Coursework

Introduction

You recently graduated from the University of Edinburgh with flying colors and have started working at PrimeScience, a cutting-edge startup in the cryptography and blockchain space. The company has been focusing on developing mathematical libraries involving prime numbers, which are crucial for RSA encryption and cryptocurrency mining algorithms. Your company's current single-server solution can handle small datasets. Your company has just received a massive contract with a major tech company that requires processing petabytes of numerical data to discover large prime numbers for next-generation encryption systems.

Your CEO, who notoriously avoided the Distributed Systems course during University, suddenly realizes the company needs to scale beyond a single machine. The CEO recalls that you were bragging about the Distributed Systems course during the interview. Your CEO pulls you into a conference room and says, "We have 4 weeks to deliver a distributed platform that can provide large prime numbers by processing data 1000x larger than what we currently handle. The client is paying us millions, and our competitors are breathing down our necks. Can you make this happen?"

You take a deep breath, sit down at your desk, and start thinking about the project. You want to make the CEO regret not taking a Distributed System class in the past. **Of course, you may understand the regret of taking the Distributed System class.**

After analyzing the requirements, you recall how your course instructor taught you to start visualizing the system abstractly initially. You realize the project perfectly maps to classic distributed systems problems. The solution needs two crucial components:

1. A distributed filesystem, where the dataset is stored and accessed in a file server.
2. A distributed prime number finding application that will access the distributed filesystem to access the dataset.

You sit down and start capturing your thoughts around these two components.

Part 1: A user-space AFS-Like Distributed File System

The massive input datasets (multiple files, containing billions of numbers, in random order, and may have duplicates) need to be stored and accessed reliably across machines. You decide to implement an Andrew File System (AFS)-inspired user-space distributed file system.

Why AFS-like? Since prime number identification is read-heavy (workers only read input files and write final results on finding a prime number(s) in those datasets), you can simplify the design:

- No concurrent writes: Input files are static; output files are write-once
- Whole-file caching: Clients can cache entire files locally
- Simple consistency: No need for complex cache invalidation

Why user-space? Your file system workload is mostly read-only with occasional writes. You want to avoid mounting the file system, writing complex code in the Linux kernel, etc.

The filesystem will store:

- Large input files: input_dataset_001.txt, input_dataset_002.txt, etc., each containing millions of numbers
- Result files: primes.txt, one single output file containing discovered prime numbers from input_dataset_*.txt file.

You recall that you need to worry about fault-tolerance and availability in a distributed system.

Part 2: Fault-tolerant distributed prime number finding application

The prime number finding application will analyze the numbers present in the massive input files stored in the distributed file system and identify all unique prime numbers within those input files. The final output across should never have duplicates. Duplicate prime numbers will break the protocol, leading to incorrect results. Please note that the duplicates may exist across different files; each file may be processed separately.

The beauty of primality testing is that it is embarrassingly parallel - each number can be independently tested, making it perfect for distributed processing across multiple machines. You are free to choose any primality algorithm you want.

The application takes periodic snapshots to recover from faults and avoid redoing the work post-failure recovery. The application recovers from failures by restoring to the latest snapshot. (Think about how the application, snapshots, and the distributed filesystem will work in tandem to restore from a particular snapshot.)

Administrivia

- **Start Date**: 13/10/2025
- **Due Date**: 12/11/2025 (noon)
- **Questions**: We will be using Piazza for all questions. We encourage you to answer the questions. In doing so, you will learn and help each other to succeed.
- **Collaboration**: This is a group assignment. Given the amount of work to be done, we recommend you work as a group (5 members). Working individually on the project will be a daunting task.
- **Programming Language**: Any language you prefer.
- **Tests**: Some test case descriptions will be provided.
- **Presentation**: We will ask you to present a demo that effectively showcases your system's functionality and correctness. You will be graded based on your presentation. Please note that the presentation is necessary for the grading. If you do not present, we will not grade your project.
- **Design Document**: You will also submit a 2-page document that discusses the design details and the system model, assumptions, etc. We do not want a sophisticated document. Please do not spend a lot of time on it. Rather, focus on the design, implementation, and testing of the entire framework.
- **Office Hours**:
    - Office hours will take place in Appleton 4.07 from 1 to 5 p.m. on every weekday, starting October 13, 2025. When there is a lecture on that day, office hours will begin at 3:00 p.m. and will still last for 2 hours.
    - During office hours, we will prioritize design-related questions and are happy to help you understand the broader picture. Programming-related questions will be given lower priority. We can surely help with coding issues if time permits, but design questions can take a VIP pass and cut to the front of the line.
    - Please email Yuvraj to schedule office hours with him.

Background

To complete this coursework, you will need a good understanding of file system, distributed snapshots, and primality testing.

- **File System**:

  https://pages.cs.wisc.edu/~remzi/OSTEP/file-intro.pdf, https://pages.cs.wisc.edu/~remzi/OSTEP/file-implementation.pdf, https://pages.cs.wisc.edu/~remzi/OSTEP/dist-afs.pdf

- **Primality Testing**:

  https://cp-algorithms.com/algebra/primality_tests.html

- **RPC**:

  https://dl.acm.org/doi/pdf/10.1145/2080.357392

- **Distributed Snapshot**:

  https://opencourse.inf.ed.ac.uk/sites/default/files/https/opencourse.inf.ed.ac.uk/ds/2024/chandy.pdf, https://decomposition.al/blog/2019/04/26/an-example-run-of-the-chandy-lamport-snapshot-algorithm/

- **Replication**:

  https://www.cs.utexas.edu/~lorenzo/corsi/cs380d/papers/statemachines.pdf

Objectives

As mentioned on the course website, on completion of this course, the student will be able to:

- Outcome 1: Develop an understanding of the principles of distributed systems and be able to demonstrate this by explaining them.
- Outcome 2: Being able to give an account of the trade-offs which must be made when designing a distributed system and make such trade-offs in their own designs.
- Outcome 3: Develop practical skills in the implementation of distributed algorithms in software so that they will be able to take an algorithm description and realize it in software.

- Outcome 4: Being able to give an account of the models used to design distributed systems and to manipulate those models to reason about such systems.
- Outcome 5: Being able to design efficient algorithms for distributed computing tasks.

Environment

The coursework should be done on a UNIX-based platform (Linux preferred). You are welcome to use any of the DICE machines or Virtual DICE. For Apple users, you can also refer to this guide and set up a Linux VM within your local machine. For Windows users, WSL is an excellent choice. Your clients and servers can be represented as processes, containers, or VMs.

Specification

Part 1: AFS-Like User-space Distributed File System (45%)

You must implement a highly available user-space distributed file system inspired by the Andrew File System (AFS). The clients will run the distributed application and execute the primality testing algorithm.

The clients will open the remote file. Upon file open, the client will request the entire file from the server and store it in the local drive. Subsequent read and write requests will be redirected to said local copy. When the client issues a close, the file contents are flushed to the server if the file is modified. We will be using a protocol similar to AFSv1 (to make life a little simpler).

You will create RPCs to interact with the distributed file system. For the local read and writes, you will use the standard POSIX APIs. You need to support basic file system functionality - open, create, read, write, close files, and all the other relevant functionality that you may need.

The client side will initialize with a local directory pathname where the remote files are cached. For example, the client might store such files in /tmp/afs. The local directory pathname should be specified while starting the client.

The server initializes with two directory pathnames; this path is where the files in this file system reside (i.e., the persistent state of the file system is stored). The clients will ask the server for the directory path upon initialization. You should assume all the input files reside in the server directory beforehand. You do not have to create or delete them. The output file is stored in another directory. Only the output file will be created and deleted (in case of a recovery). However, please follow the rules to name the input and output files as mentioned above.

Your file system server should be highly available. You should rely on replication to ensure high availability. If one server is down, clients can access files from other replicas. You can either use simple state machine replication or implement consensus algorithms such as Paxos or Raft. Once the faulty server recovers, you will need to sync the server to ensure all the replicas have a consistent view. **Please think carefully. As we have emphasized in the lecture, do not overcomplicate things — stick to a simple approach.**

Your file system servers need to handle faults gracefully and recover properly without creating correctness issues. (Check above to identify what it means to design a correct application.)

Task Breakdown

- Task 1A: Basic Client-Server with RPC (15%)

Get a single-file server working with remote clients using RPC (Remote Procedure Call). Implement basic RPC versions of open(), create(), read(), write(), and close().

- Task 1B: Client-side File Caching (5%)

Add whole-file caching to clients. For clients, they will request the entire file from the server and store it in the local drive upon the first file open. Subsequent read/write operations should happen locally, and writes will be flushed to the server when the file is closed. For the next time, the client will send a TestAuth message to the file server to determine whether the file has changed. If not, the client would use the existing locally cached copy.

- Task 2: Support fault tolerance. (10%)

Depending on your system model, you will need to identify all the faults that may occur and handle them gracefully. (Hint: Try to take the approach we discussed in class for RPC + Fault Tolerance.)

- Task 3: Replicate the File Server (15%)

Add multiple (at least three) replica servers to support server failures. There are several design approaches to choose from. One approach is to rely on simple primary-backup replication, where one primary server handles all writes, and multiple backup servers replicate the data. If the primary server fails, the replicas will elect a new primary. An alternate approach is to rely on consensus algorithms, such as Raft and Paxos, for replication.

Please note that you are not allowed to use any existing libraries to implement tasks 1, 2, and 3.

Part 2: Distributed application for finding prime numbers (35%)

The core functionality of your distributed application comprises reading a large input file from the distributed file system, processing the numbers, reporting the unique prime numbers, and saving them in one output file. For correctness, you need to ensure that the prime numbers are unique and not repeated across the output file. You can assume that all numbers in the input files are 64-bit unsigned integers, and there is one number per line.

As primality testing is embarrassingly parallel, you can distribute the computation across multiple workers (clients). You have complete freedom to choose the application architecture. You may use a Coordinator-worker design, where a centralized coordinator assigns work to multiple workers. An alternate approach is to use a peer-to-peer technique, where workers self-organize, share work, and coordinate without a central coordinator. A third alternative approach is to design a map-reduce-like solution. In any case, you will need to show that your application scales with the number of workers.

Your application needs to handle worker failures. You will do so by periodically taking consistent snapshots using the Lamport-Chandy algorithm and storing all snapshot data in the distributed file system. The snapshot comprises the state of the system (workers), the input and output files they are working on, the number of workers in the system, etc. Depending on the design, you may have to capture more metadata. You will implement a recovery mechanism that can fully restore the application from the most recent snapshot. Without snapshots, any crash would require restarting the entire computation. With such a huge dataset, restarting the computation could waste hours of work. Therefore, snapshots will help you recover quickly from the latest checkpoint. Your recovery mechanism will differ depending on the number of workers failing. For example,

if a single worker fails, all the other workers do not have to recover. Only the failed worker needs to recover from the snapshot. However, if the majority of the workers fail, it would be easy to recover all the workers from the snapshot.

Task Breakdown

- Task 4: Basic distributed application: (20%)

You need to build an application that accesses the distributed file system to find all unique prime numbers from a large input dataset and store the results.

- Task 5: Support fault tolerance using global snapshots (15%)

The application will periodically capture a consistent global snapshot of its state and save it on the distributed file system. On worker(s) failure, the application will recover from the latest snapshot instead of starting over.

Please note that you may use any method for primality testing, including existing libraries. You cannot use existing libraries for any other purposes to implement tasks 4 and 5.

Gentle Reminder

You can work on both components in parallel! Since this is a team project, you can split your team to maximize efficiency. 2-3 people can work on implementing the distributed file system while the rest of the team can focus on implementing the distributed prime finding application.

The application does NOT need to wait for the distributed file system to be complete! You can use the local file system initially. You can integrate the application with the distributed file system later, when both components are ready.

Grading and Submission

The distributed file system component and the distributed application are worth 45% and 35%, respectively. The presentation accounts for 15%, and the design report is worth 5%. If you only complete some parts of the project, you can still earn points by finishing the presentation and design document. You are not required to complete all the tasks. So, avoid stressing yourself to complete all the tasks. Upon completing tasks 1 (20%) and 4(20%), along with the presentation (15%) and design document (5%), you can get up to 60% of the points.

During the presentation, you will explain your system briefly, followed by a demo of your system. We will release more information about the presentation towards the end of the deadline.

Please remember the good scholarly practice requirements of the University regarding work for credit. You can find guidance on the School page. This also has links to the relevant University pages. You should not use any code available on GitHub, Stack Overflow, etc. You cannot use Gen AI tools for the coursework. We will run tools to identify such misconduct.

The deadline and late policy for this assignment are specified on Learn in the "Coursework Planner". Guidance on late submissions is at Late coursework & extension requests. Please note that no extensions are allowed. As the deadline had already been communicated to ITO at the start of the semester, we don't have any control. Late submissions (even 1 minute late) will not be considered, as ITO controls these policies. So please avoid last-minute submissions. You have enough time to work on the project. Start early to avoid any last-minute hassles.

Success Metrics

A successful coursework will exhibit the following:

- Scale: Process datasets much larger than a single machine could handle
- Survive: Continue operating through one file server if it is down
- Recover: Restore from snapshots without losing significant work
- Perform: Show performance improvement with additional workers
- Deliver: Produce correct results consistently and store persistently

Your presentation should focus on the above points, and your demo should convince us that your system works as intended.

The client contract is worth millions and could establish PrimeScience as the leader in distributed mathematical computing. Success means the company goes public next year. Failure means your CEO will definitely not regret not taking that Distributed Systems class. **Time to prove that distributed systems knowledge is worth its weight in prime numbers.**