Report Group 1

Improvements from 1st Stage

After receiving the teacher's feedback for the first delivery, we understood that it lacked freshness in the exchanged messages and atomicity when servers were storing reports in their database, since they were writing to different tables and a possible crash could leave the database with an incomplete report. These problems were overcome by adding a Nonce field to every message and by using transactions when performing writing in the database, respectively.

System Model

This iteration of the project considers a model similar to the one used in stage 1. The present model, however, acknowledges multiple servers, some of which can be byzantine. These byzantine servers introduce complexity in the current problem since that it is impossible to ensure that we are communicating with a correct process and receiving correct responses. The present solution tolerates scenarios where a byzantine server sends an arbitrary value as a response for a read operation and another case where a byzantine server acknowledges a write when in reality it was discarded. The existence of more servers allows byzantine clients to write different values in different servers for the same operation, compromising the system consistency. This model also aims to maintain the availability of the system by preventing Denial of Service attacks.

Byzantine Servers

We used the Byzantine Regular Register abstraction to tolerate byzantine servers. Since each register is identified by the pair `(userId, epoch)` and each user sends a unique report per epoch, we know that each register is only written once and has an implicit timestamp of 0 if it was not written to or 1 otherwise. This implies that each read operation has one of three outcomes: 1) The register is empty; 2) The register has a valid report; 3) The register has an invalid report. We also know that anyone can validate a report by verifying if it has enough valid proofs and that if every process writes and reads from a byzantine quorum (`(N + fs)/2` servers, where `fs` is the maximum number of byzantine servers), it's granted that at least one correct process will participate in any pair of operations. We can then conclude that any read operation will get a value from at least one correct process that contains the most up to date value for that register and that the reader will be able to recognize which value is the right one.

This solution, however, allowed a byzantine user to store different valid reports on different servers by sending different sets of valid proofs within each report, breaking the safety property of the register abstraction, even though being harmless to our application. Nonetheless, we decided to use the Double Echo

Broadcast algorithm when sending writes to servers to overcome this issue: the Consistency and Totality properties of this algorithm assure the same value will be written in every correct server. Since clients that initiate the broadcast just need to know if the message was delivered by a correct process, they do not participate in the entire broadcast. The client knows the broadcast is completed after receiving a quorum of confirmation messages. The course book foresaw this adaptation for client-server architectures and confirmed its correctness.

Our system required report operations (`submitLocationReport` and `obtainLocationReport`) to have atomic semantics. This was ensured by adding a writeback phase before delivering the read value, as specified in the course book. This guarantees the ordering property of the atomic registers, i.e. if a read returns a value `v` and a subsequent read returns value `w`, then the write of `w` does not precede the write of `v`. Since the writeback is an expensive operation, these semantics were only assured in these operations.

Spam and DoS

In order to prevent Spam and Denial of Service attacks, we implemented a Proof of Work (PoW) routine for the system's message exchange. This mechanism is used when a client issues a request which demands intensive work from a server. Upon receiving such request, a server returns a challenge to the client before executing any work. This challenge consists of finding a string which, when appended to the client's request, produces a hash value that has a prefix of zeros with length specified by the server. Besides this string, a unique value generated by the server must also be appended to prevent the reuse of challenge responses for the same requests. The bigger the length of the prefix, the longer it takes for the client to find the string. This prevents the spam of requests, which could affect the server's performance and lead to Denials of Service. It's infeasible for the client to circumvent this work, as for that to happen it would need to be able to reverse hash functions. Although it takes time to compute the challenge's response, the server can trivially verify it.

We decided our expensive, i.e. needing PoW, operation was `submitLocationReport` because it consists of two types of computationally heavy tasks: writes to the database within transactions and cryptographic verification of signatures.

Conclusion

This project shows the cost of tolerating byzantine processes: The higher number of exchanged messages and the required extra computation to verify every operation create a significant overhead in the servers. Unfortunately there are no cheaper solutions to solve this problems in the present model, so tolerating byzantine faults must be carefully considered in real world applications. During the development of this project we also learned that even though an algorithm is correct, it is difficult to make sure that its implementation is flawless and ensures every property of the algorithm.

Afonso Gonçalves 89399
Daniel Seara 89427
Marcelo Santos 89496