OneShot: View-Adapting Streamlined BFT Protocols with Trusted Execution Environments

Jérémie Decouchant
TU Delft
Delft, The Netherlands
j.decouchant@tudelft.nl

David Kozhaya
ABB Research
Zurich, Switzerland
david.kozhaya@ch.abb.com

Vincent Rahli
University of Birmingham
Birmingham, UK
V.Rahli@bham.ac.uk

Jiangshan Yu
University of Sydney
Sydney, Australia
J.Yu.Research@gmail.com

***Abstract*—Byzantine fault-tolerance is arguably an expensive characteristic for protocols to support, especially when considering its overhead on message complexity, number of communication phases, and number of nodes for resilience. Various works in the literature have addressed optimizing one or more of these dimensions through the use of algorithmic optimizations, trusted execution environments, and streamlined view changes.**

The best achievable message complexity, resilience, and latency to date are respectively linear complexity, $\lfloor (N-1)/2 \rfloor$, and two communication phases attained by Damysus (EuroSys'22), a streamlined hybrid protocol. This paper strictly advances the aforementioned state of the art results by introducing OneShot, a streamlined hybrid protocol that uses one communication phase in the normal case, and one or two phases otherwise. OneShot exploits the information nodes receive about the system to dynamically modify and adapt views. We prove that OneShot is safe and live, and moreover demonstrate through experimental evaluation that it improves throughput and latency by respectively up to 150% and 59% compared to the state of the art.

***Index Terms*—Fault tolerance, Consensus, Trusted component.**

I. Introduction

Consensus protocols are key building blocks of distributed systems, enabling a multitude of nodes to jointly reach agreement on decisions. In particular, Byzantine fault-tolerant (BFT) consensus guarantees that such agreement is reached even when a subset of the nodes is subject to arbitrary (a.k.a. Byzantine) faults. As a direct application of BFT consensus, Byzantine fault-tolerant state-machine replication (BFT-SMR) protocols [1] can be implemented to realize resilient distributed services and ecosystems such as blockchains.

PBFT [2] introduced the first practical BFT-SMR protocol, whose safety and liveness are guaranteed in partially synchronous networks [3]. Traditional BFT protocols, including PBFT and variants that build on top of it, optimize performance of the normal case (i.e., fault-free) execution of their algorithm, i.e., when the leader (the node leading the consensus votes) is correct and the network is synchronous for sufficiently long enough. In all other cases, a more expensive view-change protocol is needed to elect a new leader [4, 5]. Once a new leader is elected, the protocol returns to the normal case operation. So-called *streamlined* protocols have recently been designed to avoid these expensive view-changes [6, 7, 8, 9, 10, 11]. To do so, these protocols follow a unified propose-vote paradigm, and integrate view-changes in the normal case operation by rotating the leader [10]. The message complexity of streamlined protocols is linear in the number of nodes as opposed to the quadratic complexity of traditional BFT protocols. This however comes at the expense of an additional communication phase. Nevertheless, both traditional as well as streamlined protocols remain expensive in practice as they require $N = 3f+1$ nodes to tolerate f faulty nodes. Moreover, the former protocols do not scale well due to their message complexity while the latter suffer from increased latency.

In the literature, two prominent lines of work have addressed these limitations and aimed at improving the suitability of either traditional or streamlined protocols for practical uses. These efforts reduced the latency overhead and/or increased the resilience (i.e., less correct nodes needed to tolerate the same number of faulty nodes) of such protocols. The first line of work leverages trusted execution environments (TEEs), such as TPMs [12] or Intel SGX enclaves [13], to reduce latency as well as increase resilience [14, 15]. The second line explores optimizations within the design of the consensus algorithm itself to reduce latency (i.e., the number of communication phases), but cannot improve resilience [16, 17, 18, 19, 20]. The best known resilience and latency bounds to date for Byzantine consensus with linear message complexity are achieved by Damysus [15], a TEE-assisted streamlined protocol that requires $N \geq 2f+1$[1] nodes to tolerate f faults and uses two communication phases.

Despite these non-trivial advances in the literature, an obvious question remains: *can the state of the art be strictly improved along the resilience, latency (number of communication phases) and communication complexity dimensions?*

In this paper we demonstrate that we can indeed improve the state of the art, precisely by decreasing the number of communication phases without affecting neither message complexity nor resilience. Surprisingly, as shown in Sec. VI-A, this improvement comes at no additional cost regarding the

This work was partially supported by the EU Horizon Europe Research and Innovation Programme under Grant No. 101073920 (TENSOR); the Engineering and Physical Sciences Research Council (EPSRC) grant number EP/W034514/1; and by the Australian Research Council (ARC) under project DE210100019. For the purpose of open access, the author(s) has applied a Creative Commons Attribution (CC BY) license to any Accepted Manuscript version arising.

[1]This bound does not differentiate between passive and active replicas such as [21] where $f+1$ active and f passive replicas are needed.

needed TEE functionalities but arguably requires even less.

We introduce OneShot, a streamlined Byzantine consensus protocol that uses TEEs and that requires one communication phase for nodes to reach an agreement in the normal case, and one or two communication phases otherwise. When it requires two phases, together with the effort of the failed (incomplete) previous view, OneShot decides two blocks in total, making it more efficient even in this case. OneShot maintains linear message complexity and uses $N = 2f+1$ nodes out of which f can be Byzantine, herein improving over the state of the art. The main idea behind OneShot hinges on allowing nodes to dynamically adapt their behaviors by fully leveraging the knowledge available to them as well as the information they receive from others. We develop a C++ implementation of OneShot and demonstrate its superior performance compared to both HotStuff [6] and Damysus. Namely, compared to Damysus, which exhibits higher performance than HotStuff, OneShot improves average throughput and latency by up to 150% and 59%, respectively.

As a summary, we make the following main contributions:

• We describe OneShot, a novel streamlined hybrid consensus protocol with linear message complexity. OneShot tolerates a minority of Byzantine faults, and reaches a decision in only one core communication phase in the normal case.

• We provide a proof of correctness that demonstrates that OneShot ensures both safety and liveness.

• We report the results of our performance evaluation to showcase the superior performance of OneShot compared to the previous state of the art algorithms.

The rest of this paper is organized as follows. Sec. II surveys the related work. Sec. III recalls HotStuff and Damysus. Sec. IV describes our system model. Sec. V presents an overview of OneShot. Sec. VI describes OneShot in details, and Sec. VII proves its correctness. Sec. VIII details our performance evaluation. Finally, Sec. IX concludes this paper.

II. Related Work

Numerous previous efforts built hybrid BFT protocols (using TEEs) that generally target improving over various aspects of traditional BFT algorithms such as reconfiguration [22, 23, 24], proactive recovery [25], and fault-tolerance and/or performance [14, 21, 26, 27, 28, 29, 30, 31, 32, 33, 34].

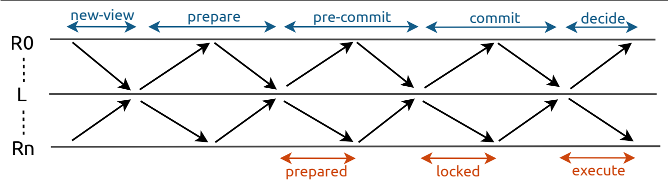
Hybrid Streamlined protocols. In the context of streamlined BFT protocols, hybrid solutions were first mentioned for LibraBFT [35], a protocol built on HotStuff [6], to possibly reduce its attack surface. HotStuff-M and VABA-M [36] proposed to use trusted logs to improve fault tolerance without worsening the communication complexity of the underlying BFT protocol: one trusted log is used for each protocol phase with an additional log for tracking views. The protocol relies on maintaining expander graphs to diffuse messages. Such graphs introduce more network traffic, extra storage overhead and their impact on throughput/latency remains open. A related approach consists in transforming crash fault tolerant protocols to BFT ones using trusted components, such as Madsen et al.'s work [37], which essentially works by running all state machines inside a trusted environment, leading to large enclaves; and Clement and al.'s work [38], which relies on smaller trusted components, that however need to sign the entire history of sent and received messages, leading to large messages. Damysus [15] is the state of the art hybrid streamlined BFT protocol. It leverages two small trusted components, that allow the use of $N = 2f+1$ nodes and 2 communication phases. OneShot improves over Damysus by reducing further the number of communication phases.

Two-phase BFT protocols. Ditto [39] is a combination of HotStuff and PBFT. Its overall behavior in the synchronous (happy) path is similar to HotStuff but relies on a lighter 2-chain commit rule instead of 3. However, the fallback mechanism in case of asynchrony is a heavier quadratic scheme compared to that of HotStuff and OneShot but matches HotStuff's pacemaker complexity. However, Ditto guarantees liveness even in periods of asynchrony: during fallback every replica acts as leader and continues to build a certified chain, one chain is then chosen at random when $2f+1$ certified chains are completed as the chain to build on. Fast-HotStuff [18] is another protocol in this line of work also with a view change of quadratic communication overhead. Abspeol et al. [19] propose a protocol with $O(N \log N)$ communication overhead that relies on zero-knowledge proofs. Wendy [20] is a streamlined protocol that relies on aggregate signatures to use two communication phases in the steady state and three in its view change. Compared to these works, OneShot uses lightweight cryptographic schemes, uses less communication phases and maintains a linear communication complexity.

Marlin [16] proposes a BFT protocol with strictly linear communication complexity, having two phases for normal case operations and two or three phases for view changes. They achieve this reduction in required number of phases in the normal operation by having the leader propose two blocks. A first block that extends the block with the highest quorum certificate that the leader received and a second virtual block that extends a block (that may or may not exist) from a "virtual" safe snapshot. Each replica can either vote for one or two blocks depending on the quorum certificate it is locked on. HotStuff-2 [17] is a two-phase variant of HotStuff that achieves the same properties, i.e., solves partially-synchronous BFT, achieves $O(n^2)$ worst-case communication and best case linear communication, and optimistic responsiveness. The authors do so by devising a solution that allows the leader of a view to know about the highest locked block and be able to convince all honest parties about it. This can be achieved, only after GST (the Global Stabilization Time, when no network delays occur), and assuming a current honest leader of a view and some consecutive previous honest leaders. Compared to OneShot, both protocols [16, 17] require (1) $N = 3f+1$ nodes to tolerate f faulty nodes (OneShot requires $2f+1$) and (2) one additional phase in both good executions (2 phases in comparison with 1 phase for OneShot) and bad ones (3 phases in comparison with 2 phases for OneShot).

Rollback attacks on hybrid protocols. Some research efforts have shed light on the fallacies related to the assumption

Fig. 1 Communication phases in HotStuff



that TEEs never crash and lose their internal state. ROTE [40] allows the TEEs (a.k.a. enclaves) of a hybrid protocol to reliably store enclave-specific counters. These enclaves use a consistent broadcast protocol to propagate their state updates and a recovery mechanism that obtains lost counters from other enclaves upon restart. ENGRAFT [41] proposes a distributed resilient in-memory key-value storage inside SGX enclaves for storing Raft meta data, leveraging network memory [42]. NARRATOR [43] provides protection against rollback attacks for clients of cloud TEEs. To do so, it relies on infrequent interactions with a blockchain that initializes a distributed system of TEEs for an application.

Gupta et al. identified three issues that hybrid protocols that rely on $2f{+}1$ nodes might face [44]: restrictive responsiveness, rollback attacks, and lack of parallelism. They show how increasing the number of replicas back to $3f{+}1$ and allowing the leader to maintain multiple trusted counters in PBFT-like protocols address these issues. The first two issues can respectively be addressed in OneShot by transferring certificates to clients, and using known defenses against rollback attacks [40, 43]. The last issue, parallelism, can for example be addressed using parallel executions [28, 45, 46].

III. HOTSTUFF AND DAMYSUS IN A NUTSHELL

HotStuff [6] is a streamlined BFT consensus protocol that adopts the lock-commit paradigm and has a communication complexity linear in the number of nodes. It requires $N{\geq}3f{+}1$ nodes to tolerate f Byzantine faults. Nodes build a chain of blocks by voting for extensions proposed by the leaders of *views* (successive rounds). HotStuff comes in two versions: (1) *Basic* HotStuff, where nodes vote on a single block per view; and (2) *chained* (or *pipelined*) HotStuff, where a view allows several blocks to simultaneous progress towards being committed. To execute a block, HotStuff employs several *phases* per view v, as depicted in Fig. 1, where time flows from left to right, and L is the leader of the view v among nodes $R_0, \ldots, R_n$. Chained HotStuff is referred to as a 3-phase protocol, as it has 3 *core* phases to agree on blocks: a prepare phase to propose blocks; a pre-commit phase to let nodes certify that a block has been prepared; and a commit phase to "lock" the prepared block to guarantee liveness [6, Sec.4.4]. Those core phases are complemented by 2 half-phases[2] in Basic HotStuff: a new-view half-phase to submit the latest prepared blocks, and a decide half-phase to execute blocks once it is safe to do so. A Basic HotStuff view is then composed of the 8 communication steps depicted in Fig. 1.

[2]As a full phase contains two communication steps, we use a half-phase to represent a phase that only contains one communication step.

Damysus [15] is the state of the art hybrid streamlined BFT protocol and is built upon Hotstuff [6]. By leveraging two small trusted components, namely the CHECKER and the ACCUMULATOR, Damysus simultaneously tolerates a minority of Byzantine faults thanks to the CHECKER and requires a smaller number of core phases thanks to the ACCUMULATOR: from 3 in Hotstuff down to 2. A Damysus view is then composed of 6 communication steps. Like Chained-HotStuff, Chained-Damysus supports pipelined operations for improved performance. For ease of presentation, this section focuses on the basic version of Damysus, as it can be easily transformed to obtain a pipelined version [15].

Trusted components. The CHECKER component is run by every replica. It provides a monotonically increasing counter to keep track of the current view and phase, and stores a view number and hash value pair for the last prepared block. The pair is included in the commitment that a non-leader replica (a.k.a. a backup) sends to the next leader in the new-view phase. The monotonically increasing counter prevents Byzantine replicas from equivocating, e.g., by proposing different blocks for the same view as a leader. The data stored in the ACCUMULATOR guarantees that a Byzantine backup sends the commitment of its latest prepared block to the leader of a new view. This guarantee ensures that at least one of the $f{+}1$ commitments received by a leader in the new-view phase (presented below) represents the highest view where a block is prepared, which is necessary for safety. The ACCUMULATOR component is only run by the leader of a view. Loosely speaking, it takes $f{+}1$ commitments as inputs, and outputs a signed message that contains a view number and hash value pair possessing the highest view among the inputs.

Communication phases. Damysus has two core communication phases, prepare and pre-commit, and uses two additional ½-phases: the new-view ½-phase to rotate the leader, and the decide ½-phase to execute blocks. This results in a total of 6 communication steps compared to 8 in HotStuff.

(1) In the **new-view** phase, each backup increments its view and sends its commitment, which contains its (view, hash) pair stored in the CHECKER, to the leader.

(2) In the **prepare** phase, the leader generates the latest prepared block from $f{+}1$ received new-view messages using the ACCUMULATOR. The leader then extends the latest prepared block and certifies it using its CHECKER. Upon receiving this signed block from the leader, each backup responds with a vote generated by the CHECKER. In Damysus, a checker's counter is incremented each time a checker is called.

(3) In the **pre-commit** phase, the leader collects $f{+}1$ prepare votes from backups, and broadcasts a combined version to backups. Backups consider the proposed block as prepared, store the view number and hash value associated to the block via the CHECKER, and replies with a vote it generates.

(4) In the **decide** phase, the leader collects $f{+}1$ commit votes from backups, and broadcasts a combined version to backups so that they can execute the block.

In Damysus every node verifies the authenticity of the messages it receives, which are signed by trusted components, before processing them.

IV. System Model

In this section, we describe OneShot's system model, which is identical to the one of Damysus [15], the hybrid BFT consensus protocol it improves upon.

Replicas and leader. A system has a static membership consisting of N replicas out of which at most f can be faulty (i.e., Byzantine). In the case of HotStuff $N = 3f+1$, while for Damysus and OneShot $N = 2f+1$. We refer to a replica using a unique id. We assume that each view has a unique leader, which is chosen deterministically and known to all nodes.

Trusted components. Each replica executes trusted components that provide the CHECKER and ACCUMULATOR services, resulting in a *hybrid* fault model, where at each faulty node all components can be tampered with except the ones providing these trusted services.

Communications. We assume that replicas communicate by exchanging messages over a fully connected communication network. Communications are reliable (i.e., messages are not lost). We adopt the partial synchrony model, where there is a known bound Δ and an unknown Global Stabilization Time (GST), such that after GST, all messages arrive within Δ after their emission [3].

Signatures. Replicas and trusted components rely on an asymmetric signature scheme. A digital signature σ is generated via the SIGN function, and verified using the VERIFY function provided by the scheme. The identity of the signer of a given signature σ is written as $\sigma\texttt{.id}$. We write m_σ for the message m signed with signature σ generated using SIGN. We also write $m_{\vec{\sigma}}$ when m is signed by the list of signatures $\vec{\sigma}$. Signatures in OneShot are generated by trusted components in Fig. 5c using their private keys. We assume that public keys are known by trusted components, replicas, and clients.

Blocks. A block b contains transactions submitted by clients. OneShot works at the block level, and we therefore leave abstract the internal details of transactions, which are mostly application-specific. We assume that a cryptographic secure hash function $\mathbf{H}$ is used to hash blocks. We write h for the hash value of a block. A block b contains the hash value of another block b' on top of which b is built. We write $b \succ h$ when b is a direct extension of a block b' with hash value h, i.e., b is built on top of b'. We also write $b_1 \succ b_2$ for $b_1 \succ \mathbf{H}(b_2)$. The relation $\succ$ can easily be checked, for example if blocks store the hash values of the blocks they extend. We write $\succ^+$ for its transitive closure. We say that a block b_1 is in *conflict with* a different block b_2 if $\neg b_2 \succ^+ b_1$ and $\neg b_1 \succ^+ b_2$. We also assume a `createLeaf` function that creates a new block extending a parent block (or simply its hash value) with client transactions.

V. Overview of OneShot

Unlike state of the art (hybrid) streamlined BFT protocols that typically have a generic behavior throughout all protocol executions, OneShot capitalizes on the fact that not all executions are identical, and that agreement can be reached in some executions within a fewer number of communication steps. To this end, OneShot operates in an execution-tailored fashion using all possible information available and succeeds in reaching agreement in fewer communication steps compared to all its predecessors in the literature.

Fig. 2 Example of a normal execution

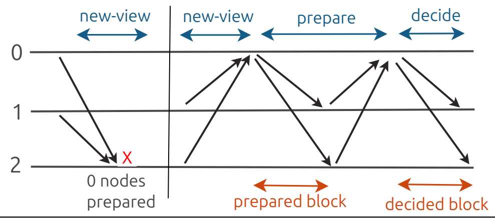


Fig. 3 Example of a catch-up execution

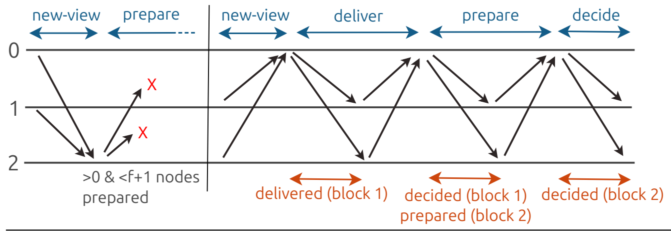
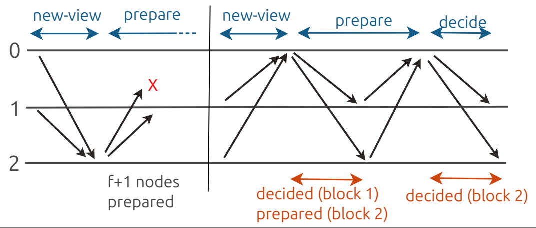


Fig. 4 Example of a piggyback execution



More specifically, OneShot features three types of executions where agreement can be reached on some block(s). The distinction amongst those three executions is based on the information available to the non-faulty leader of the view where agreement is reached:

- A **normal execution** is one where the non-faulty leader of a decisive view – a view where agreement is reached on some block(s) – does not know of any previous prepared block that has not been decided. A prepared block is a block that has been received from the leader, and a decided block is one that has been adopted by at least $f+1$ distinct nodes. As the example depicted in Fig. 2 shows, a normal execution results in agreement on 1 block within 4 communication steps. Fig. 2 depicts two successive views, where node 2 is the leader of the first view that is not decisive, and node 0 is the leader of the second view that is decisive.
- A **catch-up execution** is one where the non-faulty leader of a decisive view knows that a previous valid prepared block, say block b, has been prepared by $i : 1 \leq i < f+1$

nodes. A valid prepared block is one that builds on the latest decided block in the chain. In this case, the non-faulty leader of the decisive view initiates a catch-up mechanism to prepare block b by at least $f+1$ nodes before it proposes a new block, which extends b. As Fig. 3 illustrates, this results in a decisive view where agreement is reached on 2 blocks in a total of 8 communication steps. The total number of communication steps is counted from the instant when block b is proposed (in the previous view) until an agreement is reached on both blocks.
- A **piggyback execution** is one where the non-faulty leader of a decisive view knows that a previous valid prepared block, say block b, has been prepared by at least $f+1$ other nodes. In this case, the non-faulty leader of the decisive view piggybacks block b onto its new proposed block, which extends b. As Fig. 4 illustrates, this results in a decisive view where agreement is reached on 2 blocks in a total number of 6 communication steps.

The table below summarizes the three executions differentiated by OneShot, stating for each execution the number of communication steps needed to reach agreement as well as the number of blocks on which agreement is reached.

	#blocks	#total steps
Normal execution	1	4
Catch-up execution	2	8
Piggyback execution	2	6

VI. OneShot Details

We now provide a more detailed description of OneShot. The full pseudocode is presented in Fig. 5. We begin by describing the TEEs that OneShot relies on in Sec. VI-A, and introduce afterwards the different types of certificates used in OneShot in Sec VI-B. We then delve into the protocol's logic in Sec. VI-C and its optimizations in Sec. VI-F. We finally provide the intuitions behind OneShot's correctness in Sec. VII.

A. Trusted Services

Like Damysus, OneShot uses two similar trusted services, called CHECKER and ACCUMULATOR. The former enhances Byzantine resilience while the latter reduces the latency compared to traditional BFT protocols. However, unlike Damysus:

- CHECKER in OneShot stores a proposal instead of a prepared block, requires less secure memory, and eliminates one core function given that OneShot operates with one less core phase.
- ACCUMULATOR is not called as often (e.g., it is never called in normal executions) and can provide more information.

In particular, OneShot's trusted services store less data, and implement less functions than Damysus's. These simplifications lead to a reduced trusted computing base, and inherently to more secure components.

We elaborate further on CHECKER and ACCUMULATOR in OneShot in what follows.

a) CHECKER*:* The CHECKER service offers three functions. The first, `TEEprepare`, is used by leaders to propose new blocks, and guarantees that this can be done at most once per view. It generates proposals as described in Def. 1. The second, `TEEvote`, is used to vote for a block. The third, `TEEstore`, is used to "store" the latest proposed block. In fact, only the view at which the proposal is made is stored, as opposed to also storing the proposal's hash value like in Damysus, as at most one block can be proposed per view. As a proof of successfully storing a view in the CHECKER, `TEEstore` generates a store certificate (Def. 2) that connects the stored view with the respective block. `TEEstore` guarantees that a single store certificate can be generated per view. In order to implement the above functions, CHECKER maintains a state composed of three components:

- a confidential private key, along with the public keys of the other components.
- a (view, phase) pair that evolves as follows. Leaders call this service twice per view, first increasing the phase (from $(view, ph_0)$ to $(view, ph_1)$) when making a proposal, and then increasing the view (from $(view, ph_1)$ to $(view{+}1, ph_0)$) when storing the proposal (those two calls can be combined into one for efficiency). Backups do not make proposals and only call this service once per view increasing only the view (from $(view, ph_0)$ to $(view{+}1, ph_0)$).
- the view number of the latest prepared block.

b) ACCUMULATOR*:* The ACCUMULATOR service has a single function called `TEEaccum`. It takes a list of $f+1$ new-view certificates (see Def. 6) and asserts – by generating an accumulator (see Def. 5) – that the first element in that list is for the block with the highest proposed view (i.e., the view at which the block was prepared). This function is used by a leader as a proof that its message relays the latest proposal among the received new-view messages. This is needed for example when the leader knows about an old proposal and wants to push it forward to commitment. The ACCUMULATOR is only called by leaders in the prepare phase of catch-up executions to convince backups that they are pushing forward the proposal with the highest view, while in Damysus it is called by leaders in the prepare phase of all executions, as Damysus does not differentiate between different kinds of executions.

B. Certificates

During the various phases of the protocol, nodes create certificates as proofs that they have performed certain actions using their trusted services. This section reviews the various certificates used in Fig. 5. In that figure, we use $\vec{\phi}$ for a list of certificates, and $\vec{\phi}^n$ to indicate that the list has length n. We use a similar notation for signatures and node ids.

a) Proposals: Proposals are sent by leaders in the prepare phase (l. 8, Fig. 5a) to propose new blocks. Proposals are certified using the `TEEprepare` function, which guarantees that leaders can only make one proposal per view by incrementing the phase number stored in their trusted components.

Fig. 5 OneShot's pseudocode

(a) Non-trusted prepare, new-view, decide phases

```
1:  $view = 0$ // current view (duplicates the TEE's)
2:  $pks$ // public keys (duplicate the TEE's)
3:  $prop = \langle b, \phi_p, \phi_{qc} \rangle$ // latest prop. from a leader
4:
5:  function propose($h, \phi$)
6:      $b :=$ createLeaf($h$, client transactions)
7:      $\phi_p :=$ TEEprepare($\mathbf{H}(b)$)
8:      send $\langle b, \phi_p, \phi \rangle$ to all // skip the deliver phase
9:
10: // PREPARE PHASE
11: as a leader // NORMAL EXECUTION
12:     wait for $\phi_c$ of the form $\mathsf{prep}(view{-}1, h, v)_{\vec{\sigma}^{f+1}}$
13:     propose($h, \phi_c$)
14:
15: as a leader
16:     wait for $\vec{\phi}_n$ s.t. NV-match($\vec{\phi}, f{+}1, view$)
17:     if all $\phi_s \in \vec{\phi}_n$ are of the form $\mathsf{store}(view{-}1, h, v)\_$ then
18:         // PIGGYBACK EXECUTION
19:         $\vec{\sigma} :=$ the signatures of the $f{+}1$ store certs. in $\vec{\phi}_n$
20:         propose($h, \phi_c$) where $\phi_c := \mathsf{prep}(view{-}1, h, v)_{\vec{\sigma}}$
21:     else
22:         $\phi_0 :=$ certificate in $\vec{\phi}_n$ with highest view
23:         $acc :=$ TEEaccum($\phi_0, \vec{\phi}_n \setminus \{\phi_0\}$)
24:         if $acc$.B then
25:             propose($acc$.hash$, acc$)
26:         else // CATCH-UP EXECUTION
27:             send $\langle acc, \phi_0 \rangle$ to all // start deliver phase
28:
29: all replicas
30:     wait for $\langle b, \phi_p, \phi_{qc} \rangle$ from the leader, where $\phi_{qc}$ is for $\langle view, h \rangle$
31:     $prop := \langle b, \phi_p, \phi_{qc} \rangle$ // update $prop$
32:     abort if $\neg(\phi_p$.view $== view \wedge b \succ h \wedge \mathbf{H}(b) == \phi_p$.hash$)$
33:     send $\phi_s :=$ TEEstore($\phi_p$) to leader
34:
35: // DECIDE ½-PHASE
36: as a leader
37:     wait for $\vec{\phi}_s$, $f{+}1$ $\phi_s$ of the form $\mathsf{store}(view, h, view)_\sigma$
38:     $\vec{\sigma} :=$ the signatures of the $f{+}1$ store certs. in $\vec{\phi}_s$
39:     send $\phi_c := \mathsf{prep}(view, h, view)_{\vec{\sigma}}$ to all replicas
40:
41: all replicas
42:     wait for $\phi_c$ of the form $\mathsf{prep}(view, h, view)_{\vec{\sigma}^{f+1}}$ from leader
43:     abort if $\neg$VERIFY$(\phi_c)_{pks}$
44:     execute $b$ corresponding to $h$ & reply to clients
45:     $prop := \langle b, \phi_p, \phi_c \rangle$; $view$++ // update $prop$
46:     send $\phi_n := \phi_c$ to $view$'s leader // sending a new-view
47: // NEW-VIEW ½-PHASE
48: all replicas
49:     when timeout
50:     $\langle b, \phi_p, \phi_{qc} \rangle := prop$; $view$++
51:     $\phi_s :=$ TEEstore($\phi_p$) // if not already executed
52:     send $\phi_n := \mathsf{nv}(b, \phi_s, \phi_{qc})$ to $view$'s leader
```

(b) Non-trusted deliver phase of catch-up executions

```
1:  all replicas
2:      wait for $\langle acc, \phi_n \rangle$ where $\phi_n$ is $\langle b_1, \phi_s, \phi_{qc} \rangle$
3:          and $\phi_s$ is $\mathsf{store}(view{-}1, \mathbf{H}(b_1), v)_{\sigma_1}$ // $h_1$ was stored at $v$
4:          and $\phi_{qc}$ is for $\langle v, h_2 \rangle$ // $h_2$ was selected at $v$
5:      abort if $\neg(acc$ is valid $\wedge$ VERIFY$(\phi_n)_{pks} \wedge b_1 \succ h_2)$
6:      send $\phi_v :=$ TEEvote($\mathbf{H}(b_1)$) to leader
7:
8:  as a leader
9:      wait for $\vec{\phi}_v$, $f{+}1$ votes of the form $\mathsf{vote}(h, view)\_$
10:     $\vec{\sigma} :=$ the signatures of the $f{+}1$ votes in $\vec{\phi}_v$
11:     propose($h, \phi_{vc}$) where $\phi_{vc} := \mathsf{vc}(h, view)_{\vec{\sigma}}$ // resume (l. 29)
```

(c) TEE code

```
1:  $view = 0, phase = ph_0$ // current view/phase
2:  $sk, pks$ // 1 private (confidential) & public keys
3:  $prepv = 0$ // view of latest proposed block
4:
5:  function TEEprepare($h$)
6:      if $phase == ph_0$ then
7:          $phase := ph_1$
8:          return $\phi_p := \mathsf{prop}(h, view)_\sigma$
9:
10: function TEEstore($\phi_p$) where $\phi_p$ is $\mathsf{prop}(h, v)_\sigma$
11:     if VERIFY$(\phi_p)_{pks} \wedge \phi_p$ is from the leader $\wedge\ view \geq v \geq prepv$ then
12:         $prepv := v$;  $view$++;  $phase := ph_0$
13:         return $\phi_s := \mathsf{store}(view{-}1, h, v)_{\sigma'}$
14:
15: function TEEaccum($\phi_n, \vec{\phi}_n$) where $\phi_n$ is for $\langle v, h, v' \rangle$
16:     if $\left( \begin{array}{l} \text{VERIFY}(\phi_n, \vec{\phi}_n)_{pks} \\ \wedge \forall \phi'_n \in \vec{\phi}_n . \phi'_n \text{ is for } \langle v, h', v'' \rangle \wedge v' \geq v'' \end{array} \right)$ then
17:         $\vec{id} :=$ the ids of the nodes that signed $(\phi_n, \vec{\phi}_n)$
18:         $B :=$ certifies($h, \phi_n$)
19:         return $acc := \mathsf{acc}(B, v, h, \vec{id})_{\sigma'}$
20:
21: function TEEvote($h$)
22:     return $\phi_v := \mathsf{vote}(h, view)_\sigma$
```

Def. 1 (Proposals)**.** *A proposal ϕ_p for a block with hash value h is of the form $\mathsf{prop}(h, v)_\sigma$, where v is a view number and σ is the leader's signature. We also note ϕ_p.view the view v at which proposal ϕ_p is made.*

b) Store certificates: Store certificates produced by TEEstore certify the receipt of proposals. They are generated during the prepare phase (l. 33, Fig. 5a) by nodes once they have verified that the proposal they have received from the leader is correct, meaning that it extends a certificate from the previous view (checked l. 32, Fig. 5a). Store certificates are also generated when changing view after a timeout (l. 51, Fig. 5a) to create a certificate of the latest proposal received that is tagged with the current view, since the latest proposal might have been received in a previous view and because an old invalid certificate must not be reused by Byzantine nodes.

Def. 2 (Store Certificates)**.** *A store certificate ϕ_s is of the form $\mathsf{store}(v_2, h, v_1)_\sigma$ that certifies that a block with hash value h initially proposed in view v_1 was "stored" in view v_2.*

c) Prepare certificates: Prepare certificates are put together from store certificates by leaders (l. 39, Fig. 5a) to combine them into quorum certificates in the middle of a prepare phase, as well as in the beginning of the prepare phase (l. 20, Fig. 5a). Note that leaders require that the store certificates they use to create a prepare certificate are for the current view (l. 39), thereby requiring that the proposed block was indeed proposed and stored in the current view. A prepare certificate guarantees that at least $f{+}1$ nodes have stored a block to safely execute it. Its reception (l. 42, Fig. 5a) prompts nodes to execute the corresponding block.

Def. 3 (Prepare Certificates)**.** *A prepare certificate ϕ_c of the* *form* $\mathsf{prep}(v_2, h, v_1)_{\vec{\sigma}^{f+1}}$*, combines* $f+1$ *store certificates, and certifies that block* h *proposed in view* v_1 *was certified in view* v_2*. We use* $\phi_c\mathtt{.hash}$ *for the hash value* h *contained in* ϕ_c*.*

d) Votes: To guarantee that proposals are built on top of blocks stored by at least $f+1$ nodes, following an unsuccessful view, leaders start an additional phase of voting to finish an incomplete view. During that phase, using `TEEvote` (l. 21, Fig. 5c), nodes vote for a block from a previous view that did not complete. A vote certificate is then put together by a leader (l. 11, Fig. 5b) by combining these votes, and certifies that a block was received by at least one correct node.

Def. 4 (Votes)**.** *A vote* ϕ_v *for a block with hash value* h *is of the form* $\mathsf{vote}(h, v)_\sigma$*. A vote certificate* ϕ_{vc} *combines* $f+1$ *votes and is of the form* $\mathsf{vc}(h, v)_{\vec{\sigma}^{f+1}}$*.*

e) Accumulators: An accumulator is generated by `TEEaccum` (l. 15, Fig. 5c), which is called by a leader upon receiving new-view messages from which no certificate can be put together (l. 23, Fig. 5a). The leader then selects the new-view certificate with the highest view, and the accumulator generates a certificate that it has indeed the highest view among the provided certificates.

Def. 5 (Accumulators)**.** *An accumulator* acc *is of the form* $\mathsf{acc}(B, v, h, \vec{id})_\sigma$*, where* B *is a Boolean that indicates whether the accumulator certifies a new-view certificate for the block with hash value* h *or a predecessor block (see Def. 6) and where* $\vec{id}$ *is the vector of the* $f+1$ *ids of the nodes that contributed to the accumulator, i.e., that signed the certificates passed as arguments to* `TEEaccum`*. Given such an accumulator, we also use* $acc\mathtt{.hash}$ *for* h*, and* $acc\mathtt{.B}$ *for* B*. We say that the above accumulator* acc *is valid (used l. 5, Fig. 5b) if its signature is correct, and if* $\vec{id}$ *is a vector of* $f+1$ *unique ids.*

Note that the ids contained in an accumulator are used by nodes to pull blocks from others as discussed in Sec. VI-E.

f) Quorum certificates: We note ϕ_{qc} a quorum certificate, which can be: (i) a prepare certificate ϕ_c; or (ii) a vote certificate ϕ_{vc}; or (iii) an accumulator acc. A quorum certificate is always a set of $f+1$ certificates that guarantee that at least a correct node participated in the vote, and therefore holds the corresponding block (correct nodes only vote for blocks they have received). If ϕ_{qc} is a prepare certificate of the form $\mathsf{prep}(v-1, h, v')_{\vec{\sigma}}$ or a vote certificate of the form $\mathsf{vc}(h, v)_{\vec{\sigma}}$ or an accumulator of the form $\mathsf{acc}(true, v-1, h, \vec{id})_\sigma$, we say that ϕ_{qc} *is for* $\langle v, h\rangle$. A prepare or a vote certificate is certified by the $f+1$ nodes that signed $\vec{\sigma}$, and an accumulator is certified by the $f+1$ ids in $\vec{id}$. The uses of either $v-1$ and v in the certificates come from the fact that prepare certificates of the previous view are used to create new proposals, while for vote certificates leaders use the ones generated during the deliver phase of the current view.

g) New-view certificates: New-view certificates are generated by nodes either when a view ends successfully with a block execution (l. 46, Fig. 5a), or abnormally by timing out (l. 52, Fig. 5a). In the former case, the new-view certificate is a prepare certificate that guarantees that $f+1$ nodes have stored the prepared block using their trusted component. In the latter, the new-view certificate contains the latest proposal received by the node.

Def. 6 (New-view Certificates)**.** *A new-view certificate* ϕ_n *is either a prepare certificate* ϕ_c *or of the form* $\mathsf{nv}(b, \phi_s, \phi_{qc})$*, where* $\phi_s = \mathsf{store}(v_2, \boldsymbol{H}(b), v_1)_\sigma$ *and* ϕ_{qc} *is for* $\langle v_1, h'\rangle$*, such that* b *is a stored block that either extends* h' *(when generated in l. 52, Fig. 5a from a proposal received l. 31) or has hash value* h' *(when generated in l. 52 of Fig. 5a from a proposal received l. 45). In the latter case, we say that* ϕ_n *is* certified *by* h'*, and write* $\mathtt{certifies}(h', \phi_n)$ *(which is used l. 18, Fig. 5c). If* ϕ_n *is either* $\mathsf{prep}(v_2, h, v_1)_{\vec{\sigma}}$ *or* $\mathsf{nv}(b, \mathsf{store}(v_2, h, v_1)_\sigma, \phi_{qc})$*, we say that* ϕ_n *is for* $\langle v_2, h, v_1\rangle$*. Let* $\mathtt{NV\text{-}match}(\vec{\phi}, k, v)$ *hold if* $\vec{\phi}$ *is a collection of* k *new-view certificates for* $\langle v, \_, \_\rangle$ *signed by different nodes.*

C. Protocol Logic

As mentioned in Sec. V, OneShot distinguishes between 3 kinds of decisive executions. Upon receiving new-view messages, and depending on their content, leaders either proceed into executing a normal execution (l. 12), a catch-up execution (l. 21), or a piggyback execution (l. 17).

a) Normal execution: When a leader of a view v receives a certificate that consensus was achieved on a block b_1 in an earlier view with enough evidence that no proposal was accepted afterwards, it starts executing a normal phase. In that case, it simply proposes a block b_2 that extends/builds on b_1 (l. 12, Fig. 5a). The proposal is then "*prepared*" by backups (all nodes besides the leader) (ll. 29–33, Fig. 5a), which send a message to the leader. These replies are combined by the leader into a prepare certificate that is sent to the backups (ll. 36–39, Fig. 5a). Upon receiving such a prepare certificate (ll. 41–46, Fig. 5a), backups execute b_2, start a new view, and reply to the clients by forwarding the certificate they have received from the leader. A single message is therefore enough for a client to trust a reply.

b) Catch-up execution: When a leader of view v knows about a valid prepared block b_1 of a previous view but cannot directly generate a quorum certificate for b_1 (e.g., because it only received $i : 1 \leq i < f+1$ store certificates), it starts a catch-up execution first by initiating an additional deliver phase (Fig. 5b) to guarantee that the highest block b_1 they have received is at least stored by one correct node before extending it. A deliver phase is then triggered (l. 27, Fig. 5a), and once completed (l. 11, Fig. 5b), the nodes resume as in a normal execution from the prepare phase (l. 29, Fig. 5a). When prompted by the leader to execute the deliver phase, the nodes vote for a block from an unfinished previous view (ll. 1–6, Fig. 5b). The leader then collects those votes and triggers the prepare phase (ll. 8–11, Fig. 5b).

c) Piggyback execution: When a leader of view v knows about a valid prepared block b_1 of a previous view and can reconstruct a quorum certificate because for example it has received at least $f+1$ store certificates (l. 17, Fig. 5a), the leader of view v starts a piggyback execution by proposing a block b_2 that extends b_1 (hence the leader piggybacks b_1 on its new proposal b_2). After this nodes proceed as in a normal execution, resulting in the execution of blocks b_1 and b_2.

D. Storing and Relaying Proposed Blocks

In the previous section, we discussed how OneShot behaves based on information available to the leader. In this section, we complement this from the backups' perspective showcasing how and what information backups send to leaders in new-view messages. Precisely, replicas always send some information computed from the variable *prop* (l. 3, Fig. 5a) to the leader of a following view within new-view messages. This variable is used to store proposals from current or previous leaders and has the form $\langle b, \phi_p, \phi_{qc}\rangle$. The replicas update *prop*'s value in the following cases, otherwise it remains unchanged.

- In all executions, if a node receives a prepare certificate (i.e., verification that $f{+}1$ nodes stored a block, say b) in the decide phase, then this node records the certificate (l. 45, Fig. 5a) as the latest proposal from a leader in *prop*. In this case, the ϕ_{qc} of *prop* certifies bock b.
- In a normal or piggyback execution, if a node receives a new proposal for block b from the leader, i.e., a proposal generated by `propose` (l. 13, l. 20, or l. 25, Fig. 5a), it stores that proposal in *prop* (l. 31, Fig. 5a) where ϕ_{qc} now is a certificate of the block that b extends (which could be either a prepare certificate or an accumulator). If a decision is reached in this view, *prop* is updated as indicated in the first bullet, otherwise its value remains unchanged.[3]
- In a catch-up execution, if a node receives a vote certificate for a block b during the prepare phase, that follows a deliver phase (sent l. 11, Fig. 5b and received l. 30, Fig. 5a), it stores the certificate in *prop* (l. 31, Fig. 5a). In this case, ϕ_{qc} is a vote certificate (and not a prepare certificate or an accumulator).

E. Pulling Blocks

When a node receives a proposal $\langle b, \phi_p, \phi_{qc}\rangle$ from the leader (l. 29, Fig. 5a), it might happen that it has not received and executed the block b_0 that ϕ_{qc} certifies, and which b extends. In that case it will start voting for b, and at the same time it will pull b_0 so that it can be executed. This pulling mechanism, presented in Fig. 6, works as follows. If ϕ_{qc} is of the form $\mathsf{prep}(v', h, v)_{\vec{\sigma}^{f+1}}$, then the node checks whether it has already executed the block b_0 proposed at view v (l. 5), with hash value h. If it has not, it then checks whether it has received b_0 (l. 6), and if not, it needs to pull it from one of the nodes that signed the certificate, i.e., the nodes that signed $\vec{\sigma}^{f+1}$, one of which is correct. This is done by piggybacking a pull request to one of its messages when one of those nodes is the leader (l. 11). Leaders also piggyback blocks to their messages to the requester after receiving such a request (l. 16). To achieve this, the following needs to be added to Fig. 5: `pull` needs to be called l. 31 of Fig. 5a; and the variable `blocks`, a set of

Fig. 6 Block pulling subprotocol

```
 1: pulling = ∅      blocks = ∅
 2:
 3: function pull($\phi_{qc}$) where $\phi_{qc}$ is for $\langle v, h\rangle$
 4:     $\vec{id}$ := the ids of the $f{+}1$ nodes that certified $\phi_{qc}$
 5:     if a block with hash value $h$ has not been executed then
 6:         if a block with hash value $h$ has not been received then
 7:             add $\langle v, h, \vec{id}\rangle$ to pulling
 8:
 9: all replicas
10:     when $\langle v, h, \vec{id}\rangle \in$ pulling $\wedge\ i \in \vec{id}$ is the current leader
11:     piggyback the pull request $\langle v, h\rangle$ to the next message to $i$
12:
13: as a leader
14:     wait for a pull request of the form $\langle v, h\rangle$ from $i$
15:     $b$ is such that $\langle v, h, b\rangle \in$ blocks
16:     piggyback the pull reply $\langle v, b\rangle$ to the next message to $i$
17:
18: all replicas
19:     wait for a pull reply of the form $\langle v, b\rangle$
20:     remove $\langle v, \mathbf{H}(b)\rangle$ from pulling
```

triples of the form $\langle v, h, b\rangle$ such that $h = \mathbf{H}(b)$, used by nodes to store the blocks they have received, needs to be updated l. 31 and l. 45 of Fig. 5a, and l. 5 of Fig. 5b. To prevent potential denial-of-service attacks by Byzantine nodes, nodes answer at most once to a pulling request issued by another node, and only if they have not already sent it the requested block.

F. Optimizations

In addition to the code described above, OneShot features the following performance optimizations.

a) Avoiding Re-Votes: When a view ends normally, a node has recorded a *prop* (see the variable l. 3, Fig. 5a) of the form $\langle b, \phi_p, \phi_c\rangle$, where ϕ_p is of the form $\mathsf{prop}(h, v)_\sigma$ and ϕ_c of the form $\mathsf{prep}(v, h, v)_{\vec{\sigma}^{f+1}}$, with $h = \mathbf{H}(b)$, i.e., where ϕ_c is a prepare certificate of the proposal ϕ_p itself, and not of the block extended by ϕ_p, as discussed in Sec. VI-D. If the next view ends without any new proposal being made, then the node times-out and generates a new-view certificate (l. 52, Fig. 5a) using the above ϕ_p and ϕ_c. Without the check l. 24 of Fig. 5a, the deliver phase would be triggered and would lead to a block already stored by $f{+}1$ nodes to be voted upon again. To avoid this situation, the accumulator checks l. 18 of Fig. 5c whether the new-view certificate is certified by itself (see Def. 6), meaning that it is of the form $\langle b, \phi_s, \phi_{qc}\rangle$ and the hash value in ϕ_s is the same as the hash value that ϕ_{qc} is for, and tags the generated accumulator with a Boolean indicating whether that is the case or not.

b) Avoiding sending large blocks: When a backup sends a new-view certificate to the leader l. 52 of Fig. 5a, it sends the block of the latest proposal it knows. In case the backup already has a signature certifying that the leader has already received this block, the backup can omit re-sending it. This can for example happen when the backup has received a prepare certificate l. 42 of Fig. 5a of the form $\mathsf{prep}(v, h, v)_{\vec{\sigma}}$, where the new leader is one of the nodes that has signed $\vec{\sigma}$. In that case, the backup knows that the new leader has received b, and instead of sending $\langle b, \phi_s, \phi_{qc}\rangle$, can simply send $\langle \phi_s, \phi_{qc}\rangle$.

[3]In this case, the leader of the following view then can either trigger a catch-up execution with a deliver phase to vote upon b, or a piggy-back execution if the next leader receive this proposal from at least $f{+}1$ nodes.

c) Preempting catch-up executions: When a leader starts the deliver phase in a catch-up execution, it also keeps on waiting for a prepare certificate from the previous view. If it receives such a certificate before it has started the prepare phase, it will then stop handling votes from the deliver phase, and instead directly start a normal execution, thereby preempting the catch-up execution.

VII. Proof of Correctness

We now prove OneShot's safety and liveness under partial synchrony using a similar proof style as in [6].

Lemma 1. *OneShot is safe, i.e., correct nodes do not execute conflicting blocks.*

Proof. OneShot constrains the behavior of leaders using several mechanisms. First, in order to propose a block, a leader has to use the `TEEprepare` function to generate a certificate $\mathsf{prop}(h, v)_\sigma$. This function can be called only once per view: a certificate is generated only if $phase = ph_0$ after which the phase is incremented. Therefore in a given view, a leader can only make one proposal, which is checked by at least one correct node ($f{+}1$ nodes). A check entails (i) verifying that a proposal is built on its predecessor certificate l. 32 of Fig. 5a, and (ii) checking the validity of accumulators l. 5 of Fig. 5b.

A block can be executed by a correct node if it is stored by $f{+}1$ nodes using `TEEstore`. This guarantees that a store certificate $\mathsf{store}(view, h, v)_{\sigma'}$ is generated only when the view v is recorded as the last prepared view using $prepv$. Moreover, only proposals with views at least as high as $prepv$ can be stored. For two conflicting blocks b_1 and b_2, respectively proposed at views v_1 and v_2, to be executed by two correct nodes, both blocks must be stored by $f{+}1$ nodes. Because the two blocks are conflicting and leaders can only make one proposal per view, it must be that $v_1 \neq v_2$. Assume w.l.o.g. that $v_1 < v_2$ and let us consider what happens between the two views. The leader of view $v_1{+}1$ will either wait until it gets a prepare certificate from view v_1 signed by $f{+}1$ nodes (a normal execution), in which case it must be for b_1 because nodes, even Byzantine nodes, can only store one block per view, and there must be a node (possibly Byzantine) at the intersection of the $f{+}1$ that stored b_1 and this prepare certificate. The leader will then propose a block that extends b_1 (any block not extending b_1 cannot be accepted by $f{+}1$ nodes). Otherwise the leader will collect $f{+}1$ new-view messages, and make a proposal based on those messages. Therefore there is a (possibly Byzantine) node j that has stored b_1 and sent one of these new-view messages. If $v_1{+}1$ is a piggyback execution, the leader will re-construct a prepare certificate for b_1 since the new-view message from j is part of that certificate. In case of a catch-up execution, the leader calls its accumulator, which generates a certificate for b_1 (the last prepared block) and hence proposes an extension of b_1. Therefore, in case a view between v_1 and v_2 ends with $f{+}1$ nodes storing a block, this block must be received and extended by the next leader as it either waits for a prepare certificate from the previous view, or for $f{+}1$ new-view messages. In case a view between v_1 and v_2 ends without $f{+}1$ nodes storing the block b proposed in that view, some nodes might still store it and send it in their new-view messages. The leader of the next view might extend a block conflicting with b. However, safety is still maintained because no node has received a prepare certificate to execute b. $\square$

OneShot makes use of exponential backoff and leader election mechanisms [6], which guarantee that after the Global Stabilization Time (GST) there will eventually be a view with a correct leader such that all correct nodes stay in that view long enough to reach agreement. We now prove the progress part of this statement.

Lemma 2. *OneShot is live, i.e., after GST, there exists a bounded time period T such that if all correct nodes remain in view v during T and v's leader is correct, then a block is executed.*

Proof. If the leader of view v receives a prepare certificate from the previous view for a block b_0, it starts a normal execution, proposing a block b_1 that extends b_0. All correct nodes will accept the proposal and store it since the proposal is for the latest view. The leader is able to collect *store certificates* from $f{+}1$ nodes because at least the $f{+}1$ correct nodes will send theirs, from which it is then able to form a prepare certificate. The leader sends this certificate to all nodes, and all correct nodes will execute the block once they have pulled all previous blocks (Sec. VI-E).

If the leader receives $f{+}1$ new-view messages and is able to re-construct a prepare certificate from those messages, it starts a piggyback execution, and proceeds as above.

If the leader receives $f{+}1$ new-view messages and cannot reconstruct a prepare certificate, it calls its accumulator to generate a certificate for the highest proposal b_0. It then starts the deliver phase to get a vote certificate for b_0, thereby starting a catch-up execution. At least all $f{+}1$ correct nodes will vote for b_0, and so the leader is able to collect a vote certificate. It then extends b_0 with its own proposal b_1, in which case the view proceeds similar to a normal execution. $\square$

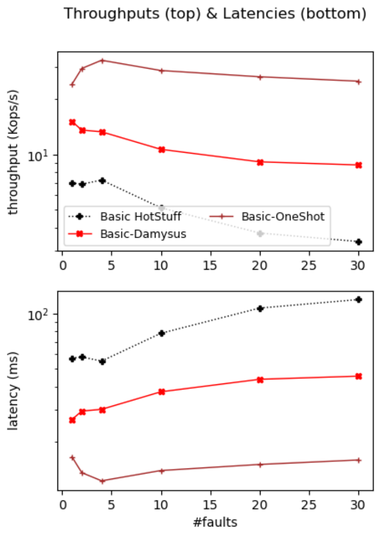
VIII. Evaluation

We evaluate OneShot's performance (including the optimizations described in Sec. VI-F), and compare it to Damysus and HotStuff. All three protocols are implemented in C++ , and both Damysus and OneShot make use of Intel SGX enclaves [13] to run their trusted services. Although our trusted services are generic enough to be potentially implemented in any trusted execution environment, we use SGX because its Linux SDK provides a convenient development environment[4]. Replicas use ECDSA signatures with prime256v1 elliptic curves (available in OpenSSL [48]), and are connected using Salticidae [49].

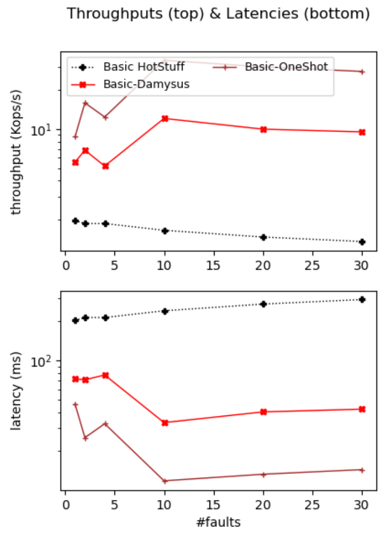
We deploy the protocols on AWS EC2 machines with one t2.micro instance per node. Each figure represents the average of 100 views (10 repetitions of 10 views each). The fault threshold varies as follows: $f \in \{1, 2, 4, 10, 20, 30\}$, resulting in systems of 91 nodes for HotStuff, and 61 nodes for Damysus

[4]We acknowledge that there are known SGX security vulnerabilities [47].

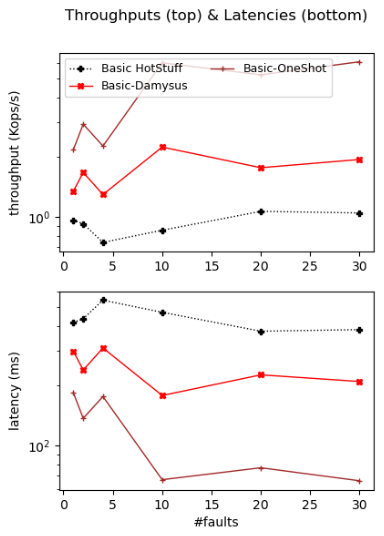
Fig. 7 Throughput and latency without failures using EU (left), US (middle) and Worldwide (right) regions



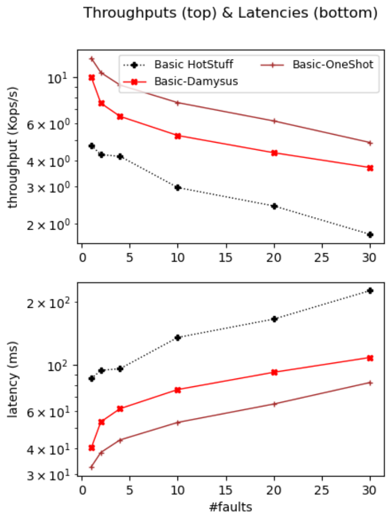
(a) EU regions & 0 B pl.



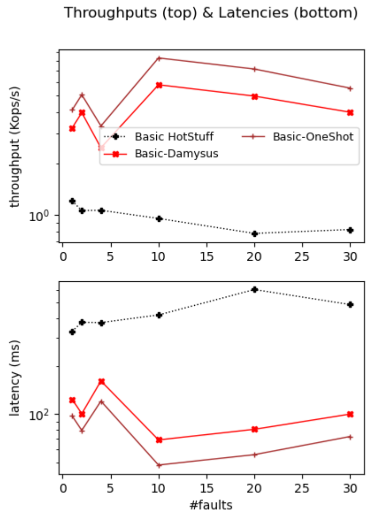
(b) US regions & 0 B pl.



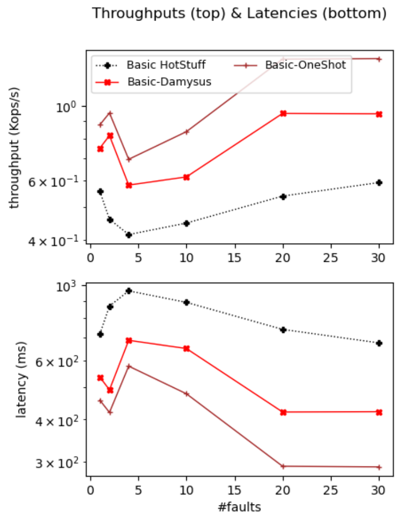
(c) Worldwide regions & 0 B pl.



(d) EU regions & 256 B pl.



(e) US regions & 256 B pl.



(f) Worldwide regions & 256 B pl.

and OneShot when $f = 30$. Blocks contain 400 transactions each, and all transaction payloads are either of size 0B or 256B. In addition to its payload, a transaction contains 2×4B for metadata (a client id, and a transaction id), as well as the hash value of the previous block of size 32B, thereby adding 40B to each transaction in addition to its payload. Therefore, in the experiments with 0B payloads, blocks have a size of 400×40B=15.6KB, and in the experiments with 256B payloads, blocks have a size of $400 \times (40 + 256)$B=115.6KB. Payloads of 0B are used to evaluate the protocols' overhead, while payloads of 256B are used to observe the trend when increasing the size of blocks, which as shown below allows observing a significant latency increase and throughput decrease for all protocols.

Note that the executions of distributed systems for solving consensus are often divided into normal case executions (fault-free executions) and "abnormal" executions (where failures happen). For example, in addition to its normal case operation, PBFT includes a view-change operation, which is triggered when the current leader is suspected to be faulty. Typically, the normal case execution of these systems (such as [16, 50]) is the one happening the most in practice. The "abnormal" executions only happen rarely. Therefore, the focus is often put on making the normal case executions fast, and on evaluating that they are indeed fast. Following this approach, the evaluation below focuses on OneShot's normal case execution.

As explained in Sec. V, OneShot's "abnormal" executions are the catch-up and piggyback executions that can happen for example when substantial network disruptions slow down some nodes, which can be compounded with the effect of Byzantine nodes being deliberately slow. However, such scenarios are rare in practice and are intrinsic to any (fault-tolerant) distributed system. Furthermore, the preemption mechanism presented in

Sec. VI-F allows preempting such catch-up and piggyback executions with improved network conditions.

We now present three sets of experiments, which allow comparing OneShot's latency and throughput with those of Damysus and HotStuff, under different network conditions: one where nodes are deployed across regions in Europe, where the largest average network latency is 29ms; one where nodes are deployed across regions in the US, where the largest average network latency is 65ms; and one where nodes are deployed across regions across the world, where the largest average network latency is 278ms. As we can observe from those experiments, OneShot performs better than Damysus and HotStuff in all considered scenarios. Note that the performance improvement over Damysus is essentially due to the reduced number of communication phases that comes with no additional TEE overhead, while the simplifications of the trusted services discussed in Sec. VI-A have little impact on this improvement. Furthermore, the results are broadly on a par with each other in all three scenarios.

a) EU deployments: Fig. 7 shows experiments with nodes deployed across 4 regions in Europe (Ireland, London, Paris, and Frankfurt, where the largest average latency is between Ireland and Frankfurt and is 29ms), comparing the throughput and latency (measured by the replicas) of the 3 protocols mentioned above. We observe in these figures the following throughput gains and latency decreases over HotStuff and Damysus, where an entry $X(Y, Z)$ means that the gain or decrease is on average $X\%$, with values ranging from $Y\%$ to $Z\%$. The throughput gains are as follows:

	HotStuff	Damysus
0B	+439% (244, 647)	+144% (59, 190)
256B	+151% (119, 174)	+36% (23, 43)

while the latency decrease are as follows:

	HotStuff	Damysus
0B	−79% (70, 86)	−57% (37, 65)
256B	−60% (54, 63)	−26% (18, 30)

b) US deployments: Fig. 7 shows experiments with nodes deployed across 4 regions in the US (North Virginia, Ohio, North California, and Oregon, where the largest average latency is between Oregon and North Virginia and is 65ms). The throughput gains are as follows:

	HotStuff	Damysus
0B	+1242% (340, 1938)	+150% (56, 201)
256B	+500% (212, 820)	+35% (27, 44)

Latency decreases:

	HotStuff	Damysus
0B	−89% (77, 95)	−59% (36, 66)
256B	−80% (67, 90)	−26% (21, 30)

c) Worldwide deployments: Fig. 7 shows experiments with nodes deployed across 11 regions across the world (4 in the US in North Virginia, Ohio, North California, and Oregon; 4 in Europe in Ireland, London, Paris, and Frankfurt; 2 in Asia in Singapore and Sydney; and 1 in Canada Central; where the largest average latency is between Sydney and Paris and is 278ms). The throughput gains are as follows:

	HotStuff	Damysus
0B	+338% (127, 602)	+131% (62, 212)
256B	+101% (57, 154)	+30% (16, 45)

Latency decreases:

	HotStuff	Damysus
0B	−73% (55, 85)	−53% (38, 68)
256B	−48% (36, 60)	−22% (14, 31)

d) Unstable and Degraded Network Conditions: While theoretically it can happen that OneShot provides worse performance than Damysus under highly unstable and degraded network conditions, as explained above, such scenarios are rare in practice. To demonstrate how seldom such cases happen in practice, we ran local experiments with 256B payloads, 10ms network latency, and artificially triggered catch-up and piggyback executions, that are triggered either 25%, 33% or 50% of the views. The only scenario where the performance of OneShot dropped to be the same as that of Damysus was when the worst case catch-up execution of OneShot was artificially triggered 50% of the time. In that case, OneShot's throughput became comparable with Damysus's, while still being higher than HotStuff's. This is in part due to the fact that those executions can potentially retransmit large blocks. In practice it would be exceedingly rare for this situation to happen as triggering catch-up executions requires substantially degraded network conditions.

IX. Conclusion

We described OneShot, the first streamlined hybrid BFT protocol that achieves the minimal number of communication rounds. It does so by leveraging the knowledge available to the nodes about the state of the system to adapt executions: normal and piggyback executions are triggered when enough nodes know about the latest proposal, leading to one and two accepted blocks, respectively; and catch-up executions are triggered when there might not be a correct node that holds the latest proposal, leading to two accepted blocks. We showed that, OneShot indeed outperforms both Damysus and HotStuff, and proved its correctness. As other streamlined protocols, OneShot can be seamlessly turned into a chained version.

References

[1] F. B. Schneider. "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial". In: *ACM Comput. Surv.* 22.4 (1990), pp. 299–319.

[2] M. Castro and B. Liskov. "Practical Byzantine Fault Tolerance". In: *OSDI*. USENIX Association, 1999.

[3] C. Dwork, N. A. Lynch, and L. J. Stockmeyer. "Consensus in the presence of partial synchrony". In: *J. ACM* 35.2 (1988), pp. 288–323.

[4] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. "Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults". In: *NSDI*. USENIX Association, 2009.

[5] A. N. Bessani, J. Sousa, and E. A. P. Alchieri. “State Machine Replication for the Masses with BFT-SMART”. In: *DSN*. IEEE, 2014.

[6] M. Yin, D. Malkhi, M. K. Reiter, G. Golan-Gueta, and I. Abraham. “HotStuff: BFT Consensus with Linearity and Responsiveness”. In: *PODC*. ACM, 2019.

[7] T. H. Chan, R. Pass, and E. Shi. “PiLi: An Extremely Simple Synchronous Blockchain”. In: *IACR Cryptol. ePrint Arch.* (2018), p. 980.

[8] T. H. Chan, R. Pass, and E. Shi. “PaLa: A Simple Partially Synchronous Blockchain”. In: *IACR Cryptol. ePrint Arch.* (2018), p. 981.

[9] E. Buchman, J. Kwon, and Z. Milosevic. “The latest gossip on BFT consensus”. In: *CoRR* abs/1807.04938 (2018). arXiv: 1807.04938.

[10] E. Shi. “Streamlined Blockchains: A Simple and Elegant Approach (A Tutorial and Survey)”. In: *ASIACRYPT*. Vol. 11921. LNCS. Springer, 2019.

[11] B. Y. Chan and E. Shi. “Streamlet: Textbook Streamlined Blockchains”. In: *AFT*. ACM, 2020.

[12] *TPM*. URL: https://trustedcomputinggroup.org/work-groups/trusted-platform-module/.

[13] *SGX*. URL: https://software.intel.com/en-us/sgx.

[14] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Veríssimo. “Efficient Byzantine Fault-Tolerance”. In: *IEEE Trans. Computers* 62.1 (2013), pp. 16–30.

[15] J. Decouchant, D. Kozhaya, V. Rahli, and J. Yu. “DAMYSUS: streamlined BFT consensus leveraging trusted components”. In: *EuroSys*. ACM, 2022.

[16] X. Sui, S. Duan, and H. Zhang. “Marlin: Two-phase BFT with linearity”. In: *DSN*. IEEE. 2022.

[17] D. Malkhi and K. Nayak. “Extended Abstract: HotStuff-2: Optimal Two-Phase Responsive BFT”. In: https://eprint.iacr.org/2023/397. 2023.

[18] M. M. Jalalzai, J. Niu, C. Feng, and F. Gai. “Fast-hotstuff: A fast and resilient hotstuff protocol”. In: *arXiv preprint arXiv:2010.11454* (2020).

[19] M. Abspoel, T. Attema, and M. Rambaud. “Malicious security comes for free in consensus with leaders”. In: *Cryptology ePrint Archive* (2020).

[20] N. Giridharan, H. Howard, I. Abraham, N. Crooks, and A. Tomescu. “No-commit proofs: Defeating livelock in bft”. In: *Cryptology ePrint Archive* (2021).

[21] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel. “CheapBFT: resource-efficient byzantine fault tolerance”. In: *EuroSys*. ACM, 2012.

[22] R. Rodrigues, B. Liskov, K. Chen, M. Liskov, and D. A. Schultz. “Automatic Reconfiguration for Large-Scale Reliable Storage Systems”. In: *IEEE Trans. Dependable Sec. Comput.* 9.2 (2012), pp. 145–158.

[23] D. S. Silva, R. Graczyk, J. Decouchant, M. Völp, and P. Esteves-Verissimo. “Threat Adaptive Byzantine Fault Tolerant State-Machine Replication”. In: *SRDS*. IEEE. 2021.

[24] P. Kuznetsov and A. Tonkikh. “Asynchronous reconfiguration with byzantine failures”. In: *Distributed Computing* (2022), pp. 1–26.

[25] M. Castro. “Practical Byzantine Fault Tolerance”. Also as Technical Report MIT-LCS-TR-817. Ph.D. MIT, Jan. 2001.

[26] B. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. “Attested append-only memory: making adversaries stick to their word”. In: *SOSP*. ACM, 2007.

[27] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. “TrInc: Small Trusted Hardware for Large Distributed Systems”. In: *NSDI*. USENIX Association, 2009.

[28] J. Behl, T. Distler, and R. Kapitza. “Hybrids on Steroids: SGX-Based High Performance BFT”. In: *EuroSys*. ACM, 2017.

[29] A. Haeberlen, P. Kouznetsov, and P. Druschel. “PeerReview: Practical accountability for distributed systems”. In: *ACM SIGOPS operating systems review* 41.6 (2007), pp. 175–188.

[30] S. B. Mokhtar, J. Decouchant, and V. Quéma. “Acting: Accurate freerider tracking in gossip”. In: *SRDS*. IEEE. 2014.

[31] A. Diarra, S. B. Mokhtar, P.-L. Aublin, and V. Quéma. “Fullreview: Practical accountability in presence of selfish nodes”. In: *SRDS*. IEEE. 2014.

[32] T. Distler, C. Cachin, and R. Kapitza. “Resource-Efficient Byzantine Fault Tolerance”. In: *IEEE Trans. Computers* 65.9 (2016), pp. 2807–2819.

[33] J. Liu, W. Li, G. O. Karame, and N. Asokan. “Scalable Byzantine Consensus via Hardware-Assisted Secret Sharing”. In: *IEEE Trans. Computers* 68.1 (2019), pp. 139–151.

[34] J. Zhang, J. Gao, K. Wang, Z. Wu, Y. Lan, Z. Guan, and Z. Chen. “TBFT: Understandable and Efficient Byzantine Fault Tolerance using Trusted Execution Environment”. In: *CoRR* abs/2102.01970 (2021). arXiv: 2102.01970.

[35] S. Bano, M. Baudet, A. Ching, A. Chursin, G. Danezis, F. Garillot, Z. Li, D. Malkhi, O. Naor, D. Perelman, and A. Sonnino. “State Machine Replication in the Libra Blockchain”. 2019.

[36] S. Yandamuri, I. Abraham, K. Nayak, and M. K. Reiter. “Brief Announcement: Communication-Efficient BFT Using Small Trusted Hardware to Tolerate Minority Corruption”. In: *DISC*. Vol. 209. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fur Informatik, 2021.

[37] M. F. Madsen, M. Gaub, M. E. Kirkbro, and S. Debois. “Transforming Byzantine Faults using a Trusted Execution Environment”. In: *EDCC*. IEEE, 2019.

[38] A. Clement, F. Junqueira, A. Kate, and R. Rodrigues. “On the (limited) power of non-equivocation”. In: *PODC*. ACM, 2012.

[39] R. Gelashvili, L. Kokoris-Kogias, A. Sonnino, A. Spiegelman, and Z. Xiang. “Jolteon and Ditto: Network-Adaptive Efficient Consensus with Asynchronous Fallback”. In: *FC*. Vol. 13411. LNCS. Springer, 2022.

[40] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun. “ROTE: rollback protection for trusted execution”. In: *USENIX Security*. 2017.

[41] W. Wang, S. Deng, J. Niu, M. K. Reiter, and Y. Zhang. “ENGRAFT: Enclave-guarded Raft on Byzantine Faulty Nodes”. In: *ACM CCS*. 2022.

[42] H. Buhrman, J. A. Garay, and J. Hoepman. “Optimal Resiliency against Mobile Faults”. In: *FTCS*. IEEE, 1995.

[43] J. Niu, W. Peng, X. Zhang, and Y. Zhang. “NARRATOR: Secure and Practical State Continuity for Trusted Execution in the Cloud”. In: *ACM CCS*. 2022.

[44] S. Gupta, S. Rahnama, S. Pandey, N. Crooks, and M. Sadoghi. “Dissecting BFT Consensus: In Trusted Components we Trust!” In: *EuroSys*. ACM, 2023.

[45] C. Stathakopoulou, T. David, M. Pavlovic, and M. Vukolic. “Mir-BFT: Scalable and Robust BFT for Decentralized Networks”. In: *J. Syst. Res.* 2.1 (2022).

[46] C. Stathakopoulou, M. Pavlovic, and M. Vukolic. “State machine replication scalability made simple”. In: *EuroSys ’22*. ACM, 2022.

[47] K. Murdock, D. F. Oswald, F. D. Garcia, J. V. Bulck, D. Gruss, and F. Piessens. “Plundervolt: Software-based Fault Injection Attacks against Intel SGX”. In: *SP*. IEEE, 2020.

[48] *OpenSSL*. URL: https://www.openssl.org/ (visited on 02/25/2020).

[49] *Salticidae*. URL: https://github.com/Determinant/salticidae (visited on 03/31/2021).

[50] P. Dutta and R. Guerraoui. “Fast Indulgent Consensus with Zero Degradation”. In: *EDCC*. Vol. 2485. LNCS. Springer, 2002.