An Open Source Implementation for Self-stabilising Byzantine-Tolerant Recycling

Self-Stabilizing Byzantine-tolerant Consensus

Tobias Hjalmarsson, tobhja@chalmers.se

Suggested Supervisor at CSE (if you have one, otherwise skip this row): Elad Schiller

Relevant completed courses Tobias Hjalmarsson:

- *EDA387, Computer Networks*
- *TDA596, Distributed Systems*
- *TDA297, Distributed Systems, advanced course*

December 3, 2023

1 Introduction

Numerous distributed applications, such as cloud computing and distributed ledgers, necessitate the system to invoke asynchronous consensus objects an unbounded number of times, where the completion of one consensus instance is followed by the invocation of another. With only a constant number of objects available, object reuse becomes vital. Georgiou, Raynal, and Schiller (1), GRS from now on, addressed the challenge of object recycling in the presence of malicious (Byzantine) processes, which can deviate from the algorithm code in any manner. GMR is also self-stabilizing, which is a powerful notion of fault tolerance. Self-stabilizing systems can recover automatically after the occurrence of arbitrary transient faults, in addition to tolerating communication and (Byzantine or crash) process failures, provided the algorithm code remains intact. We provide a recycling mechanism for asynchronous objects that enables their reuse once their task has ended, and all non-faulty processes have retrieved the decided values. This project aims to provide the first implementation of GMR using Go-Lang.

2 Problem

The studied problem considers a building block that is needed for the implementation of asynchronous consensus objects. With only a bounded number of consensus objects available, it becomes essential to reuse them robustly. We examine the case in which the repeated invocation of consensus needs to reuse the same memory space and the $(k+1)$-th invocation can only start after the completion of the k-th instance. In an asynchronous system that uses only a bounded number of objects, ensuring the termination of the k-th instance before invoking the $(k+1)$-th might be crucial, e.g., for total order broadcasting, as in some blockchains. Thus, we require SSBFT consensus objects to eventually terminate regardless of their starting state. We propose to implement GMR since it addresses the challenge of recycling asynchronous consensus objects after they have completed their task and delivered their decision to all non-faulty processes.

3 Context

In (1) GRS investigate object recycling in relation to processes which can succumb to byzantine faults. They managed to present a solution that has an expected stabilisation complexity linear to the number of byzantine processes. It also appears to be the first solution for self-stabilizing byzantine fault-tolerant for synchronous multivalued consensus. In (2) previous work on loosely-self-stabilizing binary consensus algorithms that was done in (3) is extended. Some of the new contributions are that it uses bounded memory and he algorithms presented in (2) is also to the best of my knowledge the first solution to the binary consensus that is loosely-self-stabilizing and also byzantine fault-tolerant, they also present a solution for self-stabilizing byzantine fault-tolerant multivalued consensus, these solutions are applicable to different system models as are presented in (2), these solutions also relies on the object recycling presented in (1).

As the algorithms proposed in the papers earlier discussed are likely the first of their kind implementing a pilot for GRS in practice will to the best of my knowledge be the first of its kind. It will also likely come with significant challenges when it comes to implementing and verifying the work in GRS for a practical system compared to a theoretical one.

4 Goals and Challenges

The goal of this project is to provide a pilot implementation for GRS in Go-Lang. This implementation is supposed to validate the correctness proof provided by GRS. Such an implementation can facilitate a prototype implementation later on (perhaps in this project, if there is time).

The solution by GRS includes a number of challenging building blocks that this project aims to deliver. It also relies on a number of building blocks, such as the consensus algorithm in (2), which we hope to also implement in this project. Another building block that is required for the implementation of GRS is a service for random common coins. In our pilot implementation, we will merely use a local random generation function with a common seed at each process. Later on, if we attempt to provide a prototype implementation, we can substitute this assumption with a random common coin server.

5 Approach

The functionality of the proposed solution appears in Section 2. We plan to evaluate the pilot over the following testbed, `https://supr.naiss.se/`. Since we plan to couple between the GRS solution and the consensus algorithm in (2), our evaluation criteria are borrowed from the ones of consensus algorithms, such as latency and throughput as well as number of messages (and bits), memory,

usage, and CPU usage. The intention is to deliver an open-source implementation of the studied algorithm.

6 References

References

[1] C. Georgiou, M. Raynal, and E. M. Schiller, "Self-stabilizing byzantine-tolerant recycling," in *SSS*, ser. Lecture Notes in Computer Science, vol. 14310. Springer, 2023, pp. 518–535.

[2] C. Georgiou, I. Marcoullis, M. Raynal, and E. M. Schiller, "Loosely-self-stabilizing byzantine-tolerant binary consensus for signature-free message-passing systems," in *NETYS*, ser. Lecture Notes in Computer Science, vol. 12754. Springer, 2021, pp. 36–53.

[3] A. Mostéfaoui, H. Moumen, and M. Raynal, "Signature-free asynchronous binary byzantine consensus with t ¡ n/3, o(n2) messages, and o(1) expected time," *J. ACM*, vol. 62, no. 4, sep 2015. [Online]. Available: https://doi.org/10.1145/2785953

Reference all sources that are cited in your proposal using, e.g. the APA, Harvard2, or IEEE3 style.