Formal Modelling of Byzantine Fault Tolerant Consensus Protocols using PGo

CPSC 448 Final Report

Daine Yip
Supervisor: Ivan Beschastnikh
Term: 2025 Winter Term 1

December 2025

Contents

1 Introduction

Blockchain technologies provide means to eliminate the need for a centralized authority in financial transactions and also provide privacy and anonymity. The global blockchain technology market is valued at over $32 billion and is projected to grow aggressively in the coming years. Blockchain plays an important role in today's world, particularly in decentralized finance (DeFi) systems where trust and correctness are critical.

Byzantine fault tolerant (BFT) system design is a core element of blockchain networks. A distributed system is subject to Byzantine faults where it is possible for a dishonest (or Byzantine) participant to deviate arbitrarily from the protocol, which can disrupt consensus by sending conflicting information within the network. There are n replicas in total, of which at most f are Byzantine ($n = 3f + 1$). The goal being that all correct replicas eventually reach the same decision, despite the presence of Byzantine replicas. This can be done through a leader-follower protocol like PBFT.

However, implementing BFT protocols in standard programming languages is prone to concurrency bugs, race conditions, and deadlocks that are difficult to detect through standard testing. In a concurrent environment where multiple clients submit transactions simultaneously, subtle errors such as threads overwriting shared memory can cause the entire network to misbehave. These flaws would undermine the security guarantees of the algorithm and decrease public trust in the underlying infrastructure, especially for handling financial transactions.

Therefore, bridging the gap between formal theory and executable code using formal verification tools such as TLA+ and PGo is critical for blockchain consensus algorithms. In this report, I made the following contribution: developed a formal specification of PBFT that supports concurrent client requests. The model identifies and resolves critical race conditions found in naive implementations, specifically in addressing shared-memory corruption caused in concurrent client requests. For those reasons, it provides a verified foundation that is structurally prepared for compilation into Go, resulting in a more reliable and secure system implementation.

2 Background

2.1 Practical Byzantine Fault Tolerance (PBFT)

PBFT, introduced by Castro and Liskov in 1999, is a replication algorithm designed to tolerate arbitrary (Byzantine) faults. This protocol maintains correctness in a system with $n = 3f + 1$ replicas, where f is the total number of arbitrarily behaving (faulty) replicas. PBFT is a leader-follower protocol, and is partitioned into the following 3 phases:

1. **Pre-prepare** - The primary replica receives a request and broadcasts a "pre-prepare" message to all other replicas in the network.
2. **Prepare** - Each replica independently verifies the "pre-prepare" message, then multicasts a "prepare" message to all other replicas. The replica then enters the "prepared" state.
3. **Commit** - Once a replica has $2f + 1$ "prepare" messages in its log, it multicasts a "commit" message. Finally, when a replica has accepted $2f + 1$ "commit" messages, it executes the request.

2.2 Formal Verification: TLA+ and PlusCal

TLA+ (Temporal Logic of Actions) is a formal specification language used to model concurrent and distributed systems. It allows engineers to define the precise behaviors of a system and mathematically verify properties such as safety (bad things never happen) and liveness (good things eventually happen) using the TLC model checker.

PlusCal is an algorithm language that transpiles to TLA+, offering a more familiar imperative syntax for programmers while retaining the power of temporal logic verification. PGo is a source-to-source compiler that translates Modular PlusCal (MPCal) specifications into runnable Go programs, aiming to eliminate implementation errors that occur when manually translating verified models into code.

3 Project Objectives

As outlined in the initial proposal, the primary goal of this project was to formally model a BFT consensus algorithm using MPCal and compile it into a runnable Go program.

Specific goals included:

- Developing a formal model of PBFT in MPCal.
- Verifying the model against safety and liveness properties using TLC.
- Documenting the specification decisions and verification results.
- (Stretch Goal) Compiling the model into a Go executable using PGo.

4 Implementation Strategy

The project followed a "complexity-layering" methodology, iteratively building up from basic message passing to a full concurrent BFT state machine. This approach allowed for fundamental aspects of PBFT to be broken down into manageable deliverables, allowing bugs to surface in a low-complexity environment, as well as important design decisions to be revealed earlier in the process. These checkpoints acted as essential building blocks for constructing the full complexity of PBFT.

4.1 Phase 1: Foundations (The Echo Protocol)

To establish a baseline knowledge for message passing and general PlusCal syntax, the project began with an Echo protocol. This phase was critical for understanding how TLA+ models state transitions, how the model checker functioned, and how to enforce correct termination in a distributed system.

Single Message Implementation: The initial model, `echo.tla`, implemented a synchronous single-sender, single-server architecture.

- **State Management:** The model utilized global variables `clientRequest` and `serverResponse` to simulate a shared network. The Client process updated `clientRequest` to simu-

late sending a message, while the Server process awaited `clientRequest # ""` before responding.

- **Verification:** Correctness was defined by the invariant `CorrectnessInvariant`, which asserted that if a response was received, it must match the last sent message (`serverResponse = lastSent`).
- **Termination:** The system utilized a simple boolean flag `clientTerminated` to signal completion, verifying that the system could reach a terminal state without deadlocking.

Multi-Message Expansion: The model was expanded in `echo_multimessage.tla` to handle arbitrary sequences of messages. Handling multiple messages required new data structures and variables which were key to understanding how the system would handle complex PBFT phases in later implementations.

- **Queues and Sequences:** Instead of single-value variables, this iteration introduced explicit FIFO queues (`send_queue` and `received_queue`) modeled as TLA+ sequences. This allowed for the verification of message ordering, ensuring that messages sent earlier were processed before messages sent afterwards.
- **Dynamic Cardinality Checks:** To handle variable message loads, the model incorporated dynamic counters (`messages_left` and `messagesToHandle`) initialized to `Cardinality(messageSet)`. The Server loop continued processing until `messagesToHandle` reached zero, verifying that no messages were dropped or duplicated.
- **Strict Ordering Invariant:** The correctness check was refined to include sequence numbers. The invariant verified that for any given sequence number, the payload in the `latest_messageSent` matched the `latest_serverResponse`, effectively proving that the server preserved both data integrity and transactional ordering.

4.2 Phase 2: Broadcast and Leader Logic

Transitioning from point-to-point communication to the one-to-many broadcasting required for PBFT's multicast phases involved two new servers: an iterative broadcast loop to be later modified into a procedure, as well as an arbitrary leader server where a chosen node would broadcast a message sent from the client to all other replicas.

Basic Broadcast Implementation: In `broadcast.tla`, the model introduced the concept of a "Broadcast Loop" to simulate multicasting in a reliable network.

- **Broadcasting to the Set Difference:** The broadcasting process utilized a local variable `toSend`, initialized to `replicaSet \ {self}`. The process entered a `while toSend # {}` loop, and non-deterministically picked a recipient `r` from the set, constructing a message, appending it to the global `sent_queue`, and removing `r` from `toSend`.
- **Broadcast:** The Client process itself acted as a placeholder for the broadcaster. This allowed us to verify the broadcast logic (`BroadcastLoop`) in isolation before introducing the complexity of inter-replica communication. Once the Broadcast logic was successful, it was refactored into a procedure that would be used for inter-replica communication.

Arbitrary Primary Replica and Architecture Changes for PBFT:
`broadcast_arbLeader.tla` applied the Broadcast procedure to a system more aligned with PBFT infrastructure. Implemented with a single Client, who sends a message to an arbitrary leader, which then uses Broadcast to distribute the message to the remaining replicas.)

- **Client-Leader-Replica Flow:** The Client process was refactored to send a single request to a specific leader node (`to |-> leader`) rather than broadcasting directly.
- **Arbitrary Leader Selection:** The Replica processes introduced a conditional check `if (chosenLeader = {self})` to determine behavior. This logic allows an arbitrary replica in the configuration to act as the Primary if designated, without resulting in any code changes.
- **Replica-Side Broadcast:** Upon receiving a client request, the designated leader replica initiates the `BroadcastLoop` to the remaining backups (`replicaSet \ {self}`). This established the standard "Primary → Backups" communication pattern required for the Pre-Prepare phase implemented next.

4.3 Phase 3: The PBFT State Machine

With the broadcast primitives established, the core consensus logic was implemented in `broadcast_prepare.tla`. This phase introduced the distinct phases of PBFT ("preprepare",

"prepare", "commit") and the state transitions required to move a replica from initialization to responding to the client.

- **Boolean State Flags:** With the addition of multiple phases of PBFT, we modified the previous `replicaTerminated` flag with a Boolean state flag for the completion of each phase (`replicaTermination_preprep`, `replicaTermination_prepare`, `replicaTermination_commit`). This flag system allowed us to know when each replica could progress to the logic associated with different phases in the PBFT process.
- **Linear Phase Transitions:** The process logic was structured as a strictly linear sequence of guarded actions:
  - **Pre-Prepare:** The `WaitForPrePrepare` step blocked until a message with `type = "preprepare"` arrived at the head of the inbox.
  - **Prepare & Commit:** Completion of the previous phase automatically enabled the broadcasting of the next vote type.
- **Critical Finding: Head-of-Line Blocking:** An outcome of this phase was the identification of a concurrency bug caused by strict sequential processing. The verification revealed that if a Commit message arrived before a Prepare message (due to one replica being too "slow"), the replica would deadlock because it was strictly `await`-ing the Prepare type at the head of its queue. This limitation directly motivated the shift to the set-based `received_log` architecture used next.

Before introducing concurrent clients, a critical refactoring phase was conducted in `PBFT_messagelogrefactor.tla` to address the strict sequential phases enforced by `broadcast_prepare.tla`. This refactor was necessary for allowing replicas to ingest any message received in their inbox, regardless of the type, and make state transition decisions based on a `received_log` rather than messages received at the head of their inbox. This used two key mechanisms:

- **Message Pumping:** In `broadcast_prepare.tla`, replicas were designed to block and wait for specific message types in a strict order (e.g., `await type="preprepare"`). This enforced a rigid sequential flow that could stall if messages arrived out of order. The refactored model implements a "message pump" pattern: a generic `MessageHandling` action that consumes any message at the head of the inbox immediately upon arrival

and persists it to a local `received_log`. This ensures the inbox is continuously drained regardless of the message type or current replica state.

- **Log-Based State Calculations:** Instead of transitioning states in direct response to a message arrival, state transition rules were implemented by continuously monitoring the `received_log`. These actions calculate whether the necessary conditions are met, such as checking if a specific number of preprepare messages exist, or if the count of prepare/commit messages satisfies the quorum, which then moves the replica to a new state. This separation allows the replica to process messages asynchronously while maintaining correct state transitions.

4.4 Phase 4: Refactoring and Concurrency (Final Model)

The final iteration, `PBFT_concurrentClient.tla`, represented a significant leap in complexity. While previous phases modeled a single transactional flow, this phase introduced concurrent client handling where multiple requests pass through the consensus phases simultaneously.

- **Concurrency vs. Pipelining:** The model was designed to support inter-client concurrency, allowing Client A and Client B to have requests processed in an interleaved fashion. However, it does not support request pipelining from a single client; each client must wait for their previous request to complete before sending a new one. This design choice simplifies the client logic while still allowing testing of the PBFT system with concurrent client inputs.
- **State Tracking Challenge:** The primary challenge in this phase was maintaining the distinct states of multiple overlapping client requests. Unlike the model used with only one `client_request` ongoing, a replica might simultaneously be Committed for Sequence #1 while Pre-Prepared for Sequence #2. This required a fundamental architectural shift from global state flags to a sequence-indexed state mapping, allowing replicas to track progress for different transactions independently.
- **Memory Isolation Strategy:** Scaling to concurrent processes revealed a critical race condition in earlier models. Previously, a global variable `msg_struct` was used as a temporary buffer for message construction. In a concurrent environment however,

Replica 1 could overwrite this buffer while Replica 2 was reading from it. This was resolved by refactoring all temporary buffers into process-local variables (e.g., `variables ... curr_msg = ...`). This ensures memory isolation between threads.

5 Technical Specifications of the Final Model

The final artifact, `PBFT_concurrentClient.tla`, is a formal specification of the PBFT normal-case operation.

5.1 System Architecture

- **Processes:** The system consists of a set of Replica processes and a set of Client processes.
- **Sequencing and Communication:** The protocol relies on a selected leader replica to act as the ordering authority. Upon receiving a `client_request`, the leader assigns it a unique, monotonically increasing internal sequence number. This sequence number is propagated in the PrePrepare message and serves as the primary key for all subsequent consensus phases.
- **State Machine Mapping:** To manage concurrency, replicas do not hold a single "current state." Instead, they maintain a `req_state` map (e.g., `[seq` $\rightarrow$ `"prepared"]`). This map associates a specific consensus stage (init, prepared, committed, done) with each internal sequence number. This data structure allows the replica to effectively handle the correct phase's logic on transactions when multiple messages are being handled at once. This allows inter-client request concurrency, and also ensures that progress on Sequence #2 does not impact the finalization of Sequence #1.

5.2 Key Technical Resolutions

The formal verification process identified critical implementation flaws that are often missed in standard software testing.

Race Condition Elimination: Earlier iterations utilized global variables (`msg_struct`) for message assembly. This led to data corruption traces where multiple replicas executed

the Broadcast procedure simultaneously, overwriting each other's payloads. The final model enforces strict message structure variables for each process, ensuring thread safety during message construction.

6 Limitations

While the model successfully verifies the core PBFT logic for a minimal configuration, practical limitations in the model checking environment constrained the scale and depth of verification.

- **State Space Explosion & Hardware Constraints:** The introduction of concurrent clients caused an exponential growth in the reachability graph. Verification was successfully completed with a configuration of 2 replicas and 2 clients, which took approximately 7 minutes. However, attempts to scale the model to 3 or 4 replicas resulted in the TLA+ Toolbox crashing due to memory exhaustion and timeout errors. The size of the state space for $N \geq 3$ exceeded the available RAM on the testing machine, preventing a complete traversal of the reachability graph.
- **Quorum Logic vs. "Happy Flow":** Due to the scaling limitations, the current verified model primarily tests the "happy flow" where all replicas return a response. It does not fully exercise the non-blocking quorum logic where the system proceeds after receiving only $2f + 1$ responses. The system currently validates that consensus *can* be reached, but not necessarily that it is reached strictly via a sub-majority quorum in the presence of silence or delay.
- **Fault Tolerance Thresholds:** To mathematically prove Byzantine resilience, PBFT requires a minimum of $N = 4$ replicas ($3f + 1$ where $f = 1$). Because the model could not be run with $N = 4$ due to the hardware constraints mentioned above, the model currently serves as a proof of correctness for the state machine and concurrency logic, but does not strictly verify the Byzantine fault tolerance threshold.
- **Deterministic Fault Injection:** The model lacks a harness for deterministic fault injection. Ideally, verification would involve a scenario where a specific node is forced to behave maliciously (e.g., sending conflicting messages or dropping messages) at a selected phase. Currently, faults are not accounted for.

7 Missing Components

Relative to the initial proposal and the formal definition of PBFT, the following components were not fully implemented:

- **Comprehensive Invariants:** The current specification checks basic safety properties and type checks throughout the PBFT process (e.g., `TypeOK` invariant and `TerminationCorrectness` property). However, it lacks the comprehensive set of invariants defined in the original OSDI99 paper (Castro & Liskov), such as formal checks for *Committed-Local* predicates and strict view-change safety conditions.
- **Refactoring for MPCal (Modular PlusCal):** To facilitate compilation into Go, the current single-module TLA+ specification needs to be refactored into Modular PlusCal (MPCal). This involves separating the client, replica, and network logic into distinct processes that PGo can map to distinct Go structs.
- **View Change Protocol:** The current model focuses on "Normal Case" operation. The complex View Change mechanism, which handles leader failure and election, was not implemented.
- **Use of Crypto:** Cryptographic operations were abstracted. The model assumes authenticated channels and does not model signature forgery or hash collisions.

8 Future Work

- **Advanced Model Checking Techniques:** To overcome the memory exhaustion issues encountered with 3 and 4 replicas, future verification should experiment with different configurations in TLA+ Toolbox's **Simulation Mode**. Simulation Mode explores random execution traces to infinite depth, which allows for finding deep bugs in larger configurations without exhausting memory. Additionally, allocating more cores and increasing the physical allocated memory to the TLC Model Checker could allow for a complete standard verification of the $N = 4$ case.
- **PGo Compilation Pipeline:** After adding the missing components of the final PBFT implementation, the next step is to complete the refactoring of the specification into MPCal. Once modularized, the model can be passed through the PGo compiler to generate a Go prototype.

- **Implementation of Invariants:** Future iterations should formally encode the specific invariants from the PBFT paper into TLA+ operators. This would provide a rigorous mathematical guarantee that the implementation adheres strictly to the theoretical protocol.
- **Targeted Fault Injection:** Future work should introduce a process capable of deterministic fault injection of an arbitrary amount of replicas. This would allow the model checker to verify that the system maintains safety specifically when the f replicas act erratically, verifying that the system effectively handles malicious nodes under the threshold while still maintaining correctness.