Towards bounded wait-free PASIS (extended abstract)

Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson,
Michael K. Reiter, Jay J. Wylie

I. INTRODUCTION

The PASIS read/write protocol implements a Byzantine fault-tolerant erasure-coded atomic register. The prototype PASIS storage system implementation provides excellent best-case performance. Writes require two round trips and contention- and failure-free reads require one. Unfortunately, even though writes and reads are wait-free in PASIS, Byzantine components can induce correct clients to perform an unbounded amount of work. This unbounded amount of work can take one of two forms: an unbounded number of protocol steps to complete a read or arbitrarily-sized timestamps. Correct clients may have to read back in logical time to complete a read that is concurrent to a write; Byzantine components can induce a correct client to perform this protocol step an unbounded number of times. A correct client advances logical time by a single unit with every write. A Byzantine component can advance logical time by an arbitrary amount, thus creating an arbitrarily-sized timestamp. Such timestamps require every read and write to perform an unbounded amount of work.

In this extended abstract, we enumerate the avenues by which Byzantine servers and clients can induce correct clients to perform an unbounded amount of work in PASIS. We sketch extensions to the PASIS protocol [1] and Lazy Verification [2] that bound the amount of work Byzantine components can induce correct clients to perform. We present the extensions necessary to constrain Byzantine servers separately from those necessary to constrain Byzantine clients. We believe that the extensions provide *bounded wait-free* [3] reads and writes. As with (unbounded wait-free) PASIS, we assume an unbounded amount of server storage space. We also believe that an implementation that incorporates these extensions will preserve the excellent best-case performance of the original PASIS prototype.

Michael Abd-El-Malek, Gregory R. Ganger, and Michael K. Reiter are with Carnegie Mellon University.

Garth R. Goodson is with Network Appliance Inc.

Jay J. Wylie is with Hewlett-Packard Labs.

II. PASIS PROTOCOL REVIEW

This section briefly reviews the PASIS protocol, though we refer the reader to the original PASIS [1] and Lazy Verification [2] papers for more details. We assume that clients may be Byzantine; that no more than b servers are Byzantine; that authenticated channels exist between all servers and between each client and every server; and, that servers have unbounded storage space. No assumptions are made about timing (i.e., asynchronous timing model). Writes and reads are linearizable [4] and wait-free [5].

At a high level, the protocol proceeds as follows. Logical timestamps are used to totally order all writes and to identify requests that pertain to the same write across the set of servers. For each write, a logical timestamp is constructed by the client that is guaranteed to be unique and greater than that of the *latest complete candidate* (the most recent completely written object). This is accomplished by a READ-TIMESTAMP operation that queries servers for the greatest timestamp they host. Fragments are generated by erasure-coding objects. To complete a write, a client writes timestamp-fragment pairs to a quorum of servers.

To perform a read, a client performs a READ-LATEST operation that issues requests to a set of servers. Once each member of a quorum of servers replies with the latest timestamp-fragment pair that it hosts, the client classifies responses. Classification of responses is based on the number of servers that return timestamp-fragment pairs with a common timestamp. Classification is performed on responses that share the highest timestamp. We refer to the set of responses that share that timestamp as the

candidate. A candidate is classified *complete*, *repairable*, or *incomplete*. If a candidate is classified complete, the object is decoded from the erasure-coded fragments and returned. If a candidate is classified repairable, the object is decoded, erasure-coded fragments are written to additional servers, and once a quorum of servers host the candidate, it is returned. If a candidate is classified incomplete, the candidate is discarded, a READ-PREVIOUS operation is performed that requests responses with the next highest timestamp from servers, and classification begins anew.

For simplicity, consider the simplest threshold quorum that tolerates b Byzantine servers: $N = 4b + 1$ servers with quorums of size $q = 3b + 1$. In such a system, the repairable threshold is $r = b+1$; candidates with fewer than r responses are classified incomplete; candidates with r or more responses, but fewer than q, are classified repairable; and, candidates with q or more response are classified complete. The classification rules and quorum intersection properties ensure that, if a client classifies a candidate as complete, then any subsequent client that classifies the candidate will classify it as repairable or complete, thus ensuring linearizability.

Self-verifying timestamps protect the integrity of stored objects. Logical timestamps have the following structure: $\langle LogicalTime, ClientID, cross\_checksum\rangle$. Correct clients increment $LogicalTime$ by one with every write. The client ID, $ClientID$, ensures that concurrent writes from different clients have different timestamps. The *cross checksum* [6] is an array of N cryptographic hashes, one for each fragment/server. The cross checksum protects the fragments against corruption by Byzantine servers: clients validate fragments returned by servers against the cross checksum in the timestamp. Embedding the cross checksum in the timestamp **and** having clients verify the timestamp at read-time protects against *poisonous writes*. A poisonous write occurs when a Byzantine client writes inconsistent erasure-coded fragments. Because clients verify timestamps at read-time, a client detects the poisonous write and re-classifies the corresponding candidate as incomplete.

In PASIS [1], Byzantine clients could introduce an unbounded number of poisonous writes. With Lazy Verification [2], PASIS servers verify the integrity of stored objects during idle time or whenever a threshold of the number of unverified locally-stored objects is reached. Lazy Verification bounds the number of poisonous writes that can exist in the system at any time; it does so with nominal impact on best-case performance.

III. Byzantine servers

In response to a READ-LATEST, READ-PREVIOUS, or READ-TIMESTAMP request, a Byzantine server can fabricate timestamp-fragment pairs with arbitrarily-high timestamps. In PASIS, such responses can lead to a correct client performing an unbounded number of READ-PREVIOUS operations to complete a read or storing an arbitrarily-sized timestamp.

A. Incomplete candidates

If a Byzantine server responds to a READ-LATEST request with a fabricated timestamp-fragment pair that has an arbitrarily-high timestamp, a client has to perform a READ-PREVIOUS operation based on the fabricated timestamp. The client may contact the Byzantine server(s) again, and they could fabricate another arbitrarily-high timestamp, just slightly lower than before. A client would then perform another READ-PREVIOUS operation, and so on.

The problem is that the correct client uses the highest timestamp returned to a READ-LATEST or READ-PREVIOUS operation. The client ought to use the $b + 1^{\text{st}}$ timestamp in the ordered list of timestamps returned from a quorum of servers: e.g., if timestamps $\{3, 3, \mathbf{2}, 2, 1, 1, 1\}$ are returned and $b = 2$, then timestamp **2**, the "3^{rd} highest" timestamp, is used. We use "$b + 1^{\text{st}}$ highest" timestamp to mean "the $b+1^{\text{st}}$ timestamp in the ordered list of timestamps returned from a quorum of servers" in the remainder of this extended abstract. The $b + 1^{\text{st}}$ highest timestamp is guaranteed to be no greater than a timestamp from a correct server (i.e., it may be fabricated, but it is not arbitrarily high). It is also guaranteed to be no less than the timestamp of the latest complete candidate.

To bound the number of READ-PREVIOUS operations a correct client must perform, the client passes the $b + 1^{\text{st}}$ highest timestamp into the READ-PREVIOUS operation. In response to a READ-PREVIOUS request with timestamp TS, a server returns the timestamp-fragment pair it hosts with a timestamp less than or equal to TS. This is necessary so that the next iteration of the read

algorithm performs classification on the candidate with the $b + 1^{\text{st}}$ highest timestamp. (In PASIS, the server returned the timestamp-fragment pair with a timestamp strictly less than TS.) Basing READ-PREVIOUS on the $b+1^{\text{st}}$ highest timestamp preserves all necessary properties for linearizability: a correct client will not read behind the latest complete candidate. It also bounds the number of READ-PREVIOUS operations a correct client must perform to complete a read.

B. Arbitrarily-sized timestamps

A client constructs the timestamp for a write based on the highest timestamp returned to a READ-TIMESTAMP operation. As with the READ-PREVIOUS operation, a correct client ought to use the $b + 1^{\text{st}}$ highest timestamp returned to READ-TIMESTAMP to protect against responses from Byzantine servers. Correct clients base timestamps on one that is no greater than one returned by a correct server and no less than that of the latest complete candidate. This preserves best-case performance and linearizability, and bounds the amount of work a Byzantine server can induce a correct client to perform.

IV. Byzantine clients

A Byzantine client can perform incomplete or repairable writes to any server, correct and Byzantine alike. A Byzantine client can construct an arbitrarily-sized timestamp for its writes. Lazy Verification bounds the number of poisonous writes a Byzantine client can perform via server verification[1]. Server verification must be extended to bound the amount of work Byzantine clients can induce correct clients to perform via incomplete or repairable writes and arbitrarily-sized timestamps.

A. Incomplete or repairable writes

A correct client does not perform multiple concurrent writes (to the same object). Therefore, only a Byzantine client performs multiple incomplete or repairable writes. Servers must either verify this property or, in the course of trying to verify this property, bound the number of incomplete candidates a client read needs to process.

There are two avenues by which Byzantine clients can induce correct clients to process an unbounded number of incomplete candidates to complete a read: (i) a Byzantine client performs multiple writes to an incomplete threshold of servers, and (ii) a Byzantine client performs multiple writes to a repairable threshold of servers.

Consider case (i): multiple writes to an incomplete threshold of servers. Verification by the servers that receive the incomplete writes determine that the client is Byzantine. Verification by the servers that do not receive the writes may or may not determine that the client is Byzantine; detection depends on the quorum of servers that respond. It is sufficient though, for those servers that receive the writes to determine that the client is Byzantine to bound the number of incomplete writes.

When a server determines that a client is Byzantine, it stops accepting write requests from the client. This bounds the number of incompletes the server hosts from a given Byzantine client. Note that the server continues to accept repairs of candidates written by the Byzantine client from other clients though. Indeed, the server may even repair a write it knows to be from a Byzantine client during verification.

Consider case (ii): multiple writes to a repairable threshold of servers. A server may or may not determine that the client is Byzantine; it depends on the quorum of servers that respond during verification. If the server does not determine that the client is Byzantine, then it must have classified a candidate as repairable and performed repair. In doing so, the server bounds the number of incomplete candidates that a client may have to process to complete a read.

B. Arbitrarily-sized timestamps

To prevent a Byzantine client from writing arbitrarily-sized timestamps, a server must be able to verify that a timestamp is well-constructed. If the server hosts a candidate that the client could have based its timestamp on, timestamp verification can be done locally. A server verifies that the timestamp for a write request is no more than one greater than the latest timestamp it hosts. This local check is sufficient in the common case.

[1] A server performs verification whenever a threshold on the number of unverified versions it hosts for an object is reached. Server verification consists of performing a read to determine the latest complete candidate and to identify poisonous candidates. The threshold dictates the trade-off between how often a server performs verification and how high the bound is on the number of candidates a correct client must process to complete a read.

Unfortunately, due to concurrency or failures, a server may not host the candidate upon which a client bases its write timestamp. To ensure that a server can verify such a timestamp, it is necessary to again modify READ-TIMESTAMP and for the server to perform timestamp-verification. To ensure that its write timestamp can be verified, a client must base the write timestamp upon a candidate it observes at a complete threshold of servers. In concurrency- and failure-free executions, the timestamp of the latest complete candidate is returned to READ-TIMESTAMP and so this comes at no cost. In the face of concurency or failures, a client performing a READ-TIMESTAMP may have to READ-PREVIOUS or perform repair. In so doing, the client ensures that the timestamp upon which it bases its write timestamp exists at a complete threshold of servers and is greater than or equal to that of the latest complete candidate. (Note that a correct client can base its write timestamp on the timestamp of a poisonous write that exists at a complete threshold of servers.)

Timestamp-verification is effectively a READ-TIMESTAMP operation that does not repair candidates. Since a correct client bases its write timestamp on one it observes at a complete threshold of servers, a server will find a repairable or complete candidate with a timestamp no less than one smaller than the write timestamp (because of quorum intersection properties). If no such timestamp is found, the server determines that the client is Byzantine and does not accept the write request.

This extension preserves the best-case performance of the PASIS protocol. It introduces additional situations in which concurrency and failures lead to a bounded amount of additional work by clients or servers.

V. Discussion

The motiviation for bounded wait-free PASIS comes from the recent work on non-skipping timestamps by Bazzi and Ding [7]. Most recently, Cachin and Tessaro incorporated non-skipping timestamps into an erasure-coded Byzantine storage system [8]. We believe that the extensions for bounded wait-free PASIS that protect against arbitrarily-sized timestamps implement non-skipping timestamps. We are not aware of a prior non-skipping timestamp construction that eschews digital signatures.

Given Chockler's recent impossibility result regarding constrained storage space and wait-free storage [9], our assumption of unbounded server storage space is necessary for bounded wait-free PASIS. With bounded server storage space, PASIS was obstruction-free [10]. We believe that the extensions we have sketched preserve the obstruction-freedom guarantee if server storage space is bounded (*constrained* in Chockler's terminology).

We refer the reader to our other papers for a more thorough treatment of relevant work [1], [2], [11].

Acknowledgment

We thank Christian Cachin, Felix C. Freiling, and Jaap-Henk Hoepman for organizing the Dagstuhl Seminar "From Security to Dependability". We thank the CyLab Corporate Partners for their support and participation. This work is supported in part by Army Research Office grant number DAAD19-02-1-0389, by NSF grant number CNS-0326453, and by Air Force Research Laboratory grant number FA8750-04-01-0238. We thank the members and companies of the PDL Consortium (including APC, EMC, Equallogic, Hewlett-Packard, Hitachi, IBM, Intel, Microsoft, Network Appliance, Oracle, Panasas, Seagate, and Sun) for their interest, insights, feedback, and support.

References

[1] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter, "Efficient Byzantine-tolerant erasure-coded storage," in *International Conference on Dependable Systems and Networks*, 2004.

[2] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, "Lazy verification in fault-tolerant distributed storage systems," in *Symposium on Reliable Distributed Systems*, 2005.

[3] M. Herlihy, "Impossibility results for asynchronous PRAM," in *Symposium on Parallel Algorithms and Architecture*, 1991, pp. 327–336.

[4] M. P. Herlihy and J. M. Wing, "Linearizability: a correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, pp. 463–492, 1990.

[5] M. Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages*, vol. 13, no. 1, pp. 124–149, 1991.

[6] L. Gong, "Securely replicating authentication services," in *International Conference on Distributed Computing Systems*. IEEE Computer Society Press, 1989, pp. 85–91.

[7] R. A. Bazzi and Y. Ding, "Non-skipping timestamps for Byzantine data storage systems," in *International Symposium on Distributed Computing*, 2004, pp. 405–419.

[8] C. Cachin and S. Tessaro, "Optimal resilience for erasure-coded Byzantine distributed storage," in *International Conference on Dependable Systems and Networks*, 2006, pp. 115–124.

[9] G. Chockler, "On the space requirements of robust storage implementations," in *Dagstuhl Seminar From Security to Dependability*, 2006.

[10] M. Herlihy, V. Luchangco, and M. Moir, "Obstruction-free synchronization: double-ended queues as an example," in *International Conference on Distributed Computing Systems*. IEEE, 2003, pp. 522–529.

[11] J. J. Wylie, "A read/write protocol family for versatile storage infrastructures," Carnegie Mellon University, Tech. Rep. CMU–PDL–05–108, 2005.