Concurrency Testing of PBFT

How do different exploration strategies perform for detecting concurrency bugs in PBFT?

Martin Petrov

Supervisors: Burcu Kulahcioglu Ozkan, Ege Berkay Gulcan

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 25, 2023

Name of the student: Martin Petrov
Final project course: CSE3000 Research Project
Thesis committee: Burcu Kulahcioglu Ozkan, Ege Berkay Gulcan, Johan Pouwelse

Abstract

Consensus algorithms, as well as distributed systems in general, are vulnerable to concurrency bugs due to non-determinism. Such bugs are hard to detect since it is necessary to test using a lot of different scenarios and even then, there is no guarantee to find one.

Controlled concurrency testing is a proposed solution to that problem. Its main strength is allowing for more control over the order in which threads are executed, using controlled scheduling [1].

In this paper, we utilize Coyote, a concurrency testing framework, to test for concurrency bugs in PBFT, the seminal consensus algorithm. Random Walk, Delay-Bounding, Probabilistic Concurrency Testing (PCT) and Q-Learning are the exploration strategies we performed experiments with. The goal is to find schedules that lead to concurrency bugs. We test for 2 bugs that are seeded by us into the algorithm.

A comparison of the results shows that fair exploration strategies outperform their unfair variations. Finally, we conclude that PCT and fair PCT are the best-performing strategies for the benchmarks we use.

1 Introduction

As our world becomes more and more reliant on distributed systems and the internet, ensuring the reliability and security of these systems is of increasing importance. Consensus algorithms are one of the fundamental building blocks of today's distributed systems. They are being used for achieving agreement between nodes and ensuring a consistent view of the system state.

PBFT, short for Practical Byzantine-Fault Tolerance, is an example of such algorithm. It is considered seminal for consensus algorithms since previous works were either relying on synchrony or were not efficient enough to be considered practical [2]. The creation of the PBFT algorithm has inspired further research into the field of Byzantine-Fault tolerance algorithms and the development of improvements and different variations of PBFT, such as Aardvark [3] and Zyzzyva [4].

Even though the algorithm provides benefits such as resilience against malicious attacks and high throughput, it is prone to concurrency related issues, such as dealing with multiple requests at the same time, message orderings and others. Therefore, concurrency testing is a critical component for guaranteeing the correct operation and safety of an algorithm. It consists of testing the algorithm on many different concurrency scenarios to ensure proper execution in an asynchronous setting. It is crucial to test for since concurrency introduces non-determinism, which leads to harder detection of bugs and dealing with them[5].

Being able to find a concurrency bug is considered hard, but being able to reproduce it is even harder. The ability to replay a bug reduces the complexity of debugging and taking care of it. CCT, short for Controlled concurrency testing, is the solution we inspect. It utilizes different exploration schedulers and allows for better management of the exponential space of possible orderings of thread executions. The exploration strategies give control to the developers to create less likely executions and thus test a specific schedule they suspect of leading to a bug.

Conducting research on different concurrency testing strategies is important since it will contribute to the development of more robust and reliable consensus algorithms. The strategies we are performing the experiments with are Random Walk, Delay-Bounding, PCT and Q-Learning. The research will also provide insights into the challenges and opportunities of testing distributed consensus algorithms, which can aid the development of testing strategies in the future.

The main research question is the following:

How do different exploration strategies perform for detecting concurrency bugs in PBFT?

The main question can be divided into the following subquestions:

1. Which exploration algorithm performs the best for detecting concurrency bugs?
2. Which exploration algorithm performs the fastest detection of a concurrency bug?
3. How effective is the random scheduler in comparison to the other techniques?
4. Does the number of iterations correlate to the amount of bugs found when using QL?
5. How do fair exploration techniques compare to unfair ones?

The rest of the paper is structured as follows: we go through the required background knowledge in section 2. Section 3 is about the methodology, containing detailed information of the steps taken to conduct the research. Then we continue with the Experimental Setup in section 4, which provides details about the hardware and the software used for the experiments and the configurations of the strategies. After that, section 5 refers to the results of the experiments and provides an analysis of them. Section 6 focuses on the ethical aspects of conducting this paper. Finally, in section 7 we provide conclusions of our research, a discussion about the limitations, and detailed recommendations regarding possible improvements and further research into the field.

2 Background

This section provides the relevant background knowledge of the consensus algorithm and the exploration strategies, needed to understand the methodology and the results of the research. It also introduces the notation which is used in the rest of the paper. Details of the algorithm which are not relevant to the experiments and results of the paper are omitted.

2.1 Practical Byzantine Fault Tolerance

Figure 1 displays an overview of how PBFT works in theory. The algorithm is designed to operate as a state machine, which is replicated across different operating nodes [2]. The replicated nodes are called *replicas* and are identified with

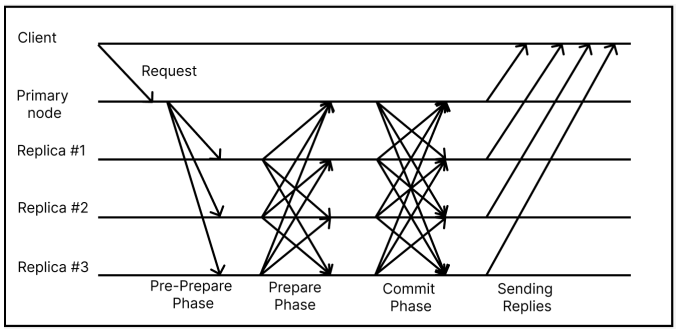

Figure 1: A normal case operation of PBFT

the integers between 0 and *R*, where *R* is the total amount of replicas. The algorithm works with the so-called *views*, which are configurations of replicas. Each view sets its own primary node. The primary node is elected via the formula $p = v \bmod R$, where *p* is the replica node and *v* is the view number. It has the task of multicasting the client's request to the other replicas. If it is deemed faulty, the client can request a view change which would lead to a change of the primary node.

The algorithm starts with the client sending a request to the primary node. When the request is received, the primary node starts a three-phase protocol of multicasting the request across all replicas. The phases are *pre-prepare*, *prepare* and *commit*.

The *pre-prepare* phase consists of the primary sending a pre-prepare message to all replicas. The message consists of the client's request message, a sequence number *n*, the digest of the client request and the view number. The digest is an output of a hashing function, but we will not focus on it since it is not relevant for concurrency testing. The last three are signed by the primary before being sent. When a backup receives a pre-prepare message, it accepts it if:

1. it is in the same view;
2. it has not previously accepted a pre-prepare message for the same view and sequence number but with a different digest;
3. the signatures in the request and the pre-prepare message are correct;
4. and the sequence number is between h and H (h and H are a lower and an upper bound, which are predefined)

If a replica accepts the pre-prepare message, it enters the *prepare* phase. It does that by multicasting a prepare message to all other backups and adding both messages to its logs. The prepare message consists of the view number, the sequence number and the digest from the pre-prepare message. A replica accepts such message if:

1. it is in the same view as the one in the message;
2. the signatures are correct;
3. and the sequence number is between h and H

We define the *prepared* predicate to be true in the following case: in the execution of a cluster of size $3f + 1$, once a node has in its logs a client's request message, a pre-prepare message in view v for it, and $2f$ prepare messages from different replicas, that match the pre-prepare message, the *prepared* predicate for it is set to true. Once the predicate is true, it can enter the *commit* phase. It does that by multicasting a commit message to the other replicas. The message consists of the sequence number, view number and the digest of the client's request message. The replicas accept the commit messages using the same conditions as the prepare message ones - correct signatures, view number and sequence number.

We define the *committed-local* predicate to be true in the following case: When a node has received $2f + 1$ commit messages from distinct nodes that match the pre-prepare of the client's request message, that is, having the same view number, sequence number, and digest, and has satisfied the *prepared* predicate, the *comitted-local* for it is set to true. Once it is set to true, it can start preparing for sending a reply back to the client. That is done by executing the operation requested by the client and including the result of it in a reply message, together with its view number.

Once the client receives $f + 1$ replies with valid signatures and matching results from different replicas, it accepts the result from the reply messages. The algorithm works on the assumption that the maximum amount of faulty replicas is f.

2.2 Controlled Concurrency Testing and Exploration Strategies

The idea behind controlled concurrency testing is to enable test reproducibility and therefore provide more control over the executions of tests. That is needed because concurrency bugs often lead to flaky tests and it is important to track down the rare schedules leading to them. Having the ability to replay them allows developers to have a better understanding of the implementation and what is going wrong with it.

Controlled concurrency testing utilizes exploration strategies to search through the exponentially many interleavings of a program's execution [5]. In this paper we are going to utilize the following exploration strategies:

Random Walk

The random walk strategy is a scheduler that randomly decides the next action to be performed from the set of all possible actions. It is usually used as a baseline for other strategies [6].

QL

QL [7], short for Q-Learning, is an exploration strategy based on reinforcement learning. It requires a fingerprint, which is a unique hash of the program state, used to distinguish the states. QL aims to learn a model that maximizes the unique fingerprints in a test run [5], which leads to exploring more program states and thus maximizing the coverage.

PCT

PCT is utilizing a priority-based exploration strategy. The idea behind it is to assign a randomly-generated priority to each process, which is used to decide on the next action at a scheduling point. PCT has two parameters that can be configured - *d* and *n*. The first parameter is the bug depth - the number of times PCT lowers the priority of the operation with

the highest priority to the lowest possible. These scheduling points are uniformly spread throughout the whole iteration. The second parameter, n, is the amount of steps to be explored throughout an iteration [5]. We denote the algorithm as PCT-X, where X is the value of the bug depth.

Delay-Bounding

Delay-Bounding [8] is an exploration strategy which utilizes the idea of having delays in the execution of tasks. The non-delayed tasks are performed until they are completed and only then the delayed ones are scheduled. The delays, which have an upper bound of d, can be seen as *deviations* from a deterministic schedule. We denote the algorithm as DB-X, where X is the value of the delay.

Fair and unfair exploration strategies

Madanlal Musuvathi and Shaz Qadeer introduce the concept of fair and unfair schedules [9]. Unfair schedules are characterized by starving threads of execution for an infinitely long period of time. They are not realistic to occur nowadays but can be really helpful when testing for concurrency bugs. Fair schedules on the other hand do not produce such scenarios and are therefore more close to a real-world scenario. That's why it is deemed important to explore the differences in testing for concurrency with fair and unfair strategies. Out of the aforementioned strategies, only PCT and DB incorporate fair and unfair variations. We denote the fair version of PCT with F-PCT and the fair version of DB with F-DB. The denotations we use for the unfair variations are the normal ones - PCT and DB.

3 Methodology

This section contains information about the methodology of implementing PBFT and conducting controlled concurrency testing on it. It gives insight on the decisions that have been made on selecting the framework for testing and implementation, details regarding our version of the algorithm, and explanations of the seeded bugs.

3.1 Framework Selection

The Coyote framework has been chosen as the primary tool for implementing PBFT and conducting concurrency testing. It provides a powerful in-memory and state machine programming model which allows programming at a high level of abstraction. It also provides a rich set of testing features, including exploration strategies, test reproducibility, control over specific explorations, such as deadlocks, atomic races, etc., thus making it suitable for our purposes.

3.2 PBFT Implementation

We implement PBFT using the Coyote framework. The Actor model of Coyote is used to represent the nodes in the network and messages between them are exchanged via the *sendEvent* function.

The implementation aims to capture the essential aspects of the consensus algorithm while simplifying its complexity for testing purposes. It includes the necessary components, such as the replica nodes, message exchange protocols, and the Byzantine fault tolerance mechanisms.

Hashing of the requests and digests is mocked by simple function calls which basically return the input. Timestamps are not used and the results of the "operations" requested by the client are also mocked.

3.3 Concurrency Bug Seeding

In order to test the performance of the different exploration strategies, we seed concurrency bugs within our implementation of PBFT. These bugs are designed to simulate concurrency issues that can occur in distributed systems. We consider the following bugs:

1. First benchmark

   The first bug we implement reveals itself in the prepare phase of the algorithm - since our implementation supports multiple client requests, we implement a dictionary data structure that maps the digest of a client's request to a boolean, stating whether the replica satisfies the *prepared* predicate. In our initial implementation, when a replica receives a pre-prepare request, it adds the digest of the replica as a key and sets the value to false. When a replica receives a commit message, it checks whether the predicate is true in the dictionary. The Heisenbug occurs in the rare cases in which a node receives a commit message by another replica before processing a pre-prepare message for its digest. In these cases, the dictionary does not contain the key and throws an exception when trying to fetch the value of a non-existent key from it. The bug can be easily avoided by just checking if the dictionary contains such key beforehand. It is a good candidate for a concurrency bug since it occurs in rare scenarios and is therefore used as a part of a faulty implementation of PBFT.

2. Second benchmark

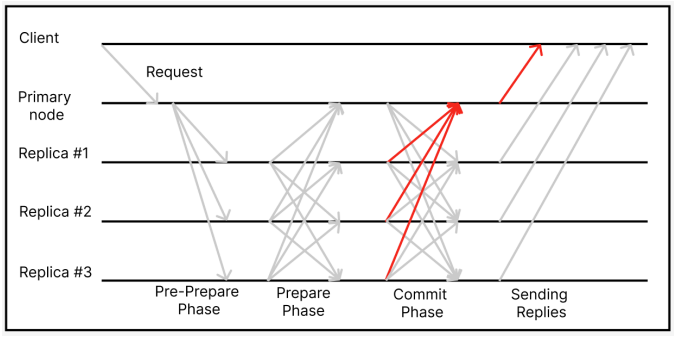

Figure 2: Faulty case of PBFT with only f nodes needed for majority when accepting a reply message

   The second bug is related to a change in the number of nodes needed for reaching a majority for consensus. The bug could be seeded in a lot of places in the implementation - when checking the *prepared* predicate, the *committed-local* predicate, and when the client is receiving reply messages. It was arbitrarily decided to seed the bug in the reply phase. As stated in the background, the client waits for $f + 1$ replies with valid signatures from different replicas before accepting a result of a request.

We change the number needed for majority from $f+1$ to f. This allows for concurrency bugs, which occur only when the f faulty nodes are the first group of nodes with the same signatures to send their replies to the client.

Figure 2 displays an example of the bug, with a client and 4 nodes, the primary node being the only faulty one, thus $f = 1$. Exchanged messages which are not important for the execution are left in a gray color. We inspect the case in which the primary node is the first to receive $2f + 1$ commit messages and the first to send a reply message to the client. Assuming that the result of the performed operation is faulty, the client accepts it, since it waits for f replies with the valid signatures, instead of $f + 1$.

3.4 Exploration Strategies

The Coyote framework provides the four exploration strategy mentioned in section 2 - RW, QL, PCT and DB. These exploration strategies determine how Coyote explores the execution space of the PBFT implementation during testing. After performing the experiments for the two bugs, their performance is compared.

4 Experimental Setup

This section introduces the specifications of the machine used to perform the experiments and gives detailed information about the configurations used for the exploration strategies when performing the tests.

4.1 Hardware and Software Specifications

The specifications of the hardware on which the experiments are performed are of the utmost importance for the validity and potential reproducibility of the results. The experiments used for obtaining the results in this paper were run on a machine, containing a 6-core Intel i7-9750H processor and 16 gigabytes of RAM. The machine runs Windows 10 as an operating system.

The implementation is built utilizing C#, version 7.0, and Coyote, version 1.7.9. The actual implementation has been made publicly available in a github repository[1].

4.2 Performance Metrics and Experiments

In order to evaluate the exploration strategies the following simple but effective metrics are used - the amount of bugs found for a number of test iterations, and the minimal amount of iterations needed to find a bug. Exploration strategies are compared to each other and the one which leads to the biggest amount of bugs found is considered the best.

All exploration strategies but QL run 100 iterations each, since their exploration runs are independent of each other. QL on the other hand is learning a model which maximizes the unique fingerprints. Therefore, we decided to run experiments with 100, 200, 500, 1000, and 2000 iterations in order to gather data, used for observing whether bug detection improves with increasing the iterations, answering the fourth subquestion stated in section 1.

[1]https://github.com/MartinPetrov12/Concurrency-Testing-Of-PBFT.git

The experiments are run with 5 actors - 1 client, 1 primary and 3 replica nodes. We also run the experiments with 1 and with 2 requests from the client. The sum of all messages exchanged between them comes to 32 for each request from the client. Therefore, the values for the bug depth of PCT and the delays of DB were kept under 32. We test the algorithms with values 2, 5, 7 and 10, with 10 being the default value for PCT and DB in the implementations of the exploration strategies Coyote incorporates. The bounded strategies are run three to four times with each parameter and the averages are taken for them to reduce the chance of abnormal test runs.

5 Results and Analysis

In this section we present and analyse the results obtained from the experiments with the seeded bugs and the different exploration strategies. As a result, answers to the subquestions of the main research question are given.

5.1 QL Analysis

Table 1: Comparison of QL explorations with different amounts of iterations. X axis - number of iterations, Y axis - bug. The value in each cell represents the division of the amount of found bugs and the number of iterations

	100	200	500	1000	2000
Bug #1 1 request	0.018	0.0195	0.0185	0.0152	**0.0204**
Bug #1 2 requests	0.009	0.015	0.0134	0.0167	**0.0196**
Bug #2 1 request	**0.343**	0.312	0.305	0.289	0.303
Bug #2 2 requests	0.496	**0.5135**	0.475	0.474	0.507

QL is the only exploration algorithm out of the ones we look into whose test runs are dependent of the number of iterations - as stated in section 2.2, it is a learning-based strategy which aims for maximizing the unique fingerprints in a program under test. That being said, one would think that increasing the coverage in a test run would lead to finding more bugs per iteration. In order to explore that, we define the E/I ratio as the number of bugs divided by the number of iterations (errors to iterations) for a test run.

The obtained results for the two benchmarks are conflicting with eachother. A clear uptrend can be identified in both versions of the first benchmark - with an increase of the iterations there is also an increase in the E/I ratio. That however is not the case for the second benchmark - the one request version running 100 iterations is finding 10% more bugs on average in comparison to the other QL explorations. The two request version running with 200 iterations outperforms the 500 and 100 iteration runs with 7.5% and is slightly better than the 2000 iteration run.

From our experience, it is observed that QL performs better when it uses more iterations for bugs that are occurring rarely but the same can not be concluded for more frequent bugs.

Table 2: PCT with various bug depths

	PCT-D-2	PCT-D-5	PCT-D-7	PCT-D-10
Bug #1 1 request	6	**8**	**8**	**8**
Bug #1 2 requests	9	10	11	**12**
Bug #2 1 request	31	**36**	26	24
Bug #2 2 requests	43	**47**	**47**	46

Table 3: F-PCT with various bug depths

	F-PCT-D-2	F-PCT-D-5	F-PCT-D-7	F-PCT-10
Bug #1 1 request	4	10	10	**11**
Bug #1 2 requests	7	11	**12**	**12**
Bug #2 1 request	**38**	24	28	28
Bug #2 2 requests	44	40	50	**54**

5.2 Fair and Unfair Strategies

The PCT exploration strategy does not perform as well when it uses unfair scheduling as opposed to when it uses a fair one. For every variation of the benchmarks the best-performing fair exploration finds more bugs than the best-performing unfair one. When it comes to the delay parameter, PCT performs the best with bug depth of 5, and F-PCT performs the best with a bug depth of 10. Only for the one request variation of the second benchmark, F-PCT performed the best with a value of 2.

Table 4: DB with various delay values

	DB-D-2	DB-D-5	DB-D-7	DB-D-10
Bug #1 1 request	2	4	**5**	1
Bug #1 2 requests	2	2	**4**	2
Bug #2 1 request	9	**17**	15	**17**
Bug #2 2 requests	38	43	43	**48**

Table 5: Fair-DB vs Fair-DB with specified delay

	F-DB-2	F-DB-D-5	F-DB-D-7	F-DB-D-10
Bug #1 1 request	2	3	0	**5**
Bug #1 2 requests	2	**4**	1	3
Bug #2 1 request	8	12	20	**21**
Bug #2 2 requests	40	46	47	**52**

When experimenting with DB and F-DB we reach the same conclusion as with PCT - the best-performing fair exploration strategy outperforms the best-performing unfair one for each bug and their variations. The best values for the delay parameter are in the range between 7 and 10.

Overall, we can conclude that fair exploration strategies are more suitable for testing our benchmarks in comparison to unfair ones.

5.3 Fastest Bug Detection

Table 6: Minimal amount of test iterations needed to find a bug

	RW	PCT	F-PCT	DB	F-DB	QL
Bug #1 1 request	11	**4**	**4**	82	5	56
Bug #1 2 requests	17	31	**11**	28	12	16
Bug #2 1 request	**3**	8	4	5	4	**3**
Bug #2 2 requests	**2**	**2**	3	5	6	7

When performing the experiment runs with the different exploration strategies it is also kept track of the first iteration in which a bug is found. For both variations of the first benchmark F-PCT has been the fastest. For the second benchmark however the results are pretty close due to the fact that bugs are found more frequently in it. That being said, F-PCT is the third-best-performing exploration strategy out of all for both variations, with PCT being tied for first for the 2 request variation. Since PCT is also tied for first with F-PCT in the first benchmark, it can be concluded that a combination of running PCT and F-PCT would be the fastest to find a bug for our benchmarks.

5.4 Best Performing Strategy

Table 7: Amount of found bugs after 100 iterations. Parameters used: PCT - 5, F-PCT - 10, DB - 10, F-DB - 10

	RW	PCT	F-PCT	DB	F-DB	QL
Bug #1 1 request	1	8	**11**	1	5	2
Bug #1 2 requests	2	10	**12**	2	3	2
Bug #2 1 request	31	**36**	28	17	21	31
Bug #2 2 requests	44	47	**54**	48	52	49

When performing the final evaluation, the best-performing variations of each algorithm were picked - PCT with a bug depth parameter of 5, F-PCT with 10, and both delay-bounding strategies with a delay of 10. Best-performing is evaluated by the variation which is the first for finding the most bugs among all other variations, for all bugs. The values for QL are equal to the average E/I for each bug from table 1, multiplied by 100. We use the Random walk strategy as a baseline for evaluating all other strategies.

Analysing the results for the first benchmark, it is observed that most algorithms are barely better than the baseline. DB and QL do not perform particularly well for it - for the 1 request version, DB has performed the same as RW and QL is better by just one more found bug. Using the fair version of DB is the third best strategy for it, however, PCT and F-PCT are the clear best strategies for this benchmark. That can also be seen in the 2 request version, where they have performed at least three times better than the third-best strategy. In both variations of the bug, F-PCT is better than PCT with at least 20% so it can be concluded that it is the best strategy for the first bug.

From analysing the results of the second benchmark, we observe that the random strategy is performing surprisingly well - it is tied for the second best-performing strategy for the 1 request variation. DB and fair DB are worse than the baseline level. F-PCT with a bug depth of 10 is not better than the baseline. However, when run with a bug depth of 2, F-PCT is the strategy with the most found bugs. For the 2 request variation of the benchmark, all strategies are performing better than the baseline, with F-PCT and F-DB being the only strategies able to find more than 50 bugs in 100 iterations. It can be concluded that F-PCT is the best strategy out of all for our benchmarks.

5.5 Conclusion of the Analysis

From the performed experiments, it can be concluded that PCT and F-PCT are performing the best from all strategies for our benchmarks, both in terms of finding bugs and the iterations needed to detect a bug. Different bug depth values are better for the different bugs though - bug depth of 2 for the fair version of PCT is outperforming the other ones for the 1 request variation on of the second benchmark, and a bug depth of 10 performs best for the rest. With including more bugs it would become harder to pick a single value for the parameter. Also it is not guaranteed that PCT and F-PCT are going to remain the best-performing strategies. Therefore when testing for concurrency bugs in PBFT we recommend focusing on building a portfolio of exploration strategies instead of picking a single one.

6 Responsible Research

The most important aspect when it comes to responsible research in this paper is the ability to reproduce the results. We provide detailed information about the hardware and software environment in section 4.1. By making the repository publicly available we allow anyone who wishes to perform the experiments themselves to do so.

We run the bounded strategies between three and four times each for all parameters and took the average of the results to attempt to reduce the randomness effect. However, we run the tests with just 4 values for their parameters. Since we do not focus on optimizing the parameters due to a lack of time, it is possible that using different values could lead to different results and conclusions. That is why we encourage people willing to run the experiments to build a portfolio of exploration strategies with various configurations and experiment with it, instead of taking our conclusions and directly applying them.

7 Conclusions and Future Work

This paper compared different exploration strategies for concurrency testing on a version of the Practical Byzantine Fault Tolerance algorithm. We found that for our seeded bugs, fair exploration strategies performed better than unfair ones, and that a combination of PCT and F-PCT would lead to the best results, both in terms of number of bugs found and in terms of iterations needed of finding a bug. The parameters of the bounded algorithms proved to be important during testing and we recommend building a portfolio of different exploration strategies with different configurations when performing tests on PBFT.

It is important to acknowledge the limitations of this study. Our implementation of PBFT does not fully capture the complexity of a real-world implementation - we have not performed tests that include a *view change* in the sequence of operations and we have not tested for more than one bug in a single test run - the bugs were seeded separately from each other. Also, the concurrency bugs we have tested for are not enough to represent the entire range of potential bugs that can arise in the implementation of the consensus algorithm.

Possible improvements would be:

1. Seeding more complex bugs and performing further research on the differences between fair and unfair strategies in terms of performance and possible use cases. Adding more bugs would also provide more insight into the question regarding the correlation between the number of iterations in a test and bugs found when using QL.
2. Incorporating delays in the form of sleeping a thread in the implementation - that would improve the already created bugs by simulating an asynchronous environment while testing.
3. More experiments with more parameters for the bounded strategies - that will increase the reliability of the results and will possibly lead to finding more bugs.

References

[1] C. Wen, M. He, B. Wu, Z. Xu, and S. Qin, "Controlled concurrency testing via periodical scheduling," in *Proceedings of the 44th International Conference on Software Engineering*, pp. 474–486, 2022.

[2] M. Castro, B. Liskov, *et al.*, "Practical byzantine fault tolerance," in *OsDI*, vol. 99, pp. 173–186, 1999.

[3] A. Clement, E. Wong, L. Alvisi, M. Dahlin, M. Marchetti, *et al.*, "Making byzantine fault tolerant systems tolerate byzantine faults," in *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, The USENIX Association, 2009.

[4] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: speculative byzantine fault tolerance," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pp. 45–58, 2007.

[5] P. Deligiannis, A. Senthilnathan, F. Nayyar, C. Lovett, and A. Lal, "Industrial-strength controlled concurrency

testing for c# programs with coyote," in *Tools and Algorithms for the Construction and Analysis of Systems: 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22–27, 2023, Proceedings, Part II*, pp. 433–452, Springer, 2023.

[6] P. Thomson, A. F. Donaldson, and A. Betts, "Concurrency testing using controlled schedulers: An empirical study," *ACM Transactions on Parallel Computing (TOPC)*, vol. 2, no. 4, pp. 1–37, 2016.

[7] S. Mukherjee, P. Deligiannis, A. Biswas, and A. Lal, "Learning-based controlled concurrency testing.," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 230–1, 2020.

[8] M. Emmi, S. Qadeer, and Z. Rakamarić, "Delay-bounded scheduling," in *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages*, pp. 411–422, 2011.

[9] M. Musuvathi, S. Qadeer, and M. Musuvathi, "Fair stateless model checking," in *PLDI 08: Programming Language Design and Implementation*, Association for Computing Machinery, Inc., June 2008.