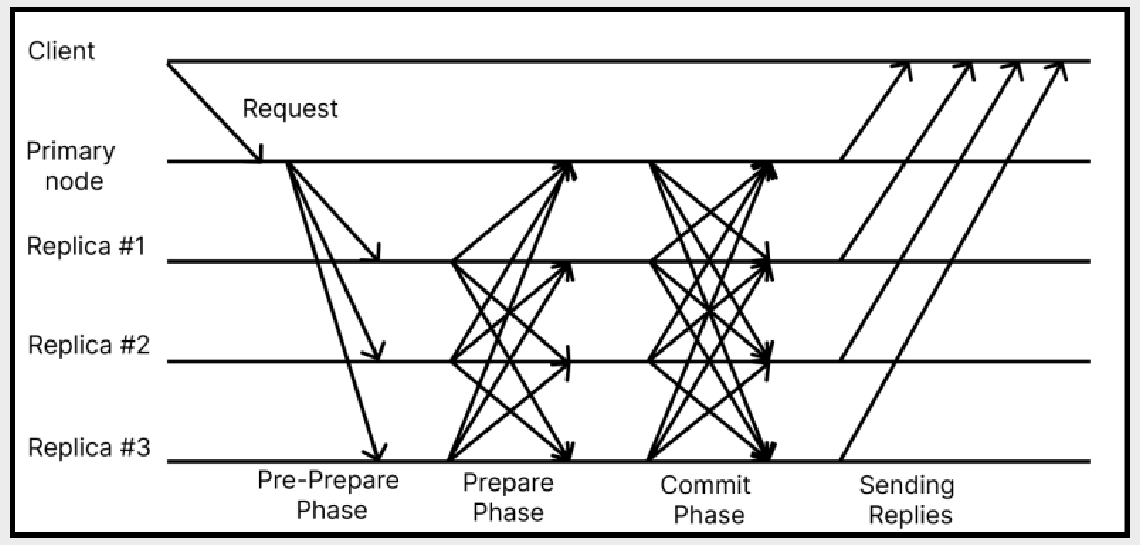
How do different exploration strategies perform for detecting concurrency bugs in PBFT?

Martin Petrov
M.A.Petrov@student.tudelft.nl

Burcu Kulahcioglu Ozkan
Ege Berkay Gulcan

1. Background

1. Practical byzantine Fault Tolerance (PBFT)
- Seminal consensus algorithm
- Tolerates *f* faulty nodes of total 3f+1 nodes
- Consists of three phases - Pre-Prepare, Prepare and Commit



2. Controlled Concurrency Testing - allows for control over over the order in which threads are executed

3. Exploration strategies:
- Random Walk - randomly chooses next action to be performed. Used as a baseline
- PCT - randomly assigns priorities to tasks
- Delay-Bounding - applies delays to some tasks
- Q-Learning - utilizes reinforcement learning, maximizes the unique fingerprints in a test run

2. Research questions:

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?

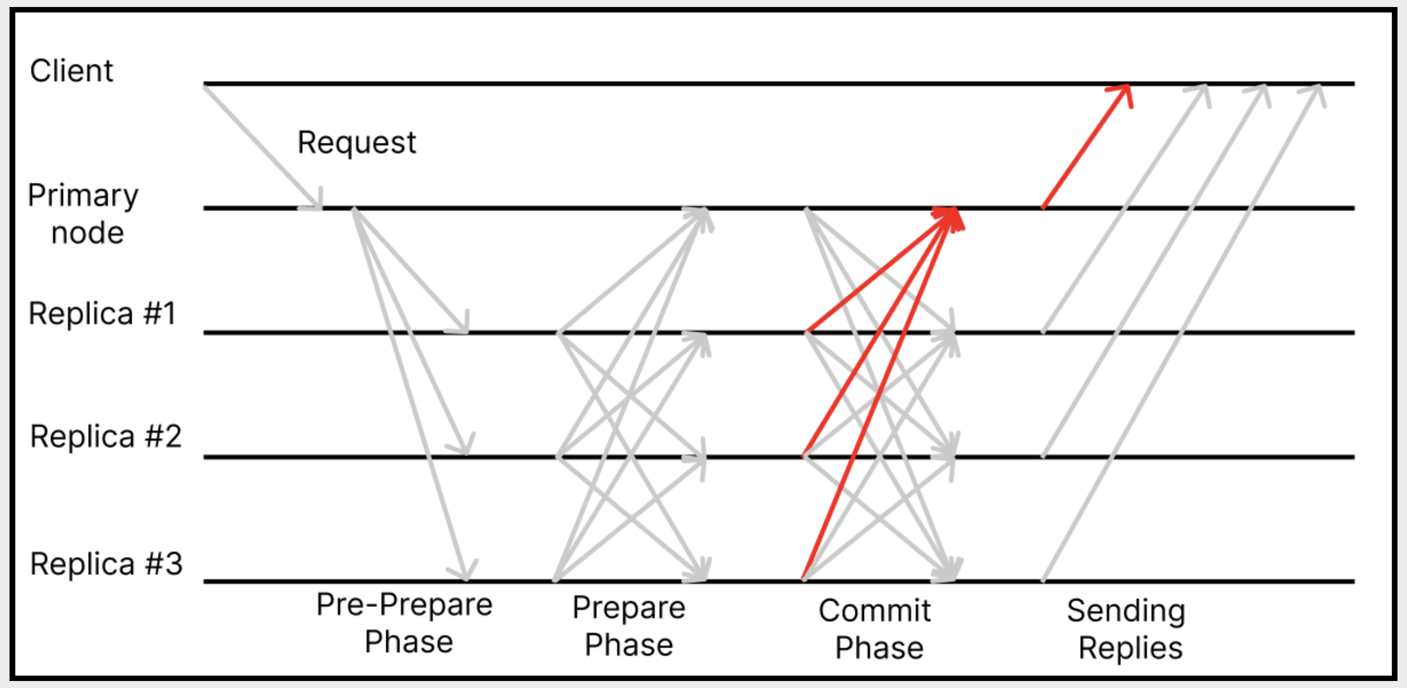
3. Methodology

1. Framework selection
- Coyote - allows for reproducing test executions and provides the exploration strategies

2. PBFT Implementation
- Implemented via Coyote's actor model

3. Seeding of bugs
- First benchmark: the bug is in the prepare phase of the algorithm. Our implementation keeps track of the *commit-local* predicate for each request via a dictionary. 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.

- Second benchmark: The second bug is related to a change in the number of nodes needed for reaching a majority for consensus. It was arbitrarily decided to seed the bug in the reply phase. 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.



4. Results and 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.

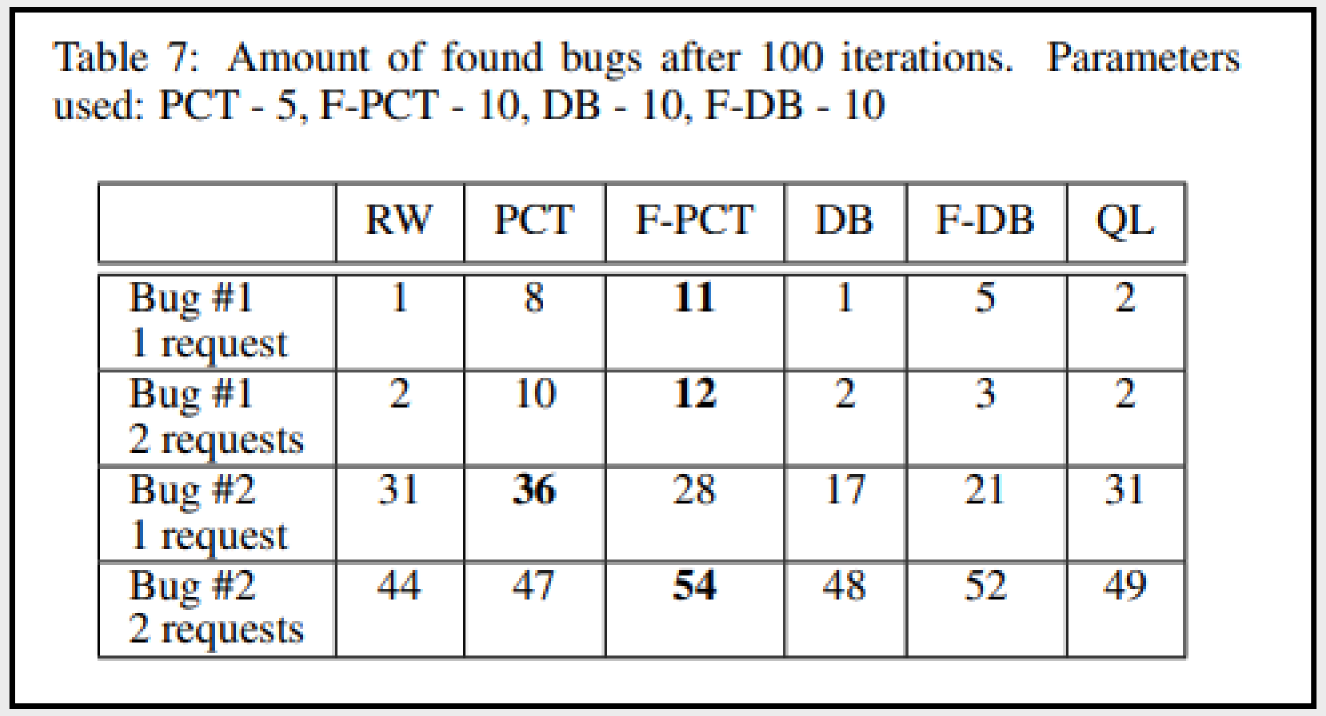


Table 7: Amount of found bugs after 100 iterations. Parameters used: PCT - 5, F-PCT - 10, DB - 10, F-DB - 10

	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

The results indicate that the QL exploration algorithm, which aims to maximize unique fingerprints in a program, shows a dependence on the number of iterations. Increasing the coverage in a test run leads to finding more bugs per iteration in some cases, but conflicting results were obtained for different benchmarks.

In conclusion, based on the experiments conducted, it can be recommended to build a portfolio of exploration strategies instead of relying on a single strategy when testing for concurrency bugs in PBFT. PCT and F-PCT strategies showed promising performance, but further research and experimentation are needed to validate their effectiveness and explore other potential strategies.

5. Conclusion and future improvements

We have compared 4 different exploration strategies. F-PCT is the best performing for our bugs. Future improvements:
- Optimizing parameters for bounded strategies
- Seeding bugs involving a view change
- Further research on fair and unfair strategies for their performance and possible use cases