Automatic Generation of Distributed Algorithms with Generative AI

Diogo Vaz, David R. Matos, Miguel L. Pardal, Miguel Correia
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa – Lisbon, Portugal
{diogo.vaz, david.r.matos, miguel.pardal, miguel.p.correia}@tecnico.ulisboa.pt

***Abstract*—Fault-tolerant distributed algorithms such as Reliable Broadcast, Causal Broadcast, Total Order Broadcast, and Consensus, are at the core of many modern distributed systems. However, the development of distributed algorithms by humans is a laborious and complex process. This work presents a novel approach to generating distributed algorithms using Generative Artificial Intelligence that allows for automating the process of generating such algorithms. The paper also summarizes our initial results on using the approach to generate Reliable Broadcast algorithms.**

***Index Terms*—Fault-Tolerant Distributed Algorithms, Reliable Broadcast, Automatic Algorithm Generation, Generative AI, Reinforcement Learning, Automatic Algorithm Validation**

I. Introduction

Distributed systems are made up of several components that are connected by communication networks. During normal operation, some of these components may fail, for example, due to power outages, software defects, or malicious attacks. These flaws can affect the overall regular operation of the system. As a result, fault tolerance must be provided so that the distributed system can maintain normal operation even in the presence of failures.

For this reason, we need to design and implement *fault-tolerant distributed algorithms* [4], [9]. These algorithms have been widely studied over the years for different problems (Reliable Broadcast, Causal Broadcast, Total Order Broadcast and Consensus) and different faults (Crash, Omission, Byzantine). However, the process of creating a fault-tolerant distributed algorithm is a manual, time-consuming, and complex procedure, from the development of the algorithm itself to its validation. This is especially true when considering flaws, because the algorithms are sophisticated and small adjustments frequently impose a complete redesign. Furthermore, modern fault-tolerant algorithm research does not focus just on correctness: efficiency is also a major concern, increasing the complexity of the algorithm development process.

In this paper, we propose a new approach that aims to automatically *generate fault-tolerant distributed algorithms using machine learning*. More precisely, we intend to *use generative AI to generate fault-tolerant distributed algorithms*. Our claim is that by having a human input a description of the algorithm he needs, it is possible to obtain a correct and efficient algorithm. We believe that this can be the beginning of a new path of research in the distributed computing field that can get adoption in the next years.

Generative Artificial Intelligence (Generative AI) first appeared with *Generative Adversarial Networks* (GANs), a class of deep learning algorithms capable of generating new samples (e.g., images, videos) from examples [8]. More recently, the area evolved to the generation of text and code, with *ChatGPT*[1] and *GitHub CoPilot*[2]. ChatGPT, in particular, is gaining traction / adoption [19] due to its ability to generate text and answer questions in prose, poetry, or code.

We apply this idea of generative AI to the generation of distributed algorithms. In an attempt to experiment with current generative AI solutions to the task of generating fault-tolerant distributed algorithms, we have tested, on both ChatGPT (3.5 version) and GitHub CoPilot, the generation of a fault-tolerant Reliable Broadcast algorithm. Interestingly, when asked about Byzantine fault-tolerant Reliable Broadcast [9] algorithms, ChatGPT correctly mentioned the classical Bracha algorithm [3]. However, when asked to provide pseudo-code for such an algorithm, it provided an inefficient fault-tolerant broadcast algorithm or presented an algorithm that was not totally correct. On the GitHub Copilot test, we have not obtained any code for the defined problem.

It is possible to envisage various approaches to achieve this goal. In the paper, we present the first, based on *Reinforcement Learning* (RL), a form of learning based on experience – in this case, the experience of generating algorithms. Our approach follows an iterative process over two phases: *generation*, to obtain candidate algorithms, and *validation*, to evaluate their correctness. The process goes on discovering new algorithms and converging towards one that is correct and efficient in terms of a set of metrics. In this paper, we present a first application of the proposed approach to the generation of fault-tolerant Reliable Broadcast algorithms: a non-trivial distributed algorithm to which there are many solutions in the literature [2], [9].

Research on the automatic generation of algorithms has focused mainly on security protocols [20], [21]. For distributed algorithms, we identified a work that automatically synthesizes threshold-based distributed algorithms [15] and another that automatically investigates and validates consensus algorithms [23], both using brute-force approaches. These works are limited and are not based on any type of machine learning or AI. On the other hand, works based on AI to generate

[1]https://openai.com/blog/chatgpt/
[2]https://github.com/features/copilot

code have been concentrated on local, non-distributed, single-threaded code and mainly using supervising machine learning techniques [1], [17]. Using Reinforcement Learning, we have identified two interesting works: one that uses reinforcement learning to optimize models to generate code [16] and another that uses reinforcement learning to generate matrix multiplication algorithms [7]. However, the two works are very different from what we propose.

The rest of the paper is organized in the following way: Section II presents the scheme for generating distributed algorithms that utilize reinforcement learning. Section III summarizes our results on applying the approach to Reliable Broadcast. Section IV discusses future work, and Section V concludes the paper.

II. Generation of Distributed Algorithms

For the task of learning to solve a distributed problem, we decided to emulate the trial and error strategy used by human researchers to solve this problem. Consequently, we decided to use Reinforcement Learning [22]. Reinforcement Learning is a process based on experience in which an agent chooses actions in specified states and receives rewards depending on the choices. The states are the observations that the agent gets from its environment. How the agent acts is defined by the policy, which in this case is a map of the perceived states to the actions to be conducted in those situations. For each action chosen, the agent receives a reward reflecting its decision. With time, the agent will begin to understand the value of each state, which is the total amount of reward the agent expects to accrue in the future, beginning with that state. While rewards are short-term indications, values translate the state's long-term attractiveness, taking into consideration the states that are expected to follow, as well as the rewards related to them.

The reason for this method is that the agent will be capable of learning/generating a right and efficient algorithm by developing several algorithms, either accurate or erroneous, without prior knowledge of the state of the art of concrete solutions of distributed algorithms. Furthermore, other machine learning techniques such as supervised or unsupervised learning would require a dataset with distributed algorithms to train the agent. To our knowledge, such a dataset does not exist and would be laborious and complex to create.

As explained above, our approach iterates in two phases: generation and validation. The first phase, *generation*, corresponds to the problem of code/program generation, but with important distinctions, since distributed algorithms run in parallel across several nodes and are susceptible to errors, whereas local programs are not. For this phase, we have designed an agent that can construct correct and efficient algorithms based on the inputs provided by researchers, which include assumptions about the environment, faults to tolerate, and the features of the algorithm. Our system is designed to be adaptable, allowing researchers to tweak the specification and acquire new protocols.

The second phase, *validation*, corresponds to the problem of validating the algorithms generated. In this phase, our aim is to create a second agent capable of validating, in due time, the algorithm generated against the properties of the distributed problem to be solved. For this purpose, research on automatic validation of distributed algorithms has been using a set of model-checking languages and frameworks such as the TLA+ language and tools [13], the Spin framework with the PROMELA language [6] or the ByMC framework [12]. We have decided to use the Spin/PROMELA framework. This choice is based on the flexibility presented by the tool that allows different algorithms to be modeled, as well as the existence of extensive documentation and an active community.

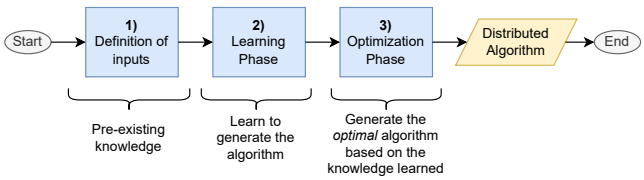

Fig. 1. Procedure of the generation of one algorithm.

III. Generating Reliable Broadcast Algorithms

We have explored the approach on a more simple distributed problem: Reliable Broadcast. A Reliable Broadcast algorithm ensures, essentially, that every message broadcasted by a correct process is eventually delivered by all correct processes. More specifically, the protocol is defined by the following properties [5], [9]:

- *RB-Agreement*: if a correct process delivers a message m, then all correct processes will eventually deliver the same message m;
- *RB-Validity*: if a correct process broadcasts a message m, then it will eventually deliver that message m;
- *RB-Integrity*: for any message m, every correct process delivers m at most once and only if m was previously broadcast by some correct process.

Next, we explain how reinforcement learning was applied to the generation of Reliable Broadcast algorithms.

The solution considers an *agent* that has the goal of generating correct and efficient Reliable Broadcast algorithms, that is, algorithms that satisfy the Reliable Broadcast properties and minimizes the efficiency metrics, such as the number of messages sent, the number of communication steps, and the number of messages needed to execute the algorithm. The architecture of the solution is made up of a main agent, *RB-Learner*, which collaborates with an auxiliary agent, *RB-Oracle*.

The entire procedure to generate a correct and efficient algorithm is represented in Figure 1. The procedure begins with the *definition of the inputs* for the problem to be solved. In this step, users give the agents some information and pre-existing knowledge. The *learning inputs* include the number of simulations and episodes to run. The *generation process inputs* include the rewards and heuristics to be used, both having domain knowledge about the problem [14]. The *validation process inputs* include the specifications to validate

the algorithm, such as the failure modes and their tolerance ratios, the number of nodes to model, and the properties to be validated.

After the definition of the inputs, the *learning phase* starts with the RB-Learner agent executing a set of *learning episodes*. In each episode, RB-Learner executes the *generation process*. During this process, the agent analyzes a group of *heuristics* [18], i.e. rules that discard invalid actions in specific states such as the impossibility of generating an empty algorithm. After the analysis, the agent selects and adds an *action*, i.e. representations of behavior such as sending or delivering messages, to the algorithm. A policy guides the selection of the action. In this work, the agent follows the *Upper Confidence Bound* (UCB) policy [22], a policy based on the idea of being *optimistic under uncertainty* For each added action, the agent receives a *reward* representing the cost of the action related to the efficiency of the action for the algorithm. For example, an action that instructs processes to send only one message will have a lower cost than an action that instructs processes to send messages to all the processes, since it represents a higher number of messages being sent during the execution of the algorithm. The generation process was implemented using the *Q-Learning* [22] reinforcement learning algorithm, a broadly used algorithm that employs a table designated *QTable* to map the values of each action to each state. When the algorithm is complete, the RB-Learner gives it to the RB-Oracle agent, which executes the *validation process*, i.e., assesses whether the generated algorithm actually solves the problem. In this process, the agent uses the *Spin* [10] framework and the *PROMELA* language. In essence, Spin simulates the execution of the created algorithm in a specific system model, performing an exhaustive exploration of the state space, and verifying if none of the three RBcast properties (RB-Agreement, RB-Validity, and RB-Integrity) is violated. Then, the RB-Oracle returns the validation result to the RB-Learner, which it uses as part of the generation process, i.e. RB-Leaner receives a final reward depending if the algorithm is correct or not.

When all learning episodes are executed, the learning phase ends, followed by the *optimization phase*. In this phase, the agent performs a last episode called *optimization episode*. The agent runs the *optimal generation process* in this episode to build the *optimal algorithm* based on the knowledge gathered from the previous episodes. Since the agent may not be able to learn to solve the problem, the validation procedure is repeated one more time to ensure that the optimal algorithm is correct. In the end, a correct and efficient algorithm is exported.

We conducted experiments with the objective of generating correct and efficient No-Failure tolerant, $\lfloor (N-1)/2 \rfloor$ Crash-Failure tolerant and $\lfloor (N-1)/3 \rfloor$ Byzantine-Failure tolerant algorithms. We executed a total of 12.000 generation episodes for each experiment. It took ± 9 hours to run the No-Failure experiment, ± 3 days to run the Crash-Failure experiment, and ± 7 days to run the Byzantine-Failure experiment. All tests were run on a Debian 10 machine with 32 vCPUs and 64 GB of memory.

Figure 2 presents the total number of algorithms generated in each test. As expected, with the increase in the complexity of the problem to solve – from a No-Failure tolerant to a Byzantine-Failure tolerant – the agent needs to generate and explore more algorithms to converge to a correct and efficient algorithm. In all the tests performed, our agent successfully generated an efficient and correct algorithm, equivalent to one already presented in the literature. In the No-Failure experiment, it converged to the algorithm presented in [4]. In the Crash-Failure experiment, it converged to the algorithm presented in [9]. In the Byzantine failure case, although the agent generated an algorithm equivalent to the one presented in [11], at the end, the agent converged to a new efficient algorithm presented in Figure 3. After an analysis, we concluded that this algorithm works only for $F = 1$ and $N \geq 4 \in \mathbb{N}$, where F is the maximum number of faulty processes in the system and N is the total number of processes in the system. Compared to the Imbs and Raynal algorithm [11], the new Figure 3 presents two improvements: (1) Figure 3 sends a total of $(N-1) + N(N-1)$ messages, instead of $N^2 + N$ messages sent by the algorithm in [11]; and (2) it only needs $F + 1$ messages to deliver, instead of $(N + F)/2$ needed by the algorithm in [11]. Since, for $F = 1$ and $N \geq 4 \in \mathbb{N}$, we have $(N+F)/2 > F+1$, then the new algorithm is also more efficient from a message delivery point of view.

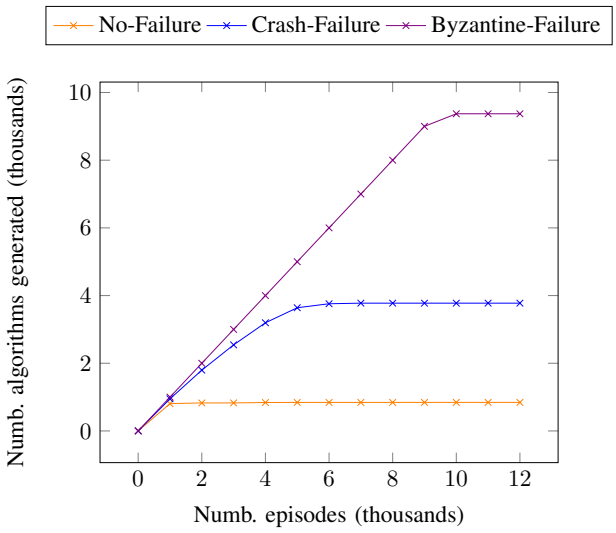

Fig. 2. Total number of algorithms generated in each experiment.

IV. Future Work

The use of machine learning techniques to generate new information/data – a field called *Generative AI* – is receiving attention these days, mainly due to products such as *GitHub CoPilot* and *ChatGPT*. Similarly to those products, in this work, we propose the use of reinforcement learning techniques to solve a specific but complex problem: to generate correct and efficient fault-tolerant distributed algorithms. This approach has already been explored with a distributed problem, the Reliable Broadcast problem, and, by the results obtained,

1: *when RB-Broadcast(m) do*:
2: SEND to neighbours(<*type0*,m>) if received (<*type0*,m>) from 0 distinct parties and not already sent;
3: STOP if received (<*type0*,m>) from 0 distinct parties;

4: *when receive(m) do*:
5: SEND to neighbours(<*type1*,m>) if received (<*type0*,m>) from 1 distinct parties and not already sent;
6: SEND to neighbours(<*type1*,m>) if received (<*type1*,m>) from $F + 1$ distinct parties and not already sent;
7: DELIVER(<m>) if received (<*type1*,m>) from $F + 1$ distinct parties and not already delivered;
8: STOP if received (<*type0*,m>) from 0 distinct parties;

Fig. 3. Generated Byzantine-tolerant algorithm.

we identify the potential and capacity of this strategy to solve different and more complex distributed algorithms. We believe that being this work the beginning of a new field of research in the distributed computing field, a research community can grow around the approach that we purpose, with the objective of increasing its capacity to solve more distributed problems, discovering new algorithms, and promoting research.

However, there are two main issues that will need to be addressed in the future: (1) with the increase in the complexity of the distributed problem to be solved, the complexity and, probably, the size of the algorithm to be generated will also increase. Therefore, this can result in a higher number of possible actions and states to be explored, which consequently will lead to an increase in the time needed to train the agent. To help address this issue, more computing resources will be needed to train the agent in due time. For example, for Reliable Broadcast, it was possible to obtain solutions in a single computer, whereas for other distributed algorithms (e.g., Consensus, Total Order Broadcast) we may need parallel/distributed computing. Another possible solution is to define new heuristics in order to reduce the number of states and algorithms to be explored; (2) the fact that the agent relies on inputs from previous works can bias the agent when finding distributed algorithms. To address this issue, it is important to decouple the agent from data based on previous work by decreasing the number of inputs necessary to be given to the agent. Also, new inputs can be created and added to the agent, e.g., thresholds that are not based on previous works, but can help to generate new algorithms.

V. Conclusion

Over the years, various problems and variants of fault-tolerant algorithms have been discussed. However, this study is intricate and has always been grounded in a human-oriented process. We suggest a method to automate this procedure based on Reinforcement Learning techniques that can generate correct and efficient algorithms. A first study with the proposed method successfully demonstrates the capacity of the approach in generating correct and efficient Reliable Broadcast algorithms. Based on the characteristics of the problem, the paper shows that the solution can produce accurate and effective algorithms, either already developed or new. To our knowledge, this is the first work to combine the fields of generation and validation into an autonomous process capable of producing accurate and effective distributed algorithms using machine learning.

Acknowledgment

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UIDB/50021/2020 (INESC-ID) and 2022.10788.BD.

References

[1] B. Aşıroğlu, B. R. Mete, E. Yıldız, Y. Nalçakan, A. Sezen, M. Dağtekin, and T. Ensari. Automatic HTML code generation from mock-up images using machine learning techniques. In *2019 Scientific Meeting on Electrical-Electronics & Biomedical Engineering and Computer Science*, pages 1–4, 2019.

[2] S. Bonomi, J. Decouchant, G. Farina, V. Rahli, and S. Tixeuil. Practical Byzantine reliable broadcast on partially connected networks. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 506–516, 2021.

[3] G. Bracha. An asynchronous [(n-1)/3]-resilient consensus protocol. In *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing*, pages 154–162, 1984.

[4] C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.

[5] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and efficient asynchronous broadcast protocols. In *Annual International Cryptology Conference*, pages 524–541. Springer, 2001.

[6] G. Delzanno, M. Tatarek, and R. Traverso. Model checking paxos in spin. *Electronic Proceedings in Theoretical Computer Science*, 161:131–146, aug 2014.

[7] A. Fawzi, M. Balog, A. Huang, T. Hubert, B. Romera-Paredes, M. Barekatain, A. Novikov, F. J. R Ruiz, J. Schrittwieser, G. Swirszcz, et al. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, 2022.

[8] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial networks. *Communications of the ACM*, 63(11):139–144, 2020.

[9] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Cornell University, Department of Computer Science, May 1994.

[10] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[11] D. Imbs and M. Raynal. Simple and efficient reliable broadcast in the presence of Byzantine processes. *arXiv preprint arXiv:1510.06882*, 2015.

[12] I. Konnov and J. Widder. ByMC: Byzantine model checker. In *International Symposium on Leveraging Applications of Formal Methods*, pages 327–342. Springer, 2018.

[13] L. Lamport and S. Merz. Specifying and verifying fault-tolerant systems. In *Formal Techniques in real-time and fault-tolerant systems*, pages 41–76. Springer, 1994.

[14] A. D. Laud. *Theory and application of reward shaping in reinforcement learning*. University of Illinois at Urbana-Champaign, 2004.

[15] M. Lazic, I. Konnov, J. Widder, and R. Bloem. Synthesis of distributed algorithms with parameterized threshold guards. In *21st International Conference on Principles of Distributed Systems (OPODIS 2017)*, 2018.

[16] H. Le, Y. Wang, A. D. Gotmare, S. Savarese, and S. C. Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *arXiv preprint arXiv:2207.01780*, 2022.

[17] T. H. Le, H. Chen, and M. A. Babar. Deep learning for source code modeling and generation: Models, applications, and challenges. *ACM Computing Surveys*, 53(3):1–38, 2020.

[18] D. B. Lenat. The nature of heuristics. *Artificial Intelligence*, 19(2):189–249, 1982.

[19] D. Milmo and agency. ChatGPT reaches 100 million users two months after launch, Feb 2023. https://www.theguardian.com/technology/2023/feb/02/ chatgpt-100-million-users-open-ai-fastest-growing-app - *The Guardian*, Feb 2023.

[20] A. Perrig and D. Song. Looking for diamonds in the desert - extending automatic protocol generation to three-party authentication and key agreement protocols. In *Proceedings 13th IEEE Computer Security Foundations Workshop*, pages 64–76. IEEE Comput. Soc, 2000.

[21] D. Song, A. Perrig, and D. Phan. AGVI — Automatic Generation, Verification, and Implementation of Security Protocols. In *Computer Aided Verification*, pages 241–245. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. Series Title: Lecture Notes in Computer Science.

[22] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2018.

[23] P. Zielinski. Automatic verification and discovery of byzantine consensus protocols. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 72–81. IEEE, 2007.