Verifying Consensus Protocols from LLM-assisted TLA$^+$: A Case Study of Byzantine Reliable Broadcast

Shuhe Cao
Shandong University
Jinan, China
shcao@mail.sdu.edu.cn

Xin Wang*†
Shandong University
Jinan, China
wangxin87@sdu.edu.cn

Chenxu Wang
Shandong University
Qingdao, China
cxwang.phd@mail.sdu.edu.cn

Xiao Sui
Tsinghua University
Beijing, China
suixiao@mail.tsinghua.edu.cn

Sisi Duan*‡
Tsinghua University
Beijing, China
duansisi@tsinghua.edu.cn

Abstract

TLA$^+$ (Temporal Logic of Actions) is a formal specification language well-suited for distributed systems. However, writing proper TLA$^+$ scripts requires high domain expertise. When it comes to modeling Byzantine behaviors for Byzantine fault-tolerant consensus protocols, the simulation of malicious behavior is a fundamental challenge: overly simplified modeling misses critical vulnerabilities, and verbose modeling leads to state-space explosion.

In this paper, we present `TLAssist`, a large language model (LLM)-assisted tool for semi-automated TLA$^+$ generation tailored for Byzantine reliable broadcast (RBC) protocols. We provide a highly structured workflow and domain-specific data format to improve the quality of LLM prompts. Using six RBC protocols as case studies, we have some interesting findings. First, the specification generated by `TLAssist` outperforms many open-source TLA$^+$ scripts we are aware of, including those written by domain experts. Second, `TLAssist` can assist domain experts in identifying deep design flaws. Notably, our case study on the (2, 3, 4)-Optimistic RBC (a CCS'25 distinguished paper) captures a subtle issue that causes the violation of the *totality* property. Finally, `TLAssist` is useful for non-experts. Namely, we show that by revising the protocols slightly, the generated error traces effectively show complex corner cases that can facilitate understanding of the design.

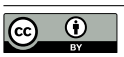
CCS Concepts

• **Security and privacy → Distributed systems security**.

Keywords

Byzantine reliable broadcast, temporal logic of actions, large language model

*Corresponding authors.
†Xin is also with State Key Laboratory of Cryptography and Digital Economy Security, Shandong University, and Shandong Key Laboratory of Artificial Intelligence Security.
‡Sisi is also with State Key Laboratory of Cryptography and Digital Economy Security, Zhongguancun Laboratory, and Shandong Institute of Blockchains.



This work is licensed under a Creative Commons Attribution 4.0 International License.

1 Introduction

Fault-tolerant consensus is one of the most fundamental problems in distributed computing, with applications in cloud computing [1, 20], blockchains [9, 24], and secure multiparty computation [10, 11]. Consensus protocols are notoriously difficult to get right [22, 40]. Many protocols without thorough analysis or peer reviews have been shown to be wrong [22, 51, 59, 60]. In fact, even well-known protocols such as PBFT [26], Zyzzyva [43], and Raft [64] have been found to suffer from subtle issues and might fail to achieve the security goals [5, 12, 63]. This is exacerbated by the fact that, in the implementation of consensus protocols, engineers might modify the codebase for performance optimization but end up implementing an unproven, failure-prone protocol [40].

TLA$^+$ (Temporal Logic of Actions) is a formal specification language that is particularly suitable for distributed systems [45, 46]. In TLA$^+$, a system is specified as a state machine, and mathematical expressions are used to model the global state in a precise manner. The model checking tool of TLA$^+$ is called TLC, which iterates all the state transitions to identify design errors. TLA$^+$ has been widely used to verify distributed systems, especially cloud computing platforms [40, 62, 65, 66, 78, 86] or distributed databases [34] that often adopt crash fault-tolerant protocols such as Zab [42] and Raft [64]. Some research papers also claim that their protocols are *strengthened* via model checking from TLA$^+$ [38, 72].

Despite the maturity of TLA$^+$ and its tools, TLA$^+$ is not easy to use, mainly because the expressions are not easy to understand [44]. When it comes to the simulation of Byzantine behaviors (i.e., arbitrary failures), TLA$^+$ often needs to simplify the workflow of the protocol, as otherwise the number of states to be iterated will explode. Thus, writing TLA$^+$ that can run efficiently and meanwhile identify issues requires high expertise. Many research papers' only contribution is writing TLA$^+$ specification of some protocols [19, 37, 86]. Thus, it is not surprising that verifying the correctness of TLA$^+$ specification itself might be error-prone, creating a circularity issue.

Recently, the rapid development of large language models (LLMs) has significantly advanced many applications in software engineering tasks. For example, Microsoft's Copilot, a code generation tool powered by Codex, has surpassed 20 million all-time users, and developers using GitHub Copilot complete coding tasks 55% faster than those without the tool [2]. Many recent efforts have been

made to improve the quality of code generation or code completion [29, 53, 80, 92]. Thus, it is interesting to know whether we can use LLM to improve or facilitate the writing of TLA$^+$.

Our research problem. In this work, we pick Byzantine reliable broadcast (abbreviated as RBC in this paper) [7, 16, 23, 33, 58, 72] as a case study and build `TLAssist`, an LLM-assisted tool for TLA$^+$ generation. We choose RBC since it is one of the most fundamental primitives in the study of consensus protocols. Ever since RBC was first proposed in the 80s [16, 17], the problem has been extensively studied until now [7, 23, 33, 54, 58, 72]. Meanwhile, many people have tried model checking for RBC protocols from TLA$^+$ [3, 4, 72, 76, 81], so we have enough examples to compare with.

Can we build an LLM-assisted TLA$^+$ tool that generates TLA$^+$ specification directly from the pseudocode?

The reason why we focus on generating TLA$^+$ specification from pseudocode is threefold. First, as several prior papers have claimed, providing a TLA$^+$ specification for a proposed protocol can strengthen their work [38, 72]. This is possibly because the formal TLA$^+$ specification can better guide developers in implementing protocols that match the pseudocode as closely as possible. Second, as mentioned above, some implementations of the protocols might modify the workflow for performance optimization, but end up implementing an unproven protocol. Instead of asking the protocol designer to prove the protocol again, revising TLA$^+$ scripts and running a quick check can greatly simplify the procedure. Last but not least, before writing formal proofs of the protocols, the protocol designers may use our tool to quickly identify error traces via TLC, the model checking tool of TLA$^+$. In this way, designing a protocol and proving its correctness can be facilitated.

Technical challenges. Building such a tool is not easy. First, the hallucination problem of LLM might make it challenging to generate a correct TLA$^+$ script. In fact, our experience shows that, unlike code generation for which hallucination often comes from issues such as packages or paths [73], the hallucination for generating TLA$^+$ often comes from missing necessary actions for correct state transition in TLA$^+$ and is more difficult to debug. Second, successful TLA$^+$ compilation does not necessarily mean that the TLA$^+$ is correctly written, and some issues might not be caught. Thus, it becomes more difficult to debug an LLM-generated TLA$^+$ specification. Finally, due to the exhaustive iteration of all possible states of TLC, lowering the number of states (which, as mentioned above, is directly related to running time) while still generating correct TLA$^+$ can be challenging.

Overview of our approach and our findings. We build `TLAssist`, an automatic TLA$^+$ generation tool for RBC pseudocode via LLM. Our tool consists of several crucial building blocks designed specifically for TLA$^+$, including a JSON-based structured data format to make LLM understand the pseudocode better, and an error feedback loop to improve the quality of the generated script. Using our tool, we can generate correct TLA$^+$ scripts for most RBC protocols and its related primitives.

As shown in Table 1, we use three protocols to assess `TLAssist` and six protocols as case studies. Our findings are summarized below.

- The quality of scripts generated by our tool can be better than scripts written by domain experts. As a case study (Sec. 6.1), we show that the TLA$^+$ script included in the paper of a recent RBC protocol called (2, 3, 4)-Optimistic RBC [72] has some subtle flaws that fail to properly model Byzantine behavior.
- Our generated TLA$^+$ scripts can be used to find design issues. We show that the (2, 3, 4)-Optimistic RBC protocol [72] suffers from a totality (a liveness property) violation (Sec. 6.1). We have confirmed the issue with the authors.
- We show that our tool can also aid developers and beginners in this area (Sec. 6.2). Namely, we can modify some workflow in the pseudocode, and the generated TLA$^+$ scripts and the model checking results can identify some corner cases that cause security violations. Using three notable examples, SigBRB [7], CT RBC [56], and 2-round-BRB [6] as the case study, we show that one does not have to completely review the proofs again to identify the issues. Such a process can facilitate understanding for beginners and developers.
- We conduct a preliminary generality study on Streamlet [27] and HotStuff [87], two well-known BFT protocols (Sec. 6.5). Our study shows that `TLAssist` can already generate TLA$^+$ for both protocols without syntactic errors. However, to make them more useful, additional works still need to be done.
- Our experience of designing `TLAssist` also reveals some challenges in using TLA$^+$ for model checking of consensus protocols in general, especially for Byzantine consensus protocols. Some challenges are related to the pseudocode (Sec. 6.4) and some are related to TLA$^+$ itself (Sec. 7). We have open-sourced our scripts for reproducibility of the results.

Protocol	TLA$^+$ validated?	Usage
AEB* [21]	[4]	assessing `TLAssist`
ADB† [21]	[4]	assessing `TLAssist`
Bracha's broadcast [16]	[81]	assessing `TLAssist` & case study 4 (Sec. 6.4)
(2, 3, 4)-Optimistic RBC [72]	[72]	case study 1 (Sec. 6.1)
CT RBC (Optimized) [56]	no	case study 2 (Sec. 6.2)
SigBRB [7]	no	case study 2 (Sec. 6.2)
2-round-BRB [6]	no	case study 2 (Sec. 6.2)
ADD-based RBC [33]	no	case study 3 (Sec. 6.3)

Table 1: Protocols studied in this work. *AEB denotes authenticated echo broadcast, which is also known as consistent broadcast [69]. †ADB denotes authenticated double-echo broadcast. The definition is equivalent to RBC and the construction is equivalent to Bracha's broadcast. We consider it mainly because the pseudocode of ADB and Bracha's broadcast are written in different styles.

2 Related Work

Model checking for distributed systems. Model checking for distributed systems, especially those relying on crash fault-tolerant protocols, has been widely studied. Besides verifying the design of the system, model checking is also well known for finding *deep* software bugs that cannot be easily found from software testing tools [49, 66, 78]. Moreover, it can also be used to build an approach to verify whether the implementation of a protocol matches the specification of the design [40].

Most studies so far focus on systems built upon crash fault-tolerant protocols such as Paxos and Raft. In comparison, the verification of Byzantine fault-tolerant protocols is more challenging, mainly because TLA$^+$ needs to *simulate* arbitrary behavior and there is no clear guideline on how to simulate them. Moreover, since TLC iterates all the state transitions to identify errors, and arbitrary behavior is extremely challenging to exhaustively iterate, the number of states might explode, and the script will run out of CPU memory. While some recent efforts have been made on blockchain-related protocols on Byzantine behaviors [13, 19, 37, 50], such studies are still preliminary for Byzantine consensus. We believe our work can be extended to build more powerful approaches for formal verification of Byzantine fault-tolerant protocols.

Code generation by LLMs. Code generation, also known as program synthesis [79], refers to the computational capacity to produce software code based on predefined specifications. With the emergence of large language models (LLMs), these systems can now be trained on vast amounts of code and human-written descriptions, and generate code via natural language prompts.

Recent studies on logical problem-solving indicate that LLMs perform significantly better on complex tasks when prompted to decompose problems into sequential reasoning steps [70, 82, 85]. In this paper, we build on this concept and introduce a decomposition framework specifically designed for TLA$^+$ generation.

LLM-assisted model checking. There is a growing trend of using LLMs to facilitate model checking [15, 25, 77, 91]. A recent study [15] showed that using different prompting strategies, LLMs fail to generate TLA$^+$ with high quality. Namely, out of 205 generated samples, only 8.6% of them can be successfully compiled in TLC. Such a finding is aligned with ours when building `TLAssist`. In fact, in our tasks of using TLA$^+$ to verify the correctness of reliable broadcast, we find that even if a TLA$^+$ script can be successfully compiled in TLC, the script might still fail to identify design issues. We thus further refine the evaluation metrics (Sec. 3) to assess the quality of TLA$^+$. Besides, some efforts have been made to generate TLA$^+$ from the source code [31, 32], and the goal is often to find *deeper* software bugs that cannot be easily identified via conventional software testing tools.

Another line of work is *verifiable code generation* [75, 83, 84]. Namely, during LLM-assisted code generation, the formal specification is generated simultaneously to *prove* the correctness of the generated code. We consider it an interesting direction for future work to expand our work to code generation for consensus protocols.

Byzantine reliable broadcast and its related broadcast primitives. Byzantine broadcast (BC) is one of the primitives mentioned in the Byzantine generals problem [48], and BC is a synchronous primitive, where the message processing and transmission have a known upper bound. BC defines a problem where a sender sends a message to all replicas, and correct replicas will deliver the same message. RBC is an asynchronous (there is no bound on message processing and transmission) variant of BC that does not guarantee termination of the protocol under a faulty sender. RBC can be used as a black box to build asynchronous BFT [35, 56, 89, 90], secure multiparty computation [10, 55]. Its optimization mostly focuses on the communication complexity and the number of communication rounds in the optimistic case [7, 33, 58, 72].

Other RBC related primitives include echo broadcast [74], consistent broadcast [69], and asynchronous verifiable information dispersal (AVID) [8, 23]. In this paper, we also consider a few variants of RBC, such as AEB mentioned in Table 1.

3 Preliminaries

In the study of Byzantine consensus, there are n replicas $\{p_1, \cdots, p_n\}$, among which at most f are Byzantine faulty replicas. Byzantine faulty replicas behave arbitrarily. Non-faulty replicas are called correct replicas. Since TLA$^+$ does not support subscript, we might also use notions such as p and q to denote replicas in this paper.

The primitives we study assume an asynchronous network, where there does not exist an upper bound on message processing and transmission, but the message sent by a correct sender can eventually be received by a correct receiver. In an asynchronous network, the optimistic value of f is $f \leq \lfloor \frac{n-1}{3} \rfloor$. A Byzantine *quorum* is a set of $\lceil \frac{n+f+1}{2} \rceil$ replicas. If we assume $n = 3f+1$, the quorum size is $2f + 1$.

Byzantine reliable broadcast (RBC). The RBC abstraction allows a sender p_s to reliably broadcast a message to the replicas. An RBC protocol is specified by two events *broadcast* and *deliver* such that the following properties hold:

- **Validity**: If a correct replica p_s *broadcasts* a message m, then every correct replica eventually *delivers* m.
- **Agreement**: If some correct replica *delivers* a message m, then every correct replica eventually *delivers* m.
- **Integrity**: For any message m, every correct replica *delivers* m at most once. Moreover, if a replica *delivers* a message m with sender p_s, then m was previously *broadcast* by replica p_s.

The agreement property can often be split into two properties defined below.

- **Totality:** If a correct replica *delivers* some message, every correct replica eventually *delivers* some message.
- **Consistency:** If some correct replicas *delivers* a message m and another correct replica *delivers* a message m', then $m = m'$.

In this way, consistency is a *safety* property, and totality is a *liveness* property.

We show the pseudocode of Bracha's broadcast in Fig. 1, and show the pseudocode of all RBC protocols studied in this paper in Appendix D.

TLA$^+$ notions. TLA$^+$ (Temporal Logic of Actions) is a formal specification language, developed by Lamport [45, 46] for the modeling and verification of complex systems. TLA$^+$ describes what a system should do and the properties it must maintain via the transition of states. Using Bracha's broadcast protocol (Fig. 1) as an example, we show its TLA$^+$ state transitions in Fig. 2. The core syntax of TLA$^+$ consists of the following components. We also summarize most frequently used notations in Table 2.

- **Constants and variables**. Like most programming languages, constants represent the static parameters of a TLA$^+$ specification and variables denote parameters that might change over time. For data types such as tuples and arrays, a *sequences* data structure is used to represent hashmaps. Specifically,

Bracha's broadcast with p_s as the sender

- As the sender p_s, send (SEND, m) to all replicas, where m is its input
- Upon receiving (SEND, m) from p_s and an (ECHO) message has not been sent
  - Send (ECHO, m) to all replicas
- Upon receiving $n - f$ matching (ECHO, m) messages and a (READY) message has not been sent
  - Send (READY, m) to all replicas
- Upon receiving $f + 1$ matching (READY, m) messages and a (READY) message has not been sent //amplification
  - Send (READY, m) to all replicas
- Upon receiving $n - f$ matching (READY, m) messages
  - *deliver m*

Figure 1: Bracha's broadcast protocol [16].

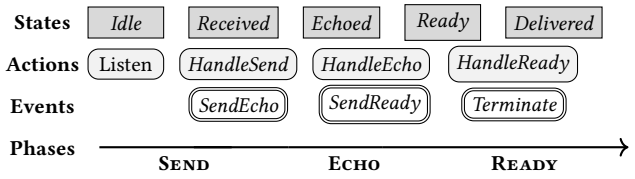


Figure 2: TLA$^+$ state transitions for RBC. A state denotes whether some message has been sent, e.g., *Received* denotes the fact that a message is received from the sender p_s. An action denotes the actions taken after some message is received, e.g., *HandleSend* denotes that a (SEND) message has been received from the sender p_s. An event denotes a local computation that may lead to an action that changes the state. For instance, after the *HandleSend* action, a replica will trigger the *SendEcho* event, after which the state is transitioned from *Received* to *Echoed*.

$$struct1 == [a \mapsto 1], struct2 == [b \mapsto \{\}]$$

denote two hashmaps, where *struct1* maps *a* to *1* and *struct2* maps *b* to a set {}. One can use *struct1.a* or *struct1*["*a*"] to denote *1*.

- **States**. A state denotes an assignment of values to the variables. It represents a system snapshot at a specific moment in time (e.g., *Idle*, *Received*, *Delivered*).
- **Actions**. An action describes how the state changes and is usually triggered via an event (i.e., local computation). Formally, an action is a predicate on a set of unprimed variables v and a set of primed variables v', where v represents the current state and v' represents the next state. For example, the *HandleSend* action is taken if a (SEND) message is received from p_s. After an action is triggered, the state is updated, where the set of received messages by all replicas changes.
- **Invariants**. Invariants are some claims that must be *True* on every reachable state. Invariants are often used to model the safety properties. For instance, the TLA$^+$ statement that models the consistency of RBC can be represented as:

  $Consistency \triangleq \forall\, p, q \in Correct : (delivered[p] \neq "" \wedge delivered[q] \neq "") \Rightarrow delivered[p] = delivered[q]$,

  where $\forall$ denotes *for all*, p and q denote two different replicas, $delivered[p]$ denotes the value delivered by p, and $C1 \Rightarrow C2$ denotes that if condition $C1$ is true, then $C2$ is true.
- **Properties**. Properties, or temporal properties, use a set of temporal operators to describe logical statements across time. Properties are often used to model the liveness properties. For example, the validity property of RBC can be represented as:

  $Validity \triangleq sender \in Correct \Rightarrow \Diamond(\forall\, p \in Correct : delivered[p] = receivedS[sender])$,

  where $C1 \Rightarrow \Diamond C2$ denotes that if condition $C1$ is true, then $C2$ is *eventually* true, and $receivedS[sender]$ denotes the value received from the sender.

Notation	Description
P	All replicas.
$Correct$	All correct replicas and $Correct \subseteq P$.
$Faulty$	Byzantine replicas and $Faulty \subseteq P$.
V	The set of possible messages sent by the sender.
$sender$	The designated sender of an RBC instance, where $sender \in P$.
$receivedS$	Received (SEND) messages of the replicas. A mapping $P \rightarrow V \cup \{""\}$. $receivedS[p]$ is the value received by replica p from the sender; "" denotes that no value has been received.
$receivedE$	Received (ECHO) messages of the replicas. A mapping $P \rightarrow [V \rightarrow \text{SUBSET } P]$. $receivedE[p][v]$ is the set of replicas from which replica p has received an (ECHO) message with content v.
$receivedR$	Received (READY) messages of the replicas. A mapping $P \rightarrow [V \rightarrow \text{SUBSET } P]$. $receivedR[p][v]$ is the set of replicas from which replica p has received a (READY) message with content v.

Table 2: Selective TLA$^+$ notations used in this paper.

TLC. TLC is the model checker for TLA$^+$. It processes TLA$^+$ script as input and identifies potential errors by searching every possible state of the system. Specifically, TLC verifies that the invariants are maintained in every single iterated state and properties eventually hold. TLC consists of the following notions we will use in this paper.

- **Distinct states**. It refers to the total number of unique states that TLC has discovered and stored in its memory (e.g., the state queue). If a TLA$^+$ script includes s distinct states, it means that TLC iterates all of them by the end of the model checking.
- **Diameter**. Diameter represents the maximum number of steps required from one initial state to any reachable state. If a protocol should finish in roughly 10 steps, but TLC reports a diameter of 50, it means TLC has found retry loops, or failure scenarios that take much longer to resolve than the "happy path".
- **Error parser**. Before TLC starts to discover distinct states, it reads the TLA$^+$ script to report syntax errors such as undefined operators.
- **Error trace**. During TLC execution, if any actions violate an invariant, TLC provides the error trace that shows the specific sequence of actions and state transitions.

Evaluation model and metrics. Similar to software code quality, we assess TLA$^+$ scripts based on the following criteria.

- **Syntactic correctness**. A TLA$^+$ script is syntactically correct if there are no syntactic errors.
- **Functional correctness**. A TLA$^+$ script is functionally correct if it correctly models the transition of states of the corresponding protocol.
- **Property checking**. A TLA$^+$ script passes the property checking, if there is no violation of the invariants or the properties.
- **`Pass@k`**. We use the `Pass@k` [29] metric to evaluate syntactic correctness, functional correctness, and property checking. For each experiment (execution), we run our tool `k` times. If at least one trial succeeds (i.e., all the metrics are satisfied), the execution is considered successful. We then repeat each experiment `x` times. If `y` experiments succeed, `pass@k` is `y/x`. We set `x` as five in our experiments.
- **Model efficiency**. A TLA$^+$ script is efficient if it is functionally correct and able to identify security issues, and the number of distinct states and the diameter are small enough. A script is more efficient if it can be run for a shorter period of time. We quantify this efficiency by reporting the distinct states and diameter during TLC model checking.

4 Challenges and Overview of `TLAssist`

4.1 Challenges

Challenge 1: Generating TLA$^+$ code is different from code generation, and there are some unique challenges. Recent progress of LLM has proved that LLM is powerful in code-related tasks [29, 53, 80, 88, 92]. However, writing TLA$^+$ code can be fundamentally different from code generation for programming languages. First, TLA$^+$ is a formal specification language rather than a normal programming language. Lamport defines TLA$^+$ as a language to *model software above the code level and hardware above the circuit level* [47]. In terms of the concrete language, TLA$^+$ is specified in mathematical expressions. We show an example of the pseudocode, TLA$^+$ specification, and software implementation in Golang in Fig. 3.

Since TLA$^+$ is specified in mathematical expressions that are highly succinct, the correctness becomes more challenging to reason about compared to languages that are closer to natural language. Namely, even a minor notational difference can mean completely different things. For instance, in our examples of invariants and properties shown in Sec. 3, the condition $C1 \Rightarrow C2$ denotes that if $C1$ is true, then $C2$ should be immediately true, while $C1 \Rightarrow \Diamond C2$ denotes that if $C1$ is true, $C2$ is eventually true. While adding $\Diamond$ is crucial, it then becomes extremely challenging for LLM to make it right due to its nature of generating output in a probabilistic manner. Second, code generation for programming languages often focuses on organizing one task into sub-tasks, specifying the inputs/outputs of each sub-task, and writing functionalities for each task individually. In contrast, TLA$^+$ requires the modeling of the task (e.g., a protocol or a system design of a protocol) into states, specifying the state transitions, and verifying the correctness of the invariants and properties. This is fundamentally different from writing software code and might involve a long chain of thought. Some recent works on LLM-assisted TLA$^+$ generation for non-consensus-related tasks also echo our finding [15, 91].

- Upon receiving $f + 1$ matching (Ready, m) messages and a (Ready) message has not been sent
  - Send (Ready, m) to all replicas

(a) Pseudocode.

$$
\begin{aligned}
&SendReady(self) \triangleq \\
&\quad \land\ \exists\ m \in V : Cardinality(receivedR[self][m]) \geq f + 1 \\
&\quad \land\ receivedR' = [p \in P \mapsto [v \in V \mapsto \\
&\qquad \text{IF } v = m \text{ THEN } receivedR[p][v] \cup \{self\} \\
&\qquad\qquad \text{ELSE } receivedR[p][v]]] \\
&\quad \land\ \text{UNCHANGED } \langle vars \rangle
\end{aligned}
$$

(b) TLA$^+$ specification. V denotes all possible values that can be sent and received, P denotes all replicas, and *vars* denote variables besides *receivedR*. TLA$^+$ does not have the actual *network communication*. In this example, a *receivedR* parameter is a three-dimensional hashmap used to model the messages sent and received by all replicas. For example, $receivedR[self][v]$ denotes the set of replicas from which the replica receives. In the *SendReady* condition, $Cardinality(receivedR[self][m]) \geq f + 1$ denotes the fact that a replica receives more than $f + 1$ matching (Ready) messages from distinct replicas. After this condition is satisfied, a new state marked by $receivedR'$ is created. For any replica p and message m, the replica *self* that has sent the (Ready) message is added to $receivedR'$.

```
func HandleReady (m) {
  if len (received[m]) >= f + 1 && sendReady == false {
    sendReady = true
    broadcast (READY, m)
  }}
```

(c) Go implementation.

Figure 3: Comparison of the pseudocode, TLA$^+$ specification, and software implementation for the same amplification step of Bracha's broadcast.

Challenge 2: Writing syntactically correct TLA$^+$ code does not mean that TLA$^+$ is correctly written. Like any integrated development environment (IDE), the model checking tool TLC checks the syntactic correctness of a TLA$^+$ script. However, even if a TLA$^+$ script does not have syntactic errors, it does not mean that the script is *useful*, i.e., the script might not be useful in identifying any design issues at all. This is mainly because the formalization in TLA$^+$ overly simplifies the design such that functional correctness is not achieved.

We illustrate a TLA$^+$ example in Fig. 4, where the behavior of a Byzantine replica is highlighted in brown. To equivocate, a Byzantine replica should send different messages to different replicas. However, this script uses the global variable *globalEchos* to store the values received from the (Echo) messages. Thus, all correct replicas will see the same message v, i.e., the Byzantine replica sends the same message to all replicas and cannot equivocate. While this simplified modeling can be caught by a TLA$^+$ expert, this script does not involve any TLC error. Thus, if LLM generates a script like this, no errors can be reported.

```
VARIABLES globalEchos
ByzantineEcho(self) ≜
  ∧ ∃ v ∈ V : globalEchos' = globalEchos ∪ {⟨self, v⟩}
  ∧ UNCHANGED ⟨vars⟩
```

Figure 4: Example of an overly simplified TLA$^+$ script. Due to the use of the *globalEchos* parameter, Byzantine replicas cannot equivocate and equivocation cannot be simulated.

Challenge 3: Writing an efficient TLA$^+$ script requires modification of the workflow and relies on human expertise. When it comes to iterating Byzantine behavior, TLA$^+$ has its unique challenge. Namely, the design philosophy of TLA$^+$ is to iterate over all possible *failure scenarios* to capture the logical errors in the underlying protocol or system design. However, it is impossible to iterate arbitrary behavior. In fact, even if some arbitrary behaviors are iterated, it is easy to exhaust the state space and make the TLA$^+$ script unusable. In the TLA$^+$ examples by TLA$^+$ foundation (i.e., the official maintainer of TLA$^+$), the protocol might be simplified, and a limited number of Byzantine behaviors are iterated. Take Bracha's broadcast as an example, the (Send) message is often omitted, and the iteration of states begins at the point where correct replicas already enter the state *received*. Furthermore, correct replicas have only two possible states: some receive message m, and others do not. In the concrete TLA$^+$ example for Bracha's broadcast protocol [1], each replica's initial state is : $pc \in [Proc \rightarrow \{\text{"V0"}, \text{"V1"}\}]$, where pc is the current state, *Proc* denotes all replicas, "V0" denotes some message received and "V1" denotes no message is received. If the sender is Byzantine, some replicas may have an initialized state as "V0" while some have state "V1".

For people who are familiar with both RBC and TLA$^+$, simplifying the states to this situation might be sufficient for modeling Byzantine behavior. However, it becomes very challenging for human experts who are only familiar with RBC or TLA$^+$ to model Byzantine behavior properly. Later, we show in our case studies that even for human experts, it is challenging to write TLA$^+$ correctly and efficiently, but our `TLAssist` can generate TLA$^+$ specifications with decent quality.

4.2 Overview of `TLAssist`

As shown in Fig. 5, `TLAssist` consists of two stages: specification generation; verification and refinement.

Stage I: TLA$^+$ specification generation. This stage takes the target pseudocode and its security properties as input and generates TLA$^+$ specification in three steps: ① A knowledge base is built based on the TLA$^+$ scripts of some known protocols. Given the TLA$^+$ examples in the knowledge base, structured templates in the JSON format are built manually; ② The structured templates of the knowledge base and the target pseudocode are used as a prompt to LLM, which transforms target pseudocode to the same format as the structured template; ③ A draft TLA$^+$ specification is generated via LLM prompt.

Stage II: Verification and refinement. Stage II takes the draft TLA$^+$ specification from stage I as input and generates a refined target TLA$^+$ specification through an iterative feedback loop. The goal is to generate a useful TLA$^+$ script without syntactic errors, which can be used to verify the security properties. Namely, after TLA$^+$ is generated, we write a script that automatically verifies the correctness via the TLC model checker (step ④). The verification includes facts such as whether there are syntactic issues and whether the number of distinct states is too small or too large, etc. For the latter case, if there are too few distinct states, only limited behaviors are simulated, so the TLA$^+$ script is not that useful; if there are too many distinct states, the TLA$^+$ script can easily explode the memory of the machine. If the verification fails, the parsing file and error trace will be integrated into subsequent prompts of LLM (procedures starting from step ③ are repeated). The iterative process continues until one of two termination conditions is met: 1) the generated TLA$^+$ passes the TLC verification, and the TLA$^+$ script is output; 2) the predefined number of iterations is completed, and TLA$^+$ still has errors, and the TLA$^+$ script and the error trace are output.

After the two stages, the generated TLA$^+$ is ideally: (1) executable with no syntactic errors; (2) accurately formalizes the pseudocode (input).

The two-stage generate-and-refine workflow is similar to many LLM-based tasks [30, 39, 71]. Our main contribution lies in integrating domain-specific knowledge into the workflow to improve the quality of the generated TLA$^+$ scripts. Below, we summarize our main claims, and we will discuss them in detail in Sec. 5.

- First, we manually modify the TLA$^+$ scripts in the knowledge base to improve the modeling of Byzantine behavior (Claim 1 in Sec. 5.1). In this way, the generated TLA$^+$ scripts are both useful and efficient.
- Second, the structured template is carefully designed with both manually written comments on the knowledge base and grouped lines of the TLA$^+$ scripts (Claim 2). As mentioned in several works [14, 41, 52], prompting in a more structured format, such as JSON and Markdown, can improve output quality. We build upon this idea in our work. Specifically, we extract the behaviors of each replica from the TLA$^+$ script in the knowledge base and group the behaviors of each party into the same block. Later, we show that without the template, a direct LLM prompt always fails. Furthermore, removing some blocks from our structured template also lowers the quality of the generated scripts.
- Finally, our feedback information in stage II can improve the quality of the generated script (Claim 3). One nuance here is that an error trace reported via TLC may denote either an incorrect design of RBC or a not well-written TLA$^+$ script. Our feedback loop provides *enough* information, such as the errors and the number of states, for the LLM to correct the syntactic errors. During our experimentation, we find that our feedback loop is significantly more effective than regenerating TLA$^+$ from scratch.

We justify our claims experimentally (Sec. 5.3). Our case studies (Sec. 6) show that our generated TLA$^+$ scripts are both useful and efficient.

[1] TLA$^+$ Examples: https://github.com/tlaplus/Examples/blob/master/specifications/aba-asyn-byz/aba_asyn_byz.tla Last accessed: Apr 2026.

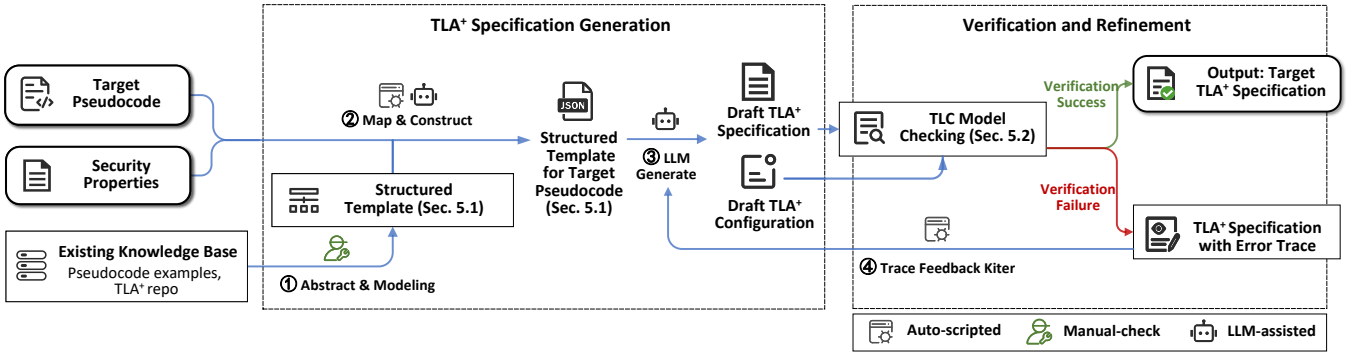


Figure 5: Workflow of TLAssist for automated TLA+ generation and refinement.

5 TLAssist: TLA+ Generation for Protocol Verification

We now discuss our workflow and the main claims in detail. In Sec. 5.3, we provide justification for the claims.

5.1 Stage I: TLA+ Specification Generation

Recall that stage I takes input as the pseudocode and security properties of a protocol and generates a draft TLA+ specification. A crucial step in this stage is defining the structured template for the TLA+ examples in the knowledge base, and using the LLM to transform the target pseudocode into the same structured template. We now discuss our knowledge base and the structured template in detail.

The knowledge base. We use the TLA+ examples from the official GitHub repository by TLA+ foundation[2] to build the knowledge base[3]. The repository has over 60 TLA+ examples for different distributed protocols.

We choose three protocols to build our knowledge base: folklore reliable broadcast (FRBC)[4] [28, Figure 4], asynchronous Byzantine consensus (ABC) [5] [18, Figure 3], and asynchronous reliable broadcast (ARBC)[6] [74, Figure 7]. The three protocols are variants of RBC, although the names (which are directly copied from the official repository) can be misleading. We summarize them below.

- FRBC is a crash fault-tolerant broadcast protocol built from failure detectors.
- ABC is an asynchronous binary agreement protocol [57, 89], where each replica holds one bit as input and correct replicas agree on the same bit.
- ARBC is a variant of RBC. In terms of the concrete construction, it can be viewed as Bracha's broadcast where the (Echo) phase is removed and each replica can deliver many times.

The security definitions of the three protocols are thus quite different from those for RBC. We chose these three as they are the examples from the official repository that are closest to RBC.

When we build our knowledge base, we refine the official TLA+ scripts and claim that the refinement can better model Byzantine behavior. We will discuss the details in Sec. 5.3.

> ***Claim 1:*** *Refining the modeling of Byzantine behaviors on top of the official TLA+ examples by TLA+ foundation can improve the efficiency and effectiveness of our generated TLA+ specifications for RBC.*

The structured template in JSON. As shown in Fig. 6, our JSON template follows the basic notions of TLA+ shown in Sec. 3 and organizes the examples into six major blocks: Constants, Variables, Actions, Invariants, Properties, and llm_hints. The first five blocks match the TLA+ notions, and the last block consists of manually written comments. For the first five blocks, the main novelty is the structure. First, for some of the blocks, the key and values are directly copied from the TLA+ script. For example, the `constant name` and `comment` on the constant are directly copied from the TLA+ script. We show an example that transforms the TLA+ specification in Fig. 3b into Fig. 7, i.e, our JSON template. Second, for some blocks, the key and values are copied from TLA+ script but are structured differently. For instance, given a variable `variable name`, the `type` (which is used for type checking of the variable) and `init` (the initialized value of the variable) are often defined in different blocks in the TLA+ script. However, we group the definitions under the same JSON block in our structured template.

Furthermore, we manually add comments in natural language and `llm_hint` to connect the pseudocode with the TLA+ script. The LLM hints are mostly instructions on how to better simulate Byzantine behaviors. For instance, `comments on the invariants` are the security properties defined in the paper of the corresponding protocol.

> ***Claim 2:*** *The structured template is crucial from the perspective of prompt engineering to improve the quality of the generated TLA+ scripts.*

Transforming the input (pseudocode and security properties) into a draft TLA+ specification. We build the structured template for the TLA+ examples in the knowledge base manually, denoted as the structured template (knowledge base). We then use the built

[2] TLA+ foundation: https://foundation.tlapl.us/. Last accessed: Apr 2026.
[3] TLA+ Examples: https://github.com/tlaplus/Examples Last accessed: Apr 2026.
[4] TLA+ of the folklore reliable broadcast: https://github.com/tlaplus/Examples/tree/master/specifications/bcastFolklore. Last accessed: Apr 2026.
[5] TLA+ of the asynchronous Byzantine consensus protocol: https://github.com/tlaplus/Examples/blob/master/specifications/aba-asyn-byz/aba_asyn_byz.tla. Last accessed: Apr 2026.
[6] TLA+ of the asynchronous reliable broadcast protocol: https://github.com/tlaplus/Examples/tree/master/specifications/bcastByz. Last accessed: Apr 2026.

```
{
  "Constants": [{
    "constant name": "comments on the constant",
  }],
  "Variables": [{
    "variable name": {
       "description": "comments on the variable (**)",
       "type": "type invariants (*)",
       "init": "initialized value (*)",
    }],
  "Actions": [{
    "name" : "action name",
    "conditions": "conditions of the action",
    "effect" : "event after the condition is triggered",
  }],
  "Invariants": [{
    "name" : "invariant name",
    "description": "comments on the invariants (**)",
    "spec_instance": "invariant definition",
  }],
  "Properties": [{
    "name" : "invariant name",
    "description": "comments on the property (**)",
    "spec_instance": "property definition",
  }],
  "llm_hint (**)": {
    "note" : "comments on the notes (**)",
  }
}
```

Figure 6: The JSON template. The values marked with (*) are copied from TLA$^+$ examples in the knowledge base but the copied lines are not necessarily adjacent in the examples. The values marked with (**) are manual comments to improve the quality of the prompts.

```
{
  "Actions": [{
    "name" : "SendReady (self)",
    "conditions": "∃ m \in V :
    ↪  Cardinality(receivedR[self][m]) ≥ f + 1",
    "effect" : "Choose such a v, ∀ p ∈ P : receivedR[p][v]
    ↪  := receivedR[p][v] ∪ {self}"
  }],
}
```

Figure 7: The structured template of TLA$^+$ specification in Fig. 3b.

templates as a prompt to LLM, which takes as input the target pseudocode and outputs a structured template of the target pseudocode. After the structured template (target pseudocode) is generated, we use it as the input of another prompt, which generates a draft TLA$^+$ specification. In the prompts, we manually add some comments to improve the quality of the output. We provide the details of the two prompts in Appendix E.

5.2 Stage II: Verification and Refinement

Stage II takes the draft TLA$^+$ specification as input and generates a refined target TLA$^+$ specification through an iterative process. The goal is to generate a useful TLA$^+$ script with the syntactic correctness, functional correctness, and property checking as defined in Sec. 3.

There are three outcomes by the end of the feedback loop:

Case 1: The target TLA$^+$ specification is correctly output without any errors reported. The TLA$^+$ specification is correctly generated, and the target pseudocode does not violate its security properties.

Case 2: The target TLA$^+$ specification is correctly output with at least one error trace that denotes the violation of some security property. The TLA$^+$ specification is correctly written. However, an error trace is reported due to the violation of some security property. In this case, the target pseudocode fails to achieve the desirable security properties.

Case 3: The target TLA$^+$ specification is not properly written. `TLAssist` fails to generate a correct TLA$^+$ specification, i.e., either due to failure of syntactic correctness or functional correctness.

The distinction between the three cases is usually not that clear. We manually check *all* generated TLA$^+$ specifications to classify the output. For the second case, we also manually check the pseudocode and the security proofs to validate the results. We will expand the findings later in our case studies. Below, we focus on discussing stage II.

The model checking. We verify the security properties via the TLC model checker. To run a TLA$^+$ script, TLC needs a configuration file as shown in Fig. 8. In `TLAssist`, the configuration file is filled by LLM via our second prompt in stage I. It is worth mentioning that in this concrete example, the V constant is important in simulating Byzantine behavior. Specifically, it specifies that the sender can equivocate by sending *v1* or *v2* to different replicas. Although the sender can behave arbitrarily, modeling only two possible values is good enough in our study of RBC. We will discuss in details in Sec. 5.3 when we discuss the justification of claim 1.

```
CONSTANTS
   Correct = {p1, p2, p3}
   Faulty = {p4}
   V = {v1, v2}
SPECIFICATION Spec        //Specifies the behavior specification
checked by TLC
INVARIANT Consistency
PROPERTY Validity
PROPERTY Totality
```

Figure 8: Example of a TLC configuration.

Trace feedback. The generated TLA$^+$ script and configuration file are executed by TLC automatically. If there are syntactic errors or violations of some invariants or properties, an error trace is automatically created by TLC. A tricky part of the feedback loop is that the violation of some security property might either be caused by the pseudocode that is not provably secure or the fact that the TLA$^+$ script is not correctly written. We show an example in Appendix E.1 and our complete feedback prompt in Appendix E.2.

Claim 3: *The trace feedback loops effectively refines the TLA$^+$ script.*

5.3 Justification of Claims 1-3

We justify claims 1-3 via experiments. Below, we describe our experimental setup and our experimental results. The same setup will be used for our case studies shown later.

Setup. We implement `TLAssist` in Python, and the codebase involves around 1,500 LoC. We evaluate the performance of `TLAssist` on a general-purpose computer with an AMD Ryzen 9 AI 370 processor, 8 cores, and 16GB of memory. We use three LLMs for our work: `GPT-4.1`, `Kimi-K2-0711-Preview`, and `Claude-Sonnet-4.5`. We also use one small model `Qwen2.5-Coder-7B-Instruct` for one of our experiments. In the following, unless otherwise mentioned, our experiments are conducted using `Claude-Sonnet-4.5`.

We use three provably secure protocols for our experiments, including Authenticated Echo Broadcast (AEB) [21, 74], Authenticated Double-Echo Broadcast (ADB) [21], and Bracha's broadcast [18]. Note that ADB is equivalent to Bracha's broadcast. Since their pseudocodes are written differently, our experiments can also be used to assess the impact of pseudocode styles on our `TLAssist` tool.

Improving the knowledge base for simulation of Byzantine behavior (Claim 1). As mentioned above, the official TLA$^+$ examples omit the (Send) message, and Byzantine behavior is modeled into two states for each replica: the replica receives some m; the replica does not receive any message. If correct replicas receive some messages, the messages they receive are the same. Our finding is that such modeling cannot fully model the behavior of the sender and needs to be improved. Namely, the violation of the safety properties (e.g., consistency) requires some correct replicas to *deliver m1* and some correct replicas to *deliver m2*. However, if correct replicas that receive some message from the sender receive the same message, consistency will never be violated.

We thus propose a refined model that defines the constant V as $\{m, m'\}$, i.e., correct replicas might receive different messages from the sender. The modeling of the correct sender and the Byzantine sender is now:
$receivedS = [p \in Correct \mapsto \text{CHOOSE } v \in V : \text{TRUE}]$ //correct sender;
$receivedS \in [Correct \to V \cup \{\text{""}\}]$ //faulty sender.

Namely, if the sender is correct, all replicas begin the simulation with the state of receiving some $v \in V$. If the sender is incorrect, correct replicas might begin with the state of receiving a distinct value in V or not receiving any value. Furthermore, we also adapt other variables to model the Byzantine behavior mentioned above.

The simplification of Byzantine behavior above is reasonable. Namely, if the violation of some security property is possible where the sender sends many distinct messages, the violation should be possible if the sender sends at most two distinct messages. Note that other Byzantine behaviors such as omission (a message that is supposed to be sent to *all* replicas is sent to a subset of replicas) are not simplified and they are by default iterated in TLC.

We use Bracha's broadcast to conduct an experiment to validate the claim above. Specifically, we remove the amplification step in Fig. 1 and generate a TLA$^+$ script using `TLAssist` (with our improvement). We also use the default approach in the official example as a comparison. In our case, the script we generate takes about 1 minute to report an error trace indicating a violation of the totality property. The model's diameter is 22, and the number of distinct states is 1.36×10^6. In contrast, for the script using the vanilla modeling, no errors are reported while the diameter is 14, and the number of distinct states is only 8,596.

Protocol	Template?	SC		FC		PC	
		Pass@1	Pass@3	Pass@1	Pass@3	Pass@1	Pass@3
AEB	No	60.0%	80.0%	20.0%	40.0%	0.0%	0.0%
ADB	No	60.0%	100.0%	40.0%	60.0%	0.0%	20.0%
Bracha*	No	100.0%	100.0%	40.0%	60.0%	20.0%	40.0%
AEB	Yes	100.0%	100.0%	80.0%	100.0%	100.0%	100.0%
ADB	Yes	100.0%	100.0%	100.0%	100.0%	80.0%	100.0%
Bracha*	Yes	100.0%	100.0%	100.0%	100.0%	80.0%	100.0%

Table 3: Performance of TLA$^+$ generation with and without our structured template. * Bracha denotes Bracha's broadcast. LLM used: `Claude-Sonnet-4.5`. SC, FC, PC denote syntactic correctness, functional correctness, and property checking, respectively.

Protocol	Functional correctness		Property checking	
	Pass@1	Pass@3	Pass@1	Pass@3
ADB	80.0%	100.0%	60.0%	100.0%
ADB w/o variables	60.0%	80.0%	20.0%	20.0%
ADB w/o actions	60.0%	60.0%	0.0%	0.0%

Table 4: Performance of TLA$^+$ generation with different structured templates. LLM used: `Claude-Sonnet-4.5`.

$Init \triangleq$
  $\wedge\ echos = [p \in P \mapsto [q \in P \mapsto Nil]]$
$UponDeliver(self) \triangleq$
  $\wedge\ Cardinality(\{\ p \in P : echos'[self][p] = \text{Message}$
$\}) > (N + F) \div 2$
  $\wedge$ UNCHANGED $\langle vars \rangle$
$SendEcho(self) \triangleq$
  $\wedge\ receiveS[self] \in V$
  $\wedge\ \ Cardinality(echos[self][receiveS[self]]) = 0$
  $\wedge$ UNCHANGED $\langle vars \rangle$
$Totality \triangleq \Box(\forall p, q \in Correct :$
    $(delivered[p] \wedge delivered[q]) \Rightarrow \text{TRUE}\ )$

Figure 9: Example of an incorrect TLA$^+$ generation of Bracha's broadcast via raw LLM. gray, brown, red denote errors that cause violation of syntactic correctness, functional correctness, and property checking, respectively. The errors of *Nil* and *echos'* denote undefined replicas and undefined state, respectively. The wrong treatment of the *Cardinality*() function creates a situation with no state transition at all. Finally, the definition of the totality property uses a $\Box$ notion that denotes *always* rather than *eventually*, and thus cannot properly model the security property.

Enhancing TLA$^+$ generation through the structured template (Claim 2). To evaluate the effectiveness of our structured template,

we study three protocols: AEB, ADB, and Bracha's broadcast. We evaluate the `Pass@k` of three metrics of the protocols with and without the structured template: syntactic correctness, functional correctness, and property checking. As shown in Table 3, the pass rate of all three metrics is consistently higher when the structured template is used. The pass rate for property checking (i.e., whether the protocol achieves the desirable security properties) is highly different with and without our template.

We believe the reason is two-fold. First, as mentioned by several prior works [14, 41, 52], structured prompts via Markdown or JSON can improve the quality of the LLM outputs. In our case, although we add some other content to our prompt, the JSON template is highly structured. This can guide LLM to create better reasoning about the TLA$^+$ examples in the knowledge base. Second, we manually group the necessary information in the same block, i.e., the type and initialized value of each variable are intentionally grouped into one block. Such manual grouping creates an even better structure of the TLA$^+$ examples.

We show a commonly failed example generated by a raw LLM in Fig. 9. Common syntactic errors include treating *Nil* as a replica and referencing a non-existent state. Common failures on functional correctness include omitting critical state transitions. Finally, common failures that lead to security property violations include incorrect modeling of the invariants and properties.

We also conducted an evaluation to assess the impact of different components in our structured template components. We focus on the ADB protocol in this case. As shown in Table 4, we remove the blocks on variables and actions, and repeat our experiments. Our results show that the performance is the highest using our final structured template.

Note that ADB is actually equivalent to Bracha's broadcast, but they are written in different pseudocode styles. We show the pseudocode of ADB in Fig. 15 of Appendix D. Thus, our experiments shown in Table 3 can also be viewed as results on a different pseudocode style. Our results show that different pseudocode styles have little impact on the quality of the generated TLA$^+$ scripts.

Model	Protocol	SC		FC		PC	
		`one` *	`three` *	`one` *	`three` *	`one` *	`three` *
GPT	AEB	20.0%	60.0%	20.0%	40.0%	0.0%	40.0%
	ADB	20.0%	60.0%	20.0%	20.0%	0.0%	40.0%
	Bracha	40.0%	80.0%	20.0%	20.0%	20.0%	60.0%
Kimi	AEB	60.0%	80.0%	20.0%	40.0%	20.0%	40.0%
	ADB	60.0%	80.0%	0.0%	20.0%	0.0%	40.0%
	Bracha	100.0%	100.0%	0.0%	0.0%	60.0%	80.0%

Table 5: Performance comparison of different LLMs on TLA$^+$ generation through trace feedback. Bracha denotes Bracha's broadcast. Abbreviations used in the table: GPT (`GPT-4.1`), Kimi (`Kimi-K2-0711-Preview`). *Here, `one` and `three` denote the pass rate after running one round and three rounds of the feedback loop, respectively. SC, FC, PC denote syntactic correctness, functional correctness, and property checking, respectively.

Refining TLA$^+$ output quality through trace feedback (Claim 3). We show in Table 5 the success rate of the three metrics after running one round and three rounds in our feedback loop, respectively. As shown in the table, our feedback loop can significantly increase the pass rate of all three metrics.

Model	Protocol	SC		FC		PC	
		`Pass@1`	`Pass@3`	`Pass@1`	`Pass@3`	`Pass@1`	`Pass@3`
GPT	AEB	60.0%	80.0%	40.0%	60.0%	40.0%	60.0%
	ADB	60.0%	100.0%	20.0%	60.0%	40.0%	60.0%
	Bracha	80.0%	100.0%	20.0%	60.0%	60.0%	80.0%
Kimi	AEB	80.0%	100.0%	40.0%	100.0%	40.0%	100.0%
	ADB	80.0%	100.0%	20.0%	40.0%	40.0%	60.0%
	Bracha	100.0%	100.0%	0.0%	20.0%	80.0%	80.0%
Qwen2.5	AEB	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
	ADB	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
	Bracha	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%

Table 6: Performance comparison of different language models on TLA$^+$ generation. Bracha denotes Bracha's broadcast. Abbreviations used in the table: GPT (`GPT-4.1`), Kimi (`Kimi-K2-0711-Preview`), and Qwen2.5 (`Qwen2.5-Coder-7B-Instruct`). SC, FC, PC denote syntactic correctness, functional correctness, and property checking, respectively.

Model	Avg. cost (USD)	Avg. latency (s)	Pass rate (%)
GPT	0.017	11.7	56.7%
Kimi	0.005	30.9	66.7%
Claude	0.029	18.4	93.3%

Table 7: Comparison of different LLMs in terms of monetary cost, latency, and pass rate for property checking.

TLA$^+$ generation using different LLMs and a small model. We show in Table 6 the performance of different LLMs. Our results show that large language models can correctly generate useful TLA$^+$ scripts while the experiments on the small model `Qwen2.5` fail. The results are expected, as TLA$^+$ specifications can be quite challenging to make correct. We also report the monetary cost (in USD), the latency of generated scripts, and the pass rate of the LLMs in Table 7. Our results show that our `TLAssist` tool is quite cost-effective.

6 Case Studies

We conduct four case studies using `TLAssist`: (2, 3, 4)-Optimistic RBC [72], SigBRB [7], CT RBC [56], and ADD-based RBC [33]. As mentioned in the introduction, `TLAssist` can assist protocol designers and developers in terms of improving the design and development process. It can also generate TLA$^+$ scripts with decent quality, so the model checking process can be facilitated.

6.1 Case Study 1: The (2, 3, 4)-Optimistic RBC Protocol

(2, 3, 4)-Optimistic RBC is a recent RBC protocol that provides a latency-reduction fast path that terminates in two communication steps. For completeness, we show the pseudocode of the protocol in Fig. 10. In the paper, a TLA$^+$ script is provided [72, Figure 11]. Meanwhile, a different TLA$^+$ script is included in the open-source repository[7]. We call them TLA$^+$ (paper) and TLA$^+$ (open-source), respectively.

[7]TLA$^+$ (open-source): https://github.com/qyu100/SFSailfish/blob/overview/formal-specifications/TwoStepOptimiticBroadcastSafety.tla (Last accessed: April 2026)

(2, 3, 4)-Optimistic RBC protocol with p_s as the sender

- **Propose.** The sender p_s sends (Propose, m) to all replicas.
- **Echo.** Upon receiving the first (Propose, m) from the sender, replica p_i sends (Echo, m) to all parties.
- **Vote.** Upon receiving (Echo, m) from $\lceil \frac{n}{2} \rceil$ non-sender replicas, send (Vote, m) to all replicas if not already sent.
- **Ready.** Replica p_i sends a (Ready, m) message to all replicas (if not already sent) under either of the following conditions:
  - Upon receiving (Echo, m) from $\lceil \frac{n+f-1}{2} \rceil$ non-sender replicas,
  - Upon receiving (Vote, m) from $\lceil \frac{n+f-1}{2} \rceil$ non-sender replicas,
  - Upon receiving $f+1$ (Ready, m).
- **Opt Commit.** Upon receiving (Echo, m) from $\lceil \frac{n+2f-2}{2} \rceil$ non-sender replicas, p_i *delivers* m and terminates.
- **{3,4}-step Commit.** Upon receiving $2f+1$ (Ready, m) messages, p_i *delivers* m and terminates.

Figure 10: (2, 3, 4)-Optimistic RBC protocol with p_s as the sender [72, Figure 1, Section 3]. The notions are slightly changed to be consistent with other protocols in this paper.

We compare TLA$^+$ (paper) and TLA$^+$ (open-source) with the TLA$^+$ scripts generated by `TLAssist`. Additionally, we use our generated TLA$^+$ script to assess the protocol. Our findings on this case study are summarized below.

- The TLA$^+$ (paper) can be successfully run by TLC without any errors reported. However, it has a subtle issue such that even if the amplification step is removed, no errors can be reported. Note that a violation of the totality property should be reported.
- The TLA$^+$ (open-source) does not have the issue above. However, the script has an extremely large number of distinct states, which cannot be easily run on our machine.
- Our generated script does not suffer from the issues above. Furthermore, our generated script reports a subtle liveness issue in the design of the (2, 3, 4)-Optimistic RBC protocol. Specifically, a violation of the totality property is reported. We have manually checked the proofs and contacted the authors of (2, 3, 4)-Optimistic RBC to confirm the issue.

The issue on TLA$^+$ (paper). The slow path of (2, 3, 4)-Optimistic RBC (that terminates in four communication steps in the worst case) is the same as that in Bracha's broadcast. Namely, upon receiving $f+1$ matching (Ready, m) messages and a (Ready) message has not been sent, each replica sends a (Ready, m) to all replicas. If such an amplification step is omitted, the totality property fails to hold. Namely, if a correct replica *delivers* some message m, there would be no guarantee that every correct replica eventually sends a (Ready, m) message.

In TLA$^+$ (paper), if we remove the amplification step, i.e., line 45: $\vee$ *Cardinality*(*Readys*(*self*, *v*)) $\geq F + 1$; of [72, Figure 11], the TLC is expected to report an error trace on the violation of totality. However, no errors are reported. After manual investigation, we found that the issue is likely due to incomplete modeling of the Byzantine sender. As shown in Fig. 11a, TLA$^+$ (paper) uses a *pc* hashmap to denote whether some party has sent a message. Here, $pc' = [pc \text{ EXCEPT } ![self] = \text{"Done"}]$ denotes that the sender has already sent a message. The transition from the state "l1" to "Done" indicates that a Byzantine sender can only send at most one message, i.e., the equivocation cannot be properly modeled. In contrast, the script generated by `TLAssist` does not suffer from this issue. As shown in Fig. 11b, there is a dedicated *ByzantineEcho* action that can be triggered indefinitely. Other RBC phases are modeled similarly. Thus, it can well iterate real Byzantine behavior in TLC.

```
CONSTANTS P, Faulty, V
Byzantine(self) ≜
  ∧ pc[self] = "l1"
  ∧ ∃ v ∈ V :
    ∃ t ∈ {"propose", "echo", "vote", "ready"} :
      ∃ p ∈ P \ Faulty : msgs' =
          (msgs ∪ {[src ↦ self, dst ↦ p, type ↦ t, val ↦ v]})
  ∧ pc' = [pc EXCEPT ![self] = "Done"]
  ∧ UNCHANGED ⟨sender, bcastValue, delivered⟩
```

(a) Incomplete modeling of the Byzantine sender in TLA$^+$ (paper).

```
ByzantineEcho(self) ≜
  ∧ self ∈ Faulty
  ∧ ∃ d ∈ Correct, v ∈ V :
    ∧ receivedE' = [receivedE EXCEPT ![d][v] = @ ∪ {self}]
  ∧ UNCHANGED ⟨sender, receivedS, delivered,
                receivedV, receivedR⟩
```

(b) Modeling of Byzantine behaviors in TLA$^+$ (`TLAssist`).

Figure 11: Comparison of modeling of Byzantine behaviors. The issue that causes the failure of identifying totality violation is highlighted in gray.

	Distinct states	Time
TLA$^+$ (paper)	51,419	< 1 min
TLA$^+$ (open-source)	$>> 5.0 \times 10^9$	N/A
TLA$^+$ (TLAssist)	4.09×10^8	50 min

Table 8: Comparison of TLA$^+$ (paper), TLA$^+$ (open-source), and TLA$^+$ (`TLAssist`) for $n = 4$ and $f = 1$. The number of distinct states of TLA$^+$ (open-source) is 5.0×10^9 after running the script for over eight hours and the number is still growing fast. This means that the memory of the CPU will soon be fully used and the script cannot be successfully run.

The issue on TLA$^+$ (open-source). The TLA$^+$ (open-source) does not have the issue mentioned above. However, it has an extremely large number of distinct states, which causes a memory explosion in the machine we use for validation. We show the number of distinct states and runtime for $n = 4$ and $f = 1$ in Table 8. Both TLA$^+$ (open-source) and TLA$^+$ (`TLAssist`) have a significantly larger number of distinct states than TLA$^+$ (paper), due to the more comprehensive modeling of Byzantine behavior. Compared to TLA$^+$ (open source), the script generated by `TLAssist` has about 4.09×10^8 distinct states and can be finished in 50 minutes. In contrast, TLA$^+$ (open-source) cannot be successfully finished. We switch to a more powerful machine (the *c5.12xlarge* instance on Amazon Web Services with 48

vCPU and 96GB of memory) and run the script for eight hours. The number of distinct states was still growing, so we stopped running the script.

The subtle liveness issue of (2, 3, 4)-Optimistic RBC. We use the script generated by `TLAssist` to identify a subtle liveness issue of (2, 3, 4)-Optimistic RBC. We provide the error trace of the violation in Appendix F. Since the error trace is not that readable for readers who are not familiar with TLA$^+$, we manually summarize the attack path below. Namely, consider a scenario with $n = 10$ and $f = 3$, where p_1, p_2, p_3 are Byzantine and p_1 is the sender. The Byzantine sender p_1 partitions correct replicas into two groups: $G_1 \in \{p_4, p_5, p_6, p_7\}$ and $G_2 \in \{p_8, p_9, p_{10}\}$.

- p_1 sends (Propose, m_1) to G_1 and (Propose, m_2) to G_2. Excluding the behavior of Byzantine replicas, there are four (Echo, m_1) messages and three (Echo, m_2) messages.
- Both p_2 and p_3 send (Echo, m_1) and (Echo, m_2) to all replicas. There are now six (Echo, m_1) messages and five (Echo, m_2) messages. According to the (2, 3, 4)-Optimistic RBC protocol, upon receiving six (Echo, m_1) messages (i.e., $\lceil \frac{n+f-1}{2} \rceil$ (Echo, m_1) from non-sender replicas in the pseudocode), each replica in G_1 sends (Ready, m_1). Meanwhile, upon receiving five (Echo, m_2) messages (i.e., $\lceil \frac{n}{2} \rceil$ (Echo, m_2) messages from non-sender replicas), each replica in G_1 sends (Vote, m_2). After this, there are four (Ready, m_1) messages and four (Vote, m_2) messages. Note that in (2, 3, 4)-Optimistic RBC, each correct replica is allowed to send (Vote, m_2) after sending (Echo, m_1), so the conditions above hold.
- Both p_2 and p_3 send (Vote, m_2) to all replicas in G_2, but not replicas in G_1. There are now six (Vote, m_2) messages, i.e., there are $\lceil \frac{n+f-1}{2} \rceil$ (Vote, m_2) message from non-sender replicas. According to the protocol, any replica in G_2 (which has not sent a (Ready) message yet) will send a (Ready, m_2) message.
- p_1, p_2 and p_3 send (Ready, m_1) to all replicas in G_1. Since any replica in G_1 receives seven (Ready, m_1), it *delivers* m_1 via the *{3,4}-step commit* step in (2, 3, 4)-Optimistic RBC.
- Meanwhile, all replicas in G_2 have already sent a (Ready, m_2) message. Thus, they will not send (Ready, m_1), even after $f + 1$ (Ready, m_1) are received. The amplification phase cannot be triggered to satisfy the totality property of the protocol.

The issue above is caused by an error in the proof, where the amplification step cannot be used to ensure totality anymore. In fact, in the worst-case scenario, the protocol may also violate the consistency property, but the problem can be fixed by increasing the threshold for the *{3,4}-step commit* step. We omit the details here.

> **Finding 1:** Writing TLA$^+$ script is challenging even for experts. `TLAssist` can be helpful for domain experts to generate an efficient TLA$^+$ script that captures the security violation of the design.

6.2 Case Study 2: SigBRB, CT RBC, and 2-round-BRB

We study SigBRB [7], CT RBC [23] and 2-round-BRB [6, Fig. 1]. SigBRB is a communication-efficient RBC protocol based on threshold signatures. CT RBC is an asynchronous verifiable information dispersal protocol, where the *dispersal phase* can also realize RBC. 2-round-BRB is an RBC protocol with a fast path that terminates in two communication steps. While our generated TLA$^+$ scripts have not identified any design issues for the three protocols, we use them to answer an interesting question:

Can we remove or optimize some communication step to make the protocol still provably secure?

To answer the question above, we study the following three variants:

SigBRB (w/o amplification). SigBRB has a very special amplification design. Briefly speaking, as shown in Fig. 17 in Appendix D, SigBRB uses a consistent broadcast [69] style message dissemination phase, followed by three communication steps for each replica to retrieve the sent message. Specifically, before each replica is ready to send a (Ready) message, it sends a (Disperse) message if it has not done so yet. It was mentioned in the paper that there exists a half-baked idea where if the *Ready* phase is removed, correctness cannot be guaranteed. In our case study, we revise the protocol slightly by removing the amplification step, but keeping the other workflow. We call it SigBRB (w/o amplification).

CT RBC (optimized). In the study of HoneyBadgerBFT [56], CT RBC is used, but the protocol is slightly revised. We show the pseudocode in Fig. 16 of Appendix D. In the vanilla CT RBC protocol, both (Echo) and (Ready) messages carry the data fragments corresponding to the *broadcast* message. In the variant by HoneyBadgerBFT, the (Ready) messages only carry the Merkle tree root of the data fragments. Such an optimization can reduce the network bandwidth consumption, but it is not even mentioned by the paper as a contribution. We study this variant and assess whether the optimization is correct.

2-round-BRB (w/o forwarding). The protocol has a special *forwarding* step in the *commit* phase (Fig. 14 in Appendix D). We remove the forwarding step.

Our case studies on the three variants can be useful for both protocol designers, beginners in this field, and even developers who do not have strong expertise in this field. For instance, developers might often wonder whether some engineering optimization might end up implementing an incorrect protocol. Beginners might be interested in learning whether some communication steps in the protocol are necessary before understanding the proofs. We believe our `TLAssist` tool provides a nice way of *learning* the protocols.

Analysis results. Our generated TLA$^+$ scripts of the three protocols above show that: (1) SigBRB (w/o amplification) and 2-round-BRB (w/o forwarding) lead to the violation of the totality property; (2) the CT RBC (optimized) protocol is correct. We show the details of the discussion in Appendix F.2.

> **Finding 2:** `TLAssist` can be helpful for beginners and developers who lack strong expertise in the research on distributed broadcast protocols. Namely, by slightly revising the protocol and using TLA$^+$ to find error traces, we can facilitate understanding of the protocol without diving deep into the proofs.

6.3 Case Study 3: ADD-based RBC

We study ADD-based RBC, an RBC protocol built from online error correcting (OEC) code [11]. In fact, the SigBRB protocol mentioned

earlier is also built from OEC, so the same findings also apply to SigBRB. We show the pseudocode of ADD-based RBC in Fig. 18 of Appendix D.

The OEC has a very special feature with computation using extra trials. Namely, consider an erasure coding algorithm such as Reed-Solomon code [68] that corrects up to f errors. The encode () algorithm takes as input a message m and encodes it into n fragments. In RBC (CT RBC, for example), the sender sends one fragment to each replica, and replicas exchange their received fragments. When some replica collects at least $t + 1$ valid fragments, it runs a decode () algorithm, which takes as input the fragments and outputs the message m. Different from the vanilla decode () algorithm, the OEC starts to decode when there are at least $2f + 1$ fragments (denoted as **d**). The idea is to obtain a message m that matches with at least $2f + 1$ fragments in **d**. If the trial fails, OEC may take up to r trials (where $r \in [f]$). In this way, if $2f + 1$ correct replicas hold *correct* fragments (that are generated via the same encode () algorithm), OEC can ensure correct output. ADD-based RBC exploits this idea to build a communication-efficient RBC protocol.

In our case study, our generated script by TLAssist can be correctly executed with no reported errors. However, we find that the TLA$^+$ modeling has its own limitations where the underlying OEC algorithm cannot be easily modeled. Namely, the data fragments can only be modeled as *a set of possible values* (e.g., as shown in Fig. 3) as follows:

$SymbolOf \triangleq \text{CHOOSE } f \in [P \to Symbols] : \quad \forall p, q \in P : (p \neq q) \Rightarrow (f[p] \neq f[q])$

In our case, we have to manually improve the modeling to make it work. Thus, modeling protocols with underlying building blocks cannot fully capture the error-correcting feature of OEC. In fact, even the modeling above is not possible without expert knowledge on RBC, OEC, and TLA$^+$.

In fact, for protocols that rely on different building blocks such as cryptographic algorithms, TLA$^+$ can only simplify them without fully modeling their workflow. We believe this is a limitation of TLA$^+$ itself. However, TLA$^+$ is still quite useful in identifying design issues.

> **Finding 3:** TLA$^+$ cannot fully model the protocols that rely on building blocks such as erasure coding and advanced cryptographic algorithms. However, TLA$^+$ can still be very useful in identifying design issues related to the main workflow.

6.4 Case Study 4: Bracha's Broadcast

As shown in Fig. 1, the predicate of "and an (Echo) message has not been sent" is necessary. We show another variant in Fig. 12. In example 2, the pseudocode is in fact incorrect, as many (Echo) messages might be sent. We find that several known works all follow example 2 and ignore this predicate [33, 56]. For experts in this domain, it is well known that only one (Echo) message can be sent. However, for non-experts and developers, ignoring this predicate in the pseudocode might result in the wrong protocol being implemented.

Our generated script by TLAssist reveals a violation of totality for example 2. The concrete error trace occurs with $n = 4, f = 1$, where p_0 is a Byzantine sender. The error trace proceeds as follows: (1) p_0 sends (Send, m_1) to p_1 and p_2 and (Send, m_2) to p_2 and p_3; (2) p_2, as a correct replica, sends both (Echo, m_1) and (Echo, m_2); (3) p_0 sends (Echo, m_1) to p_1 and p_2, and (Echo, m_2) to p_3; (4) Both p_1 and p_2 send (Ready, m_1), while p_3 sends (Ready, m_2); (5) p_0 sends (Ready, m_1) to p_1 and p_2, and both replicas *deliver* m_1. Meanwhile, p_3 cannot receive $n - f$ matching (Ready) messages and will never *deliver*.

// **Example 1**

– Upon receiving (Send, m) from p_s and an (Echo) message has not been sent, send (Echo, m) to all replicas // Correct

// **Example 2**

– Upon receiving (Send, m) from p_s, send (Echo, m) to all replicas // Incorrect, and a violation of totality can occur

Figure 12: Pseudocode of writing the (Echo) phase of Bracha's broadcast.

6.5 Generality Study: Streamlet and HotStuff

We restrict ourselves to reliable broadcast in this paper, mainly because: (1) other types of consensus protocols are usually more complicated with different security definitions; (2) there are several open-source TLA$^+$ specifications for reliable broadcast, so we have some examples to compare with. We believe our generic workflow can be applied to other types of consensus protocols, but the details still need to be refined.

As a preliminary study, we directly use TLAssist to generate TLA$^+$ for Streamlet [27] and HotStuff [87], two well-known BFT protocols (atomic broadcast [21], to be specific).

With TLAssist, the generated script for both protocols can be correctly executed without any syntactic errors. However, the generated TLA$^+$ scripts do not achieve functional correctness or property checking. Both scripts suffer from missing details such as the epoch number in Streamlet and missing information in quorum certificates in HotStuff. These details are unique to atomic broadcast protocols, and our knowledge base does not have any example of atomic broadcast. Unfortunately, we are still in the process of searching for high-quality TLA$^+$ examples for BFT protocols. Thus, we leave the generality study of TLAssist to other protocols as interesting future work.

> **Finding 4:** TLAssist can already be used to model other types of consensus protocols without any syntactic errors. However, to make the scripts useful in identifying issues, additional efforts are needed.

7 Lessons Learned and Future Work

Our experience of building TLAssist further validates the challenges mentioned in Sec. 4.1. Namely, generating TLA$^+$ scripts that are syntactically correct is less challenging than generating TLA$^+$ scripts that are *useful* in identifying design issues. We emphasize that our contribution lies in integrating domain-specific knowledge with a relatively straightforward LLM-based iterative workflow to generate *useful* TLA$^+$ scripts. We identify some limitations of TLAssist that could lead to interesting future work.

Generality of TLAssist. Our generality study in Sec. 6.5 shows that TLAssist is promising in generalizing to other types protocols. However, to make the generated scripts useful in identifying design or implementation issues, additional works need to be done. In the literature of formal methods, a similar observation has been made [36, 67]. Thus, we believe the treatment of the definitions of invariants and properties is crucial for future work on other types of protocols.

Generalizing TLAssist to work on small models. As we briefly show in one of our experiments, our TLAssist tool cannot be used on small models, as the scripts generated often contain a large number of syntactic errors. It is interesting to know whether our tool can be expanded to support small models.

Using TLAssist to identify implementation issues. As mentioned in the introduction, TLA$^+$ is used by the industry to identify the issues in system design [61]. This can be particularly useful for distributed systems. Namely, even if a protocol is provably secure in theory, there is still a gap between the theory and the real systems. TLA$^+$ can thus be used to fill in the gap. In our paper, we focus on the theory itself. As shown in our case studies, TLAssist can be useful for identifying subtle design issues that domain experts may not easily capture. We believe it is an interesting future direction to expand TLAssist to identify system design issues in the implementation.

8 Conclusion

We present TLAssist, an LLM-assisted tool that transforms the pseudocode of Byzantine reliable broadcast protocols into temporal-logic-of-actions (TLA$^+$) scripts for model checking. We show that, with a carefully designed workflow, we can leverage the power of LLMs to generate TLA$^+$ scripts to identify design issues. Using six reliable broadcast protocols as case studies, we show that TLAssist can benefit domain experts in writing TLA$^+$ to identify design issues, and benefit beginners and developers in learning whether modifications or engineering optimizations of the protocols might lead to design issues.

Acknowledgments

This work was supported in part by the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (JYB2025XDXM114), the Shandong Provincial Key R&D Program (2026CXPT036), the National Natural Science Foundation of China (62602394), the Natural Science Foundation of Shandong Province (ZR2026QC0754), and Tsinghua Shuimu Scholar program.

References

[1] 2010. Apache ZooKeeper. *https://zookeeper.apache.org/* (2010).
[2] 2026. GitHub Copilot Statistics [2026]. *https://www.aboutchromebooks.com/github-copilot-statistics/* (2026). Accessed: 2026-07.
[3] 2026. TLA for Bracha's broadcast. https://github.com/yenanw/brb_tla_plus/tree/main/specs/bracha_brb. Accessed: 2026-07.
[4] 2026. TLA for broadcast algorithms. https://github.com/peterzeller/TLA-broadcast-algorithms/blob/master/tla/reliable_broadcast.tla. Accessed: 2026-07.
[5] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Rama Kotla, and Jean-Philippe Martin. 2017. Revisiting fast practical byzantine fault tolerance. *arXiv preprint arXiv:1712.01367* (2017).
[6] Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. 2021. Good-Case Latency of Byzantine Broadcast: A Complete Categorization. In *PODC*.
[7] Nicolas Alhaddad, Sourav Das, Sisi Duan, Ling Ren, Zhuolun Xiang Mayank Varia, and Haibin Zhang. 2022. Balanced Byzantine Reliable Broadcast with Near-Optimal Communication and Improved Computation. *PODC*.
[8] Nicolas Alhaddad, Sourav Das, Sisi Duan, Ling Ren, Mayank Varia, Zhuolun Xiang, and Haibin Zhang. 2022. Brief announcement: Asynchronous verifiable information dispersal with near-optimal communication. In *PODC*. 418–420.
[9] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. 2018. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *EuroSys*. ACM, 30.
[10] Michael Ben-Or, Ran Canetti, and Oded Goldreich. 1993. Asynchronous secure computation. In *STOC*. ACM, 52–61.
[11] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. 1994. Asynchronous secure computations with optimal resilience. In *PODC*. ACM, 183–192.
[12] Christian Berger, Hans P Reiser, and Alysson Bessani. 2021. Making reads in BFT state machine replication fast, linearizable, and live. In *SRDS*. IEEE, 1–12.
[13] Nathalie Bertrand, Pranav Ghorpade, Sasha Rubin, Bernhard Scholz, and Pavle Subotić. 2025. Reusable formal verification of dag-based consensus protocols. In *NASA Formal Methods Symposium*. Springer, 138–158.
[14] Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, et al. 2024. Graph of thoughts: Solving elaborate problems with large language models. In *AAAI*, Vol. 38. 17682–17690.
[15] Arslan Bisharat, Brian Ortiz, Eric Spencer, Khushboo Bhadauria, TaiNing Wang, George K Thiruvathukal, Konstantin Laufer, and Mohammed Abuhamad. 2026. Can LLMs Write Correct TLA+ Specifications? Evaluating Natural-Language-to-TLA+ Generation. *arXiv preprint arXiv:2606.05792* (2026).
[16] Gabriel Bracha. 1984. An asynchronous [(n-1)/3]-resilient consensus protocol. In *PODC*. ACM, 154–162.
[17] Gabriel Bracha. 1987. Asynchronous Byzantine agreement protocols. *Information and Computation* 75, 2 (1987), 130–143.
[18] Gabriel Bracha and Sam Toueg. 1985. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)* 32, 4 (1985), 824–840.
[19] Sean Braithwaite, Ethan Buchman, Igor Konnov, Zarko Milosevic, Ilina Stoilkovska, Josef Widder, and Anca Zamfir. 2020. Formal specification and model checking of the tendermint blockchain synchronization protocol (short paper). In *FMBC*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 10–1.
[20] Mike Burrows. 2006. The Chubby lock service for loosely-coupled distributed systems. In *OSDI*. USENIX Association, 335–350.
[21] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. 2011. *Introduction to reliable and secure distributed programming*. Springer Science & Business Media.
[22] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. 2001. Secure and efficient asynchronous broadcast protocols. In *Annual International Cryptology Conference*. Springer, 524–541.
[23] Christian Cachin and Stefano Tessaro. 2005. Asynchronous verifiable information dispersal. In *SRDS*. IEEE, 191–201.
[24] Christian Cachin and Marko Vukolic. 2017. Blockchain Consensus Protocols in the Wild. In *DISC*.
[25] Jialun Cao, Yaojie Lu, Meiziniu Li, Haoyang Ma, Haokun Li, Mengda He, Cheng Wen, Le Sun, Hongyu Zhang, Shengchao Qin, et al. 2025. From informal to formal–incorporating and evaluating llms on natural language requirements to verifiable formal proofs. In *ACL*. 26984–27003.
[26] Miguel Castro and Barbara Liskov. 2002. Practical Byzantine fault tolerance and proactive recovery. *TOCS* 20, 4 (2002), 398–461.
[27] Benjamin Y. Chan and Elaine Shi. 2020. Streamlet: Textbook Streamlined Blockchains. In *AFT*. 1–11.
[28] Tushar Deepak Chandra and Sam Toueg. 1996. Unreliable Failure Detectors for Reliable Distributed Systems. *J. ACM* 43, 2 (1996).
[29] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
[30] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024. Teaching large language models to self-debug. In *ICLR*, Vol. 2024. 8746–8825.
[31] Qian Cheng, Saad Mohammad Rafid Pial, Ruize Tang, Yiming Su, Emilie Ma, Finn Hackett, Ivan Beschastnikh, Yu Huang, and Tianyin Xu. 2026. Specula: Scaling formal specifications for autonomous model checking of system code. *arXiv preprint arXiv:2607.25333* (2026).
[32] Qian Cheng, Ruize Tang, Emilie Ma, Finn Hackett, Peiyang He, Yiming Su, Ivan Beschastnikh, Yu Huang, Xiaoxing Ma, and Tianyin Xu. 2025. SysMoBench: Evaluating AI on Formally Modeling Complex Real-World Systems. *arXiv preprint arXiv:2509.23130* (2025).
[33] Sourav Das, Zhuolun Xiang, and Ling Ren. 2021. Asynchronous Data Dissemination and Its Applications. In *CCS*.
[34] A Davis, Max Hirschhorn, and Judah Schvimer. 2020. Extreme modelling in practice. *arXiv preprint arXiv:2006.00915* (2020).
[35] Sisi Duan, Xin Wang, and Haibin Zhang. 2023. Practical Signature-Free Asynchronous Common Subset in Constant Time. In *CCS*.

[36] Stewart Grant, Hendrik Cech, and Ivan Beschastnikh. 2018. Inferring and asserting distributed system invariants. In *ICSE*. 1149–1159.
[37] Matthias Grundmann and Hannes Hartenstein. 2025. Security of the Lightning Network: Model Checking a Stepwise Refinement with TLA+. In *iFM*. 313–335.
[38] Rachie Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. 2015. The next 700 bft protocols. *TOCS* 32, 4 (2015), 12:1–12:45.
[39] Samuel Holt, Max Ruiz Luyten, and Mihaela van der Schaar. 2024. L2mac: Large language model automatic computer for extensive code generation. In *ICLR*, Vol. 2024. 36762–36822.
[40] Heidi Howard, Markus A Kuppe, Edward Ashton, Amaury Chamayou, and Natacha Crooks. 2025. Smart Casual Verification of the Confidential Consortium Framework. In *NSDI*. 259–276.
[41] Jinhao Jiang, Kun Zhou, Zican Dong, Keming Ye, Wayne Xin Zhao, and Ji-Rong Wen. 2023. Structgpt: A general framework for large language model to reason over structured data. In *EMNLP*. 9237–9251.
[42] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. 2011. Zab: High-performance broadcast for primary-backup systems. In *DSN*. IEEE, 245–256.
[43] Ramakrishna Kolta, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2009. Zyzzyva: speculative Byzantine fault tolerance. *TOCS* (2009).
[44] Markus A Kuppe. 2023. Teaching TLA+ to Engineers at Microsoft. In *Formal Methods Teaching Workshop*. Springer, 66–81.
[45] Leslie Lamport. 1994. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16, 3 (1994), 872–923.
[46] Leslie Lamport. 1999. Specifying concurrent systems with TLA+. *Calculational System Design* (1999), 183–247.
[47] Leslie Lamport. 2026. A high-level view of TLA+. https://lamport.azurewebsites.net/tla/high-level-view.html. Accessed: 2026-07.
[48] Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine generals problem. *TOPLAS* 4, 3 (1982), 382–401.
[49] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F Lukman, and Haryadi S Gunawi. 2014. SAMC: Semantic-Aware model checking for fast discovery of deep bugs in cloud systems. In *OSDI*. 399–414.
[50] Elaine Li, Traian Serbănuță, Denisa Diaconescu, Vlad Zamfir, and Grigore Rosu. 2020. Formalizing correct-by-construction casper in coq. In *ICBC*. IEEE, 1–3.
[51] Rujia Li, Jingyuan Ding, Qin Wang, Keting Jia, Haibin Zhang, and Sisi Duan. 2025. Does finality gadget finalize your block? a case study of binance consensus. In *USENIX Security*. 4109–4125.
[52] Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2023. Let's verify step by step. In *ICLR*.
[53] Fang Liu, Yang Liu, Lin Shi, Zhen Yang, Li Zhang, Xiaoli Lian, Zhongqi Li, and Yuchi Ma. 2026. Beyond functional correctness: Exploring hallucinations in llm-generated code. *IEEE Transactions on Software Engineering* (2026).
[54] Thomas Locher and Victor Shoup. 2025. MiniCast: Minimizing the communication complexity of reliable broadcast. In *Eurocrypt*. 96–115.
[55] Donghang Lu, Thomas Yurek, Samarth Kulshreshtha, Rahul Govind, Aniket Kate, and Andrew Miller. 2019. Honeybadgermpc and asynchromix: Practical asynchronous mpc and its application to anonymous communication. In *CCS*. 887–903.
[56] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The honey badger of BFT protocols. In *CCS*. 31–42.
[57] Achour Mostefaoui, Hamouma Moumen, and Michel Raynal. 2014. Signature-free asynchronous byzantine consensus with t< n/3 and o (n 2) messages. In *PODC*. ACM, 2–9.
[58] Kartik Nayak, Ling Ren, Elaine Shi, Nitin H. Vaidya, and Zhuolun Xiang. 2020. Improved Extension Protocols for Byzantine Broadcast and Agreement. In *DISC*.
[59] Joachim Neu, Ertem Nusret Tas, and David Tse. 2021. Ebb-and-flow protocols: A resolution of the availability-finality dilemma. In *SP*. 446–465.
[60] Joachim Neu, Ertem Nusret Tas, and David Tse. 2022. Two more attacks on proof-of-stake GHOST/Ethereum. In *ConsensusDay*. 43–52.
[61] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. 2015. How Amazon web services uses formal methods. *Commun. ACM* 58, 4 (2015), 66–73.
[62] Zhi Niu, Luming Dong, Yong Zhu, and Li Chen. 2022. Verifying zookeeper based on model-based runtime trace-checking using TLA+. In *ICCSIE*. 13–18.
[63] Diego Ongaro. 2015. Email to raft-dev mailing list, subject line: bug in single-server membership changes. https://groups.google.com/g/raft-dev/c/t4xj6dJTP6E. Accessed: 2026-07.
[64] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *ATC*. 305–319.
[65] Lingzhi Ouyang, Yu Huang, Binyu Huang, and Xiaoxing Ma. 2023. Leveraging TLA+ Specifications to Improve the Reliability of the ZooKeeperCoordination Service. In *SETTA*. 189–205.
[66] Lingzhi Ouyang, Xudong Sun, Ruize Tang, Yu Huang, Madhav Jivrajani, Xiaoxing Ma, and Tianyin Xu. 2025. Multi-Grained Specifications for Distributed System Model Checking and Verification. In *Eurosys*. 379–395.
[67] Laurent Perrussel and Bernard Rothenburger. 1997. Relationships Checking. *Information Modelling and Knowledge Bases VIII* 8 (1997), 1.
[68] Irving S Reed and Gustave Solomon. 1960. Polynomial codes over certain finite fields. *J. Soc. Ind. Appl. Math.* 8, 2 (1960), 300–304.
[69] Michael K Reiter. 1994. Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *CCS*. 68–80.
[70] Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. 2024. Code-aware prompting: A study of coverage-guided test generation in regression setting using llm. *Proceedings of the ACM on Software Engineering* 1 (2024), 951–971.
[71] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems* 36 (2023), 8634–8652.
[72] Nibesh Shrestha, Qianyu Yu, Aniket Kate, Giuliano Losa, Kartik Nayak, and Xuechao Wang. 2025. Optimistic, Signature-Free Reliable Broadcast and Its Applications. In *CCS*.
[73] Joseph Spracklen, Raveen Wijewickrama, AHM Nazmus Sakib, Anindya Maiti, and Bimal Viswanath. 2025. We have a package for you! a comprehensive analysis of package hallucinations by code generating LLMs. In *USENIX Security*. 3687–3706.
[74] TK Srikanth and Sam Toueg. 1987. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing* 2, 2 (1987), 80–94.
[75] Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. 2024. Clover: Closed-loop verifiable code generation. In *International Symposium on AI Verification*. Springer, 134–155.
[76] TLA+ Community. 2026. TLA+ Examples: A Collection of TLA+ Specifications. https://github.com/tlaplus/Examples. Accessed: 2026-07.
[77] TLA+ Foundation. 2026. TLAi+Bench. https://github.com/tlaplus/TLAiBench. Accessed: 2026-07.
[78] Dong Wang, Wensheng Dou, Yu Gao, Chenao Wu, Jun Wei, and Tao Huang. 2023. Model checking guided testing for distributed systems. In *Eurosys*. 127–143.
[79] Jianxun Wang and Yixiang Chen. 2023. A review on code generation with llms: Application and evaluation. In *MedAI*. IEEE, 284–289.
[80] Yanlin Wang, Yanli Wang, Daya Guo, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. 2024. Rlcoder: Reinforcement learning for repository-level code completion. *arXiv preprint arXiv:2407.19487* (2024).
[81] IOTA Foundation / Wasp. 2026. Bracha Reliable Broadcast in TLA+. https://github.com/iotaledger/wasp/blob/develop/packages/gpa/rbc/bracha/BrachaRBC.tla. Accessed: 2026-07.
[82] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, et al. 2022. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682* (2022).
[83] Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Jianan Yao, Weidong Cui, Yeyun Gong, Chris Hawblitzel, Shuvendu Lahiri, Jacob R Lorch, Shuai Lu, et al. 2025. Autoverus: Automated proof generation for rust code. *PACMPL* (2025).
[84] Kaiyu Yang, Gabriel Poesia, Jingxuan He, Wenda Li, Kristin Lauter, Swarat Chaudhuri, and Dawn Song. 2024. Formal mathematical reasoning: A new frontier in ai. *arXiv preprint arXiv:2412.16075* (2024).
[85] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *NeurIPS* 36 (2023), 11809–11822.
[86] Jia-Qi Yin, Hui-Biao Zhu, and Yuan Fei. 2020. Specification and verification of the Zab protocol with TLA+. *JCST* 35, 6 (2020), 1312–1323.
[87] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT consensus with linearity and responsiveness. In *PODC*.
[88] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation. In *EMNLP*.
[89] Haibin Zhang and Sisi Duan. 2022. PACE: Fully Parallelizable BFT from Reproposable Byzantine Agreement. In *CCS*.
[90] Haibin Zhang, Sisi Duan, Boxin Zhao, and Liehuang Zhu. 2023. WaterBear: Practical Asynchronous BFT Matching Security Guarantees of Partially Synchronous BFT. In *Usenix Security*.
[91] Yuhao Zhou and Stavros Tripakis. 2026. Towards language model guided TLA+ proof automation. In *FM*. Springer, 619–640.
[92] Jian Zuo, Junzhe Liu, Xianyong Wang, Yicheng Liu, Navya Goli, Tong Xu, Hao Zhang, Umamaheswara Rao Tida, Zhenge Jia, and Mengying Zhao. 2025. ComplexVCoder: An LLM-Driven Framework for Systematic Generation of Complex Verilog Code. *arXiv preprint arXiv:2504.20653* (2025).

A Open Science

This research is committed to the principles of open science, ensuring transparency and accessibility throughout the research process. Due to the double-blind evaluation, the artifact is currently made available at https://github.com/cshuhe/TLAssist. We are committed to the following principle:

- **Code availability**: Our codebase will be made publicly available via a stable URL. Our codebase consists of all the LLM prompts, the scripts of running `TLAssist`, and all the generated TLA$^+$ specifications mentioned in this paper. We will provide detailed guidelines for other researchers to replicate our experiments and validate the results.
- **Transparent methodologies**: Our publications will include comprehensive descriptions of methodologies, covering algorithms and experimental setups. This transparency allows other researchers to fully understand our methods and extend our work.
- **Preprints and open access**: To enhance the dissemination of our research, preprints of our paper will be made available on platforms such as arXiv or the IACR Cryptology ePrint Archive before formal publication.

B Ethical Considerations

Stakeholder identification and impacts. We identify the direct stakeholders as the researchers and developers who design and implement Byzantine reliable broadcast (RBC)-related protocols, and authors of the open-source TLA$^+$ repositories used as comparison. The indirect stakeholders include the end-users who interact with systems that use RBC-related protocols as the underlying building blocks, and also the broader security community.

Research team and methodology. Our work does not involve any human subjects or personally identifiable information (PII). To build `TLAssist`, we use the TLA$^+$ specifications of the official repository by TLA$^+$ foundation. To conduct a comparative analysis, we use publicly available TLA$^+$ specifications hosted on GitHub as a comparison. We acknowledge the intellectual property and effort of the authors of these open-source repositories and use their TLA$^+$ specifications for benchmarking and validation purposes. We have cited the links to all repositories when mentioned in this paper.

Benefits to stakeholders. RBC is one of the most fundamental problems in distributed consensus, and consensus protocols are known to be extremely difficult to prove and implement. The primary benefit of our `TLAssist` tool is to assist protocol designers in improving the design process and facilitate understanding of the protocols and proofs for beginners and developers.

Potential harms and mitigation. The generated scripts of our `TLAssist` tool can be used to find design issues. As one example, we find the totality violation of a known RBC protocol. We have contacted the first author of the paper to confirm the issue. We do not foresee significant risks of identifying design issues, and on the contrary, find the issues beneficial to the community.

The primary potential risk is that the TLA$^+$ specifications generated by `TLAssist` are assisted by large language models. Thus, the specifications generated without expert review might still *miss* important details of the protocol. Namely, design issues might still exist even if no errors are reported via model checking. We would like to emphasize that model checking cannot fully replace the security proofs of the corresponding protocols.

C Generative AI Usage

In developing our LLM-assisted tool for TLA$^+$ generation, we integrated ChatGPT, Kimi, Qwen, and Claude into the tool's internal workflow. We also use Gemini and Claude to assist implementation and polish a few paragraphs of the manuscript. The core scientific contributions, including all functional designs, case studies, and experimental logic, were developed entirely by the authors. All AI-assisted content and code drafts were strictly manually verified by the authors. All the generated TLA$^+$ specifications are reviewed by human.

D Additional Definitions and Pseudocode of the RBC Protocols

We show the pseudocode of the Authenticated Double-Echo Broadcast (ADB) in Fig. 15, as discussed in Sec. 5.3. We show the pseudocode of the (2, 3, 4)-Optimistic RBC protocol in Fig. 10, as discussed in Sec. 6.1. Meanwhile, we show the pseudocode of the CT RBC (optimized), ADD-based RBC, SigBRB and 2-round-BRB in Fig. 16, Fig. 18, Fig. 17, and Fig. 14 . The notations of some protocols are slightly changed to be consistent with the presentation of this paper.

- **Input.** T
- for $0 \le r \le f$
  - Wait till $|T| \ge 2f + r + 1$
  - Let $M := \mathsf{decode}(f + 1, r, T)$
  - Let $T' := \mathsf{encode}(M, m, f + 1)$
  - if $2f + 1$ fragments in T' match with T
    - return M

Figure 13: Information-theoretic online error correcting (OEC) protocol. The encode and decode algorithms are encoding and decoding algorithms of error correcting code such as Reed-Solomon [68].

Erasure coding and online error correction (OEC). For CT RBC, SigBRB, and ADD-based RBC, we will also need erasure coding and/or online error correction (OEC) code, so we discuss the definitions here.

An (p, n) erasure coding scheme over an alphabet $\mathcal{M}$ consists of two algorithms: $\mathsf{encode}() : \mathcal{M}^p \to \mathcal{M}^n$; $\mathsf{decode}() : \mathcal{M}^n \to \mathcal{M}^p$. The $[d_1, \cdots, d_n] \leftarrow \mathsf{encode}(v, n, p)$ algorithm takes as input a data block v, consisting of p data fragments, and outputs $n > p$ coded fragments. The $\mathsf{decode}(p, n, [\boldsymbol{b}])$ algorithm takes as input any p-size coded fragments and outputs the original data block containing n data fragments. In RBC protocols, p is often set as $f + 1$.

OEC was first introduced by Ben-Or, Canetti, and Goldreich [10]. The OEC protocol takes a set T tuples, where for each tuple (j, a_j), j is an index $j \in [n]$ and a_j is a fragment of a Reed-Solomon codeword [68]. The OEC protocol then tries to decode a message M such that Reed-Solomon encoding of M matches with at least $2f + 1$ elements in T. In the concrete construction, the protocol performs up to r trials of reconstruction, so up to $2f+r+1$ elements in T can be

used as input of the decoding algorithm. The OEC protocol is error-free and information-theoretically secure against any adversary that corrupts up to f fragments. We show the protocol in Fig. 13.

2-round-BRB with p_s as the sender

- **Propose**. The designated sender p_s with input m sends (PROPOSE, m) to all replicas.
- **Vote**. When receiving the first proposal (PROPOSE, m) from the sender, send a vote message for m to all replicas in the form of (VOTE, m)$_i$.
- **Commit**. When receiving $n-f$ signed vote messages for m,
  - forward these vote messages to all other replicas //forwarding step
  - *deliver* m and terminate.

Figure 14: 2-round-BRB protocol [6, Figure 1, Section 3]. The notions are slightly changed to be consistent with other protocols in this paper.

Authenticated Double-Echo Broadcast

Uses: AuthPerfectPointToPointLinks, instance al.

- **upon event** $\langle$ *brb, Init*$\rangle$ **do**
  - $sentecho$:= FALSE; $sentready$:= FALSE; $delivered$:= FALSE
  - $echos := [\perp]^n$; $readys := [\perp]^n$
- **upon event** $\langle$ *brb, Broadcast* $\mid m\rangle$ **do** // only process s
  - **forall** $q \in \Pi$ **do** trigger $\langle$ *al, Send* $\mid q, [\text{SEND}, m]\rangle$
- **upon event** $\langle$ *al, Deliver* $\mid p, [\text{SEND}, m]\rangle$ such that $p = s$ and $sentecho$ = FALSE **do**
  - $sentecho$:= TRUE
  - **forall** $q \in \Pi$ **do** trigger $\langle$ *al, Send* $\mid q, [\text{ECHO}, m]\rangle$
- **upon event** $\langle$ *al, Deliver* $\mid p, [\text{ECHO}, m]\rangle$ **do**
  - **if** $echos[p] = \perp$ **then** $echos[p] := m$
- **upon exists** $m \neq \perp$ such that $\#\{p \in \Pi \mid echos[p] = m\} > \frac{n+f}{2}$ and $sentready$ = FALSE **do**
  - $sentready$:= TRUE
  - **forall** $q \in \Pi$ **do** trigger $\langle$ *al, Send* $\mid q, [\text{READY}, m]\rangle$
- **upon event** $\langle$ *al, Deliver* $\mid p, [\text{READY}, m]\rangle$ **do**
  - **if** $readys[p] = \perp$ **then** $readys[p] := m$
- **upon exists** $m \neq \perp$ such that $\#\{p \in \Pi \mid readys[p] = m\} > f$ and $sentready$ = FALSE **do**
  - $sentready$:= TRUE
  - **forall** $q \in \Pi$ **do** trigger $\langle$ *al, Send* $\mid q, [\text{READY}, m]\rangle$
- **upon exists** $m \neq \perp$ such that $\#\{p \in \Pi \mid readys[p] = m\} > 2f$ and $delivered$ = FALSE **do**
  - $delivered$:= TRUE
  - trigger $\langle$ *brb, Deliver* $\mid s, m\rangle$

Figure 15: ADB protocol (which is equivalent with Bracha's broadcast) [21, Algorithm 3.18].

CT RBC (optimized) with p_s as the sender

- As the sender p_s, **upon** input(v):
  - let $\{s_j\}_{j\in[n]}$ be the blocks of an $(n-2f, n)$-erasure coding scheme applied to v
  - let h be a Merkle tree root computed over $\{s_j\}$
  - send (VAL, h, b_j, s_j) to each replica p_j, where b_j is the jth Merkle tree branch
- **upon** receiving (VAL, h, b_i, s_i) from p_s,
  - send (ECHO, h, b_i, s_i) to all replicas
- **upon** receiving (ECHO, h, b_j, s_j) from replica p_j,
  - check that b_j is a valid Merkle branch for root h and leaf s_j, and otherwise discard
- **upon** receiving valid (ECHO, $h, \cdot, \cdot$) messages from $n-f$ distinct replicas,
  - interpolate $\{s'_j\}$ from any $n-2f$ leaves received
  - recompute Merkle root h' and if $h' \neq h$ then abort
  - if (READY, h) has not yet been sent, send (READY, h) to all replicas
- **upon** receiving $f+1$ matching (READY, h) messages, if READY has not yet been sent, send (READY, h) to all replicas
- **upon** receiving $2f+1$ matching (READY, h) messages, wait for $n-2f$ (ECHO) messages, then decode v

Figure 16: CT RBC (optimized) with p_s as the sender [56, Figure 2, Section 4].

SigBRB with identifier id and sender s

- **Initialization**:
  - $proof_1 \leftarrow \perp, msg \leftarrow \perp, val \leftarrow \perp, pset_1 \leftarrow \emptyset, T \leftarrow \emptyset$
- The sender sends (BCB-Send, M) to all replicas, where M is the input
- **upon** receiving (BCB-Send, M) from p_s
  - $msg \leftarrow M$ and send (BCB-Rep, M, σ_i) to p_s, where σ_i is a partial threshold signature
- **upon** receiving (BCB-Rep, M, σ_j) from p_j and $i = s$
  - Wait until $n - f$ valid partial signatures are received, combine them into a threshold signature σ
  - Send (BCB-Final, M, σ) to all replicas
- **upon** receiving (BCB-Final, M, σ) from p_s
  - If σ is a valid threshold signature on M and $M = msg$
    - $\boldsymbol{d} \leftarrow \mathsf{encode}(f + 1, n, msg)$
    - for $j \in \{1, \ldots, n\}$, send (Disperse, d_j) to p_j
- **upon** receiving $f + 1$ matching (Disperse, d_i^*) **do**
  - fix d_i^* and send (Reconstruct, d_i^*) to all replicas
- **upon** receiving (Reconstruct, d_j) from p_j
  - add d_j to T
  - **for** $0 \le r \le f$ **do**
    - **wait until** $|T| \ge 2f + r + 1$
    - **if** $\mathsf{decode}(f + 1, T, r) = M$ **then**
      - $val \leftarrow M$, send (Ready, i) to all replicas
      - **if** (Disperse) has not been sent
        - $\boldsymbol{d} \leftarrow \mathsf{encode}(f + 1, n, M)$
        - **for** $j \in \{1, \ldots, n\}$ //amplification
          - send (Disperse, d_j) to p_j
- **upon** receiving $n - f$ (Ready) messages
  - **if** $val \neq \perp$, *deliver val*

Figure 17: SigBRB protocol [7, Algorithm 4, Section 5].

ADD-based RBC with p_s as the sender

- **input** m //only for p_s node
- send (Propose, m) to all
- **input** $P(\cdot)$ //for all nodes, predicate $P(\cdot)$ returns true unless otherwise specified
- **upon** receiving (Propose, m) from the broadcaster **do**
  - **if** $P(m)$ then
    - let $h := \mathrm{hash}(m)$
    - send (Echo, h) to all
- **upon** receiving $2f + 1$ (Echo, h) messages and not having sent a READY message **do**
  - send (Ready, h) to all
- **upon** receiving $f + 1$ (Ready, h) messages and not having sent a READY message **do**
  - send (Ready, h) to all
- **upon** receiving $2f + 1$ (Ready, h) messages **do**
  - **if** received (Propose, m) and $h = \mathrm{hash}(m)$ then
    - ADD (m)
  - **else**
    - ADD ($\perp$)

Figure 18: ADD-based RBC [33, Algorithm 3, Section 4]. The ADD () function is the same as the (Disperse) and (Reconstruct) phases of SigBRB, and we omit the details here.

```
State 11: SendEcho                                          //Action name
∧ delivered = (p1 :> "" @@ p2 :> "" @@ p3 :> "")
∧ receivedE =
   p1 :> (v1 :> {p1, p2} @@ v2 :> {p3}) @@
   p2 :> (v1 :> {p4, p1, p2} @@ v2 :> {p3}) @@
   p3 :> (v1 :> {p1, p2} @@ v2 :> {p3})
∧ receivedR =
   p1 :> (v1 :> {p2} @@ v2 :> {}) @@
   p2 :> (v1 :> {p2} @@ v2 :> {}) @@
   p3 :> (v1 :> {p2} @@ v2 :> {})
∧ receivedS = (p1 :> v1 @@ p2 :> v1 @@ p3 :> v2)
∧ sender = p1
```

Figure 19: Example of a TLA$^+$ error trace. The line highlighted in brown denotes that no correct replicas *deliver* any value.

E Examples and Prompts of LLM generation

E.1 An Example of the Feedback Loop

As mentioned in Sec. 5.2, in our feedback loop, a TLC error might be caused by the fact that the TLA$^+$ script is not properly written. The error trace is sent back to LLM to refine its generated script.

We show an example in Fig. 19, where the validity was violated due to improper TLA$^+$ specification. Namely, by printing out the delivered messages, it is shown in the error trace that none of the correct replicas has *delivered* any value. The error is caused by the fact that TLA$^+$ script does not model the *receivedS* variable properly, a variable that models the received values of the replicas. This error trace is then appended to the subsequent prompts as feedback for the feedback loop.

E.2 Prompts

We show prompts of LLM generation for different stages of `TLAssist`. As mentioned in Sec. 5.1, we first transform the target pseudocode into a structured template (pseudocode) via LLM, as shown below.

built_prompt = (

"The content of 'draft.md' file is in variable: 'draft_content'"

"The target pseudocode is in variable: 'pseudocode'"

"Please follow the mapping rules between target pseudocode and JSON template in 'draft.md' file, and convert the pseudocode into a sensible JSON format that satisfies the following requirements: "

" Do NOT model an explicit PROPOSE/SEND action. Instead, receivedS is a mapping from processes to V "

" For SendEcho (or equivalent) action, use only the guard 'receivedS[self] ∈ V'. Do NOT add conditions like Cardinality(Sends(...))."

" If the model emits any Sends-related helpers/actions, remove them."

" 'Constants' and 'Variables' can be partially removed or added according to the requirements of the pseudocode. "

" 'Actions' structures should follow the draft.md style; you may add or omit sub-items based on the pseudocode (do not omit 'Next' and 'Spec') "

" Do NOT add n or f into 'Constants'. Use derived expressions $n == Cardinality(Correct \cup Faulty)$ and $f == Cardinality(Faulty)$ in TLA$^+$, so omit them from 'Constants'."

" Do NOT invent helpers/fields not in the pseudocode (e.g. 'EchoAuthors')."

" Byzantine action: model faulty processes as able to inject only the message types appearing in pseudocode (but do not model SEND/PROPOSE messages)."

" Do not model message-sending state with boolean variables (e.g., sentReady). Use set-based conditions instead, inferring sends from $self \notin receivedR[self][v]$ or similar predicates."

" If SymbolOf is used, it must be interpreted exactly as follows: it abstracts encoded message fragments and is instantiated as a total function from processes to fragments with a fixed one-to-one correspondence; moreover, both the description and the instance of SymbolOf must be copied verbatim from the JSON file, with no rewriting or abbreviation allowed."

" When reasoning about the condition of each action in 'Actions', the pseudocode must be interpreted faithfully, and variables must not be introduced or assigned arbitrarily."

" Copy the 'Invariants' and 'Properties' sections from the input JSON verbatim into the output JSON. Do not modify names, descriptions, or 'spec_instance' fields—preserve them exactly."

" CRITICAL BYZANTINE EQUIVOCATION LOGIC: Read the PROPOSE/ECHO behavior in the pseudocode carefully. IF the pseudocode is Authenticated Echo Broadcast(AEB) or Authenticated Double-Echo Broadcast(ADB) or Optimistic reliable broadcast, then ignore this BYZANTINE EQUIVOCATION LOGIC and remain consistent with 'draft.md'."

"MODE A (Default): IF the pseudocode explicitly checks "and an (Echo) message has not been sent" (or similar guards preventing multiple echos), use single-value mapping: 'receivedS' type is Correct -> $V \cup \{""\}$.'SendEcho' condition uses $receivedS[self] \in V$."

"MODE B (Set-based): IF the pseudocode does NOT have a "has not been sent" guard (e.g., "upon receiving <PROPOSE, M> ... do send <ECHO, M>"), you MUST model the Byzantine sender sending multiple different values to the same process. Apply these updates:'receivedS' type: Correct -> SUBSET V;'receivedS' init: If sender $\in$ Correct, receivedS maps all correct processes to a singleton set {CHOOSE $v \in V$: TRUE}; otherwise unconstrained mapped to SUBSET V;'SendEcho' condition: Must extract the value using $\exists\ val \in receivedS[self]$ and MUST not add the guard $\forall\ v \in V : self \notin receivedE[self][v]$;'SendEcho' effect: Use the extracted 'val' instead of 'receivedS[self]';'Validity' property: Must safely extract the single value if the sender is correct: LET Val == IF sender $\in$ Correct THEN CHOOSE $v \in V : v \in receivedS[sender]$ ELSE " "."

" Do not add self = sender as a condition for SendDisperse in SigBRB; the DISPERSE round can be performed by any correct node. And Do not forget ByzantineFinal action."

"The Authenticated Echo Broadcast pseudocode should not include the Totality property in 'Properties', since AEB is not required to satisfy totality. The Authenticated Double-Echo Broadcast pseudocode should include the Totality property. The overall structure of both pseudocode specifications should remain consistent with 'draft.md'. DO NOT use 'echos' and 'readys'."

" Please produce an initial JSON draft that follows the above. Return ONLY the JSON (inside a JSON block is preferred). "

)

After the structured template (pseudocode) is generated, the template and the structured template (knowledge base) are used as prompt for LLM to generate the draft TLA$^+$ specification. We show our prompt as follows.

Please generate a complete TLA$^+$ specification, including the configuration section, based on the following inputs:
=== PSEUDOCODE ===
{pseudo_code}

=== JSON CONFIG ===
{json_config}

Generation requirements:
1. Replace the following symbols as indicated: change $\in$ to $\backslash in$, $\wedge$ to /\, $\exists$ to $\backslash E$, $\cup$ to $\backslash cup$, $\backslash setminus$ to \, / to $\backslash div$, $\backslash Diamond$ to <>, etc.
2. If the specification uses Cardinality and natural number operations, make sure to EXTENDS FiniteSets, Naturals, Sequences, TLC first.
3. Generate TLA$^+$ specification, do not generate PlusCal specification. The MODULE name must be "generated_spec".
4. "Constants" and "Variables" must strictly follow the JSON settings. Do not add or delete "Variables" and "Constants".
5. If the property in "Properties" has spec_instance, then it should be written strictly according to spec_instance. And do not use Placeholder.
6. Each part of the generated TLA$^+$ code should be strictly generated according to JSON format, especially the receivedS variable's init logic. Ensure the ELSE branch is truly unconstrained (a set of functions), not just hard-coded to empty strings.
7. The module body must include:
vars ==$\ll$ sender, receivedS, delivered, receivedE, receivedR $\gg$. The variable names in vars must match the Variables declared at the top of the module.
8. Append a single configuration block at the end of the file and use exactly the format below — do not duplicate it, do not add extra separators or comments.

———————- MODULE generated_spec ———————-
(Place your main TLA$^+$ module content here)
==
==== CONFIG ====
CONSTANTS
Correct = {{p1, p2, p3}}
Faulty = {{p4}}
V = {{v1, v2}}
(List all constant definitions here according to the JSON format, one definition per line)
SPECIFICATION Spec
INVARIANT TypeInv
INVARIANT Consistency
PROPERTY Validity
PROPERTY Totality
(INVARIANT and PROPERTY are set according to the json file; list each property on its own line)
==
Notes:
The configuration block must be unique and complete; it must include Spec, INVARIANT, and PROPERTY, and must appear strictly after the main module. The configuration block must start with "==== CONFIG ====", "CONSTANTS ...", etc., and must end with at least ten equals signs.

Each parameter in the configuration must appear one per line, named as in the example.
When generating the TLC configuration file (e.g., config.cfg), output all constant values using native TLA$^+$ syntax.
For the feedback loop for stage-II, we have another prompt, as shown below.

TLC inspection found the following {error_type} error.
{error_msg} error is approximately located on lines {start} to {end}, and the corresponding code snippet is shown below (arrows indicate the error line).
{code_snippet}
Please fix this part of the code based on the above information and return the complete ".tla" file content (retaining the original "====CONFIG===" configuration section).
Please provide the complete repaired TLA$^+$ code.

F Error Traces of TLC Verification

F.1 Error Trace of (2, 3, 4)-Optimistic RBC

We show error traces of the violation for (2, 3, 4)-Optimistic RBC first. As mentioned in Sec. 6.1, we set $n = 10, f = 3$ and replicas p_1, p_2, p_3 as Byzantine. To enhance clarity, we omit the redundant initial states and present only the critical traces that demonstrate a violation of the totality property below. We show the complete error traces in our artifacts.

```
State 28: StepCommit                                   //Error trace of state 28
∧ sender = p1
∧ delivered = (p4 :> m1 @@ p5 :> "" @@ p6 :> "" @@ p7 :> "" @@
  p8 :> "" @@ p9 :> "" @@ p10 :> "")
∧ receivedE =
  p1 :> (m1 :> {p4, p5, p6, p7} @@ m2 :> {p8, p9, p10}) @@
  p4 :> (m1 :> {p4, p5, p6, p7, p2, p3} @@ m2 :> {p8, p9, p10, p2, p3}) @@
  p5 :> (m1 :> {p4, p5, p6, p7, p2, p3} @@ m2 :> {p8, p9, p10, p2, p3}) @@
  p6 :> (m1 :> {p4, p5, p6, p7, p2, p3} @@ m2 :> {p8, p9, p10, p2, p3}) @@
  p7 :> (m1 :> {p4, p5, p6, p7, p2, p3} @@ m2 :> {p8, p9, p10, p2, p3}) @@
  p8 :> (m1 :> {p4, p5, p6, p7, p2, p3} @@ m2 :> {p8, p9, p10, p2, p3}) @@
  p9 :> (m1 :> {p4, p5, p6, p7, p2, p3} @@ m2 :> {p8, p9, p10, p2, p3}) @@
  p10 :> (m1 :> {p4, p5, p6, p7, p2, p3} @@ m2 :> {p8, p9, p10, p2, p3}) @@
  p2 :> (m1 :> {p4, p5, p6, p7} @@ m2 :> {p8, p9, p10}) @@
  p3 :> (m1 :> {p4, p5, p6, p7} @@ m2 :> {p8, p9, p10})
∧ receivedR =
  p1 :> (m1 :> {p4, p5, p7, p10} @@ m2 :> {p6, p8, p9}) @@
  p4 :> (m1 :> {p1, p4, p5, p7, p10, p2, p3} @@ m2 :> {p6, p8, p9}) @@
  p5 :> (m1 :> {p4, p5, p7, p10} @@ m2 :> {p6, p8, p9}) @@
  p6 :> (m1 :> {p4, p5, p7, p10} @@ m2 :> {p6, p8, p9}) @@
  p7 :> (m1 :> {p4, p5, p7, p10} @@ m2 :> {p6, p8, p9}) @@
  p8 :> (m1 :> {p4, p5, p7, p10} @@ m2 :> {p6, p8, p9}) @@
  p9 :> (m1 :> {p4, p5, p7, p10} @@ m2 :> {p6, p8, p9}) @@
  p10 :> (m1 :> {p4, p5, p7, p10} @@ m2 :> {p6, p8, p9}) @@
  p2 :> (m1 :> {p4, p5, p7, p10} @@ m2 :> {p6, p8, p9}) @@
  p3 :> (m1 :> {p4, p5, p7, p10} @@ m2 :> {p6, p8, p9})
∧ receivedS = (p1 :> m1 @@ p4 :> m1 @@ p5 :> m1 @@ p6 :> m1 @@ p7
:> m1 @@ p8 :> m2 @@ p9 :> m2 @@ p10 :> m2 @@ p2 :> m1 @@ p3 :>
m1)                    //Mapping of replica identities and their received (SEND)
∧ receivedV =
  p1 :> (m1 :> {p4} @@ m2 :> {p5, p6, p8, p9, p10}) @@
  p4 :> (m1 :> {p4} @@ m2 :> {p5, p6, p8, p9, p10, p2, p3}) @@
  p5 :> (m1 :> {p4} @@ m2 :> {p5, p6, p8, p9, p10, p2, p3}) @@
  p6 :> (m1 :> {p4} @@ m2 :> {p5, p6, p8, p9, p10, p2, p3}) @@
  p7 :> (m1 :> {p4} @@ m2 :> {p5, p6, p8, p9, p10, p2, p3}) @@
  p8 :> (m1 :> {p4} @@ m2 :> {p5, p6, p8, p9, p10, p2, p3}) @@
  p9 :> (m1 :> {p4} @@ m2 :> {p5, p6, p8, p9, p10, p2, p3}) @@
  p10 :> (m1 :> {p4} @@ m2 :> {p5, p6, p8, p9, p10, p2, p3}) @@
  p2 :> (m1 :> {p4} @@ m2 :> {p5, p6, p8, p9, p10}) @@
  p3 :> (m1 :> {p4} @@ m2 :> {p5, p6, p8, p9, p10})
```

F.2 Error Trace of SigBRB (w/o amplification) and 2-round-BRB (w/o forwarding)

We show error traces of the correctness violation for the modified SigBRB (w/o amplification). Note that SigBRB does not suffer from any issues. We modify SigBRB for one of our case studies to showcase the use of `TLAssist`. As mentioned in Sec. 6.2, the modified SigBRB protocol suffers from the violation of totality. We set $n = 4, f = 1$ and replica p_1 as Byzantine. We show the complete error traces in our artifacts and present the traces that cause the violation below.

```
State 13: Decide                                       //Error trace of state 13
∧ sender = p1
∧ mStar = (p1 :> "" @@ p2 :> s2 @@ p3 :> "" @@ p4 :> s4)
∧ decodeRound = (p1 :> 0 @@ p2 :> 0 @@ p3 :> 0 @@ p4 :> 0)
∧ delivered = (p2 :> "" @@ p3 :> "" @@ p4 :> "v1")
∧ receivedDis =
  p1 :> (s1 :> {p2} @@ s2 :> {} @@ s3 :> {} @@ s4 :> {}) @@
  p2 :> (s1 :> {} @@ s2 :> {p2} @@ s3 :> {} @@ s4 :> {}) @@
  p3 :> (s1 :> {} @@ s2 :> {} @@ s3 :> {p2} @@ s4 :> {}) @@
  p4 :> (s1 :> {p1} @@ s2 :> {} @@ s3 :> {} @@ s4 :> {p1,p2})
∧ receivedFinal =
  p2 :> (v1 :> TRUE ) @@
  p3 :> (v1 :> FALSE ) @@
  p4 :> (v1 :> FALSE )
∧ receivedReady = (p1 :> {p2, p4} @@ p2 :> {p2, p4} @@ p3 :> {p2, p4} @@
p4 :> {p1, p2, p4})
∧ receivedS = (p2 :> v1 @@ p3 :> v1 @@ p4 :> v1)
∧ receivedRec =
  p1 :> {≪ p2, s2 ≫, ≪ p4, s4 ≫} @@
  p2 :> {≪ p1, s1 ≫, ≪ p2, s2 ≫, ≪ p4, s4 ≫} @@
  p3 :> { ≪ p2, s2 ≫, ≪ p4, s4 ≫} @@
  p4 :> {≪ p1, s1 ≫, ≪ p2, s2 ≫, ≪ p4, s4 ≫}
∧ receivedRep = (p2 :> (v1 :> {})) @@(p3 :> (v1 :> {})) @@ (p4 :> (v1 :> {}))
∧ val = (p1 :> "" @@ p2 :> v1 @@ p3 :> "" @@ p4 :> v1)
```

Explaining the error trace. As mentioned in Sec. 6.2, if we remove the amplification step of SigBRB, there is a violation of the totality. we now discuss the error trace in the case with $n = 4, f = 1$. In this script, p_1 is a Byzantine sender, p_1 acts honestly before it sends the (BCB-Final) message. Namely, after it collects valid signatures, it sends (BCB-Final) message only to p_2. According to the protocol, p_2 sends (Disperse) to all replicas. In the rest of the phases, p_1 only sends valid (Disperse) and (Reconstruct) messages to all correct replicas except p_3. By the protocol, p_4 can collect $n - f$

data fragments and *deliver* the corresponding message. However, without a message from p_1, p_3 is unable to collect enough data fragments to reconstruct the message. The totality property is thus violated. In the error trace above, the line highlighted in brown show that p_3 cannot *deliver*.

We also show error traces of the violation of totality for 2-round-BRB (w/o forwarding). Note that 2-round-BRB itself is provably secure. In this error trace, $n = 4$, $f = 1$ and replica p_4 is the Byzantine sender. We show the complete error traces in our artifacts and present the state that causes the violation below.

```
State 11: StepCommit                                  //Error trace of state 11
∧ sender = p4
∧ delivered = (p1 :> v1 @@ p2 :> v1 @@ p3 :> "")
∧ receivedS = (p1 :> v1 @@ p2 :> v1 @@ p3 :> v2)
∧ receivedVote =
   p1 :> (v1 :> {p1, p2, p4} @@ v2 :> {p3, p4}) @@
   p2 :> (v1 :> {p1, p2, p4} @@ v2 :> {p3, p4}) @@
   p3 :> (v1 :> {p1, p2} @@ v2 :> {p3, p4})
```

Explaining the error trace. The sender sends v_1 to p_1 and p_2, while sending v_2 to p_3. Then, p_1 and p_2 vote for v_1, whereas p_3 votes for v_2. The Byzantine sender p_4 sends its (Vote) on v_1 to p_1 and p_2, but not to p_3. Consequently, p_1 and p_2 can collect $n - f$ votes on v_1 from $\{p_1, p_2, p_4\}$ and *deliver* v_1. However, p_3 cannot collect $n - f$ matching votes and is unable to deliver. In the error trace above, the line highlighted in brown shows that p_3 cannot *deliver*.