QUANTAS-WACOFA

QUANTITATIVE **U**SER-FRIENDLY **A**DAPTABLE
NETWORKED **T**HINGS **A**BSTRACT **S**IMULATOR

WITH **A** **C**OUPLE **O**F **F**EATURES **A**DDED

A thesis submitted to
Kent State University in partial
fulfillment of the requirements for the
Nesterenko-Certified CS Researcher
Qualification

by
Zaz Brown
July 2023

1 Dedication

To Prof. Nesterenko, who runs a research boot camp more challenging than military boot camp. Who somehow managed to extract 2 years of my life in a mere 3 weeks.

Most importantly, I dedicate this work to my cat, Tailwind, who has stood by me through these hard weeks and provided me with love and support. And also to my wife, who fed the cat.

Contents

2 Abstract

This paper documents some additions to Quantas[3]. Primarily, the ability to introduce Byzantine behavior via "infections". An infection is simply a function that modifies a peer, and can be specified in the config file. An "infected" node is simply a node with modified behavior. It is no longer *guaranteed* to be correct, but it *may* be correct.

3 Implementation

3.1 Add infrastructure to "infect" nodes

Infections were originally implemented as higher-order functors (see 4.1), but now they are simply functions that accept a Peer as argument and modify the methods of that peer. To achieve this, new function pointers were added to `Peer` that wrap existing functionality.

3.2 Add visualization infrastructure

I created a Python script that will automatically generate plots based on Quantas' JSON output. In the future, it would make sense to have Python generate the JSON input also, as that is cumbersome (see 7.2).

3.2.1 Git supermodule for analyzing results

I factored out my new code for visualizing results into a git supermodule, meaning that it is a git repository that will contain Quantas as a submodule. This layer of abstraction allows both Quantas developers and end-users to fork the visualization repository and make changes to that without those changes being intertwined with the changes to Quantas itself.

Once both Quantas and the visualization repository are both stable after this flurry of activity, I plan to add Quantas as a submodule to the visualization repository.

3.3 Change benchmarking measure

`latency` is not updated if commits do not occur. This caused some problems as you can see in Results. Nghia switched to `throughput`-based benchmarking, which seems to work more smoothly. See 7.8 for ideas for further improvements.

3.4 Minor implementation details

3.4.1 Add `multicast` for specified proportion of peers

I improved upon Rachel Bricker's `randomMulticast` that sends messages to a random number of peers. There is now an optional parameter to specify the proportion of peers you would like to send a message to.

```
// Multicasts to a random sample of neighbors without repetition.

// Sample size is a random uniform distribution between 0 and neighbours
template <class message>
void NetworkInterface<message>::randomMulticast(message msg) {
    randomMulticast(msg, -1);
}

// Same, but sample size is number of neighbours * p, rounded down.
template <class message>
void NetworkInterface<message>::randomMulticast(message msg, float p) {

    int amountOfNeighbors;
    if (p < 0)
        amountOfNeighbors = uniformInt(0, _neighbors.size());
    else
        amountOfNeighbors = _neighbors.size() * p;

    // previous implementation
}
```

3.4.2 Next: Change output file extension: `.txt` → `.json`

This makes it more obvious what format the output is in. Thanks to Mitch for the suggestion.

3.4.3 Next: Automatic linting

Fix all whitespace errors and set up automatic linting using a git pre-commit hook.

4 Abandoned approaches

4.1 Infections as higher-order functors

In my first implementation, I created an Infection class which created higher-order functors, but this seemed to introduce complexity for no benefit. I switched Infections to simply being functions that accept callbacks instead.

```
#include <functional>
#include "Peer.hpp"

using std::function;

namespace quantas {

template<class type_msg>
class Infection {
    function<void(Peer<type_msg>*,function<void()>)> _infection;
    // allow infections that don't take performComputation as an argument
    function<void(Peer<type_msg>*)> _fn;
public:
    Infection(function<void(Peer<type_msg>*,function<void()>)> fn) :
        _infection(fn) {}
    Infection(function<void(Peer<type_msg>*)> fn) : _fn(fn) {}

    /**
        * An infection is a higher-order functor:
        *
        * @param the peer that we are performing computation on
        * @param the original performComputation function
        * @return a modified version of performComputation
        */
    void operator()(Peer<type_msg>* peer, function<void()> performComputation) {
        if (_infection != nullptr)
            _infection(peer, performComputation);
        else
            _fn(peer, performComputation);
    }
};

}
```

4.2 Infections accepting callbacks

My next approach was to code an infection as replacement methods that would accept, as a callback, the original method that it would replace. I abandoned this approach in favor of infections as functions that modify a peer. This seems to be the most natural approach, and one that is intuitively easy to explain.

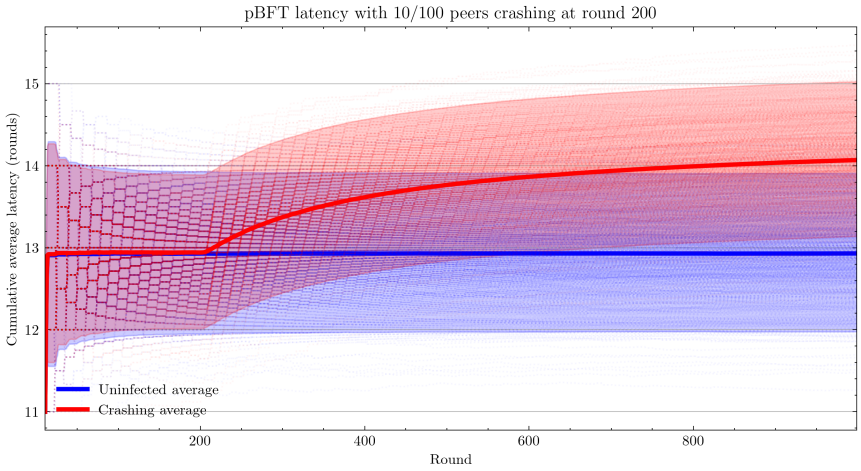
5 Results

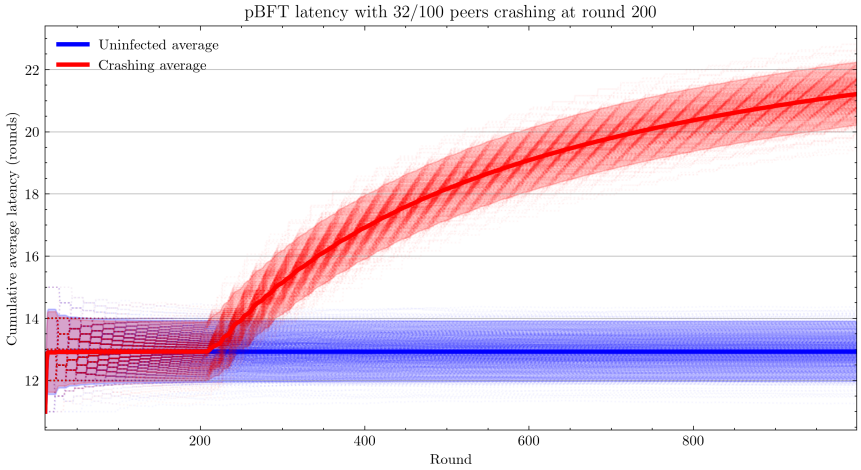
Unless stated otherwise, 1,000 simulations were run using PBFTPeer for both the control case (where all peers are correct indefinitely) and the Byzantine case (where peers are infected at time 200). All simulations were run with a delay of 10.

Plot symbol	Meaning
Semi-transparent blue dot	Results of a simulation of all-correct peers
Semi-transparent red dot	Results of a simulation including "infected" peers
Blue shaded region	95% confidence interval
Red shaded region	95% confidence interval

5.1 Latency

I began by using a `crash` infection that simply halts a peer and the pre-programmed `latency` log of the PBFTPeer. Latency is a self-report from peer 0 of the time since the last commit. This has the drawback that it logs average cumulative latency. Hence the output plot is not trivial to interpret: sudden jumps in latency appear as a transition from a constant function to a hyperbolic one, which is exactly what we see below.

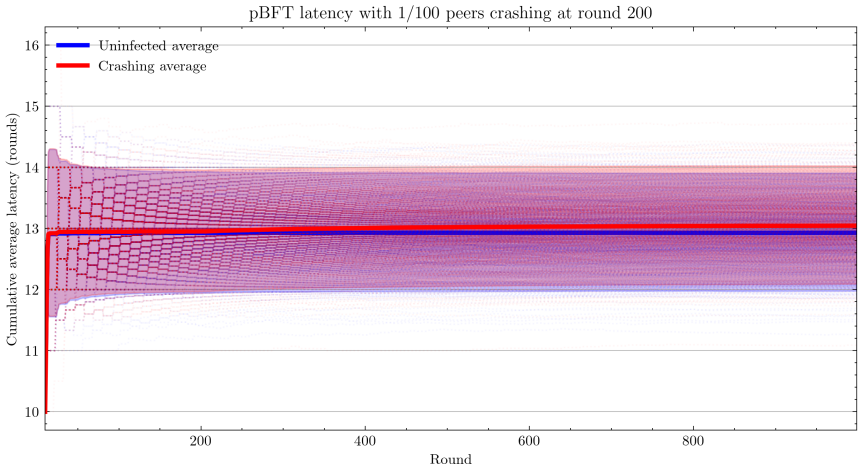


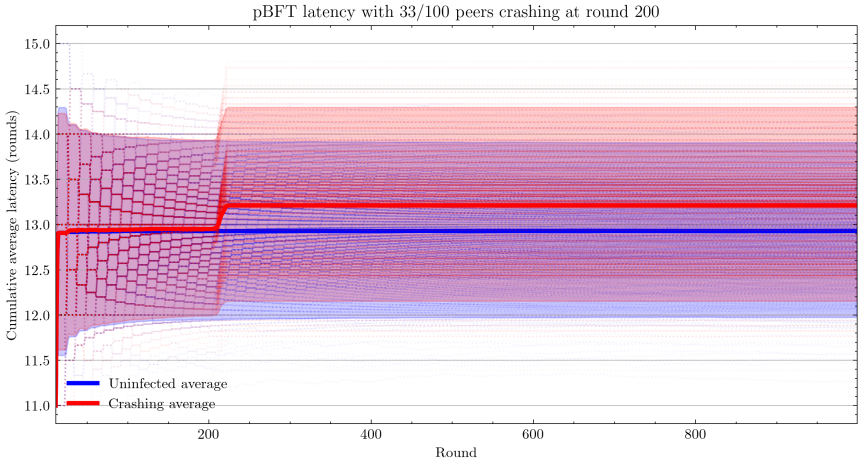


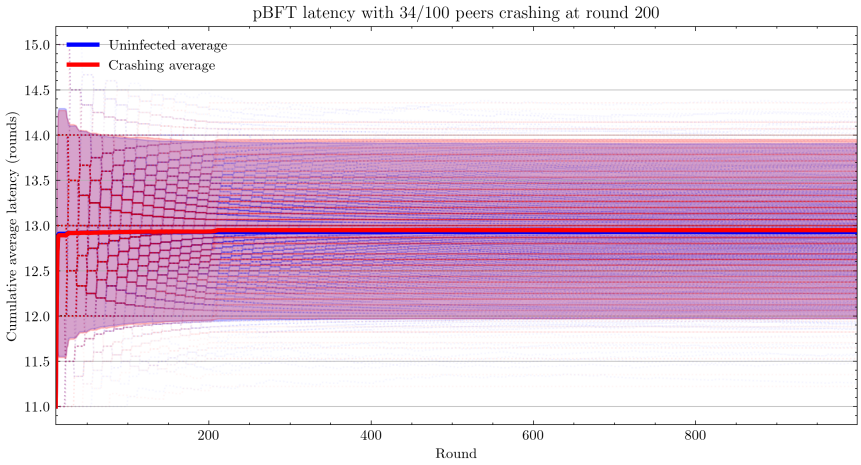
Note that the patterns you see are not plotting artifacts, they are the actual cumulative averages of each simulation. Cumulative averaging of integral changes results in the hyperbolic geometry that you see.

5.1.1 Edge cases

Here, infected nodes flatline because crashed nodes stop reporting the latency.

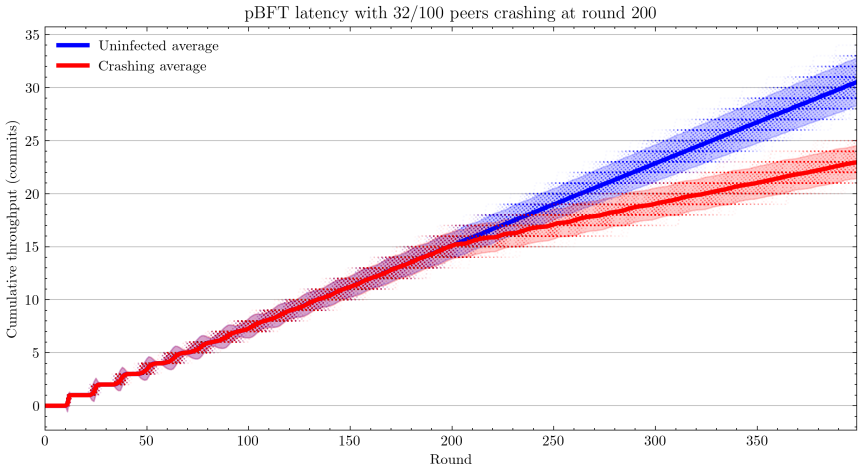




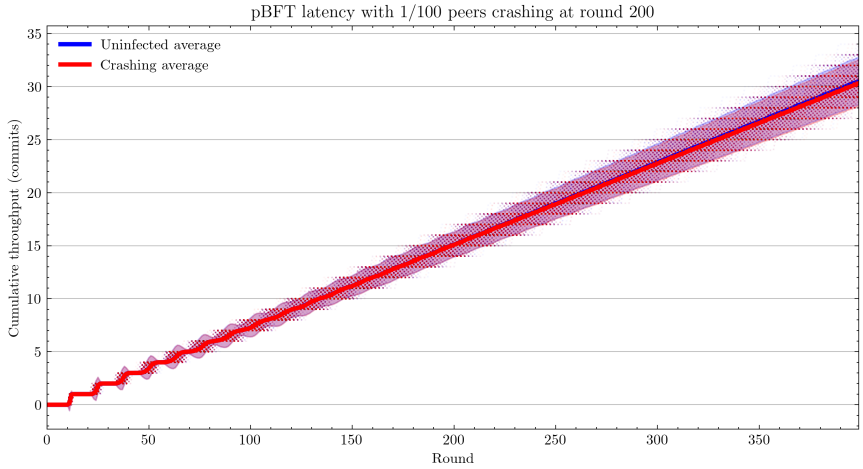


5.2 Throughput

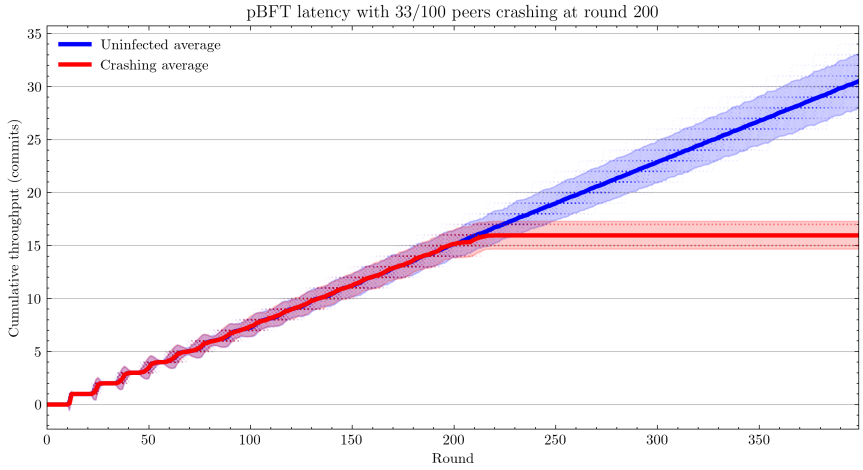
Next, I switched to measuring throughput, which is the total number of commits that have occurred (according to peer 0).



5.2.1 Edge cases



Using 100 simulations instead of 1,000:

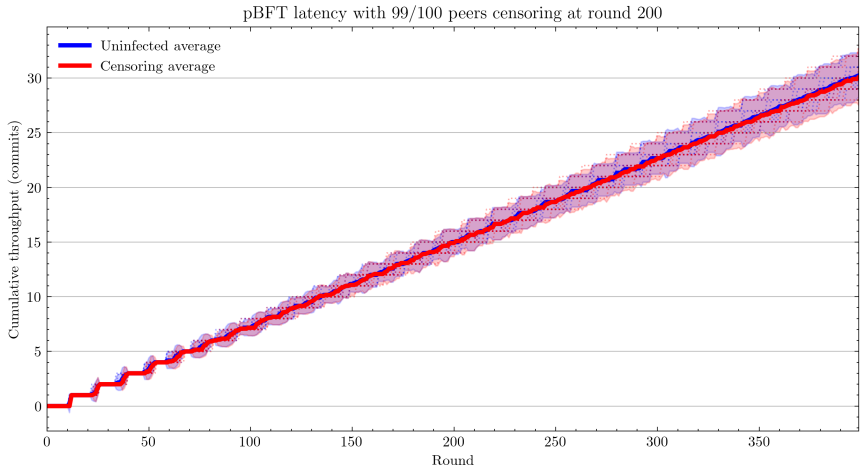


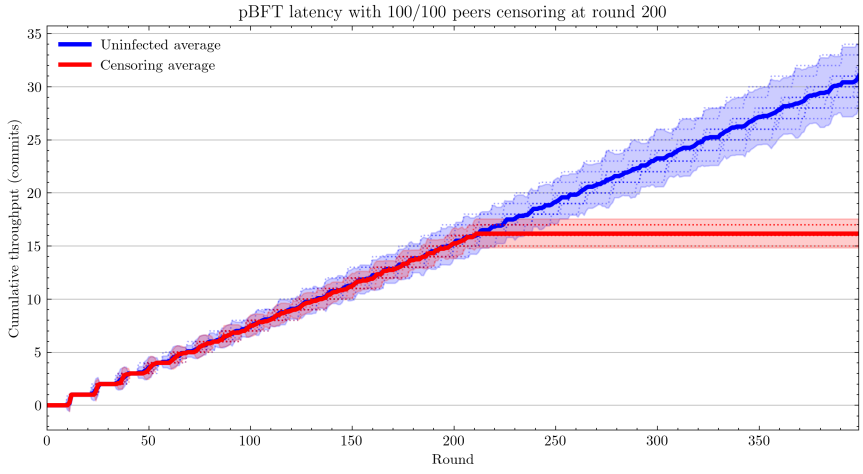
Interestingly, we see that pBFT is failing for 33/100 nodes crashing. This is surprising because this is the most benign kind of fault and pBFT is designed to handle $\frac{n-1}{3}$ faults, where n is the number of peers[1].

5.3 With censorship

Note: From this point forward, all plots are with 12 simulations for each of control and infected cases.

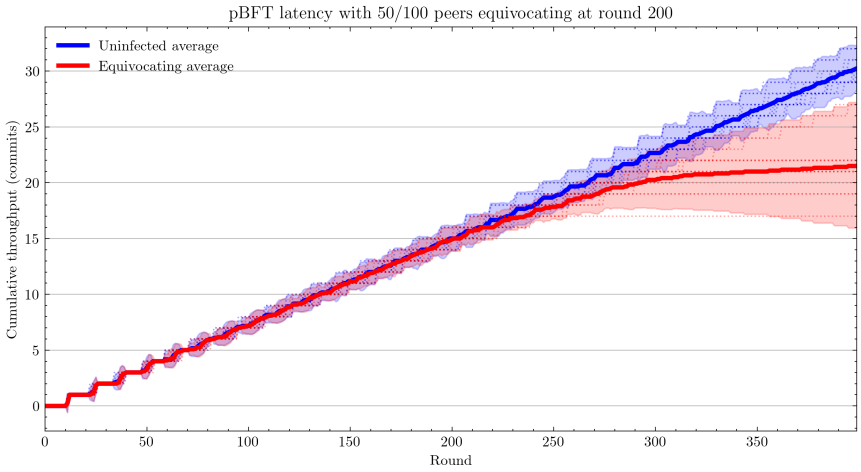
Applying censorship resulted in no statistically significant change in throughput up to 99/100 nodes. Censoring the final node caused the system to fail. This could be related to view changes not being implemented. As the code is written now, peer 0 is hard-coded to be the view leader, and my code infects nodes in reverse order, leaving peer 0 until last (the reason for this is peer 0 is also conventionally the node that handles logging).

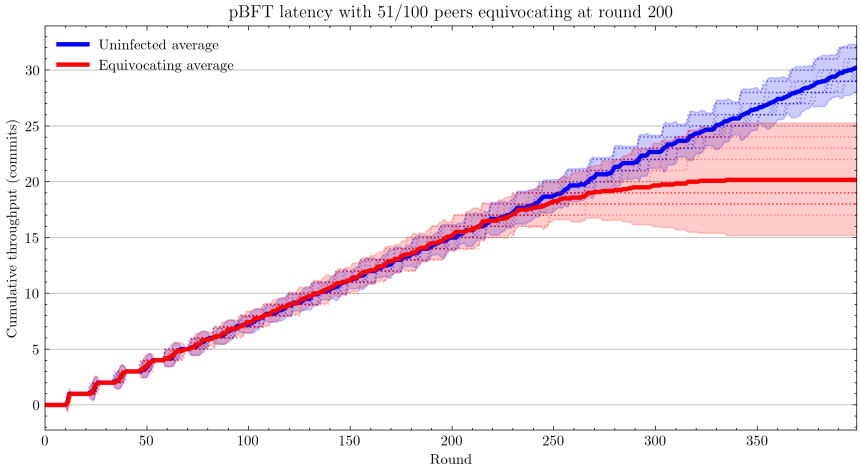




5.4 With equivocation

Next I tested equivocating on the commit message by sending the commit to a random selection of peers (see 3.4.1) and not sending it to the rest. With more than 45 equivocating peers, this began to slow down the network drastically.





6 Unresolved questions

6.1 Is [&] an appropriate way to capture this?

```
std::function<void(message)> send = [&] (message msg) {
    NetworkInterface<message>::broadcast(msg);
};
```

6.2 Is passing this best practice?

```
virtual void defaultComputation () = 0;
std::function<void(Peer*)> computationPerformer = [] (Peer* peer) {
    peer->defaultComputation();
};
void performComputation () { computationPerformer(this); };
```

6.3 Unnecessary looping?

Is looping over _neighbours in the below code necessary, or can it be removed? Is there any reason sending a message with a target that doesn't exist would cause issues?

```
// Send to a single designated neighbor
template <class message>
void NetworkInterface<message>::unicastTo(message msg, long dest){
    for(auto it = _neighbors.begin(); it != _neighbors.end(); it++){
        if(*it == dest) {
            Packet<message> outPacket = Packet<message>(-1);
            outPacket.setSource(id());
            outPacket.setTarget(*it);
            outPacket.setMessage(msg);
            _outStream.push_back(outPacket);
        }
    }
}
```

7 Suggestions for Future Research

7.1 Allow taking JSON on `stdin`

The convention with Unix-style programs is to use `program -` to mean: Instead of reading from the file represented by the first argument, read from `stdin` instead. This will make it more convenient for programs that use Quantas to feed it JSON they generated. Instead of writing a file and giving that to Quantas, or passing a file by file descriptor, which is not portable, programs will have the third option of passing data via `stdin`.

7.2 Move JSON input out of Quantas repository

It is not conventional to store the input to a program in the same repository as the code for that program. I suggest removing the Makefile entries for different program inputs and moving the JSON input files into the supermodule, Quantas-analysis. This allows end-users to develop version-control their own analysis independently of Quantas core development, allowing seamless updates of Quantas.

7.3 Deal with peers symmetrically

If we are able to refactor Quantas in a way that we never subscript peers before they are randomly shuffled, it could provide benefits to user by allowing people to hack on Quantas more easily. For example, when I first started building Byzantine peers, I found it more convenient to make the first X peers Byzantine. This is only a safe approach if nothing else in the codebase picks a range of peers deterministically. Unfortunately for me, `_peers[0]` is hard-coded as indispensable.

```
_peers[0]->endOfRound(_peers);
```

7.4 Output file management

First, I created `results` to keep all of my results in one place. Then I created various subdirectories to manage results for different parameters. Then I modified Quantas and moved `results` to `old/results0` because I wanted to keep a copy of the results from before I modified Quantas. It would be better if Quantas handles all of this administrative work.

We could do this simply by storing results in e.g. `results/v1.0/pBFT/delay10/rounds400/peers100/infect10`. This seems

like a convoluted way of doing things, but it actually reduced duplication of computation while experimenting with Quantas. A user may want to try many different approaches before deciding on the one they want to present, by which point their output files have become an organizational nightmare. This would also eliminate the chance of user error in naming the output files; currently it is easy to set `rounds: 400` in the config file, but forget to change the output filename to reflect that parameter.

This approach could be taken even further to eliminate JSON input files altogether and have the entire input state represented in the filepath. This would allow us to avoid accidental duplication of work by checking the `mtime` on the files, like `make` does. A hybrid option would also be possible, where a JSON file exists in `pBFT/` and those options are added to all simulations.

7.4.1 Automatic output file generation

We could even take this one step further and add a filesystem watcher so that, when an empty `.json` file is created in a subdirectory, Quantas adds it to a queue to generate output for it.

7.5 Build a Domain-Specific Language

When I first created a way to infect nodes, I hard-coded into `Simulation.hpp` I then moved this functionality into the config file, but if we keep doing this for every feature, the complexity of the config files will balloon. It might be better to make the config files Turing-complete, i.e. make a DSL.

I expect that making a DSL will lead to simplifications of Quantas and could even allow techniques from compiler optimization to be used to speed up Quantas.

Note that a DSL is not compatible with the previous suggestion to move all configuration to the file path, but it is compatible with the hybrid approach of storing some configuration in the file path.

7.6 Make every peer log

At the moment, logging is done from the perspective of one peer only.

7.7 Community engagement

7.7.1 Don't display the main repository as a fork

GitHub displays notices at the top of the Quantas repository:

forked from khood5/distributed-consensus-abstract-simulator

This branch is 124 commits ahead of khood5:master.

This gives the impression that QuantasSupport/Quantas is not the canonical source for Quantas. This can be easily fixed by contacting GitHub customer support.

7.7.2 Git authors should be individuals

Having QuantasSupport registered as a user rather than an organization and then using that anonymous account to commit to the repository is highly irregular. This can cause community trust issues due to not knowing which individuals authored each part of the project. Quantas contributors should be informed that using a full name for FOSS contributions is the norm.

The Quantas repository can be moved under an organization, such as QuntasSim in a way that automatically redirects QuantasSupport/Quantas → QuantasSim/Quantas.

7.7.3 Project discoverability

Use tags. Create a banner for the repository. Create a logo.

7.8 Simplify logging: log everything

The way logging is currently done could be simplified: We could simply log every event and the time at which it occurred. Currently, pBFT outputs a time-series of throughputs, which is very easy to work with, but it is not the natural format for this data. We have a number of different events occurring on different rounds. If we were to simply record pairs (time, event) that would allow more flexibility when it comes to analyzing the data. The parsing is more complex than a time series, but libraries exist that will handle that for you. This avoids having to modify the code for the computation and re-run the simulation, which is expensive.

7.9 Unit testing

Once I had a working visualization system, I would use that after each change I made. I would visually check that the plot generated looked the same as the previous one to ensure I hadn't made any coding errors. It would be better to automate this. You could have both a light test suite, to briefly

check correctness before each commit, and a more extensive test suite that runs automatically on GitHub when a pull request is submitted.

7.10 Latin hypercube sampling

We could use Latin hypercube sampling to reduce the number of tests needed for a desired precision. With LHS, Precision is proportional to $1/n$ instead of $1/\sqrt{n}$. LHS is an extension of stratification that works by replacing each type of random decision with a dimension in a Latin Hypercube. This may be problematic in our case, though, as generation of high-dimensional Latin hypercubes is hard. One solution is dimensionality reduction: Pick only your most correlated categories of decisions and use LHS for those categories, while using random number generation for the rest. [2]

7.10.1 Alternatively, replace PRNG with QRNG

A simpler solution may be to replace most instances of pseudorandom number generators with quasirandom number generators. QRNGs produce low-discrepancy sequences that increase uniformity of the resultant sequence.

References

[1] Miguel Castro and Barbara Liskov. "Practical Byzantine Fault Tolerance". In: *Proceedings of the Third Symposium on Operating Systems Design and Implementation*. OSDI '99. New Orleans, Louisiana, USA: USENIX Association, 1999, pp. 173–186. ISBN: 1880446391.

[2] G Damblin, M Couplet, and B Iooss. "Numerical studies of space-filling designs: optimization of Latin Hypercube Samples and subprojection properties". In: *Journal of Simulation* 7.4 (Nov. 2013), pp. 276–289. ISSN: 1747-7786. DOI: 10.1057/jos.2013.16. URL: http://dx.doi.org/10.1057/jos.2013.16.

[3] Joseph Oglio et al. "QUANTAS: Quantitative User-friendly Adaptable Networked Things Abstract Simulator". In: *Proceedings of the 2022 Workshop on Advanced tools, programming languages, and PLatforms for Implementing and Evaluating algorithms for Distributed systems* (July 2022). DOI: 10.1145/3524053.3542744. URL: http://dx.doi.org/10.1145/3524053.3542744.