Distributed Systems S25

I. Instructions and Important Notes

A list of project topics is given below along with a brief description of what is expected. You are free to work with any programming language or framework of your choice, and decide on the exact specifications of your implementation. You are also free to choose a project of your own, for this you would be required to give a formal description of the MVP and Project Outcomes. This will be approved post discussion with the TAs.
There are two components to the project:

- Project proposal: You are required to submit a short description of what you plan to do for the project. This should include details about the problem you are solving, framework/programming language you plan to work with and the deliverables. This will also act as a checkpoint for TAs to give feedback and establish a baseline for the final evaluation. Everything committed in this document would have to be fulfilled in the final submission.
- Project demonstration: This is the project evaluation. You would be required to give a live code demonstration along with a presentation on your project.

Note: In the preference form, you only need to fill the project numbers and not names. The preference form is **not FCFS**. We will try our best to give everyone their preferred topics but we cannot guarantee the final allotment.

II. Project Timeline

- March 3 : Project Allocation
- April 13 : Project Deadline
- April 16-18 : Project Evaluations

III. Project Topics

1. Distributed shared memory — Treadmarks

ABSTRACT: Develop a **distributed shared memory (DSM) system** implementing a **master-slave architecture** where the master node, with the help of slave nodes, **stores and retrieves data**. The system should ensure **consistency and fault tolerance** to handle node failures effectively, maintaining a reliable and scalable distributed environment. Implement key features such as **release consistency, lazy release synchronization, and dynamic process management** to enhance performance and usability in distributed systems.

2. Distributed File System — GFS

   Paper Link: https://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf

   ABSTRACT: In this project you are required to implement a small replica of the Google File System. The paper attached contains the detailed design for implementing a "scalable distributed file system for large distributed data-intensive applications". In your implementation of GFS, we expect you to implement the following features for the MVP (for more details on these, refer to the paper):

   - Write
   - Read
   - Replication
   - Record Append
   - Re-Replication
   - Stale Replica Handling
   - Garbage Collection
   - Operation Log
   - Benchmarking

3. Distributed File System — Haystack

   ABSTRACT : Implement a **distributed storage system** inspired by **Haystack**, optimized for **small file storage and retrieval**. The system should minimize metadata overhead while ensuring **fast lookups** using a **flat namespace** and a **volume-based storage approach**. Students should develop an **API (e.g., gRPC)** to handle **file uploads, retrievals, and deletions**, and simulate a **distributed environment** by incorporating **a caching layer (e.g., Memcached) to reduce disk reads** and deploying **multiple storage nodes** to distribute files efficiently, similar to a **Content Delivery Network (CDN)**. The system should support **replication and fault tolerance** like **NFS or object storage systems**.

4. Distributed Pub-Sub System — Pulsar

   ABSTRACT: Develop a **publish-subscribe (pub-sub) messaging system** similar to **Apache Pulsar**, enabling **asynchronous message streaming** between producers and consumers. The system should support **topic-based messaging**, where producers publish events to a **distributed topic**, and multiple subscribers consume messages in real-time. Implement **message persistence, ordering guarantees, and fault tolerance** by leveraging **Pulsar's multi-tier storage** and **broker architecture**. The system should also handle **multiple consumers (fan-out pattern)** and demonstrate features like **load balancing, durable subscriptions, and message retention policies**.

5. Distributed Pub-Sub System — Kafka

   ABSTRACT: Develop a **distributed publish-subscribe (pub-sub) messaging system** similar to **Apache Kafka**, enabling **asynchronous communication** between producers and consumers. The system should support **topic-based messaging**, where producers publish events to a **distributed topic**, and multiple subscribers consume messages in real-time. Implement additional features such as **load balancing to efficiently distribute workload, scalability to handle increasing message traffic, and failure handling mechanisms for producers and consumers** to ensure system reliability and fault tolerance.

6. Distributed Database — DynamoDB

   ABSTRACT : In this project, you will **design and implement a distributed key-value store** inspired by **Dynamo**, a system originally developed by Amazon to ensure **high availability and scalability**. The goal is to build a system that can

store and retrieve data efficiently while remaining **fault-tolerant**, meaning it can continue functioning even if some nodes fail.

Your system will support:

- **Consistent hashing for partitioning** — Ensuring data is evenly distributed across multiple nodes.
- **Data replication across multiple nodes** — Preventing data loss if a node fails
- **Quorum-based reads/writes** — A method to balance consistency and availability.
- **Failure detection using gossip protocol** — Nodes communicate and share information about failures.
- **Vector clocks for conflict resolution** — Handling concurrent updates without losing data.
- **Merkle trees for data synchronization** — Efficiently checking and fixing inconsistencies between replicas.

By the end of this project, you will deploy multiple **distributed database nodes** and test different configurations to understand **real-world trade-offs** between **consistency, availability, and partition tolerance (CAP theorem).**

7. Anonymity network using onion routing — Tor

   Paper: https://www.ieee-security.org/TC/SP2020/tot-papers/syverson-1997.pdf

   ABSTRACT: Develop an **anonymity network** using **onion routing**, enabling **secure and private communication** over a distributed network. The system should support **multi-layered encryption**, where messages are encrypted in layers and relayed through a **path of randomly selected nodes**, ensuring that no single node knows both the source and destination. Implement **path selection, end-to-end encryption, and traffic routing** to enhance privacy and security. The system should also address **traffic analysis resistance, load balancing, and bandwidth optimization**, while incorporating **logging and auditing mechanisms** for monitoring network health without compromising anonymity.

8. Distributed Hash table — Chord

Paper: https://pdos.csail.mit.edu/papers/ton:chord/paper-ton.pdf

ABSTRACT: Develop a **decentralized Distributed Hash Table (DHT)** using **Chord**, enabling **efficient lookup and routing** in a peer-to-peer network. The system should support **network construction, consistent hashing, node failure recovery, and efficient search mechanisms** to ensure scalability and robustness. Implement additional features such as **fault tolerance to handle dynamic node join/leave scenarios and support for data operations** to enhance usability and reliability in distributed environments.

9. Decentralised DHT — CAN

   ABSTRACT: Develop a **decentralized Distributed Hash Table (DHT)** using **Content Addressable Network (CAN)** to enable efficient **key-value storage and retrieval** in a **distributed environment**. The system should support **zone partitioning, routing algorithms, fault tolerance, and load balancing** to ensure scalability and reliability. Implement additional features such as **data encryption for security, data integrity checks to prevent corruption, and caching & replication strategies** to optimize performance and enhance system resilience.

10. Decentralised overlay network — Tapestry

    In this project, you will design and implement a **decentralized overlay network** inspired by **Tapestry**, focusing on key functionalities such as **efficient routing**, **resource location,** and handling **dynamic node memberships**. Your system should be *robust*, *scalable*, and adhere to the principles of a *decentralized network*. You will also need to ensure **fault tolerance** and **adaptability to node failures or additions**. The deliverables include a written report detailing your implementation, challenges faced, and results, along with a presentation. This project aims to deepen your understanding of decentralized systems and their *practical applications*.

11. Distributed File Sharing System — BitTorrent

    ABSTRACT: Implement a **peer-to-peer (P2P) file-sharing system** based on **BitTorrent**, where files are split into **chunks** and distributed across multiple peers for **efficient downloading**. The system should include a **tracker server** to maintain active peer lists and facilitate **peer discovery**, along with a **torrent**

client capable of **initiating downloads, requesting chunks, and sharing pieces with other peers**. Implement **peer communication protocols** to handle **handshakes, chunk requests, and uploads**, ensuring that peers can act as both **seeders and leechers**. Additionally, students should simulate **multiple peers** and compare file distribution speed with a **centralized file transfer model**, demonstrating the scalability benefits of **P2P networking**.

12. Distributed File Sharing System — Napster

    Paper Link: https://www.scribd.com/document/456414891/P2P-Design-Document

    Students can refer to the above paper for implementation ideas for napster style distributed file sharing system.
    ABSTRACT: Napster was one of the first widely used peer-to-peer (P2P) file sharing systems, which allowed users to share music files directly with each other. The older versions of the systems used to have a single server which stores the files in its directory that are received from the clients, but napster enables direct peer to peer file transfer. Students have to implement a distributed file sharing system similar to napster which includes main features like decentralized file sharing which enables users to share files directly from their computers with others on the network, centralized indexing server where a central server is used to maintain an index of files available on the network and store file metadata and peer addresses, facilitating search and discovery and finally Peer-to-Peer Communication protocol which facilitated direct file transfers between users, bypassing traditional client-server architectures. The server must index the content of all the peers (i.e., Clients) that register with it. and also provide search facility to peers. The server must register and listen to clients that wants to download (
    As a server) and search for a filename and ask to download it *(As a client).* The peers must provide the following interface to the users:

    1. Register – registers the file into the server
    2. Search – searches the server for a file and returns the list of Clients
    3. Download – downloads the file from another Client

13. Distributed web crawler/search engine

In this project, you will design and implement a **distributed web crawler** and **search engine**, inspired by systems like *Google's early architecture*. Your system should be able to **crawl web pages**, **index content**, and **support basic search queries** in a **distributed** manner. You will need to ensure *scalability*, *efficiency*, and *fault tolerance*, handling tasks like **URL distribution**, **data partitioning**, and **query processing across multiple node**s. Deliverables include a working prototype, a report detailing your design, implementation, and challenges, and a presentation to share your findings. This project will help you understand the complexities of building large-scale distributed systems for *real-world applications*.

You may also consider using the Map Reduce framework for parts of the project.

14. Real-Time Collaborative Text Editor using CRDT

    ABSTRACT: Develop a **real-time collaborative editing system** using **Conflict-Free Replicated Data Types (CRDTs)** to enable **synchronous and asynchronous editing** across distributed clients. The system should support **real-time synchronization, data structure implementation, and conflict resolution**, ensuring consistency across multiple users. Implement **undo/redo mechanisms, access control, and offline editing**, with seamless **sync-on-reconnection** to maintain a smooth user experience. The system should explore two different CRDT approaches: **state-based CRDTs (strong consistency)** and **operation-based CRDTs (high scalability)** to balance performance and reliability in distributed environments.

15. Leader election algorithm — RAFT

    In this project, you will implement the **Raft leader election algorithm** to manage a distributed system. Your task is to create a system where nodes can **communicate**, **detect failures**, and **elect a new leader automatically**. The system should ensure **consistency** and handle scenarios like **leader failures** and **network partitions**. You will need to develop a working prototype that demonstrates the Raft algorithm in action. The deliverables include a written report detailing your implementation, challenges faced, and results, along with a presentation. This project aims to deepen your understanding of leader election in distributed systems and their *practical applications*.

16. Distributed Cache — Redis

    About Redis - https://www.youtube.com/watch?v=fmT5nlEkl3U&t=50s

    ABSTRACT: Redis (Remote Dictionary Server) is an **in-memory key-value store** widely used for **caching, session management, real-time analytics, and message brokering**. It supports **data persistence**, **replication**, and **sharding**, making it highly scalable for distributed workloads. Redis uses an **asynchronous event-driven architecture** for high throughput and low latency. This project aims to build a **distributed caching system** inspired by **Redis**, focusing on **high availability, scalability, and low-latency access** to frequently requested data. The system will implement **data partitioning, replication, eviction policies, and a custom client API** to provide efficient caching similar to Redis but without directly using Redis itself.

17. Distributed Federated Learning

    Traditional machine learning requires **centralized data collection**, which can lead to **privacy concerns, bandwidth limitations, and security risks**. **Federated Learning (FL)** addresses these challenges by enabling **model training across multiple decentralized devices** without transferring raw data.

    However, **scalability, communication efficiency, and fault tolerance** remain key bottlenecks. This project proposes a **Distributed Federated Learning (DFL) framework**, leveraging **decentralized coordination and communication-efficient updates**, to enhance **scalability, privacy, and robustness**.

    By the end of this project, we aim to:

    Develop a federated learning framework that supports multiple distributed nodes.

    Optimize model aggregation to reduce network overhead and enhance training performance.

    Improve privacy & security by integrating techniques like **differential privacy & secure aggregation**.

    Test on real-world datasets (e.g., healthcare, IoT, and financial datasets).

    This proposal focuses on making **federated learning more efficient, decentralized, and resilient** while ensuring data privacy and reducing communication costs.

18. Distributed Messaging Queue — RabbitMQ

    In this project, you will **design and implement a distributed messaging queue system** using **RabbitMQ**, a widely used **message broker** that enables reliable communication between different services. The goal is to build a system that efficiently **queues, processes, and delivers messages** in a **scalable, fault-tolerant, and asynchronous** manner.

    Your system will support:

    - **Message queuing and delivery** — Ensuring messages are processed in order.
    - **Publisher-Subscriber and Work Queue models** — Two different messaging patterns.
    - **Message persistence** — Ensuring messages are not lost if a failure occurs.
    - **Acknowledge & Retry mechanisms** — Handling failures gracefully.
    - **Load balancing with multiple consumers** — Distributing work efficiently.
    - **Clustering & High Availability** — Running RabbitMQ across multiple nodes.
    - **Exchange types (Direct, Topic, Fanout, Headers)** — Different ways to route messages.

    By the end of this project, you will deploy multiple **RabbitMQ nodes** and test different messaging patterns to understand real-world **trade-offs between reliability, scalability, and performance**.

19. Distributed Chess Engine

    ABSTRACT: Develop a **distributed chess engine** that utilizes **parallel computing and distributed search techniques** to efficiently explore the game tree and evaluate moves. The system should support **scalable task distribution, concurrency management, and workload balancing** to improve computational efficiency. Students can implement **various search algorithms and optimizations**, such as **parallel game tree exploration, heuristic-based pruning, or transposition table sharing**, based on their design choices. Additionally, the system should incorporate **fault tolerance mechanisms** to handle node failures and ensure reliable computation in a distributed environment.

20. Distributed Task Scheduler

ABSTRACT: A distributed task scheduler is a software tool that manages task execution across multiple servers or nodes in a distributed environment. It is designed to efficiently distribute tasks across multiple worker instances, ensuring reliable assignment, execution, and monitoring in a fault-tolerant manner and coordinates job assignments, monitors worker health, and handles job retries in case of failures. Students can use various algorithms to implement the task scheduling, leader election, leader-worker coordination or selection of workers to assign tasks like round robin, random assignment, priority based assignment etc. The key features to be implemented are:

- **Task Queuing:** Placing tasks in a queue and monitoring it to decide when to execute each task based on predefined rules.
- **Task Definition:** Defining tasks based on the work that needs to be done, with dependencies on other tasks or specific execution conditions.
- **Task Execution:** Workers pull tasks from the queue and execute them, with the scheduler assigning tasks based on server load, task priority, and resource availability.
- **Monitoring and Reporting:** Tracking the status of each task and providing alerts to administrators if a task fails, with options to retry.
- **Failure Recovery Mechanism:** This includes strategies for handling failed tasks or worker nodes, such as re-queuing failed jobs or redistributing workloads to other available workers.
- **Scalability:** Adding additional worker nodes to handle more tasks as demand increases, with the scheduler dynamically adjusting to ensure efficient resource use.
- **Load Balancing:** A load balancer integrated to evenly distribute workloads across worker nodes based on their capacity and current load, optimizing resource utilization.

You could also implement REST/gRPC APIs for task submission and monitoring.

21. Sharded key value store — MongoDB

ABSTRACT: This project involves designing and implementing a **distributed key-value storage system** inspired by **MongoDB**. The system will feature horizontal scalability, fault tolerance, and efficient query handling—all while maintaining consistency and availability across a distributed environment. Built from scratch using distributed systems principles, the project will implement **replication, sharding, and consistency mechanisms** without depending on MongoDB itself.

22. Real Time Video Synchronization

    ABSTRACT: Develop a **real-time video synchronization system** enabling **simultaneous playback across multiple users** with minimal latency. The system should support **synchronized start, pause, seek, and playback speed adjustments**, ensuring all participants experience the video in sync. Implement additional features such as **load balancing to manage concurrent users, scalability to support large groups, and failure handling mechanisms to recover from network disruptions or client crashes**, ensuring a seamless viewing experience.

23. File Synchronizer — Unison or Tra

    Reference: https://web.mit.edu/6.033/2005/wwwdocs/papers/unisonimpl.pdf

    ABSTRACT: File synchronization tools like **Unison** and **Tra** enable efficient **bi-directional synchronization** of files across multiple machines. They are used for **backup, version control, and distributed collaboration**. These tools ensure that changes in one system are accurately reflected across all synchronized nodes while handling **conflicts, network failures, and version mismatches**. This project aims to develop a **distributed file synchronization system** inspired by **Unison or Tra**, enabling real-time or scheduled synchronization of files across multiple devices. The system will handle **conflict resolution, version tracking, network latency issues, and file compression** to optimize synchronization efficiency.

24. Distributed Tracing System — Dapper

    ABSTRACT: Develop a **distributed tracing system** that enables **end-to-end request monitoring and performance analysis** across a distributed architecture. The system should include a **central trace collector** responsible for **aggregating, storing, and analyzing traces** from multiple services while ensuring **consistency and synchronization** across distributed nodes. The system should support **trace propagation, event correlation, and latency measurement** to provide insights into request flows and system bottlenecks. Students can implement **various tracing techniques**, such as **sampling strategies, context propagation, and log aggregation**, while maintaining **synchronization mechanisms** to ensure accurate event ordering. Additionally, the system should incorporate **fault tolerance mechanisms** to handle node failures and ensure reliable trace collection in large-scale distributed environments.