File Synchronization System

1st Dat Nguyen Tran Quoc (10422017)* 2nd Giap Nguyen Minh (10422024)* 3rd Xuan Nguyen Phu (10422065)*
Department of Computer Science
Vietnamese-German University
Binh Duong, Vietnam
10422017@student.vgu.edu.vn
10422024@student.vgu.edu.vn
10422065@student.vgu.edu.vn

4th Quy Tran Phuc (10422071)*
5th Nhat Hoang Quang (10422060)*
Department of Computer Science
Vietnamese-German University
Binh Duong, Vietnam
10422071@student.vgu.edu.vn
10422060@student.vgu.edu.vn

Abstract

This report presents the design and implementation of a distributed file synchronization system developed to maintain consistent file states across multiple client machines operating within a local network. Unlike centralized solutions, this system functions entirely on local storage, managing file content and metadata directly within the file system without relying on external databases. The application supports real-time directory monitoring, automatic conflict resolution, and efficient synchronization of file changes, ensuring that updates made on one device are accurately reflected on others. Through a combination of file watchers, version tracking mechanisms, and socket-based peer communication, the system provides a lightweight yet robust solution for collaborative environments requiring offline or LAN-based file sharing. This report outlines the core architecture, synchronization logic, and key design decisions supporting local-only, decentralized operation.

I. Problem Statement

In modern distributed environments, users often need to keep files synchronized across multiple devices and locations. Manual synchronization is error-prone and inefficient, especially when files are frequently updated or edited concurrently by different users. The challenge is to design a distributed file synchronization system that ensures data consistency, supports version control, resolves conflicts, and scales to multiple clients, all while being robust to failures and easy to use.

II. Required Functionality:

The file synchronization system is designed to provide reliable, consistent, and user-controllable file sharing across multiple clients within a local network. To achieve this, the system offers a comprehensive set of functionalities that ensure data integrity, support collaborative workflows, and provide users with clear control over synchronization behavior.

These functionalities address key challenges in distributed file management, including version tracking, conflict resolution, real-time monitoring, and coordination between clients. Rather than relying on external storage or databases, the system operates fully on local file systems, making it lightweight, privacy-preserving, and suitable for offline or LAN-based environments.

The command-line interface (CLI) forms the primary point of interaction, enabling users to initiate synchronization, monitor system status, inspect file history, and resolve conflicts. Behind the scenes, the system uses background processes to automate synchronization and manage file events with minimal user intervention.

The following subsections describe each core functionality in detail, along with the corresponding CLI commands that allow users to interact with and control the synchronization system.

A. Bidirectional Synchronization

The system supports real-time bidirectional synchronization across all connected clients. When a file is modified, created, or deleted on one client, the change is propagated to the server and then to other clients. This ensures consistency across the distributed system. Users can manually trigger synchronization using the `sync` command, though changes are usually detected and synchronized automatically via the background file watcher.

B. Version Control

Every file modification results in a new version being saved. This allows users to maintain a complete history of file changes over time. The command `versions` lists all available versions of a given file, while `download-version <filename> <version>` retrieves a specific version, and `restore <filename> <version>` replaces the current file with a previous version. This is useful for reverting unintended edits or recovering lost data.

C. Conflict Detection and Resolution

Conflicts arise when the same file is edited simultaneously on different clients before synchronization occurs. The system detects such cases and flags them as conflicts. Users can view all unresolved conflicts with the `conflicts` command. Each conflict is uniquely identified and can be resolved using the `resolve <conflict-id>` command, allowing the user to select which version to keep or to merge manually.

D. Command-Line Interface (CLI)

Users interact with the system through a robust and interactive CLI. It supports the following commands:

- `status` – Displays current sync status and recent activities.
- `list` – Lists all synchronized files with metadata such as version, last modified time, and conflict state.
- `delete <filename>` – Deletes a file from both the local directory and the synchronization state.
- `rename <oldname> <newname>` – Renames a file while preserving its history.
- `conflicts` – Lists all active conflicts with their respective file names and IDs.
- `resolve <conflict-id>` – Resolves a conflict manually.
- `versions <filename>` – Displays the version history of a specific file.
- `download-version <filename> <version>` – Downloads a specific historical version of a file.
- `restore <filename> <version>` – Reverts a file to a chosen version.
- `pause` / `resume` – Temporarily suspend and resume synchronization operations.
- `config` – Displays current configuration settings such as client ID and sync path.
- `help` – Displays all available commands and their usage.
- `quit` – Exits the application safely.
- `touch <filename>` - Create a blank file.
- `import` - Import a file from local directories.

E. Automatic File Monitoring

Each client continuously monitors a designated local folder for changes using a file system watcher. When a file is added, removed, or modified, the event is captured, and a sync is scheduled. This automation allows users to work without needing to manually initiate sync operations.

F. Pause and Resume Synchronization

To support user control and performance optimization, the system allows synchronization to be paused using the `pause` command. This is especially useful when performing large batch file operations. Synchronization can be resumed with the `resume` command, which restarts monitoring and sync processes without losing previously queued changes.

G. Multi-client Support

The system supports multiple clients operating simultaneously within the same network. Each client is uniquely identified and managed independently. The server resolves changes from all clients in a consistent order and ensures that conflict detection mechanisms are enforced when overlapping changes occur. To connect to the server within a network, users are required to type their username as to create a sync folder for synchronization process. Moreover, it is compulsory to type the correct IP address of the server to connect to the server.

H. Health Monitoring and Error Recovery

The system includes robust error handling and automatic retry mechanisms. If a sync operation fails (due to disconnection or file access issues), the error is logged, and the operation is retried. Health checks ensure each client maintains a stable connection to the server and can recover from transient failures without data loss.

I. Configuration Access

Users can access the system configuration using the `config` command. This displays essential settings including the local sync directory, client identifier, and operational flags such as pause state or conflict mode. Configuration is typically read from a local file on startup and can be modified between sessions.

III. System Architecture

A. Component Diagram

Figure 1 illustrates the overall architecture of the file synchronization system, which follows a client-server model. The system is designed to support distributed file syncing, conflict resolution, and version control across multiple clients using a centralized coordination server.

On the **client side**, the core components include:

- **CLI Interface:** Provides a command-line interface through which users interact with the application.
- **File Watcher:** Monitors local file changes in real-time.
- **Sync Manager:** Coordinates synchronization tasks, interacts with the API client, and resolves conflicts.
- **API Client:** Handles communication between the client and the server over HTTP.
- **Local Storage:** Stores all user files and their metadata locally.

On the **server side**, the architecture includes:

- **REST API Server:** Acts as the main entry point for all client requests and routes them to the appropriate backend services.
- **Queue Manager:** Manages background tasks such as file processing using asynchronous job queues.
- **File Processor:** Processes uploaded file chunks and reconstructs complete files.
- **File Storage:** Temporarily stores file chunks and completed files.
- **Chunk Alerter:** Notifies the system when file chunks are ready for processing.
- **Metadata Store:** Maintains metadata such as file versions, modification timestamps, and ownership details.

This modular and asynchronous architecture ensures scalability, robustness, and real-time file synchronization across clients, while also supporting conflict resolution and file versioning through centralized server coordination.

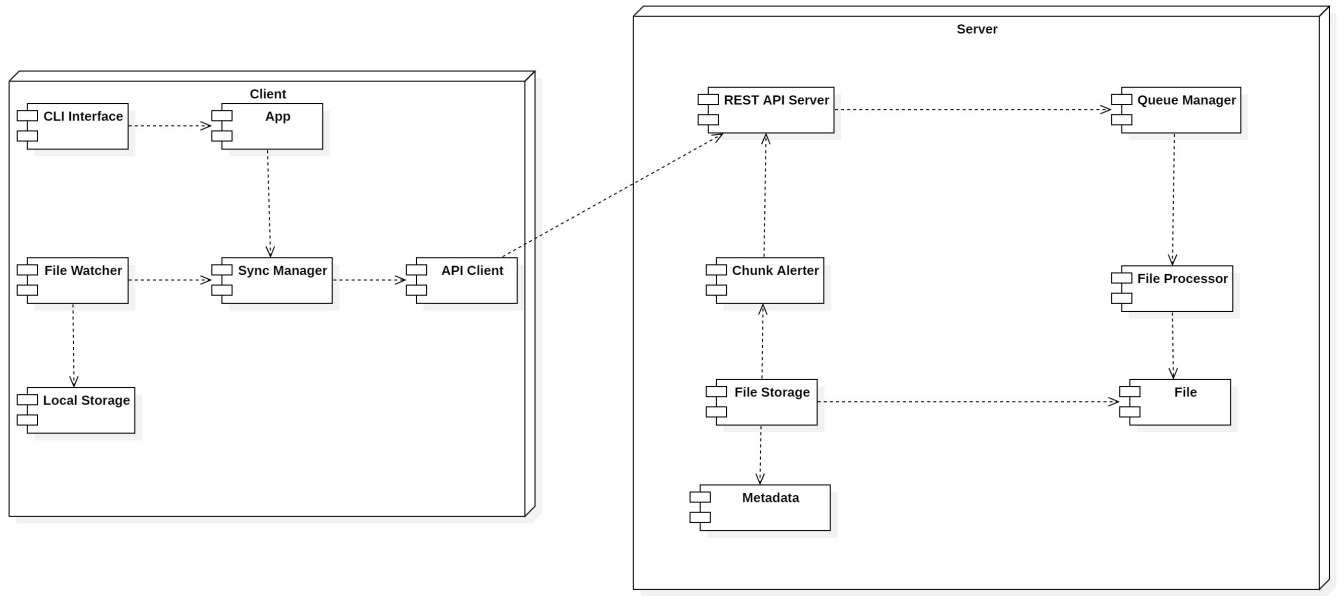


Fig. 1: Component Diagram Describes System Architecture

B. Flow Chart

Figure 2 provides an overview of the core functionalities and user interactions within the File Synchronization System. It visually represents how different events—such as user commands, file changes, and periodic file watching—trigger corresponding actions in both the client and server components.

The system continuously monitors local events (e.g., file changes, user input, scheduled polling) and reacts accordingly. For example:

- **File changes** on the client are detected and sent to the server for upload and storage.
- **User commands**, such as `sync`, `status`, or `quit`, trigger specific responses, including uploading files, showing system status, or exiting the system.

- The system uses a **polling mechanism** to check for updates from the server. If new files are detected, they are downloaded to the client.
- In case of **file conflicts** (e.g., concurrent modifications on different clients), the user is prompted to resolve them using a dedicated conflict resolution process.

Key operations—like uploading, downloading, displaying status, and resolving conflicts—are part of a closed loop that always returns to monitoring mode, ensuring the system remains responsive and synchronized in real time.

This diagram highlights important aspects of the system such as **bidirectional communication**, **conflict detection**, **status checking**, and **robust handling of user commands**, giving a clear picture of how the synchronization engine works under various conditions.

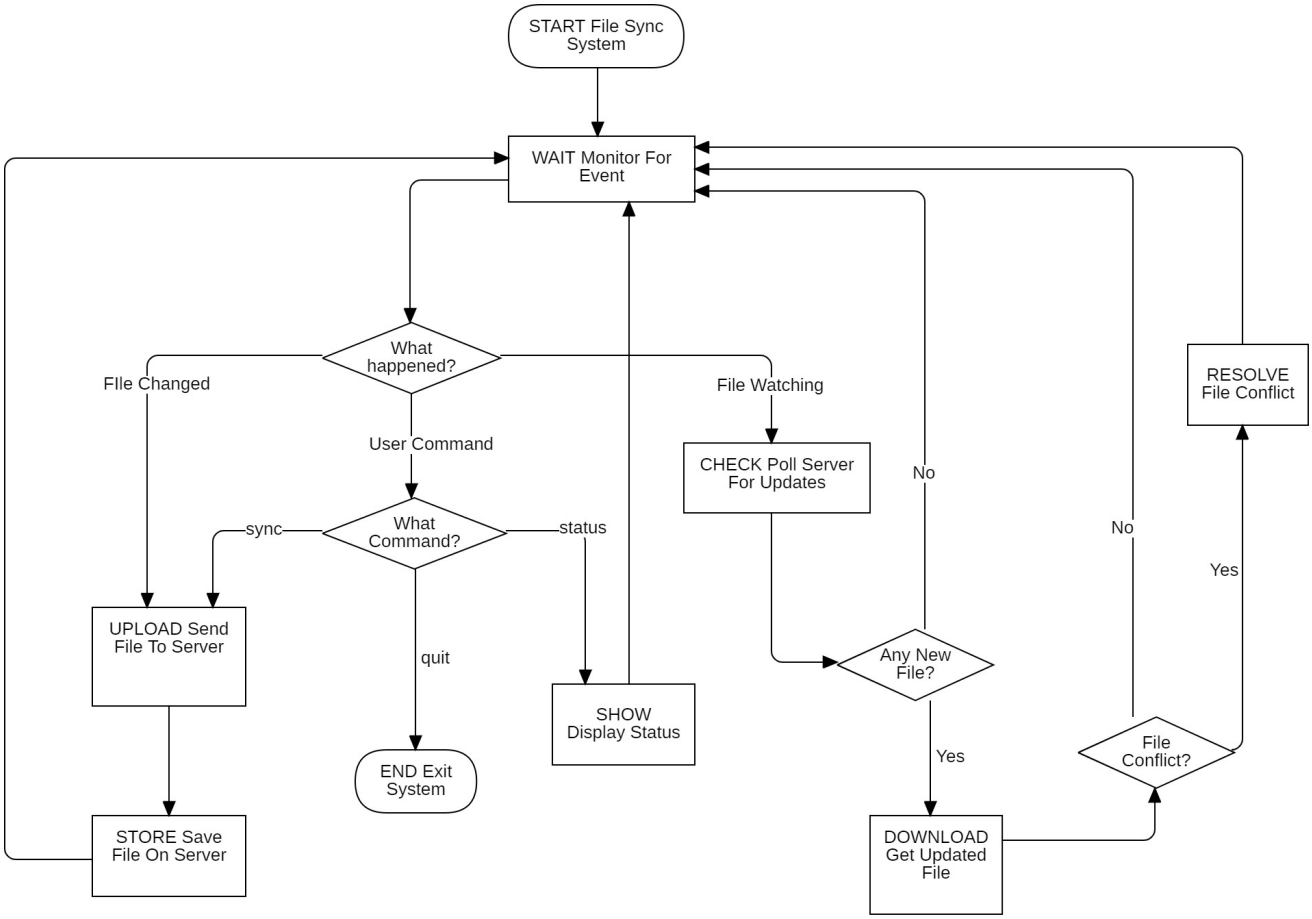


Fig. 2: Flow Chart describes the Flow of Activities

C. Project Structure

Our project is developed using JavaScript (Node.js) as the primary programming language, with a modular architecture separating the client-side and server-side responsibilities. The system is designed to perform reliable file synchronization across distributed clients, incorporating conflict resolution, chunked file transfers, and a centralized version-tracking mechanism.

To ensure scalability, maintainability, and ease of development, we have adopted a well-defined directory structure. The layout of the `file-synchronizer` system is described below:

The following is the structure of the `file-synchronizer` project, which includes both client and server-side components:

```
file-synchronizer/
 LICENSE
 README.md
 Diagrams.mdj                # UML or design diagrams
 FlowchartDiagram1.jpg       # System flowchart
```

```
client/
   package.json
   src/
      app.js              # Client entry point
      api/
         api-client.js    # Handles API requests to server
      sync/
         sync-manager.js # Core sync logic (conflict, sync, etc.)
      ui/
         cli-interface.js# Command-line interface for user
      watcher/
         file-watcher.js # Watches local folder for changes
   sync-folder/            # Main sync folder for this client
      ...                 # Synced files and subfolders
   downloads/              # Downloaded versions, etc.
   ...                     # Other client-specific folders
server/
   package.json
   src/
      api/
         server.js       # Express API server
      queues/
         queue-manager.js# Job queues (optional)
      storage/
         file-storage.js     # File and version storage logic
         metadata-storage.js # Metadata and conflict tracking
         files/              # Main file storage
         versions/           # Versioned file storage
         metadata/           # Metadata files (JSON, etc.)
         chunks/             # Chunked uploads
      test/
         test-system.js  # Server-side tests
      workers/
          chunk-alerter.js
          file-processor.js
```

This structure promotes modularity, easy navigation, and efficient maintenance throughout the development lifecycle.

IV. Distributed System Characteristics

A. Fault Tolerance

The system is designed to be fault-tolerant, ensuring continued operation and data consistency even in the presence of client or server failures, as well as concurrent modifications. The following are the three primary fault scenarios addressed by the system:

a) Client Crash or Disconnection: When a client crashes or disconnects unexpectedly, the rest of the system continues to operate without interruption. Other clients remain fully functional and can continue to synchronize with the server. Once the disconnected client is restarted, it automatically resumes synchronization from the point of failure. Local change logs and version tracking enable the system to identify unsynchronized changes and update them accordingly.

b) Server Crash or Unavailability: If the server becomes unavailable (e.g., due to a crash or network issue), clients will report a synchronization failure. A typical error message displayed in the CLI is:

```
Sync error: connect ECONNREFUSED 172.16.129.88:3000
Periodic sync failed: connect ECONNREFUSED 172.16.129.88:3000
```

Despite these errors, clients continue to monitor local files and queue any changes. Once the server is back online, synchronization resumes automatically. This ensures that no changes are lost and no manual intervention is needed beyond restarting the server.

c) Concurrent Modifications and Conflict Handling: When two or more clients modify the same file concurrently before synchronization, the system detects a conflict. Only the client whose version differs from the server's will be notified of the conflict. The conflicting client can execute the `conflicts` command to list all unresolved conflicts. Then, using the `resolve <conflict-id>` command, the user is shown a detailed comparison of their version versus the server version—including timestamps, file sizes, and file content. It is then up to the user to manually reconcile the differences by editing the local file, after which the conflict is cleared and the updated file can be synchronized normally.

The system ensures fault tolerance through the following mechanisms:

- **Crash recovery:** Clients and server can restart without data loss; unsynchronized changes are safely re-queued.
- **Persistent event queues:** Local file change events are stored persistently until successful synchronization is confirmed.
- **Resumable sync:** Clients automatically resume syncing after recovering from crashes or disconnections.
- **Conflict resolution interface:** Users are notified of conflicts and provided tools to manually resolve them with full visibility into the changes.
- **Graceful degradation:** In case of server failure, clients remain operational in offline mode and resume syncing when possible.

B. Scalability

The system is designed with scalability in mind, ensuring it can efficiently support multiple clients, large files, and growing synchronization demands without degradation in performance.

a) Multiple Clients and Devices: Any number of clients can connect to the synchronization server from different devices. Each client is initialized with the server's IP address and can join or leave the distributed system dynamically. This allows flexible participation from various machines across the same network without reconfiguring the server or other clients.

b) Stateless Client Architecture: Clients operate independently and are unaware of other connected clients. They only communicate with the central server, which handles synchronization, versioning, and conflict resolution. This decoupling allows clients to be added or removed at any time without disrupting the overall system operation, making the architecture highly modular and maintainable.

c) Performance Consistency: The system addresses scalability through multiple mechanisms. Background job queues handle file processing asynchronously, preventing blocking operations that could degrade performance:

Listing 1: Asynchronous job processing

```
this.alertQueue = new Queue('alert-queue', { ... });
this.fileQueue = new Queue('file-queue', { ... });

this.alertQueue.process(async (job) => { ... });
this.fileQueue.process(async (job) => { ... });
```

Chunked file uploads allow the system to handle large files efficiently by breaking them into manageable segments, reducing memory consumption and enabling resumable transfers. The stateless REST API design supports horizontal scaling by allowing multiple server instances to be deployed behind a load balancer.

d) Efficient Synchronization Protocol: The system uses an efficient polling-based synchronization mechanism. Clients periodically contact the server to check for file updates. Only modified files or chunks are transferred during synchronization, which minimizes bandwidth usage and reduces the server load. Chunked uploads also allow the system to handle large files in smaller parts, improving reliability and performance during high-volume data transfers.

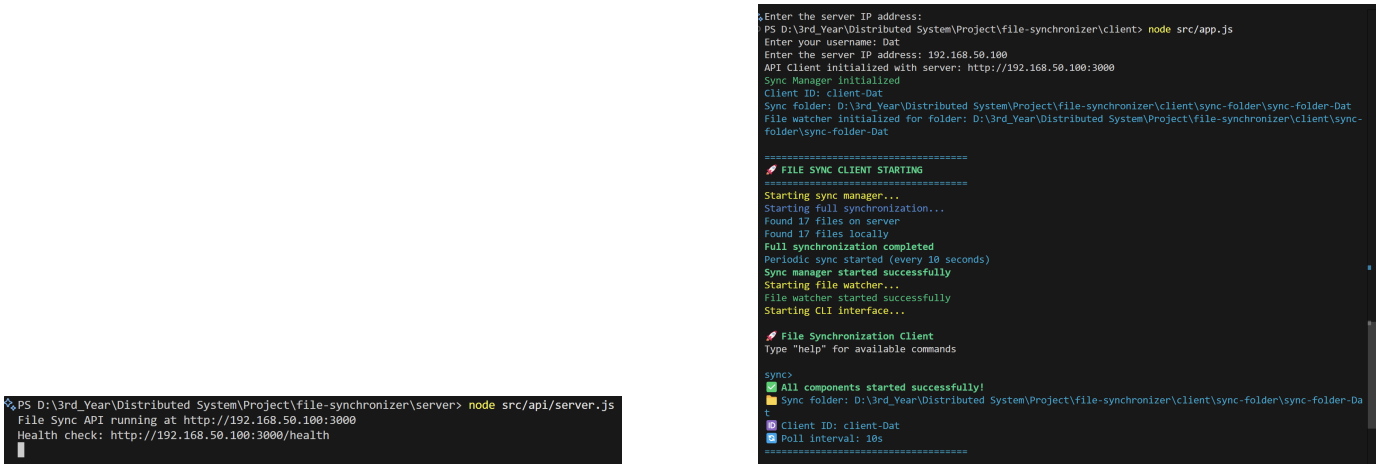
The scalability of the system is enabled through the following key features:

- **Asynchronous server processing:** The server uses job queues and worker threads to handle multiple file operations (upload, download, versioning) in parallel without blocking other clients.
- **Chunked file uploads:** Large files are divided into smaller parts (chunks) for upload, reducing memory consumption and supporting resumable transfers.
- **Dynamic client participation:** New clients can join the network at any time without prior registration or affecting existing clients.
- **Minimal inter-client dependency:** Clients only communicate with the server, not with each other, which simplifies scaling and fault isolation.
- **Server-side extensibility:** The server can be upgraded or horizontally scaled as the number of clients grows, including the use of load balancers or microservice decomposition.

V. System Result Presentation

A. Server initialization and client connection

First, a server is initialized so that clients can use the IP address of server to connect. To connect to the server, clients first need to type the username, which creates a synchronization folder. Then clients needs to correctly type the IP address of server to successfully connect to the server. The following figures describes the state after successful initialization of server and connection of clients



(a) Successful Server Initialization

(b) Successful Client Connection to Server

Fig. 3: Successful Client-Server Connection

B. Auto Synchronization Process

The system is designed to automatically synchronize after a short fixed time. Using CRUD operations, the server automatically synchronizes with the clients after the operations occur.

1) File Creation: describes the result of synchronization after the client creates a blank file. After creation, the file is synchronized to the server and stored in the server file storage for further storage Figure 4

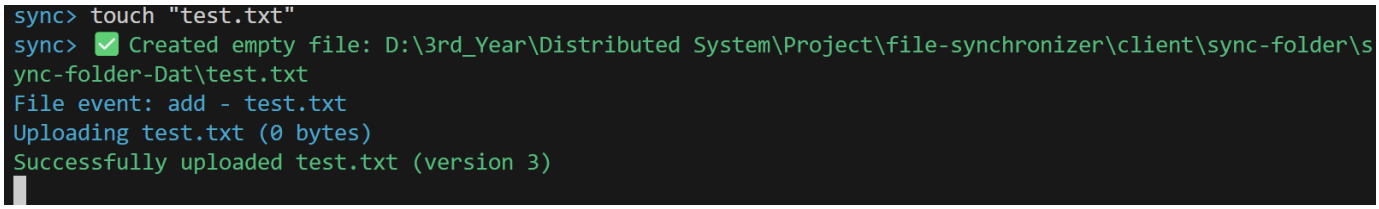


Fig. 4: User Create a Blank File

2) File Read: Figure 5 describes the result when clients need to list the versions of a specific file. The system returns a list of versions of files, which is stored in the **versions** folder, while the information of timestamp and client ID is stored in the **metadata** folder, where the metadata of each file is stored.

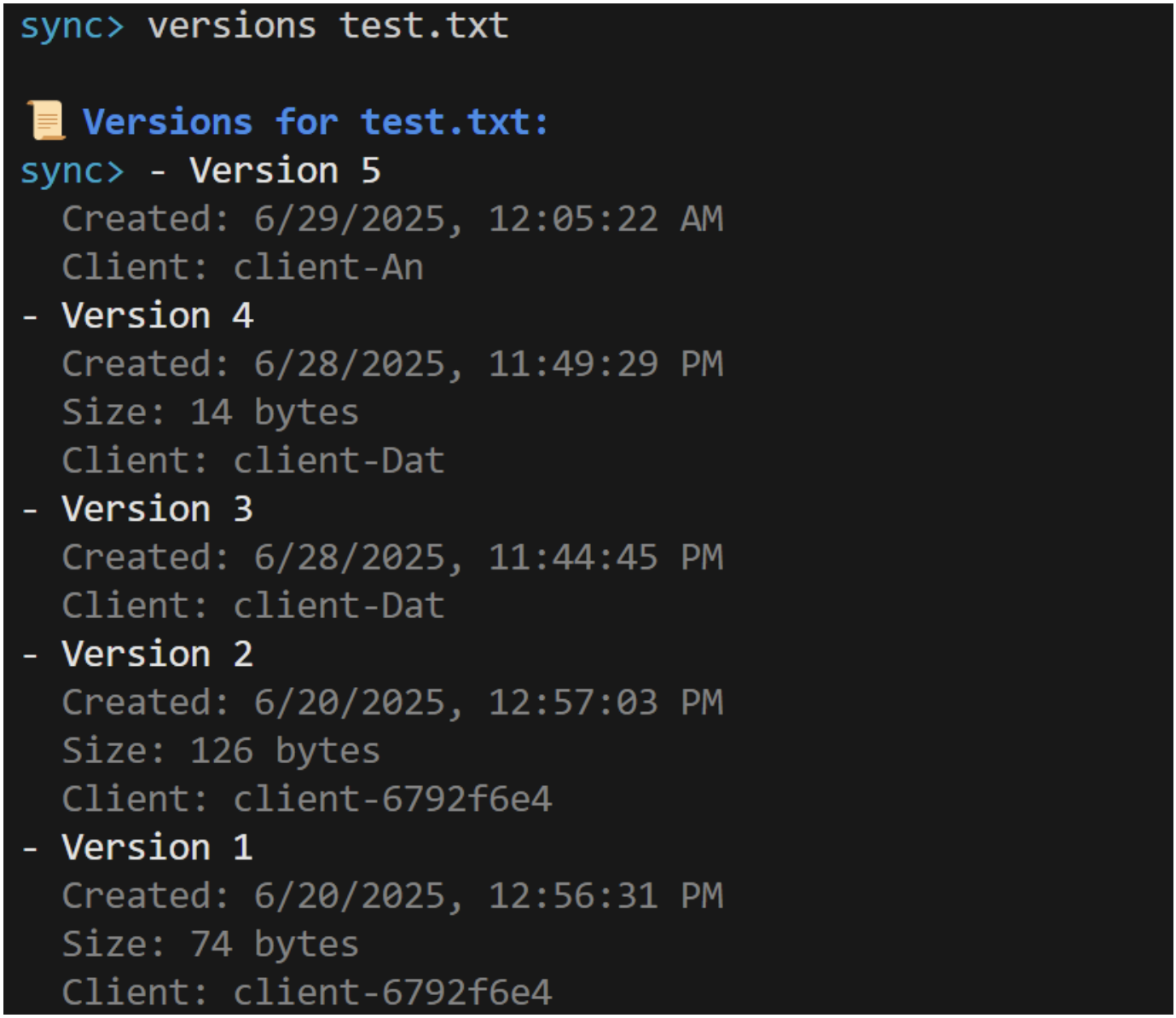


Fig. 5: User Read File Versions of a File

3) File Update: Figure 6 describes the result when clients updates a file. Then the file is synchronized and updated to the server and a new version is stored in the **versions** folder and

Fig. 6: User Update File

4) File Delete: Figure 7 describes the result when clients delete a file. The file is first deleted from the synchronization folder, which is then deleted on the server. However, the **versions** and **metadata** is still available in the server storage for further actions, e.g., clients need to restore the file.

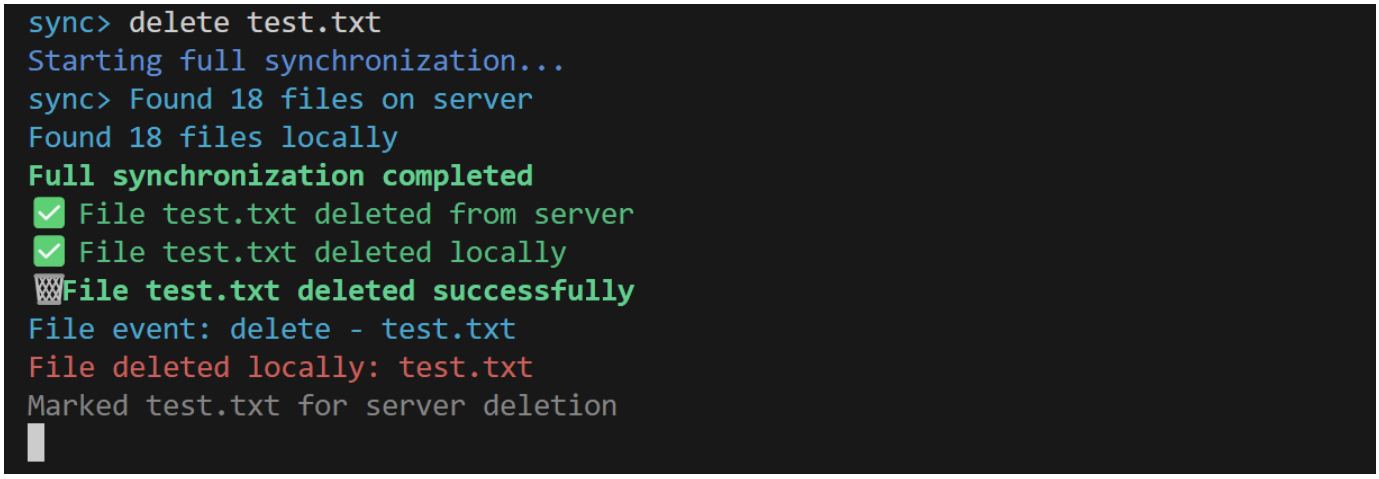


Fig. 7: User Update File

C. Fault Tolerance

a) Crash Client: First, when a client gets crashed when using the synchronization, the server and other clients still function normally, and the crashed client can connect again by typing the username and IP address of the server to which he/she wants to connect.

b) Server Crash: Second, when the server gets crashed after clients have connected to the server, clients still function normally. However, they cannot synchronize with other clients, and any CRUD operations will occur locally. After server returns active, the system is active and clients can continue the process of local files synchronization.

Figure 8 describe the error of client when the server crashes.

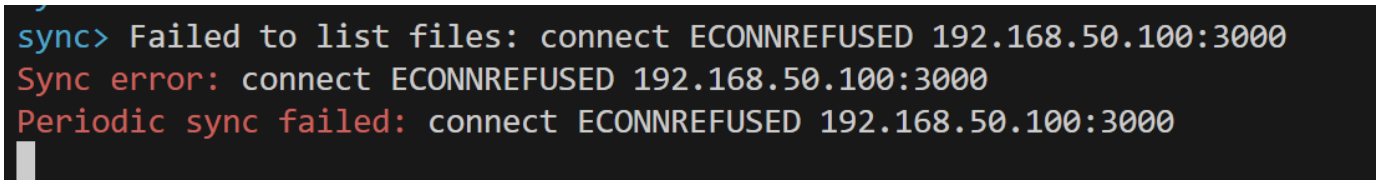


Fig. 8: Client Error due to Server Crash

c) Conflict Resolution: Conflicts during synchronization can arise when multiple clients make concurrent edits to the same file before the server can propagate updates to all parties. One common scenario involves two or more clients updating a file nearly simultaneously. When this occurs, only one version—typically the first to reach the server—is accepted as the latest version, while subsequent updates by other clients trigger a conflict detection mechanism.

Figure 9 illustrates such a conflict scenario, where a client updates test.txt shortly after another client has already uploaded a different version to the server. Upon detecting the mismatch between the local version and the server version, the client receives a conflict warning, showing detailed metadata such as file size, modification timestamp, and content previews for both local and server copies. This transparency helps users understand the source of the conflict.

To preserve all user changes and avoid data loss, the system automatically renames the conflicted file to include the origin of the conflict—e.g., test_conflicted_by_client-An.txt. This file is then uploaded to the server and downloaded by other clients, serving as an explicit indication that a versioning conflict has occurred.

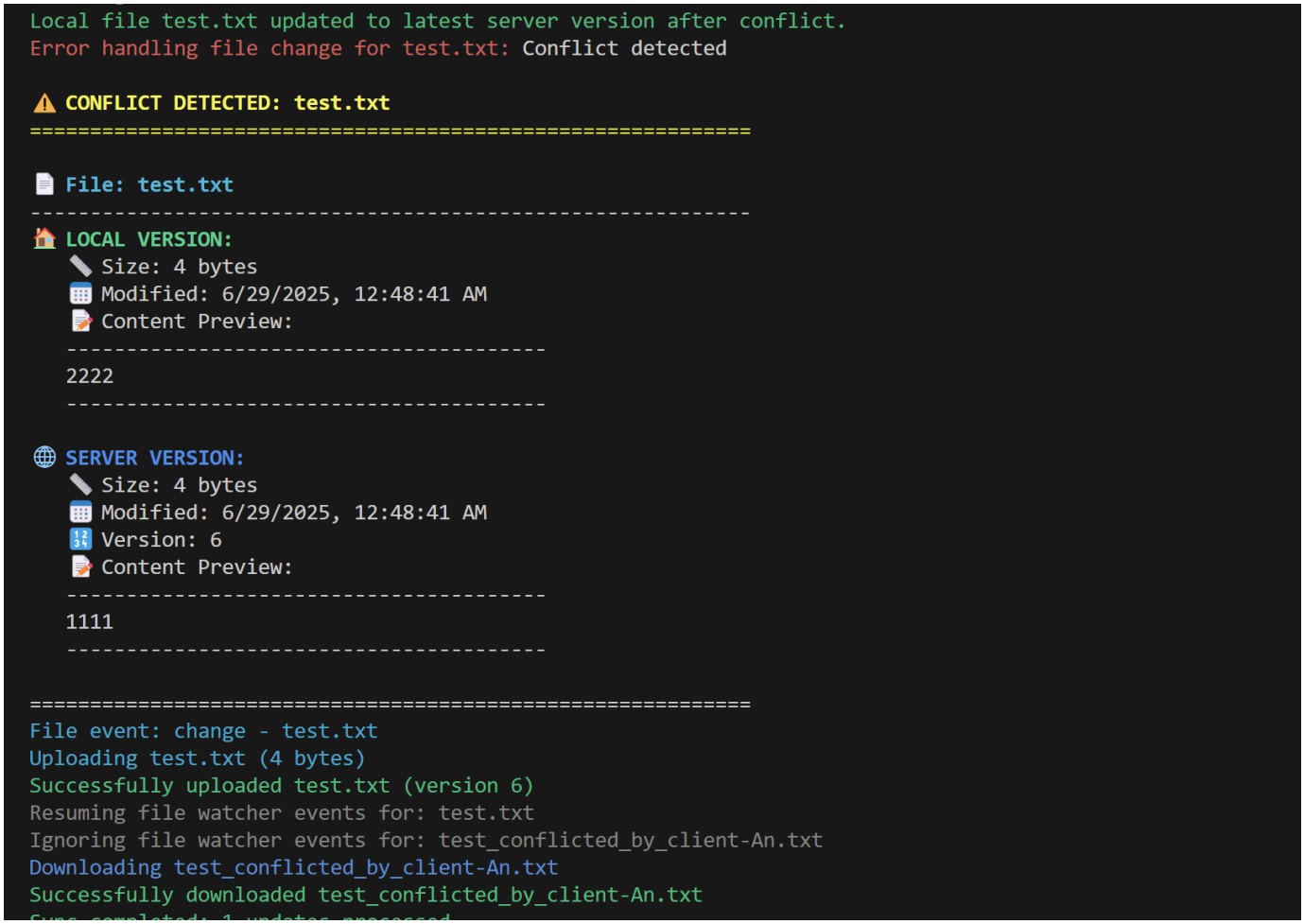


Fig. 9: User Update File

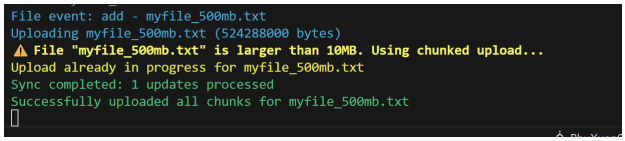
D. Scalability

In case of clients uploading large files, chunks are introduced to handle files of large size. To support scalability when uploading large files, the system splits files into fixed-size chunks. Each chunk is uploaded separately, allowing parallel transfer, resumable uploads, and efficient storage through deduplication. This chunking mechanism reduces bandwidth usage and improves fault tolerance in distributed environments.

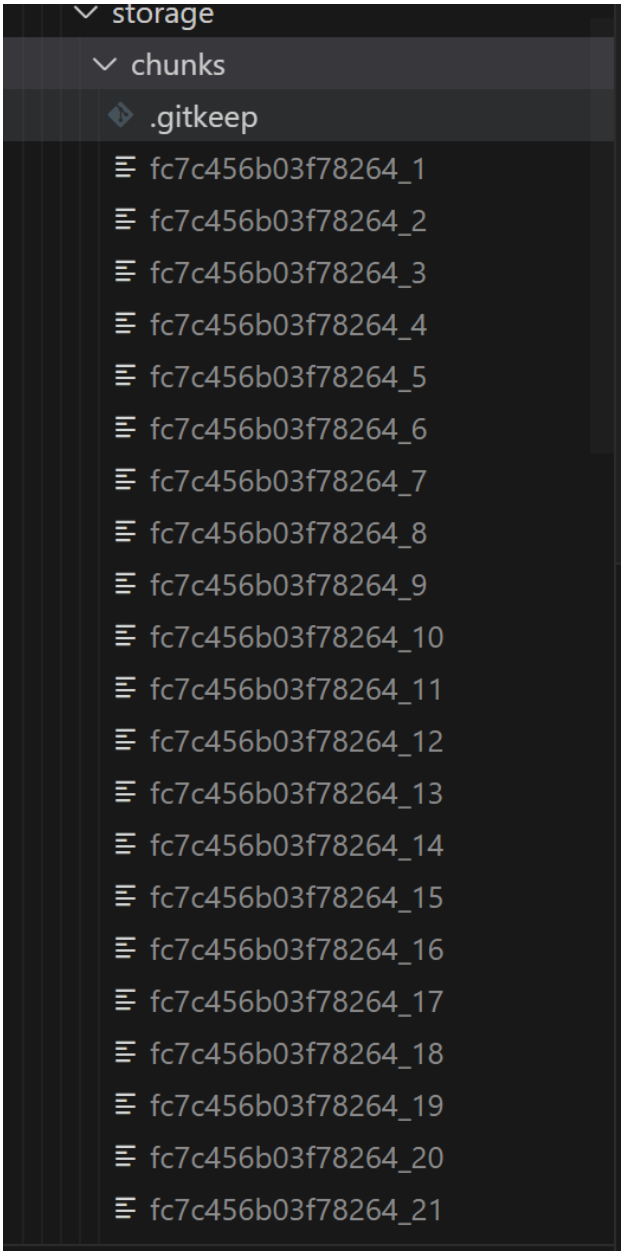
To support the efficient synchronization of large files, the system implements a chunking mechanism where files are divided into fixed-size blocks of **4 MB**. This chunk size is chosen as a balanced trade-off between memory usage, network throughput, and system responsiveness.

The key advantages of the 4 MB chunking approach include:

- **Efficient delta uploads:** When a file is modified, only the chunks that have changed (determined using hash comparison) are re-uploaded. This minimizes bandwidth usage, especially for large files with small edits.
- **Parallel transfers:** Multiple 4 MB chunks can be uploaded or downloaded concurrently, utilizing available bandwidth more effectively and reducing total sync time.
- **Resumable uploads:** In case of network interruptions, only the remaining chunks need to be retried. The system keeps track of uploaded chunks, making the process fault-tolerant.
- **Deduplication and storage optimization:** Identical chunks across different files or file versions are detected using content hashing and stored only once on the server, saving disk space.
- **Memory and buffer efficiency:** A 4 MB chunk is small enough to process efficiently in memory buffers, reducing pressure on client devices while still being large enough to minimize overhead.



(a) Clients Upload Large Files



(b) Chunk Folder

Fig. 10: Large File Upload Handling

VI. DISTRIBUTED SYSTEM CHALLENGES AND SOLUTIONS

The development of our file synchronization system required addressing fundamental distributed system challenges. This section analyzes how our implementation tackles several core distributed system challenges identified by Coulouris et al., demonstrating both the strengths and limitations of our approach.

A. Openness

Challenge: Designing the system to be extensible and allow reimplementation of components without affecting the overall architecture.

Our Solution: The system employs a modular architecture that separates concerns into distinct, loosely-coupled components. The client-side architecture divides functionality between API communication (api-client.js), synchronization logic (sync-manager.js), user interface (cli-interface.js), and file monitoring (file-watcher.js). Server-side components are similarly modularized with separate modules for API handling, storage management, and background processing.

The command system demonstrates extensibility through a pluggable architecture:

Listing 2: Extensible command system

```
this.commands = {
   'status': this.showStatus.bind(this),
   'sync': this.startSync.bind(this),
   'resolve': this.resolveConflictById.bind(this),
   // Additional commands can be easily added
};
```

This design allows new features to be added by implementing new command handlers without modifying existing functionality. Similarly, the queue-based background processing system enables the addition of new job types and processors

B. Security

Challenge: Ensuring confidentiality, integrity, and availability of data in a distributed environment.

Our Solution: The system implements several security measures, though this remains an area for improvement. Data integrity is maintained through MD5 checksums calculated for each file:

Listing 3: Integrity verification through checksums

```
calculateChecksum(buffer) {
   return crypto.createHash('md5').update(buffer).digest('hex');
}
```

Each client is assigned a unique identifier for tracking and isolation:

Listing 4: Client identification

```
this.clientId = options.clientId ||
   `client-${crypto.randomBytes(4).toString('hex')}`;
```

Below is an example of a file storage in server.

Listing 5: File JSON

```
{
  "fileId": "0d047828d24059ce",
  "fileName": "hello.txt",
  "version": 2,
  "size": 7,
  "checksum": "e1faffe9c3c801f2f8c3fbe7cb032cb2",
  "clientId": "client-Dat",
  "lastModified": "2025-06-27T05:37:01.014Z",
  "restoredFrom": "1",
  "createdAt": "2025-06-27T05:37:01.016Z",
  "updatedAt": "2025-06-27T05:37:01.016Z",
  "chunks": null
}
```

System availability is monitored through health check endpoints. However, the current implementation lacks encryption for data in transit and at rest, and does not include authentication mechanisms. These limitations represent areas for future enhancement to achieve enterprise-grade security.

C. Concurrency

Challenge: Managing shared resources accessed by multiple clients simultaneously while maintaining data consistency.

Our Solution: The system addresses concurrency through conflict detection and resolution mechanisms. When multiple clients modify the same file concurrently, the server detects conflicts by comparing client IDs and file checksums:

Listing 6: Conflict detection logic

```
if (entry.clientId !== clientId && entry.checksum !== checksum) {
   isConflict = true;
}

const conflictEntry = {
   id: require('crypto').randomBytes(8).toString('hex'),
   fileName,
   reason: 'Simultaneous modification detected',
   // Additional conflict metadata
};
```

Client-side conflict handling provides users with tools to manually resolve conflicts through an interactive interface. The system maintains version history, allowing users to examine different versions and make informed decisions about conflict resolution. While this approach requires manual intervention for conflicts, it ensures data integrity and gives users full control over their data. At the client level, the system prevents duplicate operations through pending operation tracking using maps to ensure the same file is not processed by multiple concurrent operations:

Listing 7: Preventing duplicate uploads and downloads

```
class SyncManager {
   //
   this.pendingUploads = new Map();
   this.pendingDownloads = new Map();
   //
}
// Prevent duplicate uploads
if (this.pendingUploads.has(fileName)) {
   console.log(`Upload already in progress for ${fileName}`.yellow);
   return;
}
this.pendingUploads.set(fileName, filePath);
// ... upload logic ...
this.pendingUploads.delete(fileName);
```

This dual approach ensures that before starting any upload or download operation, the system checks if the file is already being processed. If so, it skips the operation, preventing race conditions and redundant network usage. Client-side conflict handling provides users with tools to manually resolve conflicts through an interactive interface, maintaining version history and ensuring data integrity while giving users full control over conflict resolution decisions.

D. Transparency

Challenge: Hiding the complexity of distribution from users and making the system appear as a unified, local service.

Our Solution: The system achieves transparency through a unified command-line interface that abstracts the distributed nature of file operations. Users interact with files as if they were purely local, while synchronization, conflict detection, and version management occur transparently in the background.

The automatic file monitoring system operates invisibly, detecting changes and triggering synchronization without user intervention. The CLI provides a consistent interface regardless of whether operations involve local files or require server communication. Status information and progress indicators keep users informed without exposing the underlying distributed system complexity.

E. Summary of Solutions

Our file synchronization system successfully addresses distributed system challenges through careful architectural design and implementation choices. The modular, Node.js-based architecture provides excellent cross-platform compatibility and extensibility. Background processing and chunked transfers support scalability, while comprehensive error handling and conflict resolution ensure system reliability.

The primary areas for improvement lie in security (encryption and authentication) and advanced QoS features. Despite these limitations, the system demonstrates how fundamental distributed system principles can be applied to create a practical, usable file synchronization solution that operates effectively in real-world environments.

The implementation serves as a solid foundation that could be extended with additional features such as end-to-end encryption, role-based access control, advanced conflict resolution algorithms, and more sophisticated performance monitoring and optimization capabilities.

VII. Member Contribution Evaluation

Member Full name (Student ID)	**Tasks Complete**	**Contribution Rate**
Dat Nguyen Tran Quoc (10422017)	Plan and implement the main functionalities of system; Handle client& server crash error handling	100%
Giap Nguyen Minh (10422024)	Handle multiple clients connection to server; Test and fix encountered errors	100%
Phu Nguyen Xuan (10422065)	Implement additional functions; Implement file chunks for large file sending (Scalability)	100%
Quy Tran Phuc (10422071)	Handle conflict resolution and version control (Fault Tolerance); Refactor the metadata storage to different files (Scalability)	100%
Quang Nhat Hoang (10422060)	Work on additional commands and document report	100%

TABLE I: Summary of backend test results