Peer-to-Peer File Backup and Recovery System (P2PBRS)

COEN 366

FALL 2025

December 6th, 2025

Authors:

Mik Driver (40244456)

Julia María Arpón Pita (40349033)

Pablo Rodriguez Canizares (40349103)

Joshua Lafleur (40189389)

GitHub:

https://github.com/COEN366-FALL2025-Group25/P2PBRS-Group25

Concordia University

TABLE OF CONTENTS

1 INTRODUCTION

The Peer-to-Peer Backup and Recovery System (P2PBR) project for COEN 366 has helped us understand how networks work regarding data transmission, how reliable TCP and UDP can be in communications and how errors are faced.

The main objective of this project was to build a backup system where various peer nodes store each other's chunk files. There is a central server which coordinates communication between nodes, but it never receives the packets, its role is to monitor peers and help them connect with each other.

In the application, every peer node can have different roles:

- File owner (OWNER): a node that will upload its own files to be stored by other peer nodes.
- Storage (STORAGE): it takes the files that were uploaded and stores them by chunks if it has enough space.
- Both (BOTH): can upload files and store chunks.

When a file is backed up, it is splitted into chunks and distributed across different peers if they have enough space, making the system very reliable.

To guarantee the working of the project, we needed to use UDP and TCP communication.

- UDP: lightweight control messages such as registration, heartbeats and backup requests.
- TCP: for sending the actual chunks of data.

To ensure proper communication, apart from the reliability given by TCP, we also used checksum to verify integrity from end to end (CRC32).

Overall, the making of this project has given us an essential basis in real life communication networks and systems.

2 SYSTEM DESIGN

2.1 Architecture Overview

The system consists of a central server and peer nodes. The server has many tasks including maintaining the peer registry, generating backup/restore plans, monitoring heeartbeats, and enacting recovery of files. It does not handles the file data directly since it is cetralized. This ensures privacy and protection of the data. Alternatively, the peer nodes exchange the data using TCP connections while using UDP to communicate ith the server. The main phases are peer registration, backup plan creation and chunk transfer, restore plan retrieval and chunk download, heartbeat monitoring, and automatic failure detection and replication.

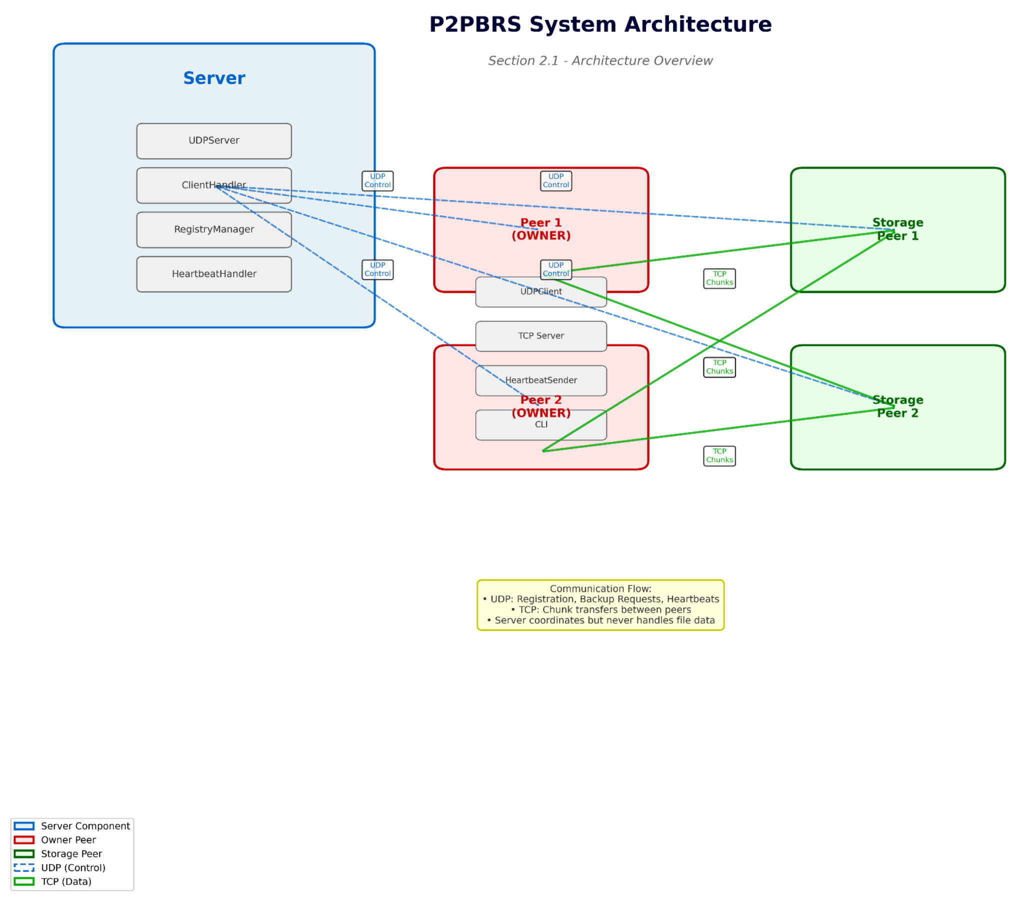

Figure 1: Architecture Overview

2.2 Server Design

The centralized server has many responsibilities. It must maintain peer registry (roles, ports, capacity, status), create backup and recovery plans, track file chunks and their locations, receive and process heartbeats, detect failures and initiate replication and also broadcast updated peer information. The data structures that are most important are the RegistryManager (a thread-safe peer registry), BackupManager.Plan (chunk placement map), fileChunkOwners (splits file into chunks then sends to peers), and peerStoredChunks (has the peer receive and store the chunks sent). Threading is used to ensure system performanc, allowing multiple actions to be performed at once. It is used in the main UDP listener, per-request handler threads and also in the continuous heartbeat checking.

2.3 Peer Design

The system allows for many peers ot exist in th enetwork at once. Each of the existing peers has a role, an IP adress, UDP/TCP ports, its storage capacity, and a state & heartbeat timestamp. Components that are working concurrently include the UDP listener thread, the TCP server for chunk reception, the heartbeat sender and the CLI thread. Peers use request IDs and CompletableFuture to correlate UDP responses with requests.

2.4 Protocol Messages

Messages are text-based and space-delimited. They are sent using UDP whenever speed is required. For example for registration, backup/restore requests, heartbeats, and replication control. Alternatively, they are passed over a secure TCP connection when data integrity is required. For example chunk upload/download, replication, and integrity responses. The systems inplemnts a checksum to ensure data sent over the connections is correct. Specifically, it uses CRC32 for chunk-level and whole-file verification.

2.5 Heartbeats & Failure Handling

The heartbeat subsystem is a key mechanism of the network. Thanks to it, the server can know which node is dead, de-register that node and start a recovery plan for all the stored chunks that node had.

Once the peer has successfully registered in the system, it launches a thread dedicated exclusively to send heartbeat packets: the HeartbeatSender. This thread sends the messages of format: HEARTBEAT | <RQ#> | <PeerName> | <ChunksStored> | <Timestamp> every 5 seconds. For the server part, there is another thread called HeartbeatHandler which is in charge of checking all last timestamps from all nodes and comparing it to the actual time. If it exceeds the limit maximum of 10 seconds, it deregisters the node and starts, if possible, a recovery plan. This is repeated every 2 seconds.

When a peer is considered “down”, it starts a recovery for all the stored chunks it had. The server starts looking for replicates of the lost chunks throughout the network, and when it finds them, it tries to find a replacement peer to store them.

The reason why it is done in a different infinite thread in both the server and the peers is because we need to keep sending and receiving these specific packets even when a backup or a recovery is being done. It is very important that the nodes keep sending heartbeats for the correct function of the network.

2.6 Assumptions & Clarifications

When running the system it is assumed there are no spaces in filenames, that the request numbers generated per peer are unique, and that storage directories are writable.

3 IMPLEMENTATION

3.1 Tools and Technologies

The system uses Java 11, Maven, SnakeYAML (for registry persistence), JLine (for the custom peer CLI), and Java net and zip libraries (for networking + CRC). Github was used for code tracking.

3.2 Key Modules

The modules of the Server are the ServerMain, UDPServer, HeartbeatHandler, RegistryManage. Alternatively the peer modules include PeerNode, UDPClient, TCPServer, HeartbeatSender.

3.3 Persistence

The server stores its registry in a YAML file (registry.yaml). The file is loaded at startup and is written when there are changes to the information. This is how the backup plans are stored in memory.

3.4 Error and Exception Handling

The system includes extensive error handling. A few examples of relevant handling include duplicate or invalid registration of peers, missing chunks during the restore process, checksum mismatch (in many cases), TCP connection errors between peers, and timeout and retry for failed UDP requests.

3.5 Class Diagram

The following UML diagram illustrates the main classes in the P2PBRS system and their relationships.

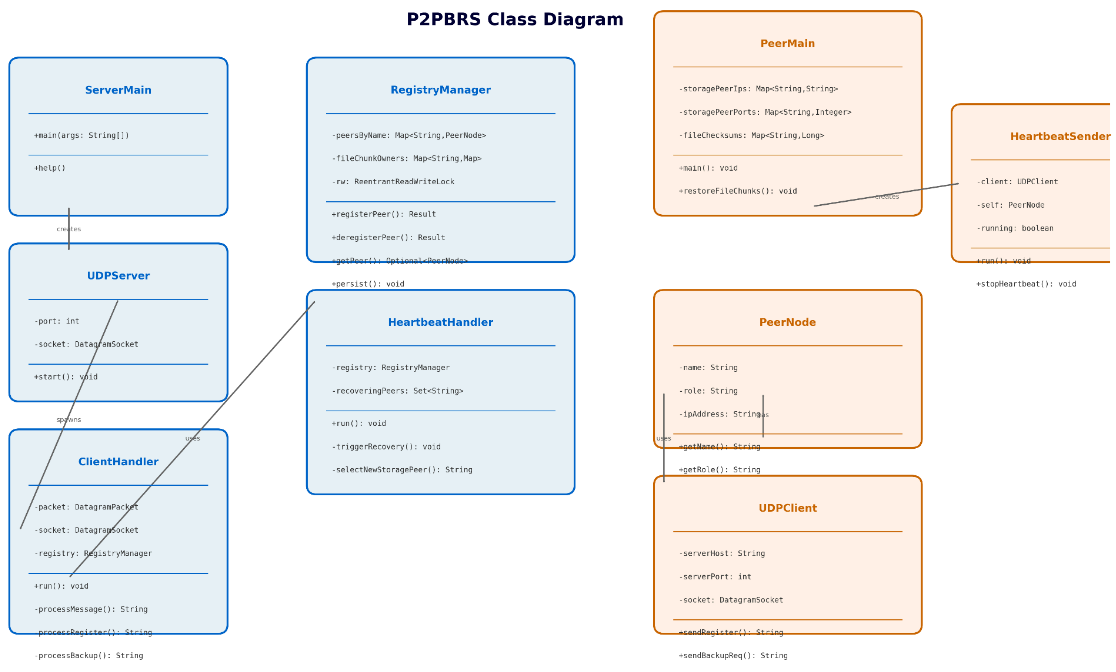

Figure 2: Class Diagram

4 TESTING

4.1 Test Environment

The system was tested using multiple console windows on a single physical machine. Each node (server, 2 clients, 2 storage nodes) ran in a separate terminal/console window to simulate separate machines.

An example of node donfigurations is…

Server: Port 5000 (UDP)

Client 1: OWNER role, UDP 5001, TCP 6001

Client 2: OWNER role, UDP 5002, TCP 6002

Storage Node 1: STORAGE role, UDP 5003, TCP 6003, 100MB capacity

Storage Node 2: STORAGE role, UDP 5004, TCP 6004, 100MB capacity

The peer application has a custom interactive command-line interface using JLine library. This proveds command history (arrow keys), tab completion and many command. Commands include bbackup, restore, deregister, help, exit .

All UDP/TCP messages are printed to the console with timestamps. For the server, thread names are included in server logs for debugging. Any rrror messages clearly marked with ERROR: prefix.

4.2 Test Scenarios

The following scenarios are successfully implemneneted and can be demonstrated with the above format: registration and de-registration, backup with multiple chunks, recovery under normal and failure conditions, heartbeat timeout and replication, and Server crash and restart (persistence test). See Appendix I for in depth procedure and results.

5 CONCLUSION

In this project, a Peer-to-Peer Backup and Recovery System (P2PBRS) was designed, implemented and analyzed. By distributing data among multiple peers, the system increases fault tolerance, reduces reliance on single points of failure, and enables scalable resource sharing across the network. The architectural overview and class diagrams highlight how each component contributes to reliability, availability, and secure data exchange.

Through this project, imortant concepts in networking and protocols were studied such as decentralized control, replication strategies, node coordination, and metadata management. The design demonstrates how peers can independently store, retrieve, and verify data while maintaining global consistency through lightweight communication protocols. Practical engineering trade-offs were demonstrated and explored when modeling the system. This includes how replication factors affect storage overhead, how node churn influences data durability, and how trust and integrity must be maintained in a distributed environment.

Overall, the P2PBRS architecture implements a functional peer-to-peer backup service. Future improvements to the system could include integrating encryption for end-to-end security, optimizing replication algorithms using dynamic policies, and perhaps adding incentives for resource sharing. The design process followed not only clarifies the core structure of a robust peer to peer system but also provides a roadmap for extending it into a production-grade distributed backup platform.

6 TEAM CONTRIBUTIONS

Name	Contributions
Mik	● Built TCP client and TCP server subsystems for chunk transfer, acknowledgments, retries, and restoration. ● Implemented unsolicited message handling (STORE_REQ, REPLICATE_REQ, etc.). ● Wrote peer-side filesystem handling: chunk writing, reading, and full-file reconstruction. ● Implemented major portions of ClientHandler, including message routing, backup/restore plan creation, peer selection logic, and error handling. ● Assisted in designing the overall system architecture ● Participated in team discussions on requirements, assumptions, and design trade-offs ● Updated Documentation
Julia	● Built HeartbeatHandler for liveness detection, including timeouts, replication triggers, peer removal notifications, and automated recovery logic. ● Implemented ServerMain, including startup, shutdown, and thread orchestration. ● Designed and refined protocol message formats for server-to-peer communication. ● Assisted in designing the overall system architecture ● Participated in team discussions on requirements, assumptions, and design trade-offs ● Updated Documentation
Pablo	● Developed UDPServer for packet listening and dispatching. ● Contributed to error resilience, failure scenario handling, and peer cleanup mechanisms. ● Implemented the HeartbeatSender peer thread and connection monitoring logic. ● Designed and executed unit tests, integration tests, and multi-peer simulations (backup, restore, chunk loss, recovery). ● Assisted in designing the overall system architecture ● Participated in team discussions on requirements, assumptions, and design trade-offs ● Updated Documentation
Josh	● Implemented PeerMain, including CLI, command parsing, storage setup, file chunking, and checksum generation. ● Added CRC32 checksum verification for backup, restore, and replication workflows. ● Developed RegistryManager with full read/write lock protection, peer registration, deregistration, and lookup logic. ● Implemented YAML-based persistence, including serialization, deserialization, and state rebuilding on server startup. ● Developed UDPClient, including request-response matching, asynchronous listener threads, futures, timeouts, and message building. ● Assisted in designing the overall system architecture ● Participated in team discussions on requirements, assumptions, and design trade-offs ● Updated Documentation

APPENDIX I

Test 1: Registration and De-registration

Procedure:

1. Start server
2. Register Client1 as OWNER
3. Register StorageNode1 as STORAGE
4. Register StorageNode2 as STORAGE
5. Register Client2 as OWNER
6. Attempt duplicate registration (same name, should fail)
7. Deregister C1 8. Verify C1 removed from registry

Results: All registrations successful. Duplicate registration rejected with "ERROR: Name already registered". Deregistration successful, peer removed. Server persisted state to registry.yaml

Test 2: Backup with Multiple Chunks

Procedure:

1. Register all 5 nodes
2. Client1 backs up testfile.txt (3 chunks, 4096 bytes/chunk)
3. Verify server creates backup plan
4. Verify StorageNode1 and StorageNode2 receive STORE_REQ messages
5. Verify chunks stored in storage directories 6. Verify BACKUP_DONE confirmation

Results: Server selected both storage nodes. Chunks distributed: chunk 0 → StorageNode1, chunk 1 → StorageNode2, chunk 2 → StorageNode1. All chunks stored successfully. Checksums verified at storage peers. BACKUP_DONE received by server.

Test 3: Recovery under Normal Conditions

Procedure:

1. Client2 requests restore of testfile.txt
2. Server provides RESTORE_PLAN with chunk locations
3. Client2 retrieves chunks via TCP
4. Client2 verifies file checksum
5. Sends RESTORE_OK

Results: RESTORE_PLAN correctly identified chunk locations. All chunks retrieved successfully. File checksum matched original. File restored to restored_testfile.txt

Test 4: Recovery with Missing Chunk

Procedure:

1. Manually delete chunk file from StorageNode1 storage
2. Client2 attempts restore
3. StorageNode1 responds with CHUNK_DATA ... ERROR
4. Client2 sends RESTORE_FAIL

Results: Missing chunk detected, RESTORE_FAIL sent with reason "Chunk_0_Not_Found". Partial file not saved.

Test 5: Heartbeat Timeout and Replication

Procedure:

1. All nodes registered and running
2. Kill StorageNode1 process (simulate failure)
3. Wait 10+ seconds for heartbeat timeout
4. Verify server detects failure
5. Verify server triggers replication
6. Verify StorageNode2 replicates chunk to another peer (if available)
7. Verify chunk tracking updated

Results: Server detected StorageNode1 failure after ~10 seconds. Server identified chunks stored by StorageNode1. Server found StorageNode2 as source for replication. Replication completed via TCP. Chunk tracking updated in registry. Backup plan updated with new chunk location.

Test 6: Server Crash and Restart (Persistence Test)

Procedure:

1. Register multiple peers
2. Perform backups (create chunk tracking)
3. Stop server 4. Restart server
5. Verify registry reloaded from YAML
6. Verify chunk tracking restored
7. Attempt restore (should work)

Results: Server loaded registry.yaml on startup. All peers restored in registry. Chunk tracking restored. Restore operation succeeded using restored metadata. Backup plans not fully restored (reconstructed from chunk tracking).

Test 7: Concurrent Operations

Procedure:

1. Client1 starts backup of large file
2. Client2 simultaneously starts restore
3. StorageNode1 receives chunks from Client1 while serving chunks to Client2
4. Heartbeats continue during operations

Results: Multiple TCP connections handled concurrently. Heartbeats not blocked by chunk transfers. All operations completed successfully.

Test 8: Checksum Validation

Procedure:

1. Client1 backs up file
2. Manually corrupt chunk file in storage
3. Client2 attempts restore
4. Verify checksum mismatch detected

Results: Chunk checksum mismatch detected during restore. Error logged. RESTORE_FAIL sent with checksum error reason.

APPENDIX II

This form sets out the requirements for originality for work submitted by students in the Faculty of Engineering and Computer Science. Submissions such as assignments, lab reports, project reports, computer programs and take-home exams must conform to the requirements stated on this form and to the Academic Code of Conduct. The course outline may stipulate additional requirements for the course.

1. Your submissions must be your own original work. Group submissions must be the original work of the students in the group.
2. Direct quotations must not exceed 5% of the content of a report, must be enclosed in quotation marks, and must be attributed to the source by a numerical reference citation[1]. Note that engineering reports rarely contain direct quotations.
3. Material paraphrased or taken from a source must be attributed to the source by a numerical reference citation.
4. Text that is inserted from a web site must be enclosed in quotation marks and attributed to the web site by numerical reference citation.
5. Drawings, diagrams, photos, maps or other visual material taken from a source must be attributed to that source by a numerical reference citation.
6. No part of any assignment, lab report or project report submitted for this course can be submitted for any other course.
7. In preparing your submissions, the work of other past or present students cannot be consulted, used, copied, paraphrased or relied upon in any manner whatsoever.
8. Your submissions must consist entirely of your own or your group's ideas, observations, calculations, information and conclusions, except for statements attributed to sources by numerical citation. 9. Your submissions cannot be edited or revised by any other student.
10. For lab reports, the data must be obtained from your own or your lab group's experimental work. 11. For software, the code must be composed by you or by the group submitting the work, except for code that is attributed to its sources by numerical reference.

You must write one of the following statements on each piece of work that you submit: For individual work: **"I certify that this submission is my original work and meets the Faculty's Expectations of Originality"**, with your signature, I.D. #, and the date.
For group work: **"We certify that this submission is the original work of members of the group and meets the Faculty's Expectations of Originality"**, with the signatures and I.D. #s of all the team members and the date.

A signed copy of this form must be submitted to the instructor at the beginning of the semester in each course.

I certify that I have read the requirements set out on this form, and that I am aware of these requirements. I certify that all the work I will submit for this course will comply with these requirements and with additional requirements stated in the course outline.

Course Number: COEN 366 Instructor: Ahmed Bali, Ph.D. Date: Dec 6th, 2025

Name: Mik Driver I.D. # 40244456 Signature: [illegible]

Name: Julia María Arpón Pita I.D. # 40349033 Signature: Julia AP

Name: Pablo Rodriguez Canizares I.D. # 40349133 Signature: [illegible]

Name: Joshua Lafleur I.D. # 40189389 Signature: [illegible]