CS5232: Formal Specification and Design Techniques (AY22/23 Semester 2)

Project Proposal

February 2023

Students:
Achmad Yogi Prakoso (A0250692A)
Ng Geon Woo Robin (A0218422M)

1 Introduction

Chain Replication [2] provides a storage system that supports storing, queries, and updating objects. In managing distributed resources, some tradeoffs might need to be considered. For example, reserving consistency sometimes needs to sacrifice throughput, which later triggers debate among system designers. This design claims to provide strong consistency while keeping high throughput and availability.

Figure 1 shows Chain Replication Protocol. It works like a linked list in which each node connects to at most two nodes: predecessor and successor. One end of the chain is the head, and another is the tail. The head is responsible for receiving update requests, whereas the tail receives queries and replies (including replies for update requests). The middle nodes only forward messages from one node to the other node and cannot receive any requests from clients. Under this scenario, anything written in the tail, the data must be written in any other nodes as well - hence it has a strong consistency.

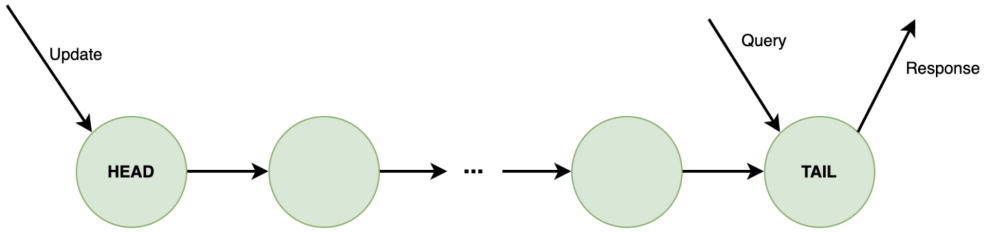

Figure 1: Chain Replication Protocol

In a high level detail, each node in the protocol will keep three attributes: $Hist_{objID}$, $Pend_{objID}$, and $Sent_i$. Any request whose the id is stored in $Hist_{objID}$ has been processed, otherwise it will be queued in $Pend_{objID}$. $Sent_i$ is a collection of sync requests from a node to node i as a bookkeeping

that node i has not acknowledged those requests. To these extents, we know that

$$Hist^{j}_{objID} \preceq Hist^{i}_{obdID}$$

for some i, j such that $i \leq j$, which is later called *Update Propagation Invariant*. We also notice that when a message has been sent out of a node, that does not mean that the message will immediately arrive at the destination node, assuming that they use a best-effort networking protocol. This invariant is then called *In-process Request Invariant* such that

$$Hist^{i}_{objID} = Hist^{j}_{objID} + Sent_i$$

We will use TLA+ as the formalization tool to ensure that this protocol works properly as intended, including the behavior of Update Propagation Invariant and In-process Request Invariant. We will also investigate its behavior towards server failures and ensure the objects' consistency.

2 Explanation

Chain Replication is an algorithm for supporting large-scale storage services that exhibit high throughput and availability without sacrificing strong consistency guarantees. Chain replication has real-world uses in its domain, being used in systems such as Delta, a low-dependency object storage system by Meta. This will be an interesting and non-trivial algorithm to formalise in TLA+ as, unlike transaction commit as covered in lectures where resource managers can function independently of one another, the equivalent resource managers (nodes) in Chain Replication interact directly with its successor and predecessor nodes.

Furthermore, as nodes follow a strict order, there is no longer symmetry in the algorithm in regard to node failures. Concretely, the failure of the head node would lead to vastly different state transitions than a failure in a middle or tail node. The interaction between multiple node failures will also be very interesting to analyse as a result of this increased complexity.

All these factors combine to give rise to complex and non-trivial state transitions that would be impractical to be analysed without tools like TLA+. By formalising the algorithm in TLA+, we could analyse these complex traces to ensure that the invariants mentioned in the paper hold for all traces, helping prove the correctness of the algorithm in the process. We could check liveness conditions to ensure that no combination of node failures would result in inconsistency between the head and tail nodes. We would also gain better knowledge of the utility and limitations of TLA+ through this process.

3 Related Works

The closest comparison is perhaps Primary/Backup protocol. The significant difference regarding their topology is that Primary/Backup protocol uses parallel backup servers where the primary is

responsible for all incoming requests from clients and synchronizes the data to all backups. Chain Replication uses serial servers that traverse the request from one to another. It is then easier to remove a faulty node or add a new node, resulting in lower duration needed for transient outages. However, in Primary/Backup protocol, it requires less time to respond to the clients' requests because the primary is directly connected to the backup - it is proportional to the maximum latency of the working nodes. In Chain Replication, a message must traverse through all the nodes before the client can get its reply. Hence, the response time is proportional to the sum of all the nodes' latency.

Another related work is GFS (Google File System) [1]. It provides a fault-tolerant service using cheap hardware products. However, it does not consider consistency because the nature of their file usability does not require it to be consistent. Once data are written, they rarely modify them. Any incoming data will be just appended to the file without keeping them being serialized in concurrent events. Chain replication, however, while keeping a fault-tolerant service, it also preserves strong consistency.

4 Project Feasibility

We can define correctness based on the safety and liveness of the protocol. The paper demonstrates Chain Replication performance compared with Primary/Backup in terms of throughput, which is a good indicator of liveness. However, the paper does not demonstrate its safety further, so we focus more on its safety and consistency in this project, which the paper does not include. We might face several challenges when building a formal specification for the protocol during the implementation.

First, the fundamental part is that we have to establish the equivalent TLA+ properties and invariance towards the protocol that the paper only explains its high-level overview. We need to apply many state transitions such that we might produce some implicit modifications that give different outputs or behaviors.

Second, in Chain Replication, all servers participate in the message deliverability (serial). Hence, some data might be lost if server failures happen too frequently. If we allow message loss or server failure in TLA+, we need to find some way to limit their occurrence such that the probability of the events is less than half or smaller.

5 Goals

We have identified the following project deliverables we aim to achieve by the end of the course.

Firstly, we would model the base behaviour of message passing between nodes used in chain replication using TLA+ as a simplified specification. This will act as our minimal viable prototype (MVP) where we do not model an unstable network, reordering or dropping messages between nodes, as well as node failures. We can use the MVP to check liveness conditions and the invariants mentioned in the paper.

After an MVP has been produced, we would create an implementation of the chain replication specification in TLA+, adding the behaviour of node failure and recovery. This will be done in stages, modelling failure in the head node, then the tail node and finally the middle nodes. At each stage, we will check if the implementation still follows the specifications mentioned in the MVP. The final implementation, properly modelling all 3 possible failures and recovery actions will act as our main goal and deliverable in this project.

Possible stretch goals were also considered, this includes extending our implementation to model additional events such as adding new nodes to the chain when the chain length gets below a specified threshold and modelling possible network errors during message passing such as reordered or dropped requests.

6 Timeline

Table 1: Timeline

Task	**Week**
Research implementation details of Chain Replication Algorithm	5-6
Formalise States in MVP model	7
Formalises Transition Events in MVP model	7
Create MVP Specification in TLA+	7-8
Debugging and testing MVP	8-9
Formalise additional states for implementation	9
Formalises additional transition events for implementation	9
Create Implementation for MVP Specification	9-10
Work on stretch goals	10-11
Create presentation slides	11-12
Finish and submit the final report	12-13

References

[1] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system, 2003.

[2] Robbert Van Renesse and Fred B Schneider. Chain replication for supporting high throughput and availability., 2004.