Google File System with Dynamic Replication

Project Report
Team 15

Chirag Dhamija *2022101039*   *Prithvi Karthik* *2022101020*

November 27, 2024

1 Problem

The Google File System (GFS) is a scalable, distributed file system designed to handle large data-intensive applications by providing reliable storage and high-throughput access to data. While GFS maintains multiple replicas of data for fault tolerance, its static replication mechanism falls short in adapting to dynamic workload conditions. In a high-traffic scenario, statically configured replicas may lead to hotspots and degraded performance, while underutilized replicas consume unnecessary storage resources.

The goal of this project was to design and implement **dynamic replication** within a GFS-like system, allowing the system to adjust the number of replicas based on real-time load, access patterns, and server status. This enhancement aims to improve system efficiency, availability, and performance in varying workload environments.

2 Assumptions

- Update operations can only be performed if both the primary and secondary servers are operational for the file.
- A file cannot be created with the same name if it already exists in the GFS.

- Master replication is not supported in the current implementation.

3 How to Run the Code

1. Execute the bash script `precompile.sh`.
2. Start the Master server by running `python3 master.py`.
3. Launch at least three Chunk Servers by running `python3 chunkserver.py <port_number>`.
4. Use the client to interact with the system by running `python3 client.py <file_name> <operation>`.

4 GFS Architecture

- **Master:** A Master server tracks metadata and coordinates system operations.
- **Chunk Servers:** Multiple servers store file chunks and handle data requests.
- **Clients:** Applications interacting with the system for reading, writing, and managing files.
- **File:**
  - Files are divided into fixed-size chunks of 12 bytes (this size can be adjusted by changing the `self.chunk_size` variable in each file).
  - Each chunk is identified by a unique 64-bit global ID (handle).
  - Chunks are distributed across Chunk Servers.
  - Each chunk has a default replication factor of three (static replication).
- **Metadata:** Maintained by the Master and includes:
  - File-to-chunk mappings.
  - Locations of the primary and secondary Chunk Servers for each chunk.

5 GFS Basic Operations

- The client fetches metadata from the Master for operations.
- Data flows directly between clients and Chunk Servers.
- Minimizing the Master's involvement in data transfer reduces the single-master bottleneck.

6 GFS Master

- The Master operates as a single server and handles all metadata-related tasks.
- Metadata includes file-to-chunk mappings and Chunk Server locations for each file chunk.

7 GFS Operations - Read, Write, and RecordAppend

7.1 Read Operation (`read`)

1. **Client → Master**: Client sends a request for the file name/path to retrieve metadata.
2. **Master → Client**: Master responds with chunk handles, chunkserver locations (primary and replicas).
3. **Client → Chunkservers**: For each chunk, the client requests data from the primary chunkserver. If the primary doesn't respond, the client contacts the secondary chunkserver.
4. **Chunkservers → Client**: Chunkservers send the requested chunk data.
5. **Client Reassembles Data**: The client combines the chunks in the correct order to reconstruct the entire file.

7.2 Write Operation (`write`)

1. **Client → Master**: Client sends the file name/path and write offset.
2. **Master → Client**: Master responds with the location of the last chunk, its replicas, and the location of new chunks if they are created.
3. **Client → Primary Chunkserver**: Client sends the file chunk number, write offset, and data to be written.
4. **Primary Chunkserver → Replica Chunkservers**: Primary chunkserver instructs the secondary chunkservers to write the data in the same order.
5. **Replica Chunkservers → Primary Chunkserver**: Secondary chunkservers acknowledge the successful write.
6. **Primary Chunkserver → Client**: Primary chunkserver sends the final acknowledgment of a successful write.

Note: The primary chunkserver enforces a consistent update order across all replicas and waits for all replicas to finish writing before responding.

7.3 RecordAppend Operation (`append`)

1. **Client → Master**: Client sends a request to append data to a specific file with the file name and data.
2. **Master → Client**: Master returns metadata of the last chunk, including primary and secondary chunkserver locations.
3. **Client → Primary Chunkserver**: Client contacts the primary chunkserver to append data and sends the secondary chunkserver locations.
4. **Case 1: Last Chunk Size Exceeds Chunk Size**
    - Primary chunkserver checks if there is enough space in the last chunk. If not, it pads the chunk with '
    - Primary chunkserver informs the client that there is insufficient space, and the client contacts the master.

- Master creates new chunks, assigns primary and secondary chunkservers, and updates metadata.
- New primary chunkserver creates new chunks and replicates them to secondary chunkservers.
- New secondary chunkservers acknowledge the successful append.
- New primary chunkserver sends an acknowledgment to the client after receiving acknowledgments from all chunkservers.

5. **Case 2: Last Chunk Size Fits Within Chunk Size**

   - Primary chunkserver appends the data to the last chunk.
   - Primary chunkserver instructs secondary chunkservers to append the data to replica chunks.
   - Secondary chunkservers send an acknowledgment after appending the data.
   - Primary chunkserver sends the final acknowledgment to the client after receiving acknowledgment from secondary chunkservers.

7.4 Heartbeat Mechanism

The heartbeat mechanism in our implementation is crucial for monitoring the status of Chunk Servers and ensuring efficient management of replication and fault tolerance.

7.4.1 Chunk Server Implementation

Each Chunk Server maintains a dedicated thread responsible for periodically sending heartbeat messages to the Master. These heartbeat messages include:

- Chunk Server ID
- Timestamp
- The number of requests handled since the last heartbeat.

The interval for sending heartbeats is configurable and ensures the Master is regularly updated on the state of all Chunk Servers.

7.4.2 Master Implementation

The Master handles heartbeats using a multithreaded approach. The threads are organized as follows:

1. **Heartbeat Reception Thread:** Receives heartbeat messages from all Chunk Servers and adds them to a processing queue.
2. **Heartbeat Processing Thread:** Iterates through the queue of received heartbeats. For each message:
   - It checks if the number of requests since the last heartbeat exceeds a predefined threshold.
   - If the threshold is exceeded, the Master initiates dynamic replication.
   - It updates the timestamp of the last received heartbeat for the respective Chunk Server.
3. **Failure Detection Thread:** Monitors the timestamps of the last received heartbeats from all Chunk Servers. If the time elapsed exceeds three times the heartbeat interval, the Chunk Server is marked as failed. The Master then initiates dynamic replication to redistribute the data from the failed Chunk Server to ensure fault tolerance.

This multithreaded approach enables the Master to efficiently handle heartbeat messages, maintain an up-to-date view of the system's state, and dynamically manage replication based on workload and fault conditions.

8 Additional Operations

Other side operations such as overwrite, upload, delete, and rename a file have also been implemented.

9 Dynamic Replication

Dynamic replication is a critical feature that helps ensure optimal load balancing and fault tolerance in the system. The Master dynamically adjusts the replication of chunks based on three specific conditions:

9.1 Conditions for Dynamic Replication

1. **High Request Rate to a Single Chunk:** When the number of requests to a specific chunk exceeds a predefined threshold, the Master increases the replication factor for that particular chunk. This ensures that multiple Chunk Servers can serve requests for the chunk, reducing latency and balancing the load across the system.

2. **High Request Rate to a Single Server:** If the total number of requests handled by a single Chunk Server surpasses a predefined threshold, the Master increases the replication factor for all chunks stored on that server. This allows the load to be distributed across additional servers, preventing bottlenecks and ensuring high performance.

3. **Chunk Server Failure:** In the event of a Chunk Server failure, detected through the heartbeat mechanism, the Master initiates dynamic replication for all chunks stored on the failed server. This ensures data availability and fault tolerance by redistributing the chunks to other healthy servers in the cluster.

9.2 Replication Process

In all three cases, the dynamic replication process follows these steps:

- The Master identifies the chunks that require increased replication based on the condition triggered.

- The Master sends a request to a Chunk Server that already holds the chunk, providing a list of target servers that do not currently have the chunk.

- The Chunk Server retrieves the relevant chunks and distributes them to the specified target servers.

- The Master updates the replication metadata to reflect the new replicas and their locations.

This dynamic replication ensures that the system remains resilient to varying workload conditions and server failures while maintaining high performance and data integrity.

10 Impact

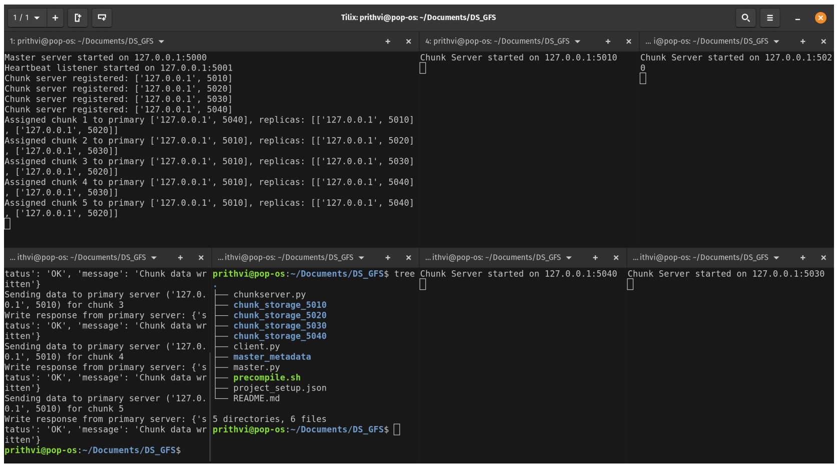
The dynamic replication mechanism significantly enhances the Google File System (GFS) by addressing critical system performance metrics. The following qualitative improvements demonstrate the major additions of the features:

- **Enhanced Load Balancing:** The dynamic replication mechanism distributes workload evenly across chunkservers by replicating heavily accessed chunks. This reduces bottlenecks and improves overall system throughput, ensuring scalability even under high request rates.
- **Increased Fault Tolerance:** The system handles chunkserver failures seamlessly by redistributing chunks from failed servers. This ensures uninterrupted access to data and maintains reliability even in the event of hardware failures.
- **Scalability and Adaptability:** The system dynamically adjusts the replication factor based on workload and server conditions, making it highly scalable and adaptable to varying demands. It provides a robust mechanism to handle spikes in requests to specific chunks or servers.
- **Optimized Resource Utilization:** Targeted replication minimizes unnecessary duplication, efficiently using storage and network resources while ensuring high availability for critical chunks.
- **Improved User Experience:** By minimizing latency and maintaining consistent performance, the dynamic replication mechanism ensures faster access to data, enhancing the overall user experience.

While we lacked scale of data to perform quantitative benchmarking or generate performance graphs, some metrics that could be benchmarked include:

- **Request Latency:** Measuring response times for read/write operations before and after replication under varying workloads.
- **Throughput:** Comparing the number of requests handled per unit time in the presence of dynamic replication against a static replication baseline.

11 Images


Tilix: prithvi@pop-os: ~/Documents/DS_GFS
1: prithvi@pop-os: ~/Documents/DS_GFS
4: prithvi@pop-os: ~/Documents/DS_GFS
... i@pop-os: ~/Documents/DS_GFS
... ithvi@pop-os: ~/Documents/DS_GFS

```
Master server started on 127.0.0.1:5000
Heartbeat listener started on 127.0.0.1:5001
Chunk server registered: ['127.0.0.1', 5010]
Chunk server registered: ['127.0.0.1', 5020]
Chunk server registered: ['127.0.0.1', 5030]
Chunk server registered: ['127.0.0.1', 5040]
Assigned chunk 1 to primary ['127.0.0.1', 5040], replicas: [['127.0.0.1', 5010]
, ['127.0.0.1', 5020]]
Assigned chunk 2 to primary ['127.0.0.1', 5010], replicas: [['127.0.0.1', 5020]
, ['127.0.0.1', 5030]]
Assigned chunk 3 to primary ['127.0.0.1', 5010], replicas: [['127.0.0.1', 5030]
, ['127.0.0.1', 5020]]
Assigned chunk 4 to primary ['127.0.0.1', 5010], replicas: [['127.0.0.1', 5040]
, ['127.0.0.1', 5030]]
Assigned chunk 5 to primary ['127.0.0.1', 5010], replicas: [['127.0.0.1', 5040]
, ['127.0.0.1', 5020]]
```

```
Chunk Server started on 127.0.0.1:5010
```

```
Chunk Server started on 127.0.0.1:502
0
```

```
tatus': 'OK', 'message': 'Chunk data wr
itten'}
Sending data to primary server ('127.0.
0.1', 5010) for chunk 3
Write response from primary server: {'s
tatus': 'OK', 'message': 'Chunk data wr
itten'}
Sending data to primary server ('127.0.
0.1', 5010) for chunk 4
Write response from primary server: {'s
tatus': 'OK', 'message': 'Chunk data wr
itten'}
Sending data to primary server ('127.0.
0.1', 5010) for chunk 5
Write response from primary server: {'s
tatus': 'OK', 'message': 'Chunk data wr
itten'}
prithvi@pop-os:~/Documents/DS_GFS$
```

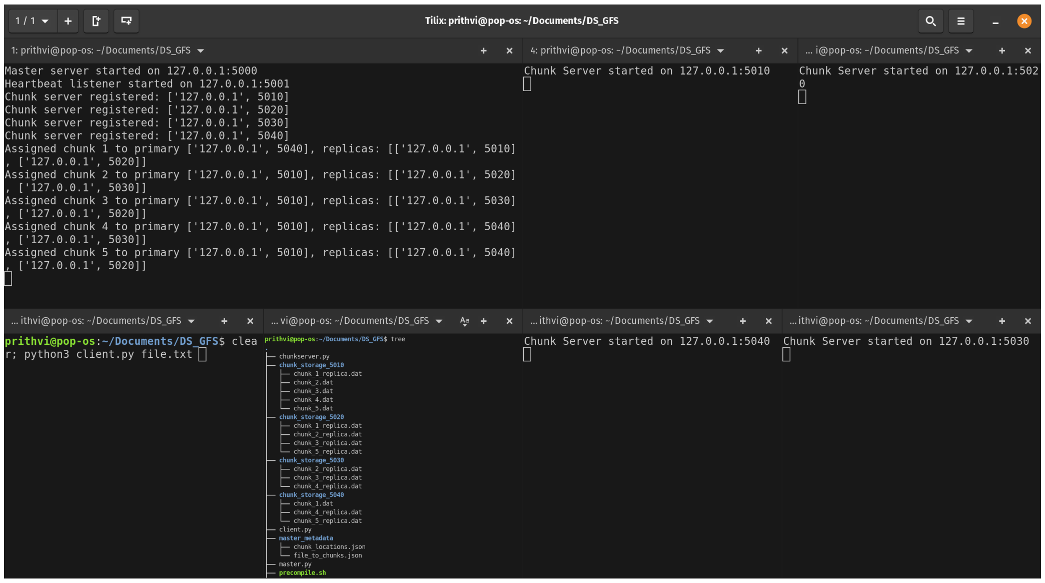
```
prithvi@pop-os:~/Documents/DS_GFS$ tree
.
├── chunkserver.py
├── chunk_storage_5010
├── chunk_storage_5020
├── chunk_storage_5030
├── chunk_storage_5040
├── client.py
├── master_metadata
├── master.py
├── precompile.sh
├── project_setup.json
└── README.md

5 directories, 6 files
prithvi@pop-os:~/Documents/DS_GFS$
```

```
Chunk Server started on 127.0.0.1:5040
```

```
Chunk Server started on 127.0.0.1:5030
```

Figure 1: Write to a File


Tilix: prithvi@pop-os: ~/Documents/DS_GFS
1: prithvi@pop-os: ~/Documents/DS_GFS
4: prithvi@pop-os: ~/Documents/DS_GFS
... i@pop-os: ~/Documents/DS_GFS
... ithvi@pop-os: ~/Documents/DS_GFS
... vi@pop-os: ~/Documents/DS_GFS

```
Master server started on 127.0.0.1:5000
Heartbeat listener started on 127.0.0.1:5001
Chunk server registered: ['127.0.0.1', 5010]
Chunk server registered: ['127.0.0.1', 5020]
Chunk server registered: ['127.0.0.1', 5030]
Chunk server registered: ['127.0.0.1', 5040]
Assigned chunk 1 to primary ['127.0.0.1', 5040], replicas: [['127.0.0.1', 5010]
, ['127.0.0.1', 5020]]
Assigned chunk 2 to primary ['127.0.0.1', 5010], replicas: [['127.0.0.1', 5020]
, ['127.0.0.1', 5030]]
Assigned chunk 3 to primary ['127.0.0.1', 5010], replicas: [['127.0.0.1', 5030]
, ['127.0.0.1', 5020]]
Assigned chunk 4 to primary ['127.0.0.1', 5010], replicas: [['127.0.0.1', 5040]
, ['127.0.0.1', 5030]]
Assigned chunk 5 to primary ['127.0.0.1', 5010], replicas: [['127.0.0.1', 5040]
, ['127.0.0.1', 5020]]
```

```
Chunk Server started on 127.0.0.1:5010
```

```
Chunk Server started on 127.0.0.1:502
0
```

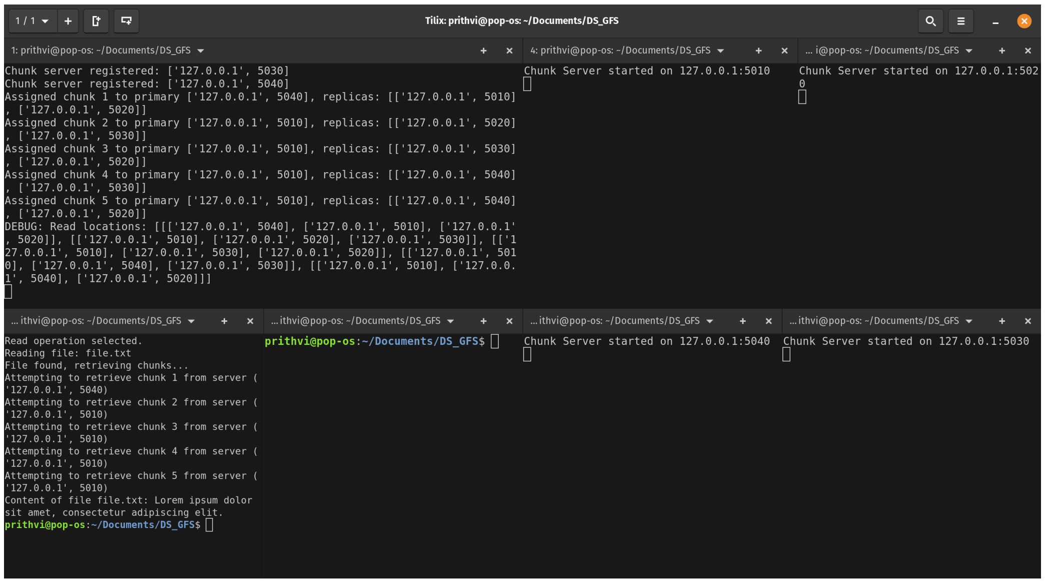
```
prithvi@pop-os:~/Documents/DS_GFS$ clea
r; python3 client.py file.txt
```

```
prithvi@pop-os:~/Documents/DS_GFS$ tree
.
├── chunkserver.py
├── chunk_storage_5010
│   ├── chunk_1_replica.dat
│   ├── chunk_2.dat
│   ├── chunk_3.dat
│   ├── chunk_4.dat
│   └── chunk_5.dat
├── chunk_storage_5020
│   ├── chunk_1_replica.dat
│   ├── chunk_2_replica.dat
│   ├── chunk_3_replica.dat
│   └── chunk_5_replica.dat
├── chunk_storage_5030
│   ├── chunk_2_replica.dat
│   ├── chunk_3_replica.dat
│   └── chunk_4_replica.dat
├── chunk_storage_5040
│   ├── chunk_1.dat
│   ├── chunk_4_replica.dat
│   └── chunk_5_replica.dat
├── client.py
├── master_metadata
│   ├── chunk_locations.json
│   └── file_to_chunks.json
├── master.py
├── precompile.sh
```

```
Chunk Server started on 127.0.0.1:5040
```

```
Chunk Server started on 127.0.0.1:5030
```

Figure 2: Write to a File : Output



```
Chunk server registered: ['127.0.0.1', 5030]
Chunk server registered: ['127.0.0.1', 5040]
Assigned chunk 1 to primary ['127.0.0.1', 5040], replicas: [['127.0.0.1', 5010]
, ['127.0.0.1', 5020]]
Assigned chunk 2 to primary ['127.0.0.1', 5010], replicas: [['127.0.0.1', 5020]
, ['127.0.0.1', 5030]]
Assigned chunk 3 to primary ['127.0.0.1', 5010], replicas: [['127.0.0.1', 5030]
, ['127.0.0.1', 5020]]
Assigned chunk 4 to primary ['127.0.0.1', 5010], replicas: [['127.0.0.1', 5040]
, ['127.0.0.1', 5030]]
Assigned chunk 5 to primary ['127.0.0.1', 5010], replicas: [['127.0.0.1', 5040]
, ['127.0.0.1', 5020]]
DEBUG: Read locations: [[['127.0.0.1', 5040], ['127.0.0.1', 5010], ['127.0.0.1'
, 5020]], [['127.0.0.1', 5010], ['127.0.0.1', 5020], ['127.0.0.1', 5030]], [['1
27.0.0.1', 5010], ['127.0.0.1', 5030], ['127.0.0.1', 5020]], [['127.0.0.1', 501
0], ['127.0.0.1', 5040], ['127.0.0.1', 5030]], [['127.0.0.1', 5010], ['127.0.0.
1', 5040], ['127.0.0.1', 5020]]]
```

```
Chunk Server started on 127.0.0.1:5010
```

```
Chunk Server started on 127.0.0.1:502
0
```

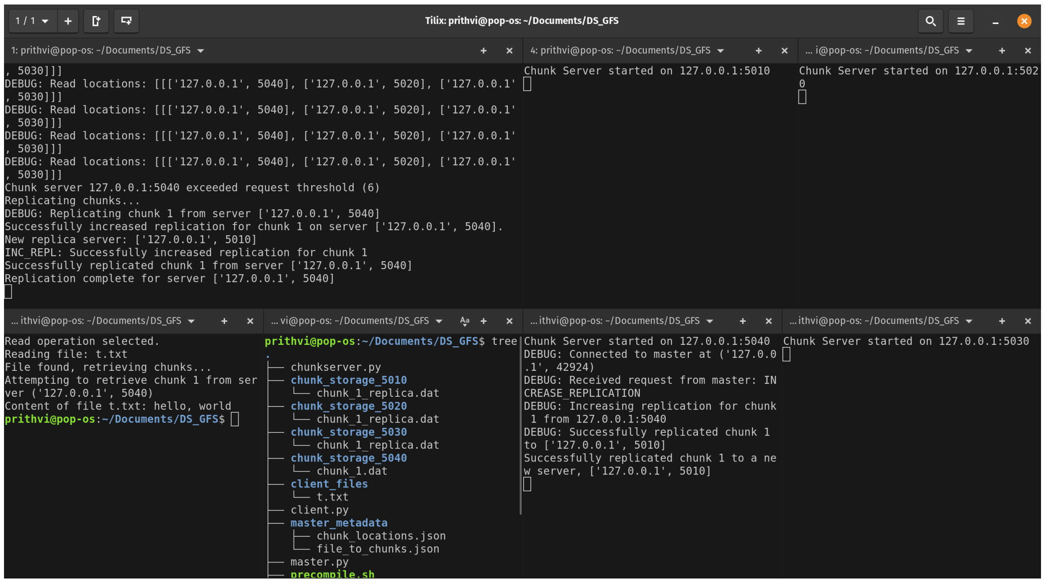
```
Read operation selected.
Reading file: file.txt
File found, retrieving chunks...
Attempting to retrieve chunk 1 from server (
'127.0.0.1', 5040)
Attempting to retrieve chunk 2 from server (
'127.0.0.1', 5010)
Attempting to retrieve chunk 3 from server (
'127.0.0.1', 5010)
Attempting to retrieve chunk 4 from server (
'127.0.0.1', 5010)
Attempting to retrieve chunk 5 from server (
'127.0.0.1', 5010)
Content of file file.txt: Lorem ipsum dolor
sit amet, consectetur adipiscing elit.
prithvi@pop-os:~/Documents/DS_GFS$
```

```
prithvi@pop-os:~/Documents/DS_GFS$
```

```
Chunk Server started on 127.0.0.1:5040
```

```
Chunk Server started on 127.0.0.1:5030
```

Figure 3: Read from a File



```
, 5030]]]
DEBUG: Read locations: [[['127.0.0.1', 5040], ['127.0.0.1', 5020], ['127.0.0.1'
, 5030]]]
DEBUG: Read locations: [[['127.0.0.1', 5040], ['127.0.0.1', 5020], ['127.0.0.1'
, 5030]]]
DEBUG: Read locations: [[['127.0.0.1', 5040], ['127.0.0.1', 5020], ['127.0.0.1'
, 5030]]]
DEBUG: Read locations: [[['127.0.0.1', 5040], ['127.0.0.1', 5020], ['127.0.0.1'
, 5030]]]
Chunk server 127.0.0.1:5040 exceeded request threshold (6)
Replicating chunks...
DEBUG: Replicating chunk 1 from server ['127.0.0.1', 5040]
Successfully increased replication for chunk 1 on server ['127.0.0.1', 5040].
New replica server: ['127.0.0.1', 5010]
INC_REPL: Successfully increased replication for chunk 1
Successfully replicated chunk 1 from server ['127.0.0.1', 5040]
Replication complete for server ['127.0.0.1', 5040]
```

```
Chunk Server started on 127.0.0.1:5010
```

```
Chunk Server started on 127.0.0.1:502
0
```

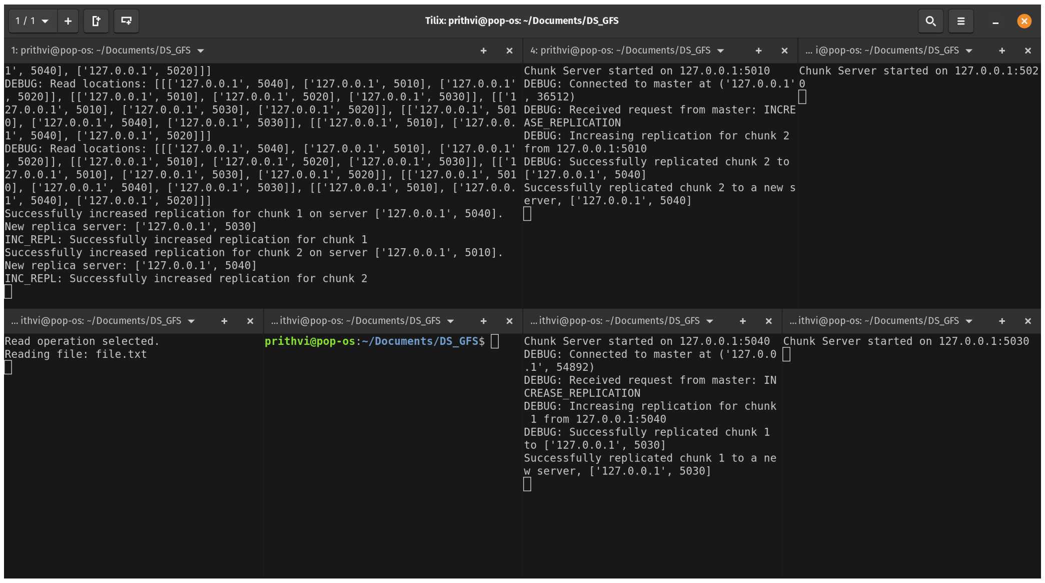
```
Read operation selected.
Reading file: t.txt
File found, retrieving chunks...
Attempting to retrieve chunk 1 from ser
ver ('127.0.0.1', 5040)
Content of file t.txt: hello, world
prithvi@pop-os:~/Documents/DS_GFS$
```

```
prithvi@pop-os:~/Documents/DS_GFS$ tree
.
├── chunkserver.py
├── chunk_storage_5010
│   └── chunk_1_replica.dat
├── chunk_storage_5020
│   └── chunk_1_replica.dat
├── chunk_storage_5030
│   └── chunk_1_replica.dat
├── chunk_storage_5040
│   └── chunk_1.dat
├── client_files
│   └── t.txt
├── client.py
├── master_metadata
│   ├── chunk_locations.json
│   └── file_to_chunks.json
├── master.py
├── precompile.sh
```

```
Chunk Server started on 127.0.0.1:5040
DEBUG: Connected to master at ('127.0.0
.1', 42924)
DEBUG: Received request from master: IN
CREASE_REPLICATION
DEBUG: Increasing replication for chunk
 1 from 127.0.0.1:5040
DEBUG: Successfully replicated chunk 1
to ['127.0.0.1', 5010]
Successfully replicated chunk 1 to a ne
w server, ['127.0.0.1', 5010]
```

```
Chunk Server started on 127.0.0.1:5030
```

Figure 4: Dynamic Replication: Replicating single chunk



```
1', 5040], ['127.0.0.1', 5020]]]
DEBUG: Read locations: [[['127.0.0.1', 5040], ['127.0.0.1', 5010], ['127.0.0.1'
, 5020]], [['127.0.0.1', 5010], ['127.0.0.1', 5020], ['127.0.0.1', 5030]], [['1
27.0.0.1', 5010], ['127.0.0.1', 5030], ['127.0.0.1', 5020]], [['127.0.0.1', 501
0], ['127.0.0.1', 5040], ['127.0.0.1', 5030]], [['127.0.0.1', 5010], ['127.0.0.
1', 5040], ['127.0.0.1', 5020]]]
DEBUG: Read locations: [[['127.0.0.1', 5040], ['127.0.0.1', 5010], ['127.0.0.1'
, 5020]], [['127.0.0.1', 5010], ['127.0.0.1', 5020], ['127.0.0.1', 5030]], [['1
27.0.0.1', 5010], ['127.0.0.1', 5030], ['127.0.0.1', 5020]], [['127.0.0.1', 501
0], ['127.0.0.1', 5040], ['127.0.0.1', 5030]], [['127.0.0.1', 5010], ['127.0.0.
1', 5040], ['127.0.0.1', 5020]]]
Successfully increased replication for chunk 1 on server ['127.0.0.1', 5040].
New replica server: ['127.0.0.1', 5030]
INC_REPL: Successfully increased replication for chunk 1
Successfully increased replication for chunk 2 on server ['127.0.0.1', 5010].
New replica server: ['127.0.0.1', 5040]
INC_REPL: Successfully increased replication for chunk 2
```

```
Chunk Server started on 127.0.0.1:5010
DEBUG: Connected to master at ('127.0.0.1'
, 36512)
DEBUG: Received request from master: INCRE
ASE_REPLICATION
DEBUG: Increasing replication for chunk 2
from 127.0.0.1:5010
DEBUG: Successfully replicated chunk 2 to
['127.0.0.1', 5040]
Successfully replicated chunk 2 to a new s
erver, ['127.0.0.1', 5040]
```

```
Chunk Server started on 127.0.0.1:502
0
```

```
Read operation selected.
Reading file: file.txt
```

```
prithvi@pop-os:~/Documents/DS_GFS$
```

```
Chunk Server started on 127.0.0.1:5040
DEBUG: Connected to master at ('127.0.0
.1', 54892)
DEBUG: Received request from master: IN
CREASE_REPLICATION
DEBUG: Increasing replication for chunk
 1 from 127.0.0.1:5040
DEBUG: Successfully replicated chunk 1
to ['127.0.0.1', 5030]
Successfully replicated chunk 1 to a ne
w server, ['127.0.0.1', 5030]
```

```
Chunk Server started on 127.0.0.1:5030
```

Figure 5: Dynamic Replication: Replicating Chunk Server

12 Conclusion

The implementation of this Google File System (GFS) project demonstrates a basic distributed system, including principles such as replication, fault tolerance, and consistency. By incorporating features such as the heartbeat mechanism, dynamic replication, and efficient read/write operations, this improves robustness and adaptability in real-world scenarios.

The source code and presentation are available at the following links:

- **GitHub Repository:** `https://github.com/chiragdhamija/DS_GFS`
- **Presentation:** Canva

13 Team Work Distribution

- **Chirag Dhamija**: Implementation of core GFS, including basic operations like read, write, append and metadata management.
- **Prithvi Karthik**: Development of dynamic replication, including workload-based replication and fault tolerance mechanisms.

14 References

[1] Jayalakshmi D S, Patnaik Ranjana, and Srinivasan Ramaswamy. Dynamic data replication across geo-distributed cloud data centres. pages 182–187, 01 2016.

[2] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, October 2003.

[3] Da-Wei Sun, Gui-Ran Chang, Shang Gao, Li-Zhong Jin, and Xing-Wei Wang. Modeling a dynamic data replication strategy to increase system availability in cloud computing environments. *Journal of Computer Science and Technology*, 27(2):256–272, March 2012.

[4] May Phyo Thu, Khine Moe Nwe, and Kyar Nyo Aye. Dynamic replication management scheme for distributed file system. In Thi Thi Zin and Jerry Chun-Wei Lin, editors, *Big Data Analysis and Deep Learning Applications*, pages 139–148, Singapore, 2019. Springer Singapore.