Learning Distributed Protocols with Zero Knowledge

Yujie Hui
The Ohio State University
hui.82@osu.edu

Drew Ripberger
The Ohio State University
ripberger.8@osu.edu

Xiaoyi Lu
University of California, Merced
xiaoyi.lu@ucmerced.edu

Yang Wang
The Ohio State University
wang.7564@osu.edu

Abstract

The success of AlphaGo Zero shows that a computer can learn to play a complicated board game without relying on the knowledge from human players. We observe that designing a distributed protocol is similar to playing board games to some extent: when determining the next action to take, they both want to ensure they can win even when a smart opponent tries to drive the game/protocol to the worst case. In this work, we explore whether we can apply similar techniques to learn a distributed protocol with zero knowledge. Towards this goal, we model the process in a distributed protocol as a state machine, and further rely on model checking to validate the correctness of the learned state machine. With this approach, we successfully learned a correct atomic commit protocol with three processes, and upon that, we further discuss future work.

1 Introduction

Designing and optimizing a distributed protocol is challenging, often taking decades of efforts. Examples include the series of works on Atomic Commit [4, 8, 22, 35], Paxos [13, 14, 21, 23, 26, 30, 33], and Byzantine Fault Tolerance [1, 3, 5, 11, 15]. This work explores whether we can leverage AI techniques to design and optimize distributed protocols. In particular, to minimize human effort, we expect this approach to be zero-knowledge: *a human expert only needs to specify the desired properties of the target protocol, but does not need to suggest any internal details of the protocol*.

AI has made great successes in board games like chess and go: AlphaGo Zero can beat human Go champions by self playing and training without relying on knowledge from human players [28]. We observe designing a distributed protocol is similar to playing board games to a large extent: a process needs to choose an action so that, even an opponent drives the system to the worst case, certain properties are never violated. In a distributed protocol, such an opponent could be a malicious process trying to break the protocol, a selfish process trying to get more resources, or random crashes and message losses. To explore this idea, we address the following challenges:

- Model a distributed protocol with zero knowledge. Unlike a board game, which typically has well-defined states and actions, a distributed protocol does not. How to model a distributed protocol becomes the first challenge we face: this model should be generic enough to describe most of the protocols and should be able to be mapped to an AI model. Fortunately, we find the distributed system community has already provided an answer: we can model a process in a distributed protocol as a state machine. The state machine approach has been widely used [25], which proves its generality. A state machine, which determines the next state and output based on its input and current state, can be mapped directly to an AI model. A human expert can specify the desired properties of a state machine, without knowing its internal details.

Machine Learning for Systems Workshop at 37th NeurIPS Conference, 2023, New Orleans, LA, USA.

- Validate the learned protocol. Learning often cannot achieve full accuracy, but a distributed protocol is expected to be fully correct. To address this challenge, we combine learning with model checking, which is widely used to verify the correctness of distributed protocols [17, 32]. If checking fails, we can use the counter examples provided by model checking to further train the model.

We have applied our approach to learn a simplified atomic commit protocol (no recovery). So far, our approach can learn a 3-node protocol that can pass model checking. Based on such results, we discuss future directions.

2 Approach

In this section, we first describe how to model a distributed protocol and then present the learning procedure and the verifying procedure.

2.1 Modelling a Distributed Protocol with Zero Knowledge

Following the widely used state machine approach, we model each participant process of a distributed protocol as a deterministic state machine. A state machine has a set of states, inputs, outputs, and a transition function to determine the (next state, output) from (current state, input). Figure 1a shows an example state machine, which indicates that, if a process is in state 2, and receives three 2s as input, it will transition to state 3.

Modelling a process as a state machine has several benefits. First, the state machine can be mapped directly to a neural network, which takes (current state, input) as the input to compute (next state, output). Second, because of the popularity of the state machine approach, we know it is generic enough to describe most distributed protocols. Finally, it allows a human expert to express the desired properties without knowing how the protocol works. For example, for the Atomic Commit protocol, which tries to ensure that all processes either all commit or all abort [16], a human expert can assign Commit and Abort to two states in the state machine (Figure 1a), and specify a rule stating that if one process reaches Commit, no other processes can reach Abort, and vice versa. This is the minimal a human expert needs to do to design a protocol. On the other hand, the AI model does not need to understand these meanings, can incorporate additional states that have no specific meanings assigned by the human expert (e.g., state 7), and can determine the conditions of transitions by itself.

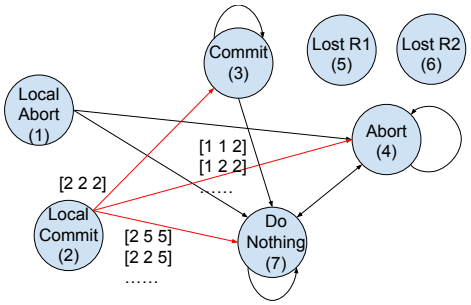
To simplify training without losing generality, we make several assumptions: we assume each state will include the output, so a process will broadcast its state to other processes. We assume the protocol works in multiple rounds. Before the first round, the state machine of each process will be given an initial state, set up by our training procedure (e.g., "Local Abort" and "Local Commit" in Figure 1a). At the beginning of each round, a process will broadcast its own state to every process including itself, while some of the messages may be lost due to process crashes (a lost state is represented as a special state as shown in Figure 1a). Then each process will apply its state machine, using the received states as the input, to determine the next state of the state machine. A process will broadcast the updated state in the next round. In this work, we assume all processes use the same state machine and only consider crash failures. We discuss alternatives in Section 3.

Without a prior knowledge of the protocol, we don't know the number of states or rounds required for the protocol, so our strategy is to start from smaller numbers of states and rounds and add more states and/or rounds if we cannot learn a correct state machine.

2.2 Training the AI model

At a high level, our training process is similar to that of AlphaGo Zero: we start from a randomly initialized neural network as the state machine, and generate the training set by randomly choosing initial states and lost messages; given the state machine, the initial states, and the lost messages, the whole protocol can "self play" as discussed in the prior section, in which each process will generate a sequence of states; finally we determine the rewards depending on whether these states meet the properties specified by the human expert and use the rewards to train the neural network.

For example, for the Atomic Commit protocol, we design our reward function as follows [8]. In one run of the protocol, a process's final decision is FINAL_COMMIT if it has reached COMMIT state but not ABORT state, is FINAL_ABORT if it has reached ABORT state but not COMMIT



(a) Learned state machine. For readability, it only shows conditions for a subset of transitions (red).

Layer Type	Size	Output
FC-1	3x128	1x128
FC-2	128x64	1x64
FC-3	64x16	1x16
Output	16x3	1x3

(b) The neural network architecture used to learn the state machine on the left.

Figure 1: Learning a state machine to solve Atomic Commit.

state, is FINAL_CONFLICTING if it has reached both COMMIT and ABORT states, and is FINAL_UNDECIDED if it has neither reached COMMIT state nor reached ABORT state.

1. If a process' decision is FINAL_CONFLICTING, we set the reward to be -2000.
2. If a process' initial input is LOCAL_ABORT and its decision is FINAL_COMMIT, we set the reward to be -2000.
3. If some processes' decisions are FINAL_COMMIT and some processes' decisions are FINAL_ABORT, we set the reward to be -2000.
4. If a process' initial input is LOCAL_ABORT and there are no lost states, we set the reward to be +100 if all processes' decisions are FINAL_ABORT and -2000 otherwise.
5. If the initial states of all processes are LOCAL_COMMIT and there are no lost states, we set the reward to be +100 if all processes' decisions are FINAL_COMMIT and -2000 otherwise.
6. If a process' decision is FINAL_UNDECIDED, we set the reward to be -0.5.

Rules 1-5 try to achieve correctness by giving large negative reward when correctness is violated. Rule 6 tries to encourage liveness. These reward values are set based on multiple attempts, and we will investigate the impact of different reward values in the future. We adopt the neural network based Q-table in deep q-learning algorithm [19] to map the input and action. Since the rewards can only be determined in the last round but the AI model needs to be applied at every round, we propagate the final rewards back from the last round to earlier rounds. Unlike AlphaGo Zero [27], which back propagates the final rewards by Monte-Carlo Tree Search algorithm to update the model, our final rewards are split equally as the training target for both rounds to update the model. Besides, we use ϵ-greedy search algorithm [24] to balance exploration and exploitation, while AlphaGo Zero uses upper confidence bound algorithm [2].

2.3 Validating the learned model

To prove a distributed algorithm correct, we must clarify what properties a correct solution should satisfy. For node $n \in N$, consider the local decision, b_n, that serve as the start states for the protocol; and the set of final decisions, F_n. In order to capture the possibility of one or more nodes losing a message, we also consider a set of all messages, M, that were distributed between the nodes during the execution of the protocol and the associated predicate $L = \exists m \in M(m = Lost)$. Based on Rules 1-5, we formally defined five required properties for an Atomic Commit protocol (see Appendix).

In order to prove that the learned model satisfies these properties, we implemented a verifier using the Z3 Theorem Prover [6]. Using the Z3py Python API we created a generic interface to prove properties of a distributed algorithm whose transitions through its state machine are defined arbitrarily.

The set of states that a node can be in at any given time is called S. For example, for Atomic Commit, S contains the states shown in Figure 1a. To extract the behavior of the model, our verifier generates an exhaustive list of transitions for the N nodes by evaluating the learned model over every $s \in S^N$, where $\delta(s)$ is the resulting state. For each transition that is made during the execution in the protocol,

the verifier then adds an assertion in the Z3 solver that a set of input variables equivalent to s implies that the resulting state must be equivalent to $\delta(s)$.

Our verifier then asserts against the solver that at least one of the five properties are **false**. Solving this system leveraging Z3 allows us to check the resultant model against our desired properties. If the system is `satisfiable`, we can confirm a model generated is an incorrect implementation of atomic commit as there exists an execution that violates the required properties. Following this logic, if the system is `unsatisfiable`, we conclude the algorithm is correct. Note that although these correctness properties are specific to the target protocol, the verifier is generic across different protocols and provides an interface for specifying the corresponding correctness properties.

2.4 Preliminary Results

We have applied our approach to learn a simplified Atomic Commit protocol, which does not consider recovery of failed processes. Figure 1b demonstrates the model architecture that we use. Our model contains three fully connected layers (FC) and one output layer. The input to the first FC layer is all messages that one process has received after one round of information exchange. We use ReLU as the activation function after each FC layer. The model will generate 3 values that represent expected rewards for taking each action. Our implementation is based on TF-Agents [9] and Python. We utilize a 16-core AMD 7302P CPU to conduct training process due to the simplicity of the model architecture. Learning rate is decaying from 0.01 to 0.001 during training. ϵ factor of exploration search algorithm is decayed gradually until reaches 0.1.

With around one hour of training, our approach can successfully learn a state machine for a 3-node Atomic Commit protocol, which can pass the model checking discussed in Section 2.3.

3 Future work

More complicated protocols. We will explore other protocols, like distributed locking [7], Paxos [13, 14], and Byzantine Fault Tolerance (BFT) [15]. For BFT, since a process can be malicious, we will allow different processes to use different state machines.

More processes. Training a protocol involving more processes will incur a higher computational overhead. Furthermore, people often expect a genetic protocol working for any number of processes. To address this challenge, we observe that the structure of a state machine (i.e., number of states and transition edges) usually won't change with the number of processes—the changing part is the condition of each transition edge. This observation may help us to extrapolate a larger or even genetic state machine from a number of smaller protocols. Prior works have shown this is promising [18, 34].

Properties other than correctness. We envision liveness is a challenging property to model because it cannot be checked in a finite number of rounds. We may tune the rewards to encourage the protocol to use fewer messages, use fewer rounds of message exchanges, or make a decision earlier, etc.

4 Related Work

Modern Reinforcement Learning (RL) algorithm combined with deep neural network has shown the ability to outperform a professional human in many fields such as Go [20], video games [20, 12], and real-world problems [31, 29]. Furthermore, a self-taught AlphaGo Zero program from DeepMind can outperform human Go champions without relying on human knowledge [28].

With our best effort, we do not find related work that uses AI techniques to learn a distributed protocol in general. Though, one work [10] utilizes a similar self-play reinforcement learning algorithm to learn a specific distributed directory protocol. They model the protocol into two competitive agents. One agent tries to learn a protocol while another agent tries to find challenging inputs for the learning agent. Our work considers more general scenarios, targeting generating distributed protocols with pre-defined objectives.

5 Conclusion

This work explores whether we can use techniques like AlphaGo Zero to learn a distributed protocol by self playing with zero knowledge from human experts. With our preliminary results, we discuss possible future directions for further improvement.

References

[1] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-scalable byzantine fault-tolerant services. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, page 59–74, New York, NY, USA, 2005. Association for Computing Machinery.

[2] Peter Auer. Using confidence bounds for exploitation-exploration trade-offs. *Journal of Machine Learning Research*, 3(Nov):397–422, 2002.

[3] 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.

[4] 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, page 251–264, USA, 2012. USENIX Association.

[5] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, page 177–190, USA, 2006. USENIX Association.

[6] 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*, pages 337–340. Springer, 2008.

[7] E.W. Dijkstra. Co-operating sequential processes. In *Programming languages : NATO Advanced Study Institute : lectures given at a three weeks Summer School held in Villard-le-Lans, 1966 / ed. by F. Genuys*, pages 43–112, United States, 1968. Academic Press Inc.

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

[9] Sergio Guadarrama, Anoop Korattikara, Oscar Ramirez, Pablo Castro, Ethan Holly, Sam Fishman, Ke Wang, Ekaterina Gonina, Neal Wu, Efi Kokiopoulou, Luciano Sbaiz, Jamie Smith, Gábor Bartók, Jesse Berent, Chris Harris, Vincent Vanhoucke, and Eugene Brevdo. TF-Agents: A library for reinforcement learning in tensorflow. `https://github.com/tensorflow/agents`, 2018. [Online; accessed 25-June-2019].

[10] 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)*, pages 501–505. IEEE, 2021.

[11] 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, page 45–58, New York, NY, USA, 2007. Association for Computing Machinery.

[12] 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.

[13] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, may 1998.

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

[15] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, jul 1982.

[16] Butler Lampson and Howard E Sturgis. Crash recovery in a distributed data storage system. 1979.

[17] 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)*, pages 399–414, Broomfield, CO, October 2014. USENIX Association.

[18] 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*, SOSP '19, page 370–384, New York, NY, USA, 2019. Association for Computing Machinery.

[19] 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.

[20] 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.

[21] 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, page 358–372, New York, NY, USA, 2013. Association for Computing Machinery.

[22] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. Consolidating concurrency control and consensus for commits under conflicts. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 517–532, Savannah, GA, November 2016. USENIX Association.

[23] 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, page 305–320, USA, 2014. USENIX Association.

[24] Avery Parkinson. The Epsilon-Greedy algorithm for reinforcement learning. `https://medium.com/analytics-vidhya/the-epsilon-greedy-algorithm-for-reinforcement-learning-5fe6f96dc870`, 2019.

[25] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, dec 1990.

[26] Rong Shi and Yang Wang. Cheap and available state machine replication. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 265–279, Denver, CO, June 2016. USENIX Association.

[27] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[28] 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.

[29] 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.

[30] Sarah Tollman, Seo Jin Park, and John Ousterhout. EPaxos revisited. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 613–632. USENIX Association, April 2021.

[31] 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.

[32] Dong Wang, Wensheng Dou, Yu Gao, Chenao Wu, Jun Wei, and Tao Huang. Model checking guided testing for distributed systems. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 127–143, New York, NY, USA, 2023. Association for Computing Machinery.

[33] Yang Wang, Lorenzo Alvisi, and Mike Dahlin. Gnothi: Separating data and metadata for efficient and available storage replication. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 413–424, Boston, MA, June 2012. USENIX Association.

[34] 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)*, pages 405–421. USENIX Association, July 2021.

[35] 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, page 263–278, New York, NY, USA, 2015. Association for Computing Machinery.

6 Appendix

The formally defined properties for an Atomic Commit protocol are shown below. We verified these properties using the Z3 Theorem Prover mentioned in Section 2.3.

P1. $\neg\exists n \in N((\exists f \in F_n(f = Commit)) \wedge (\exists f \in F_n(f = Abort)))$

P2. $\forall n \in N((b_n = LocalAbort) \implies \neg\exists f \in F_n(f = Commit))$

P3. $\neg\exists n, m \in N((\exists f_n \in F_n, f_m \in F_m((f_n = Commit \wedge f_m = Abort) \vee (f_n = Abort \wedge f_m = Commit))$

P4. $\neg L \implies \forall n \in N((b_n = LocalAbort) \implies \forall f \in F_n(f = Abort))$

P5. $\neg L \implies \forall n \in N((b_n = LocalCommit) \implies \forall f \in F_n(f = Commit))$