \neq \bot$ and thus all correct parties decide v in this wave. Therefore, after a small number of $\frac{n}{f+1}$ waves all correct parties decide in expectation. Note that the sequence of chosen LBV instances form a properly composed execution, and thus since parties return only values returned from chosen LBVs, our algorithm inherits its safety guarantees from the leader-based protocol the LBV is instantiated with. An illustration of the algorithm appears in Figure 3.

6 ACE Instantiation

There are many possible ways to instantiate the ACE framework. We choose to evaluate ACE in the byzantine failure model with $n = 3f + 1$ parties and a computationally bounded adversary due to the attention it gets in the Blockchain use-case. For the LBV abstraction, we implement a variant of HotStuff [51]. For the leader-election we implement the protocol in [9, 16], and for the Barrier we give an implementation that operates in the same model. All protocols use a BLS threshold signatures schema [14] that requires a setup, which can be done with the help of a trusted dealer or by using a protocol for an asynchronous distributed key generation [33]. Communication is done over TCP to provide reliable links. Due to the space limitation, implementation details can be found in the full paper [49]. The communication complexity of a single LBV is linear and that of the barrier and leader-election is quadratic, leading to an expected total quadratic communication, for each slot.

Our evaluation compares the performance of ACE's SMR instantiated with HotStuff, we refer to as *ACE HotStuff*, with the base HotStuff SMR implementation. To compare apples to apples, the base HotStuff and ACE HotStuff share as much code as possible. In Section 6.1 we present the tests' setup. Then, in Section 6.2, we measure ACE's overhead during failure-free synchronous periods, and in Section 6.3 we demonstrate ACE's superiority during asynchronous periods and network attacks.

6.1 Setup

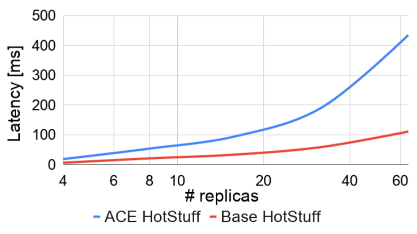
We conducted our experiments using c5d.4xlarge instances on AWS EC2 machines in the same data center. We used between 1 and 16 virtual machines, each with 4 replicas. The duration of every test was 60 seconds, and every test was repeated 10 times. The size of the proposed values is 10000 bytes. The latency is measured starting from when a new slot has begun until a decision is made. The throughput is measured in one of two ways. In tests where we altered the number of replicas, the throughout is the total number of bytes committed, divided by the length of the test. In tests where we show the throughout as a function of time, we aggregate the number of committed bytes in 1 second intervals. We did not throttle the bandwidth in any run, rather we altered the transmission delays between the machines, using NetEm [7].

6.2 ACE's overhead

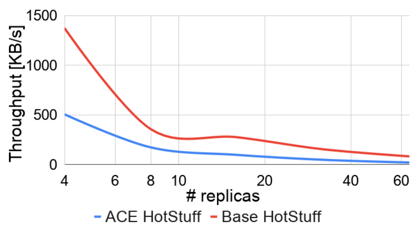
The first set of tests compare ACE HotStuff performance with that of base HotStuff under optimistic, synchronous, faultless conditions. Figure 4 depicts the latency and throughput. The delay on the links was measured to be under $1ms$. The latency increases with the growth in the number of replicas since each replica must handle an equal growth in the number of messages. Furthermore, as ACE HotStuff has a larger overhead than base HotStuff, the latency grows faster.

Figure 5 shows the latency and throughput with different delays added to the links. The latency of ACE HotStuff is twice that of base HotStuff. This is expected, as ACE is expected to execute 1.5 waves per slot, leading to 1.5x the latency. Add on the additional barrier, leader election abstraction and we arrive at 2x reduction in performance.

These tests show that the performance cost of using ACE is about 2x reduction in performance in the optimistic case. In the next tests we argue this cost is sometimes worth paying, as liveness of partially synchronous algorithms can be easily affected.

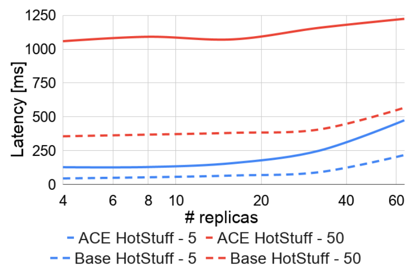


(a) Latency.

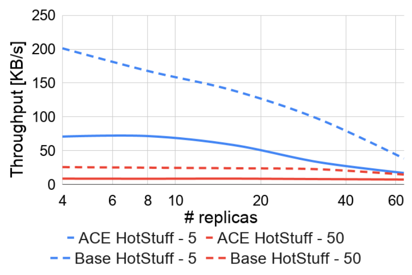


(b) Throughput.

Figure 4 Optimistic case with no network delay.



(a) Latency.



(b) Throughput.

Figure 5 Optimistic case under different network delays.

6.3 ACE's superiority

From here on we choose a configuration of 32 replicas and set the transmission delay to be 5ms unless specified otherwise. The second set of tests compare ACE HotStuff and base HotStuff in adverse conditions concerning message delays. These tests manipulate two factors, the transmission delays (controlled via NetEm [7]), and the view timeout strategy.

The first test sets base HotStuff view timers to a fixed constant of 100ms, the time needed for a commit assuming a 5ms transmission delay. The test measures the performance drop during a short period in which transmission delays are increased, simulating asynchrony. For the first third of the test the network delay is 5ms, for the next third the delay is 10ms, and finally the delay returns to 5ms.

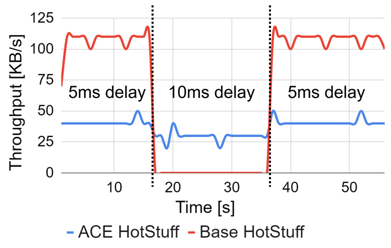


Figure 6 Throughput with a fluctuating transmission delay.

Figure 6 compares the throughput of ACE HotStuff and base HotStuff. While the network delay is 5ms, base HotStuff outperforms ACE HotStuff. However, once the network delay begins to fluctuate, the throughput of base HotStuff goes to 0 since no leader has enough time to drive progress. ACE HotStuff only sees a drop in throughput proportional to the delay, meaning that it continue to progress at network speed.

Note that since the views in base HotStuff are leader-based, byzantine parties (or any other adversarial entity) can achieve the same "asynchronous" effect presented above by only slowing down the leaders. In the next test we demonstrate the above using a *distributed denial of service (DDoS)* attack, in which leaders are flooded with superfluous requests in an attempt to overload them and delay their progress in the leader-based phase.

Figure 7 compares the throughput of ACE HotStuff and base HotStuff, where the attack starts at the halfway mark of the test. The byzantine parties coordinate their attack by adaptively choosing a single correct party and flooding it with superfluous requests. In base HotStuff, byzantine parties target correct leaders (byzantine leaders are making progress). In ACE HotStuff, there is no designated leader, thus byzantine parties choose an arbitrary correct party to attack. Our logs show that in base HotStuff progress is mainly made in views where byzantines parties are leaders. If they would not drive progress, the throughput would drop near 0.

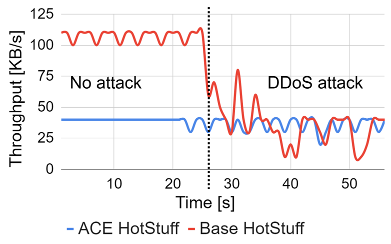


Figure 7 Throughput under DDoS attack.

The previous two scenarios operated base HotStuff with a fixed aggressive view timer, which was based on the expected network delay. This caused premature timer expiration during periods of increased delays (due to asynchrony or attacks). One might think that a possible solution can be to set a very long timeouts that will never expire, thus letting the base HotStuff protocol progress in network speed. However, the downside of conservative timers is that byzantine parties can perform a *silent attack* on the protocol's progress by not driving views when they are leaders, forcing all parties to wait for the long timeouts to expire.

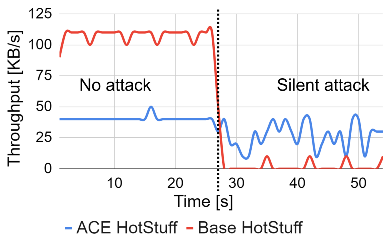


Figure 8 Throughput with conservative timeouts under byzantine silence attack.

The next test evaluates base HotStuff with a conservative view timer of 1 second, fixed to be much higher than expected needed to commit a view, under the silent attack starting at the half way mark. Figure 8 presents the results. Before the attack, base HotStuff indeed progresses in network speed, but during the attack, the throughput drops significantly since a few consecutive byzantine leader might stall progress for seconds. In ACE HotStuff we see a much smaller drop, but more fluctuation. This is due to the fact that byzantine leaders do not drive progress in their LBV instances, and thus the expected number of waves until a decision is now higher.

As the scenarios above demonstrate, neither being too aggressive nor being too conservative works well for base HotStuff during asynchrony or attacks. Therefore, in practice, when HotStuff is deployed it typically adjusts timers during execution according to progress or lack of it. The most common method (used also by PBFT [19] and SBFT [29]) is to increase timeouts whenever timers expires too early, and decrease them whenever progress is made in order to try to learn the network delay and adapt to it's dynamic changes. To test this method, we implement an adaptive version, starting with a delay of t. If a timeout is reached in a view before a decision is made we set the next view's timeout to $1.25t$. Otherwise, the next view's timeout is set to $0.8t$.

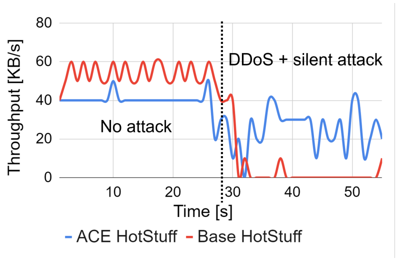


Figure 9 Throughput with adjusting timeouts under a combination of DDoS and silence attacks.

We evaluate this method against the following attack that combines insights from the previous ones. The results are shown in Figure 9. In the second half of the experiment, byzantine parties perform a DDoS attack on correct leaders, causing the view timers to increase, and then perform the silence attack (in views they act as leaders) to stall progress as much as possible. As expected, base HotStuff throughput drops to almost zero, whereas ACE HotStuff continues driving decisions. Same as in the previous test, ACE HotStuff suffers from fluctuation due to the probability to choose a byzantine leader that did not made progress in its LBV instance. Another interesting phenomenon is the x2 performance drop of base HotStuff before the attack begins compared to previous tests. This is due to the timeout adjustment mechanism, which reduces the timers after every successful view, resulting in a too short timeout in every second view.

While the timer adjustment algorithm can be further enhanced, it is an arms race against the adversary – for each method, there is an adversarial response. In addition, although this evaluation is focused on HotStuff, the only ingredient of the algorithm that is under attack is the timeout, hence the evaluation exemplifies the weakness of all leader-based view by view algorithms. Therefore, our evaluation suggests that the overhead of ACE in the optimistic case is worth paying when high availability is desired under all circumstances.

7 Related work

The agreement problem was first introduced by Pease et al. [46], and has since received an enormous amount of attention [17, 8, 50, 19, 34, 41, 39, 10, 20, 11, 42, 40, 23]. One of the most important results is the FLP [27] impossibility, proving that deterministic solutions in the asynchronous communication models are impossible. Below we describe work that was done to circumvent the FLP impossibility, present two related frameworks that were previously proposed for the agreement problem, and discuss alternative fairness definitions. For space limitations, we compare our SMR definition to other systems in the literature in the full paper [49].

Agreement in the partial synchrony model. A practical approach to circumvent the FLP impossibility is to consider the partial synchrony communication model [44, 45, 29, 51, 34], which was first proposed by Dwork et al. [25] and later used by seminal works like Paxos [35] and PBFT [19]. As explained in detail in Section 3, protocols designed for this model never violate safety, but provide progress only during long enough synchronous periods. Despite their limitations, they are widely adopted in the industry due to their relative simplicity compared to the alternatives and their performance benefits during synchronous periods. For example, Casandra [1], Zookeeper [2], and Google's Spanner [26] implement a variant of Paxos [35]; and VMware's Concord [3], the Libra Network [6] and IBM's Hyperledger [5], implement SBFT [29], HotStuff [51] and PBFT [19], respectively.

Agreement in the asynchrony model. As first shown by Ben-Or [12] and Rabin [37], the FLP impossibility result does not stand randomization. Meaning that the randomized version of the Agreement problem, which guarantees termination with probability 1, can be solved in the asynchronous model provided that parties can flip random coins. The algorithms in [12, 37] are very inefficient in terms of time and message complexity, and there has been a huge effort to improve it over the years. Some considered the theoretical full information model, in which the adversary is computationally unbounded, and showed more efficient algorithms that relax the failure resilience threshold [30, 32]. These are beautiful theoretical results but too complex to implement and maintain.

A more practical model for randomized asynchronous agreement is the random oracle model in which the adversary is computationally bounded and cryptographic assumptions (like the Decisional Diffie–Hellman [22]) are valid. In the context of distributed computing, this model was first proposed by Cachin et al. [16, 15]. In [15] they proposed an almost optimal algorithm for the agreement problem. A variant of this algorithm was later implemented in Honeybadger [42] and Beat [24], which are the first academic asynchronous SMR systems. The protocol in [15] is optimal in terms of resilience to failures and round complexity, but has an inefficient $O(n^3)$ communication cost. Improving the communication cost was an open problem for almost 20 years, until it was recently resolved in VABA [9]. ACE borrows a lot from VABA [9].

Frameworks for agreement. There are a few previously proposed agreement frameworks [28, 36, 18] that we are aware of. The authors of [28] and [18] propose frameworks allowing for dynamically switching between protocols. They observed that no byzantine SMR can outperform all others under all circumstances, and introduce a general way for a system designer to switch between implementations whenever the setting changes. Our work is very different from theirs. While they defined an abstraction in order to compose different SMR view-by-view implementations to achieve better performance in the partially synchronous model, our LBV abstraction provides an API to decouple the leader-based phase from the view-change phase in each view, which in turn allows us to compose LBV instances in a novel way that avoids leader demotions via timeouts and boost liveness in asynchronous networks.

Vertical Paxos [36] is a class of consensus algorithms that separates the mechanism for reaching agreement from the one that deals with failures. The idea is to use a fast and small quorum of parties to drive agreement, and have an auxiliary reconfiguration master to reconfigure this quorum whenever progress stalls. The protocol for agreement relies on the participation of all parties in the dedicated quorum, and thus stalls whenever some party fails. The master is emulated by a bigger quorum, which uses an agreement protocol to agree on reconfiguration, and thus can tolerate failures.

Fairness. Although the Agreement and SMR problems have been studied for many years, the question of fairness therein was only recently asked, and we are aware of only few solutions that provide some notion of it [11, 42, 38, 9]. Prime [11] extends PBFT [19] to guarantee that values are committed in a bounded number of slots after they first proposed, and FairLedger [38] uses batching to ensures that all correct party commits a value in every batch. However, in contrast to ACE, both protocols are able to guarantee fairness only during synchronous periods. Honeybadger [42] is an asynchronous protocol that, similarly to FairLedger, batches values proposed by different parties and commits them together atomically. It probabilistically bounds the number of epochs (and accordingly the number of slots) until a value is committed, after being submitted to $n - f$ parties. The VABA [9] protocol does not use batching, and provides a per slot guarantee that bounds the probability to choose a value proposed by a correct party during asynchronous periods. ACE provides similar fairness guarantees during asynchrony, but also guarantees equal chance for each correct party during synchrony.

8 Discussion

In this paper we introduced ACE: a general model agnostic framework for boosting asynchronous liveness of any leader-based SMR system designed for the partially synchronous model. The main ingredient is the novel *LBV* abstraction that encapsulates the properties of a single view in leader-based view-by-view algorithms, while providing an API to control the scheduler of the two phases, leader-based and view-change, in each view. Exploiting this separation, ACE provides a novel algorithm that composes LBV instances in a way that avoids timers and provides a randomized asynchronous SMR solution.

ACE is model agnostic, meaning that it does not add any assumptions on top of what are assumed in the instantiated LBV implementation, thus provides a generic liveness boosting for both byzantine and crash-failure SMRs. In order to instantiate ACE with a specific SMR algorithm, all a system designer needs to do is alter the code of a single view to support LBV's API; this should not be too complicated as the view logic must already implicitly satisfy the required API's properties.

In addition to boosting liveness, ACE is designed in a way that inherently provides fairness due to its randomized election of leaders in retrospect. Moreover, ACE provides a clear separation between safety, which relies on the LBV implementation, and liveness, which is given by the framework. As a result, a system designer that chooses to instantiate ACE gets a modular SMR implementation that is easier to prove correct and maintain – if a better agreement protocol is published, all the designer needs to do in order to integrate it in the system is to alter the LBV implementation accordingly.

To demonstrate the power of ACE we implemented it, instantiated it with the state of the art HotStuff [51] protocol, and compared its performance to the base HotStuff implementation. Our results show that while ACE suffers a 2x performance degradation in the optimistic, synchronous, failure-free case, it enjoys absolute superiority during asynchronous periods and network attacks.

References

1 Apache cassandra. `http://cassandra.apache.org/`. Accessed: 2020-05-03.

2 Apache zookeeper. `https://zookeeper.apache.org/`. Accessed: 2020-05-03.

3 Concord-bft. `https://vmware.github.io/concord-bft/`. Accessed: 2020-05-03.

4 etcd: Reliable key-value store. `https://etcd.io/`. Accessed: 2020-05-03.

5 Hyperledger. `https://www.hyperledger.org/`. Accessed: 2020-05-03.

6 Libra. `https://libra.org/en-US/`. Accessed: 2020-05-03.

7 Netem. `https://www.linux.org/docs/man8/tc-netem.html`. Accessed: 2020-05-03.

8 Michael Abd-El-Malek, Gregory R Ganger, Garth R Goodson, Michael K Reiter, and Jay J Wylie. Fault-scalable byzantine fault-tolerant services. In *Operating Systems Review*, 2005.

9 Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically optimal validated asynchronous byzantine agreement. In *PODC 2019*, New York, NY, USA, 2019. ACM.

10 Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. Customizable fault tolerance forwide-area replication. In *Reliable Distributed Systems, 2007. SRDS 2007.*, 2007.

11 Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. Prime: Byzantine replication under attack. *IEEE Transactions on Dependable and Secure Computing*, 2011.

12 Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *PODC 1983*. ACM, 1983.

13 Alysson Bessani, João Sousa, and Eduardo EP Alchieri. State machine replication for the masses with bft-smart. In *DSN 2014*. IEEE, 2014.

14 Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2001.

15 Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Advances in Cryptology*, 2001.

16 Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *Journal of Cryptology*, 18, 2000.

17 Ran Canetti and Tal Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, 1993.

18 Carlos Carvalho, Daniel Porto, Luís Rodrigues, Manuel Bravo, and Alysson Bessani. Dynamic adaptation of byzantine consensus protocols. In *SAC 2018*, 2018.

19 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *OSDI 1999*, 1999.

20 Allen Clement, Edmund L Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *NSDI*, 2009.

21 Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. Netpaxos: Consensus at network speed. In *SOSR 2015*, 2015.

22 Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976.

23 Sisi Duan, Hein Meling, Sean Peisert, and Haibin Zhang. Bchain: Byzantine replication with high throughput and embedded reconfiguration. In Marcos K. Aguilera, Leonardo Querzoni, and Marc Shapiro, editors, *Principles of Distributed Systems*, pages 91–106, Cham, 2014. Springer International Publishing.

24 Sisi Duan, Michael K Reiter, and Haibin Zhang. Beat: Asynchronous bft made practical. In *CCS 2018*, 2018.

25 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

26 James C. Corbett et al. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 2013.

27 Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

28 Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 bft protocols. In *Proceedings of the 5th European conference on Computer systems*, pages 363–376. ACM, 2010.

29 Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. Sbft: a scalable and decentralized trust infrastructure. In *DSN 2019*, 2019.

30 Bruce M Kapron, David Kempe, Valerie King, Jared Saia, and Vishal Sanwalani. Fast asynchronous byzantine agreement and leader election with full information. *ACM Transactions on Algorithms*, 2010.

31 Jonathan Katz and Chiu-Yuen Koo. On expected constant-round protocols for byzantine agreement. In *Annual International Cryptology Conference*, pages 445–462. Springer, 2006.

32 Valerie King and Jared Saia. Byzantine agreement in polynomial expected time. In *STOC 2103*. ACM, 2013.

33 Eleftherios Kokoris-Kogias, Alexander Spiegelman, Dahlia Malkhi, and Ittai Abraham. Bootstrapping consensus without trusted setup: Fully asynchronous distributed key generation. *IACR Cryptol. ePrint Arch.*, 2019.

34 Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *ACM SIGOPS Operating Systems Review*. ACM, 2007.

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

36 Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, PODC '09, page 312–313, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1582716.1582783.

37 Daniel Lehmann and Michael O Rabin. On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem. In *POPL*. ACM, 1981.

38 Kfir Lev-Ari, Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi. FairLedger: A Fair Blockchain Protocol for Financial Institutions. In Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller, editors, *23rd International Conference on Principles of Distributed Systems (OPODIS 2019)*, volume 153 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:17, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.OPODIS.2019.4.

39 Jinyuan Li and David Maziéres. Beyond one-third faulty replicas in byzantine fault tolerant systems. In *NSDI*, 2007.

40 Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolić. {XFT}: Practical fault tolerance beyond crashes. In *OSDI 2016*, 2016.

41 J-P Martin and Lorenzo Alvisi. Fast byzantine consensus. *DSN 2006*, 2006.

42 Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *CCS2 2016*. ACM, 2016.

43 Achour Mostéfaoui and Michel Raynal. Signature-free asynchronous byzantine systems: from multivalued to binary consensus with $t < n/3$, $o(n^2)$ messages, and constant time. *Acta Informatica*, 2017.

44 Brian M Oki and Barbara H Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *PODC*. ACM, 1988.

45 Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 305–319, 2014.

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

47 Michael O Rabin. Randomized byzantine generals. In *SFCS 1983*. IEEE, 1983.

48 Nicola Santoro and Peter Widmayer. Time is not a healer. In *STACS 1989*. Springer, 1989.

49 Alexander Spiegelman and Arik Rinberg. Ace: Abstract consensus encapsulation for liveness boosting of state machine replication, 2019. arXiv:1911.10486.

50 Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating agreement from execution for byzantine fault tolerant services. *SIGOPS Oper. Syst. Rev.*, 37(5):253–267, 2003. doi:10.1145/1165389.945470.

51 Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *PODC 2019*. ACM, 2019.