Attested Append-Only Memory: Making Adversaries Stick to their Word

Byung-Gon Chun[†] Petros Maniatis[⋆] Scott Shenker[†‡] John Kubiatowicz[†]
[†]UC Berkeley [⋆]Intel Research Berkeley [‡]ICSI

ABSTRACT

Researchers have made great strides in improving the fault tolerance of both centralized and replicated systems against arbitrary (Byzantine) faults. However, there are hard limits to how much can be done with entirely untrusted components; for example, replicated state machines cannot tolerate more than a third of their replica population being Byzantine. In this paper, we investigate how minimal trusted abstractions can push through these hard limits in practical ways. We propose Attested Append-Only Memory (A2M), a trusted system facility that is small, easy to implement and easy to verify formally. A2M provides the programming abstraction of a trusted log, which leads to protocol designs immune to *equivocation* – the ability of a faulty host to lie in different ways to different clients or servers – which is a common source of Byzantine headaches. Using A2M, we improve upon the state of the art in Byzantine-fault tolerant replicated state machines, producing A2M-enabled protocols (variants of Castro and Liskov's PBFT) that remain correct (linearizable) and keep making progress (live) even when half the replicas are faulty, in contrast to the previous upper bound. We also present an A2M-enabled single-server shared storage protocol that guarantees linearizability despite server faults. We implement A2M and our protocols, evaluate them experimentally through micro- and macro-benchmarks, and argue that the improved fault tolerance is cost-effective for a broad range of uses, opening up new avenues for practical, more reliable services.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems; D.4.5 [**Operating Systems**]: Reliability

General Terms

Algorithms, Design, Reliability, Security

Keywords

Equivocation, Attested append-only memory, Byzantine-fault tolerance, Replicated state machines, Shared storage

SOSP'07, October 14–17, 2007, Stevenson, Washington, USA.

1. INTRODUCTION

In the distributed systems literature, it has long been a goal to offer clients the illusion of interacting with a single, reliable, fail-stop server, despite the occurrence of Byzantine server faults. While the initial results along these lines were largely theoretical, in recent years there has been an increasing interest in producing practical Byzantine-fault tolerant systems, as exemplified by PBFT [13], Q/U [6], Ivy [33], Plutus [20], SUNDR [26], HQ [14], and Zyzzyva [22].

The fault-tolerance properties of such systems can be divided into *safety* guarantees, properties that must be true at all times, and *liveness* guarantees, properties that must become true within finite time from all execution states of the system. For replicated state machines (e.g., PBFT, Q/U, HQ, and Zyzzyva) the target safety guarantee is *linearizability* [17]: completed client requests appear to have been processed in a single, totally ordered, serial schedule that is consistent with the order in which clients submitted their requests and received their responses. The corresponding liveness guarantee is that a correct client's request is eventually processed. It is well established that if servers have no trusted components, then no replicated system can provide these safety and liveness guarantees when more than a third of its replicas are faulty [25].

To improve on these results, some researchers have explored relaxed correctness properties. For instance, *fork* consistency* [27] is a weaker safety property than linearizability, but can be achieved when less than two thirds of the replica population are faulty. In single-server systems, the choice is only between 0% "replica" faults (the server is non-faulty) and 100% "replica" faults (the server is faulty). SUNDR showed how to achieve *fork consistency* (slightly stronger than fork*, but still weaker than linearizability) in the presence of a faulty server and non-faulty clients.

In this paper, our goal is to understand how the fault tolerance of such systems might be improved through the use of realistic trusted abstractions. Of course, placing the entire application (operating system, application software, hardware, intervening network) into the trusted computing base trivially solves the problem, but this is totally impractical. Our focus here is on small-footprint trusted primitives that have simple interfaces, are broadly applicable, and can be implemented easily and cost effectively. We argue that a *trusted log* abstraction, which we call Attested Append-Only Memory or A2M for short, is such a primitive. The power of A2M lies in its ability to eliminate *equivocation* – telling different stories to different entities – from the possible failure modes of untrusted components; for example, a faulty replica in a replicated system cannot undetectably answer the same question with different answers to different clients.

Section 2 motivates our choice of trusted abstraction, through examples from both replicated and single-server systems. Section 3

presents our first contribution, A2M, in more detail, describing its interface, typical usage patterns, and implementation alternatives that trade off efficiency for the size and complexity of the trusted computing base.

Next, we delve deeper into our second contribution: specific system designs for replicated state machines and shared storage that use A2M to improve their fault tolerance, in the context of agreement-based replicated state machines (Section 4) and other centralized and distributed protocols (Section 5). These include:

- A2M-PBFT-E is an A2M variant of Castro and Liskov's Practical Byzantine Fault Tolerance (PBFT) protocol. Similar to PBFT, A2M-PBFT-E guarantees safety and liveness with up to $\lfloor \frac{N-1}{3} \rfloor$ faulty replicas out of N total; however, whereas PBFT offers no guarantees whatsoever when this upper bound of faulty replicas is crossed, A2M-PBFT-E can still guarantee safety without liveness when the number of faulty replicas is more than $\lfloor \frac{N-1}{3} \rfloor$ but no more than $2\lfloor \frac{N-1}{3} \rfloor$. This is an important advantage for applications, such as high-volume banking, in which correctness (captured by safety) under heavy faults is desirable, even if it is not accompanied by availability (captured by the liveness property).

- A2M-PBFT-EA is an extension of PBFT that can guarantee both safety and liveness with up to $\lfloor \frac{N-1}{2} \rfloor$ replica faults: whereas PBFT needs a three-fold replication to tolerate a given number of faults, A2M-PBFT-EA needs only two-fold replication. The additional complexity of A2M-PBFT-EA may be justifiable in applications that require both low replication *and* high fault tolerance, as might be the case for critical applications with very high replication costs, such as dependable software for space missions.

- A2M-Storage is an A2M-enabled single-server storage service similar to SUNDR [26]. A2M-Storage leverages A2M to guarantee linearizability whereas SUNDR, without help from trusted components, can only provide fork consistency.

Section 6 presents an experimental evaluation of the A2M approach, using microbenchmarks on our implementation of A2M and two of our A2M-enabled protocols, A2M-PBFT-E and A2M-PBFT-EA. We also show macrobenchmarks on NFS running on top of A2M-PBFT-E and A2M-PBFT-EA, which suggest that the cost of using A2M to increase fault tolerance (or, conversely, reduced redundancy) is minimal: using an A2M module through a system call-like interface, the overhead of NFS on top of A2M-PBFT-EA is about 4% compared to that of NFS on top of traditional PBFT, or about 24% compared to NFS on top of an untrusted NFS server, for the benefit of reducing replication factor from 3 to 2.

We discuss the appropriate level for a trusted abstraction in Section 7, describe related work in Section 8, and then conclude in Section 9.

2. MOTIVATION

In this section, we detail the fundamental motivation behind our work, starting with our basic assumptions and target system environments, and continuing with specific illustrations of an adversary's power against existing systems, which will motivate our A2M design in Section 3, and A2M-related protocols in Sections 4 and 5.

2.1 Setup

We consider client-server systems where a service is accessed and shared by multiple clients connected over a public network. The service can be implemented as a single server (e.g., a file server) or multiple servers (e.g., replicated state machines). Clients request *authenticated* operations from the service, the service executes those operations, which may change the service state, and returns responses to the requesting clients.

2.2 Assumptions

We use standard assumptions about the network model and about cryptography. In the network, packet drops, reorderings, and duplications can occur but retransmissions of a message eventually deliver it. However, though finite upper bounds exist for message delivery and operation execution times, those bounds are not known to protocol entities. A faulty node cannot violate intractability assumptions about standard cryptography. Therefore, the adversary cannot produce pre-images or collisions for cryptographic hash functions[1] or forge previously unseen signatures for private signing keys he does not possess.

In this paper, we consider fault models that depend on the cause of the node's misbehavior. In particular, we distinguish between two cases: (i) the node's owner is well-intentioned but unaware the node's software has been compromised by a third-party (*faulty application model*), and (ii) the node's Byzantine behavior is because of a malicious owner instructing it to do so (*faulty operator model*). The nature of the trusted computing base is quite different in the two cases. In the first model, the trusted computing base is set up by the service owner; for instance, a bank owns all nodes and ensures, through physical security and other means, that only its nodes can provide the service. Our concern here is to combat software attacks such as worms and viruses against those centrally administered nodes. In the second model, we do not trust owners but trust a third party (e.g., a special service provider or a trusted hardware manufacturer) to set up the trusted computing base; for instance, a malicious storage server can manipulate all aspects of its node except what lies within the trusted device, which is the purview of the device provider.

In the traditional Byzantine-fault model, the cause of Byzantine behavior is not of immediate consequence – that is, tolerant protocols work well regardless of whether the operator or a virus writer are doing the misbehaving. Nevertheless, the practical decision to apply or not a solution to a target environment depends exactly on whether the designer can explain why the Byzantine-fault bound will not be violated; the justification is dependent on whether that environment consists of a single administrative domain (benign operator, potential software attacks) or multiple administrative domains (potentially malicious operators, potential software attacks).

2.3 Notation

For conciseness, throughout the paper we use the authentication notation of Yin et al. [39], according to which we denote by $\langle X \rangle_{S,D,k}$ an authentication certificate that any node in a set D can regard as proof that k distinct nodes in S said X. For example, a traditional digital signature on X from p that is verifiable by the entire replica population R would be $\langle X \rangle_{p,R,1}$, two signatures from p and q put together would be $\langle X \rangle_{\{p,q\},R,2}$, and a MAC from p to q with a shared key would be $\langle X \rangle_{p,q,1}$. As a convention, we use p to denote the singleton set $\{p\}$, and ∞ as shorthand for the universal set of all principals. When we use this notation to describe collective certificates made up of individual signatures, as

[1] A one-way – or pre-image resistant – hash function h is one for which there is no polynomial-time algorithm that, given α, can find a previously unknown β such that $\alpha = h(\beta)$. A collision-resistant hash function h is one for which there is no polynomial-time algorithm that can find two values α and β for which $h(\alpha) = h(\beta)$.

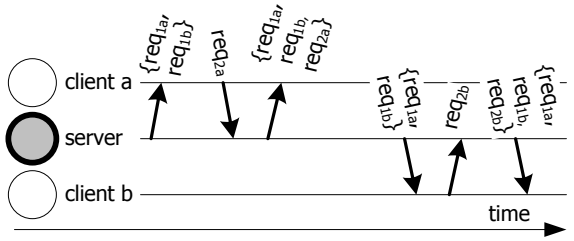

Figure 1: A forking attack example of two clients and one malicious server. The server convinces clients a and b of different system states.

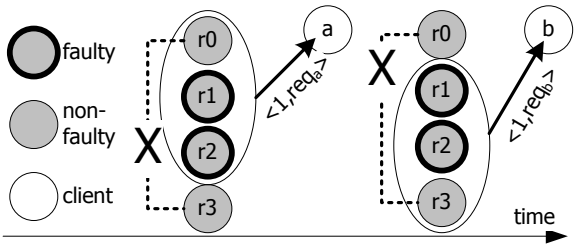

Figure 2: An example that shows the violation of linearizability in PBFT when two replicas are faulty out of four replicas. Faulty servers r_1 and r_2 convince non-faulty servers r_0 and r_3 to commit different requests.

for the second example above, we usually remove any signer identification from the collective certificate format: for example, the certificate $\langle X \rangle_{\{p,q\},R,2}$ above could correspond to the individually signed messages $\langle p, X \rangle_{p,R,1}$ and $\langle q, X \rangle_{q,R,1}$.

We use $h()$ to denote a one-way collision-resistant hash function such as SHA-256, and $\|$ to denote the bit-string concatenation operator.

2.4 Equivocation

In deterministic systems that aim to guarantee linearizability, lying is bad enough, but lying in different ways to different people is even worse. The "prototype" problem behind Byzantine-fault tolerant agreement, the "Byzantine generals problem," has been demonstrated unsolvable in a population of three parties when one is faulty [25], precisely because of equivocation. Beyond agreement, especially when there is a single server to contend with, equivocation can wreak just as much havoc: a server can feign ignorance to client A for data that it has promised client B it would broadcast to all. Even when it does not drop information, a faulty server can order sequential requests – think about two concurrent writes a and b to the same shared variable – in different ways when responding to different clients, potentially changing the presumed state of the system substantially: one client sees a as the dominating write whereas the other client sees b instead.

In what follows, we present two detailed examples of equivocation attacks against single-server and replicated systems, to motivate our focus on eliminating equivocation through trusted system abstractions.

2.4.1 Servers Equivocating to Clients

We consider a log-structured storage server shared by multiple clients as an illustrative example. For example, in a straw-man design for SUNDR [26], to request an operation, a client first acquires a lock at the server and downloads the entire operation log, a time-ordered collection of signed client operations. The client checks whether the log is correct by verifying the signatures and by checking that the log contains all of its own operations in order; it then creates what must be the server's current state by starting with an initial state and then applying the logged operations in order, as a correct server would have in a linearized system. It executes its operation based on the constructed state, thus finding out the result of this operation. It then appends its signed operation to the end of the log, sends the updated log back to the server, and releases the lock.

A faulty server can mount a forking attack [26] by concealing operations, which causes the system's state to diverge into multiple possibilities for different clients. Suppose two clients access a server as shown in Figure 1. Client a performs req_{1a}, client b performs req_{1b}, and client a performs req_{2a}. The latest state of the server becomes $\{req_{1a}, req_{1b}, req_{2a}\}$ as far as client a is concerned. Now, client b retrieves the log of the server to perform a new operation req_{2b}. The faulty server drops req_{2a} off the tail of the log, only returning $\{req_{1a}, req_{1b}\}$. Client b executes its operation and has the log state $\{req_{1a}, req_{1b}, req_{2b}\}$. The system state is now forked with regards to these two clients. The cause of the problem is the ability of the faulty server to misrepresent its operation log to the two clients, equivocating on what its state is according to who is asking.

Systems vulnerable to this kind of equivocation attacks are shared file systems such as Plutus [20], SUNDR [26], and Ivy [33], quorum-based replicated state machines such as Q/U [6], and timestamping systems such as Timeweave [30]. SUNDR and Timeweave alleviate the effects of equivocation, offering fork consistency, a weaker property than linearizability. For example, SUNDR maintains state about the server's timeline at individual clients; once forked, all clients within the same fork enjoy a linearized view of the system, but do not see state changes in another fork. Unfortunately, even then, unless two clients on different forks compare their notes, they cannot know that the server maintains multiple versions of its state and history.

2.4.2 Servers Equivocating to Servers

To demonstrate equivocation problems among servers, we consider BFT replicated state machines. In particular, we choose Practical Byzantine Fault Tolerance (PBFT) [13] since it has had a profound impact on the systems literature. Though we give more detailed background on PBFT in Section 4.1, for the purposes of this illustration, a PBFT client is satisfied with a result to its request if it receives at least $\lfloor \frac{N-1}{3} \rfloor + 1$ replies from distinct replicas out of the N total replicas, all with a matching result; a PBFT replica can commit a request to its local state as long as a quorum of $2\lfloor \frac{N-1}{3} \rfloor + 1$ replicas agree on the request's ordering in history.

Given this behavior, PBFT guarantees safety (linearizability) and liveness, as long as no more than $\lfloor \frac{N-1}{3} \rfloor$ replicas are faulty; if more than $\lfloor \frac{N-1}{3} \rfloor$ replicas are faulty, PBFT does not guarantee safety (and liveness is meaningless without safety): faulty replicas can fool non-faulty replicas to commit different request histories, and different clients may accept replies corresponding to different request histories, violating linearizability.

To illustrate, consider $N = 4$; replicas r_1 and r_2 are faulty, and non-faulty replicas r_0 and r_3 cannot temporarily communicate with each other (Figure 2). Client a sends req_a to the system. The two faulty replicas convince r_0 to commit and execute req_a first, since the three of them form a quorum of $3 = 2\lfloor \frac{N-1}{3} \rfloor + 1$. Later client b sends req_b to the system. The two faulty replicas convince r_3 to commit and execute req_b first, since r_3 never saw req_a. Faulty servers r_1 and r_2 equivocate to non-faulty servers r_0 and r_3.

Furthermore, the ability of faulty servers to equivocate to non-faulty servers also allows the service to equivocate to clients, as in the previous section. For example, clients a and b experience

via their accepted replies two different histories, in which req_a and req_b are, respectively, the single, first committed request, violating linearizability. The problem arises because of the faulty replicas equivocating to clients. The faulty replicas are allowed to tell client a, with r_0's help, that req_a is committed in their history at sequence number 1, and also to tell client b, with r_3's help, that req_b is committed in their history at the same sequence number.

Systems vulnerable to servers equivocating to servers are agreement-based Byzantine-fault tolerant state machine replication protocols such as PBFT [12] and BFT2F [27]. BFT2F supports fork* consistency by maintaining state at clients.

3. ATTESTED APPEND-ONLY MEMORY

In the previous section, we argued that the adversary's ability to equivocate undetected – e.g., to claim to have two different histories depending on which host it is talking to – is a fundamental weapon against safety, both in single-server and replicated services. Here we describe an *attested append-only memory* (A2M), a simple attestation-based abstraction that, when trusted, can remove the ability of adversarial replicas to equivocate without detection. Using an A2M implementation within the trusted computing base, a protocol can assume that a seemingly correct host can give only a single response to every distinct protocol request – for some protocol specific definition of "distinct" request –, even when that same request is retransmitted multiple times by different clients or replicas, and even if that response is undetectably faulty.

Informally, an A2M equips a host with a set of trusted, undeniable, ordered logs (illustrated in Figure 3). Each such log has an identifier q (unique within the same computer) and consists of a sequence of values, each annotated with (1) a log-specific sequence number that is incremented from 0 with every new value appended to the log, and (2) an incremental cryptographic digest of all log entries up to itself. Only a suffix of the log is stored in A2M, starting with the slot in the "low" position $\mathcal{L} \geq 0$ and ending with the last slot in the "high" position $\mathcal{H} \geq \mathcal{L}$.

A2M essentially offers reliable services a bit-commitment scheme [34] for sequential logs, placed within the trusted computing base. Section 3.1 describes the A2M interface, Section 3.2 presents simple usage scenarios illustrating how A2M can help a service to remove equivocation from the arsenal of Byzantine-faulty parties, and Section 3.3 explores the implementation options for A2M, along with the trust-efficiency trade-off for each.

3.1 Interface

An A2M log offers methods to `append` values, to `lookup` values within the log or to obtain the `end` of the log, as well as to `truncate` and to `advance` the log suffix stored in memory. There are no methods to replace values that have already been assigned.

- `append`(q, x) takes a value x, appends it to the log with identifier q, increments the highest assigned sequence number $\mathcal{H}$ by 1, populates the slot at that position with x, and computes the cumulative digest $d_{\mathcal{H}} = h(\mathcal{H} \| x \| d_{\mathcal{H}-1})$, where $d_0 = 0$. This method does not cause any values to be forgotten, i.e., it does not affect $\mathcal{L}$; if the log is unable to allocate storage to the new entry, the method fails.

- `lookup`$(q, n, z) \rightarrow \langle$LOOKUP$, q, n, z, x, w, n', d\rangle_{A2M_q,\infty,1}$ takes log identifier q, a sequence number n and a nonce z (for freshness), and returns a LOOKUP attestation. w is the type of the attestation: if sequence number n has not been assigned yet (i.e., $n > \mathcal{H}$) then w is UNASSIGNED and $n' = \mathcal{H}$; if n was assigned once but has now been forgotten (i.e., $n < \mathcal{L}$), then w is FORGOTTEN and $n' = \mathcal{L}$; if slot n has been skipped over via the `advance` method (see below) then w is SKIPPED and n' is the sequence number of the `advance` call that caused the skip; finally, if n is a slot that was filled via `append` or `advance` (see below), then w is ASSIGNED and $n' = n$. x and d are the assigned log value and digest when w is ASSIGNED) and 0 otherwise.

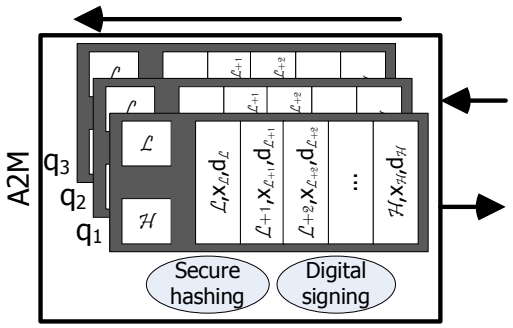

Figure 3: Structure of an *attested append-only memory* (A2M). An A2M contains a set of distinct logs (q_i) that map sequence numbers (in the range of $\mathcal{L}_i$ to $\mathcal{H}_i$) to values.

- `end`(q, z) is similar to `lookup`, but returns the last entry of the given log (currently in position $\mathcal{H}$). Attestations from `lookup` and `end` have the same format except for the request name END in the beginning.

- `truncate`(q, n), where $n \in (\mathcal{L}, \mathcal{H}]$, forgets all log entries with sequence numbers lower than n, setting $\mathcal{L}$ to n. All subsequent `lookup` requests for entries below n will be henceforth of type $w =$ FORGOTTEN.

- `advance`(q, n, d, x) allows log q to skip ahead by multiple sequence numbers. It takes a sequence number $n > \mathcal{H}$, a digest d, and a value x. It operates similarly to `append`, but instead of using $d_{\mathcal{H}-1}$ in the digest computation, it uses the given d; skipped sequence numbers are reported as SKIPPED in `lookups`. Any subsequent `lookup`(q, n'', z) request for a sequence number n'' that was skipped by this `advance` will return an attestation of the form $\langle$LOOKUP$, q, n'', z, x,$ SKIPPED$, n', d\rangle_{A2M_q,\infty,1}$, which contains information about the `advance` method that caused the skip, until the slot is finally FORGOTTEN.

3.2 A2M Usage

Equipped with A2M in its trusted computing base, a reliable service can mitigate the effects of Byzantine faults in its untrusted components, by being able to rely on some small fallback information about individual operations or histories of operations that cannot be tampered with.

During setup, the untrusted component (e.g., a server) must make known to all possible verifiers (e.g., clients or other servers) the authentication keys for its A2M module and the identifier of the A2M log used for each distinct purpose. As far as a verifier is concerned, the A2M authentication key and log identifier are part of the untrusted component's identity. Therefore, a particular A2M-enabled component is allowed to use only its associated A2M.

An untrusted component $\mathcal{C}$ can commit individual data items or operations by `append`ing them to an A2M log. For example, to prove that it has committed to a data item D, the component can

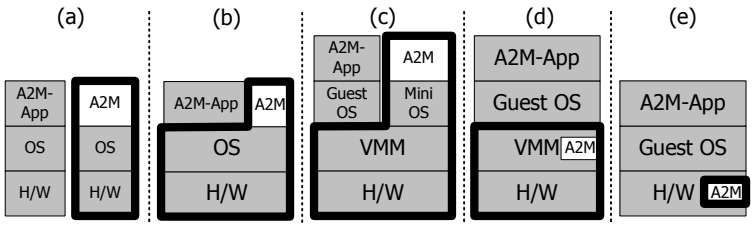

Figure 4: A2M implementation scenarios. Thick boxes delineate the trusted computing base. (a) trusted service, (b) trusted software isolation, (c) trusted VM, (d) trusted VMM, and (e) trusted hardware.

execute $\texttt{append}(q, h(D))$. The data item is hashed before appending to facilitate A2M implementations in which every log slot has a fixed length.

An interested verifier can establish that the data item is, indeed, in the untrusted component's committed state by demanding the attestation $\langle \text{LOOKUP}, q, n, z, x, \text{ASSIGNED}, n, d \rangle_{A2M_C, \infty, 1}$ for some sequence number n and nonce z, where $x = h(D)$ [2]. This conclusively establishes that the untrusted component indeed put the data item D somewhere into its committed log. The sequence number n can be further constrained (e.g., it can be associated with individual protocol steps) to ensure that the untrusted component only commits a single data item for that protocol step; in this sense, multiple verifiers who are mutually disconnected can be assured that the component cannot equivocate on the contents of its n-th slot.

To ensure that the untrusted component has a particular data item as the last element in its log, a verifier can provide the untrusted component with a random nonce z and demand the attestation $\langle \text{END}, q, n, z, x, \text{ASSIGNED}, n, d \rangle_{A2M_C, \infty, 1}$. As long as the request type is END, the nonce is the verifier-supplied nonce, and the value $x = h(D)$, the verifier can establish that as of the time of nonce transmission to the component, the last entry in the log was that containing D, and thus no trailing entries were spuriously chopped off by the untrusted component.

The untrusted component is not bound to committing to individual data items in sequential log slots; it can use `advance` to skip some sequence numbers. For example, if it only needs to commit to a value for every k-th sequence number, instead of $\texttt{append}(q, h(D))$ as above, it can use $\texttt{advance}(q, n, 0, h(D))$ for $n = ik$. Invocation of `advance` does not "unprove" things that the A2M has attested to before. It merely gives up the ability to attest to a real value for the skipped sequence numbers, and disassociates the newly appended request's digest from the log's cumulative history digest thus far, which is not required when committing to individual data items.

When interested in entire histories of data items (e.g., request logs), verifiers can make use of not only the committed data item itself, but also the cumulative digest d. Thanks to the collision-resistant properties of the hash function used, there is a single sequence of data items appended to log q for which the cumulative digest is d. Therefore, by comparing the digests in two LOOKUP attestations from two different untrusted servers, a verifier can establish conclusively that the two servers have committed to the same history up to the looked up sequence number. `advance` can be used, as above, to disassociate two portions of the log, for example, when part of the log is missing during a node's recovery.[3]

To revisit the scenario of a storage server that maintains a log for committed client requests but maliciously drops some off the end when talking to a victim client (Section 2.4.1), consider forcing the server to maintain that log in A2M. Client b can demand a fresh END attestation from the server's A2M log, along with the history itself, and ensure that the included digest is indeed the cumulative digest of the history; this guarantees to b that the server has not omitted any requests from the end of its committed log in its response, eliminating this particular problem. Similarly, to revisit the replicated scenario in which malicious replicas profess to different committed requests to different non-faulty replicas, convincing them to commit divergent requests (Section 2.4.2), consider requiring replicas to place such messages into an A2M message log before transmitting them. Now a non-faulty replica, before it allows itself to be convinced by another replica's message, ensures that the message is attested in a LOOKUP attestation drawn from the message sender's A2M message log. In this way, the faulty replica cannot equivocate to two different non-faulty replicas to effect the scenario.

These simple illustrations miss many finer details. We present detailed A2M-enabled protocol designs that achieve fault tolerance that they did not possess before, or increase their fault tolerance, in Sections 4 and 5.

3.3 Implementation Considerations

The fundamental premise behind an implementation of A2M is that it is harder to subvert than the main application. Different implementation scenarios (illustrated in Figure 4) lead to different threat models and degrees of trust in the resulting system, and are appropriate for different applications. Our contribution is a novel division of functionality between trusted and untrusted components, not a specific implementation of it – our experimental evaluation in Section 6 is a proof of concept, but other implementation scenarios are possible, some of which we characterize below.

The implementation scenarios we present are a separate service offered by a trusted provider or a hardened component (Figure 4(a)), a software-isolated module (Figure 4(b)), a trusted virtual machine (Figure 4(c)), a trusted virtual machine monitor (Figure 4(d)), and trusted hardware (Figure 4(e)). These implementations are viable in the face of different threats. All five implementations work under the faulty application model (external attacks against server software) but only (a) and (e) work under the faulty operator model (malicious operators that operate and can manipulate entire servers).

In the simplest case, A2M can be a software abstraction implemented as a service visible to applications via an RPC-like interface (Figure 4(a)). For instance, it could be a service offered by a trusted

[2]Note that we use $A2M_p$ to denote the authentication principal corresponding to host p's A2M module. Trusting A2M means that host p cannot forge authenticators by $A2M_p$ without A2M's cooperation, and that even then, it can only coerce A2M to generate such authenticators as per the A2M interface.

[3]It is important to point out that agreement of two A2M logs on the same sequence number and digest *does not imply* necessarily that the two logs must also agree on attestations about all preceding sequence numbers and digests; the use of `advance` legitimately contradicts this implication. It is possible to change the interface so as to guarantee this implication, but this is not required for our case studies in this paper.

provider, such as Amazon's S3 [1], or by a separate, hardened component with significantly greater assurances in the face of software errors than the main application software and hardware. This is similar to notarization-like approaches [15, 30, 40] that rely on a trusted write-once medium external to the main system. Though the entire application stack can fail (application, operating system, and hardware), as long as the A2M is running on a trusted system the application can be protected. The big drawback with this implementation scenario is its network-bound nature – in fact, many of its prior instances in practice use this external write-once medium once a day or so – as well as the requirement that everyone needs on-line access to the trusted A2M service provider. Applications with fairly slow request rates such as shared backup services, long-term digital preservation, or certificate authorities may be able to absorb the high-latency interaction with A2M in their relatively infrequent state changes.

Figure 4(b) presents a more decentralized approach, in which the A2M implementation relies on the software-based isolation between A2M and an A2M-enabled application. This approach takes advantage of programming language type and memory safety for isolation. Therefore, A2M can be implemented as a library. For instance, in the Singularity [19] operating system, the A2M module would be a program that runs as a separate software-isolated process in the same address space. If the Singularity isolation mechanism is trusted, it is possible to trust A2M even if the A2M-enabled application is untrusted. Similarly, in the Java Virtual Machine (JVM) [3], an application using A2M runs in a sandbox, which constitutes a safe execution environment. The assumption is that if the JVM interpreter, JVM core classes, and an operating system that runs the JVM can be trusted, A2M can be trusted, even if the A2M-enabled Java application is not. Though the isolation is no longer physical as with the scenario of Figure 4(a), communication between the application and A2M is fast since they are both in the same address space.

Figure 4(c) presents the A2M implementation that relies on the inherent fault isolation properties of a virtual machine monitor (VMM). In the figure, the A2M module is a user-space program running on a small, verifiable operating system on top of a VMM. As long as the VMM and the mini-operating system are trusted to be exploit-free, it is possible to trust the A2M abstraction, even if the application and its general-purpose operating system are compromised. For instance, the virtual Trusted Platform Module (vTPM) [11] has this architecture. Communication between the application and A2M is only subject to VMM-optimized RPCs, which systems such as Xen [10] make very efficient.

Further reducing the trusted footprint, the A2M implementation could be placed within the VMM, as in Figure 4(d). Here, the assumption is that a small VMM (or, indeed, a microkernel) can be carefully implemented (or formally verified) as bug-free, isolating the correctness of the A2M implementation from potential operating system or application errors above the VMM. For instance, Xen's trusted hypervisor interfaces [10] could host such an implementation scenario. Both VMM approaches reduce the cost of contacting A2M and can yield efficient, interactive performance for applications such as file systems or transaction processing systems.

Finally, Figure 4(e) places the A2M within the hardware itself. Since it tends to be much harder to coerce a hardware module to operate against its specification than it is for software modules, especially without physical access to the hardware, this scenario provides the greatest level of trust in A2M. Hardware implementation options might be to extend a standard Trusted Platform Module (TPM) with some additional non-volatile RAM or an Intel Active Management Technology (AMT) chip [2], or to use a programmable secure coprocessor such as IBM's commercially available PCIXCC [8] board, a programmable PCI-X card with cryptographic primitives as well as physical and electrical tamper-resistance. Tamper resistance offers increased *physical security*: even a malicious host operator armed with electrical probes cannot coerce A2M to give responses that are inconsistent with its specification or to reveal its authentication key material, except for extremely expensive physical cryptanalytic attacks that are unrealistic for most practical situations. Moreover, whereas in the past tamper resistance implied low performance, products such as the PCIXCC coprocessor make a hardware A2M implementation potentially the best performing one – albeit most expensive – among our scenarios. Nevertheless, pervasive hardware implementations of new programming abstractions tend to be slow to arrive, slow to change, and slow to turn into commodities, making this a more tenuous scenario, except for the most sensitive applications.

In this paper, we experiment with a software A2M implementation. Values stored within A2M logs can have a configurable fixed size, e.g., 32 bytes. The A2M sequence number field needs to have a size large enough to hold sequence numbers of long-running applications (e.g., 160 bits). We implement authentication based on both digital signatures and MACs (with a slightly modified interface from that in Section 3.1), though we describe the digital signature version of all protocol designs for simplicity.

4. A2M STATE MACHINE REPLICATION PROTOCOLS

In this section, we present state machine replication protocols through the use of A2M, improve their fault tolerance by rendering equivocation extinct or evident. First, in Section 4.1, we present a brief overview of the salient features of Castro and Liskov's PBFT protocol for replicated state machines. Second, in Section 4.2, we present a simple extension of PBFT, in which A2M protects clients from the replicas' misbehavior, retaining PBFT's safety and liveness for up to $\lfloor \frac{N-1}{3} \rfloor$ faulty replicas out of N, but also guaranteeing safety without liveness for up to $2\lfloor \frac{N-1}{3} \rfloor$ faulty replicas. Second, Section 4.3 goes further to protect not only clients from replica misbehavior in PBFT, but also replicas from each other, allowing the fault tolerance of the protocol to go up to $\lfloor \frac{N-1}{2} \rfloor$ with *both* safety and liveness.

4.1 Background: PBFT

Castro and Liskov's PBFT protocol [13] is a replicated, fault-tolerant mechanism for implementing a *state machine* [37]: an abstraction that represents a deterministic service, in which a starting state (e.g., an empty database) and the sequence of read-compute-write operations at the service determine precisely the state of that service at the end of the operation sequence. Such state machines are relatively straightforward to implement on a single, single-threaded server at an individual computer, though any faults at that computer always cause a service failure. For fault-tolerance reasons, it often makes sense to implement the state machine abstraction over a population of such potentially faulty computers interconnected via a potentially faulty network, hoping that even if some computers fail, the service as a whole can continue functioning correctly. Unfortunately, implementing the state machine abstraction over such a population and network is no simple task. In PBFT, each participating computer implements the entire state machine on its local replica of the service state, and replicas communicate with each other to ensure that they all execute the same sequence of operations, and mask individual computers' faults. We describe the protocol in more detail below.

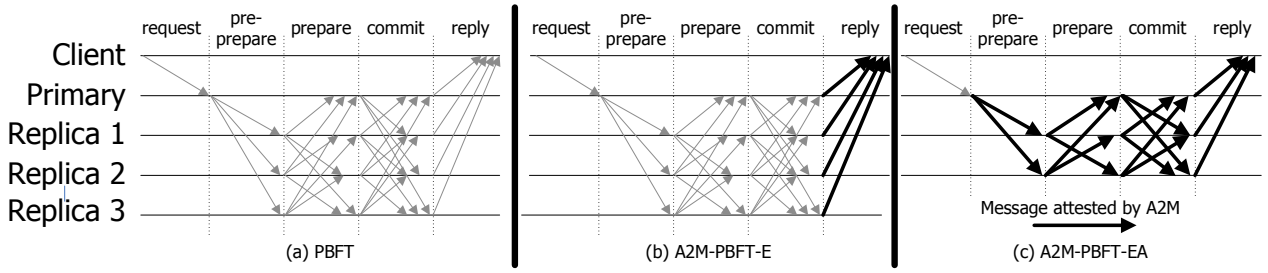

Figure 5: Three-phase agreement protocol. Thicker lines denote messages that are attested to using A2M.

In PBFT, a client c multicasts a request message $\langle \text{REQUEST}, o, t, c \rangle_{c,R,1}$ to the N service replicas in replica set R, where o is the operation requested, and t is the timestamp. The client accepts a reply for its request (and only then can submit another) when it receives $\lfloor \frac{N-1}{3} \rfloor + 1$ valid matching REPLY messages, forming the *reply certificate* $\langle \text{REPLY}, v, n, t, c, r \rangle_{R,c,\lfloor \frac{N-1}{3} \rfloor + 1}$, where v is the view number, n is the assigned sequence number, and r is the result of the request. A *view* is a particular assignment of roles to replicas: the single active *primary* vs. the passive *backups*; when the primary changes, so does the view number v.

Replicas linearize requests via a three-phase agreement protocol (Figure 5(a)), starting when the primary (chosen to be the replica with identifier $p \equiv v \mod N$) multicasts to R a newly received request message req, encapsulated within a message $\langle \text{PREPREPARE}, v, n, req \rangle_{p,R,1}$. When backup replica i receives this PREPREPARE, it multicasts to R a $\langle \text{PREPARE}, v, n, req \rangle_{i,R,1}$ message. Once replica j has collected $2\lfloor \frac{N-1}{3} \rfloor + 1$ PREPREPARE or PREPARE messages from distinct replicas for this request (which constitute the *prepared certificate* for this request of the form $\langle \text{PREPARE}, v, n, req \rangle_{R,R,2\lfloor \frac{N-1}{3} \rfloor + 1}$), the request becomes *prepared*. To complete the protocol, a replica with a prepared request then multicasts to R a $\langle \text{COMMIT}, v, n, req \rangle_{j,R,1}$ message. When replica k collects $2\lfloor \frac{N-1}{3} \rfloor + 1$ such messages (which constitute the *committed certificate* of the form $\langle \text{COMMIT}, v, n, req \rangle_{R,R,2\lfloor \frac{N-1}{3} \rfloor + 1}$), the replica has established the linearized sequence for this request, committing to execute it as soon as it can; this concludes the *agreement* portion of the PBFT protocol for this request, whose purpose is to ensure that the replicas agree on a single operation sequence for the service, as more clients submit requests for further operations.

A replica can execute the request in its local state as soon as it has finished executing the committed requests for all sequence numbers lower than n. It packages the result in a REPLY message, which it sends to the client directly. When the client has received a quorum of such matching replies – the reply certificate described above – the *execution* portion of the protocol concludes; the purpose of the execution portion is to represent to the client accurately the service state (and reply to the client's request accordingly), as determined by executing the sequence of operations that the agreement protocol portion maintains.

Though the request log can itself represent the service state, replicas periodically garbage-collect their operation log to reduce storage consumption: they create a checkpoint of their local state at a particular sequence number n and a cryptographic hash s of that state. When replica i creates such a checkpoint, it multicasts to R a $\langle \text{CHECKPOINT}, n, s, i \rangle_{i,R,1}$ message. Once it has collected a checkpoint certificate $\langle \text{CHECKPOINT}, n, s \rangle_{R,R,2\lfloor \frac{N-1}{3} \rfloor + 1}$, the replica deems that checkpoint "stable," and truncates its operation log up to sequence number n.

When replica i has out-of-date service state (e.g., due to transient network partitions or because it is slow), it can catch up with the rest by retrieving missing committed requests, along with their committed certificates, from another, more up-to-date replica. If other replicas no longer have those certificates in their logs due to garbage collection, the lagging replica can fetch the latest stable checkpoint and certificate, and then any subsequent committed requests after that checkpoint.

Finally, PBFT has a view-change protocol that changes the system's primary when the primary is suspected faulty. When backup replica i in view v times out waiting for a request to commit, it suspects the primary as faulty, and multicasts to R a $\langle \text{VIEWCHANGE}, v+1, n, s, C, P \rangle_{i,R,1}$ message, where n is the sequence number for the latest stable checkpoint, s is the digest of the stable checkpoint, C is a stable checkpoint certificate, and P is a set of prepared certificates whose sequence number is higher than n.

When a new primary ($p = v + 1 \mod N$) collects a new view certificate V that consists of $2\lfloor \frac{N-1}{3} \rfloor + 1$ valid VIEWCHANGE messages containing correct C and P, it multicasts to R a $\langle \text{NEWVIEW}, v+1, V, O \rangle_{p,R,1}$ message, where O is a set of PREPREPARE messages in the new view. To determine O, let ℓ be the sequence number of the latest stable checkpoint in V, and let u be the highest sequence number in P. For each sequence number between $\ell + 1$ and u, the primary creates a PREPREPARE message if a prepared certificate exists in V, or a PREPREPARE message for a no-op operation otherwise (to skip that sequence number in the new view).

When a backup replica receives a NEWVIEW message, it verifies O is correctly computed by performing the same procedure as the primary. If the message is valid, the replica adds the new information to its log, logs and multicasts to R PREPARE messages for each message in O, and enters view $v + 1$. The backup processes messages with a view number v' higher than the current view only after it receives a valid NEWVIEW message for v'.

4.2 A2M-PBFT-E

In this section, we describe A2M-PBFT-E, a simple extension of PBFT that uses A2M logs to protect the execution portion of PBFT (hence the "E" suffix of the acronym); that is, it ensures that replicas cannot equivocate about their locally computed results for a particular requested client operation when replying to that or any other client (Figure 5(b)). As before, we consider a population R of N replicas.

4.2.1 Design

Replicas: An A2M-PBFT-E replica i maintains all state maintained by a PBFT replica, as well as an A2M log for what it believes as the agreed request sequence; that log has identifier q_i. Other replicas and clients identify this replica as a pair $\langle i, A2M_i \rangle$ of principals, i for the replica node itself, and $A2M_i$ for the replica's A2M module. As a convenience, we use $A2M_R$ to mean the set of all A2M principals used by replicas in R.

An A2M-PBFT-E replica is functionally identical to a PBFT replica with regards to agreement, but differs on protocol aspects that involve execution, namely client interaction and checkpoint management.

Once replica i collects a committed certificate for sequence number n, it executes the request req on its local application state obtaining result r, it appends the associated request to its log q_i with $\texttt{append}(q_i, h(req))$, and uses $\texttt{lookup}(q_i, n, n)$ to obtain the A2M attestation $\langle \text{LOOKUP}, q_i, n, n, h(req), \text{ASSIGNED}, n, d \rangle_{A2M_i,R,1}$. Finally, it packages the regular PBFT reply message and the attestation into a single message, which it sends back to the client.

As per PBFT, replica i performs garbage collection on its log and A2M request history by exchanging CHECKPOINT messages. When replica i creates a checkpoint, it multicasts to R a $\langle\langle \text{CHECKPOINT}, n, s, d' \rangle_{i,R,1}, \langle \text{LOOKUP}, q_i, n, n, x, \text{ASSIGNED}, n, d \rangle_{A2M_i,R,1}\rangle$ message where n is the sequence number of the last executed request to produce the checkpoint state, s is the state digest, d' is the A2M digest for sequence $n-1$ (need not be attested), and x is the hash of the n-th committed request. The checkpoint becomes stable when a replica collects a checkpoint certificate $\langle\langle \text{CHECKPOINT}, n, s, d' \rangle_{R,R,2\lfloor \frac{N-1}{3} \rfloor+1}, \langle \text{LOOKUP}, n, n, x, \text{ASSIGNED}, n, d \rangle_{A2M_R,R,2\lfloor \frac{N-1}{3} \rfloor+1}\rangle$. The replica adds this information to its log, removes all messages with sequence number up to n from the log, and performs $\texttt{truncate}(q_i, n)$.

When replica i performs a state transfer, it performs the regular-PBFT process of fetching and installing a state with a stable checkpoint certificate and subsequent agreement messages into its message log. In addition to this, an A2M-PBFT-E replica must also update its A2M request log, by performing $\texttt{advance}(q_i, n, d', x)$, and then appending all subsequently committed requests in ascending sequence order.

Clients: In A2M-PBFT-E, a client c is identical to a PBFT client, except it expects from replica i reply messages of the form $\langle\langle \text{REPLY}, v, n, t, c, r \rangle_{i,c,1}, \langle \text{LOOKUP}, q_i, n, n, h(req), \text{ASSIGNED}, n, d \rangle_{A2M_i,R,1}\rangle$ for its pending request req. This is the PBFT REPLY along with the A2M-attested content of the n-th A2M log entry at the sender. To consider its request completed and accept the result, a client waits until it collects a reply certificate $\langle\langle \text{REPLY}, v, n, t, c, r \rangle_{R,c,2\lfloor \frac{N-1}{3} \rfloor+1}, \langle \text{LOOKUP}, n, n, h(req), \text{ASSIGNED}, n, d \rangle_{A2M_R,R,2\lfloor \frac{N-1}{3} \rfloor+1}\rangle$.

Note that the size of the reply certificate is $2\lfloor \frac{N-1}{3} \rfloor + 1$ in A2M-PBFT-E, as opposed to $\lfloor \frac{N-1}{3} \rfloor + 1$ in PBFT. However, the popular read-only optimization in PBFT – in which read-only requests can be answered by replicas immediately upon reception without a three-phase commit – also requires replies of size $2\lfloor \frac{N-1}{3} \rfloor + 1$, making this difference moot in practice.[4]

4.2.2 Correctness

At a high level, we show that if at most $\lfloor \frac{N-1}{3} \rfloor$ replicas are faulty, A2M-PBFT-E does not cause clients to accept more replies than they would under PBFT (therefore does not violate safety) and does not block operations that would have proceeded in PBFT (i.e., does not remove liveness). When the number of faulty replicas ranges between $\lfloor \frac{N-1}{3} \rfloor + 1$ and $2\lfloor \frac{N-1}{3} \rfloor$, we show that A2M-PBFT-E can only assign to any sequence number a unique client request, and that the reply delivered to clients for any sequence number is that which a non-faulty replica would have produced processing the sequence requests in order.

Case 1: When no more than $\lfloor \frac{N-1}{3} \rfloor$ replicas are faulty, the safety of A2M-PBFT-E follows from PBFT's safety: A2M-PBFT-E attestations in replies at worst *prevent* a client from accepting a reply that PBFT would otherwise accept (if the REPLY portion of the message matches but the A2M portion does not); A2M-PBFT-E attestations never cause what would have been an unacceptable set of REPLY messages in PBFT to be acceptable. The same holds for liveness, since the addition of the A2M log attestation in REPLY messages cannot hinder progress: there exist at least $2\lfloor \frac{N-1}{3} \rfloor + 1$ non-faulty replicas that maintain their A2M request logs correctly, and as a result, there always exists a quorum of $2\lfloor \frac{N-1}{3} \rfloor + 1$ replicas that can provide clients with a REPLY certificate. Replicas can also create a stable checkpoint since there always exists a quorum of $2\lfloor \frac{N-1}{3} \rfloor + 1$ non-faulty replicas to produce a CHECKPOINT certificate.

Case 2: When faulty replicas are more than $\lfloor \frac{N-1}{3} \rfloor$ and no more than $2\lfloor \frac{N-1}{3} \rfloor$, we argue inductively that for every sequence number, any non-faulty client can only accept a unique request – which establishes that there exists a single linearized schedule of requests – and can only accept the correct result value for that linearized schedule. In the base case, consider a client accepting req_1 for sequence $n = 1$. Since the corresponding REPLY certificate (of size $2\lfloor \frac{N-1}{3} \rfloor + 1$) includes at least one non-faulty replica, the reply and result certainly correspond to what that non-faulty replica would do with a singleton schedule containing only req_1. Suppose another non-faulty client accepts a different request req_2 and result for the same sequence number $n = 1$. Such a client would also possess a valid REPLY certificate of the same size; the two certificates contain at least one replica in common. However, since that replica is bound by A2M to supply the same A2M log entry to both clients, the A2M attestation of that replica present in the two certificates must be identical, which means that the two certificates must match; this means $req_1 = req_2$, since the request hashes using a collision-resistant hash function also match. This is a contradiction, so there can be no such req_2.

The inductive step for sequence number $n+1$ given a linearized schedule up to n is similar. Any two clients accepting a reply for $n+1$ will have matching requests for that sequence number (as witnessed by the matching request hashes in the two log attestations), *and* matching request histories up to that sequence number (as witnessed by the digest d in the A2M log attestations). Therefore, the result computed by the non-faulty replica in each of the two reply certificates must correspond to the same request history and, due to the deterministic nature of the state machines we consider here, must produce the same result.

Replicas participating in a reply that have used the state transfer mechanism at some point in their history do not affect this correctness argument. After accepting a stable checkpoint certificate, a replica has an n-th A2M log entry that is identical to all the replicas in the checkpoint certificate, including at least another non-faulty replica. Furthermore, the state described in the checkpoint is that held by at least another non-faulty replica.

4.2.3 Discussion

In the A2M-PBFT-E presentation above, A2M is used to protect only the sequence of committed requests, as they are presented to clients in REPLY messages. However, when faulty replicas are at

[4] A2M-PBFT-E supports this read-only optimization by replacing LOOKUP attestations with END attestations in the client reply, and using a client-supplied nonce in the attestation, when handling a read-only request; this proves to the client that the result provided is drawn from the latest state of the service, rather than an earlier state (in which case, faulty up-to-date replicas would have advanced their committed request log beyond the attestation they are required to return freshly).

least $\lfloor \frac{N-1}{3} \rfloor + 1$, they can confuse non-faulty replicas by equivocating during agreement. For example, in Figure 2, the use of A2M will not prevent the faulty replicas from causing non-faulty replica r_0 to place request req_a in its A2M position 1 and, at the same time, causing non-faulty replica r_3 to place req_b in its A2M at the same position. Though no client will accept inconsistent replies (since reply messages contain A2M attestations), the replicas themselves are not protected. For the purposes of the protocol, one of the two non-faulty replicas effectively becomes faulty when convinced to adopt a fork in the request history.

The great benefit of A2M-PBFT-E is that such misbehavior causes the system to stop making progress but not to violate its correctness breaking linearizability. In the simplest scenario, an operator who notices lack of forward progress can take the system offline, identify the history fork (where committed histories diverged), repair the divergent replicas, change their A2M log identifiers, advance their new A2M logs to an earlier correct sequence number from which A2M-PBFT-E can do state transfers, and restart the system with no loss beyond transient unavailability and human effort.

However, a natural next step is to remove this denial-of-service attack from the arsenal of the adversary, by ensuring that the agreement portion of the protocol is itself also protected from equivocation. In the next section, we describe A2M-PBFT-EA, a PBFT extension that protects not only the execution portion (i.e., client-facing messages) against equivocation, but also the agreement portion (i.e., replica-facing messages), thereby increasing the fault tolerance of PBFT with both safety *and* liveness.

4.3 A2M-PBFT-EA

To protect against equivocation during agreement, A2M-PBFT-EA (the "EA" suffix stands for *Execution+Agreement*) requires replicas to append to A2M logs all protocol messages before sending them to their peers (Figure 5(c)). Unlike the history log, message logs need not protect a sequence of entries, but only an individual message; therefore, A2M's `advance` is used to place a message into an A2M message log, as opposed to `append`. Unlike A2M-PBFT-E and PBFT, which can have multiple requests in flight at the same time, in A2M-PBFT-EA we require that non-faulty replicas handle one request at a time, in increasing sequence-number order.[5] This ensures that messages are appended to their corresponding A2M logs in the order of their corresponding sequence number. By protecting protocol steps from equivocation, A2M-PBFT-EA requires only one – potentially faulty – replica in the intersection of two quorums. Note, in comparison, that PBFT requires at least one *non-faulty* replica in the intersection of two quorums.

When configured with A2M-PBFT-E's quorum sizes, A2M-PBFT-EA has the same safety and liveness properties as A2M-PBFT-E. In what follows, we instead present A2M-PBFT-EA with quorum sizes that allow it to tolerate up to $\lfloor \frac{N-1}{2} \rfloor$ faults with both safety and liveness. We present here a three-phase version of A2M-PBFT-EA to make easier the comparison to the original PBFT. In fact, other versions of the protocol, including a two-phase version, are straightforward to design and prove correct, but we defer the details to a longer version of this paper.

4.3.1 Design

Clients: An A2M-PBFT-EA client is similar to an A2M-PBFT-E client, but it expects reply certificates of size $\lfloor \frac{N-1}{2} \rfloor + 1$ instead of $2\lfloor \frac{N-1}{3} \rfloor + 1$.

[5]PBFT offers a runtime setting (the high- and low-watermark values) that can be configured to guarantee this requirement.

Replicas: All certificates (for prepared and committed requests, for view changes, and for checkpoints) in A2M-PBFT-EA have size $\lfloor \frac{N-1}{2} \rfloor + 1$, as opposed to $2\lfloor \frac{N-1}{3} \rfloor + 1$ in A2M-PBFT-E.

In addition to a committed request history log, an A2M-PBFT-EA replica i maintains five message logs: PREPARE (which also contains PREPREPARES) and COMMIT for the three-phase agreement, CHECKPOINT for garbage collection, and VIEWCHANGE and NEWVIEW for view changes. Before sending any such PBFT message $\langle \mathcal{M} \rangle$, an A2M-PBFT-EA replica inserts that message to the corresponding message log $m_{\mathcal{M},i}$ (via an `advance` call), uses `lookup` to obtain an attestation $\langle \mathcal{E} \rangle_{A2M_i,R,1}$ for that message, and sends $\langle \langle \mathcal{M} \rangle, \langle \mathcal{E} \rangle_{A2M_i,R,1} \rangle$ to the intended destination. Conveniently, a message that has been committed to A2M in this way need not itself be authenticated to its destination principal; the A2M attestation of the message hash is enough to protect that message from integrity attacks and to make it non-repudiable. Non-attested messages still need to be authenticated as before. Since message logs are typically used for individual attestations and not for message histories, an `advance` call is sufficient, as opposed to an `append`.

A non-faulty replica might have to send multiple versions of a PREPREPARE/PREPARE or a COMMIT message for a given sequence number n, but for different views. The protocol *flattens* the $\langle v, n \rangle$ identifier of such messages to fit them in the A2M log entry sequence space, by partitioning log sequence numbers into two parts: the x most significant bits (e.g., 64 bits) represent a view number while the remaining y bits (e.g., 96 bits) represent a PBFT request sequence number. The log entry number for a PREPREPARE/PREPARE or COMMIT message about view v and sequence number n is then $n + v2^y$; we use $[v|n]$ to denote this flattened number in what follows. Note that the A2M module is oblivious to this "overloading" of its sequence number space; no changes are required to the A2M interface.

To illustrate the concepts of message attestation and identifier flattening, we present as an example the prepare phase of A2M-PBFT-EA. Where a PBFT replica i would send the PREPARE message $prep = \langle \text{PREPARE}, v, n, req \rangle$, an A2M-PBFT-EA replica commits the message to its corresponding log m_p by invoking $\texttt{advance}(m_p, [v|n], 0, h(prep))$, extracts the corresponding LOOKUP attestation $att = \langle \text{LOOKUP}, m_p, [v|n], [v|n], h(prep), \text{ASSIGNED}, [v|n], d' \rangle_{A2M_i,R,1}$, and then bundles and sends $\langle prep, att \rangle$. When an A2M-PBFT-EA replica receives such an attested PREPARE message, it verifies the A2M authentication, and then checks that the value attested is the hash of the included PREPARE message. When a replica collects $\lfloor \frac{N-1}{2} \rfloor + 1$ such messages that match req for the same sequence number n and view v, the request is prepared. The commit phase is similar to the prepare phase described. The checkpoints, state transfer, and execution portions of A2M-PBFT-EA are the same as with A2M-PBFT-E, except for the addition of message attestations in certificates and the different quorum sizes.

View Change: View changes are different from PBFT and A2M-PBFT-E. In PBFT, the quorum forming a NEWVIEW certificate is guaranteed to contain at least one non-faulty replica with the latest committed requests, thanks to the quorum size and the maximum number of faulty replicas. In contrast, the A2M-PBFT-EA quorum size can guarantee, in the worst case, that a single potentially-faulty replica with the latest committed requests will participate in the view change. To address the challenge, an A2M-PBFT-EA replica must be forced to give its latest A2M-committed information, which requires a fresh, shared nonce in the associated `lookup` A2M operations. To accomplish this, the protocol requires an extra phase before the normal view-change protocol, which enables

replicas to construct a fresh nonce for the subsequent phases (via WANTVIEWCHANGE messages). For similar reasons, the protocol must ensure that replicas committed to a view change (as evidenced by their issuance of an attested VIEWCHANGE message) cannot subsequently help commit requests in the previous view. Therefore, a VIEWCHANGE message in A2M-PBFT-EA requires the sending replica to explicitly *abandon* the previous view: a replica does this by `advance`ing its COMMIT message log to the end of the old view and attesting to this advancement within its VIEWCHANGE message. We present the detailed A2M-PBFT-EA view change protocol in Appendix A.

4.3.2 Correctness

At a high level, A2M-PBFT-E and A2M-PBFT-EA differ in two fundamental ways: on one hand A2M-PBFT-EA has smaller quorum sizes, but on the other hand, it requires all protocol messages to be attested to from an appropriate A2M log before use. However, the argument presented in the case 2 of Section 4.2.2 also applies to the safety of A2M-PBFT-EA. It guarantees safety with up to $\lfloor \frac{N-1}{2} \rfloor$ faults since clients accept REPLY certificates of size $\lfloor \frac{N-1}{2} \rfloor + 1$.

To show that A2M-PBFT-EA is live despite up to $\lfloor \frac{N-1}{2} \rfloor$ faults, we show a new safety invariant that is not necessary for linearizability: all non-faulty replicas agree on a single committed request sequence. That is, a faulty replica cannot convince two non-faulty replicas to commit to their respective A2M request logs different requests for the same sequence number. The argument is split into a same-view case and a different-view case. For the same-view case, it follows backwards the agreement process from appending a request to the log, to emitting a COMMIT message, to emitting a PREPARE message, showing that for two different requests to be placed in two non-faulty replicas' request logs, some A2M must be faulty, which is incompatible with our fault model. For the different-view case, the argument is similar, but must also traverse NEWVIEW certificates; view abandonment in such certificates helps show that it is not possible for a single replica (faulty or not) to have an attested COMMIT message for one request in one view, and at the same time support a view change feigning ignorance for that message, leading to a contradiction. The argument is highly technical, so we defer it to Appendix B.

5. OTHER A2M PROTOCOLS

In this section, we describe A2M-Storage, an A2M-enabled storage system on a single untrusted server shared by multiple clients. Thanks to the use of a trusted A2M module, A2M-Storage provides linearizability in contrast to SUNDR's weaker fork consistency and is simpler than SUNDR. We then briefly sketch how A2M can be used with Q/U to improve its fault tolerance.

5.1 A2M-Storage

5.1.1 Background: SUNDR

SUNDR targets the same problem as PBFT: linearize client requests and ensure that the service state used to respond to each request corresponds to a correct system having executed this linear request history. In PBFT, agreement is used among replicas to obtain a linearized request order. The presence of at least one non-faulty replica corroborating a reply to the client ensures that the agreed upon linearized order has been executed correctly producing the result in the reply. Unfortunately, in a single-server environment such as SUNDR's, there is no non-faulty replica trusted to execute linearized requests; instead, the clients must trust each other and cooperate to check themselves that requests are properly linearized and execution is performed correctly at the server.

A SUNDR server maintains the current service state (a snapshot of a shared file system), which is represented by Merkle trees [32].[6] The state is captured by a set of version structures, each of which is owned by a client (principal) and contains a hash that summarizes the whole state on which the client operates.

To perform an operation (read/write on a file), a SUNDR client submits to the server its intended request, called an *update certificate*. The server assigns an order to the request relative to pending operations that have not committed yet, and returns the latest committed version structures and ordered pending update certificates. The client ensures that the state transits correctly forward from its last committed version the server gives via a sequence of pending operations. The client can then perform its operation locally, potentially fetching missing blocks by following digests of the hash tree, compute and sign a new state digest creating a new version structure, and return it along with changed blocks to the server. The server stores the new version structure and modified blocks.

As described in simpler terms in Section 2.4.1, a SUNDR client cannot ensure that the server sends it the latest state resulting from the committed history of requests; though it cannot remove requests from the middle, the server can still chop off the tail of history past the last request known to that client, and start a new "fork" in that history, specific to the client. Until two clients on different history forks compare their notes, they cannot know the system is not linearized. This is what makes SUNDR only fork-consistent but not linearizable.

5.1.2 Design

A2M-Storage can be simpler than SUNDR, and guarantees linearizability instead of only fork consistency, thanks to the use of the trusted A2M module, which affords clients the ability to demand the latest committed request on a history, via a fresh END attestation.

The server maintains a version block, a snapshot of a file system captured by a Merkle tree, and two A2M logs. A version block holds a state digest (i.e., the root hash of a snapshot) computed as for SUNDR and a sequence number that tracks the latest A2M log sequence number with a signature signed by the latest writer. A2M has log q_h for the write request history, and log q_s for digests of version blocks, one for each state version generated by the application of writes to the state. Each write/read request is associated with a logical timestamp, of the form $\langle seq, att_{h,seq}, att_{s,seq} \rangle$, containing the request sequence number, the A2M attestation from the request history log q_h when that request was appended, and the A2M attestation from the state version log q_s when that request was executed. The client remembers the latest timestamp it has seen.

An A2M-Storage client performs write operations optimistically, assuming the timestamp it knows is the latest. When it submits a write request req for sequence number n, it also submits a nonce (for freshness), its known timestamp on which req is conditioned, and a new version block with sequence number n obtained after executing req. If the conditioned-on timestamp has not changed, the server modifies the state accordingly, stores the new version block that the client sends, and appends the request and state version digests to A2M logs q_h and q_s, respectively. In other words, execution of the request is conditioned on the latest timestamp at the server being the same as that known by the client. The server then forms its response, containing a success code, END attestations

[6]We omit the details of how files and directories are organized. What is important is that an entire file system can be cryptographically digested and verified against a set of digests efficiently.

from the two logs, and a proof that the operation was committed to the service state using the state digest function. The client accepts the response if the attestations and stage digest proofs are valid. If however the client had a stale timestamp, indicated by a failure code in the response, it updates its timestamp with the one returned by the server, and tries again potentially after fetching fresher state blocks and potentially backing off in case of write contention.

An A2M-Storage client performs read operations that include nonces. The server returns END attestations from the two A2M logs whose freshness is proven by a nonce, the version block to which the last A2M q_s entry points, and a proof that the read content is the valid part of the current snapshot. Note that the version block should include the same sequence number as the A2M attestation sequence number to be valid.

Instead of the optimistic, one-phase version of the protocol, a pessimistic two-phase version is straightforward as well, in which clients always fetch a "grant" to perform their operation at a particular sequence number, and then submit their operation with a guarantee of success, as per SUNDR.

In terms of its software architecture, A2M-Storage is similar to a version of SUNDR that entrusts the task of ordering requests and maintaining version structures to a separate, trusted component called a consistency server. In A2M-Storage, this task is "emulated" with the help of A2M, a general-purpose abstraction that works not only for SUNDR but also for other systems as we have demonstrated in other sections.

5.1.3 Correctness

A2M-Storage clients and server need maintain far less state than is necessary for SUNDR: clients only require a single global timestamp, instead of per-client version structures. Yet, A2M-Storage provides linearizability, because a client accepts a write operation as complete only when the server proves that the request is committed to its A2M logs – and A2M logs are trusted not to violate linearization. Similarly, a client accepts a read operation response as complete only when the response carries the latest timestamp, whose freshness is attested by the A2M module.

We show informally that there exists a sequential history of accepted writes, and that each read is partially ordered to the correct immediately preceding write. When a write operation is accepted by a client, we know that the operation is committed to A2M right after the conditioned-on timestamp. By following a chain of conditioned-on timestamps backwards, we can construct a single history of accepted client write operations. In addition, when a read is accepted by a client, we know that the read response carries the latest committed state version. The read operation can be placed right after the write that produces a state version attested by A2M and on which the read depends. Therefore, there exists a linearizable history of accepted write and read operations.

Since there is only one server, there is no guarantee on liveness when the server fails. Moreover, due to the nature of optimistic protocols, A2M-Storage does not provide any guarantees on fairness among clients; a greedy client can overuse the system.

5.2 A2M-Q/U

The Query/Update protocol (Q/U) [6] is a quorum-based BFT replicated state machine. It offers an optimistic protocol that completes client requests in a single round-trip message exchange between a client and the replicas, in the absence of faults and write contention. At a very high level, Q/U is similar to A2M-Storage (with more than a single server): the client sends a request along with its view of all replicas' latest timestamps, each of which contains a replica's history. Each replica commits the request if its local timestamp is compatible with the client's view; otherwise, e.g., if another client has already advanced that replica's state with another conflicting update request, the replica refuses to execute the request and sends back its latest replica history. A client is satisfied about its request's linearization if a quorum of replicas ($4f+1$ out of $5f+1$ total replicas) accept its request, making it *complete*. If fewer than $2f+1$ replicas have accepted the client's request, then it is *incomplete* and the client tries again after some back-off. When a client receives matching replies from between $2f+1$ and $4f$ replicas, the request is *repairable*. A client attempts to repair a repairable request, by trying to see if enough other replicas exist to make it complete, or by trying to convince other replicas to accept it. If a client's operation is complete, the protocol guarantees that, in any other quorum in the system, that operation would be repairable, a fundamental invariant for Q/U's linearizability guarantee.

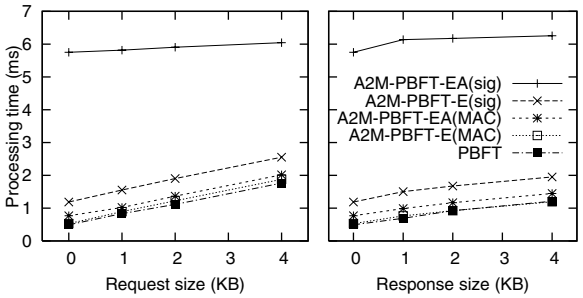

Figure 6: Microbenchmark results varying request (left) and response (right) sizes, measured in KBytes.

Q/U's linearizability properties stem from the sizes of the population N, quorums Q, and repairable sets R, given the number f of tolerable faults. A quorum must be always available even if all faulty replicas remain silent – implying $N \geq Q + f$ (1) – all quorums must intersect over a repairable set, excluding all faulty replicas – implying $2Q - N \geq R + f$ (2) – and all quorums must intersect over at least one non-faulty replica of all repairable sets of other quorums – implying $Q + R - N > f$ (3).

A2M's contribution to Q/U is that, by having replicas place accepted requests into A2M logs and having clients require an END attestation before accepting a replica's response, the sizes of quorum and repairable set intersections can be reduced. Essentially, $f+1$ replicas form a repairable set since faulty replicas commit to one history with A2M and they cannot form a repairable set with non-faulty replicas with an old history. Therefore, the above condition (3) changes to $Q + R - N \geq 1$. An A2M-enabled Q/U protocol can tolerate f faults with $N = 4f+1$, $Q = 3f+1$, and $R = f+1$, reducing the replication factor required from 5 to 4. We defer the full details to an extended version of this paper.

6. EVALUATION

In this section, we evaluate the overhead of applying A2M to BFT state machine replication. We have implemented A2M-PBFT-E and A2M-PBFT-EA (without its view change algorithm) in C/C++ with a BFT library [13, 36] ported to Fedora Core 6 and the SFSlite library [4]. The A2M protocols have versions that use signatures or MACs for authentication.

We ran our experiments with four replica nodes for A2M-PBFT-E and one client node. For A2M-PBFT-EA experiments, we use three replica nodes to tolerate one fault. The replica nodes are 1.8GHz Pentium 4 machines and the client node is a 3.2GHz Pentium 4 machine. All machines are equipped with 1GB RAM and

NFS Phase	-S	-PBFT	-A2M-PBFT-E (sig)	-A2M-PBFT-E (MAC)	-A2M-PBFT-EA (sig)	-A2M-PBFT-EA (MAC)
Copy	0.219	0.709	1.026	0.728	2.141	0.763
Uncompress	1.015	3.027	4.378	3.103	8.601	3.236
Untar	2.322	4.448	6.826	4.553	12.896	4.669
Configure	12.748	12.412	19.173	12.659	26.181	13.040
Make	7.241	7.461	9.778	7.500	11.379	7.510
Clean	0.180	0.298	0.640	0.312	0.742	0.311
Total	23.725	28.355	41.821	28.854	61.940	29.528

Table 1: Mean time to complete the six macrobenchmark phases in seconds.

Additional NFS- latency (μs)	A2M-PBFT-E (MAC)	A2M-PBFT-E (MAC) with batching	A2M-PBFT-EA (MAC)	A2M-PBFT-EA (MAC) with batching
1	28.854	28.763	29.528	29.505
10	29.598	29.025	31.299	30.188
50	32.735	30.232	36.242	32.214
250	48.784	37.237	66.441	45.199
1000	117.59	65.813	192.53	101.62

Table 2: Mean time to complete the six macrobenchmark phases in seconds for different A2M additional latency costs.

3Com 3C905C Ethernet cards, and are connected over a dual speed 10/100Mbps 3Com switch.

A2M uses SHA-1 as its digest function (also used for MACs), and NTT's ESIGN with 2048-bit keys for signatures. On a 1.8GHz machine, signature creation and verification of 20 bytes take on average $256\mu s$ and $194\mu s$, respectively.

All experiments used A2M as a library in the same address space as the PBFT protocol and the user application. However, depending on the A2M implementation scenario (see Section 3.3), A2M operations will experience a different additional interface latency cost. To account for the costs in accessing A2M, we impose by default $1\mu s$ of delay, which is a conservative system call latency[7] (Figure 4(d)) or a cross-SIP communication latency [18] (Figure 4(b)), to each A2M request using the Pentium RDTSC instruction.

In our experiments, we compare PBFT to A2M-PBFT-E and A2M-PBFT-EA, using two A2M implementations: one using signatures for authentication (denoted "sig") and one using MACs (denoted "MAC"). Shown PBFT measurements used MACs.

6.1 Microbenchmarks

We use a simple microbenchmark program, which is a part of the PBFT library. A simple client sends 100,000 null operation requests of size a bytes to replicas, which elicit replies of size b bytes from replicas. We ran experiments with a's and b's varying between 0 and 4000. Figure 6 plots the results. In all cases, operation turnaround times grow at the same pace with request/response sizes as in PBFT, with an additive overhead due to the additional A2M authentication operations (MACs or signatures) required. A2M-PBFT-E (MAC) and A2M-PBFT-EA (MAC) add a small extra cost because of the relative efficiency of MAC computation compared to the network delays. The signature-based versions of the protocol add significant computational overheads, and only become justifiable for very large replica populations, in which the cost of carrying MAC-based authenticators becomes comparatively expensive.

6.2 Macrobenchmarks: NFS

To understand the implications of using A2M-enabled protocols in real applications, we use PBFT's NFS front end on a PBFT (or A2M protocol) back end. As with BFS [13], we use a local NFS loop-back server and an NFS kernel client at the client side.

The workload we use consists of compiling a software package (`nano-2.0.3.tar.gz`) in six phases: 1) copy the file to the NFS file system (*copy*), 2) uncompress the file (*uncompress*), 3) untar the uncompressed file (*untar*), 4) run a configure script (*configure*), 5) compile the package by running make (*make*), and 6) clean up the built object and execution files (*clean*). The workload includes 8790 read-only BFT operations out of a total of 14500 operations invoked.

We compare six NFS-X protocols, where X is the name of the back-end protocol implementing the NFS interface. In addition to PBFT and our four A2M-enabled variants, we also run NFS-S, which uses a single server without replication. Table 1 shows the average time to complete each phase, out of 10 runs. The standard deviations of all results are within 4% of the mean. NFS-PBFT is 19.5% slower than NFS-S. NFS-A2M-PBFT-E (MAC) and NFS-A2M-PBFT-EA (MAC) are 1.8% and 4.1% slower than NFS-PBFT, respectively, whereas NFS-A2M-PBFT-E (sig) and NFS-A2M-PBFT-EA (sig) are 47.5% and 118.4% slower than NFS-PBFT, respectively. Overall, NFS-A2M-PBFT-E (MAC) and NFS-A2M-PBFT-EA (MAC) achieve significantly better fault tolerance at a slight increase in cost over PBFT.

6.3 Effects of A2M Placement

To explore the associated costs of other A2M implementation scenarios, we impose delays to each A2M request, varying delay duration from $10\mu s$ (for the order of magnitude of typical interprocess communication) to $1ms$ (for the order of magnitude of RPC on the same LAN).

Table 2 shows the average time to complete the macrobenchmark, out of 10 runs when the additional A2M interface latencies are 10, 50, 250, and $1000\mu s$. The mean times of NFS-A2M-PBFT-E (MAC) are 2.6, 13.5, 68.0, and 307.5% slower than the base NFS-A2M-PBFT-E (MAC) with $1\mu s$ delay; the slowdown corresponds to two delayed A2M operations and three A2M MAC verifications per BFT operation. For NFS-A2M-PBFT-EA (MAC), the mean times are 6.0, 22.7, 125.0, 552.0% slower than the base NFS-A2M-PBFT-EA (MAC) with $1\mu s$ delay; the slowdown is greater because of the greater number of A2M operations invoked during agreement steps.

[7] On a 1.8GHz Pentium 4 machine running Fedora Core 6, we ran lmbench [31] to measure the time to perform nontrivial entry into the operating system. The system call takes 0.87 μs in average.

To amortize the effect of this A2M access latency, we explore a multiple-operation batching optimization. In A2M-PBFT-E replicas bundle an `append` with its subsequent `lookup` when they send replies. In A2M-PBFT-EA replicas also bundle an `advance` with their subsequent `lookup` during agreement steps. Furthermore, the client batches A2M MAC verifications. When additional latencies are 1 and 10 μs, this batching effect is negligible. However, when additional latencies are 50, 250, and $1000\mu s$, A2M-PBFT-E with batching improves mean times by 7.6, 23.5, and 44.0% respectively and A2M-PBFT-EA with batching improves mean times by 11.1, 32.0, and 47.2% respectively.

7. THE RIGHT ABSTRACTION

In the previous sections, we have argued and experimentally demonstrated that systems incorporating in their design a small, trusted abstraction, A2M in our examples, can improve their fault tolerance at certainly tolerable cost. However, an interesting open question remains: is A2M the *right* trusted abstraction, for the types of applications we demonstrated here – state machines, replicated or centralized? Furthermore, is it the right trusted abstraction for other reliable applications that are more loosely organized than replicated state machines?

In systems that strive for linearizability, such as those forming the focus of our work here, the notion of a common event (i.e., request) history is central. Therefore, being able to commit to and compare histories seems, at a minimum, a required trusted function, which is exactly what A2M's log abstraction offers. Arguably, when histories need not be compared, as is the case when ensuring A2M-PBFT-EA replicas commit to their messages before sending them, it is sufficient to be able to commit to individual key-value pairs that are independent of all others, which is a narrower specification than what A2M offers. However, given that the difference between attested key-value pairs and attested logs is small (the computation of an incremental digest with every append), we opted to make a trusted log the basic, common abstraction that covers both replicated and single-server systems.

Would a larger trusted abstraction be preferable? Arguably, one could push an entire replicated state machine protocol, such as PBFT, into the trusted computing base. The application interface exported – an invocation method, and an execution callback [13] – is certainly simple, and applies to any deterministic application state machine. For example, one could imagine a trusted implementation of a fail-stop replicated state machine protocol, such as Paxos [24]. However, a replicated state machine abstraction, even one that is trusted not to be Byzantine, remains fairly complex to implement; it requires transmission and reception of network messages and several sets of local variables per request per remote replica. In contrast, A2M requires no network interactions, and only a circular buffer that tends to be short; although a hardware implementation of A2M appears trivial, a hardware implementation of Paxos might not be.

Beyond linearizable replicated state machines, an interesting question might be what other, orthogonal, trusted abstractions might make sense under different consistency requirements. For instance, when dispensing session guarantees weaker than linearizability (such as "read your writes" [35] or fork consistency [26]), simple trusted logical clocks [23] might be sufficient compared to an abstraction such as A2M.

8. RELATED WORK

Beyond related work we have presented as background, we address the following categories:

Trusted Devices: Trusted hardware, such as today's commodity Trusted Platform Module (TPM) hardware developed by the Trusted Computing Group [5], has been previously proposed, implemented, and marketed as a way to securely boot a sensitive host with approved software. Operations performed by the TPM are authenticated using a private signing key that resides on the module and cannot be retrieved or modified without physically destroying the module. Unfortunately, software is not bug-free, and even if correctly loaded at secure boot time, it can be overcome by exploits such as buffer overflows. As a result, while existing secure hardware can make machines strictly harder to compromise, it does not obviate the need for Byzantine-fault tolerant systems, nor does it improve their safety and liveness properties: it makes the likelihood of faults smaller, but does not improve fault bounds.

Shared Servers: Ivy [33] is a read/write peer-to-peer file system shared by multiple clients. A file system consists of a set of logs, each of which is owned by a participant who has a public-private key pair. A log is a list of immutable log records. Each log has a log-head that points to the most recent log record and the log-head is signed by the private key. A write appends a new log record and modifies the log-head to point to it. A read scans all log records owned by all participants of the file system to find appropriate information. A malicious server hosting the log-head can easily mount forking attacks by concealing log records depending on clients. With A2M, we can ensure that a malicious server tells the same sequence of log records including the most recent one. Note, however, that Ivy depends on a distributed hash table underneath, and any "strengthening" of the protocol must be predicated on a DHT with provable routing guarantees.

Plutus [20] is a shared storage system that enables file sharing without placing much trust in the file servers. All data is stored in an encrypted form, and key distribution is decentralized. A file system is represented by a hash tree whose root hash is signed. Plutus is also vulnerable to forking attacks wherein a malicious server can show different file system states to different clients.

Replicated State Machines: Byzantine-fault tolerant state machine replication has received much attention since PBFT [12] added the word "practical" in its title. Researchers have proposed several improvements on PBFT such as proactive recovery (PBFT-PR [13]), abstraction to tolerate non-determinism [36], and an architecture that separates execution from agreement to improve performance and confidentiality [39]. In all cases, however, no improvement can offer liveness and safety beyond the uniform $\lfloor \frac{N-1}{3} \rfloor$ fault bound. In Yin et al. [39], the architecture uses two groups of replicas – N agreement and M execution replicas – by dividing functionalities. This architecture can tolerate $\lfloor \frac{N-1}{3} \rfloor$ faults and $\lfloor \frac{M-1}{2} \rfloor$ faults. A2M-enabled protocols divide functionalities into committing a sequence of protocol steps to A2M and performing an original protocol. A2M-PBFT-EA can tolerate $\lfloor \frac{N-1}{2} \rfloor$ faults out of N total replicas since A2M is in a trusted computing base. Compared to agreement replicas, A2M is a small, general-purpose mechanism that is applicable to various protocols to defend against equivocation.

Recently, BFT2F [27], a PBFT variant, uses some of the ideas in SUNDR to provide linearizability and liveness up to $\lfloor \frac{N-1}{3} \rfloor$ faults, and a weaker safety property called fork* consistency without liveness for up to $2\lfloor \frac{N-1}{3} \rfloor$ faults, relying on clients' help to protect consistency. With the help of A2M, A2M-PBFT-E can instead guarantee linearizability up to $2\lfloor \frac{N-1}{3} \rfloor$ faults, and A2M-PBFT-EA guarantee both linearizability and liveness up to $\lfloor \frac{N-1}{2} \rfloor$ faults.

BAR [7] fault tolerance contains a notion of protocol-action commitment (to a quorum maintained by replicas themselves) to capture rational behavior. Also, PeerReview [16], CATS [40], and

Timeweave [30] use authenticated histories to allow fault detection given a replica's self-inconsistent history; this might be a helpful mechanism to allow A2M-based protocols to recover even when the safety fault bound is (temporarily) violated.

A2M-PBFT-EA bears a close resemblance to Paxos [24] in that they both require quorum size $\lfloor \frac{N-1}{2} \rfloor + 1$. Paxos assumes benign faults, and it is live as long as fewer than one half replicas are faulty but is safe with up to N faults. In contrast, A2M-PBFT-EA assumes Byzantine faults, but thanks to A2M a faulty node can stop or lie consistently to other replicas. A2M-PBFT-EA is both safe and live when fewer than one half replicas are faulty, but when this assumption is violated, there is no guarantee on safety and liveness.

Symmetric-Fault Tolerance: Researchers have described *symmetric faults* [38] as a specialization of Byzantine faults, and shown that for agreement protocols, a hybrid fault model that is a mixture of non-malicious faults (of size b), malicious symmetric faults (of size s), and malicious asymmetric faults (of size a) can lead to more flexible tolerance guarantees. In Thambidurai and Park [38], a modified version of the classic synchronous Oral Messages (OM) agreement algorithm can tolerate $a + s + b$ faults when $N > 2a+2s+b+r$ (for $a \leq r$) where r is the number of rounds of message exchange excluding initial transmission. Follow-on work includes analyses of fault bounds on synchronous and asynchronous approximate agreement under the hybrid fault model [9, 21]. In contrast, we focus on providing a practical, generic, small primitive that prevents equivocation and limits Byzantine hosts to behave symmetrically. We hope to explore further whether A2M can be used as a systematic way to make Byzantine faults symmetric, admitting simpler protocols with greater fault tolerance.

Abstract Shared Objects: Fleet [29] uses a consensus protocol by performing read and append operations on Timed Append-Only Arrays (TAOAs), which are single-writer multi-reader objects to which clients can append values and from which clients can read values. Each appended value is tagged with a logical timestamp vector. A TAOA is emulated by a distributed client-server protocol built atop a b-masking quorum system [28], which requires $N > 4b$ to tolerate b Byzantine faults. Unless this fault bound is violated, a TAOA provides the following properties: values are appended in a sequential order; values appended are not modified or deleted; and timestamps partially capture the order of values that different clients append. In contrast, A2M is a local primitive that can be used to enforce a node to commit to a sequential order of operations. Our goal is to slightly grow the trusted computing base to strengthen distributed trustworthy abstractions such as replicated state machines and shared storage built atop the base. In fact, implementing Fleet's TAOA and consensus protocol could be simplified if servers employ A2Ms.

9. CONCLUSIONS

In this paper, we present a trusted, log-based abstraction called Attested Append-Only Memory (A2M). Servers utilizing A2M are forced to commit to a single, monotonically increasing sequence of operations. Since the sequence is externally verifiable, malicious servers cannot present different sequences to different parties. We discuss several implementation scenarios of A2M under different threat models. We present A2M-PBFT-E, a simple variant of Castro and Liskov's PBFT protocol that can achieve safety with up to $2\lfloor \frac{N-1}{3} \rfloor$ faulty replicas. We also present A2M-PBFT-EA, a more involved variant, that can preserve safety and liveness with up to $\lfloor \frac{N-1}{2} \rfloor$ faulty replicas. Finally, we show how to achieve linearizability in single-server storage systems such as SUNDR. Our prototype implementations of A2M-PBFT-E and A2M-PBFT-EA show minor performance overhead; they are 1.8% and 4.1% slower than the PBFT base case, respectively. There are many technical details in this paper, but the bottom line is that A2M is a practical and eminently implementable tool for improving the fundamental Byzantine fault tolerance of replicated and centralized systems alike.

Acknowledgments

We would like to thank Lorenzo Alvisi, Eric Brewer, Mike Dahlin, Andrey Ermolinskiy, and Prince Mahajan for their constructive feedback. We also would like to thank the anonymous reviewers for their comments and our shepherd, Peter Druschel, for his guidance.

10. REFERENCES

[1] Amazon S3. `http://aws.amazon.com/s3/`.

[2] Intel Active Management Technology (AMT). `http://www.intel.com/technology/platform-technology/intel-amt/index.htm`.

[3] Java. `http://java.sun.com/`.

[4] SFSlite. `http://www.okws.org/doku.php?id=sfslite`.

[5] Trusted Computing Group (TCG). `http://www.trustedcomputinggroup.org/`.

[6] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-scalable byzantine fault-tolerant services. In *Proc. of SOSP*, 2005.

[7] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth. BAR fault tolerance for cooperative services. In *Proc. of SOSP*, 2005.

[8] T. W. Arnold and L. P. V. Doorn. The IBM PCIXCC: A new cryptographic coprocessor for the IBM eServer. *IBM Journal of Research and Development*, 48(3/4):475–487, 2004.

[9] M. H. Azmanesh and R. M. Kieckhafer. New hybrid fault models for asynchronous approximate agreement. *IEEE Trans. on Computers*, 45(4):439–449, 1996.

[10] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. of SOSP*, 2003.

[11] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn. vTPM: Virtualizing the trusted platform module. In *Proc. of USENIX Security*, 2006.

[12] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. of OSDI*, 1999.

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

[14] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Proc. of OSDI*, 2006.

[15] S. Haber and W. S. Stornetta. How to time-stamp a digital document. *Journal of Cryptology*, 3(2):99–111, 1991.

[16] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *Proc. of SOSP*, 2007.

[17] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, 1990.

[18] G. Hunt, M. Aiken, M. Fähndrich, C. Hawblitzel, O. Hodson, J. Larus, B. Steensgaard, D. Tarditi, and T. Wobber. Sealing OS processes to improve dependability and safety. In *Proc. of EuroSys*, 2007.

[19] G. Hunt and J. Larus. Singularity: Rethinking the software stack. *Operating Systems Review*, 41(2):37–49, 2007.

[20] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Proc. of USENIX FAST*, 2003.

[21] R. M. Kieckhafer and M. H. Azamanesh. Reaching approximate agreement with mixed mode faults. *IEEE Trans. on Parallel and Distributed Systems*, 3(1):53–63, 1994.

[22] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *Proc. of SOSP*, 2007.

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

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

[25] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. on Programming Languages and Systems*, 4(3):382–401, 1982.

[26] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proc. of OSDI*, 2004.

[27] J. Li and D. Mazières. Beyond One-third Faulty Replicas in Byzantine Fault Tolerant Systems. In *Proc. of NSDI*, 2007.

[28] D. Malkhi and M. Reiter. Byzantine quorum systems. In *Proc. of STOC*, 1997.

[29] D. Malkhi and M. K. Reiter. An architecture for survivable coordination in large distributed systems. *IEEE Trans. on Knowledge and Data Engineering*, 12(2):187–202, 2000.

[30] P. Maniatis and M. Baker. Secure history preservation through timeline entanglement. In *Proc. of USENIX Security*, 2002.

[31] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *Proc. of USENIX ATC*, 1996.

[32] R. C. Merkle. A digital signature based on a conventional encryption function. In *Proc. of CRYPTO*, 1987.

[33] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proc. of OSDI*, 2002.

[34] M. Naor. Bit commitment using pseudorandomness. *Journal of Cryptology*, 4(2):151–158, 1991.

[35] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. Demers. Flexible update propagation for weakly consistent replication. In *Proc. of SOSP*, 1997.

[36] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using abstraction to improve fault tolerance. In *Proc. of SOSP*, 2001.

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

[38] P. Thambidurai and Y.-K. Park. Interactive consistency with multiple failure modes. In *Proc. of SRDS*, 1988.

[39] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proc. of SOSP*, 2003.

[40] A. R. Yumerefendi and J. S. Chase. Strong accountability for network storage. In *Proc. of USENIX FAST*, 2007.

APPENDIX

A. A2M-PBFT-EA VIEW CHANGE

When replica i in view v_{from} suspects the primary is faulty as per the PBFT protocol, it broadcasts to R its intent to change views via a $\langle \text{WANTVIEWCHANGE}, v_{to}, z, i \rangle_{i,R,1}$ message, where z is a fresh nonce and v_{to} is $v_{from} + 1$ if the replica was not already in the midst of a view change, or $v + 1$ if the replica was in the process of switching to view v when it decided to change yet again.

When a replica collects a WANTVIEWCHANGE certificate that consists of $\lfloor \frac{N-1}{2} \rfloor + 1$ valid WANTVIEWCHANGE messages for the same view v_{to}, it computes the appropriate nonce $\mathcal{Z}$ for its attestations by hashing together all the nonces in its WANTVIEWCHANGE certificate in increasing replica identifier order. It abandons its current view v_{from} if $v_{from} < v_{to}$ (or its participation in a prior view change protocol towards view v' if $v' < v_{to}$), as well as all intervening views up to v_{to}. For all views v in $[v_{from}, v_{to})$ in order, the replica performs $\texttt{advance}(m_c, [v + 1|0] - 1, 0, 0)$ (if it has not already); $[v + 1|0] - 1$ is the last COMMIT log entry belonging to view v. Now the replica constructs its VIEWCHANGE message.

The message form is $\langle \langle \text{VIEWCHANGE}, v_{from}, v_{to}, n, s, C, P, Q, W, A, B, H \rangle, \langle \mathcal{E} \rangle_{A2M_i,R,1} \rangle$. Among the contents of the main message, v_{to}, n, s, and C are as in regular PBFT; v_{from} is as defined above, Q is the set of committed certificates with sequence number higher than n and P is the set of prepared certificates for requests that are prepared but are not committed after n, W is a WANTVIEWCHANGE certificate, A is the set of A2M COMMIT log attestations corresponding to the certificates in P, B contains the view abandonment attestations from the replica's COMMIT log (see below), and finally H is a list of committed request log entries that attest those requests in Q. $\langle \mathcal{E} \rangle$ is the attestation from the sender's A2M message log for VIEWCHANGE messages, computed via a $\texttt{lookup}(m_{vc}, v_{to}, v_{to})$ A2M command.

For each abandoned view v between v_{from} and v_{to}, the set B contains the attestation $\langle \text{LOOKUP}, m_c, [v|n' + 1], \mathcal{Z}, 0, \text{SKIPPED}, [v + 1|0] - 1, d' \rangle_{A2M_i,R,1} \rangle$, where m_c is the COMMIT log identifier, and n' is the highest sequence number in Q and P. For each abandoned view, this attestation shows that the replica could not have committed a request for a sequence number greater than those included in its Q and P sets.

When a new primary ($p = v_{to} \bmod N$) collects a new view certificate V that consists of $\lfloor \frac{N-1}{2} \rfloor + 1$ valid VIEWCHANGE messages that have the same v_{from} and v_{to} and contain correct C, P, Q, W, A, B, and H, it multicasts to R a NEWVIEW message of the form $\langle \langle \text{NEWVIEW}, v_{to}, V, O_c, O_p \rangle, \langle \mathcal{E} \rangle_{A2M_p,R,1} \rangle$; the latter part is the usual A2M attestation for the message, whereas the contents of the message are a new view certificate, with the set O_c containing PREPREPARE messages for requests to be committed, and the set O_p containing PREPREPARE messages for requests to be prepared in view v_{to}. When a replica receives the valid NEWVIEW message, it enters view v_{to}. Any requests in prepared or committed certificates for sequence numbers later than the latest stable checkpoint are prepared (issuing a new attested COMMIT message) and committed (appending the request in the request log if not already there) in order, without need for further inter-replica communication.

Note that all VIEWCHANGE messages within a NEWVIEW certificate must have the same v_{from}; this is essential for the correctness properties described next. If the primary fails to collect a quorum of such messages, it refuses to generate a NEWVIEW message. To ensure progress, any non-faulty replica that receives a VIEWCHANGE message with a v_{from} later than its own asks the issuer of that message for the NEWVIEW certificate that allowed it to enter v_{from}. Using that certificate, the lagging replica can bring itself to that view. When a timeout indicates that the previous view change attempt stalled – either due to a faulty new primary or because of v_{from} mismatches – the replica initiates another view change for the next target view number. Thanks to the eventual synchrony of our network, this guarantees that eventually enough replicas will initiate a view change with the same v_{from} and the change will go through.

B. A2M-PBFT-EA LIVENESS PROOF

We show that no two non-faulty replicas can place different requests in the same sequence number of the A2M request log. We split our argument into a same-view case, and a different-view case.

Case 1 – Same View: Suppose two non-faulty replicas have appended two different requests to the same sequence number of their respective A2M request logs, during the same view. They both did that after having constructed a valid committed certificate over two quorums. Those two quorums must have at least one common (perhaps faulty) replica i, which managed to attest to two COMMIT messages, one for each request, in each of the two quorums. This, however, is a contradiction with our assumption that A2M is trusted to avoid equivocation for the same log entry, and the collision-resistance of the hash function.

It is worth noting that along similar lines, it is trivial to show that no two non-faulty replicas can be convinced to place different requests in their COMMIT A2M log for the same sequence number and view, by the analogous argument on the prepared certificate quorums and the PREPARE A2M log of the common replica. Finally, the exact same argument can be used to show that no two non-faulty replicas can put different requests in their PREPARE A2M logs for the same sequence number and view, since the single primary for the view can only attest to a single PREPREPARE message for that sequence number in any given view.

Case 2 – Different Views: Now we must show that no two non-faulty replicas can commit two requests r and $r' \neq r$ in sequence n and in views v and $v' > v$, respectively.

We define an *active* view as a view for which a valid NEWVIEW certificate has been constructed *and* seen by a non-faulty replica. A non-faulty replica cannot commit a request in a view for which it has not seen a valid NEWVIEW certificate, therefore if a non-faulty replica commits a request in a view, then that view must be active.

We split our argument into two further subcases, first the case in which no other active views exist between v and v', and the case in which at least one active view exists between v and v'.

Case 2a – v and v' are consecutive active views: Since no other active views exist between v and v', then the NEWVIEW certificate for v' – and there can be at most one since only one NEWVIEW message can be attested by the primary for view $v_{to} = v'$ – must have $v_{from} \leq v$. This is because at least one non-faulty replica must have produced a VIEWCHANGE message for the certificate, and that non-faulty replica guarantees that its v_{from} represents an active view, which cannot be later than v (or it would have to be v'). As a result, this NEWVIEW certificate contains view abandonments for all views in its $[v_{from}, v_{to})$ range, which includes $[v, v')$.

Now consider three quorums, the one that produced the committed certificate for r in view v (denoted $\mathcal{Q}$), the one that produced the NEWVIEW certificate to v' (denoted $\mathcal{V}$), and the one that produced the committed certificate for r' in view v' (denoted $\mathcal{Q}'$). Let $i \in \mathcal{Q} \cap \mathcal{V}$, which always exists thanks to quorum intersection.

Replica i unavoidably contributed an attested COMMIT message for r at sequence number n in the committed certificate for v along with the rest of quorum $\mathcal{Q}$. What can have been i's VIEWCHANGE contribution to the NEWVIEW certificate in quorum $\mathcal{V}$ with regards to sequence number n? If i reported a valid stable checkpoint no earlier than n in its VIEWCHANGE, then the resulting, unique NEWVIEW certificate for v' should convince any non-faulty replica that sees it to never commit anything else at n in view v', since n belongs in the past; this contradicts our assumption that some non-faulty replica will in fact commit r at n in view v'.

If instead i reported a stable checkpoint earlier than n in its VIEWCHANGE, it can only have reported the same COMMIT attestation for request r at n, since that VIEWCHANGE message must contain a view abandonment for v as we showed above, and omitting an attestation for the COMMIT log entry $[v|n]$ is not an option; to omit it successfully, it would have to produce an abandonment attestation $\langle \text{LOOKUP}, m_c, [v|n'+1], \mathcal{Z}, 0, \text{SKIPPED}, [v+1|0]-1, d' \rangle_{A2M_i,R,1}$ for some $n' < n$, which is disallowed by the A2M interface given the existence of an ASSIGNED attestation for entry $[v|n]$ and the inequality $[v|n'+1] \leq [v|n] < [v+1|0] - 1$.

This leaves the common replica i between quorums $\mathcal{Q}$ and $\mathcal{V}$ only with the option of reporting request r as prepared in view v. As a result, any correct replica in quorum $\mathcal{Q}'$, which can only commit requests in view v' after having seen the NEWVIEW certificate for that view, must have issued at least a PREPARE message for request r in view v' while processing the NEWVIEW certificate. However, since this replica is also a member of the committed certificate for request r' in view v', it must also have prepared and subsequently committed that request r'. This clearly contradicts not only the properties of the A2M message logs at that replica, but also the operation of a non-faulty replica. This completes the proof for this subcase.

Case 2b – v and v' are not consecutive active views: Suppose there are $v_1, v_2, ..., v_{k-1}$ active views between $v(= v_0)$ and $v'(= v_k)$. We can prove inductively on the intervening active views that at least a prepared certificate for request r at sequence n will be propagated to view v', preventing a commitment of a conflicting request r' at the same sequence number there.

In the base case, we can use the argument of the previous subcase 2a to show that the NEWVIEW certificate for view v_1 will either preclude any subsequent commitment to sequence n or will contain at least a prepared certificate for request r at that sequence number.

To show the inductive step, assume that the NEWVIEW certificate for view v_i contains a prepared certificate for request r – that is the only viable choice since, if it contains a stable checkpoint for n or later, then no subsequent view will admit a different committed request r', leading to a contradiction. Now consider the NEWVIEW certificate, formed by quorum $\mathcal{V}$, that will lead away from v_i to v_{i+1}. Any non-faulty replica in $\mathcal{V}$ (there must be at least one), must have seen the earlier NEWVIEW certificate leading to v_i, or else it would be unable to assume v_i as its active view. Therefore, that replica must also have prepared that same request r in view v_i, including the prepared certificate in its VIEWCHANGE contribution to the later NEWVIEW certificate.

The induction proves that committed request r at n in active view v will either preclude the commitment of another request at n in view v' (because somewhere in between a NEWVIEW certificate contained a stable checkpoint for a sequence at or after n), or cause the inclusion of a COMMIT attestation for the same r at n in all subsequent valid NEWVIEW certificates. This contradicts the assumption that a non-faulty replica at active view v', which must have seen such a NEWVIEW certificate, will commit request r' at n in view v'. This last subcase concludes the proof that two commitments for the same sequence number at different non-faulty replicas must commit the same request.

Beyond quorum availability (i.e., ensuring that no quorum can be blocked from forming due to non-faulty replicas caused to commit incorrect requests), A2M-PBFT-EA also guarantees that no replica is left behind during view changes: a replica only abandons its current view v if it has collected a WANTVIEWCHANGE certificate; even if the current view change does not complete due to network faults or a faulty new primary, the replica can retransmit the WANTVIEWCHANGE certificate until eventually enough other non-faulty replicas have received it to complete the view change, or to trigger another one with a different primary. This is guaranteed by the eventual synchrony of our network and processing model.