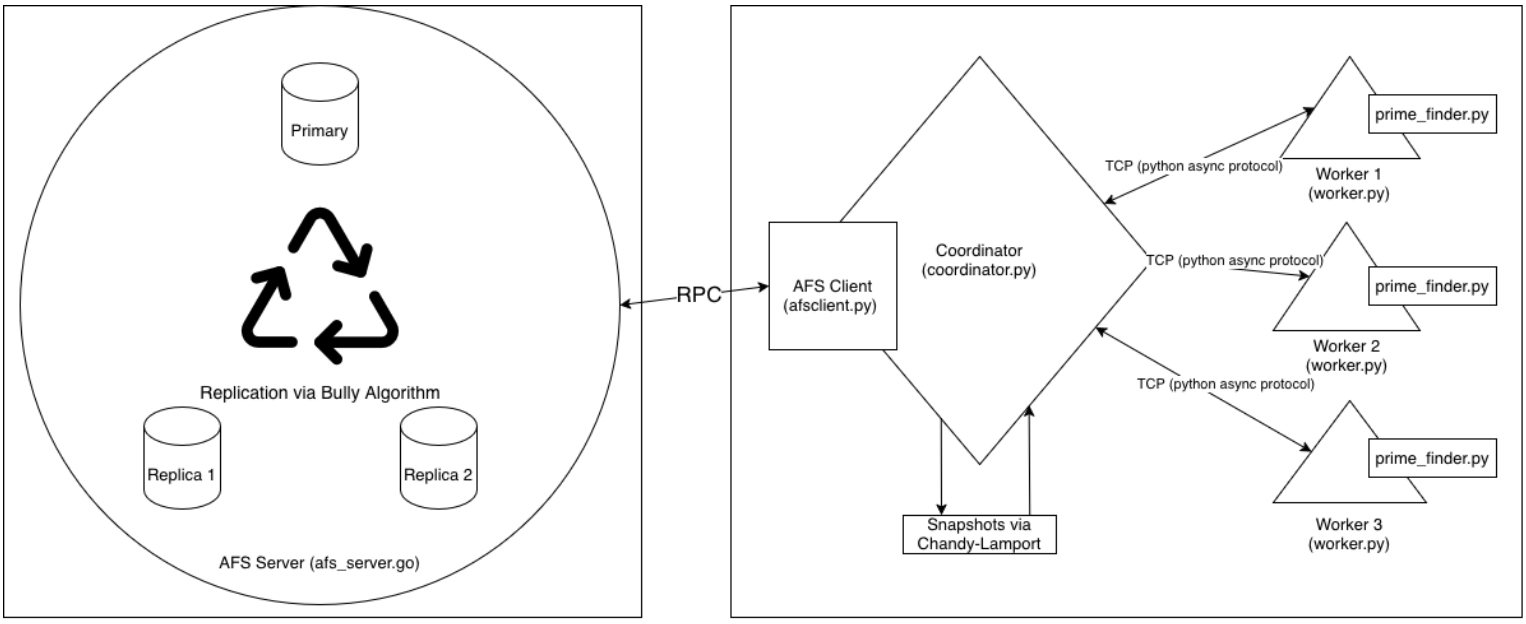
# DISTRIBUTED PRIME NUMBER TESTER

## Introduction

Our team was tasked with designing a prime number tester and a file system which runs in a distributed manner and is easily scalable. Our team decided to go with an AFS-like user space file system written in Go and Python. We also elected to use a Coordinator-Worker model for our prime finder and used python to write it. This document will provide an overall high-level overview our project, will justify the design decisions, discuss our testing, and identify areas for future improvement. The philosophy we had going into this project was that: “Keeping our system simple is the best way to have a good system”.

## Design

This section discusses the designs of our file system and distributed prime number testing.




AFS-Like File Server

Prime Finder

### Instructions for Running the System

- Run system through test functions – detailed in the Testing section
- Run and stop system through the run_system and quit_system functions in tests.py

## AFS-Like File System

- The aim was for simplicity with the file system design; therefore, the file system was modelled after the Andrew File System. This architecture was chosen for its ease of implementation, and the fact that the file system reads and writes whole files instead of reading or writing blocks of a file. It simplifies concurrent read and write edge cases.
- The file system acts as our database.
- We have two types of files: input files, and output file(s).
- The file system compromises of an AFS server written in Go and an AFS client written python.
- The AFS server and client uses Remote Procedure Calls (RPC) to communicate with each other.
- The server implementation uses the bully algorithm to maintain three AFS servers, one primary and two replicas. The number of replicas can be easily scaled up, if needed, to have higher availability and increasing the time between full cluster reboot. The whole cluster needs to be rebooted after a certain number of replicas go down, hence increasing time between reboots is highly positive trait. AFS' simple writes help by guaranteeing that the replicas are getting full updated files written to them instead of the updated files being written to them in a piecemeal fashion hence eliminating the worry of the replicas having a partly updated file when the primary server goes down.
    - The AFS server is made fault tolerant by the implementation of replicas and using the bully algorithm to elect a primary server when one of the servers goes down.
- Since prime numbers might only be found very sporadically, the prime number testing operation is assumed to be a read heavy operation. The overall system will not be able to be productive at all if the file system is not available. Hence, the system design focuses on high availability instead of high reliability.

## Fault Tolerant Distributed Prime Testing

- The prime finder is based on a coordinator-worker architecture for it's ease and simplicity of implementation and scalability.
- The coordinator assigns tasks, tells the workers which files to work on, handles all the connections to the workers, handles writing primes to an output file, via the AFS client to the server, when a prime is found. This allows the whole system to simplify communications, explained in the worker section below. Only the coordinator needs to be a top-of-the-line machine allowing our system to scale easily and rapidly.
- The workers are treated as cheap expendable labour. The workers only calculate primes and report the results back to the coordinator. This makes spinning up new workers very trivial making our system highly scalable. The network overhead is also reduced with this

system because the workers do not need to coordinate between each other constantly, reducing network traffic and reducing network calls massively. With the reduced traffic our workers also could be potentially geographically disparate from each other.
    - The Fermat primality test is being used to find prime numbers. This method was a deliberate choice because probabilistic tests are fast and accurate making it a perfect choice for in this application, adding to the ease of scalability of our system.
- The downside of this approach is that our system has a highly critical single point of failure i.e. the coordinator. This is mitigated by having a snapshot taken of the state of the coordinator, allowing the coordinator to recover rapidly from a failure.

Snapshots

- Chandy-Lamport algorithm is used for snapshotting.
- Snapshots are implemented on the coordinator in the prime finder. This allows the coordinator to recover from failures in a rapid manner in case of failures.
- Snapshots are taken every 30 seconds.
- The snapshots record files completed, files pending, output buffer, and file count.

## Testing Design

- Snapshot Creation Test (test.py > def worker_snapshot_test) - Snapshot creation test runs AFS, coordinator, and the workers for 45 seconds, which is our snapshot timer + buffer time, and tests if a snapshot has been created in that time.
- Worker Failure Test #1 (test.py > def worker_failure_test) - If a worker goes down and comes back up, a check is done to see if the file that worker was working on gets eventually completed or not.
- Worker Failure Test #2 (test.py > def worker_failure_test) - If a worker goes down, test to see if the worker can join again either with a new id or it's old unique id. Tests to also see if the worker fails, the process associated with the worker is tore down.
- AFS Primary Server Failure Test (test.py > afs_primary_failure_test)- Tests to see if replicas take over after primary FS server failure.
- Server Replication Test (test.py > def afs_replication_test) – Data is sent to the primary AFS server, test checks if the replicas have been updated with the data or not.
- Coordinator Failure Test (test.py > def coordinator_failure) – Test checks if coordinator recovers from a snapshot, if it does down. Liveness guarantees that all the tasks are completed or will eventually get completed.

## Future improvements

- The AFS client can be implemented fully in Go. Go having better concurrency than Python makes it a better choice. Python was used in this instance for ease of implementation given the time constraints.
- Use Paxos implementation for electing primary AFS server after failure instead of the bully algorithm.
- Make caching more robust and using mutex properly.
- Fully distributed snapshot system – The workers need to recover from snapshots.