Learning Provably Correct Synchronous Crash Fault Tolerant Distributed Protocols With Minimal Human Knowledge

Anonymous authors
Paper under double-blind review

Abstract

Provably correct distributed protocols, which are a critical component of modern distributed systems, are highly challenging to design and have often required decades of human effort. These protocols allow multiple agents to coordinate to come to a common agreement in an environment with uncertainty and failures. As a starting point, this work focuses on synchronous Crash Fault Tolerance (CFT) protocols, a widely used family of distributed protocols, in a bounded setting with a small number of agents. We formulate protocol design as a search problem over strategies in a game with imperfect information, and the desired correctness conditions are specified in Satisfiability Modulo Theories (SMT). However, standard methods for solving multi-agent games fail to learn correct protocols in this setting, even when the number of agents is small. We propose a learning framework, GGMS, which integrates a specialized variant of Monte Carlo Tree Search with a transformer-based action encoder, a global depth-first search to break out of local minima, and repeated feedback from a model checker. Protocols output by GGMS are verified correct via exhaustive model checking for all executions within the bounded setting. We further prove that, under certain assumptions, the search process is complete: if a correct protocol exists, GGMS will eventually find it. In experiments, we show that GGMS can learn correct protocols for larger settings than existing methods.

1 Introduction

Consider a coordination game where multiple agents must reach agreement on a shared decision, but each agent observes only its own local state and the messages it receives—some of which may be lost. An adversary controls which messages fail to arrive. The agents win if they all reach the same valid decision; they lose if any agent decides differently or violates a safety constraint. Crucially, this is not a game where high expected reward suffices—we need *guaranteed* correctness under worst-case adversarial play. A protocol that works 99.9% of the time is useless; a single counterexample renders it unusable.

This formulation captures the essence of *distributed protocol design*, a fundamental problem in building reliable systems. When you use a database, make a payment, or store a file in the cloud, distributed protocols ensure that multiple machines either all agree on what happened, or the operation safely aborts—even when machines crash or messages vanish. Designing these protocols has historically required decades of human ingenuity: The above formulation represents the consensus problem, a classic topic in distributed protocols, and consensus alone has motivated over 40 years of research spanning Paxos (Lamport, 1998; 2001; Moraru et al., 2013; Ongaro & Ousterhout, 2014) and Byzantine fault tolerance (Lamport et al., 1982; Castro & Liskov, 1999; Kotla et al., 2007; Giridharan et al., 2024).

A concrete example. In the *consensus* problem, three processes P1, P2, P3 must agree on a single value. Suppose P1 and P2 start proposing "0" while P3 proposes "1". In round one, each process broadcasts its proposal—but P3 crashes mid-broadcast, so P1 receives P3's message while P2 does not. Now P1 sees {0, 0, 1} while P2 sees {0, 0, ?}. Despite this asymmetry, a correct protocol must lead both to the same final decision. The FloodSet algorithm (Lynch, 1996) achieves this in $f + 1$ rounds (where f is the maximum failures): each process repeatedly broadcasts everything it has received, and in the final round decides on

the minimum value. The key insight is that at least one round must be failure-free, allowing all processes to synchronize (see §2.1).

Can we automate the discovery of such protocols? The game-theoretic structure—partial observability, adversarial uncertainty, hard safety constraints—suggests this problem might be amenable to the search-based learning that succeeded in games like Go and poker. We formulate protocol design as search over strategies in an imperfect-information game, where correctness properties are formally specified and exhaustively verified.

We pursue this through *zero-knowledge synthesis*: given only a specification of what constitutes a correct outcome, can a learning system discover protocols without human-designed examples? This methodology serves two purposes. First, it reveals what structure is *necessary* for correctness versus merely *conventional*—when our system independently discovers FloodSet-like protocols, this confirms the structure is dictated by the problem constraints rather than historical accident. Second, it provides a rigorous baseline: success validates that the search space is tractable, while the framework can naturally incorporate human knowledge (warm-starting, architectural constraints) when desired.

However, standard game-playing approaches fail here. In AlphaGo-style MCTS (Silver et al., 2017), self-play learns policies that win *in expectation*. Distributed protocols require something stronger: a policy that *never* loses against *any* failure pattern. Additionally, multiple correct protocols often exist, and naive learning can mix transitions from different protocols—a *superposition problem*—causing failures even when each protocol would individually be correct. These challenges defeat prior methods even for 3–4 agents.

We develop **Guided Global Monte Carlo Tree Search (GGMS)**, integrating three key ideas. (1) *Model checking as a hard oracle*: any candidate protocol is exhaustively verified against all possible executions (for the given process count). Violations produce counterexamples that feed back into training. This exhaustive verification—not the learning process—is what makes output protocols *provably correct*. (2) *Global depth-first search*: when learning produces ambiguous transitions (multiple outputs with similar probability), often due to the superposition problem mentioned above, GGMS freezes one choice and continues. If no correct protocol exists under current freezes, it backtracks systematically. This guarantees eventual convergence if a correct protocol exists. (3) *Guided sampling*: a phased curriculum starts with less ambiguous scenarios (failures only in later rounds, unambiguous initial states), allowing the effects of frozen transitions to propagate before relaxing to harder cases.

We make the following contributions. (1) We formulate distributed protocol design as search over state machines in an imperfect-information game with formal correctness specifications and exhaustive verification. (2) We propose GGMS, combining MCTS with a transformer encoder, global DFS, and iterative model-checking feedback. We prove that under mild assumptions, the search is complete: GGMS will not miss correct solutions (§4.2). (3) We demonstrate that GGMS learns correct protocols where standard MCTS fails, scaling to 4 processes with 3 failures.

Our evaluation shows GGMS achieves substantially higher success rates than MCTS baselines across all tested configurations. We discuss limitations including synchronous network assumptions and bounded process counts, and outline directions for relaxing these assumptions and integrating with LLM-based approaches.

2 Background

2.1 Distributed Protocols as State Machines

The distributed systems community models protocols using the *state machine approach*: each process is a state machine that takes its current state and incoming messages as input, and outputs a new state and messages to send. This enables both precise specification and formal verification.

Protocols vary along several dimensions. *Failure models* specify what can go wrong: in *crash failures*, a failed process simply stops responding; in *Byzantine failures*, a failed process may behave arbitrarily. *Timing models* specify synchrony assumptions: in *synchronous* networks, message delays and clock drift

are bounded; in *asynchronous* networks, no such bounds exist. This paper focuses on crash failures in synchronous networks—the simplest non-trivial setting—and discusses extensions in §6.

The FloodSet algorithm. We formalize the consensus example from §1. Recall the requirements: (1) every correct process eventually decides, (2) all decisions are identical, and (3) the decision must be some process's initial proposal.

FloodSet (Lynch, 1996) works in $f+1$ rounds, where f is the maximum number of processes that can crash. Each process p maintains a set W, initialized with p's own proposal. Each round, every process broadcasts W and updates $W := W \cup \bigcup_j Received_j$ where $Received_j$ is the message sent by process j, i.e., state W of process j. After $f+1$ rounds, each process decides $\min(W)$ (or any deterministic function on W).

Why does this work? Among $f+1$ rounds, at least one round has no failures (since at most f processes can fail total). In that round, all surviving processes receive identical messages and reach the same W. This W will not change in subsequent rounds, so $\min(W)$ remains consistent.

As a state machine: the state is W, the transition function unions received sets into W, and the final output applies $\min(\cdot)$.

2.2 Model Checking

Given a state machine and formal correctness properties, *model checking* verifies that no execution violates the properties. For a fixed number of processes N, this can be done by exhaustive enumeration of all possible initial states and failure patterns. More efficient approaches use SMT solvers (De Moura & Bjørner, 2008) and/or exploit symmetry (Leesatapornwongsa et al., 2014).

For example, the consensus agreement property (i.e., all decisions are identical) is formalized as:

$$\textbf{P1: } \neg\exists\, n, m \in \{1, 2, ..., N\}, f_n \in F_n,\ f_m \in F_m : \big(f_n = \texttt{decision:0} \wedge f_m = \texttt{decision:1}\big) \quad (1)$$

where F_n denotes the final decisions of process n and `decision:0/1` means that the process decides to accept proposal 0/1. Full property specifications appear in Appendix A.

Model checking provides correctness guarantees for a *concrete* N. Proving correctness for *arbitrary* N requires inductive formal verification (Hawblitzel et al., 2015; Ma et al., 2019; Yao et al., 2021; Zhang et al., 2025b)—an important direction we leave to future work.

3 Problem Formulation

We now formalize the synthesis problem. Our goal is to *learn* a state machine that satisfies correctness properties under all possible failure scenarios, given only a specification of what correct behavior means—not how to achieve it.

3.1 State Machine Model

Each process runs an identical deterministic state machine. The input at each round consists of: messages received from all processes (we model the state of the state machine as a message sent to itself), the round number, and optionally the process ID. The output is a new state, which is broadcast in the next round.

States fall into four categories:

- **Initial states** I: possible starting states (e.g., `init:0`, `init:1` for binary consensus)
- **Decision states** D: final outputs visible externally (e.g., `decision:0`, `decision:1`)
- **Lost state** L: a special symbol indicating a message was not received
- **Internal states**: intermediate states for coordination (e.g., `internal:a`, `internal:b`)

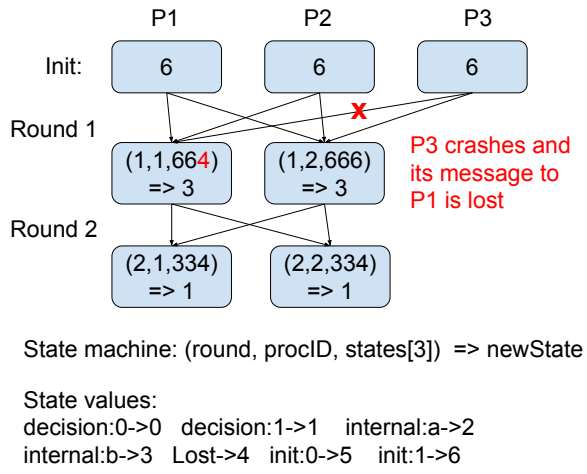

Figure 1: Simulating the FloodSet protocol. P3 crashes in Round 1 and causes P1 and P2 to have diverged inputs, but P1 and P2 still converge eventually following the protocol.

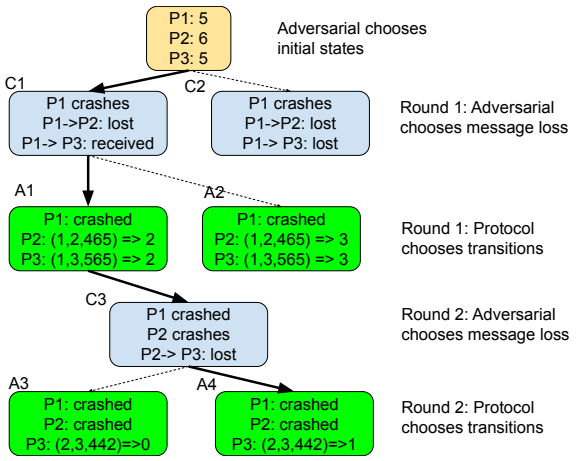

Figure 2: Applying MCTS. Adversarial chooses initial states and lost messages that are likely to violate correctness. Protocol chooses transitions that are likely to achieve correctness.

The human specifies only I, D, and correctness properties C. The learning system must discover how many internal states are needed and what transitions to use—analogous to AlphaGo Zero learning game strategies given only the rules.

3.2 Execution Model

Execution proceeds in synchronous rounds. Before round 1, each process receives an initial state. In each round: (1) every process broadcasts its current state to all others, (2) some messages may be lost due to crashes, and (3) each process applies its transition function to compute a new state. In the final round, processes transition to decision states.

Modeling crashes. We model crash failures through message loss: if process p crashes during round i, some recipients may receive p's round-i message while others receive L (lost). After crashing, all of p's subsequent messages are lost. This captures the partial-broadcast semantics that make consensus non-trivial.

Figure 1 illustrates an execution of the FloodSet algorithm, where P3 crashes mid-broadcast in round 1, causing P1 and P2 to observe different inputs—yet both still converge to the same decision.

3.3 Formal Problem Definition

Definition 3.1 (Protocol Specification)**.** A target protocol is a tuple $DP = (I, D, C)$: initial states I, decision states D, and correctness properties C.

Definition 3.2 (Setting)**.** A setting is a tuple $S = (n, r, f, k)$: number of processes n, rounds r, maximum failures f, and internal states k.

Definition 3.3 (Scenario)**.** A scenario $Sc = (Init, Loss)$ specifies initial states $Init[1..n]$ and message losses $Loss \subseteq \{(i, j, t) : \text{message from } i \text{ to } j \text{ lost in round } t\}$.

Definition 3.4 (State Machine)**.** A state machine SM is a transition function: $[round, procID, inputStates] \mapsto newState$. The state machine should provide a transition for every $[round, procID, inputStates]$, although in a certain state machine, some $[round, procID, inputStates]$ may never be reached.

Synthesis goal: Given DP and S, find SM such that for *all* scenarios Sc consistent with S, executing SM satisfies C. Sc is consistent with S if they have the same n and r and message losses in Sc can be caused by no more than f crashes as discussed in §3.2. The execution of SM is defined in §3.2.

A setting is **feasible** if such an SM exists. Since feasibility is unknown a priori, our approach starts with small settings and increases r or k if synthesis fails.

3.4 Assumptions and Limitations

Our model makes simplifying assumptions:

- Identical state machines: all processes run the same code, precluding Byzantine behavior
- Broadcast only: processes send identical messages to all others, precluding point-to-point protocols
- Synchronous timing: messages sent in round i arrive by round i's end
- Final-round decisions: processes decide only in the last round

These restrictions define a tractable starting point. We discuss relaxations in §6.

4 Learning a Distributed Protocol

4.1 Monte-Carlo Tree Search

Inspired by the similarity between distributed protocols and board games, we first apply the Monte Carlo Tree Search (MCTS) simulation approach.

We model the whole process as two players in a game. The protocol player tries to find the right transitions in its state machine so that all processes in the distributed protocol can always achieve the correctness properties. The adversarial player tries to find the scenarios that can defeat the protocol player's state machine, i.e. making it violate the correctness properties. While it may be possible to train a model for the adversarial player as well, our current implementation relies on random exploration for the adversarial player and relies on the model checker to make the final checking.

The whole simulation process works as shown in Figure 2. In one simulation, the adversarial player first selects the initial state for each process and messages to lose for the first round; our simulation follows the procedure in §3 to let each process broadcast its state, apply the message loss selected by the adversarial player, and compute the input for each process; then the protocol player selects the new state for each process based on its input. The simulation repeats this process until the maximum number of rounds is reached. By executing the simulation multiple times, allowing both players to explore different transitions, we can merge their results to build a search tree (Figure 2).

Our MCTS implementation is similar to that of AlphaGo-Zero, with the following differences. First, we do not use the value network in our MCTS. In AlphaGo-Zero, the value network is used to predict the expected reward, which guides the MCTS simulations. However, the number of rounds of the distributed protocol is much smaller than that in Go. This means we can simulate until the end of a protocol to get the real reward without relying on the value network.

Second, in AlphaGo-Zero, the MCTS algorithm is also used for inference, i.e., when actually playing the game or running the protocol. However, we can only use the trained policy network for inference because the internal information, such as visited count, rewards, and probabilities, is invisible between different processes during actual running. In other words, during inference, among multiple transitions from the same input, a process will always apply the one with the highest probability.

The details of our MCTS implementation are in Appendix C.5.

4.2 Ensuring Convergence with Global DFS

MCTS alone can occasionally converge to a correct state machine, but often fails to do so. Our investigation shows that the primary reason is the superposition problem, that is, there often exist multiple versions of correct state machines, and MCTS may end up in a situation where it learns some transitions from one version and some other transitions from another version, but when these transitions are combined, they do not generate a correct state machine.

We use the consensus protocol as an example to illustrate how superposition, that is, combining transitions from multiple correct versions of the protocol can occur and affect the convergence of MCTS. Keep in mind that consensus requires that 1) every process makes the same decision, and 2) if the initial input to every process is `init:0`, then the decision must be `decision:0`; if the initial input to every process is `init:1`, then the decision must be `decision:1`; if some processes has `init:0` as the input and some have `init:1`, then the decision could be either `decision:0` or `decision:1`.

In a protocol like Flood-Set, each process uses an internal state to record its intention, and multiple processes exchange their intentions to resolve divergence among processes. For example, if a process observes that all processes have `init:0` in the first round, it may change its internal state to `internal:a`, indicating that it intends to go to decision 0. Note that if a process observes some `init:0` and some `Lost`, it should transit to `internal:a` as well, since `Lost` may be from a process with `init:0`. Similarly, if a process observes `init:1` or `Lost`, but no `init:0`, then it can transit to `internal:b`, indicating that it intends to go to decision 1. However, if a process observes both `init:0` and `init:1`, it can transit to either `internal:a` or `internal:b`. Processes can exchange such intentions for multiple rounds until a consensus can be reached.

There are at least two reasons for the existence of multiple versions of the correct protocols. First, without human knowledge, the protocol may assign certain meanings to arbitrary internal states, creating multiple equivalent protocols. For example, while the above example uses `internal:a` for intention `decision:0` and `internal:b` for intention `decision:1`, we can swap this mapping to create an equivalent protocol. This creates a problem for MCTS. When MCTS simulates the scenario with all processes having `init:0` as input; it may find that it is feasible for a process to transit to `internal:a` in this case, and finally transit to `decision:0`. When MCTS simulates the scenario with all processes having `init:1` as input; it may also find that it is feasible for a process to transit to `internal:a` in this case, and finally transit to `decision:1`. Although the solution to each individual scenario is correct, combining them is incorrect, since we should not let the all `init:0` scenario and the all `init:1` scenario transit to the same internal state, as there is no way to distinguish them in the later rounds.

The second reason comes from the inherent ambiguity allowed by the protocol, that is, if some processes have `init:0` as input and some have `init:1`, then the decision could be either `decision:0` or `decision:1`. To give a concrete example about how this causes problems for MCTS, suppose that there are three processes participating in this protocol. Their input states are [`init:0`, `init:0`, `init:1`]. Suppose Process 0 crashes in the first round; its message is received by Process 1, but not Process 2 (Scenario 1). So Process 1's input is [`init:0`, `init:0`, `init:1`] (Input A) and Process'2 input is [Lost, `init:0`, `init:1`] (Input B). Due to the ambiguity of the initial input, we know that Input A and Input B can transit to either `internal:a` or `internal:b`, as long as they transit to the same internal state. If MCTS only simulates this scenario, it may find this constraint and thus give [$A \rightarrow$ `internal:a`] and [$B \rightarrow$ `internal:a`] a higher probability than [$A \rightarrow$ `internal:b`] and [$B \rightarrow$ `internal:b`]. However, Input A and Input B may appear separately in other scenarios as well. For example, if Process 0 does not crash in the above example, all processes will have Input A but no Input B (Scenario 2); if Process 0 crashes and both Process 1 and Process 2 miss its message, then both will have Input B but no Input A (Scenario 3). Since MCTS simulates each of these scenarios independently, it may end up preferring [$A \rightarrow$ `internal:a`] in Scenario 2 and preferring [$B \rightarrow$ `internal:b`] in Scenario 3. And when MCTS combines the probabilities from all scenarios, it may let A and B transit to different internal states. Again, in this example, MCTS finds a correct solution for each scenario, but it is incorrect to combine those solutions.

Our approach, Guided Global Monte Carlo Tree Search (GGMS), addresses this issue by enhancing MCTS with Depth-First Search (DFS). The key idea is that, if multiple transitions from the same input have similar probability, GGMS should try to freeze it to one transition (i.e., don't allow random exploration for this input

during MCTS) and then keep training to see whether it can get a correct state machine. In the above example, by freezing the transition [$A \rightarrow$ `internal:a`], we hope that keeping training will motivate [$B \rightarrow$ `internal:b`]. Since there might be multiple such ambiguity points, GGMS may repeat freezing multiple times. And since freezing may be wrong (e.g., freezing [$A \rightarrow$ `internal:a`] and [$B \rightarrow$ `internal:b`] for whatever reason), GGMS also needs to unfreeze certain transitions when it hits a dead end. This procedure leads to a DFS-like search, in which GGMS keeps freezing certain transitions until either it succeeds in finding a correct state machine or it hits a dead end. In the latter case, it unfreezes prior frozen transitions in an DFS manner.

Theorem 4.1 (Search Completeness)**.** *With a feasible setting, assuming GGMS's unfreezing condition is accurate (i.e., GGMS does not unfreeze when there exists a correct state machine including all frozen transitions), GGMS can eventually find a correct state machine.*

This is because, for a specific setting, the number of possible state machines is finite. Therefore, DFS can eventually explore all possible state machines, ensuring that we can find a correct one.

However, naive DFS (i.e., randomly freezing a transition) may take too long. Consider a protocol which involves two initial states, two internal states, two decision states, three processes, and three rounds, and assume that process ID does not affect transitions, the total number of possible inputs to the state machine is $3(round) \times 2(ownState) \times 3^2(otherState) = 54$. And since each of these inputs can transit to two values, the total number of state machines is 2^{54}. Using DFS to explore each state machine is too expensive. By combining DFS and MCTS, GGMS relies on DFS to break ambiguity, and relies on MCTS to provide hints about what transitions to freeze and "propagate" the effect of freezing (e.g., if we freeze [$A \rightarrow$ `internal:a`], then MCTS can find that [$B \rightarrow$ `internal:b`]).

We present the details of our DFS algorithm in §C.3, which includes the conditions for freezing and unfreezing and how to determine which transitions to freeze or unfreeze. Our current unfreezing condition, which is based on exhaustive search, is accurate but not scalable, and we discuss possible ways to replace exhaustive search with a more scalable Z3-based solver.

Note that in practice, there is always a time limit for training, so despite the eventual convergence guarantee, GGMS may not succeed within the time limit. However, this property means that we can always devote more time and/or resources to increase the chance of success.

4.3 Accelerating Convergence with Guided MCTS

While DFS helps to address the ambiguity issue, its speed is sometimes not satisfactory due to the following reasons. First, as discussed before, since GGMS can give a meaning to an arbitrary internal state, it needs multiple rounds of freezing to break the ambiguity among them. With more rounds in the protocol and more states, this process requires more rounds of freezing. To address this problem, at the beginning of training, GGMS freezes all transitions in one particular scenario (Sc), so that the simulation can reach a correct decision. Such a group freezing strategy helps GGMS to break the ambiguity among internal states more quickly. Furthermore, we can prove that, under certain conditions, it will not hurt the capability of GGMS to find a correct state machine.

Definition 4.2 (Definite scenario)**.** A scenario Sc consistent with setting $S = (n, r, f, k)$ and specification (I, D, C) is *definite* if, for every process i, there exists a decision value $d_i \in D$ such that every correct state machine for S assigns decision d_i to process i (assuming process i does not crash) when executed under Sc.

Theorem 4.3 (Group Freezing Preserves Completeness)**.** *In a feasible setting, assuming that 1) the scenario of this particular simulation is a definite scenario, and 2) for each pair of (round, procID), this approach only freezes one transition* $[(round, procID, inputA) \rightarrow B]$*, there exists a correct state machine compatible with all frozen transitions* ($Transitions_{fix}$)*.*

We provide the formal proof in Appendix B. Intuitively, if a correct state machine has $[(round, procID, inputA) \rightarrow C]$, we can always swap B and C for (round, procID) to get an equivalent protocol.

Second, as discussed before, we expect MCTS to propagate the effect of freezing. In the above example, when GGMS freezes [$A \rightarrow$ `internal:a`], it expects MCTS to find that it should choose [$B \rightarrow$ `internal:b`].

In practice, such propagation does not always succeed, since propagation often relies on a particular scenario to identify the relationship. In the above example, in order for GGMS to learn that A and B should transit to the same state, A and B should happen together in the same simulation. In other scenarios where only B occurs, GGMS may find that it is OK for B to transit to either `internal:a` or `internal:b`, due to the ambiguity problem discussed above. If simulations including both A and B happen rarely, but simulations including only B happen often, GGMS may not give $[B \rightarrow \texttt{internal:b}]$ a high probability. This problem is particularly troublesome when the random initialization of the state machine gives $[B \rightarrow \texttt{internal:b}]$ a high probability to begin with.

To address this, we introduce a guided sampling method. After freezing the initial path, GGMS simulates only those scenarios where message losses occur in the final round and initial states lead to a definite decision. After the model becomes fully correct in this stage, GGMS relaxes the restriction to allow message losses in the last two rounds. GGMS then repeats simulation and learning until the model achieves full correctness under this relaxed setting. GGMS repeats this until it allows message losses in all rounds. Finally, GGMS relaxes to allow all possible initial states.

Such a design is based on the observation that there is less ambiguity when the protocol is in later rounds and when the protocol starts with initial states that lead to definite decisions. By first simulating in these scenarios, GGMS can better avoid the "noise" from ambiguity, and thus better propagate the effects of freezing. Then, after such propagating has settled (i.e., the corresponding transitions reach a high probability), GGMS can further propagate its effects by relaxing crash and initial states scenarios. We present the details in Appendix C.1.

4.4 Validating the Learned Model

Exhaustive verification is what makes GGMS outputs *provably correct*. Our verifier enumerates all x^n initial state configurations (x is the number of initial states and n is the number of processes) and all message loss patterns consistent with at most f crash failures, checking that the learned state machine satisfies all safety properties. A protocol is returned only if it passes this complete check. Verification is not the bottleneck in practice, as MCTS simulation dominates running time; details appear in Appendix C.4. In the future, assuming MCTS will be optimized, we will switch to more efficient verifiers like Z3

GGMS integrates verification into a counterexample-guided loop: after each episode, the verifier either confirms correctness (triggering termination or relaxation per §4.3) or returns counterexamples whose sampling rate is increased in subsequent episodes (Appendix C.2).

5 Evaluation

We apply GGMS to learn synchronous atomic commit and consensus protocols. Atomic commit may look similar to consensus to some extent: Processes start with proposals "abort" or "commit" and try to reach the same decision. However, atomic commit is different from consensus in two ways: First, if a process proposes "abort", then the final decision of atomic commit must be "abort" (consensus allows "commit" if another process proposes "commit"). Second, if a process crashes, the final decision could be "abort", even if all processes propose "commit" (consensus requires "commit" in this case). Such differences lead to protocols that are subtly different from consensus. For example, we cannot simply run a consensus on the initial proposals, because if only one process proposes "abort" but fails to send it before it crashes, a consensus protocol will decide "commit" in the end, but this violates the requirement of atomic commit.

We document the formal properties of these protocols in Appendix A. Synchronous consensus is a well-studied field, with protocols such as FloodSet and Primary Backup (Bressoud & Schneider, 1996). Synchronous atomic commit is not well-studied as far as we know, because well-known atomic commit protocols, such as two-phase commit (2PC) (Gray, 1978), target asynchronous environment.

For synchronous consensus, human knowledge tells that a setting S with two internal states and $r > f$ (r is the number of rounds and f is the number of failures allowed) is feasible. We verified the same conclusion for synchronous atomic commit. We report results on such feasible settings. As a sanity check, we tested

	Success rate			Running time (minutes)											
	MCTS	MCTS+DFS	GGMS	MCTS				MCTS+DFS				GGMS			
				avg	min	max	std	avg	min	max	std	avg	min	max	std
ac-2-1	50%	60%	100%	133	15	301	107	52	25	96	25	15	8	29	9
ac-3-2	0	40%	60%	–	–	–	–	1413	722	1825	519	813	175	1441	560
ac-4-1	10%	20%	100%	184	184	184	–	192	191	193	1	413	310	630	109
ac-4-2	0	0	70%	–	–	–	–	–	–	–	–	754	563	938	132
con-2-1	90%	90%	100%	46	3	180	64	25	3	55	20	8	7	11	1
con-3-2	80%	80%	100%	1106	49	2387	891	273	148	693	181	118	104	151	16
con-4-1	80%	70%	100%	328	76	1576	508	348	146	654	214	268	253	280	9
con-4-2	30%	40%	90%	678	470	922	228	1783	1277	2560	553	717	610	970	144
con-4-3	0	0	100%	–	–	–	–	–	–	–	–	3358	1857	5693	1796

Table 1: Success rate and running time of different methods. We run each setting 10 times.

	Success rate			Running time (minutes)											
	GGMS	Crash-Early	Freeze-Closest	GGMS				Crash-Early				Freeze-Closest			
				avg	min	max	std	avg	min	max	std	avg	min	max	std
ac-3-2	60%	0	0	813	175	1441	560	–	–	–	–	–	–	–	–
con-3-2	100%	20%	100%	118	104	151	16	673	75	1272	847	129	109	219	34
con-4-2	90%	80%	100%	717	610	970	144	1072	400	2108	519	742	649	1014	138

Table 2: Ablation study about guiding strategies. We run each setting 10 times.

some infeasible settings and did not get a correct protocol. In the following report, we use a triple prot-n-f to represent the setting: prot represents the protocol ("ac" for atomic commit and "con" for consensus). n is the total number of processes. We set $r = f + 1$. We compare GGMS with MCTS and MCTS+DFS to understand the effectiveness of DFS and Guided MCTS. MCTS includes the techniques described in §4.1; MCTS+DFS include §4.1 and §4.2; GGMS includes §4.1, §4.2, and §4.3.

We use a Transformer model for most of our experiments, but we compare to an MLP model. We run all experiments on CloudLab (Duplyakin et al., 2019). We document all model and machine settings in §D.

Found protocols. For consensus, GGMS finds protocols that are similar to the FloodSet protocol. For atomic commit, GGMS finds protocols that modify FloodSet to adapt to the additional requirement of atomic commit. Concretely, these protocols treat "lost message" as "abort" in the first round and ignore "lost message" in later rounds (see §E for an example): this is because "lost message" in the first round may be from a process that proposes to "abort", so the protocol has to be conservative; in later rounds, however, we cannot treat "lost message" as "abort", because if so, a crash of a process may cause other processes to diverge (a diverge in the first round can be mitigated by later rounds).

Success rate. Table 1 shows the success rate of different methods. As DFS can guarantee eventual success, this success rate is based on a limited time, which is documented above. For one setting, we run a method 10 times and report the number of times it can succeed in finding a correct protocol. For MCTS and MCTS+DFS, if they had a low success rate at a certain setting, we did not further try them on larger settings. As shown in this table, in all settings, GGMS consistently achieves higher success rates than MCTS and MCTS+DFS. From the training logs, we find that MCTS is primarily bothered by the superposition problem, and increasing training time does not help much. MCTS+DFS tries to address the superposition problem by freezing transitions, but when it freezes wrong ones, it will need unfreezing, which takes a lot of time.

We further perform an ablation study for guidance strategies in GGMS: 1) GGMS starts by allowing crashes in later rounds, and we test the strategy to start by allowing crashes in earlier rounds; 2) when determining which transitions to freeze, GGMS prioritizes transitions in later rounds, and we test the strategy with no such prioritization, i.e., selecting transitions from the same input with the closest probabilities. As shown in Table 2, starting by allowing crashes in later rounds is generally helpful for all cases. Prioritizing transitions in later rounds during freezing is helpful for atomic commit, which is the challenging case, but it is not much helpful for the less challenging consensus cases.

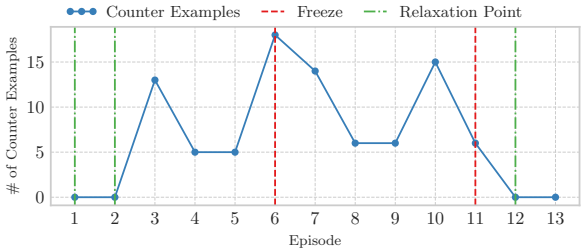

Figure 3: Number of counter examples for ac-3-2 (no unfreezing).

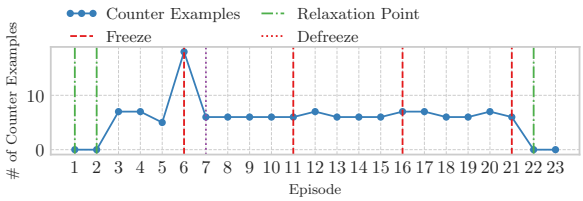

Figure 4: Number of counter examples for ac-3-2 (with unfreezing).

Running time. Table 1 shows the running time to converge to a correct model (excluding unsuccessful runs). GGMS' speed is faster than or similar to that of MCTS and MCTS+DFS in most of the settings. Note that excluding unsuccessful runs is favorable for MCTS and MCTS+DFS: In a difficult setting, MCTS and MCTS+DFS may fail many trials, which do not count, but GGMS may succeed after a long run, which increases its average running time.

Our further analysis shows that in GGMS, the simulation time for MCTS dominates the overall time compared to the training and validation time. For atomic commit, we also observe a significant variation in running time, usually due to incorrect freezing leading to unfreezing.

Scalability. The running time increases rapidly with larger settings, as expected. We currently implement GGMS with a single threaded Python program (except the verifier, which is parallelized). Techniques like C implementation and parallelization should be able to bring a significant performance improvement. However, the exponential growth pattern of the running time will probably persist. As a result, our speculation is that we may be able to scale to 8-10 processes for some protocols, but probably not to 100 processes.

However, scaling to a large number of processes may not be necessary to design a distributed protocol. For human experts, a general practice is to 1) design a protocol for a small number of processes, then 2) distill the insights from these small instances and generalize these insights for a generic protocol with an arbitrary number of processes, and finally 3) prove the correctness of the generic protocol. Therefore, as long as our tool can accomplish step 1), it may provide useful insights for human experts. Automating step 2) is our future work, probably involving the help from LLMs. 3) is well established in the distributed system community (Ma et al., 2019; Yao et al., 2021; Zhang et al., 2025b).

Number of counter examples during training. Figure 3 shows the number of counterexamples found by our verifier in one setting. At the beginning, the number of counterexamples is low, since guided MCTS focuses on some specific scenarios. Then, after relaxation, guided MCTS starts to explore more scenarios, leading to more counterexamples. GGMS performs two rounds of freezing, and the number of counterexamples drops to zero. Finally, after more relaxation, GGMS does not find more counterexamples. Figure 4 presents another example. GGMS first freezes a transition. It then discovers that, under this frozen condition, no correct model can be learned. Consequently, GGMS unfreezes the transition and continues training. The system eventually learns a correct model, though it requires more time compared to the example shown in Figure 3.

Effects of ML models. Our current implementation uses the Transformer model as discussed above. We also tried the MLP model. For the following settings ac-2-1, con-2-1, ac-3-2, and con-3-2, the success rate of using the MLP model is 100%, 70%, 10%, and 20%, respectively, compared to 100%, 100%, 60%, and 100% of using the Transformer model.

6 Future work

Integrating with LLMs. Recent work on LLM-based reasoning and code generation, including AlphaProof and AlphaEvolve (Hubert et al., 2025; Novikov et al., 2025), demonstrates the power of combining learned models with formal verification or evaluation oracles. In the future, we will explore whether we can fuse the strengths of LLMs and our work. In particular, our work can provide verification, (counter-)examples, and exploration for LLMs, especially when LLMs cannot find a correct protocol by itself, and LLMs can synthesize a generic protocol that works for an arbitrary number of processes, which is a challenge for our work.

Extending this work. §3 discusses several limitations of this work. Allowing a process to make a decision early before the last round is a straightforward extension of this work. Allowing a process to send different messages to different processes will significantly increase the search space. As a middle ground, we may allow a process to send its state to a subset of processes. To model asynchronous networking, we need to change our simulator to include more complicated message loss scenarios. To model Byzantine processes, we can incorporate adversarial models so that two models play against each other.

7 Related Work

We briefly position GGMS among several related families of methods, focusing on differences in goals and outcomes.

Classical distributed protocols and verification. Classical distributed protocols such as two-phase commit and atomic commit (Gray, 1978; 2005; Corbett et al., 2012; Zhang et al., 2015), Paxos and its variants (Lamport, 2001; 1998; Lamport et al., 1982; Moraru et al., 2013; Ongaro & Ousterhout, 2014; Castro & Liskov, 1999; Kotla et al., 2007; Giridharan et al., 2024) are hand-designed by experts and then validated by formal proofs or model checking. Frameworks such as IronFleet, I4, DistAI, and Basilisk (Hawblitzel et al., 2015; Ma et al., 2019; Yao et al., 2021; Zhang et al., 2025b) start from a human-written protocol and focus on proving its correctness, often for arbitrarily many processes. In contrast, GGMS treats the protocol itself as the object of synthesis: starting only from a specification of safety properties and a fixed number of processes, it searches in the space of global state machines and uses exhaustive model checking to ensure that the learned protocol satisfies the specification. Thus, classical work assumes the protocol and proves properties, whereas GGMS automates protocol construction and then certifies the resulting state machine (for the chosen bound).

Self-play and learning in multi-agent systems. Self-play deep reinforcement learning with tree search has achieved superhuman performance in perfect- and imperfect-information games such as backgammon, Atari, Go, poker, and Hanabi (Tesauro et al., 1995; Mnih et al., 2013; 2015; Silver et al., 2017; Lample & Chaplot, 2017; Wan et al., 2018; Tan et al., 2019; Li et al., 2020; Bard et al., 2020; Brown & Sandholm, 2019a; Anthony et al., 2017; Brown & Sandholm, 2019b; Sudhakar et al., 2025). These methods aim to maximize expected reward and typically output a neural policy that is not formally verified. GGMS borrows Monte Carlo Tree Search and self-play as search and data-generation tools, but with a different objective: instead of winning a game on average, it must synthesize a protocol that satisfies strict safety properties under worst-case crashes and message losses.

Closer to our setting, Khanchandani et al. (2021) use self-play to learn algorithms for a distributed directory problem, and verification-guided multi-agent RL approaches (e.g., Zhang et al. (2025a); Belardinelli et al. (2024)) combine learning with model checking to enforce temporal-logic specifications. These methods learn policies or algorithms and typically use model checking as a verification step or shaping signal, with correctness evaluated on sampled executions or specific scenarios. GGMS instead searches directly in a

symbolic space of global state machines for asynchronous crash-prone message-passing systems, and uses model checking as a hard oracle: any violating protocol is rejected, and counterexamples are fed back into training until no counterexamples remain within the explored state space. In this sense, GGMS is closer to protocol synthesis with built-in verification than to performance-driven policy learning.

Automata learning and program synthesis. The global state machine representation in GGMS is reminiscent of deterministic finite automata (DFA). Classical DFA learning methods (de la Higuera, 2005; Heule & Verwer, 2010) infer an automaton consistent with a fixed set of positive and negative example traces. GGMS also produces a finite-state machine, but its data is not a static dataset: executions are generated online via MCTS-guided self-play and iteratively refined using counterexamples from model checking, and the target is satisfaction of distributed safety properties under all crash and message-loss patterns modeled by the verifier.

At a higher level, GGMS is also related to classical program synthesis, particularly counterexample-guided inductive synthesis (CEGIS) as introduced in program sketching (Solar-Lezama, 2008). In CEGIS, a synthesizer proposes candidate programs from a constrained search space, and a verifier either accepts the candidate or returns a counterexample input; the counterexample is added to the training set and the loop repeats. GGMS follows a similar high-level architecture: a search procedure proposes candidate protocols and a model checker either accepts them or returns offending executions. However, typical CEGIS systems operate on general-purpose sequential programs (often using SMT-based search) and reason about input/output behavior, whereas GGMS searches over a structured space of distributed protocol state machines under asynchronous semantics and crash/message-loss faults, guided by reinforcement learning and MCTS rather than purely symbolic search. Moreover, GGMS explicitly explores the global state space of the protocol (up to a bound) to guarantee correctness for all executions within that model.

Large-scale reasoning and coding agents. Recent large-scale systems also combine powerful search or reinforcement learning with formal environments. AlphaProof is an AlphaZero-inspired agent for formal mathematics that operates inside the Lean theorem prover (Hubert et al., 2025). Each problem instance is a formal theorem; the agent observes proof states, proposes tactics, and uses MCTS guided by a transformer "proof network" to search for proofs, with the Lean kernel checking correctness. The goal is to *find proofs* of given statements within a fixed formal system, and the outcome is a proof term.

AlphaEvolve is an evolutionary coding framework for scientific and algorithmic discovery (Novikov et al., 2025), in which large language models propose edits to candidate programs that are then executed and scored by problem-specific evaluation functions. The framework has been applied to discover new algorithms and improve implementations in several domains. In AlphaEvolve, candidate solutions are arbitrary programs, and their correctness or utility is judged by external evaluation metrics defined by the user.

GGMS is complementary to both: rather than proving theorems in a general-purpose proof assistant or evolving arbitrary code under user-defined metrics, GGMS operates in a fixed asynchronous message-passing model with crash and message-loss faults, searches over a constrained symbolic space of distributed protocols, and uses exhaustive model checking of all executions within this model to certify correctness (for a fixed number of processes).

8 Conclusion

This work explores the new area of learning provably correct distributed protocols with partial observability. We propose GGMS, which combines model checking to ensure correctness, DFS to ensure eventual convergence, and guided MCTS to accelerate convergence. As a preliminary exploration, we further discuss possible future directions.

References

Thomas Anthony, Zheng Tian, and David Barber. Thinking fast and slow with deep learning and tree search. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus,

S. V. N. Vishwanathan, and Roman Garnett (eds.), *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pp. 5360–5370, 2017. URL https://proceedings.neurips.cc/paper/2017/hash/d8e1344e27a5b08cdfd5d027d9b8d6de-Abstract.html.

Nolan Bard, Jakob N Foerster, Sarath Chandar, Neil Burch, Marc Lanctot, H Francis Song, Emilio Parisotto, Vincent Dumoulin, Subhodeep Moitra, Edward Hughes, et al. The hanabi challenge: A new frontier for ai research. *Artificial Intelligence*, 280:103216, 2020.

Francesco Belardinelli, Wojtek Jamroga, Munyque Mittelmann, and Aniello Murano. Verification of stochastic multi-agent systems with forgetful strategies. In *Proceedings of the 23rd International Conference on Autonomous Agents and Multiagent Systems*, AAMAS '24, pp. 160–169, Richland, SC, 2024. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 9798400704864.

Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault tolerance. *ACM Trans. Comput. Syst.*, 14(1):80–107, February 1996. ISSN 0734-2071. doi: 10.1145/225535.225538. URL https://doi.org/10.1145/225535.225538.

Noam Brown and Tuomas Sandholm. Superhuman ai for multiplayer poker. *Science*, 365(6456):885–890, 2019a.

Noam Brown and Tuomas Sandholm. Superhuman ai for multiplayer poker. *Science*, 365(6456):885–890, 2019b. doi: 10.1126/science.aay2400. URL https://www.science.org/doi/abs/10.1126/science.aay2400.

Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *3rd Symposium on Operating Systems Design and Implementation (OSDI 99)*, New Orleans, LA, February 1999. USENIX Association. URL https://www.usenix.org/conference/osdi-99/practical-byzantine-fault-tolerance.

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pp. 251–264, USA, 2012. USENIX Association. ISBN 9781931971966.

Colin de la Higuera. A bibliographical study of grammatical inference. *Pattern Recogn.*, 38(9):1332–1348, September 2005. ISSN 0031-3203. doi: 10.1016/j.patcog.2005.01.003. URL https://doi.org/10.1016/j.patcog.2005.01.003.

Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340. Springer, 2008.

Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pp. 1–14, July 2019. URL https://www.flux.utah.edu/paper/duplyakin-atc19.

Neil Giridharan, Florian Suri-Payer, Ittai Abraham, Lorenzo Alvisi, and Natacha Crooks. Autobahn: Seamless high speed bft. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, pp. 1–23, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400712517. doi: 10.1145/3694715.3695942. URL https://doi.org/10.1145/3694715.3695942.

James N Gray. Notes on data base operating systems. *Operating systems: An advanced course*, pp. 393–481, 2005.

Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pp. 393–481. 1978.

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. Ironfleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pp. 1–17, 2015.

Marijn J. H. Heule and Sicco Verwer. Exact dfa identification using sat solvers. In *Proceedings of the 10th International Colloquium Conference on Grammatical Inference: Theoretical Results and Applications*, ICGI'10, pp. 66–79, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3642154875.

Thomas Hubert, Rishi Mehta, Laurent Sartran, Miklós Z. Horváth, Goran Žužić, Eric Wieser, Aja Huang, Julian Schrittwieser, Yannick Schroecker, Hussain Masoom, Ottavia Bertolli, Tom Zahavy, Amol Mandhane, Jessica Yung, Iuliya Beloshapka, Borja Ibarz, Vivek Veeriah, Lei Yu, Oliver Nash, Paul Lezeau, Salvatore Mercuri, Calle Sönne, Bhavik Mehta, Alex Davies, Daniel Zheng, Fabian Pedregosa, Yin Li, Ingrid von Glehn, Mark Rowland, Samuel Albanie, Ameya Velingker, Simon Schmitt, Edward Lockhart, Edward Hughes, Henryk Michalewski, Nicolas Sonnerat, Demis Hassabis, Pushmeet Kohli, and David Silver. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature*, 2025. doi: 10.1038/s41586-025-09833-y. URL https://www.nature.com/articles/s41586-025-09833-y.

Pankaj Khanchandani, Oliver Richter, Lukas Rusch, and Roger Wattenhofer. Learning algorithms with self-play: A new approach to the distributed directory problem. In *2021 IEEE 33rd International Conference on Tools with Artificial Intelligence (ICTAI)*, pp. 501–505. IEEE, 2021.

Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative byzantine fault tolerance. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pp. 45–58, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595935915. doi: 10.1145/1294261.1294267. URL https://doi.org/10.1145/1294261.1294267.

Guillaume Lample and Devendra Singh Chaplot. Playing fps games with deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, may 1998. ISSN 0734-2071. doi: 10.1145/279227.279229. URL https://doi.org/10.1145/279227.279229.

Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, 2001.

Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, jul 1982. ISSN 0164-0925. doi: 10.1145/357172.357176. URL https://doi.org/10.1145/357172.357176.

Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. SAMC: Semantic-Aware model checking for fast discovery of deep bugs in cloud systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pp. 399–414, Broomfield, CO, October 2014. USENIX Association. ISBN 978-1-931971-16-4. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/leesatapornwongsa.

Junjie Li, Sotetsu Koyamada, Qiwei Ye, Guoqing Liu, Chao Wang, Ruihan Yang, Li Zhao, Tao Qin, Tie-Yan Liu, and Hsiao-Wuen Hon. Suphx: Mastering mahjong with deep reinforcement learning. *arXiv preprint arXiv:2003.13590*, 2020.

Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996. ISBN 9780080504704.

Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A Sakallah. I4: incremental inference of inductive invariants for verification of distributed protocols. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 370–384, 2019.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pp. 358–372, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323888. doi: 10.1145/2517349.2517350. URL https://doi.org/10.1145/2517349.2517350.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025. doi: 10.48550/arXiv.2506.13131. URL https://arxiv.org/abs/2506.13131.

Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pp. 305–320, USA, 2014. USENIX Association. ISBN 9781931971102.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, EECS Department, University of California, Berkeley, December 2008. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-177.html.

Arjun Vaithilingam Sudhakar, Hadi Nekoei, Mathieu Reymond, Miao Liu, Janarthanan Rajendran, and Sarath Chandar. A generalist hanabi agent. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=pCj2sLNoJq.

Tian Tan, Feng Bao, Yue Deng, Alex Jin, Qionghai Dai, and Jie Wang. Cooperative deep reinforcement learning for large-scale traffic grid signal control. *IEEE transactions on cybernetics*, 50(6):2687–2700, 2019.

Gerald Tesauro et al. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3): 58–68, 1995.

Zhiqiang Wan, Chao Jiang, Muhammad Fahad, Zhen Ni, Yi Guo, and Haibo He. Robot-assisted pedestrian regulation based on deep reinforcement learning. *IEEE transactions on cybernetics*, 50(4):1669–1682, 2018.

Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. DistAI: Data-Driven automated invariant learning for distributed protocols. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pp. 405–421. USENIX Association, July 2021. ISBN 978-1-939133-22-9. URL https://www.usenix.org/conference/osdi21/presentation/yao.

Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pp. 263–278, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450338349. doi: 10.1145/2815400.2815404. URL https://doi.org/10.1145/2815400.2815404.

Mingyue Zhang, Nianyu Li, Yi Chen, Jialong Li, Xiao-Yi Zhang, Hengjun Zhao, Jiamou Liu, and Wu Chen. Learning verified safe neural network controllers for multi-agent path finding. *Proceedings of the AAAI Conference on Artificial Intelligence*, 39(22):23369–23377, Apr. 2025a. doi: 10.1609/aaai.v39i22.34504. URL https://ojs.aaai.org/index.php/AAAI/article/view/34504.

Tony Nuda Zhang, Keshav Singh, Tej Chajed, Manos Kapritsos, and Bryan Parno. Basilisk: Using provenance invariants to automate proofs of undecidable protocols. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pp. 1–17, 2025b.

A Formal definitions of investigated protocols

The complete definitions of each state are provided in Table S1 and Table S2, for the atomic commit and consensus protocols, respectively. Based on these state definitions, we formally define the properties of each protocol following prior work. Let N denote the set of all processes. Each process has an initial state b_n, and a set of final decisions, F_n. Let M represent all messages exchanged between processes, and let $L = \exists m \in M(m = Lost)$ indicate that there exists at least one lost message.

State	Meaning
`init:abort`	Initial state representing intent to abort
`init:commit`	Initial state representing intent to commit
`internal:a`	Internal state
`internal:b`	Internal state
`decision:abort`	Final decision to abort
`decision:commit`	Final decision to commit

Table S1: State Definitions for Atomic Commit Protocol

State	Meaning
`init:0`	Initial state representing intent to commit 0
`init:1`	Initial state representing intent to commit 1
`internal:a`	Internal state
`internal:b`	Internal state
`decision:0`	Final decision to commit 0
`decision:1`	Final decision to commit 1

Table S2: State Definitions for Consensus Protocol

Atomic commit protocol properties. Rule P1 states that no two processes should reach conflicting decisions. Rule P2 ensures that if all processes initially intend to commit and there are no message losses, then all processes must reach a commit decision. Rule P3 indicates that if any process initially intends to abort, then all processes must reach an abort decision. Rule P4 requires that every process must eventually reach a final decision.

P1.

$$\neg\exists n, m \in \{1, 2, ..., N\}\ \exists f_n, f_m \in F\ \big(f_n = \texttt{decision:abort} \wedge f_m = \texttt{decision:commit}\big) \quad (2)$$

P2.

$$\big(\forall n \in \{1, 2, ..., N\},\ b_n = \texttt{init:commit}\big) \wedge (\neg L) \Rightarrow \big(\forall n \in N,\ \forall f_n \in F_n,\ f_n = \texttt{decision:commit}\big). \quad (3)$$

P3.

$$(\exists n \in \{1, 2, ..., N\},\ b_n = \texttt{init:abort}) \Rightarrow\ (\forall n \in \{1, 2, ..., N\},\ \forall f_n \in F_n,\ f_n = \texttt{decision:abort}) \quad (4)$$

P4.

$$\forall n \in N, \exists f_n \in F\ \big(f_n = \texttt{decision:abort} \vee f_n = \texttt{decision:commit}\big) \quad (5)$$

Note that the well-known atomic protocols like two-phase commit (2PC) and three-phase commit (3PC) make the asynchronous networking assumptions that every message has a non-zero chance of being lost. As discussed in Section 3, our current implementation supports only synchronous networking assumptions. Such a difference leads to a difference in the protocol properties: In the asynchronous versions, we can prove that there always exist scenarios that a correct process cannot decide. In the synchronous versions, such scenarios do not exist.

Consensus protocol properties. Rule P1 states that no two processes should reach conflicting decisions. Rules P2 and P3 assert that if all processes start with the same initial proposal, then all final decisions must match that proposed value. Rule P4 requires that every process must reach a decision by the end of the protocol.

P1.

$$\neg\exists n, m \in \{1, 2, ..., N\} \quad \exists f_n, f_m \in F \quad \big(f_n = \texttt{decision:0} \wedge f_m = \texttt{decision:1}\big) \tag{6}$$

P2.

$$(\forall n \in \{1, 2, ..., N\}, b_n = \texttt{init:1}) \qquad \Rightarrow \qquad (\forall f_n \in F, f_n = \texttt{decision:1}) \tag{7}$$

P3.

$$(\forall n \in \{1, 2, ..., N\}, b_n = \texttt{init:0}) \qquad \Rightarrow \qquad (\forall f_n \in F, f_n = \texttt{decision:0}) \tag{8}$$

P4.

$$\forall n \in \{1, 2, ..., N\}, \exists f_n \in F \quad \big(f_n = \texttt{decision:0} \vee f_n = \texttt{decision:1}\big) \tag{9}$$

B Proof of Theorem

Theorem B.2. *In a feasible setting, assuming 1) the scenario Sc of this particular simulation is a definite scenario, and 2) for each pair of (round, procID), this step only fixes one transition (round, procID, inputA) → B, then there exists a correct state machine with all the fixed transitions ($Transitions_{fix}$) in this step.*

Proof. Assuming there exists a correct state machine SM_0, we prove that we can construct a state machine SM_{r-1}, such that, 1) SM_0 and SM_{r-1} are equivalent, i.e., for every scenario S, every non-crashed process makes the same decision with SM_0 and SM_{r-1}; and 2) SM_{r-1} includes all the transitions in $Transitions_{fix}$.

We construct by induction of $r-1$ steps (r is the number of maximal rounds). Assuming SM_{i-1} is already constructed ($1 < i < r$), we construct SM_i in the following way. We search for procID, such that transition $[round = i, procID, inputA] \to B$ exists in $Transitions_{fix}$ and $[round = i, procID, inputA] \to C$ exists in SM_{i-1} and B is different from C. Then we swap B and C in SM_{i-1} to get SM_i: 1) For any input, if there exists a transition $[i, procID, input] \to C/B$ in SM_{i-1}, we change it to $[i, procID, input] \to B/C$ in SM_i; 2) In round i+1, for any procID2, if there exists a transition $[i+1, procID2, input] \to D$ in SM_i, where the input vector includes C (or B) from procID, we change B into C and C into B in the input vector.

First, we can prove that SM_i includes transitions in $Transitions_{fix}$ whose round is smaller than i+1. We prove by induction. When $i = 1$, SM_1 is constructed from SM_0. Under Sc, SM_0 and $Transitions_{fix}$ must reach the same $[round = 1, procID, inputA]$ in the first round, since the input is from initial states. And if a $[round = 1, procID, inputA]$ transit to different states in SM_1 and $Transitions_{fix}$, our construction changes the output of the transition in SM_1 to match that in $Transitions_{fix}$. Then for the later round i, we can prove it in the same way. Under Sc, $Transitions_{fix}$ and SM_{i-1} must reach the same $[round = i, procID, inputA]$ in round $i-1$, as they have the same transitions for Sc. Then if $[round = i, procID, inputA]$ transit to different states in $Transitions_{fix}$ and SM_{i-1}, our construction forces them to be the same in $Transitions_{fix}$ and SM_i Note that this only works if for each pair of (round, procID), $Transitions_{fix}$ only includes one transition. Otherwise, the construction may need to swap multiple pairs of values, and they may conflict, which means the construction may not be possible (e.g., we cannot both swap B and C and swap B and D).

Second, we can prove that SM_{i-1} and SM_i are equivalent. Assuming a simulation applies transition $(i, procID, input) \to B/C$ in SM_{i-1}, the simulation will get the same input for procID in round i when applying SM_i, as SM_{i-1} and SM_i have the same set of transitions before round i. Then the simulation will

Algorithm 1: Pseudo code of GGMS

```
model ← init_model();
phase_ID ← 0;
training_buffer ← [];
failed_scenarios ← [];
freeze_list ← [];
/* Each main loop iteration is an episode                                    */
while true do
    for i ← 1 to 100 do
        scenario ← sample_scenario(phase_ID, failed_scenarios);
        training_data, reward ← run_mcts(scenario, model);
        training_buffer.append(training_data);
        determine_unfreeze(reward, freeze_list);
    end
    determine_freezing(training_buffer, freeze_list);
    update_model(training_buffer, model);
    failed_scenarios ← validate(phase_ID, model);
    if failed_scenarios == [] then
        if phase_ID == lastPhase then
            terminate;
        end
        else
            phase_ID ← phase_ID + 1;
        end
    end
end
```

apply $(i, procID, input) \rightarrow C/B$ in SM_i, i.e., output of SM_{i-1} and SM_i swap B and C for procID. However, since our construction also swaps B and C in the input of round i+1 from procID, we can know that in round i+1, every process will still transit to the same state. Therefore, SM_{i-1} and SM_i are equivalent.

With the above steps, we can construct a state machine SM_{r-1} that is equivalent to SM_0 and that includes all the transitions from $Transitions_{fix}$ till round r-1. In round r, every state machine transits to the decision state. We can prove that SM_{r-1} must have the same transitions as those in $Transitions_{fix}$ for round r, with no need for swapping. This is because, for Sc that generates $Transitions_{fix}$, SM_{r-1}, and $Transitions_{fix}$ will apply the same transitions till round r-1, so every process should have the same input at the beginning of round r. If SM_{r-1} and $Transitions_{fix}$ have different transitions for the same input in round r, they will lead to different decisions of some processes. This contradicts our assumption that Sc is a definite scenario.

Therefore, we have proved that SM_{r-1}, which is equivalent to SM_0 and must be correct, has all the transitions of $Transitions_{fix}$.

□

C Implementation Details

C.1 Overview

Algorithm 1 presents the high-level pseudo code of GGMS.

`model` is a neural network representing the state machine we want to learn. As discussed in Section 3, it takes $[round, procID, inputStates]$ as the input and outputs a $newState$. In fact, to facilitate learning, we let it output a probability for each potential value of $newState$. During inference, GGMS will choose the value with the highest probability.

Algorithm 2: Pseudo code of choosing a scenario

```
1  Function sample_scenario(phase_ID, failed_scenarios):
2      u ← Uniform(0, 1);
3      if u < 0.3 or failed_scenarios == [] then
4          scenario ← uniform_sample(generate_all_scenarios(phase_ID));
5      end
6      else
7          scenario ← uniform_sample(failed_scenarios);
8      end
9      return scenario;
10 end
```

`phase_ID` is used to implement the guided MCTS (Section 4.3). When set to 0, it means that GGMS will only simulate scenarios with definite initial states and message losses in the last round. When set to 1, it means that GGMS will simulate scenarios with definite initial states and message losses in the last two rounds, etc. Finally, when set to r (the total number of rounds), it means that GGMS can use any initial state and message losses in any round.

`training_buffer` is a bounded buffer to store training data collected during MCTS. Each item in the buffer records the probability of a transition from $[round, procID, inputStates]$ to one potential output value.

`failed_scenarios` records scenarios that caused prior validation to fail. As discussed, GGMS will use such scenarios to retrain the model.

`freeze_list` is used to implement DFS (Section 4.2). Like a conventional DFS implementation, `freeze_list` is a stack, and each item in the stack represents one frozen transition $[round, procID, inputStates] \rightarrow newState$. Each item also records whether other values have been frozen for the same $[round, procID, inputStates]$ in the past.

The whole algorithm works in multiple episodes. In each episode, it runs MCTS on 100 scenarios (line 7). Each scenario is either randomly chosen from scenarios allowed by the current phase or from past failed scenarios (line 8). Running MCTS on the scenario will generate some training data, which will be added to the training buffer, and a reward (lines 9-10). GGMS will determine whether it needs to unfreeze depending on the reward (line 11).

Then, after simulating 100 scenarios, GGMS will determine whether it needs to freeze more transitions based on the new training data (line 13). Note that unfreeze and freeze are determined at different timings: If MCTS cannot find a good model for one scenario, that is already enough to trigger unfreeze, and that is why unfreeze is determined after simulating every scenario. However, determining freezing often requires information from multiple scenarios, which can lead to potentially different transitions for the same input, and that is why GGMS determines freezing after trying a number of scenarios.

Then GGMS updates the model using the new training data (line 13) and then validates the new model (line 14). If validation does not find any failed scenarios, GGMS will either terminate if this is the last phase, or proceed to the next phase otherwise (lines 16-23).

C.2 Use counterexamples to retrain

Algorithm 2 presents the details of how GGMS selects scenarios to simulate. It has a 70% chance to choose a failed scenario in the past, if any, and 30% chance to randomly choose a scenario allowed by the current phase.

Algorithm 3: Pseudo code of determining unfreezing

```
1  Function determine_unfreeze(reward, freeze_list):
       /* Unfreeze when reward is negative and some frozen transition was activated */
2      if reward < 0 and has_activated(freeze_list) then
           /* At least one entry can be popped out */
3          while true do
4              entry ← freeze_list.pop_one_activated();
5              if entry is not fully explored then
6                  freeze_to_new_value(entry);
7                  freeze_list.push(entry);
8                  Break;
9              end
10         end
11     end
12 end
```

Algorithm 4: Pseudo code of determining freezing

```
1  Function determine_freezing(training_buffer, freeze_list):
       /* Find inputs whose outputs have close probabilities and freeze one */
2      tmp ← find_ambiguous_inputs(training_buffer, p_min=0.2, diff_max=0.1);
       /* Sort: later round first, then fewer lost messages first */
3      tmp ← sort(tmp, key=[round_desc, lost_msgs_asc]);
4      if tmp ≠ [] then
5          cand ← tmp[0];
6          entry ← cand.freeze_outputs;
7          freeze_list.push(entry);
8      end
9  end
```

C.3 Enhancing MCTS with DFS

Algorithm 3 presents the logic for determining whether to unfreeze a frozen transition and, if so, which transition to unfreeze. Our current implementation uses the condition that the reward is negative, which means that MCTS cannot find a model to reach correct decisions for this scenario, and at least one frozen transition is activated (line 2). In our current implementation, MCTS is given enough time to fully explore all state machines relevant to this scenario, so the negative reward is an accurate condition to trigger unfreezing. However, such exhaustive search scales poorly. For better scalability, we may replace this condition with a Z3-style validation to prove that, given the frozen transitions, the model can never reach correct decisions for the corresponding scenario, no matter what other transitions this model includes. We will explore this in the future.

If the condition is met, GGMS unfreezes transitions in the DFS manner. It pops an activated transition from the stack (line 4). If the corresponding input of the transition is not fully explored (line 5), which means that GGMS has not tried to freeze it to all the possible values, GGMS will try to freeze it to a value that has not been explored (line 6) and push this entry back into the stack. Otherwise, GGMS will keep popping until it can find such an entry.

Algorithm 4 presents the logic of determining whether to freeze a new transition and, if so, which one to freeze. Our current implementation uses the heuristics that if multiple outputs for the same input have a probability larger than 0.2 and the difference between their probabilities is within 0.1, then GGMS considers them as targets for freezing (line 2). If multiple such inputs exist, GGMS sorts them based on their round number and the number of lost messages in the input and chooses the one with the highest round number and

Algorithm 5: Pseudo code of validation

```
1  Function validate(phase_ID, model):
       /* Enumerate all patterns for the phase and simulate (can run in parallel)  */
2      init_state_patterns ← gen_all_inputs(phase_ID);
3      msg_loss_patterns ← gen_loss_patterns(phase_ID);
4      all_scenarios ← init_state_patterns × msg_loss_patterns;
5      failed ← [];
6      foreach scenario ∈ all_scenarios  in parallel do
7          ok ← simulate(model, scenario);
8          if not ok then
9              failed.append(scenario);
10         end
11     end
12     return failed;
13 end
```

the lowest number of lost messages as input (lines 3-5). This is based on our experience in the importance of such transitions. Note that this heuristic does not affect the eventual convergence of DFS. Finally, GGMS pushes the newly determined frozen transition into `freeze_list`.

C.4 Brute-force validation

Algorithm 5 presents our brute-force algorithm to validate whether the model is fully accurate. It enumerates all the possible scenarios by doing a cross product of all the possible initial state patterns and all the message loss patterns (lines 2-4). Generating all the possible initial state patterns is straightforward: Suppose that there are N processes and x possible initial state values. GGMS enumerates all ways to assign initial states to different processes, generating a total of x^N patterns. Generating all message loss patterns is more complex. GGMS first generates all possible process crash patterns given *phase_ID*. Then, GGMS generates all message loss patterns based on such crash patterns. In any round before a process crashes, GGMS marks all its messages as not lost. In any round after a process crashes, GGMS marks all its messages as lost. In the same round as a process crashes, its messages may or may not be lost, and thus GGMS enumerates all such possibilities. Finally, GGMS performs a cross product of the possible message loss patterns of each process.

Then GGMS simulates each of the scenarios to see whether any of them will cause the model to reach incorrect decisions (lines 6-11). Our current implementation parallelizes such simulation of multiple scenarios. As discussed in Section 5, validation is not the bottleneck of our current implementation. In the future, we plan to replace it with a Z3-based validation for better scalability.

C.5 Monte-Carlo Tree Search

Algorithm 6: Pseudo code of Monte-Carlo Tree Search

```
1  Function run_mcts(scenario, model, freeze_list):
       /* protocol: protocol representation                                   */
2      protocol ← initial(scenario);
3      buffer ← [];
4      while not protocol.is_done() do
           /* Run Monte-Carlo tree search from current state                  */
5          P ← simulate(protocol, model, freeze_list);
           /* Pick transition based on simulation results                     */
6          transition ← select(P);
           /* Advance to next round                                           */
7          protocol.step(transition, scenario);
           /* Non-zero reward only at the last round                          */
8          reward ← get_reward();
           /* Record (state, simulated probabilities)                         */
9          buffer.append(⟨current_state(), P⟩);
10     end
11     return buffer, reward;
12 end
13 Function simulate(protocol, model, freeze_list):
14     for i ← 0 to iter do
           /* Store visited path during simulation                            */
15         visited ← [];
16         while protocol is not done do
               /* Select transitions and message losses based on Equation 10  */
17             transition ← select_transition(model, freeze_list);
18             loss ← select_message_loss();
19             protocol.next(transition, loss);
20             visited.append(transition, loss);
21         end
22         reward ← get_reward();
23         backup(reward, visited);
24     end
25 end
```

Algorithm 6 shows the details of the MCTS simulation. At the beginning (line 2), it initializes a new *protocol* object, representing the full protocol execution state with a specific *scenario*. Then it simulates the protocol starting from the current state (line 13). We will describe the `simulate` function in detail later. It returns a probability distribution over transitions for the corresponding state, which is then stored in the training buffer (line 9). Next, it selects a transition based on the simulation results and moves to the next protocol state. This process repeats until the protocol terminates.

The `simulate` function primarily consists of four components. Figure 2 illustrates an example of MCTS: it shows the search tree constructed during the simulation of one episode, and how the path is selected through it. We will introduce these main components in both Algorithm 6 and Figure 2.

- **Selection** (line 17 and line 18). The algorithm will traverse the tree from the root. When selecting a transition from the current node, if a transition is in the `freeze_list`, GGMS selects it directly. Otherwise, GGMS selects a transition based on a balance between exploitation and exploration based on Upper Confidence Bound score as shown in Equation 10. The transition with the highest $U(s, a)$ will be selected in the simulation. $Q(s, a)$ is the accumulative average rewards that taking transition a in state s during simulation. $P(s, a)$ is the probability of choosing transition a in state s, given by the policy

Layer Type	Input Shape	Output Shape
One-hot Encoder	(input)	(input, encode_dim)
Transformer Encoder	(input, encode_dim)	(input, hidden)
GlobalAveragePooling1D	(input, hidden)	(hidden)
Output	(hidden)	(output)

Table S3: The Transformer architecture used to learn the state machine

network. $N(s, a)$ represents the number of times transition a has been chosen in state s during tree search simulations. c_{puct} is a constant parameter that controls the balance between exploitation and exploration. The selection will terminate when the protocol terminates. All visited transitions during selection will be stored.

$$U(s,a) = Q(s,a) + c_{puct} \cdot P(s,a) \cdot \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)} \tag{10}$$

The selection of transitions and message losses follows the same logic but with opposing objectives: the transition selector aims to maximize the final reward, while the message loss selector aims to minimize it. Unlike transition selection, message loss selection relies solely on accumulated rewards without guidance from the network. As shown in Figure 2, within a single iteration of simulation, the search may follow a path such as C1 $\rightarrow$ A1 $\rightarrow$ C3 $\rightarrow$ A4 (illustrated with solid arrows). In other iterations, different paths may be selected.

Note that, as shown in this algorithm, although each simulation has a targeted scenario, it will explore other reachable scenarios during its search to avoid obviously wrong transitions to other scenarios.

- **Expansion**. When an unexpanded node is reached, all of its available child nodes will be added to the tree for further simulation. For each new node, the visited count and the accumulative reward will be initialized to 0.

- **Evaluation** (line 22). When the *protocol* reaches termination, the final reward is computed based on a predefined reward function (+1 if no correctness property is violated and -1 otherwise). As shown in Figure 2, MCTS selects a particular path in this simulation (C1 $\rightarrow$ A1 $\rightarrow$ C3 $\rightarrow$ A4). At round 1, processes P2 and P3 choose the transition `internal:a(2)`. At round 2, the only living process, P3, selects transition `decision:1(1)`. According to the reward function, this selection results in a final reward of +1.

- **Backup** (line 23). The final reward is backed up along the search path stored in `visited`. The visit count and accumulated rewards of the corresponding nodes are updated accordingly. For example, suppose the selected path in this iteration is C1 $\rightarrow$ A1 $\rightarrow$ C3 $\rightarrow$ A4, and the final reward is +1. In this case, the visit count of all nodes along the path is incremented by 1. The accumulated rewards of nodes A1 and A4 are increased by 1, indicating the protocol has made the right transitions, while those of nodes C1 and C3 are increased by the opposite, -1, indicating the adversary is not able to defeat the protocol. These values will be used to compute the probability of each activated transition at the end of `simulate`.

D Experiment setting

We use a Transformer model, whose architecture is presented in Table S3. The Transformer model includes only the encoder component of the standard Transformer architecture, as the task does not require contextual information like translation tasks. Additionally, we use a one-hot encoding layer at the input to transform categorical values into unique vectors. The Transformer encoder outputs a tensor of shape batch $\times$ input_length $\times$ hidden_dim. We apply a global pooling layer to aggregate the outputs across the input sequence into a vector of shape batch $\times$ hidden_dim, which is then passed through an output layer to obtain the final prediction vector. We also tried the MLP model. Table S4 demonstrates the MLP model architecture that we use. It contains three fully connected layers (FC) and one output layer. We use ReLU as the activation function after each FC layer. We use cross-entropy as the loss function for both models.

Layer Type	Size	Output
FC-1	3x128	1x128
FC-2	128x64	1x64
FC-3	64x32	1x32
Output	32x3	1x3

Table S4: The MLP architecture used to learn the state machine

Algorithm 7: Atomic commit found by GGMS (f=1)

Initialization :
  $V_i \leftarrow \{v_i\}$ either init:commit or init:abort
Round 1 :
  if no message loss and every received V_j is init:commit
    $V_i \leftarrow internal : a$
  else
    $V_i \leftarrow internal : b$
Round 2 :
  if every received V_j is internal:a
    $V_i \leftarrow decision : commit$
  else
    $V_i \leftarrow decision : abort$

We run all experiments on CloudLab (Duplyakin et al., 2019). The server we use is equipped with a 16-core AMD 7302P CPU running at 3.00GHz and has 128GB of memory. We implement the system in Python and use Keras for model training. The learning rate is set to 0.001, and the cross-entropy loss function is used to update the model parameters. In our experiments, we simulate 100 scenarios in each episode. For each simulation, the number of iterations is determined by the scale of the setting, calculated as $\#\texttt{rounds} \times \#\texttt{processes} \times 1000$. We freeze a new transition every 5 episodes to ensure that all previously frozen transitions have propagated. To detect training failure, we set a timeout for each run based on the scale. Specifically, we allocate 1 day for 2-process settings, 2 days for 3-process settings, and 4 days for 4-process settings. Some distributed protocols require all processes to make the same decision, so a process may not need the process ID as the input, since processes with the same input, regardless of the process ID, should transit to the same state. We find that this is true for both the atomic commit protocol and the consensus protocol we have investigated, so we disabled the process ID in our experiments to reduce search space.

E Found atomic commit protocol

Algorithm 7 shows the pseudo code of the atomic commit protocol found by GGMS.