Hybrids on Steroids: SGX-Based High Performance BFT [*]

Johannes Behl
TU Braunschweig
behl@ibr.cs.tu-bs.de

Tobias Distler
FAU Erlangen-Nürnberg
distler@cs.fau.de

Rüdiger Kapitza
TU Braunschweig
rrkapitz@ibr.cs.tu-bs.de

Abstract

With the advent of trusted execution environments provided by recent general purpose processors, a class of replication protocols has become more attractive than ever: Protocols based on a hybrid fault model are able to tolerate arbitrary faults yet reduce the costs significantly compared to their traditional Byzantine relatives by employing a small subsystem trusted to only fail by crashing. Unfortunately, existing proposals have their own price: We are not aware of any hybrid protocol that is backed by a comprehensive formal specification, complicating the reasoning about correctness and implications. Moreover, current protocols of that class have to be performed largely sequentially. Hence, they are not well-prepared for just the modern multi-core processors that bring their very own fault model to a broad audience. In this paper, we present *Hybster*, a new hybrid state-machine replication protocol that is highly parallelizable and specified formally. With over 1 million operations per second using only four cores, the evaluation of our Intel SGX-based prototype implementation shows that Hybster makes hybrid state-machine replication a viable option even for today's very demanding critical services.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; D.4.5 [*Operating Systems*]: Fault Tolerance; D.4.7 [*Operating Systems*]: Distributed Systems

General Terms Design, Performance, Reliability

Keywords State-Machine Replication, Trusted Execution, Multi-Core, Scalability

1. Introduction

Pure Byzantine fault-tolerant (BFT) service replication has to employ $3f + 1$ service replicas to ensure that a service remains operational even if f of these replicas behave arbitrarily faulty [3, 4, 6, 10, 13, 16, 19, 24, 30, 31, 41, 43, 48]. With a hybrid fault model where small trusted subsystems are assumed to fail only by crashing, the number of required replicas can be lowered to $2f + 1$ [15, 20, 28, 44, 45]. In consequence, hybrid replication protocols require less diversification of replicas and tend to require less resources.

On the downside, while the originator PBFT [13] comes with a comprehensive formal specification [12], hybrid protocols mostly rely on that foundation independent of their particular needs. However, without a formal specification, reasoning about such complex protocols becomes difficult. Besides this drawback concerning the theory, there is also a major practical one which seems to be a direct consequence of the employed fault model: All proposed hybrid protocols rely on a kind of sequential processing. That hinders them from taking full advantage of modern multi-core systems and from adopting new parallelization schemes, helping traditional BFT protocols reach unprecedented performance levels [9]. Further, it is exactly these modern platforms that make a hybrid fault model widely accessible. The latest general purpose processors can provide so-called trusted execution environments that protect a software component even against malicious behavior of an untrusted operating system. Hence, hybrid protocols could in fact greatly benefit from technologies such as Intel SGX [34] or ARM TrustZone [5].

For these reasons, we present *Hybster*, a new hybrid state-machine-replication protocol that is highly parallelizable and formally specified. Hybster was designed after a thorough analysis of existing proposals (Section 4) and builds on a tailored subsystem that leverages trusted execution environments (Section 5.1). It is composed of a sequential basic protocol that avoids problems of its existing counterparts (Section 5.2) and a parallelized variant using a consensus-oriented parallelization [9] (Section 5.3). The evaluation of an SGX-based implementation shows that Hybster is able to achieve over 1 million operations per second on quad-core machines (Section 6), considerably more than the 72 thousand published for hybrid protocols so far [28] and sufficient even for very demanding services [1, 2, 35, 46].

[*] This work was partially supported by the German Research Council (DFG) under grant no. KA 3171/1-2 and DI 2097/1-2.

EuroSys '17 April 23–26, 2017, Belgrade, Serbia

ISBN 978-1-4503-4938-3/17/04…$15.00
DOI: http://dx.doi.org/10.1145/3064176.3064213

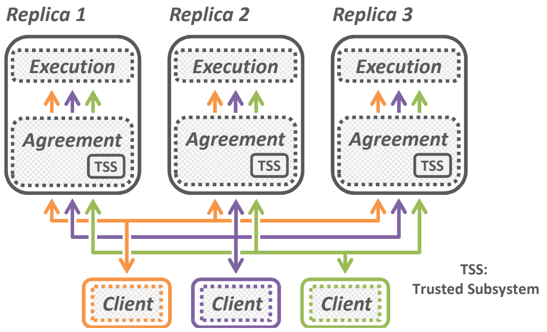


Figure 1. Hybrid BFT state-machine replication.

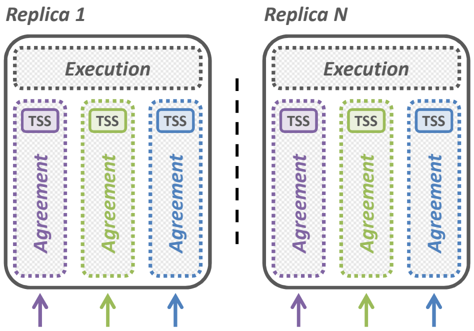


Figure 2. A parallelizable hybrid - the idea.

2. Background

Byzantine fault-tolerant state-machine replication [13, 40] allows systems to operate continuously even if some of their replicas become faulty and no longer behave according to the specification. Relying entirely on untrusted components, a minimum of $3f + 1$ replicas are required to tolerate up to f arbitrary faults [13, 36]. To reduce the computational and communication costs associated with fault tolerance, different approaches have been presented in recent years that utilize trusted subsystems [15, 18, 22, 28, 32, 37, 44, 45, 47]. Such systems use a hybrid fault model assuming that some components are untrusted and may fail arbitrarily, while other components are trusted and therefore only fail by crashing. Most of these hybrid systems reduce the minimum number of required replicas to $2f + 1$ by utilizing their trusted subsystem to prevent undetected *equivocation*, that is, they remove the replica's ability to make conflicting statements without being convicted by other replicas.

Figure 1 shows the basic architecture of a (hybrid) BFT system based on state-machine replication. When a client issues a request to the replica group, the replicas execute the request and each sends a reply. By accepting a result only if at least $f+1$ replies from different replicas match is a client able to tolerate up to f faulty replies. To ensure consistent replies, replicas carry out a Byzantine fault-tolerant state-machine replication protocol, which itself relies on a set of subprotocols with different objectives. The three most important subprotocols are briefly discussed in the following: 1) The *ordering protocol* establishes a global total order in which requests and other commands have to be executed. For this purpose, the protocol assigns a unique order number to each request and ensures that a sufficient number of correct replicas reaches consensus on these assignments. 2) The *checkpointing protocol* periodically creates consistent snapshots of replica states and uses them to compile a proof that the overall system has made a certain progress. This is crucial as such a proof allows other subprotocols to perform garbage collection. 3) The *view-change protocol* ensures that the system makes progress despite faulty replicas. In particular, this protocol represents a means that allows the replica group to safely switch between configurations (i.e., views) even if up to f of the group members are arbitrarily faulty.

3. Hybrid Yet Parallel – The Problem and the Approach

To our knowledge, the ordering protocols of all existing state-machine replication systems based on a hybrid fault model require a sequential processing either of consensus instances that are performed to agree on the order in which replicas must execute commands [15] or of all incoming messages [20, 28, 44, 45]. Not being able to parallelize the agreement, however, fundamentally restricts the performance of such systems on modern platforms [9].

Unfortunately, sequential processing seems to be inherent to the hybrid fault model itself: From an abstract perspective, all hybrid systems prevent undetected equivocation by cryptographically binding sensitive outgoing messages to a unique monotonically increasing timestamp by means of the trusted subsystem. If a faulty replica makes conflicting statements by sending different messages to different replicas in one and the same phase of a protocol, it can only do so with different timestamps, allowing other replicas to detect such a behavior. However, the unique timestamps establish a virtual timeline for each replica that ultimately induces the sequential nature inherent to all existing hybrid protocols.

In this paper, we address this problem with an approach that circumvents this limitation and enables parallel processing of hybrid consensus instances as depicted in Figure 2: Instead of a single virtual timeline, each replica uses multiple independent timelines; instead of a single trusted subsystem, each replica is equipped with as many subsystems as the targeted degree of parallelism requires. This is perfectly backed by the concept of consensus-oriented parallelization [9] where equal processing units are responsible for a statically assigned subset of consensus instances, thereby preventing the opportunity for equivocation.

However, such an approach is only possible when the type of trusted subsystem allows it. Given the state-of-the-art in that area, the FPGA-based CASH [28], multiplying the subsystem would translate into equipping a system with as many extension cards as it has processor cores, impractical for many-core machines. Contrarily, using a trusted execution environment provided by the central processing unit itself, or more precisely by using Intel's variant called Software Guard Extensions (SGX) [34], this is perfectly feasible.

4. Existing Hybrids Revisited

As mentioned above, none of the existing hybrid protocols [15, 20, 28, 45] is directly prepared for non-sequential ordering using multiple virtual timelines. Still, these protocols represent a starting point for the design of a new one. We therefore first analyze existing systems, in particular *A2M-PBFT* [15] and *MinBFT* [45], and identify their conceptional building blocks as well as their strengths and weaknesses before we present our parallelizable protocol thereafter.

4.1 Someone to Trust

Replication protocols are complex and comprise a large state when the throughput is high, increasing the risk of subtle errors. Thus, to substantiate the trust assumption, the proposals considered here make use of trusted subsystems that are as small as possible. While several ways exist for implementing these subsystems (e.g., in software, using configurable hardware, or specialized chips [15, 28, 32, 45]), we examine the abstractions they are based on, that is, what functionality they provide and how much state they have to maintain.

Proposals. To our knowledge, there are three proposals for such abstractions in the context of hybrid service replication:

The hybrid protocol *A2M-PBFT* [15] is based on a trusted subsystem that provides attested append-only memory, short *A2M*. A2M stores values, in the case of A2M-PBFT digests of outgoing messages, in distinct trusted logs. Log entries are consecutively numbered and can only be written in ascending order. For each entry used, A2M issues certificates that prove that a particular value has been placed at this position of a trusted log. Although A2M logs can be truncated, active log entries are nevertheless part of A2M's state.

Another proposal for a general trusted subsystem is *TrInc* [32]. TrInc provides a set of increasing counters and binds unique counter values to message digests by calculating a certificate over value and digest. In contrast to A2M, the certified digests do not become part of TrInc's state. In [32], the authors present an algorithm that shows how the interface of A2M can be mapped to that of TrInc.

MinBFT [45] is a hybrid protocol that is directly specified on the basis of trusted counters. Compared to TrInc where counter values are explicitly set when certificates are created, the trusted subsystem *USIG* employed by MinBFT exhibits a simpler interface with only a single counter implicitly incremented at each certification. The interface of *CASH*, the FPGA-based trusted subsystem behind *CheapBFT* [28] and its successor *ReMinBFT* [20], is comparable to that of USIG but supports multiple independent counters.

Assessment. A principal goal behind trusted subsystem abstractions is to provide an interface that is as expressive as necessary to be most widely applicable but as restrictive as possible to entail only a small trusted computing base [38], an important argument to justify the trust in the first place.

A2M needs to store all certified messages as part of the subsystem, clearly a disadvantage compared to USIG where the size of the state is very small and even known in advance, hence fostering hardware implementations such as CASH. Further, A2M's interface offers more mechanisms and is therefore more complex than that of USIG. Consequently, the in our opinion most promising abstraction is TrInc. Its counter-based interface can be regarded as slightly more complex than that of USIG but it is also more versatile. Additionally, the size of the (still relatively small) state can be bounded by limiting the maximum number of counters.

► ***Logs vs. Counters.*** A flexible trusted counter abstraction such as TrInc promises the best tradeoff between an expressive interface and a small trusted computing base.

4.2 Cope with Equivocation

While the trusted subsystem provides the mechanism, the hybrid replication protocol implements the policy, which in this case is how the system deals with conflicting messages (i.e., ambiguous statements of Byzantine processes).

Approaches. Besides the different abstractions of the employed trusted subsystems, A2M-PBFT and MinBFT pursue fundamentally different approaches for coping with equivocation. In A2M-PBFT each possible message of each possible protocol instance is stored in a log entry defined a priori, that is, each message is located in a predetermined place in the virtual timeline of a replica. Contrarily, in MinBFT, this place is not known in advance but determined at run time. It depends on the current state of the unique counter to which value an outgoing message is assigned to. For example, if a replica sends two messages A and B for the protocol instances 100 and 101, respectively, A2M-PBFT requires the messages to be assigned to the log entries with exactly these numbers, 100 and 101. In MinBFT, the counter values used could be 521 and 523, depending on which other messages the replica has sent before and in between A and B.

Evaluation. The consequences are far-reaching. A faulty replica in A2M-PBFT is not able to generate two conflicting messages A and B for a protocol instance 100 that are both equipped with a certificate that is (1) valid and (2) contains the expected log entry number 100. In that sense, the A2M-PBFT approach *prevents* equivocation. MinBFT instead allows generating certificates for two conflicting messages A and B for one protocol instance 100 that are both valid if taken by themselves. Their counter values are, however, unique, which enables receiving replicas to observe this faulty behavior by comparing the two messages or by processing them in the order of their counter values. In that sense, the MinBFT approach does not prevent equivocation but allows *detecting* it [45]. On the one hand, A2M-PBFT requires slightly more complex trusted subsystems, but on the other hand, it restricts the hostile, Byzantine outer world more and earlier in their actions than the MinBFT approach.

► ***Prevention vs. Detection.*** Preventing equivocation further restricts the capabilities of a Byzantine replica and can consequently simplify the protocol design.

4.3 Order Requests

The main task of the replication protocol is to establish an order for requests and other commands in which all correct replicas must execute them to maintain a consistent state. Fulfilling this task, the ordering subprotocol is a decisive factor in the performance of a replicated system.

Options. Following the traditional state-machine replication approach, there are two options for how many phases of message exchange the ordering in hybrid protocols requires:

A2M-PBFT uses a *three-phase ordering protocol* adopted from PBFT [13]: A distinguished replica, the *leader*, proposes an assignment of requests to an order number o by sending a PRE-PREPARE message to all other replicas. In the following two phases, replicas acknowledge that proposal to all others while waiting for a *quorum* after each phase, with a quorum comprising messages from $f+1$ among $2f+1$ replicas: If they receive a PRE-PREPARE, they send a PREPARE; if they receive a quorum of PREPAREs, they send a COMMIT; if they receive a quorum of COMMITs, the requests are eligible for execution.

The second option for a hybrid ordering protocol is to rely on only *two phases* as done by MinBFT: The leader sends its proposal in a PREPARE and all other replicas acknowledge it with a COMMIT. Requests are executed after a quorum of COMMITs has been received.

Side Effects. With respect to performance, a two-phase ordering as employed by MinBFT is apparently superior. It saves one communication phase where all replicas send messages to each other; hence it spares network bandwidth and reduces end-to-end latency.

However, performing only two phases has a notable side effect. A sole message sent by a replica for a consensus instance o indicates that the requests ordered by o are potentially executed by some replica in the group. Given 3 replicas and a quorum size of 2, a replica r can execute the requests of an instance o when it has received the PREPARE from the leader l and generated its COMMIT. Provided that the third replica never sent a message for o and either replica l or r is temporarily disconnected due to network problems. Further provided that an error is suspected and the current state of the system has to be determined. In the given situation, there is only one message, the PREPARE from l or the COMMIT from r, that indicates the execution at replica r.

With three phases, requests are not executed before quorums in the second and third phase have been reached; thus, replicas can provide a quorum of matching messages, called a *quorum certificate*, to announce that a request has been potentially executed. Since at least one correct replica is part of a quorum, all messages in a quorum certificate are confirmed to be correct according to the protocol. This does not hold for a single message in a two-phase ordering.

► ***Two vs. Three Phases.*** Two-phase ordering provides better performance but requires additional efforts to safely determine the system state in the case of errors.

4.4 Change Views to Conclude

While the ordering subprotocol is decisive for the performance of a system based on state-machine replication, the view-change protocol is crucial for its safety. If something does not proceed as expected, the other protocols rely on the view-change protocol to re-establish a stable configuration that allows the system to make progress. At the same time, the view-change protocol depends on the situation the other protocols leave behind when a view-change is invoked.

Challenges. One of the main responsibilities of the view-change protocol is to ensure that all commands executed by some correct replicas in some view will be eventually executed by all correct replicas in this or any later view. It can be argued that these responsibilities are also the main challenges for hybrid state-machine replication protocols: In the non-hybrid PBFT, there is at least one correct replica in the intersection of any two quorums. Following the example of the previous section, this replica is able to propagate a quorum certificate for consensus instances potentially executed at some replicas to later views. In contrast, a hybrid protocol only has one replica in any intersection of quorums. The protocol has to ensure that executed instances are properly propagated to a subsequent view even if this replica is faulty.

Recipes. For that purpose, MinBFT uses the counter values of its trusted subsystem USIG to force even faulty replicas to provide a complete history of all outgoing messages. That way, all potentially executed consensus instances are revealed during a view-change. With a mere equivocation detection and a two-phase ordering, this would result in an arbitrarily complex view-change protocol. If a faulty leader sent different PREPARE messages for one consensus instance, this could only be detected by comparing the histories provided by different replicas. Therefore, MinBFT also uses a form of equivocation prevention for PREPAREs: It does not use dedicated order numbers but relies on the value of the counter certificate to determine the total order. Nevertheless, for all other protocol messages, the described detection mechanism is used, which may lead to complex situations during the view-change. Also the checkpointing protocol employed to discard obsolete messages becomes more complex since replicas have to ensure that they are always able to provide a complete history of outgoing messages. This, however, makes it difficult to guarantee that these histories remain bounded in size when a view-change takes multiple rounds to find a new leader.

Histories of outgoing messages to cope with equivocation complicate matters, but the problem has an essential core. In PBFT, the correct replicas in the intersection of quorums will announce any command that might have been executed; in hybrid protocols, single replicas must prove that no more than the stated commands can have been executed. In the one case, quorum certificates are used to announce that something potentially did happen; in the other case, histories are used to prove that something certainly did not happen.

For that very reason, A2M-PBFT relies also on a history. For each round in which no leader was able to be elected, a replica in A2M-PBFT must add a message to a list proving that it indeed has not sent any ordering message since the last known stable configuration, the last view where it actively participated in the ordering of commands. In the employed partially synchronous system model, however, there is no a priori known upper bound on the number of rounds needed to elect a new leader and hence no known upper bound for this list, hence no known upper bound for the memory usage or the message sizes. The list will always be finite because a new stable view will eventually be reached, as long as the system model assumes that processes have access to a sufficient, virtually unlimited amount of memory. In a model where available storage is finite, a replica could have already run out of memory or messages could have grown to an impractical size before enough replicas agree on a new leader. The ordering protocol can rely on the checkpointing protocol to perform garbage collection. There is no such protocol that backs the view-change protocol.[1]

► ***Histories vs. ?*** Histories lead to an unknown upper bound for the memory demand of existing protocols, but an alternative is missing so far.

5. Hybster

Following the analysis of existing systems, we propose a hybrid replication protocol that is designed around a two-phase ordering and a concept that prevents equivocation by means of multiple instances of a TrInc-based trusted subsystem realized using Intel SGX; we dub this new hybrid protocol *Hybster*. Besides this unique set of building blocks, Hybster is distinct from other hybrids in three ways:

1. ***Relaxed.*** Opposed to its existing counterparts, Hybster does not try to force faulty replicas during a view-change to reveal all consensus instances for which they sent an order message. Instead, Hybster is based on the observation that it suffices to ensure that only instances must be revealed that are not guaranteed to be propagated by correct replicas, yet have potentially led to the execution of requests. Putting it the other way round, Hybster allows faulty replicas to conceal own messages as long as they are not critical to ensure safety. This reduces the complexity of the view-change protocol significantly and prevents histories.
2. ***Formal.*** As a consequence, there is not only an informal description of the basic protocol of Hybster as presented in this paper, but it is also formally specified [8].
3. ***Parallelizable.*** Finally, Hybster has the ability to perform consensus instances in parallel. It can benefit from modern multi-core platforms whereas existing hybrid protocols are conceptually confined to a sequential ordering of requests.

The presentation of Hybster is divided into three parts: First, there is the trusted subsystem that provides the mechanisms employed by the protocol to prevent faulty replicas from making conflicting statements. Second, we present the basic, unparallelized replication protocol. Essentially, this is the protocol that is performed among the replicas, the external protocol. The third part delves into its parallelized realization, which is mainly a view of the internal protocols performed by the concurrent processing units of each replica.

5.1 TrInX – SGX-Based Trusted Counters

Hybster builds on a trusted subsystem abstraction that is akin but not equal to TrInc [32]. We call this new abstraction *TrInX*. TrInX tailors and extends TrInc mainly for two reasons: to account for the particular needs of Hybster and to increase performance.

Rationale. Adapting the abstraction for Hybster is viable and reasonable since it primarily targets platforms providing so-called *trusted execution environments*. Such environments are provided by modern CPUs and can be regarded as a special execution mode of the processor that protects parts of the system even against undesired access by the untrusted operating system. This allows creating trusted subsystems on a software basis and makes it feasible to tailor solutions for particular applications. Further, at least Intel's implementation of trusted execution environments, Intel SGX [34], supports creating multiple independent instances of a trusted subsystem. In conjunction with the general approach behind Hybster, multiplying the subsystem promises an increased performance while leaving the trusted computing base unaffected. Nonetheless, the trust also depends on the complexity of the subsystem itself. Thus, it remains advisable to keep the complexity as low as possible and to make use of existing and well-studied abstractions as far as possible.

Description. For the sake of simplicity, we assume that all instances of TrInX are initialized appropriately by a trusted administrator before the replica group is brought to life. During initialization, instances are provided with a unique identifier, the number of required trusted counters, and a secret key shared among all TrInX instances. All counters are set to 0. As is the case for Intel SGX, we further assume that the execution platform provides a means to prevent undetected replay attacks where an adversary saves the (encrypted) state of a trusted subsystem and starts a new instance using the exact same state to reset the subsystem. After being properly set up, a TrInX instance can create different kinds of certificates for a message m using a specified trusted counter tc and also a specified new counter value tv':

• ***Continuing Counter Certificates.*** When such a certificate is requested, a TrInX instance tss_i accepts a new value tv' that is greater than or equal to the current value tv of the counter tc. It then calculates a *message authentication code (MAC)* based on (1) the secret key shared among all instances of the group, (2) its own instance ID, (3) the ID of the counter tc, (4) the intended new value tv', (5) the current value tv, and (6) the message m itself. Subsequently, the tss_i sets the counter value of tc to tv' and returns the

[1] A circumstance PBFT explicitly addresses in its authenticator view-change protocol with bounded space [11].

calculated MAC. Thus, continuing counter certificates are only unique, only bound to a single message if $tv' > tv$. With $tv' = tv$, multiple certificates with the same counter value can be created for different messages. If another TrInX instance tss_j is used to verify a message m equipped with such a certificate, it has to be provided with the instance ID of the issuing TrInX instance tss_i, the expected values for tc, tv', and tv together with m. The verifying instance tss_j can then perform the same calculation on the basis of the shared key. If the MAC included in the certificate equals the calculated MAC, the certificate is valid and stems from the TrInX instance tss_i since the secret key is only known by trusted TrInX instances and no instance issues a certificate that includes the instance ID of another TrInX instance. To refer to this type of certificate, we use the notation $\langle m \rangle_{\tau(tss,tc,tv',tv)}$ or $\tau(tss, tc, tv', tv)$ where tss represents the issuing TrInX instance. This is the type of certificate that is also provided by TrInc.

- ***Independent Counter Certificates.*** To verify a continuing certificate, the previous counter value tv has to be known, although it often suffices to ensure that a message is unique, that is, that there is no other valid certificate for the same counter value tv'. For that purpose, TrInX provides *independent counter certificates* $\tau(tss, tc, tv', -)$ that do not include the previous value tv but are only issued for new counter values tv' that are greater than the current value tv.
- ***Multi-Counter Certificates.*** To prove the state of more than one counter to other parties, it is more efficient to issue certificates over multiple counters instead of creating multiple certificates. For that purpose, TrInX offers the option to create continuing or independent *multi-counter certificates*.
- ***Trusted MAC Certificates.*** Traditional MACs can be used to ensure the authenticity of messages but they do not provide non-repudiability. More than one party possesses the secret key, thus a party can deny the fact that it has sent a message in question. This is not possible when a digital signature is employed. The private key used to certify a message is dedicated to a particular party. Unfortunately, digital signatures are far more expensive to compute than MACs [13]. However, a hybrid fault model opens up another possibility: By including the secret key and their own ID, TrInX instances can provide certificates that are slightly more expensive than traditional MACs due to the requisite switch to the trusted execution environment but provide non-repudiability like digital signatures. We call such a replacement for signatures *trusted MACs*. Instead of providing dedicated means for creating and verifying trusted MACs, it is more flexible to map them to continuing certificates with $tv' = tv$.

5.2 The Basic Protocol

The sequential basic protocol of Hybster is composed of the same core subprotocols as other state-machine replication protocols: ordering, checkpointing, state-transfer, and view-change protocol. Here, we omit the presentation of the state-transfer which can be adopted from existing systems [13].

Hybster assumes a hybrid fault model and a partially synchronous system. Neither the maximum message delay nor the storage available to processes are assumed to be infinite. Yet, for the message delay, the existing upper bound is not known a priori. Following the state-machine replication approach, Hybster relies on a group of replicas $\mathcal{R}$ with $n = |\mathcal{R}|$ where it is able to tolerate up to $f = \lfloor \frac{n-1}{2} \rfloor$ faults by using a minimum quorum size $q = \lceil \frac{n+1}{2} \rceil$. In general, f and q have to satisfy the conditions that (1) there is at least one, possibly faulty, replica in the intersection of every two quorums ($2q > n$) and (2) the number of correct replicas in the system is sufficient to form a quorum ($n \geq q + f$). An important implication of these conditions is that each quorum contains at least one correct replica ($q > f$). Further, each replica r has access to a TrInX instance that can issue certificates of the form $\tau(r, \dots)$. The instance IDs are known to all other replicas as part of the configuration.

5.2.1 Ordering

To establish a total order for requests to be executed, a distinguished replica in the group, the *leader* l, assigns order numbers to requests and proposes this assignment to all others. If a sufficient number of replicas follows the proposal, consensus is reached and the request can be executed. To determine the current leader, the system undergoes consecutively numbered *views* v, for which the leading replica is given by $l = v \bmod n$. If the current leader is suspected to be faulty, the view-change protocol is invoked to elect a new leader. Thus, the ordering protocol has to (1) ensure that no two correct replicas execute different requests with the same order number in the same view and (2) it has to enable the view-change protocol to ensure the same across views.

Hybster employs a two-phase ordering protocol where the leader of a view v proposes an assignment to an order number o via a PREPARE and all other replicas, the *followers*, acknowledge the assignment in a COMMIT message.

Predefined Counter Values. To prevent the leader l from sending different assignments for o to different followers, to prevent equivocation, the PREPARE for o has to be certified with a particular, predefined value of a trusted counter $\mathfrak{O}$ by means of the TrInX instance of l. By using an independent counter certificate as described in Section 5.1, it is guaranteed that l can only create one valid certificate with the expected value; hence, l can only create one valid PREPARE for o. Moreover, correct replicas will only acknowledge a PREPARE with a corresponding COMMIT if the certificate attached to the PREPARE is valid. Further, correct replicas will only execute the proposed request upon having collected a quorum of acknowledgments for that proposal, upon having collected a *committed certificate* for the consensus instance (v, o). Here, the PREPARE is taken as the acknowledgment of the leader and no dedicated COMMIT message is sent. In sum, since all quorums contain at least one correct replica, and since correct replicas acknowledge only valid PREPAREs of the leader of a view v, and since there can only

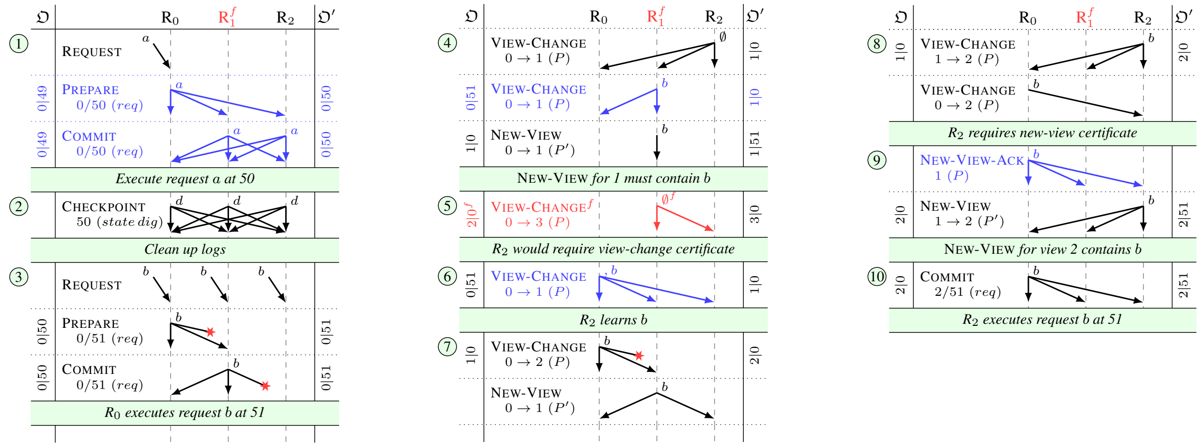


Figure 3. Basic ordering, checkpointing, and view-change protocol of Hybster with a faulty replica R_1^f.

be one valid PREPARE for an order number o in a view v, no two correct replicas will execute a different request for an instance (v, o) than proposed by the valid PREPARE.

Flattened Number Space. However, if a leader of a view v sends a PREPARE for an order number o, it cannot be guaranteed that this proposal will ever reach consensus. In a partially synchronous system, it can always be the case that even a correct leader is suspected to be faulty, perhaps because the network delayed messages. Therefore it can happen that the same replica has to send another PREPARE message for the same order number o but in a later view v' where it is again the leader. Hybster addresses this by flattening the number pair (v, o) to a single number space $[v|o]$ where v is stored in a fixed number of the most significant bits and o in the remaining bits of sequence numbers in that flattened number space [15]. Therefore, the independent counter certificate for a valid PREPARE from a leader l exhibits the form $\tau(l, \mathfrak{O}, v|o, -)$. As a consequence, all messages for higher views are assigned to higher counter values, independent of the value of the order number. This is intentional and used by the view-change protocol described in Section 5.2.3. Moreover, in order to enable the view-change protocol to determine all relevant consensus instances from previous views, a COMMIT message sent by a replica r also has to be certified with an independent certificate $\tau(r, \mathfrak{O}, v|o, -)$. This is possible since when using two-phase ordering each replica only has to send a single message for an order number o in a view v, the leader a PREPARE and all followers a COMMIT.

Example. Figure 3 shows an example case: ① To issue a command to the replicated service, a client sends one enveloped in a certified REQUEST message a to the replica group. Let R_0 be the leader of view $v = 0$ and 49 the last order number that was assigned by R_0. After having received request a, Replica R_0 verifies its correctness, assigns the next unused order number $o = 50$ to it, and sends a PREPARE message for the consensus instance $(0, 50)$ to propose the assignment to the followers R_1 and R_2. To be valid, the PREPARE has to be equipped with the certificate $\tau(R_0, \mathfrak{O}, 0|50, -)$. Consequently, the value of the trusted counter $\mathfrak{O}$ of R_0 is increased to $[0|50]$. A receiving follower verifies the proposal and acknowledges it with a corresponding COMMIT message. The COMMIT has to be certified by means of the counter $\mathfrak{O}$ and the value $[0|50]$ as well, increasing the counter values also on the follower's end. When a replica has collected a quorum of valid acknowledgments, including the PREPARE from the leader and its own message, it is allowed to execute the request a as long as it has already executed all requests proposed with order numbers lower than 50. All replicas subsequently answer the request by sending a REPLY message containing the result of the execution to the client (not depicted). The client accepts a result only when it has collected $f + 1$ matching replies.

5.2.2 Checkpointing

As is the case for other protocols [13, 15, 45], the network is assumed to be unreliable; consequently the assumed but not a priori known upper bound for message delays only holds if a message is sent sufficiently often. It cannot be known in advance, how often this is, and since Byzantine processes are fail-silent, it may even not be known if an expected response has not yet arrived because a process is faulty or because the network or the process is simply slow [23].

To resend messages when required, the replication protocols considered here keep a log of all ordering messages for a predefined number of consecutive consensus instances between so-called *low and high water marks*; to not run out of memory, they discard older messages once they have proof that enough replicas were able to execute the commands ordered by instances up to a certain order number and that their own service state at that point is correct. To create such proof, replicas regularly invoke the checkpointing protocol.

Since Hybster does not need to preserve complete histories of outgoing messages such as MinBFT, it can employ a single-phase checkpointing protocol similar to PBFT and A2M-PBFT. In the example given in Figure 3, ② replicas conduct a checkpoint after executing request a at order number 50. They save the current service state and announce this by sending a Checkpoint message to each other. Once a replica has collected a quorum of these messages, a checkpoint becomes *stable*. It cleans up its log by discarding order messages for instances up to and including 50 and Checkpoints and checkpoint states related to instances up to but excluding 50. As in other replication protocols, messages of the view-change protocol are not affected by this procedure.

Strict Ordering Window. Besides carrying out garbage collection, a replica also adjusts the low and high water mark upon reaching a stable checkpoint at an order number o. The low mark is set to o and the high mark to o plus some constant value. Messages for instances below the low water mark are discarded as described above. Moreover, a replica will not participate in any consensus instance with an order number greater than the high water mark. This essentially establishes a sliding *ordering window* of ongoing consensus instances that is advanced in checkpoint intervals. Note that such a window is effectively employed by all considered protocols for one reason or another. However, Hybster strictly adheres to this window even during a view-change.

State and Return Value Confirmation. If a replica r requires messages for consensus instances that other replicas already discarded, it has to update its service state directly. Since correct replicas only clean up their logs after collecting a quorum certificate for a checkpoint at o, there is at least one correct replica that can provide the state reached after executing o. To proof its correctness, Checkpoint messages contain the digest of this state. Moreover, the digest has to include a vector of return values containing an entry for the last requests of each client represented in the service state. When the fallen-behind replica r updates its service state, it skips some instances $o^* \leq o$ and never executes the corresponding requests. To ensure that clients can nevertheless collect enough matching Reply messages, replica r must also safely learn the return values for the instances it missed.

Trusted MAC Certificates. Since all correct replicas will reach the same state after executing a particular order number o and since the quorum certificates for checkpoints only have to confirm the digest contained in Checkpoint messages, checkpoints are not subject to equivocation. Thus, it is sufficient to equip Checkpoint messages with a trusted MAC certificate by means of some counter $\mathfrak{M}$.

5.2.3 View-Changes

If the current leader is suspected to be faulty or fails to get a quorum for its proposals, the current view has to be aborted and a new view has to be found in which a sufficient number of replicas follows the leader such that the system is again able to make progress, that is, able to order and execute issued requests. It is the responsibility of the view-change protocol to find such a view and to ensure that if a correct replica executed a request with an order number o in a previous view, no correct replica will ever execute a different request with that order number.

Following the analysis of existing protocols given in Section 4.4, the challenge of a view-change protocol based on a hybrid fault model is to ensure this with only a single replica in the intersection of any two quorums, a single replica that is potentially faulty. The view-change protocol of Hybster meets this challenge through three mechanisms:

Continuing Counter Certificates. Correct replicas execute a request with an order number o only upon having collected a committed certificate for an instance (v, o). That is, a quorum of replicas must have sent a correct order message certified with the value $[v|o]$ of the trusted counter $\mathfrak{O}$. Let (v, o_act) be the last instance a replica r has actively participated in by sending an order message. If this replica wants to support a new leader l' of a view $v + 1$, it has to send a message called View-Change containing the Prepares of all instances of its current ordering window in v. Even if replica r is faulty, by using a continuing certificate of the form $\tau(r, \mathfrak{O}, v + 1|0, v|o\_act)$, it is forced to include all Prepares up to o_act. Additionally, it is prevented from sending further order messages for the view v after creating the View-Change for $v + 1$. Thus, and due to the minimal intersection of any two quorums, l' is able to determine all proposals of the former leader l that have been potentially committed and hence executed at some correct replica in v by collecting a quorum of such View-Change messages.

A correct l' in Hybster considers a View-Change of a replica r for this quorum only if its own ordering window has at least reached the position of the window indicated by the View-Change, that is, if l' has the necessary resources to process contained Prepares in the first place. If l' requires additional ordering and checkpointing messages or even the service state to forward its window accordingly, it requests this state from r via the state-transfer protocol. As long as l' cannot adapt its window, it will not consider the message from r. Since the system model assumes that there is always a quorum of correct replicas which would provide the requested state, l' will nevertheless be able to collect the required quorum of View-Change messages.

Following, l' transfers all proposals of l that are contained in its (then current) ordering window to the view $v + 1$ by reproposing the same assignments of requests to order numbers within new Prepares for the new view. Consequently, the quorum of View-Changes a new leader uses to determine the initial state for a new view is called *new-view certificate*. l' proves the correctness of the transition from view v to view $v + 1$ to other replicas by sending a New-View containing the new-view certificate and all new Prepares.

View-Change Certificates. However, there is no guarantee that l' will provide a correct New-View in time. l' could be faulty or lack the required support from a quorum of replicas. In that case, a replica r has to send another View-Change to support the leader l'' of view $v + 2$ and this View-Change message can only be equipped with the certificate $\tau(r, \mathfrak{O}, v + 2|0, v + 1|0)$. That is, since creating the View-Change for view $v+1$ increased the counter $\mathfrak{O}$ from $[v|o\_act]$ to $[v+1|0]$, the replica r, if faulty, could pretend that it never participated in any instance in view v.

In this situation, contrary to other hybrid replication protocols, Hybster does not try to force replica r to reveal the proposals of a view v, for example, by demanding replica r to include a history of its generated View-Change messages between v and $v+2$, or more general v and $v+x$. In the partially synchronous system model it cannot be known how many rounds are needed to elect a new leader, thus how large x will become and how much state the replicas have to keep until they reach a new stable view. Instead, Hybster ensures that the possibly only correct replica r_c in a quorum of replicas supporting a leader l^x knows about all proposals potentially committed in views before $v+x$.

To achieve this, a replica r_c that followed the leader l of view v and currently supports the leader l' of view $v+1$ will only send a View-Change message to support leader l'' of view $v+2$ when it collected a quorum of View-Change messages for the view $v+1$ itself. Even if r_c never received any order message for any instance in v directly, this quorum of View-Change messages, called *view-change certificate*, is guaranteed to contain all relevant Prepares; r_c learns the potentially committed proposals of leader l afterwards and is able to propagate them in View-Change messages for views greater than $v+1$. In Hybster, no correct replica that followed a leader l of a view v will support any leader l^* of a view $v^* > v+1$ until it has collected or been provided with a view-change certificate for the view v^*-1. The certificate is guaranteed to contain all potentially committed proposals from views before v^*-1. In conjunction with continuing counters as an anchor, this mechanism works by induction.

New-View Acknowledgments. So far, the only situations considered were those in which all replicas of a quorum Q that support a leader l' of a view $v+1$ followed the same previous leader l of a view v. Provided that there was a view $v^- < v$ with the leader l^- where instances were ordered and committed. Now it can be the case that all replicas in Q except of one replica r collected a view-change certificate for view v but did not receive the New-View message for v in time. The View-Change messages for view $v+1$ of these replicas thus only contain the proposals from view v^-. Replica r, however, received the New-View for v and consequently included the Prepares of view v in its View-Change for view $v+1$. In such a situation, it would be not safe for the designated leader l' to determine the initial state for view $v+1$ only on the basis of the View-Change of replica r. The equivocation prevention mechanism of Hybster guarantees that there are no conflicting Prepares in view v. The continuing counter certificates guarantee that r cannot withhold relevant Prepares of view v. The view-change certificates guarantee that all relevant Prepares of views before v are propagated and contained in Q. Without further mechanism there is no guarantee, however, that the Prepares of view v are valid in the first place, that leader l has ever compiled a correct New-View and thereby properly transferred all relevant proposals to view v. View-change certificates ensure that at least all committed proposals are included but they can also contain additional proposals that did not reach a quorum; whether they are transferred to later views depends on which View-Change messages are used by a new leader in its new-view certificate.

In other words, it can be always the case that there are two Prepares proposing different requests for one and the same order number but in different views. For that reason, a new-view certificate in Hybster must also contain an acknowledgment of at least $f+1$ replicas that they accepted a correct New-View for the view v with which a new leader l^* determines the initial state for a view $v^* > v$. These *new-view acknowledgments* guarantee that the view v was indeed a properly established view. The acknowledgments can be gathered in two forms: All View-Changes include a view number v_from that indicates the last view a replica had accepted. Due to the required counter certificates, a replica r is only able to send a sole View-Change to support a designated leader l^*. Hence, it could be the case that l^* collects enough View-Change messages supporting it but will never be able to compile a new-view certificate since these View-Change messages do not include $f+1$ matching v_from. To prevent this, a replica r that accepts a New-View v after it has already aborted v by sending a View-Change for a view $v^* > v$ sends an explicit acknowledgment for v in the form of a New-View-Ack including all Prepares the replica learned with the New-View. Since this is only needed to ensure that all proposals of views before v are propagated by at least one correct replica, a New-View-Ack does not require a particular counter certificate; faulty replicas could simply omit sending such a message anyway.

Example. We proceed with the running example given in Figure 3 to illustrate how the view-change protocol of Hybster works: ② After reaching the checkpoint at order number 50, all replicas have discarded all ordering messages, have saved a snapshot of the current service state including the vector of return values, and possess a quorum certificate K_{50} of Checkpoint messages proving the correctness of this state. Let us assume that a client tried to issue a request b by sending it to the current leader R_0, but that the message was lost during the transmission (not depicted). Since it did not receive any reply, it resends request b. The client does not know if a faulty leader simply refused to propose request b, thus, this time, ③ it multicasts the message to all replicas in the group. Replica R_0 is correct and proposes request b once received in consensus instance $(0, 51)$ and obtains a committed certificate together with replica R_1.

Replica R_2, however, was temporarily disconnected from the group and did not receive any ordering message. ④ Therefore it is not able to collect a quorum certificate for request b within some time and suspects leader R_0 to be faulty or not supported by a quorum. By sending a View-Change message, R_2 announces that it aborted the view $v\_from = 0$ and wants to enter view $v\_to = 1$ with leader R_1. The message is certified with $\tau(R_2, \mathfrak{D}, 1|0, 0|50)$ and does not (need to) contain any Prepare. The previous counter value $[0|50]$ has the same value as the last stable checkpoint of R_2.

Upon receiving the View-Change from R_2, replica R_1 starts to behave faulty, trying to conceal request b executed by the correct replica R_0. For that purpose, it generates its own View-Change. However, because it participated in instance $(0, 51)$, the certificate for the View-Change can only be $\tau(R_1, \mathfrak{D}, 1|0, 0|51)$ (otherwise it would not be accepted) and the unforgeable previous counter value $[0|51]$ forces R_1 to include the Prepare for $(0, 51)$. Therefore, it does not send the View-Change to R_2, it merely generates a New-View for view 1, for which it is the leader. This New-View needs to be backed by a quorum of View-Change messages with $v\_from = 0$ and $v\_to = 1$, which ensures that the New-View must contain a new Prepare that reproposes request b for order number 51 in view 1.

⑤ Creating the Prepare increases R_1's counter to $[1|51]$. Now it pretends that it was not able to reach a stable view 1 (which could actually be detected as incorrect in this example, but does not need to be) or 2 and wants to enter view 3. R_1 sets its counter to $[2|0]$ and creates a View-Change with $v\_from = 0$, $v\_to = 3$, and a certificate $\tau(R_1, \mathfrak{D}, 2|0, 3|0)$. The certificate is valid and R_1 successfully cleans its counter value. However, R_2 will not act on the cleaned but valid View-Change from R_1. It will not leave its unstable view 1 before it receives another View-Change with $v\_to = 1$ to form a view-change certificate. ⑥ The missing View-Change is eventually provided by R_0 which aborts view 0 due to having received more than f messages from other replicas for view 1. The proposal of request b must be contained in this View-Change (otherwise it could not have been committed in view 0) and R_2 learns the assignment from b to order number 51.

To create a valid New-View for view 2, the replica R_2 needs q View-Change messages with $v\_to = 2$ and more than f messages with matching v_from. ⑦ When replica R_0 sends a View-Change to support R_2 as the leader, R_2 does not receive this message in time and the faulty replica R_1 tries to hinder the correct replicas from reaching a stable new view. In order to do so, R_1 sends the New-View for view 1 it created before and this New-View is installed by R_2. ⑧ Following, replica R_2 leaves view 1 for view 2. Now, the View-Changes from R_0 and R_2 do not suffice to create a new-view certificate for view 2 with a $v\_from = 1$. The certificate would not contain proof that view 1 had been established properly. ⑨ However, when R_0 learns and installs the New-View for view 1, a view it has already aborted, it sends a New-View-Ack as a supplement for its View-Change from 0 to 2. Using the acknowledgment and the two View-Change messages from R_0 and R_2, replica R_2 is able to create a valid New-View for view 2 that contains the proposal of requests b for order number 51. ⑩ Once replica R_0 receives the New-View and enters view 2, it acknowledges the proposed assignment which finally allows replica R_2 to execute the request.

Closing Remarks. To take up the discussion of Section 4, the view-change certificates allow Hybster to rely on single correct replicas within quorums to announce consensus instances that have potentially been executed by some replicas that are not a member of the quorum in question. In that sense, Hybster is able to establish a situation that is naturally given for traditional BFT protocols like PBFT by their condition of a correct replica in the intersection of two quorums. Unlike PBFT, replicas in Hybster do not need an additional phase during the ordering to collect a quorum certificate for this very announcement. Thanks to the equivocation prevention in Hybster (and in conjunction with new-view acknowledgments), a single Prepare from two-phase ordering suffices; this would not be the case if Hybster were based on an equivocation detection mechanism. The design of Hybster's view-change protocol not only allows limiting the number of required view-change messages even in phases of asynchrony induced, for instance, by faulty or simply overloaded networks. The mechanism of adjusting the own ordering window that a new leader uses to process View-Change messages for its new-view certificate is applied for all steps in the view-change protocol of Hybster. This explicitly limits the number of ordering and checkpointing messages that have to be processed and stored by replicas. In sum, replicas in Hybster are not at risk of running out of memory merely because conditions have become less optimal.

5.3 COP Inside

Based on a consensus-oriented parallelization scheme [9], replicas are composed of equal processing units, called *pillars*, that do not share state, communicate via asynchronous in-memory message passing only, and operate mostly independently from each other. Concerning the ordering of requests, the latter is achieved by partitioning the space of order numbers; each pillar is responsible for executing the consensus instances for a predefined share of order numbers given by some calculation formula.

This would not be very effective using the basic protocol of Hybster. As with other hybrid protocols, it is strictly sequential; a replica can only actively participate in a consensus instance for an order number o' if it has already processed all instances with $o < o'$. Binding messages of protocol phases a priori to values of trusted counters that can only increase directly entails this serialization. Even if distributed over multiple pillars, the pillars would have to address the sequential dependencies among instances. To circumvent the limitation of its basic protocol, Hybster allows replicas to certify messages for particular protocol phases with different trusted counters and hence enables them to split the work among multiple independent pillars each equipped with its own instance of the trusted subsystem TrInX.

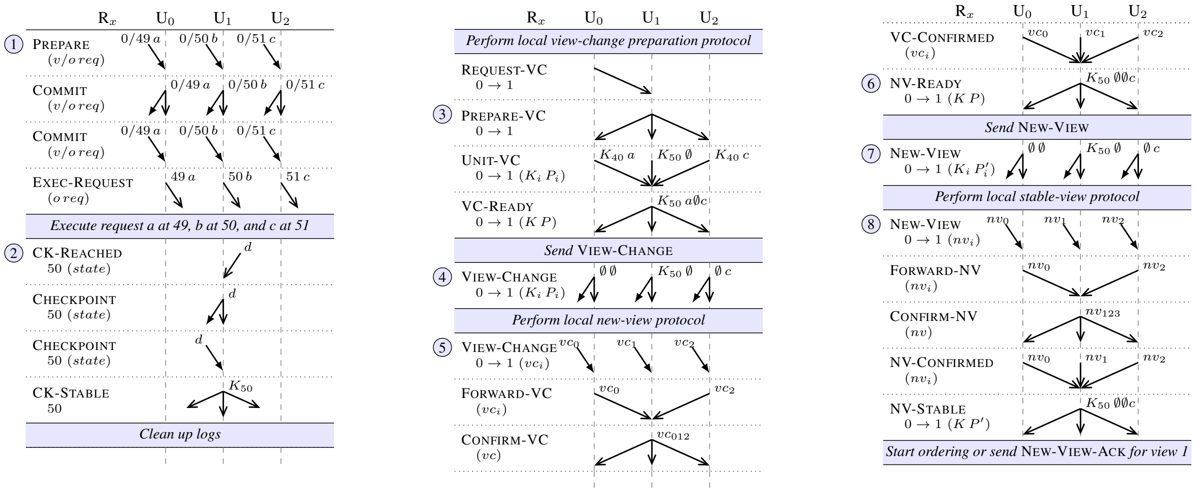

R_x U_0 U_1 U_2
1 PREPARE (v/o req) 0/49 a 0/50 b 0/51 c
COMMIT (v/o req) 0/49 a 0/50 b 0/51 c
COMMIT (v/o req) 0/49 a 0/50 b 0/51 c
EXEC-REQUEST (o req) 49 a 50 b 51 c
Execute request a at 49, b at 50, and c at 51
2 CK-REACHED 50 (state) d
CHECKPOINT 50 (state) d
CHECKPOINT 50 (state) d
CK-STABLE 50 K50
Clean up logs
Perform local view-change preparation protocol
REQUEST-VC 0 → 1
3 PREPARE-VC 0 → 1
UNIT-VC 0 → 1 (K_i P_i) K40 a K50 ∅ K40 c
VC-READY 0 → 1 (K P) K50 a∅c
Send VIEW-CHANGE
4 VIEW-CHANGE 0 → 1 (K_i P_i) ∅ ∅ K50 ∅ ∅ c
Perform local new-view protocol
5 VIEW-CHANGE 0 → 1 (vc_i) vc0 vc1 vc2
FORWARD-VC (vc_i) vc0 vc2
CONFIRM-VC (vc) vc012
VC-CONFIRMED (vc_i) vc0 vc1 vc2
6 NV-READY 0 → 1 (K P) K50 ∅∅c
Send NEW-VIEW
7 NEW-VIEW 0 → 1 (K_i P'_i) ∅ ∅ K50 ∅ ∅ c
Perform local stable-view protocol
8 NEW-VIEW 0 → 1 (nv_i) nv0 nv1 nv2
FORWARD-NV (nv_i) nv0 nv2
CONFIRM-NV (nv) nv123
NV-CONFIRMED (nv_i) nv0 nv1 nv2
NV-STABLE 0 → 1 (K P') K50 ∅∅c
Start ordering or send NEW-VIEW-ACK for view 1

Figure 4. The parallelized ordering, checkpointing, and view-change of Hybster as seen by a replica R_x with three pillars U_i.

5.3.1 Independent Ordering

In Hybster, a replica r is allowed to create message certificates of the form $\tau(r(u), \dots)$ where $r(u)$ denotes the ID of the TrInX instance of the pillar u in replica r.

Predefined Consensus Assignment. To prevent faulty replicas from creating conflicting order messages by using different TrInX instances, Hybster exploits the fact that with a consensus-oriented parallelization, consensus instances are distributed over pillars in a predefined manner; it is known a priori which pillar is responsible for which instance. Thus, an order message of a replica r is only accepted if the attached message certificate $\tau(r(u), \mathfrak{O}, v|o, -)$ is issued by the TrInX instance associated with the pillar u that is responsible for the order number o. The number of pillars a replica has and the instance IDs of their associated TrInX instances are part of the general configuration of a replica group in Hybster, and are thus equally known to all replicas.

Example. As depicted in Figure 4 from the perspective of a follower R_x, by binding the association of order numbers, trusted subsystem instances, and pillars to a particular configuration of the replica group, the parallelized ordering protocol of Hybster remains very similar to the basic variant: ① The leader l of a view $v = 0$ proposes requests a, b, and c in parallel with order numbers o = 49, 50, and 51 in its pillars U_0, U_1, and U_2. For this, l issues PREPARE messages equipped with certificates of the form $\tau(l(U_i), \mathfrak{O}, v|o, -)$. To be accepted by other replicas, o must be an order number that is associated to the pillar U_i. The pillars of R_x can process the proposals independently and once a pillar collects a committed certificate for an instance, it sends an internal EXEC-REQUEST to the execution stage, which, in turn, ensures that all requests are delivered to the service implementation in accordance with their assigned order numbers. As described in [9], gaps in the sequence of order numbers that can occur due to independent ordering are closed by proposing empty consensus instances.

5.3.2 Shared Checkpointing

To keep the pillars in step, the checkpointing in a design with a consensus-oriented parallelization scheme is not carried out separately by each pillar but shared among them. Which pillar has to execute the protocol for the kth checkpoint is determined in a round-robin fashion to distribute the additional load among all available pillars. As Hybster's design allows using a checkpointing protocol that is akin to traditional protocols not based on a hybrid fault model, the parallelized variant of Hybster does not need to apply additional means to ensure the safety of this protocol.

Example. In the example given in Figure 4, ② a checkpoint is reached after executing the requests assigned to the order number 50 and the execution stage sends the digest calculated over service state and return value vector to the responsible pillar for this checkpoint, here U_1. Subsequently, pillar U_1 carries out the instance of the checkpointing protocol. Once it has collected a quorum of CHECKPOINT messages K_{50}, it informs all other pillars and the execution stage in order to allow them to perform garbage collection.

5.3.3 Distributed View-Changes

In the parallelized variant of Hybster, information about ongoing consensus instances is distributed across pillars and each replica possesses multiple trusted counters $\mathfrak{O}$ for certifying order messages. The replica-internal view-change nevertheless has to ensure that faulty replicas are forced to reveal all supported consensus instances and that they cannot participate actively in views they have already aborted.

Split External Messages. To achieve this by still maintaining the access to instances of the trusted subsystem local to pillars, Hybster splits the view-change messages, namely VIEW-CHANGE, NEW-VIEW, and NEW-VIEW-ACK, resulting in one partial message for each pillar. Still, replicas are only allowed to abort a view completely, that is, with all of their pillars. Therefore, receiving replicas consider a view-change message only, upon receiving all of its parts, with the number of parts given by the number of pillars as configured for a particular replica. Moreover, as for the ordering, partial messages must only contain Prepare messages for order numbers the issuing pillar is responsible for. In doing so, it can be ensured that the contained sets of Prepare messages are complete. Due to the parallelized ordering, the combination of, taken by themselves, complete Prepare sets, however, can contain gaps. Therefore, a designated leader must fill the gaps by proposing empty consensus-instances similar to the approach of PBFT [13].

Example. As illustrated in Figure 4, the view-change of a replica R_x comprises three local protocols, that are coordinated by one of the pillars, here U_1: (1) The *local view-change preparation protocol* ③ collects the information about all ongoing consensus instances and ④ eventually instructs all pillars to send their part of the View-Change message. In the example, pillars U_0 and U_2 have not yet received the information that checkpoint 50 became stable but they learn the checkpoint during the view-change preparation and adapt their ordering window before they send their View-Change. (2) ⑤ Incoming View-Change or New-View-Ack messages are verified by means of the *local new-view protocol*. ⑥ Once a designated leader of a new view collected a new-view certificate, ⑦ it creates and sends the pillar-specific parts of the New-View message. (3) Finally, ⑧ a received New-View has to be verified and all pillars have to be informed if a new stable view is reached. This is the responsibility of the *local stable-view protocol*.

6. Evaluation

We implemented Hybster in a mainly Java-written prototype. The trusted subsystem TrInX is realized on the basis of Intel SGX, written in C/C++, and connected to the Java world via the Java Native Interface (JNI).

Subjects. For the evaluation, we use two different configurations of Hybster. In one configuration we use only a single pillar for the ordering of requests and supplement it with an additional thread for the execution stage and multiple threads for the client handling. This configuration uses a single instance of TrInX and measures the sequential basic protocol of Hybster. It resembles a design with a traditional parallelization scheme, albeit realized in a very efficient and optimized way. We denote this configuration with *HybsterS*. In contrast, with *HybsterX* we refer to a full configuration of Hybster that uses as many threads and TrInX instances as processor cores are available. In addition, we implemented PBFT following the consensus-oriented parallelization scheme on the same code base in order to compare it with Hybster. Besides the standard configuration that uses authenticators for the certification of messages [13], our implementation of *PBFTcop* is also able to make use of TrInX where it certifies messages with signature-like trusted MACs (see Section 5.1). We call this configuration *HybridPBFT*. Neither MinBFT [45], nor A2M-PBFT [15] are directly considered. It can be argued that HybsterS will always achieve at least the performance of these protocols. For instance, MinBFT has to process all incoming messages in-order, A2M-PBFT uses one additional ordering phase, and only HybsterS can make use of a rotating leader [9, 43].

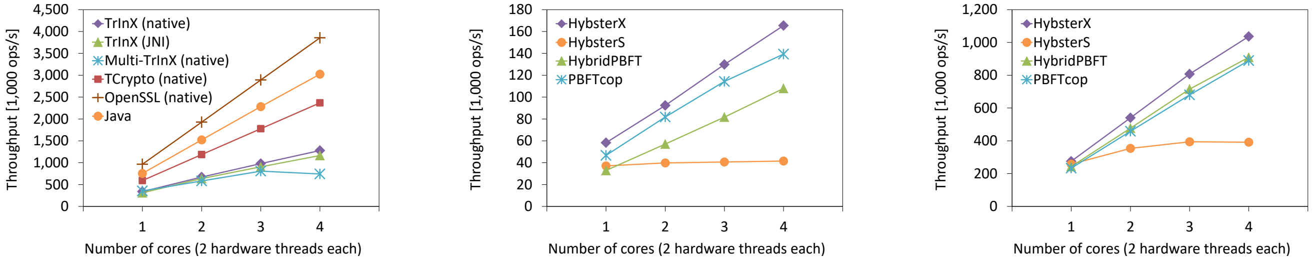
Setup. We deploy all considered configurations on a cluster of six machines, each equipped with an Intel Core i7-6700 CPU comprising four cores running at 3.4 GHz with Hyper-Threading enabled and Turbo Boost disabled and with four NICs connected via 1 Gb switched Ethernet. The software environment of the machines includes a Linux 3.19 kernel, an OpenJDK 1.8, and the Intel SGX SDK in version 1.6.

In the case of HybsterS and HybsterX, three replicas are used each hosted by one of the machines. HybridPBFT and PBFTcop require four replicas and thus four machines. Two machines are dedicated to run the client implementation that generates the workload. For that purpose, a configured number of clients constantly issues a bounded number of asynchronous requests. Clients measure the time it takes to collect a sufficient number of replies from the replica group to calculate the average latency and throughput. In addition, we monitor the CPU and network usage on all machines. SHA-256 is employed for all cryptographic operations. The presented results are the average of three runs of the benchmark in question, where each run is conducted over 120 seconds.

6.1 Trusted Subsystem

Before we investigate the performance of the different protocol configurations, we evaluate our trusted subsystem TrInX. For that purpose, we first implemented TrInX as described throughout this paper; each instance runs in its own trusted execution environment provided by Intel SGX, is dedicated to a single thread, and can be used either natively or by Java processes via JNI. Second, we implemented a variant called Multi-TrInX which hosts multiple TrInX instances within a single trusted execution environment accessed by all running threads. Finally, we compare the SGX-secured versions against plain, insecure library implementations based on the cryptography library of the SGX SDK named TCrypto, on OpenSSL, and on a pure Java implementation. We are mainly interested in how the variants scale when multiple processor cores independently certify messages.

Figure 5a shows the throughput achieved by the considered variants for the certification of 32 byte messages when the number of available cores is increased and both hardware threads of each core are used. As expected, the insecure library implementations scale perfectly since the test does not entail any synchronization among the threads. Less expected is that the native library of the SGX SDK, TCrypto, is 20 % slower than the pure Java version in this scenario and 40 % compared to OpenSSL. One explanation is that the current SDK lacks support for special processor instructions accelerating cryptographic operations (AES-NI) [7]. Other measurements show that TCrypto can overtake the Java variant slightly for larger messages; OpenSSL remains out of



(a) Trusted subsystem

(b) 0 bytes, unbatched, rotation

(c) 0 bytes, batched, rotation

Figure 5. Throughput of trusted subsystem for certifying 32 byte messages (a) and of protocols with rotating leader (b, c).

range. The secured standard TrInX implementation scales quite well, although not perfectly, up to 1.3 million certification operations per seconds (ops/s) when natively accessed simultaneously on four cores. The loss incurred by accessing it via JNI is relatively small. The Multi-TrInX version performs comparable to the others up to three cores with six threads but falls back using four cores. Since we took care that counters are not placed on the same cache line, an explanation could be that entering the same trusted environment by different threads incurs some synchronization overhead on the SDK or processor level. In any case, multiplying the subsystem instead of extending it is indeed the better alternative. Borrowing the numbers for the state-of-the-art trusted subsystems, the FPGA-based CASH from [28] where the certification of 32 byte messages with SHA-256 was measured with 57 microseconds, the advantage is clearly on the side of TrInX: CASH achieves 17,500 certifications per seconds whereas we measure 240,000 for a single instance of TrInX. More important, even if it supported an arbitrary number of counters, CASH can only be connected by a single channel, TrInX instead scales by multiplication.

6.2 Multiple Cores and Scalability

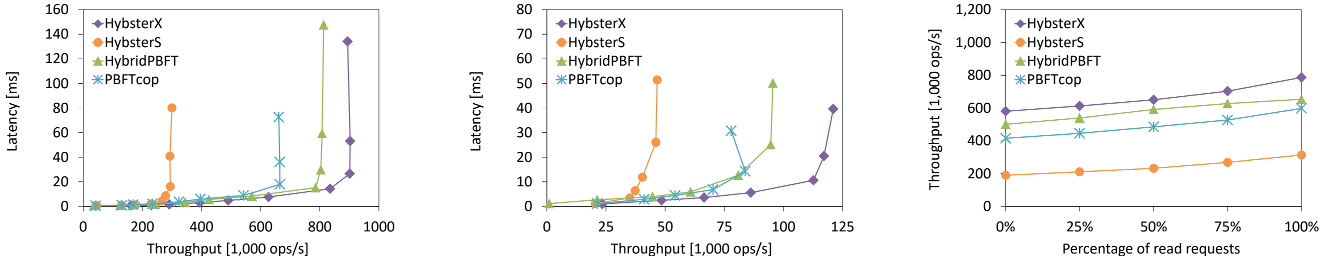
Our first benchmark of the aforementioned protocol configurations determines the maximum achievable throughput in our setup with an increasing number of available processor cores. We deploy all protocols with a microbenchmark service that returns empty results without any calculation.

The results for configurations with a rotating leader are presented in Figure 5b and Figure 5c; in the former case single consensus instances order single requests and in the latter case batches of requests. With a maximum number of only four cores, all measurements are CPU-bound, the network remains underutilized even when a fixed leader is used. As can be seen, HybsterS is the only configuration confined by a sequential ordering protocol. Without batching, it can hardly exceed 40,000 ops/s, with batching, it starts with 260,000 and reaches its limit with roughly 400,000 ops/s. Making use of a consensus-oriented parallelization, PBFT attains up to 140,000 unbatched ops/s and 890,000 with batching enabled. Its hybrid variant, HybridPBFT, is about 30 % slower when one consensus instance for each request is required and thus a lot of small messages have to be processed. Employing authenticators, PBFT calculates for a protocol phase about three cryptographic hashes for one outgoing and three hashes for incoming messages (depending on the thread configuration 1.5 to 2.6 µs for a 32 byte message). HybridPBFT requires in total only four instead of six hash operations but needs to enter the secure mode (2.4 µs for the mode switch + 0.3 µs for JNI) and suffers from a slow hash implementation (1.6 to 3.4 µs for 32 bytes); the smaller and stronger certificates are irrelevant for this test. However, HybridPBFT catches up when batches are ordered. Relying on three replicas, HybsterX requires a total of three hash operations and can make use of a two-phase ordering opposed to the three phases in (Hybrid)PBFT. With 15 to 20 %, its advantage compared to PBFT is however limited; also HybsterX has to use the cryptographic library of the current SGX SDK. However, with 165,000 unbatched and 1,040,000 batched ops/s, it achieves a speedup of 2.5 to 4 compared to its sequential but still rotating basic protocol. The to our knowledge highest published numbers for batched throughput of hybrid protocols so far: 63,000 ops/s for MinBFT and 72,000 ops/s for CheapBFT (measured on quad-core machines with 2.3 GHz connected via 1GbE) [28]. In addition, HybsterX is the first hybrid protocol able to scale; the speedup factors in our batched setup for four cores compared to one core: 3.77 with and 3.91 without rotating the leader.

6.3 Payloads and Latency

Next, we evaluate the average response time of all protocol configurations and how it behaves with an increasing workload and different payloads for requests and replies. Figure 6a shows the outcome of the measurements for all configurations without any payload and Figure 6b with a payload of 1 kilobyte for requests as wells as replies. Batching is enabled and rotation is disabled for both. Benchmarks with 128 bytes and 4 kilobytes yielded similar results.

Without payload, all protocol configurations start with a minimum latency of roughly 500 to 600 microseconds for sending a request, ordering it, and sending back at least two $(f+1)$ replies. HybsterX is able to take full advantage of its two-phase ordering and exhibits approximately 20 % lower latencies than its competitors (four vs. five message delays in sum). HybsterS is limited by its sequential execution, the latencies are comparable to PBFT and HybridPBFT that require one additional ordering phase. As expected, this holds until the protocols reach their individual saturation: Hyb-



(a) 0 bytes, batched, no rotation

(b) 1 kilobyte, batched, no rotation

(c) Coordination service (128 bytes)

Figure 6. Microbenchmark with different payloads (a, b) and coordination service with varying read/write rate (c).

sterS has its limit at 310,000, PBFT at 660,000, HybridPBFT at 810,000, and HybsterX at 900,000 ops/s. The numbers for the benchmark with 1 kilobyte payload are lower but comparable. One difference is that the network becomes an additional limiting factor in this test while the measurements without payload are completely CPU-bound. This explains why it takes longer from the first indication of a saturation until the saturation actually sets in.

6.4 Coordination Service

To conclude our evaluation, we now consider a replicated, ZooKeeper-inspired coordination service [27]. The service offers a hierarchical namespace of nodes and the possibility of creating such nodes, to destroy them, and to store data within them. Together with other functions, this API can be employed by groups of clients to implement coordination tasks. Unlike ZooKeeper, the coordination service considered here does not make use of read-optimizations, and thus provides strong consistency. We measure the throughput for a setup in which clients saturate the system by storing and retrieving nodes with 128 bytes of data. We vary the proportion of read operations; an increasing read rate leads to a lower number of large requests but to a higher number of large replies. We do not use rotation in this experiment, therefore, a single replica has to propose all client requests.

The results depicted in Figure 6c confirm the findings of the previous microbenchmarks. HybsterX attains a 10 to 20 % higher throughput than the hybrid PBFT configuration and 30 to 40 % compared to the original PBFT protocol realized with a consensus-oriented parallelization scheme. Again, compared to its own sequential basic protocol, it is able to exhibit a speedup of 2.5 to 3.0 and is only confined by the few number of cores provided by the test machines.

7. Related Work

There is a large body of work aimed at minimizing the overhead associated with Byzantine fault-tolerant state-machine replication in general, and its resource overhead in particular. In this context, many of the proposed approaches have concentrated on reducing implementation costs [14, 24], communication steps [19, 26, 30] or execution delays [21, 29, 31], but have not minimized the number of replicas a BFT system consists of. As a first step towards this goal, Yin et al. [48] presented a BFT architecture that separates agreement from execution and relies on dedicated clusters of replicas to handle each of these tasks. This allows the minimum number of execution replicas to be as low as $2f + 1$.

While reducing the number of execution replicas below $3f+1$ is possible by changing the structural organization of a BFT system, reducing the minimum number of replicas that participate in the ordering of client requests requires parts of the system to be trusted [15, 17, 18, 20, 22, 28, 37, 42, 44, 45]. Depending on the particular approach, a trusted subsystem may include an entire virtualization layer [22, 37, 44], a multicast ordering service executed on a hardened Linux kernel [17, 18], a centralized configuration service [42], or a trusted log [15], or may be as small as a trusted platform module [44, 45], a smart card [32], or an FPGA [20, 28]. A key insight in this regard is that subsystems with smaller trusted computing bases are less likely to fail arbitrarily (e.g., as the result of a successful attack) and consequently more likely to justify the trust placed in them. On the other hand, small trusted subsystems such as trusted platform modules and smart cards tend to have limited performance capabilities compared with their larger counterparts [28] and therefore pose the danger of becoming a bottleneck.

The problem of parallelizing state-machine replication has previously been investigated in the context of both crash tolerance [25, 33, 39] as well as Byzantine fault tolerance [9, 29, 31]. Hybster is the first highly parallelizable replication protocol for hybrid systems, a domain in which all protocols proposed so far require some kind of sequential processing.

8. Conclusion

Hybster is a new state-machine replication protocol based on a hybrid fault model that has been designed after a thorough analysis of existing proposals. It unifies their strengths and addresses their weaknesses by taking different routes when it comes to ensuring safety and exploiting the potential of most recent multi-core processors. Hybster is relaxed with regard to faulty replicas while remaining safe; it is parallelizable by embracing the consensus-oriented parallelization scheme and marrying it with trusted execution environments; it is formal, it is backed by a comprehensive specification [8] that facilitates reasoning about it; and Hybster is simply fast, it achieves over 1 million operations per second in a setup with only four cores, excelling other published systems by more than an order of magnitude.

References

[1] http://www.businessinsider.com/amazons-cloud-can-handle-1-million-transactions-per-second-2012-4.

[2] https://gigaom.com/2011/12/06/facebook-shares-some-secrets-on-making-mysql-scale.

[3] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proceedings of the 20th Symposium on Operating Systems Principles (SOSP '05)*, pages 59–74, 2005.

[4] Y. Amir, B. Coan, J. Kirsch, and J. Lane. Byzantine replication under attack. In *Proceedings of the 38th International Conference on Dependable Systems and Networks (DSN '08)*, pages 197–206, 2008.

[5] ARM. Security technology building a secure system using TrustZone technology (white paper). *ARM Limited*, 2009.

[6] P.-L. Aublin, S. B. Mokhtar, and V. Quéma. RBFT: Redundant Byzantine fault tolerance. In *Proceedings of the 33rd International Conference on Distributed Computing Systems (ICDCS '13)*, pages 297–306, 2013.

[7] J. Aumasson and L. Merino. SGX Secure Enclaves in Practice – Security and Crypto Review. https://www.blackhat.com/docs/us-16/materials/us-16-Aumasson-SGX-Secure-Enclaves-In-Practice-Security-And-Crypto-Review.pdf, 2016.

[8] J. Behl, T. Distler, and R. Kapitza. Hybster — A highly parallelizable protocol for hybrid fault-tolerant service replication. http://publikationsserver.tu-braunschweig.de/get/64440.

[9] J. Behl, T. Distler, and R. Kapitza. Consensus-oriented parallelization: How to earn your first million. In *Proceedings of the 16th Middleware Conference (Middleware '15)*, pages 173–184, 2015.

[10] A. Bessani, J. Sousa, and E. Alchieri. State machine replication for the masses with BFT-SMaRt. In *Proceedings of the 44th International Conference on Dependable Systems and Networks (DSN '14)*, pages 355–362, 2014.

[11] M. Castro. *Practical Byzantine Fault-Tolerance*. PhD thesis, MIT, 2001.

[12] M. Castro and B. Liskov. A correctness proof for a practical Byzantine-fault-tolerant replication algorithm. Technical report, Cambridge, MA, USA, 1999.

[13] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, pages 173–186, 1999.

[14] M. Castro, R. Rodrigues, and B. Liskov. BASE: Using abstraction to improve fault tolerance. *ACM Transactions on Computer Systems*, 21(3):236–269, 2003.

[15] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *Proceedings of 21st Symposium on Operating Systems Principles (SOSP '07)*, pages 189–204, 2007.

[16] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI '09)*, pages 153–168, 2009.

[17] M. Correia, N. F. Neves, L. C. Lung, and P. Veríssimo. Worm-IT – A wormhole-based intrusion-tolerant group communication system. *Journal of Systems and Software*, 80(2):178–197, 2007.

[18] M. Correia, N. F. Neves, and P. Veríssimo. How to tolerate half less one Byzantine nodes in practical distributed systems. In *Proceedings of the 23rd Symposium on Reliable Distributed Systems (SRDS '04)*, pages 174–183, 2004.

[19] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 177–190, 2006.

[20] T. Distler, C. Cachin, and R. Kapitza. Resource-efficient Byzantine fault tolerance. *IEEE Transactions on Computers*, 65(9):2807–2819, 2016.

[21] T. Distler and R. Kapitza. Increasing performance in Byzantine fault-tolerant systems with on-demand replica consistency. In *Proceedings of the 6th European Conference on Computer Systems (EuroSys '11)*, pages 91–105, 2011.

[22] T. Distler, R. Kapitza, I. Popov, H. P. Reiser, and W. Schröder-Preikschat. SPARE: Replicas on hold. In *Proceedings of the 18th Network and Distributed System Security Symposium (NDSS '11)*, pages 407–420, 2011.

[23] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32:374–382, Apr. 1985.

[24] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić. The next 700 BFT protocols. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys '10)*, 2010.

[25] Z. Guo, C. Hong, M. Yang, D. Zhou, L. Zhou, and L. Zhuang. Rex: Replication at the speed of multi-core. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys '14)*, 2014.

[26] J. Hendricks, S. Sinnamohideen, G. R. Ganger, and M. K. Reiter. Zzyzx: Scalable fault tolerance through Byzantine locking. In *Proceedings of the 40th International Conference on Dependable Systems and Networks (DSN '10)*, pages 363–372, 2010.

[27] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference (ATC '10)*, pages 145–158, 2010.

[28] 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 *Proceedings of the 7th European Conference on Computer Systems (EuroSys '12)*, pages 295–308, 2012.

[29] M. Kapritsos, Y. Wang, V. Quéma, A. Clement, L. Alvisi, and M. Dahlin. All about Eve: Execute-verify replication for multi-core servers. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI '12)*, pages 237–250, 2012.

[30] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *Proceedings of the 21st Symposium on Operating Systems Principles (SOSP '07)*, pages 45–58, 2007.

[31] R. Kotla and M. Dahlin. High throughput Byzantine fault tolerance. In *Proceedings of the 34th International Conference on Dependable Systems and Networks (DSN '04)*, pages 575–584, 2004.

[32] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. TrInc: Small trusted hardware for large distributed systems. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI '09)*, 2009.

[33] P. J. Marandi, C. E. Bezerra, and F. Pedone. Rethinking state-machine replication for parallelism. In *Proceedings of the 34th International Conference on Distributed Computing Systems (ICDCS '14)*, pages 368–377, 2014.

[34] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd Workshop on Hardware and Architectural Support for Security and Privacy (HASP '13)*, 2013.

[35] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song. The honey badger of BFT protocols. In *Proceedings of the 2016 Conference on Computer and Communications Security (CCS '16)*, pages 31–42, 2016.

[36] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

[37] H. P. Reiser and R. Kapitza. Hypervisor-based efficient proactive recovery. In *Proceedings of the 26th Symposium on Reliable Distributed Systems (SRDS '07)*, pages 83–92, 2007.

[38] J. M. Rushby. Design and verification of secure systems. In *Proceedings of the 8th Symposium on Operating Systems Principles (SOSP '81)*, pages 12–21, 1981.

[39] N. Santos and A. Schiper. Achieving high-throughput state machine replication in multi-core systems. In *Proceedings of the 33rd International Conference on Distributed Computing Systems (ICDCS '13)*, pages 266–275, 2013.

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

[41] J. Sousa and A. Bessani. From Byzantine consensus to BFT state machine replication: A latency-optimal transformation. In *Proceedings of the 9th European Dependable Computing Conference (EDCC '12)*, pages 37–48, 2012.

[42] R. van Renesse, C. Ho, and N. Schiper. Byzantine chain replication. In *Principles of Distributed Systems*, pages 345–359. Springer, 2012.

[43] G. S. Veronese, M. Correia, A. Bessani, and L. C. Lung. Spin one's wheels? Byzantine fault tolerance with a spinning primary. In *Proceedings of the 28th Symposium on Reliable Distributed Systems (SRDS '09)*, pages 135–144, 2009.

[44] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung. EBAWA: Efficient Byzantine agreement for wide-area networks. In *Proceedings of the 12th Symposium on High-Assurance Systems Engineering (HASE '10)*, pages 10–19, 2010.

[45] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Veríssimo. Efficient Byzantine fault-tolerance. *IEEE Transactions on Computers*, 62(1):16–30, 2013.

[46] M. Vukolić. The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication. In *IFIP WG 11.4 Workshop on Open Research Problems in Network Security (iNetSec '15)*, pages 112–125, 2015.

[47] T. Wood, R. Singh, A. Venkataramani, P. Shenoy, and E. Cecchet. ZZ and the art of practical BFT execution. In *Proceedings of the 6th European Conference on Computer Systems (EuroSys '11)*, pages 123–138, 2011.

[48] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proceedings of the 19th Symposium on Operating Systems Principles (SOSP '03)*, pages 253–267, 2003.