Sistema de Archivos Distribuido por Bloques

Integrantes:

- Alejandro Gómez Quiñones
- Joaquín Castaño Trujillo
- Sebastián Soto Ángel

Índice

Introducción

Este documento explica detalladamente la implementación de un Sistema de Archivos Distribuido (DFS) basado en bloques, siguiendo el modelo de HDFS (Hadoop Distributed File System). El sistema permite almacenar, recuperar y

gestionar archivos distribuidos en múltiples nodos, garantizando replicación y alta disponibilidad.

Arquitectura General

La arquitectura del sistema consta de tres componentes principales:

1. **NameNode**: Servidor central que gestiona los metadatos del sistema de archivos.
2. **DataNodes**: Servidores de almacenamiento que guardan los bloques de datos.
3. **Cliente**: Interfaz para interactuar con el sistema mediante comandos.

Topología del Sistema

```
[Clientes] <----> [NameNode Leader] <----> [NameNode Follower]
     |                    |                         |
     |                    v                         v
     +-------------> [DataNode-1] <------------> [DataNode-2] <--->
[DataNode-N]
```

Canales de Comunicación

Siguiendo las especificaciones del proyecto, se implementan dos canales de comunicación:

- **Canal de Control** (línea discontinua): Utiliza REST API para gestionar metadatos entre Cliente-NameNode y DataNode-NameNode.
- **Canal de Datos** (línea continua): Utiliza gRPC para transferencia eficiente de datos entre Cliente-DataNode y DataNode-DataNode.

Componentes del Sistema

1. NameNode

Funcionalidad

El NameNode es el "cerebro" del sistema, encargado de:

- Mantener la estructura de directorios y archivos
- Mapear archivos a sus bloques correspondientes
- Registrar la ubicación de cada bloque en los DataNodes
- Monitorear el estado de replicación de los bloques

- Seleccionar DataNodes óptimos para almacenamiento

Sistema de Alta Disponibilidad (NameNode Leader-Follower)

El sistema implementa un modelo de alta disponibilidad con dos roles:

NameNode Leader:

- Gestiona todas las operaciones de escritura
- Coordina la replicación de metadatos
- Asigna bloques a DataNodes
- Monitoriza el estado del sistema

NameNode Follower:

- Mantiene una copia sincronizada de los metadatos
- Participa en la elección de líder
- Proporciona failover automático en caso de caída del líder
- Puede responder consultas de solo lectura

```
# Ejemplo simplificado del sistema de elección de líder
class LeaderElection:
    def start_election(self):
        # Solicitar votos a otros nodos
        votes = self._request_votes()

        # Si obtiene mayoría, se convierte en líder
        if self._has_majority(votes):
            self._become_leader()
        else:
            self._become_follower()

    def handle_vote_request(self, term, candidate_id, last_log_index,
last_log_term):
        # Lógica para decidir si votar por un candidato
        # basado en el algoritmo Raft
        pass
```

Implementación Clave

Gestión de Metadatos:

```
# src/namenode/metadata/manager.py
class MetadataManager:
    # Gestiona todos los metadatos del sistema
    def __init__(self, db_path: str = None, node_id: str = None):
        self.db = MetadataDatabase(db_path)
        self.node_id = node_id or str(uuid.uuid4())
        self.known_nodes = set()
```

API REST:

```
# src/namenode/api/routes.py
@files_router.post("/", response_model=FileMetadata, status_code=201)
async def create_file(file_metadata: FileMetadata, manager:
MetadataManager = Depends(get_metadata_manager)):
    # Registrar un nuevo archivo en el sistema
```

Base de Datos de Metadatos

- Utiliza SQLite para almacenar metadatos de manera persistente
- Implementa tablas para archivos, bloques y ubicaciones de bloques
- Permite búsquedas rápidas de información de archivos y bloques

2. DataNode

Funcionalidad

Los DataNodes son los "músculos" del sistema, encargados de:

- Almacenar bloques de datos en el sistema de archivos local
- Proporcionar servicios gRPC para almacenar/recuperar bloques
- Replicar bloques a otros DataNodes (protocolo Leader-Follower)
- Enviar heartbeats al NameNode para reportar su estado
- Gestionar espacio de almacenamiento local

Sistema de Registro y Monitoreo

```
class DataNodeRegistration:
    def __init__(self, namenode_url, node_id, hostname, port,
storage_capacity):
        self.namenode_url = namenode_url
        self.node_id = node_id
        self.hostname = hostname
        self.port = port
        self.storage_capacity = storage_capacity

    def start_heartbeat_thread(self, stats_callback):
        # Inicia un hilo para enviar heartbeats periódicos al NameNode
        self.heartbeat_thread =
threading.Thread(target=self._heartbeat_loop, args=(stats_callback,))
        self.heartbeat_thread.daemon = True
        self.heartbeat_thread.start()
```

Estados de DataNode

```
from enum import Enum
```

```
class DataNodeStatus(Enum):
    ACTIVE = "active"
    INACTIVE = "inactive"
    DECOMMISSIONED = "decommissioned"
    MAINTENANCE = "maintenance"
```

Implementación Clave

Servicio gRPC:

```
# src/datanode/service/datanode_service.py
class DataNodeServicer(datanode_pb2_grpc.DataNodeServiceServicer):
    # Implementa servicios gRPC para operaciones de bloques
    def StoreBlock(self, request_iterator, context):
        """Almacena un bloque de datos enviado por el cliente."""
        # Recibe y almacena un bloque en chunks
```

Almacenamiento de Bloques:

```
# src/datanode/storage/block_storage.py
class BlockStorage:
    # Gestiona el almacenamiento físico de los bloques
    def store_block(self, block_id: str, data: bytes) -> Tuple[bool,
str]:
        # Almacena un bloque en el sistema de archivos local y calcula su
checksum
```

Replicación Leader-Follower:

```
# src/datanode/service/datanode_service.py
def ReplicateBlock(self, request, context):
    """Replica un bloque a otro DataNode siguiendo el protocolo Leader-
Follower."""
    # El DataNode líder recibe un bloque y lo replica a un seguidor
```

3. Cliente

Funcionalidad

El cliente proporciona la interfaz para interactuar con el sistema:

- Divide archivos en bloques para su distribución
- Coordina la subida de bloques a los DataNodes
- Descarga y reconstruye archivos a partir de bloques
- Proporciona comandos para gestionar el sistema de archivos
- Interactúa con el NameNode y los DataNodes según sea necesario

Implementación Clave

Cliente DFS Principal:

```
# src/client/dfs_client.py
class DFSClient:
    # Cliente principal para interactuar con el sistema
    def put_file(self, local_path: str, dfs_path: str, max_workers: int =
4) -> bool:
        # Sube un archivo al sistema, dividiéndolo en bloques
```

CLI (Interfaz de Línea de Comandos):

```
# src/client/cli.py
class DFSCLI:
    # Proporciona comandos para interactuar con el sistema
    def _handle_put(self, args: List[str]):
        # Maneja el comando 'put' para subir archivos
```

Divisor de Archivos:

```
# src/client/file_splitter.py
class FileSplitter:
    # Divide archivos en bloques de tamaño fijo
    def split_file(self, file_path: str) -> List[Dict]:
        # Divide un archivo en bloques y asigna IDs únicos
```

Flujo de Operaciones

1. Subida de Archivos (PUT)

1. **División en Bloques**:

- El cliente divide el archivo en bloques de tamaño fijo (configurable, por defecto 4KB para pruebas)
- Cada bloque recibe un identificador único (UUID)

2. **Distribución de Bloques**:

- El cliente solicita al NameNode una lista de DataNodes disponibles
- Se seleccionan DataNodes para cada bloque según disponibilidad y carga

3. **Escritura Directa**:

- El cliente envía cada bloque directamente al DataNode asignado usando gRPC
- El primer DataNode que recibe un bloque se designa como "Leader" para ese bloque

4. **Replicación**:

- El DataNode Leader replica el bloque a otro DataNode (Follower)
- La replicación garantiza que cada bloque exista en al menos dos DataNodes

5. **Registro de Metadatos**:

- El cliente registra el archivo y sus bloques en el NameNode
- El NameNode almacena la información de ubicación de los bloques

```
Cliente ---(REST)---> NameNode [Obtiene DataNodes óptimos]
Cliente ---(gRPC)---> DataNode Leader [Envía bloque]
DataNode Leader ---(gRPC)---> DataNode Follower [Replica bloque]
Cliente ---(REST)---> NameNode [Registra metadatos]
```

2. Descarga de Archivos (GET)

1. **Solicitud de Metadatos**:

- El cliente solicita al NameNode información sobre el archivo
- El NameNode devuelve la lista de bloques y sus ubicaciones en DataNodes

2. **Descarga Paralela**:

- El cliente descarga bloques en paralelo desde los DataNodes
- Si un DataNode falla, se intenta con otro que tenga una réplica

3. **Reconstrucción**:

- El cliente reconstruye el archivo original a partir de los bloques

```
Cliente ---(REST)---> NameNode [Obtiene info de bloques]
Cliente ---(gRPC)---> DataNode 1 [Descarga bloque 1]
Cliente ---(gRPC)---> DataNode 2 [Descarga bloque 2]
...
Cliente [Reconstruye el archivo]
```

Detalle de Componentes Clave

Particionamiento de Archivos

El sistema divide los archivos en bloques de tamaño configurable para facilitar:

- Distribución equilibrada entre nodos
- Paralelismo en lectura/escritura
- Mejor tolerancia a fallos

- Replicación eficiente

```
# src/client/dfs_client.py
def _split_file_into_blocks(self, file_path: str) -> List[Dict]:
    blocks = []
    file_size = os.path.getsize(file_path)

    # Si el archivo es más pequeño que el tamaño de bloque, crear un solo
bloque
    if file_size <= self.block_size:
        blocks.append({
            'offset': 0,
            'size': file_size,
            'block_id': str(uuid.uuid4())
        })
        return blocks

    # Dividir el archivo en bloques
    offset = 0
    while offset < file_size:
        remaining = file_size - offset
        block_size = min(remaining, self.block_size)

        blocks.append({
            'offset': offset,
            'size': block_size,
            'block_id': str(uuid.uuid4())
        })

        offset += block_size

    return blocks
```

Replicación Leader-Follower

La replicación de bloques sigue un patrón Leader-Follower:

- El primer DataNode que recibe un bloque es designado "Leader"
- El Leader es responsable de replicar el bloque a un DataNode "Follower"
- Esta estructura permite rastrear claramente la procedencia de cada bloque

```
# src/client/dfs_client.py
def _upload_block(self, file_path: str, block: Dict, datanodes:
List[Dict], is_leader: bool = False) -> bool:
    # El parámetro is_leader determina si este DataNode es líder para
este bloque
    # Se asigna como líder al primer DataNode de la lista

    # Al registrar la ubicación en el NameNode, se indica si es líder
    self.namenode_client.add_block_location(
        block['block_id'],
        datanode['node_id'],
        is_leader=is_leader
    )
```

Tolerancia a Fallos

El sistema implementa varios mecanismos para garantizar la tolerancia a fallos:

1. **Replicación de Bloques**:

- Cada bloque se almacena en al menos dos DataNodes
- Si un DataNode falla, los bloques siguen disponibles en otros nodos

2. **Heartbeats de DataNodes**:

- Los DataNodes envían regularmente señales de vida al NameNode
- El NameNode detecta DataNodes caídos cuando dejan de enviar heartbeats

3. **Reintentos en Descarga**:

- Al descargar archivos, si un bloque falla, se intenta con otras réplicas
- Múltiples reintentos con diferentes DataNodes

4. **Failover de NameNode**:

- Si el NameNode Leader falla, el Follower asume el rol de líder
- Proceso de elección basado en algoritmo similar a Raft
- Sincronización continua de metadatos entre Leader y Follower

```
# src/client/dfs_client.py
def download_block(self, block_info):
    # Si falla la descarga desde un DataNode, intenta con otros
    for location in locations:
        try:
            with DataNodeClient(hostname, port) as datanode:
                block_data = datanode.retrieve_block(block_id)
                if block_data:
                    return True, block_id, block_data
        except Exception as e:
            errors.append(f"Error con DataNode {datanode_id}: {str(e)}")
            continue

    # Si todos fallan, retorna error
    return False, block_id, None
```

Interfaz de Línea de Comandos (CLI)

El cliente implementa una interfaz de comandos completa para interactuar con el sistema:

- `put <archivo_local> <ruta_dfs> [--workers=N]`: Sube un archivo al DFS
- `get <ruta_dfs> <archivo_local> [--workers=N]`: Descarga un archivo del DFS
- `ls [ruta] [-l]`: Lista el contenido de un directorio
- `mkdir <ruta> [-p]`: Crea un directorio
- `rmdir <ruta> [-r] [-f]`: Elimina un directorio
- `rm <ruta> [-f]`: Elimina un archivo
- `cd <ruta>`: Cambia el directorio actual
- `info <ruta>`: Muestra información detallada de un archivo
- `status`: Muestra el estado del sistema

```
# src/client/cli.py
def run(self):
    """Ejecuta el bucle principal del CLI."""
    while True:
        command_line = input(f"dfs:{self.current_dir}> ")
        if not command_line.strip():
            continue

        parts = command_line.strip().split()
        command = parts[0].lower()
        args = parts[1:]

        # Mapeo de comandos a sus manejadores
        if command == "exit" or command == "quit":
            break
        elif command == "put":
            self._handle_put(args)
        elif command == "get":
            self._handle_get(args)
        # ... otros comandos
```

Flujo de Datos

API REST (Canal de Control)

La comunicación con el NameNode se realiza mediante una API REST:

1. **Endpoints de Archivos**:

- `/files`: Creación, recuperación y eliminación de archivos
- `/files/path/{path}`: Operaciones con archivos por ruta
- `/files/info/{path}`: Información detallada de archivos

2. **Endpoints de Bloques**:

- `/blocks`: Gestión de bloques y sus ubicaciones
- `/blocks/{block_id}`: Información de bloques específicos
- `/blocks/file/{file_id}`: Bloques asociados a un archivo

3. **Endpoints de DataNodes**:

- `/datanodes/register`: Registro de nuevos DataNodes
- `/datanodes/{node_id}/heartbeat`: Heartbeats de DataNodes
- `/datanodes/`: Listado de DataNodes

4. **Endpoints de Directorios**:

- `/directories/`: Creación de directorios
- `/directories/{path}`: Listado y eliminación de directorios

gRPC (Canal de Datos)

La transferencia de datos entre Cliente-DataNode y DataNode-DataNode se realiza mediante gRPC:

1. **Servicios del DataNode**:

- `StoreBlock`: Almacena un bloque enviado en chunks
- `RetrieveBlock`: Recupera un bloque y lo envía en chunks
- `ReplicateBlock`: Replica un bloque a otro DataNode
- `TransferBlock`: Transfiere un bloque a otro DataNode
- `CheckBlock`: Verifica si un bloque existe y su integridad

```
// src/common/proto/datanode.proto (Simplificado)
service DataNodeService {
  rpc StoreBlock(stream BlockData) returns (BlockResponse);
  rpc RetrieveBlock(BlockRequest) returns (stream BlockData);
  rpc ReplicateBlock(ReplicationRequest) returns (BlockResponse);
  rpc TransferBlock(TransferRequest) returns (BlockResponse);
  rpc CheckBlock(BlockRequest) returns (BlockStatus);
  rpc DeleteBlock(BlockRequest) returns (BlockResponse);
}
```

Configuración y Optimización

Tamaño de Bloques

El sistema permite configurar el tamaño de bloque:

- Por defecto: 4KB para pruebas (configurable hasta 64KB o más)
- El tamaño pequeño facilita la distribución y pruebas
- En un entorno de producción, se recomendaría un tamaño mayor (64MB-128MB)

Límites de Transferencia gRPC

Para transferencias eficientes, el sistema configura límites adecuados en gRPC:

```
options = [
    ('grpc.max_send_message_length', 8 * 1024 * 1024),  # 8MB
    ('grpc.max_receive_message_length', 8 * 1024 * 1024)  # 8MB
]
```

Paralelismo

El sistema utiliza paralelismo para operaciones de lectura/escritura:

- Transferencia de bloques en paralelo usando ThreadPoolExecutor
- Número configurable de workers para operaciones en paralelo
- Barra de progreso para seguimiento visual

Estructuras de Datos Clave

DataNodeInfo

```
class DataNodeInfo:
    node_id: str                  # Identificador único del DataNode
    hostname: str                 # Nombre de host o IP
    port: int                     # Puerto gRPC
    status: DataNodeStatus        # Estado actual (ACTIVE, INACTIVE,
etc.)
    storage_capacity: int         # Capacidad total de almacenamiento
    available_space: int          # Espacio disponible actual
    last_heartbeat: datetime      # Último heartbeat recibido
    blocks_stored: int            # Número de bloques almacenados
```

BlockInfo

```
class BlockInfo:
    block_id: str                 # Identificador único del bloque
    file_id: str                  # ID del archivo al que pertenece
    size: int                     # Tamaño en bytes
    checksum: str                 # Checksum para verificación de
integridad
    locations: List[BlockLocation] # Ubicaciones del bloque
```

BlockLocation

```
class BlockLocation:
    block_id: str                 # ID del bloque
    datanode_id: str              # ID del DataNode
```

```
    is_leader: bool                # Si este DataNode es líder para este
bloque
```

FileMetadata

```
class FileMetadata:
    file_id: str                   # Identificador único del archivo
    name: str                      # Nombre del archivo
    path: str                      # Ruta completa en el DFS
    type: FileType                 # FILE o DIRECTORY
    size: int                      # Tamaño en bytes
    blocks: List[str]              # IDs de los bloques
    created_at: datetime           # Fecha de creación
    modified_at: datetime          # Fecha de modificación
    owner: str                     # Propietario del archivo
```

Guía de Uso del Sistema

Esta sección proporciona instrucciones detalladas para poner en marcha el sistema DFS completo, desde la instalación de dependencias hasta la ejecución de todos los componentes.

Requisitos e Instalación

1. **Python 3.8+**: El sistema está desarrollado en Python y requiere una versión 3.8 o superior.
2. **Dependencias**: Instala todas las dependencias necesarias utilizando pip:

```
pip install -r requirements.txt
```

Las principales dependencias incluyen:

- FastAPI: Para la API REST del NameNode
- gRPC: Para la comunicación de datos
- SQLite: Para el almacenamiento de metadatos
- Pydantic: Para la validación y serialización de datos
- tqdm: Para mostrar barras de progreso

Ejecución del Sistema

Para poner en marcha el sistema completo, es necesario iniciar el NameNode y varios DataNodes en terminales separadas.

1. Iniciar el NameNode

Abre una terminal y ejecuta:

```
python -m src.namenode.api.main --id namenode1 --host localhost --rest-port 8000 --grpc-port 50051
```

Parámetros:

- `--id`: Identificador único para este NameNode
- `--host`: Dirección en la que escuchará el servidor
- `--rest-port`: Puerto para la API REST (canal de control)
- `--grpc-port`: Puerto para comunicación gRPC entre NameNodes

2. Iniciar los DataNodes

Para una configuración robusta con replicación, se recomienda iniciar al menos 3 DataNodes. Abre una terminal nueva para cada DataNode y ejecuta:

DataNode 1:

```
python -m src.datanode.main --node-id datanode1 --hostname localhost --port 7001 --storage-dir ./data/datanode1 --namenode-url http://localhost:8000
```

DataNode 2:

```
python -m src.datanode.main --node-id datanode2 --hostname localhost --port 7002 --storage-dir ./data/datanode2 --namenode-url http://localhost:8000
```

DataNode 3:

```
python -m src.datanode.main --node-id datanode3 --hostname localhost --port 7003 --storage-dir ./data/datanode3 --namenode-url http://localhost:8000
```

Parámetros:

- `--node-id`: Identificador único para este DataNode
- `--hostname`: Nombre de host o IP donde escuchará el DataNode
- `--port`: Puerto gRPC para transferencia de datos
- `--storage-dir`: Directorio local donde se almacenarán los bloques
- `--namenode-url`: URL del NameNode para registro y heartbeats

3. Iniciar el Cliente CLI

Finalmente, abre otra terminal para interactuar con el sistema a través del cliente CLI:

```
python -m src.client.cli --namenode http://localhost:8000
```

Esto iniciará una consola interactiva con el prompt `dfs:/>` donde podrás ingresar comandos para interactuar con el sistema.

Uso del Cliente

Una vez iniciado el cliente CLI, puedes utilizar los siguientes comandos:

```
dfs:/> help
Comandos disponibles:
  put <archivo_local> <ruta_dfs> [--workers=N]  - Sube un archivo al DFS
  get <ruta_dfs> <archivo_local> [--workers=N]  - Descarga un archivo del
DFS
  ls [ruta] [-l]                                - Lista contenido de un
directorio
  mkdir <ruta> [-p]                             - Crea un directorio
  rmdir <ruta> [-r] [-f]                        - Elimina un directorio
  rm <ruta> [-f]                                - Elimina un archivo
  cd <ruta>                                     - Cambia el directorio
actual
  info <ruta>                                   - Muestra información de
un archivo
  status                                        - Muestra estado del
sistema
  exit, quit                                    - Salir del CLI
```

Ejemplos de uso:

```
# Subir un archivo
dfs:/> put /ruta/local/archivo.txt /carpeta/archivo.txt

# Descargar un archivo
dfs:/> get /carpeta/archivo.txt /ruta/local/archivo_descargado.txt

# Listar archivos con detalles
dfs:/> ls -l

# Ver estadísticas del sistema
dfs:/> status
```

Documentación API

El NameNode proporciona documentación interactiva de su API REST. Para acceder a ella, el NameNode debe estar en ejecución:

- **Swagger UI**: http://localhost:8000/docs
- Interfaz interactiva donde puedes probar endpoints y ver ejemplos
- **ReDoc**: http://localhost:8000/redoc
- Documentación más formal y detallada de la API
- **Listado de DataNodes**: http://localhost:8000/datanodes/
- Muestra información sobre todos los DataNodes registrados
- **Especificación OpenAPI**: http://localhost:8000/openapi.json
- Especificación completa en formato JSON (no amigable para humanos)

Estas interfaces permiten explorar todos los endpoints disponibles, sus parámetros, esquemas de datos y respuestas posibles.

Conclusión

Este Sistema de Archivos Distribuido implementa todas las especificaciones requeridas:

- División de archivos en bloques y distribución entre DataNodes
- Replicación de bloques con modelo Leader-Follower
- Canales de comunicación separados para control (REST) y datos (gRPC)
- Transferencia directa Cliente-DataNode para eficiencia
- Tolerancia a fallos mediante replicación y detección de nodos caídos
- Alta disponibilidad con NameNode Leader-Follower
- Interfaz CLI completa con todos los comandos requeridos

La implementación sigue principios de modularidad, robustez y eficiencia, ofreciendo un sistema funcional que emula las características principales de HDFS a escala reducida.