PBFT and Secret-Sharing in Storage Settings

Andreas Happe
AIT Austrian Institute of Technology
18 PUBLICATIONS **256** CITATIONS
SEE PROFILE

Stephan Krenn
AIT Austrian Institute of Technology
38 PUBLICATIONS **319** CITATIONS
SEE PROFILE

Thomas Lorünser
AIT Austrian Institute of Technology
51 PUBLICATIONS **629** CITATIONS
SEE PROFILE

Malicious Clients in Distributed Secret Sharing Based Storage Networks[⋆]

Andreas Happe, Stephan Krenn, and Thomas Lorünser

AIT Austrian Institute of Technology GmbH, Vienna, Austria
{firstname.lastname}@ait.ac.at

Abstract. Multi-cloud storage is a viable alternative to traditional storage solutions. Recent approaches realize safe and secure solutions by combining secret-sharing with Byzantine fault-tolerant distribution schemes into safe and secure storage systems protecting a user against arbitrarily misbehaving storage servers.
In the case of cross-company projects with many involved clients it further becomes vital to also protect the storage system and honest users from malicious clients that are trying to cause inconsistencies in the system. So far, this problem has not been considered in the literature.
In this paper, we detail the problems arising from a combination of secret sharing with Byzantine fault-tolerance in the presence of malicious clients, and provide first steps towards a practically feasible solution.

Keywords: distributed systems ◇ secret sharing ◇ malicious clients ◇ Byzantine fault-tolerance

1 Introduction

Cloud data storage often is a scalable and cost-efficient alternative to in-house storage servers, in particular for user groups that traditionally lack professional dedicated IT support such as consumers and SMEs. But besides obvious advantages, outsourcing storage into the cloud also poses various security risks that do not exist in offline or in-house solutions, in particular concerning the confidentiality and integrity of the stored data.

For this reason, initial solutions encrypted data locally before transferring it to a single cloud provider. For better availability and to prevent vendor lock-in, advanced solutions distribute data between multiple clouds, e.g., by storing multiple replicas. However, encryption might not always be an appropriate solution — in particular when storing highly sensitive data such as electronic health records. This is because encryption can only hide the data computationally, and it is virtually impossible to reliably estimate how cryptanalysis and an adversary's computational power will develop in the far future (just think of DES which was assumed to offer sufficient security 40 years ago, and which can easily

⋆ This work was in part funded by the European Commission under grant agreement number 644962 (PRISMACLOUD).

be broken nowadays). Modern cloud storage systems based on secret sharing scheme thus also offer the option to distribute the information in an redundant yet information theoretically secure way, e.g., [8].

In parallel to this development of cloud storage and due to the increased availability of high-bandwidth Internet connections, also the usage changed: from pure archiving systems to data sharing and collaborative development systems. Therefore, protocols handling concurrent user requests have been developed, e.g., to avoid that two users writing different versions of the same file at the same time cause an inconsistency in the system. As a result, Byzantine fault tolerant (BFT) algorithms that can cope with arbitrarily malicious storage servers have been presented.

In this work we now consider the next evolutionary step of cloud storage: multi-client secret sharing based storage systems that allow for secure collaboration *even in the presence of malicious users*. This becomes necessary because of the increasing number of participants working on joint projects — e.g, in case of cross-company projects — where it must be assumed that an adversary will eventually gain access to some user's login credentials, and that he will then try to boycott the project by causing an inconsistent state in the system.

In this document, we first describe selected complications that must be addressed by such protocols in Sec. 3 and then sketch mitigation strategies in Sec. 4.

1.1 Related Work

Various secret sharing based storage systems have been proposed and implemented (cf. [15]). We will briefly summarize the most relevant ones and discuss their shortcomings when dealing with multiple potentially malicious clients.

Multiple cloud storage solutions [14,16] implement a proxy pattern: clients communicate with a central server that in turn splits up data and distributes it over multiple backend cloud storage providers. This creates a single point of trust and failure within the proxy and thus fails our availability and privacy needs.

RACS [1] utilizes erasure-coding to distribute data upon multiple cloud storage providers. It's use-case is prevention of vendor lock-in. Security and privacy is of no concern, i.e., data is not encrypted. Parallel access to stored data through multiple RACS instances is achieved through usage of Apache Zookeeper, which implements the Zab primary-backup protocol for synchronization. This allows for parallel client access but does neither protect the data's security nor can it cope with malicious Zookeeper nodes.

DEPSKY-CA [3] is a Byzantine fault-tolerant storage system. In its system model, servers do not communicate with each other, clients must synchronize access to files through a low-contention lock mechanism that uses cloud-backed lock files for synchronization. While being obstruction-free this mechanism is not safe in face of malicious clients. Similarly, Belisarius [9] integrates robust versions of secret sharing with BFT protocols, but explicitly forbids malicious clients.

Summarizing, existing proposals and solutions are dealing with information dispersal mechanisms for remote data storage, they are using secret sharing to protect confidentiality and some of them deal with Byzantine robustness. To the best of our knowledge, however, none of them supports full concurrency for a multi-user environment while allowing for malicious clients.

2 Preliminaries

In the following we briefly recap the necessary background on secret sharing and Byzantine fault tolerance.

2.1 Secret Sharing and Information Dispersal

In threshold secret secret-sharing schemes, a *dealer* splits up data d into n shares, such that at least $k \leq n$ of these shares are needed to reconstruct the original data. Conversely, an attacker with access to less than k cannot gain any information about d. Basic secret sharing schemes assume that at reconstruction time, shares are either correct or missing. *Robust* secret sharing schemes are able to detect and cope with maliciously altered shares. Conversely, *verifiable* secret sharing schemes can be used to detect dealers who are creating and distributing inconsistent shares, by enabling storage nodes to (jointly) verify that they received consistent shares.

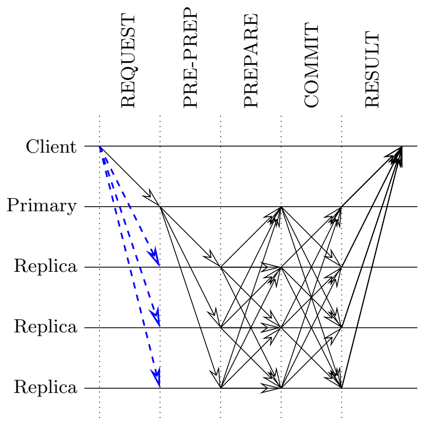
As an extension to basic secret sharing schemes, *verifiable secret sharing* (VSS) offers protection against malicious dealers. That is, in a VSS scheme the share holders can efficiently verify that they received consistent shares without violating the confidentiality of the shared message, cf., e.g., [11].

2.2 PBFT Standard Implementation

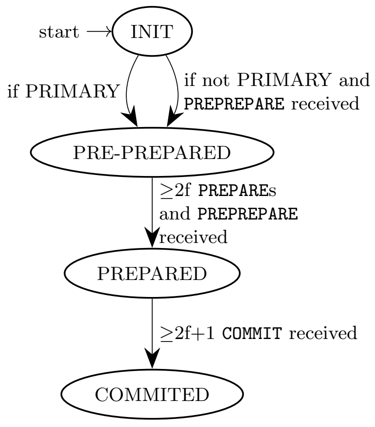
This section gives an overview of the PBFT protocol [4], which is the most prevalent BFT protocol for storage solutions. For clarity of presentation, we present the protocol in its vanilla form, and omit any possible performance optimization.

PBFT assumes a system consisting of n nodes (or *replicas*), which are connected through authenticated and private channels. At most $f = \lfloor \frac{n-1}{3} \rfloor$ nodes may be faulty. Exactly one replica is designated as primary. The system moves through a sequence of *views*, where the primary replica might be changed with every view-change in order to cope with malicious primaries. The view-concept introduces the concept of bounded synchronicity into an otherwise asynchronous system.

In the following we describe the normal mode of operation of PBFT. For error handling mechanisms we refer to the original literature [4].



(a) PBFT/SS



start
INIT
if PRIMARY
if not PRIMARY and PREPREPARE received
PRE-PREPARED
≥2f PREPAREs and PREPREPARE received
PREPARED
≥2f+1 COMMIT received
COMMITED

(b) Node

Fig. 1: Black arrows describe the message flow of the vanilla PBFT protocol. The dashed blue arrows show the extension discussed in §3.

Normal operation and transaction state-machine. In PBFT a client sends the operation to the designated primary node. The primary associates an unique counter and broadcasts the operation and the counter to all other replicas as `PREPREPARE` message. Upon receiving this message all replicas broadcast a `PREPARE` message including a hash of the operation and its sequence number. If more than $2f+1$ prepare messages (including one's own) are received by a replica, it broadcasts a `COMMIT` message. After $2f+1$ matching `COMMIT` messages have been collected by a replica and all transactions with lower sequence numbers have been performed, the requested operation is executed and the result sent to the original client. After $f+1$ matching results the client knows the operation's result. The archetypal message flow diagram as well as state diagram describing a single operation/transaction can be seen in Figure 1.

3 Integration Issues with Secret Sharing

Integrating secret sharing into the PBFT model is far from trivial: the PBFT protocols sketched above rely on the fact that all replicas contain the same data, which is not the case for a secret sharing based storage where each node receives a distinct share which does not allow for reconstructing the original data.

Consequently, the core PBFT protocol must be altered to allow for per-replica shares. This leads to various issues, including:

Matching client-requests: In the original PBFT protocol, the client transmits operation requests to the primary which in turn distributes them to all other replicas. As the operation requests might also include data, this is not applicable in the new setting, as no replica is supposed to learn the original data.

A natural modification of the original scheme thus seems to be to split up the operation into secret-shared data and plaintext (such as type of operation, etc.). The client would now transfer the same metadata and distinct shares directly to each single replica and let the replicas match this data to the primary's `PREPREPARE` message; the possible protocol flow is depicted in Figure 1. The main challenge in this approach now is how this matching of can be done reliably, in particular if the client sent duplicate, incomplete, or inconsistent requests to the system.

Clients providing inconsistent shares: A malicious client might send inconsistent shares to the replicas within a single valid operation request. Now, in the original PBFT protocol, a digest over the initial client message — containing the operation (comprising the data), a timestamp, and the an client identifier — is utilized as identifier and checked for consistency.

For PBFT/SS, this approach could only be applied to the metadata, but not to the data itself, as digests of different shares will not match, and thus consistency cannot be assured in this way.

It is thus required to efficiently detect inconsistent shares within a request that is valid from a PBFT point of view.

Clients submitting "consistent garbage": A malicious client can, while confirming to the PBFT protocol, provide consistent shares that contain destructive payloads, e.g., by overwriting sensitive files with consistent shares of random bits. This is a generic problem and needs to be addressed in any meaningful multi-client system.

Besides these client-related issues, integrating secret sharing into PBFT also causes numerous further problems, including:

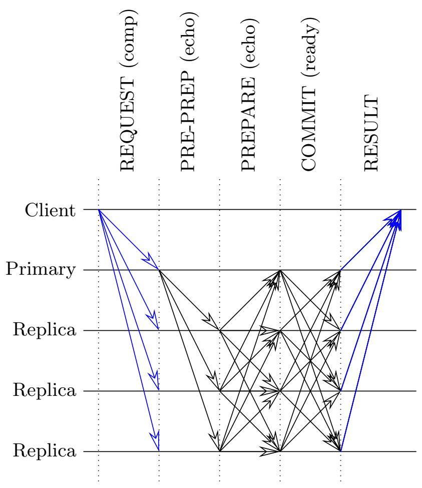
Verifying operation results: In non-secret-shared PBFT clients can verify an operation's result by comparing the replica's return values. Similar to before, because of different replica starting with different input, an operation cannot be verified by simply checking return values for equality.

Checkpointing protocol: Replicas periodically perform protocols where a check sum over the replicas' data is utilized to detect anomalies. Secret-sharing prevents simple hashing schemes as replicas do not contain the same data.

State-transfer, recovery and pro-activity: Those are important aspects of long-running systems. Protocols needed for those functions again have to be augmented to be able to cope with secret-shared data.

4 Our Approach

We now address the mentioned issues regarding malicious clients in byzantine secret-shared storage networks. We always detail the simplest solution, further performance improvements are mentioned if applicable.



(a) PBFT/VSS

AsyncVSS		BFT
Phase	Message	Message
Sharing	send	REQUEST
Sharing	echo	PRE-PREPARE
		PREPARE
Sharing	ready	COMMIT
Reconstruct	-	after COMMIT phase, before operation execution

(b) Mapping *AsyncVSS* onto BFT

Fig. 2: Integration of VSS with the modified BFT protocol.

4.1 Matching client-requests

Implicit matching of client requests over their shared meta-data (e.g. type of operation, file names, timestamp) can be performed relatively easy as all needed meta-data is part of the plain-text message portion, i.e. is not secret-shared. Clients submit their operations over authentic and integrity-protected channels: if malformed or non-matching meta-data is detected, a malicious client is indicated.

The client-provided meta-data is utilized as simple identifier, i.e. can easily be generated by using a digest over each request's metadata. To improve performance, clients can implement an explicit id scheme, i.e. unique client-assigned identifiers (e.g. random number) per matching requests. This identifiers are then used to match the requests. If a malicious client submits non-matching identifiers or reuses them, the resulting transactions never complete and are detected by either a timeout mechanism or during the eventual execution of the view-change protocol.

As this leads to increased processing and memory utilization node-side, rate-limits might be required to counter denial of service attacks. A simple solution—e.g. rate limiting each client through an exponential backoff algorithm based upon the per-client mismatch detection rate—should be sufficient for the initial system prototype.

4.2 Clients providing inconsistent shares

To detect inconsistent shares sent to the replicas, a verifiable secret sharing scheme (VSS) can be used. This allows the replicas to detect the inconsistency, but imposes computational and messaging overhead. However, to reduce the messaging overhead non-interactive VSS schemes [11] could be integrated into the PBFT protocol in order to simulate an asynchronous VSS [10]. Initial work on integration of the *AsyncVSS* protocol[2] has shown that VSS-protocols can be piggy-backed upon our existing BFT/SS protocol. *AsyncVSS* is an asynchronous computational-secure protocol requiring a minimal replica count of $>= 3f + 1$ thus providing a good match to our similar BFT requirements. Figure 2a shows the resulting protocol, Table 2b matches *AsyncVSS* phases and messages to BFT messages used in our storage protocol. No new BFT messages were required as all *AsyncVSS* messages were integrated into our BFT/SS protocol. Another benefit of this approach is, that the VSS information—i.e. that a client provided matching shares—is generated exactly before the replicas would execute and commit the corresponding operation. Replicas are thus able to detect malicious data before they would execute the corresponding operation.

Further efficiency improvements can be derived from the realization that a single operation usually comprises many secret shared blocks.[1] Batch techniques for VSS can be used to minimize the computational and messaging overhead [6].

Alternatively, servers could employ versioning to perform "speculative" acceptance: operations produce new versions while the old data version is kept until a client certificates that the data was received correctly. This approach is well known in the BFT world, e.g. the 'Zyzzyva" family of BFT algorithms[5] improves the performance of PBFT algorithms by utilizing speculative operation execution. Other byzantine systems allow for unconditional command execute and depend upon subsequent (probabilistic) error detection[12]. In both situations, an error detection mechanism must be embedded in order to allow for rollback actions.

However, versioning impacts security mechanisms, and therefore needs to be handled with care. The following section details issues that arise from utilizing speculative execution and versioning.

4.3 Clients submitting "consistent garbage"

An issue inherent to every storage solution are malicious clients that submit formally correct but semantically destructive operations. Typical examples are malicious users purposefully deleting compromising documents or the recent wave of encryption-based ransomware programs[13]. The mentioned versioning approach can be reused to mitigate the risk associated with evil clients. If malicious alterations are detected, a non-altered prior version of the document can be restored.

As only non-malicious clients can attest that an operation was semantically correct, there's a trade-off between storage and availability of older data versions. Versioning adds a "speculative" phase to stored documents' life cycle. This phase starts after each data alteration operation and ends after a quorum of clients did

[1] Secret sharing is usually applied on a several-byte level, while typical file sizes are in the range of MBs.

perform read operations upon the same data without any subsequently reported integrity violations.

Also the selection of trustworthy clients needs to be solved on a per-deployment base, examples includes selection through policy or achieving a quorum of consistent clients.

Versioning allows for performance and efficiency improvements as additional meta-data is implicitly available at the storage-server. This metadata (i.e. Change or Commit information) allows detection of causal dependencies between operations and might be utilized to introduce weaker—but more efficient—consistency models such as causal consistency (instead of the currently used sequential consistency model[7]).

5 Future Work

Metadata privacy is a challenging question which could be explored in further research. The main issue arises from trade-offs between metadata privacy, performance, and the possibility of multi-user access, especially in face of malicious users. Fully private metadata management places all metadata tasks at the client and thus severely limits exploitation of parallelization – preventing performance benefits – at the server level. Increased client-side state automatically increases opportunities for client-side attacks and thus poses new challenges in a fully Byzantine setting.

References

1. Abu-Libdeh, H., Princehouse, L., Weatherspoon, H.: Racs: a case for cloud storage diversity. In: 1st ACM Symposium on Cloud Computing. pp. 229–240. ACM (2010)
2. Backes, M., Kate, A., Patra, A.: Computational verifiable secret sharing revisited. In: Advances in Cryptology–ASIACRYPT 2011, pp. 590–609. Springer (2011)
3. Bessani, A., Correia, M., Quaresma, B., André, F., Sousa, P.: Depsky: dependable and secure storage in a cloud-of-clouds. ACM Transactions on Storage (TOS) 9(4), 12 (2013)
4. Castro, M., Liskov, B.: Practical Byzantine Fault Tolerance and Proactive Recovery. ACM Trans. Comput. Syst. 20(4), 398–461 (Nov 2002)
5. Kotla, R., Alvisi, L., Dahlin, M., Clement, A., Wong, E.: Zyzzyva: speculative byzantine fault tolerance. In: ACM SIGOPS Operating Systems Review. vol. 41, pp. 45–58. ACM (2007)
6. Krenn, S., Lorünser, T., Striecks, C.: Batch Verifiable Secret Sharing (2016), in preparation
7. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. Computers, IEEE Transactions on 100(9), 690–691 (1979)
8. Lorünser, T., Happe, A., Slamanig, D.: ARCHISTAR: Towards Secure and Robust Cloud Based Data Sharing. In: IEEE (ed.) CloudCom 2015 (2015), in press
9. Padilha, R., Pedone, F.: Belisarius: BFT Storage with Confidentiality. In: International Symposium on Network Computing and Applications. pp. 9–16. IEEE (2011)

10. Patra, A., Choudhury, A., Pandu Rangan, C.: Efficient Asynchronous Verifiable Secret Sharing and Multiparty Computation. Journal of Cryptology 28(1), 49–109 (Dec 2013)
11. Pedersen, T.P.: Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing. In: Feigenbaum, J. (ed.) Advances in Cryptology - CRYPTO 1991. LNCS, vol. 576, pp. 129–140. Springer (1991)
12. Popescu, B.C., Crispo, B., Tanenbaum, A.S., Bakker, A.: Design and implementation of a secure wide-area object middleware. Computer Networks 51(10), 2484–2513 (2007)
13. Savage, K., Coogan, P., Lau, H.: The evolution of ransomware (2015)
14. Selimi, M., Freitag, F.: Tahoe-lafs distributed storage service in community network clouds. In: BdCloud 2014. pp. 17–24. IEEE (2014)
15. Slamanig, D., Hanser, C.: On Cloud Storage and the Cloud of Clouds Approach. In: ICITST-2012. pp. 649–655. IEEE Press (2012)
16. Spillner, J., Bombach, G., Matthischke, S., Muller, J., Tzschichholz, R., Schill, A.: Information dispersion over redundant arrays of optimal cloud storage for desktop users. In: UCC 2011. pp. 1–8. IEEE (2011)