ASPAS: As Secure as Possible Available Systems

Houssam Yactine[1], Ali Shoker[2], and Georges Younes[1]

[1] HASLab, INESC TEC & University of Minho,
Braga, Portugal {houssam.a.yactin,georges.r.younes}@inesctec.pt
[2] VORTEX Colab, Porto, Portugal
{ali.shoker}@vortex-colab.com

Abstract. Available-Partition-tolerant (AP) geo-replicated systems trade consistency for availability. They allow replicas to serve clients' requests without prior synchronization. Potential conflicts due to concurrent operations can then be resolved using a conflict resolution mechanism if operations are commutative and execution is deterministic. However, a Byzantine replica can diverge from deterministic execution of operations and break convergence. In this paper, we introduce ASPAS: As Secure as Possible highly Available System that is a Byzantine resilient AP system. ASPAS follows an optimistic approach to maintain a single round-trip response time. It then allows the detection of Byzantine replicas in the background, i.e., off the critical path of clients requests. Our empirical evaluation of ASPAS in a geo-replicated setting shows that its latency in the normal case is close to that of an AP system, and one order of magnitude better than classical BFT protocols that provide stronger (total ordering) guarantees, unnecessary in AP systems.

Keywords: Availability; Integrity; Consistency; CRDT; BFT

1 Introduction

In the context of the CAP theorem [14], an Available-Partition-tolerant (AP) geo-replicated system trades consistency for availability. The design follows *optimistic replication* with a relaxed consistency model [22, 27, 18] where a replica immediately replies to the client without prior synchronization with other replicas. The essence is to reduce the response delay to a single round-trip between a replica and a client. This is desired in latency-sensitive applications where a soft (stale) state is accepted. Nevertheless, concurrent updates can lead to inconsistency that must be resolved (in the background).

This paradigm is recently getting attention with the advent of conflict resolution mechanisms as Last-Writer-Wins, Cloud Types, and Conflict-free Replicated DataTypes (CRDTs) [5, 25, 2]. Such conflict resolution mechanisms ensure *Strong Eventual Consistency (SEC)* [24] that guarantees convergence across system replicas if operations are designed to be commutative and only if faults are benign (see definition 1 in Section 3.1 for details). The idea is to relax the replica's state from being totally ordered log to a partially ordered log (POLog) [2]. Being

commutative, executing operations in the POLog will lead to equivalent states if replicas are deterministic. Since it is impractical to synchronize or "pause" an AP system to do this integrity check, it is acceptable for concurrent clients at distinct replicas to (temporarily) observe different states. This makes it hard to differentiate a deterministic execution of concurrent operations in a partial order from a non-deterministic execution caused by a Byzantine (a.k.a., arbitrary or malicious) [20] replica. This can lead to permanent system divergence.

An intuitive option to address the Byzantine problem is to assess the feasibility of classical BFT protocols [8, 16, 1, 10, 29] to an AP system. This approach was followed in OBFT [9] where the authors deferred the synchronization (consensus) of replicas, executing concurrent operations, to epochs where the entire system is "paused". A similar approach was followed in Zeno [26] that reduced the quorum size under network partitions, but imposed total ordering within a partition before replying to clients. Unfortunately, these approaches are not preferred in AP systems where the expected latency is a single round-trip.

In this paper, we introduce a novel As Secure as Possible AP System (ASPAS) that guarantees Strong Convergence (of SEC) and provides Byzantine security without compromising availability. As shown in Figure 1, ASPAS runs a frontend layer composed of loosely connected application servers (appservers) that can serve clients' requests without prior synchronization. The replica of an appserver is a datatype following the SEC model, e.g., a CRDT, in order to ensure convergence despite concurrent operations. A correct client connects to a single associated appserver. After executing the client's operation locally, the appserver propagates the request to other appservers (in the background) using a Reliable Causal Broadcast [3, 4, 15] (RCB). Delivered operations via the RCB are then executed in a deterministic way by correct appservers. In the backend, and to detect Byzantine appservers, correct appservers asynchronously forward their operations to a BFT cluster (see Figure 1). This process is decoupled from the client-appserver communication to maintain the desired latency in AP applications, i.e., a single round-trip delay. The BFT cluster is responsible for extracting, out of the different appserver POLogs, a log of *stable* operations: non-concurrent operations that are already delivered and executed by all appservers. The BFT cluster then generates a log "certificate" that ensures the log integrity up to a common state version across appservers. The BFT cluster sends the certificate to correct appservers that piggy-back it to clients asserting the integrity of data up to that state version.

An alternative possible design to the three-tiered design of ASPAS (in Figure 1) is a two-layered design using Secure RCB [7, 21, 15]. Such RCB protocols are quorum-based and, thus, require synchronization before replying to clients which increases the latency. Therefore, the prospective design ends up having another layer (but running on the same appservers this time) for arbitration, i.e., another BFT protocol to compare stable partially ordered log versions up to some offset. This coupling of delivery and arbitration on the same appserver leads to several drawbacks, among them: (1) Classical BFT protocols scale linearly $O(N)$ with the number of appservers N in the system. It is more reasonable

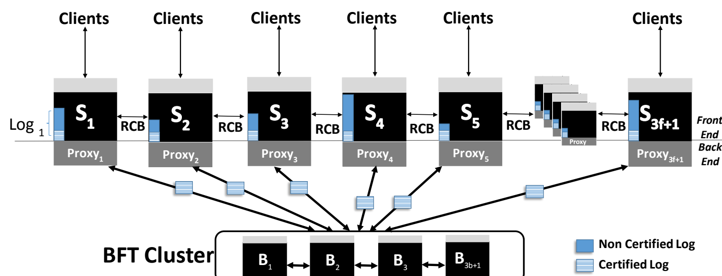


Fig. 1: ASPAS Architecture.

to decouple the execution layer from the arbitration layer as in ASPAS where the arbitration protocol runs on a smaller set of $n << N$ BFT servers. (2) Imposing high inter-replica delays (e.g., up to 100ms in geo-scalable settings) on the BFT protocol in the arbitration layer. This inter-replica overhead can be mitigated significantly if the BFT protocol runs in a well connected cluster as we do in ASPAS. (3) Imposing more memory and computational overhead on appservers due to the heavy-weight cryptographic and messaging work in BFT protocols. ASPAS offloads this overhead to the BFT cluster, and thus keeps appservers dedicated to serving clients' requests.

The three-tier design of ASPAS allows different clients to choose their level of security based on their desired freshness and availability-security tradeoff (hence the name "As Secure As Possible"). It can provide high security as close as classical BFT protocols, or high availability as close as AP systems, and a wide spectrum of options in between. However, this entails two main challenges we solve in ASPAS. The first is generating a consistent certificate despite the execution of operations in different orders on different appservers (a native pattern in AP systems). The second is to prevent the appserver from sending certificates or use different logs for different clients. This is important since client's requests in AP systems are handled by only one appserver (without consensus).

ASPAS can be used in any geo-replicated application that adopts SEC [24] and tolerates a window of unconfirmed operations. In fact, SEC-based applications trade consistency (i.e., a correctness property) for availability in the fault-recovery model. Following the same analogy, we argue that these applications also favor availability over Byzantine integrity (i.e., a correctness property) provided that the system state eventually converges. In addition, an end-user client is guaranteed to be notified of a Byzantine fault and get a correct state within a predefined tolerance window of operations. Note that this is commonly accepted in several SEC-based applications, e.g., social networks functionalities like: number of Likes, number of comments, list of comments, recommended media, number of Ads, shopping carts, real time collaborative editing, etc., as long as the system eventually converges to a correct state that the client can observe. Non end-user stateful applications can also roll-back recent changes if required. This is very common in AP-based applications where the client plays the role of a

cache or *Edge* computing node that is used as backend to end-user applications. This is driven by Edge systems in open volunteer networks like Guifi.net monitoring that used to monitor the network state, and Content Delivery Networks where a cache is used to boost the reply to end-user browsers.

We implemented ASPAS using BFT-SMaRt [1] as a backend BFT Cluster due to its well tested Java implementation (but any classical BFT protocol can be used). We conducted an empirical evaluation using YCSB benchmark workloads [11] in geo-replicated setting. We compared ASPAS with OBFT [9] as a state of the art protocol of the same SEC-based class, and with baseline configurations: AP system alone, and BFT protocol alone. The results show that the normal case latency of ASPAS is close to classical AP systems, and one order of magnitude lower than classical BFT protocols in the geo-replicated settings.

The rest of the paper is organized as follows. We start with presenting the most related works in Section 2. We then present the problem definition and ASPAS as a proposed solution in Section 3. Next, we present the empirical evaluation in Section 4, and we conclude in Section 5.

2 Related work

Byzantine Fault Tolerant (BFT) protocols often follow quorum-based State Machine Replication (SMR) [8, 16, 1, 10, 29, 23] to ensure total ordering despite the existence of a fraction f (out of $2f + 1$ or more) Byzantine replicas. These protocols, including the more scalable ones in the Blockchain realm [17, 29], are known of their high latency due to the cost of consensus. This encouraged the introduction of more relaxed agreement protocols whose latency is low enough to be used in AP systems. To provide Byzantine tolerance under partitions, Zeno [26] lets clients of different partitions to miss the updates of replicas under network partitions until it heals. However, Zeno exhibits a consensus overhead by imposing total ordering of updates within the same partition before replying to clients. On the contrary, ASPAS maintains a single round-trip latency as it makes use of CRDTs to resolve conflicts and delegates the (costly) consensus entirely to the background. Similarly, in the more practical work [13] for hardening Cassandra against Byzantine failures, a Write request is confirmed only after obtaining signed responses from a quorum of nodes—in the critical path of clients' requests. In ASPAS, the client never blocks on Writes; it applies them locally and delegates the integrity checking to a background process. Furthermore, Byzantine reliable causal broadcast (BRCB) protocols such as [3, 4, 21, 15] are used to propagate updates to different replicas despite the presence of Byzantine replicas. Such protocols do not guarantee SEC by default, because a Byzantine replica can execute local operations incorrectly to impede convergence. Including execution will again require a costly quorum-based protocol.

The most related work to ours is OBFT [9], which is an AP system that tries to ensure SEC under Byzantine faults. Since concurrent updates on different replicas may lead to conflicts, OBFT periodically "pauses" client's requests until it achieves convergence. Although it matches the different message logs and uses

CRDTs to ease the merging, as we do in ASPAS, this is done in a blocking way which is unacceptable in AP applications. ASPAS avoids fiddling with the client-appserver message exchange to maintain low latency, and delegates the integrity checking to a backend BFT cluster off the critical path of clients requests.

3 ASPAS Protocol

3.1 Problem

Modern geo-replicated AP systems trade consistency for availability. They support concurrent Write/Read operations, which allows client applications to observe—as fast as possible—different states at different replicas. Despite this, the entire system should not be broken, by diverging permanently, and thus it follows the Strong Eventual Consistency (SEC) [24] model to eventually ensure convergence.

Definition 1 (Strong eventual consistency (SEC)). *An object is* Strongly Eventually Consistent *if the following properties are satisfied:*

- Eventual delivery*: An update delivered at some correct replica is eventually delivered to all correct replicas:* $\forall i, j : f \in c_i$ *then* $\Diamond f \in c_j$
- Termination*: All method* f *executions terminate.*
- Strong Convergence*: Correct replicas that have delivered the same updates, even in different orders, have equivalent state:* $\forall i, j : c_i = c_j$ *then* $s_i \equiv s_j$.

As depicted in Def. 1, the Strong Convergence (third) property of SEC extends the Eventual Consistency [27, 22] model by stating that the execution of the same set of operations in different partially ordered logs should lead to an equivalent state. However, this (equation $c_i = c_j$ then $s_i \equiv s_j$) holds true only if the execution is deterministic, which cannot be guaranteed if the executing replica is Byzantine. Therefore, the presence of a single Byzantine replica can render the entire system convergence impossible.

As a simple example in a social network application, different clients on different replicas may *forever* view different lists of comments for the same post, even if no actions affect that post any more. Notice that this is even harder to detect when the AP system is in action due to Eventual Consistency and the lack of consensus between replicas.

3.2 System and fault models

We address ASPAS, a three-tier system model (sketched in Figure 1) composed of frontend and backend. The frontend follows the geo-replicated AP system model in which $3f + 1$ appservers are geographically located and fully replicated where at most f appservers are assumed to be Byzantine. Replicated data is assumed to satisfy SEC [24], e.g., CRDTs [25, 2]. CRDTs are backed by partially ordered logs defined using appservers' Version Clocks [19]. In ASPAS, every client has connections to all appservers, but issues its requests to a single appserver, likely

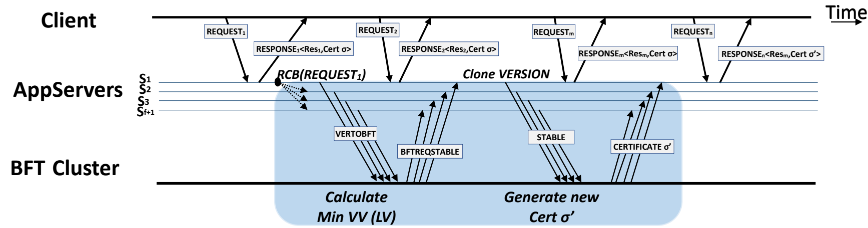


Fig. 2: Messages exchange pattern showing normal update operation and the certification steps between appservers and BFT cluster in background.

the closest one. In the backend, appservers propagate the received operations to each other using a Reliable Causal Broadcast (RCB) [3]. Appservers also push their requests to a BFT cluster that runs a classical BFT SMR protocol, e.g., [8, 1]. The cluster is composed of $3b+1$ BFT replicas (bftservers), where at most $b << f$ of them are assumed to be Byzantine. The network may (not infinitely) fail to deliver, corrupt, delay, or reorder messages. Byzantine appservers, replicas, and clients may either behave arbitrarily, i.e., in a different way to their designed purposes, or just crash and recover (benign faults). A strong adversary coordinates Byzantine replicas or appservers to compromise the replicated service and thus bring the appservers to inconsistent states. However, we assume that the adversary cannot break cryptographic techniques like: collision-resistant hashes, encryption, and signatures. We also assume all nodes to have unique identities and cryptographic keys distributed through a trusted mechanism.

3.3 An overview of ASPAS

ASPAS is composed of a frontend and backend. The frontend runs a loosely coupled geo-replicated AP service over dozens of appservers, to which clients issue their requests. The backend runs a smaller black-box BFT cluster, to which appservers send their operation logs to assert their consistency up to a certain version. To thwart blocking and delays in the AP service, these two layers are completely decoupled and operate concurrently; clients only interact with appservers, i.e., they are agnostic of the BFT cluster.

As depicted in Figure 2, ASPAS runs two concurrent phases: *Normal operation* and *Certification* (presented next in detail). The former represents the normal operation of an AP system: a client sends its requests to a single associated appserver at a time (to avoid Byzantine clients issuing different requests to different appservers). The appserver replies to the client immediately, and then propagates the operations to its counterpart appservers. The latter executes (or merge) these operations locally. Using CRDTs techniques, convergence is guaranteed as long as appservers are not Byzantine. To guard against Byzantine appservers (that can cause divergence), correct appservers piggyback a "certificate": a signed message issued by the BFT cluster in the certification phase to assert a consistent appserver state up to a certain version. In the certification phase, appservers periodically send their (partially-ordered) logs to the BFT

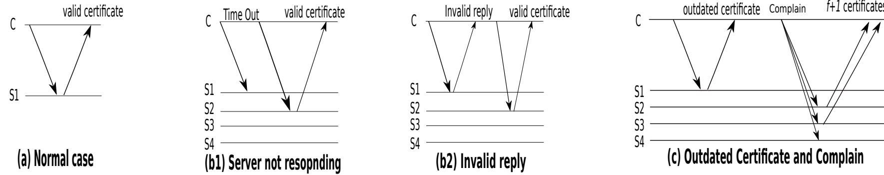


Fig. 3: Different messaging patterns of ASPAS.

cluster. The latter issues a certificate to correct appserver whose logs are matching to a certain (incremental) state version. Choosing the version and extracting the corresponding logs are explained in the mechanisms: **Stable versions** and **Log extraction** in the Section 3.6.

In the case of an invalid certificate, i.e., unauthentic, non-matching or outdated, the client launches a Complaining process (see the **Complaining** mechanism in Section 3.6) to notify about possible Byzantine appserver and switch to another associated one. An outdated certificate is defined per client via a Byzantine tolerance threshold (τ): the maximum number of unconfirmed operations the client can tolerate, until a new certificate arrives. On the other hand, if the certificate is valid but demonstrates a prior inconsistent state observed by clients (which occurs due to a Byzantine appserver or due to concurrency in AP systems), the latter can make different decisions, e.g., notify users or roll-back. This is out of the scope of our work.

3.4 Normal operation phase

At the beginning, a client gets assigned an associated appserver s. It establishes a session with s by sending its desired tolerance threshold τ. The appserver stores τ and broadcasts it with the client unique ID to all other appservers. When the client c_i invokes a new operation o, it sends a $\mathsf{REQUEST}(\langle \mathsf{lastReq} \rangle^{\alpha}_{c_i}, s)$ message to s; where $\mathsf{lastReq}$ contains the last sequence number of client's requests, the client identifier c_i and the operation o. $\langle\rangle^{\alpha}_{c_i}$ is the encrypted security token (e.g. digital signature and hash digest) signed with the private key α of c_i. Upon receipt of a valid client REQUEST, s processes the new operation, assigns it a new version vector VV, updates its vector clock, and sends $\mathsf{RESPONSE}(\langle \mathsf{LastRes[n]}, \sigma' \rangle^{\alpha}_{s_j}, c_i)$ to the client; where $\mathsf{LastRes[n]}$ is the corresponding result with a sequence number n and σ' is the appserver's last *certificate*. When the client receives RESPONSE from the associated appserver, it checks its validity (i.e., authenticity, integrity, and sequence nb). Otherwise, as shown in the Figures 3.b1 and 3.b2, if the client associated a slow appserver or received a number of invalid messages, it *complains* and switches to another appserver (see the **Complaining** mechanism in Section 3.6).

In the case of an update operation, the appserver signs and broadcasts the client's REQUEST, via the RCB, to the other appservers. Once receiving REQUEST, other appservers process it and update their vectors clocks. The execution of a new operations triggers the certification phase (see next Section 3.5).

3.5 Certification phase

This phase aims at generating consistent certificate versions across appservers. It occurs in parallel with the Normal phase to avoid any delays in the critical path of clients' requests. As sketched in Figure 2, for every executed REQUEST, an appserver sends a VERTOBFT$(m, \langle\sigma\rangle^{\alpha}_{s_j}, B))$ message to the BFT cluster. The BFT cluster, waits for specific TimeOut epochs trying to collect VERTOBFT messages, then computes a common *stable last version* (LV) as described below. Note that this delay is acceptable being not in the critical path of client's requests. If the LV is equal to a previous LV, the BFT cluster ignores it and repeats the process by waiting for new VERTOBFT messages. Otherwise, the BFT cluster sends a BFTREQSTABLE$(\langle \text{LV}\rangle^{\alpha}_{B}, S)$ to all the appservers asking them for a corresponding state to the LV. Since a convergent state requires all the logs of appservers in an AP systems, the presence of a single Byzantine appserver can affect liveness—but not correctness. However, liveness only affects the more conservative clients, i.e., whose τ is very small due to freshness requirements. These clients may be blocked waiting for a new certificate, while the other clients operate normally.

Once an appserver receives a request BFTREQSTABLE from the BFT cluster, it may already have the exact state version stored. This occurs when LV matched a previous operation VV it executed. If not, the appserver generates such a state using the *Log extraction* mechanism, described below. Afterwards, the appserver sends the generated state in a STABLE$(\langle\sigma\rangle^{\alpha}_{s_j}, B))$ message to the BFT cluster, asking for a new certificate that represents a new stable system snapshot. When the BFT cluster receives at least $f+1$ valid matching states from the appservers via messages of type STABLE, within a defined TimeOut, it generates a new certificate that contains signed hash digests of the appservers (having matching STABLE), and multicasts it as a CERTIFICATE$(\langle\sigma', \text{STABLE}\rangle^{\alpha}_{B}, CorrectS)$ message to these correct appservers only. Late correct appservers can still ask for this certificate version within a predefined timeout. This has no impact on correctness as the late appserver can receive a newer certificate that covers the current operations.

Finally, once a new certificate σ' is received, an appserver checks its validity, updates its old certificate σ with a new one σ' if valid, and starts including it in the future replies to clients.

3.6 ASPAS mechanisms

Stable versions. This mechanism is used by the BFT cluster proxy process to agree on a specific stable version for which a new certificate will be generated. This is required in AP systems since appservers run at different speeds. As operations are applied in different (partial) orders on different replicas, it is necessary to compute a common *stable last version* (LV) according to the received VERTOBFT from at least $2f+1$ appservers. LV guarantees that all operations in the causal past of this version are executed. In particular, the BFT cluster proxy process tries to generate LV in a periodic fashion. The technique is fairly simple: it tries to match the recent version vectors (of operations) received from appservers and then calculates the minVV by computing the minimum of every index apart (in

vector clock, every index corresponds to an appserver's index [18]), a condition that the new computed minVV should be greater than the last old one. Since the minVV is less than or equal all VVs, it means that all corresponding appservers are in a future state of minVV. Therefore, they will be able to generate a corresponding state. We have constrained the new calculated minVV (or what we called LV) to be strictly greater than the last previous one to avoid attacks from Byzantine appserver. The latter may try to send old versions (i.e., small VV) to prohibit generating new certificates. In addition, very high versions sent by Byzantine appservers will be ignored by the minVV function.

Log extraction. Appservers need a way to extract a STABLE state corresponding to the last stable version LV in BFTREQSTABLE, requested by the BFT server proxy. This extraction is needed as an appserver may have never passed through this exact state, although it is included in the final one (a normal behavior in commutative data types). To this end, we used Pure CRDTs [2] that retain a partial ordered log (POLog) of operations. This simplifies the generation of a materialized state using the operations of the POLog, a process we call *Cloning*. The idea is to "clone" the state of the current data type by simply extracting all operations in a POLog having timestamps t' such that $t' \leq t$ (i.e., causally related as per the Lamport's happens-before relation [19]). Any re-execution of this POLog extract will result with the same state since concurrent operations commute. Therefore, appservers can re-execute these operations with $VV \leq LV$ to result a STABLE state that must be equal on all correct appservers. Due to lack of space, we refer the reader to Pure CRDTs [2] for more information if desired.

Complaining. Complaining is the mechanism through which a client can "complain" about a potential Byzantine appserver behavior. A client can complain in two cases. The first case is upon receiving a message that holds an invalid or outdated certificate, as shown in Figure 3.c. In this case the client sends a COMPLAIN$(\langle \mathsf{N}_{\mathsf{Complain}}, \mathsf{PROOF}\rangle^{\alpha}_{c_i}, S{-}s)$ message to all appservers excluding its associated one. The COMPLAIN includes the complain's sequence number $\mathsf{N}_{\mathsf{Complain}}$ and a PROOF $:= (\langle\langle \tau, \mathsf{seqNb}_i, \sigma, \sigma' \rangle^{\alpha}_{s} \rangle^{\alpha}_{c_i})$ of complain correctness. The PROOF contains the agreed client's configurable tolerance threshold τ signed by the associated appserver, the appserver reply's sequence number seqNb_i, the old and new certificate digests σ and σ'. A correct appserver can verify the correctness of the COMPLAIN by checking if $\mathsf{seqNb}_i > \tau$ and $\sigma = \sigma'$. This proof verification is necessary to avoid Byzantine clients from using fake complaints as a DOS attack to blacklist correct appservers. Furthermore, it is used as an evidence to detect/block Byzantine clients with high threat rate. As a result of verification, the correct appservers can detect and blacklist a Byzantine appserver, and reply to the client with a BLACKLIST$(in = \langle bList, \sigma \rangle^{\alpha}_{s_j})$ message. When a client receives $f+1$ matching and valid BLACKLIST messages as a response to its COMPLAIN request, it updates its blacklist accordingly and switches to a new appserver. The client also rollbacks its state to the last correct certificate (if necessary) and resumes sending REQUEST to the new associated appserver in the normal

case. The second case of COMPLAIN occurs when the client collects $f+1$ mismatching replies INSPECTREPLY$(in = \langle\sigma\rangle^{\alpha}_{s_r})$ from different $f+1$ correct appservers according to its INSPECTION. To avoid fake COMPLAIN requested by a Byzantine client, the latter should piggyback the INSPECTION results as a proof of truth PROOF $:= (\langle\langle rndState, \sigma\rangle^{\alpha}_{s_r}\rangle^{\alpha}_{c_i})$.

Inspection. Inspection is a mechanism used by clients in a periodic fashion to make sure an appserver is not using two different logs in the backend and frontend.In ASPAS, to support various security levels for different clients, each client has to define its "tolerance threshold" τ during which it can operate until a new certificate arrives. This requires the client to hold a fairly small log of operations (smaller than τ) for which it has the ability to rollback. While a certificate is always generated according to causal order of operations (delivered via the RCB), any client with $\tau > 1$ will only receive certificates in intervals. However, certificate versions are stable versions that may not correspond to operations on all appservers (as shown in **Stable versions** mechanism in Section 3.6). Consequently, there is a need to make sure that received states between two consecutive certificates are correct. Indeed, a Byzantine appserver may assign the same VV to multiple clients' requests, while only pushes one of them to the BFT cluster. To fool the clients, the appserver can send them wrong replies joint with a correct certificate that holds the only pushed one to the BFT cluster. Inspection starts by having each client to periodically select a random non-certified reply from the appserver (rndState) in its log. Then, it sends an INSPECTION$(\langle \text{rndState}\rangle^{\alpha}_{c_i}, S)$ request to all the appservers, asking for a confirmation of correctness. The client should collect at least $f+1$ matching INSPECTREPLY$(in = \langle\sigma\rangle^{\alpha}_{s_r})$ from the appservers for its requested INSPECTION; otherwise, it will assume its associated appserver as Byzantine. In this case, it sends a COMPLAIN to the other appservers. This random INSPECTION stands as an accountable approach to prevent the misbehavior of appservers. For that, if the number of concurrent clients on the service is high, a Byzantine appserver will be blacklisted even if the rate of inspection is low. Notice that inspection should be used according to a well configured policy (defined time, number of times...) to prevent Byzantine clients from using it as a DOS attack to overload the network. Furthermore, inspection is not crucial to maintain system convergence, it is just an additional mechanism to guarantee as much as possible the correctness of client's received replies. Clients tending for high security over latency can minimize τ to one, or use a classical BFT system.

4 Evaluation

4.1 Implementation

Code. We implemented ASPAS as a Proof of Concept of 5K Java LOC. We opted for Java to have a smooth integration with the BFT-SMaRt library [1], used as the BFT Cluster. The implementation included a thin client, modular proxies (as described in Figure 1), and appserver including a basic Reliable

Broadcast protocol [3]. Being modular, ASPAS can use any classical BFT protocol, like [29, 8, 10], as a backend as long as it exposes the same BFT-SMaRt API. This is required to integrate with the BFT proxies of ASPAS.
DataTypes. We experimented ASPAS using the Counter and Set datatypes following the Pure CRDT model [2]. These two datatypes are widely used in many AP applications. As a single example on social network applications, counters can implement the number of actions: likes, dislikes, comments, views, Ads, etc.; and sets can implement collections of recommended videos, comments, posts, etc. In addition, experimenting counters and sets helps us diversify the payload size of requests and replies (although this had little impact on our results). We plan to opensource the code after refactoring.

4.2 Experimental Settings

Testbed. We prepared an experimental environment that is very close to reality using Emulab [28]. Emulab is a good choice because it allows using real machines for processes, and interestingly allows emulating the network delays using intermediary physical machines which gives the same experience as in geo-replicated link delays and router's queuing. We avoided using Grid5000 since it allows sharing the machines for several experiments which induces ambiguity to the results and makes it harder to reproduce and compare against.
Machines. We used up to 100 physical commodity real machines at Emulab each having two 8-core CPUs with RAM between 8GB and 16GB, five Ethernet 10Gb NICs, and running 64-bit Ubuntu OS. These properties allowed us to run up to 50 client processes or 10 appserver processes on the same machine. The BFT cluster was deployed on four machines to run BFT-SMaRt bftservers.
Networks. One third of these machines were used by Emulab as intermediary delay machines to mimic a real delay in geo-replicated settings. To mitigate the interference across layers and overloading network interfaces (as done in real settings), we used a different network for each layer: frontend, BFT cluster, appservers, and backend (as shown in Figure 1). Otherwise explicitly stated, we configured the network round-trip delays considering the estimated geo-replicated intra- and inter-continental average delays [12]: 70ms for the frontend network (clients and appservers), 70ms to 110ms as variant average delay for the appservers network and backend network (appservers and bftservers), and 10ms for the BFT cluster (since it is not necessary to be geo-replicated). We could have explicitly used different delays within the same network to mimic a real geo-replicated case, but we did not see a significant impact on the evaluation.

4.3 Evaluation Methodology

Metrics and benchmarks. The evaluation focuses on measuring the latency of ASPAS being the most significant Key Performance Indicator in AP systems. In addition, we also measure the throughput considering: the number of simultaneous clients with different payloads (without batching), and execution times. We assume $b = 1$ faulty bftservers out of $3b + 1$ in the BFT cluster unless otherwise

stated. We experimented the Counter and Set datatypes via microbenchmarks and YCSB [11] read/write workloads. In all cases, clients issue a sufficiently big number (up to 10K) of requests to stabilize the network queues. We did not aim to saturate the network being unrealistic in geo-replicated settings. We remove the outliers during the system warm-up and cool-down phases to measure the steady case. We ran each experiment at least five times. Latencies measured at each client are used to compute the average latency. Throughput is measured at the appservers and summed up.

Compared protocols. We compared ASPAS with OBFT [9] as a state of the art BFT protocol for AP systems. To understand the overhead of Byzantine tolerance, we compared ASPAS to two baseline configurations representing the ASPAS extremes: (1) an AP system without any BFT backend or cryptography overhead, and (2) a system running a BFT protocol, e.g., BFT-SMaRt in our case. Even if sometimes deemed to provide "eventual" consistency, we do not compare with recent, classical BFT protocols [29, 26, 6]. However, these protocols are complementary to ASPAS because they can be used as backend instead of BFT-SMaRt if they can improve the BFT cluster functionality.

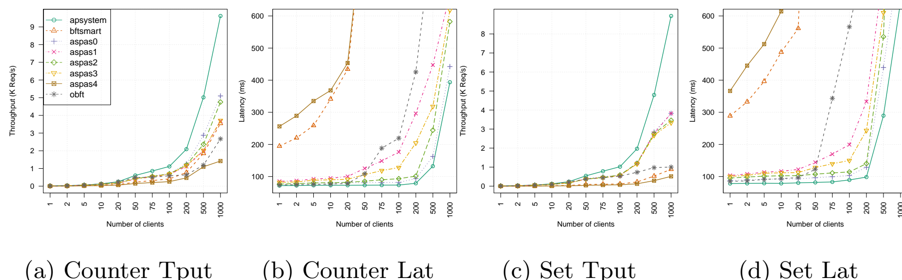
Compared ASPAS configurations. In addition, to understand the performance-security tradeoffs of ASPAS, we considered five ASPAS configurations summarized in Table 1. ASPAS1, ASPAS2, and ASPAS3 are the most interesting configurations as they represent reasonable proportions of different levels of security by clients. We argue that the percentage of $\tau = 1$ is not very low considering AP systems (which is not the common paradigm for conservative applications). ASPAS0 and ASPAS4 are used for the sake of the experiments and in special cases (e.g., under a problem in the backend or system under attack, respectively); therefore, these are not sought to be normal-case ASPAS configurations.

Configuration	% $\tau = 1$	% $\tau = 1000$	% $\tau = \infty$
ASPAS0	0%	0%	100%
ASPAS1	5%	50%	45%
ASPAS2	3%	30%	67%
ASPAS3	0%	100%	0%
ASPAS4	100%	0%	0%

Table 1: The five ASPAS interesting configurations.

4.4 Latency and Throughput microbenchmarks

Counter Figures 4b and 4a show that the latency (resp., throughput) of ASPAS, as the number of clients increase, are close to that of the optimal baseline AP System (apsystem) in most configurations (for $f = 3$). The latency of both ASPAS and apsystem range between 72ms and 400ms as the number of clients reaches 1000. This is consistent with a round-trip request/reply with link delay 70ms with few clients, and expected for 1000 clients since we ran up to 50 client processes on each machine. The same pattern also noticed for the throughput that exceeds 5K req/s with 1000 clients. The figure shows that the apsystem



(a) Counter Tput (b) Counter Lat (c) Set Tput (d) Set Lat

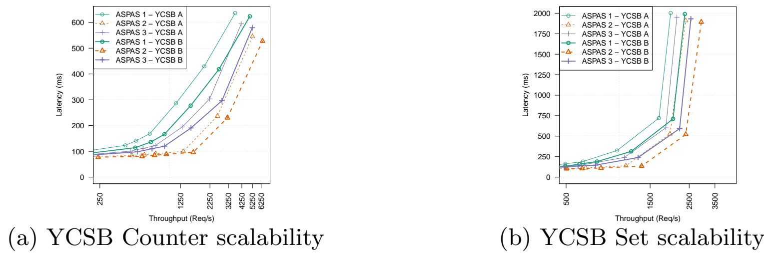
Fig. 4: Throughput and latency microbenchmarks for Counter and Set datatypes with payloads 128B and 4KB, respectively with fixed f=3 and b=1.

can scale a bit more than ASPAS, but we could not add more clients due to the limited number of machines. This result is expected as appservers do extra work by communicating with the BFT cluster, even if in the background.

Considering OBFT [9], as state of the art SEC-based protocol, its latency is slightly lower than ASPAS with few clients. This is not surprising since OBFT uses less secure HMAC-symmetric keys of 6Bytes length and SHA1 digests. However, OBFT's latency starts to increase significantly with 10 clients and more. Indeed, OBFT uses periodic (every 1000 requests) synchronization between appservers to resolve conflicts, which is, contrary to ASPAS, blocking to the client. For the same reason, the throughput of OBFT is the lowest compared to ASPAS and apsystem. This result is an evidence that using a modular approach like ASPAS to check the Byzantine behaviours in the background is less complex and maintains the desired low latency in AP Systems.

On the more conservative baseline, BFT-SMaRt only scales up to 100 clients with almost double the latency of apsystem and ASPAS0 in geo-replicated setting i.e., 70ms round-trip delay. This is expected due to the extensive two round-trip messaging pattern of the BFT-SMaRt protocol. As expected, the latency of ASPAS4 is higher than BFT-SMaRt as confirmed in the figures since all clients requests in ASPAS4 are only served after a certificate is requested from the BFT cluster (which incurs additional round-trip delay of 70ms). This was consistent with the results with few clients where the latency difference was around 78ms.

Interestingly, the latency and throughput of ASPAS in most configurations are very close to apsystem. As expected, the latency of ASPAS0 is a little higher than apsystem due to the overhead of SHA256 hashing and RSA signatures (used by BFT-SmaRt). The results show that as the number of clients with $\tau = 1$ increases, e.g., in ASPAS1 and ASPAS2, the latency (resp, throughput) increases (resp., decreases). This is referred to the impact of high delays of clients with $\tau = 1$ that follow the strong consistency. We also observed that ASPAS2 outperforms ASPAS3 (where all clients set $\tau = 1000$), due to the high proportion of clients with $\tau = \infty$ (i.e., never block) that dominates the effect of the 3% clients with $\tau = 1$. The result of ASPAS3 is promising as ASPAS scales up to 500 clients with throughput and latency close to the optimal baseline apsystem.



(a) YCSB Counter scalability (b) YCSB Set scalability

Fig. 5: Scalability under YCSB A and B workloads

Set Figures 4d and 4c convey the microbenchmarks on the set datatypes. We noticed the same patterns as in the counter case with lower scalability. This is expected since the set payload is set to 4KBytes which incurs additional Cryptographic overhead (using SHA256). The large payload overhead also dominates other protocol's factors which reflects the observation that all ASPAS configurations and OBFT curves are closer than the counter case. Another observation is that the latency and throughput get worse much faster than the counter case. In fact, the increase in the payload size causes delays in the systems that also delays the certificate at the BFT cluster. Therefore, a fraction of the clients will reach the threshold τ and thus wait some milliseconds for the certificate. The average latency shown on the graphs reflects this certificate delay overhead which we discuss further in the following sections. Finally, we expected the latency of OBFT to be worse than the counter case with more concurrent clients (i.e., leads more conflicts). This is because it uses a sophisticated process to have appservers agree on a correct state and undo incorrect ones [9].

4.5 Scalability with YCSB benchmark

To understand the behaviour of ASPAS with real workloads, we experimented it using the YCSB benchmark [11] considering Read/Write percentages for both Counter and Set cases. We only focused on the most three realistic configurations of ASPAS, i.e., ASPAS1, ASPAS2 and ASPAS3. We omit the comparison with other protocols as the results we got are very consistent with those in the previous section. We have used two widely used YCSB workloads: workload A (YCSB A) as an update-heavy scenario with 50% read and 50% update, and the workload B (YCSB B) as a read heavy scenario with 95% read and 5% update. In both scenarios, the request distribution is set to *zipfian*.

Counter: In Figure 5a, i.e., the counter datatype, the results we obtained are consistent with those in the microbenchmark case: the three ASPAS configurations scale up to 500 clients where the latency stays below 450ms, with throughput 3680 req/s in the worst case (ASPAS1). Again, this is considered acceptable in Internet-based geo-replicated applications. We believe this could be lower in reality as we ran 50 clients on a single machine in this scenario. Even in this case, the latency remains realistic although the throughput of the protocol

no longer improves when the appservers get overloaded and the broadcast cost across appserver gets higher. This is also consistent with another observation that the scalability with YCSB B is around 20% better than YCSB A in all configurations; which is expected since with lower update rate in YCSB B, less load is imposed on the system as no broadcast is needed. On the other hand, we noticed the same pattern as in the microbenchmark results where ASPAS2 outperforms ASPAS1 as it includes more clients with $\tau = \infty$ (favor availability) and less clients are conservative (with $\tau = 1$). This phenomenon is expected since clients with $\tau = \infty$ do not have to wait for a certificate, whereas those with $\tau = 1$ wait on every update. However, ASPAS3's scalability remains lower than ASPAS2 although no clients with $\tau = 1$ exist, but as the clients with $\tau = \infty$ (which have the lowest latency) also disappear in the system, the average latency increases slightly, but not as worse as ASPAS1.

Set: The results for the set datatype are conveyed in Figure 5b. the first salient observation is that the curves are closer to those in the Counter case in all ASPAS configurations and both YCSB A and B. The reason is referred to the domination of the request sizes in the Set (4KB) over other factors in the system. What supports this explanation is the low throughput in the case of the set despite the acceptable latency: although the latency is not significantly higher than the counters case (where the link delay is a main factor), the throughput (between 2000 and 3000 req/s) is significantly lower that of the counter case (6400 req/s).

5 Conclusion

In this paper, we presented ASPAS: a Byzantine resilient AP system that ensures Strong Eventual Convergence despite Byzantine behaviors. ASPAS provides backend Byzantine faults detection off the critical path of clients requests, being a requirement for highly available applications. ASPAS also provides a client-based spectrum of tradeoffs between availability and Byzantine security, which is convenient for deployments with mixed application security requirements. Although ASPAS does not affect the liveness of non Byzantine-sensitive clients, it could temporarily block conservative clients that require high freshness. Unfortunately, this is an intrinsic property of AP systems themselves, even in the crash-recovery model, that could be solved via full replication (i.e., mirroring) at the application server level.

Acknowledgments

This work is co-financed by the National Funds through the Portuguese funding agency, FCT - Fundao para a Cincia e a Tecnologia, within project UIDB/50014/2020; and the "NORTE-06-3559-FSE-000046 - Emprego altamente qualificado nas empresas Contratao de Recursos Humanos Altamente Qualificados (PME ou CoLAB)" financed by the Nortes Regional Operational Programme (NORTE 2020) through the European Social Fund (ESF).

References

1. Alysson Bessani, Joao Sousa, and Eduardo Alchieri: State machine replication for the masses with BFT-SMART. In: In Proceedings of the 44th IEEE/IFIP International Conference on Dependable Systems and Networks. IEEE (2014)
2. Baquero, C., Almeida, P.S., Shoker, A.: Making operation-based crdts operation-based. In: Distributed Applications and Interoperable Systems - International Conference, DAIS 2014. pp. 126–140 (2014)
3. Birman, K., Schiper, A., Stephenson, P.: Lightweight causal and atomic group multicast. ACM Trans. Comput. Syst. **9**(3), 272–314 (Aug 1991). https://doi.org/10.1145/128738.128742, http://doi.acm.org/10.1145/128738.128742
4. Bracha, G.: Asynchronous byzantine agreement protocols. Information and Computation **75**(2), 130–143 (1987)
5. Burckhardt, S., Gotsman, A., Yang, H., Zawirski, M.: Replicated data types: specification, verification, optimality. In: ACM Sigplan Notices. vol. 49, pp. 271–284. ACM (2014)
6. Cachin, C.: Architecture of the hyperledger blockchain fabric. In: Workshop on Distributed Cryptocurrencies and Consensus Ledgers. vol. 310 (2016)
7. Cachin, C., Kursawe, K., Petzold, F., Shoup, V.: Secure and efficient asynchronous broadcast protocols. In: Annual International Cryptology Conference. pp. 524–541. Springer (2001)
8. Castro, M., Liskov, B.: Practical byzantine fault tolerance and proactive recovery. ACM Trans. Comput. Syst. **20**(4), 398–461 (Nov 2002). https://doi.org/10.1145/571637.571640, http://doi.acm.org/10.1145/571637.571640
9. Chai, H., Zhao, W.: Byzantine fault tolerance for services with commutative operations. In: Proceedings of the 2014 IEEE International Conference on Services Computing. pp. 219–226. SCC '14, IEEE Computer Society, Washington, DC, USA (2014). https://doi.org/10.1109/SCC.2014.37, http://dx.doi.org/10.1109/SCC.2014.37
10. Clement, A., Kapritsos, M., Lee, S., Wang, Y., Alvisi, L., Dahlin, M., Riche, T.: Upright cluster services. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. pp. 277–290. ACM (2009)
11. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with ycsb. In: Proceedings of the 1st ACM symposium on Cloud computing. pp. 143–154. ACM (2010)
12. Couto, R.S., Secci, S., Campista, M.E.M., Costa, L.H.M.: Latency versus survivability in geo-distributed data center design. In: 2014 IEEE Global Communications Conference. pp. 1102–1107. IEEE (2014)
13. Friedman, R., Licher, R.: Hardening Cassandra Against Byzantine Failures. In: Aspnes, J., Bessani, A., Felber, P., Leitão, J. (eds.) 21st International Conference on Principles of Distributed Systems (OPODIS 2017). Leibniz International Proceedings in Informatics (LIPIcs), vol. 95, pp. 27:1–27:20. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2018). https://doi.org/10.4230/LIPIcs.OPODIS.2017.27, http://drops.dagstuhl.de/opus/volltexte/2018/8642
14. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. Acm Sigact News **33**(2), 51–59 (2002)

15. Guerraoui, R., Kuznetsov, P., Monti, M., Pavlovic, M., Seredinschi, D.A.: Scalable byzantine reliable broadcast (extended version). arXiv preprint arXiv:1908.01738 (2019)
16. Kotla, R., Alvisi, L., Dahlin, M., Clement, A., Wong, E.: Zyzzyva: Speculative byzantine fault tolerance. ACM Trans. Comput. Syst. **27**(4), 7:1–7:39 (Jan 2010)
17. Kwon, J.: Tendermint: Consensus without mining. Draft v. 0.6, fall **1**(11) (2014)
18. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. pp. 558–565. ACM (1978)
19. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM **21**(7), 558–565 (1978)
20. Lamport, L., Shostak, R., Pease, M.: The byzantine generals problem. ACM Trans. Program. Lang. Syst. **4**(3), 382–401 (Jul 1982). https://doi.org/10.1145/357172.357176
21. Malkhi, D., Merritt, M., Rodeh, O.: Secure reliable multicast protocols in a wan. Distributed Computing **13**(1), 19–28 (2000)
22. Saito, Y., Shapiro, M.: Optimistic replication. ACM Computing Surveys (CSUR) **37**(1), 42–81 (2005)
23. Schneider, F.B.: Replication management using the state-machine approach, distributed systems (1993)
24. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: A comprehensive study of convergent and commutative replicated data types (2011)
25. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In: Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems. pp. 386–400. SSS'11, Springer-Verlag, Berlin, Heidelberg (2011), http://dl.acm.org/citation.cfm?id=2050613.2050642
26. Singh, A., Fonseca, P., Kuznetsov, P., Rodrigues, R., Maniatis, P.: Zeno: Eventually consistent byzantine-fault tolerance. In: Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation. pp. 169–184. NSDI'09, USENIX Association, Berkeley, CA, USA (2009), http://dl.acm.org/citation.cfm?id=1558977.1558989
27. Vogels, W.: Eventually consistent: Building reliable distributed systems at a worldwide scale demands trade-offs? between consistency and availability. Queue **6**(6), 14–19 (2008)
28. White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., Joglekar, A.: An integrated experimental environment for distributed systems and networks. ACM SIGOPS Operating Systems Review **36**(SI), 255–270 (2002)
29. Yin, M., Malkhi, D., Reiter, M.K., Gueta, G.G., Abraham, I.: Hotstuff: Bft consensus with linearity and responsiveness. In: Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing. pp. 347–356 (2019)