THAPAR INSTITUTE

OF ENGINEERING AND TECHNOLOGY

(DEEMED TO BE UNIVERSITY)

Admission Batch: 2023 **Session: Even 2024-25**

Project Report
on
Decentralized File Storage Networks

OPERATING SYSTEMS (ULC-404)

Submitted By:
Saksham Mittal ([redacted])
[redacted]
B.E. Electrical and Computer Engineering
Semester - 4

DEPARTMENT OF ELECTRICAL AND INSTRUMENTATION ENGINEERING

THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
PATIALA, PUNJAB

Abstract

dFSN (Decentralized File Storage Network) is a peer-to-peer distributed file storage system designed to enable secure, encrypted storage of user files across a network of independent storage nodes. The system ensures that only the original user can reconstruct the complete file, preventing unauthorized access even by storage providers themselves. Utilizing **gRPC** for network communication and **Sodium** for encryption, dFSN operates without centralized intermediaries. Storage clusters are managed dynamically through a leader-based architecture, supporting resilience and failover. This report presents the motivations, architectural design and workflow of dFSN, outlining its goals of privacy, decentralization and potential commercial applications as an alternative to centralized cloud storage services.

Introduction

The increasing reliance on centralized cloud storage platforms like *Google Drive* and *Dropbox* raises concerns over data privacy, vendor lock-in and system vulnerabilities stemming from centralized control. To address these issues, **dFSN (Decentralized File Storage Network)** is proposed as a peer-to-peer, distributed file storage solution focused on user control, privacy and resilience.

dFSN allows users to split and encrypt their files into multiple chunks and disperse them across independently operated storage nodes. Each chunk is encrypted such that no single storage node can reconstruct the original file. Only the user retains knowledge of how the chunks are distributed, significantly enhancing security. Communication between nodes and clients is facilitated via **gRPC**, while file encryption is handled using the **Sodium** cryptographic library.

The network architecture employs a cluster-based model where each storage cluster is coordinated by a leader node. This leader node also runs a lightweight “tracking server” process to allocate storage and manage chunk replication, thereby eliminating the need for any centralized authority. Resilience mechanisms such as automatic leader election and failover support are built-in to maintain system availability despite node failures.

Beyond its core functionality, dFSN envisions commercial applications where businesses with surplus storage can participate by offering guaranteed uptime nodes for additional reliability, creating a decentralized storage economy. This introduction outlines the motivations, architectural vision and intended impact of the dFSN project, setting the foundation for the technical discussions that follow.

Notes:

- *This project differs significantly from the well-known **InterPlanetary File System (IPFS)**, a basic comparison between the two is given in later sections.*
- *This project is under active development and as of writing is open to contributions and licensed under the MIT License - **https://github.com/B4S1C-Coder/Decentralized-File-Storage***

Comparison with Existing Solutions

---

dFSN (Decentralized File Storage Network) and **IPFS** (InterPlanetary File System) both aim to solve similar problems of decentralization of file storage. However they differ in key aspects of architecture, file management and use cases.

Feature	**dFSN**	**IPFS**
Architecture	Peer-to-peer with leader-based clusters	Peer-to-peer with DHT and no central tracker
File Storage	Split and encrypted chunks	Whole files stored with CID
Redundancy	Replication across nodes with leader failover	Network-dependent, nodes must pin content
Privacy & Security	Built-in encryption, user-controlled	No built-in encryption, public by default
User Control	Users choose storage nodes/clusters	No user control over storage placement
Fault Tolerance	Leader nodes and failover mechanisms	Dependent on voluntary node participation

Table 1: Comparison between dFSN and IPFS

To conclude, dFSN emphasizes privacy, control, and fault tolerance with its encrypted file chunks, leader-based clusters, and user-controlled storage placement. In contrast, IPFS is designed for open, public distribution of files using a DHT-based system that does not provide built-in encryption or control over where files are stored.

Technology Stack Overview

- **gRPC (Google Remote Procedure Calls):** is an open-source framework for high-performance, language-agnostic remote procedure calls (RPC). It uses HTTP/2 for transport, Protocol Buffers (protobuf) for serialization, and provides features such as bi-directional streaming and multiplexing.
- **Libsodium (Sodium):** is a modern cryptographic library that provides high-level cryptographic operations such as encryption, decryption, hashing, and public-key cryptography. It is designed for security, usability, and portability.
- **SQLite:** is a lightweight, serverless, self-contained relational database engine. It is widely used for local storage due to its simplicity and minimal setup requirements.

System Design

The following are the two variants of dFSN, the first one is meant to be used within the private networks of an organization and consists of tracking servers (some points of centralization), to allow organizations to retain and govern how their network operates.

The second one eliminates the tracking servers entirely and ensures true decentralization, meaning no single party can control the network and only have their operated nodes under their domain of control.

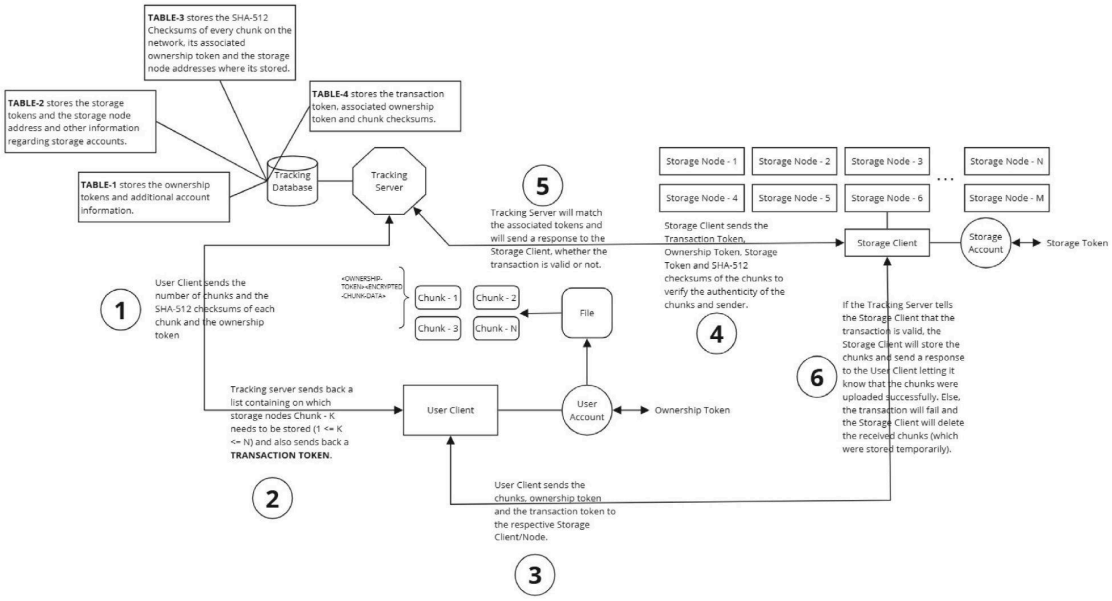

Fig. 1: Organization Moderated dFSN

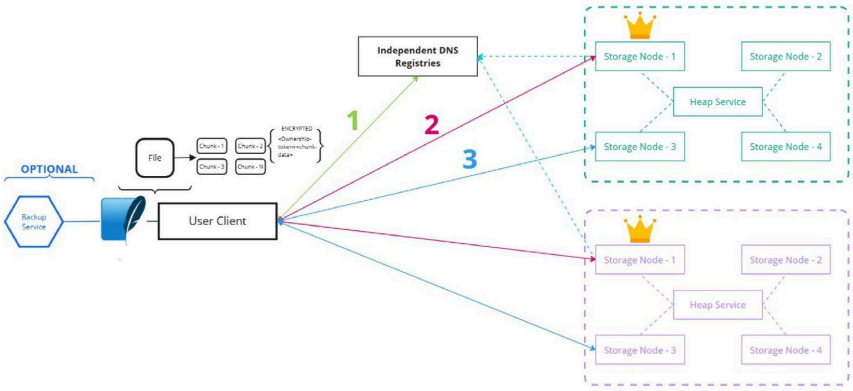

Fig. 2: Open dFSN (no point of centralization)

Note: The goal of the project is to also make the two architectures inter-operable for economic gains.

Technical Implementation

As of writing, excluding the code for User Interfaces, the project contains (including third party headers) 54,483 lines of code across 47 C++ source, header and CMake files. As a result it is not possible to show the entire source code here and only the relevant snippets will be shown in the following sections.

Repository - *https://github.com/B4S1C-Coder/Decentralized-File-Storage*

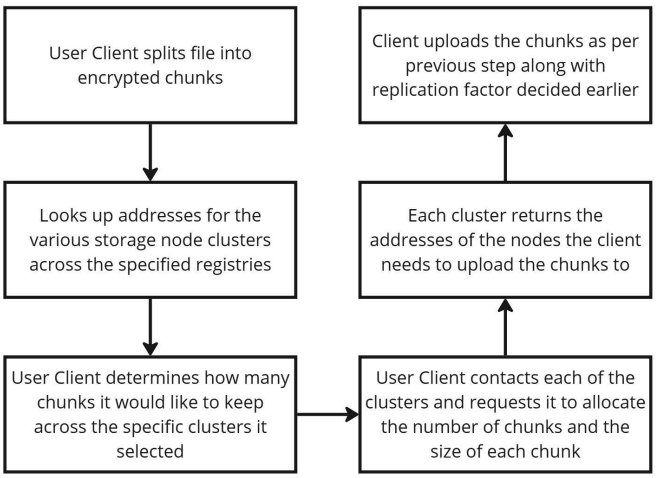

Fig. 3: User perspective workflow

File Encryption & Decryption via libsodium

```
std::vector<char>
fsn::StreamEncryptorDecryptor::xchacha20poly1305_encrypt(std::vector<char>&
inputBuffer) {

  std::vector<char> encryptedBuffer(inputBuffer.size() +
crypto_aead_xchacha20poly1305_ietf_ABYTES);
  std::vector<char> inputBufferHash =
fsn::util::primitive_calculateSHA512Hash(inputBuffer);

  unsigned long long encrypted_len = 0;

  crypto_aead_xchacha20poly1305_ietf_encrypt(
    reinterpret_cast<unsigned char*>(encryptedBuffer.data()), &encrypted_len,
    reinterpret_cast<unsigned char*>(inputBuffer.data()), inputBuffer.size(),
    reinterpret_cast<unsigned char*>(inputBufferHash.data()),
inputBufferHash.size(), NULL, m_nonce, m_key
  );

  encryptedBuffer.resize(encrypted_len);
```

```
  return encryptedBuffer;
}


std::optional<std::vector<char>>
fsn::StreamEncryptorDecryptor::xchacha20poly1305_decrypt(
  std::vector<char>& encryptedBuffer, std::vector<char>& messageHash
) {
  std::vector<char> decryptedBuffer(encryptedBuffer.size());
  unsigned long long decrypted_len = 0;

  bool decryptionSuccess = (crypto_aead_xchacha20poly1305_ietf_decrypt(
    reinterpret_cast<unsigned char*>(decryptedBuffer.data()), &decrypted_len, NULL,
    reinterpret_cast<unsigned char*>(encryptedBuffer.data()),
encryptedBuffer.size(),
    reinterpret_cast<unsigned char*>(messageHash.data()), crypto_hash_sha512_BYTES,
m_nonce, m_key
  ) == 0);

  if (decryptionSuccess) {
    decryptedBuffer.resize(decrypted_len);
    return decryptedBuffer;
  } else {
    return std::nullopt;
  }
}
```

File Splitting (and simultaneous encryption)

```
int fsn::SequentialFileSplitter::singleThreadedSplit(const std::string&
outputDirPath) {

  std::vector<char> buffer;
  buffer.reserve(m_bytesPerChunks);
  int chunkCount = 1;

  std::vector<char> token = fsn::util::primitive_generateRandomToken();

  while (true) {

    // Load the chunk data
    buffer.resize(m_bytesPerChunks);
    m_currentFile->read(buffer.data(), m_bytesPerChunks);
    std::streamsize dataSize = m_currentFile->gcount();
```

```
    if (dataSize == 0) {
      break;
    }

    buffer.resize(dataSize);

    // Encrypt the chunk data
    std::vector<char> encryptedBuffer =
m_streamEncDec.xchacha20poly1305_encrypt(buffer);

    // Calculate hash of the data
    std::vector<char> hash = fsn::util::primitive_calculateSHA512Hash(buffer);

    size_t dataEnd = hash.size() + token.size() + sizeof(size_t) +
fsn::util::to_paddedString(dataSize).size() + encryptedBuffer.size() - 1;

    // Construct the metadata for the chunk
    ChunkMetadata metadata(dataEnd, hash, token);
    std::vector<char> metedata_bytes = metadata.construct();

    // Create the final buffer to be written (without encryption)
    std::vector<char> encrypted_final;
    encrypted_final.reserve(metedata_bytes.size() + encryptedBuffer.size());
    encrypted_final.insert(encrypted_final.end(), metedata_bytes.begin(),
metedata_bytes.end());
    encrypted_final.insert(encrypted_final.end(), encryptedBuffer.begin(),
encryptedBuffer.end());

    std::vector<char> encryptedHash =
fsn::util::primitive_calculateSHA512Hash(encrypted_final);
    std::string chunkFileName = outputDirPath + "/" +
fsn::util::bytesToHex(encryptedHash);
    std::ofstream outfile(chunkFileName, std::ios::binary);

    if (!outfile.is_open()) {
      fsn::logger::consoleLog("Unable to write chunk to disk. \n",
fsn::logger::ERROR);
      return -1;
    }

    outfile.write(encrypted_final.data(), encrypted_final.size());
    outfile.close();

    m_chunkFiles.push_back(chunkFileName);
    fsn::logger::consoleLog("Chunk written to disk - " + chunkFileName);
  }
```

```
  return 0;
}
```

User side gRPC Transmission (putting and retrieving chunks)

```
void fsn::ChunkTransmitter::ingestChunk(std::unique_ptr<std::vector<char>> chunk) {
  CommData commdata;
  commdata.set_chunkdata(chunk->data(), chunk->size());
  auto hash = fsn::util::primitive_calculateSHA512Hash(*chunk);
  commdata.set_packethash(hash.data(), hash.size());

  CommResp response;
  grpc::ClientContext context;

  // Making the gRPC call
  grpc::Status status = stub_->ingestChunk(&context, commdata, &response);

  if (status.ok()) {
    std::string message = (response.chunkaccepted()) ? "Chunk Accepted" : "Chunk
Rejected";
    fsn::logger::consoleLog(message, fsn::logger::INFO);
    return;
  } else {
    std::string message = "Failed to make gRPC call: " + status.error_message();
    fsn::logger::consoleLog(message, fsn::logger::FATAL);
    return;
  }
}


std::optional<std::unique_ptr<std::vector<char>>>
fsn::ChunkTransmitter::ejectChunk(const std::vector<char>& packetHash) {
  CommChunkReq req;
  req.set_packethash(packetHash.data(), packetHash.size());

  CommChunkRes res;
  grpc::ClientContext context;

  grpc::Status status = stub_->ejectChunk(&context, req, &res);

  if (status.ok()) {
    fsn::logger::consoleLog("Chunk Received.", fsn::logger::INFO);
    return std::make_unique<std::vector<char>>(res.chunkdata().begin(),
res.chunkdata().end());
  } else {
    fsn::logger::consoleLog("gRPC call failed - " + status.error_message(),
```

```
fsn::logger::ERROR);
    return std::nullopt;
  }
}
```

Server side gRPC Transmission (accepting and sending chunks)

```
::grpc::Status fsn::ChunkIngestionImpl::ingestChunk(::grpc::ServerContext* context,
const ::CommData* req, ::CommResp* res) {

  // Extract data, grpc represents "bytes" (defined in proto) as "const
std::string"
  std::vector<char> chunkData(req->chunkdata().begin(), req->chunkdata().end());
  std::vector<char> packetHash(req->packethash().begin(), req->packethash().end());

  if (packetHash.size() != crypto_hash_sha512_BYTES) {
    fsn::logger::consoleLog("packetHash size = " +
std::to_string(packetHash.size()));
    fsn::logger::consoleLog("Chunk rejected as invalid hash provided.",
fsn::logger::WARN);
    res->set_chunkaccepted(false);
    return grpc::Status(::grpc::StatusCode::INVALID_ARGUMENT, "Invalid packet hash
size.");
  }

  // Verify hash of chunk data
  std::vector<char> calculatedHash =
fsn::util::primitive_calculateSHA512Hash(chunkData);
  bool hashesMatch = (
    sodium_memcmp(calculatedHash.data(), packetHash.data(),
crypto_hash_sha512_BYTES) == 0
  );

  if (!hashesMatch) {
    fsn::logger::consoleLog("Chunk rejected as hashes do not match.",
fsn::logger::WARN);
    res->set_chunkaccepted(false);
    return grpc::Status::CANCELLED;
  }

  // Store the chunk on disk (TO-DO: Make file name unique)
  std::ofstream chunkFile("data/chunk_" + fsn::util::bytesToHex(packetHash) +
".bin", std::ios::app | std::ios::binary);
  chunkFile.write(chunkData.data(), chunkData.size());
  chunkFile.close();
```

```
  fsn::logger::consoleLog("Chunk Accepted.", fsn::logger::INFO);

  res->set_chunkaccepted(true);
  return grpc::Status::OK;
}

::grpc::Status fsn::ChunkIngestionImpl::ejectChunk(::grpc::ServerContext* context,
const ::CommChunkReq* req, ::CommChunkRes* res) {
  std::vector<char> packetHash(req->packethash().begin(), req->packethash().end());

  if (packetHash.size() != crypto_hash_sha512_BYTES) {
    fsn::logger::consoleLog("packetHash size = " +
std::to_string(packetHash.size()));
    fsn::logger::consoleLog("Invalid hash provided. Request rejected.",
fsn::logger::WARN);
    res->set_chunkdata("");
    return grpc::Status(::grpc::StatusCode::INVALID_ARGUMENT, "Invalid packet hash
size");
  }

  std::string fileName = "data/chunk_" + fsn::util::bytesToHex(packetHash) +
".bin";

  std::ifstream chunkFile(fileName, std::ios::binary);
  if (!chunkFile) {
    fsn::logger::consoleLog("Chunk not found.", fsn::logger::ERROR);
    res->set_chunkdata("");
    return grpc::Status(::grpc::StatusCode::NOT_FOUND, "Chunk not found.");
  }

  chunkFile.seekg(0, std::ios::end);
  std::streamsize chunkSize = chunkFile.tellg();
  chunkFile.seekg(0, std::ios::beg);

  std::vector<char> chunkData(chunkSize);
  if (!chunkFile.read(chunkData.data(), chunkSize)) {
    fsn::logger::consoleLog("Unable to read chunk", fsn::logger::ERROR);
    res->set_chunkdata("");
    return grpc::Status(::grpc::StatusCode::INTERNAL, "Unable to read chunk.");
  }

  res->set_chunkdata(chunkData.data(), chunkData.size());
  return grpc::Status::OK;
}
```