**ST0263 Tópicos Especiales en Telemática / SI3007 Sistemas Distribuidos, 2025-2.**

**GridFS**

**Proyecto 1:** *Diseño e implementación de un sistema de archivos distribuido por bloques (GridDFS)*

**Objetivo General:**

Diseñar e implementar un **DFS minimalista** que permita almacenar y acceder a archivos grandes de forma distribuida en múltiples nodos, aplicando **particionamiento en bloques** pero sin replicación. Este DFS debe permitir que un cliente cargue y recupere archivos, dividiéndolos en bloques y distribuyéndolos entre distintos nodos de datos (DataNodes), con un NameNode central que gestione la ubicación de los bloques.

**Descripción del Proyecto:**

Un sistema de archivos distribuidos, permite compartir y acceder de forma concurrente un conjunto de archivos que se encuentran almacenados en diferentes nodos. Uno de los sistemas más maduros, vigente y antiguo de estos sistemas es el NFS (Network File System) desarrollado en su momento por Sun Microsystems y que hoy en día es ampliamente usado en sistemas Linux. Hay otros sistemas de archivos distribuidos como AFS (Andrew File System) y SMB (Server Message Block) conocido como CIFS.

En general hay dos acercamientos para el diseño e implementación de un DFS: 1) basado en bloques y basado en objetos.

Los DFS basados en bloques generalmente garantizan 2 aspectos: 1) la unidad de escritura y lectura es a nivel de bloque, y los bloques pueden ser distribuidos en diferentes nodos, la idea es que los bloques de un archivo estén distribuidos en un conjunto de nodos. 2) el sistema operativo cliente de un DFS garantiza transparencia en el sentido de que la API ofrecida desde el SO es igual para acceder archivos locales que remotos, porque el DFS se integra con el sistema de gestión de archivos del sistema operativo (ej: NFS, AFS, SMB, etc).

Los DFS basados en objetos (object storage, ej: AWS S3), los dos aspectos anteriores se manejan así: 1) la unidad de distribución es a nivel de archivo y no de bloque, es decir, y se garantiza que se lee o escribe un archivo como un todo y no a nivel de bloque. No está diseñado como un sistema de acceso aleatorio al archivo sino secuencial. No soporta la operación de actualización parcial del archivo, sino que se debe reemplazar todo el archivo. Son sistemas distribuidos de archivos principalmente diseñado para un enfoque WORM (Read-Once-Read-Many). Típicamente estos DFS soporta altos niveles de escalabilidad, redundancia y rendimiento. Si bien desde el cliente tiene una visión de archivo completo, en el sistema de backend podría tener (y normalmente lo hay) un mecanismo de particionamiento del archivo por bloques u otro criterio para mejorar la escalabilidad, tolerancia a fallos y rendimiento. 2) el sistema operativo local del cliente NO integra directamente la gestión de este DFS y en vez de ello se cuenta con un SDK o API para las diferentes primitivas de la gestión de archivos y normalmente tienen su propio CLI.

A nivel de recomendación para este proyecto1, realizaremos el diseño e implementación del DFS intermedio principalmente orientado a bloques, pero con la característica de WORM del almacenamiento por Objetos. Este tipo de DFS es el enfoque de sistemas de archivos como GFS y HDSF. Lo primero que deberá hacer el equipo de trabajo es leer y comprender los papers fundacionales de estos DFS:

* GFS:
  + <https://es.wikipedia.org/wiki/Google_File_System>
  + The Google File System - <https://g.co/kgs/XzwmU76>
* HDFS
  + <https://es.wikipedia.org/wiki/Hadoop_Distributed_File_System>
  + The Hadoop Distributed File System – [link](https://ieeexplore.ieee.org/document/5496972)

**Componentes del proyecto**

**1. Definición del Servicio**

Cada grupo de estudiantes debe completar la definición de este proyecto, que permita hacer más explícito el entendimiento del mismo y los requerimientos funcionales y no funcionales.

En este proyecto se construirá un **DFS por bloques minimalista, sin replicación**, siguiendo una arquitectura tipo HDFS:

* Un **NameNode** central que mantiene la **tabla de metadatos** (ubicación de bloques por archivo).
* Múltiples **DataNodes** que almacenan los bloques.
* Clientes que interactúan con el DFS a través de una **CLI (interfaz de línea de comandos)** o API mínima.

**Operaciones básicas:**

* **put <archivo>**: dividir un archivo en bloques y distribuirlos en DataNodes
* **get <archivo>**: recuperar un archivo reconstruyéndolo a partir de los bloques
* **ls, rm, mkdir, rmdir**: comandos básicos de gestión
* Cada cliente gestiona **solo sus propios archivos** mediante una autenticación básica (user/pass).
* Además debe contar con APIs – operaciones basica de lectura / escritura de archivos (bloques) desde clientes GridMR.

**2. Arquitectura del Sistema**

* **Definir tipo de arquitectura** C/S, P2P o alguna variante o integración de las mismas.
* **Tipo de arquitectura:** Maestro – Trabajador (Master–Workers)
* **Tecnologías:** Ejecución nativa en linux o en Docker para encapsular cada nodo, Python/Java/Scala para la lógica.
* **Comunicaciones de red:** el proyecto debe correr o ejecutar sobre Internet (p. ej., nodos comunicándose por REST, gRPC o sockets TCP).

1. **Componentes:**
   * **Cliente:** CLI/API para subir, bajar y gestionar archivos.
   * **NameNode:** servidor central de metadatos (ubicación de bloques y tamaño de archivos).
   * **DataNodes:** servidores que almacenan bloques.
2. **Requisitos:**
   * El tamaño de bloque puede ser **configurable** (ej. 64 MB).
   * Cada archivo se **particiona** en bloques distribuidos **sin replicación**.
   * Comunicación por **REST/gRPC/sockets TCP**.
   * Ejecución **nativa en Linux o Docker**.

**3. Protocolos de Comunicación**

* Definir cómo se comunican:
  + Cliente ↔ NameNode
  + NameNode ↔ DataNode
  + Entre nodos DataNode
* Sugerencias:
  + REST sobre HTTP
  + gRPC para comunicaciones más eficientes
  + WebSockets (si se requiere bidireccionalidad)
  + MOMs si requiere comunicaciones asincrónicas (ej: apache kafka, rabbitmq, etc)

**4. Flujo general:**

1. Cliente se autentica en el sistema
2. Cliente solicita put al NameNode.
3. NameNode define un **esquema de particionamiento** y asigna DataNodes.
4. Cliente envía bloques directamente a los DataNodes asignados. Esta **transferencia de bloques debe ser confiable**
5. NameNode guarda metadatos (archivo → lista de bloques → ubicación en DataNodes).
6. Para get, el NameNode retorna la lista de ubicaciones y el cliente recupera los bloques para reconstruir el archivo.

Igualmente cada grupo de trabajo tiene la libertad de definir otro mecanismo más óptimo de acceso a los datos.

**5. Infraestructura y Entorno**

* Usar AWS Academy con máquinas virtuales (IaaS)
* Cada nodo debe ejecutarse nativamente o en un contenedor Docker
* Los contenedores pueden estar en diferentes máquinas (uso de puertos expuestos vía Internet o red local)
* Cada nodo debe exponer una API

**Entregables**

1. **Informe técnico (PDF o Word):**
   * Objetivo y Marco teórico breve
   * Descripción del servicio y problema abordado
   * Arquitectura del sistema y diagramas
   * Especificación de protocolos y APIs
   * Algoritmos de particionamiento y distribución
   * Descripción del entorno de ejecución nativo o en Docker
   * Pruebas y Análisis de resultados
2. **Código fuente** en repositorio GitHub (bien documentado)
3. **Video demostración (10-15 min):**
   * Explicación del sistema
   * Ejecución en vivo o simulada del procesamiento distribuido

**Criterios de Evaluación**

| **Criterio** | **Ponderación** | **Aspectos Evaluados** |
| --- | --- | --- |

| **1. Diseño arquitectónico y documentación** | **20%** | - Claridad en la definición de componentes (Cliente, NameNode, DataNodes) - Diagramas de arquitectura y flujos de datos correctos - Descripción de protocolos y APIs (REST, gRPC, sockets) - Documentación completa y entendible |
| --- | --- | --- |

| **2. Gestión de bloques y metadatos** | **20%** | - Particionamiento correcto de archivos en bloques - Registro y recuperación de metadatos en el NameNode - Reconstrucción de archivos a partir de bloques - Manejo básico de errores por bloques faltantes o corruptos |
| --- | --- | --- |

| **3. Implementación funcional de CLI/API** | **20%** | - CLI funcional (put, get, ls, rm, mkdir, rmdir) - Autenticación básica de usuarios - Respuestas correctas ante errores y entradas inválidas - API documentada para interacciones programáticas |
| --- | --- | --- |

| **4. Comunicación entre nodos** | **15%** | - Cliente ↔ NameNode: solicitud y entrega de metadatos - Cliente ↔ DataNodes: transferencia de bloques confiable - DataNodes ↔ NameNode: registro inicial y reporte de bloques - **Manejo de fallas sin replicación**: detección de caídas, mensajes claros y logs del sistema |
| --- | --- | --- |

| **5. Resultados y demostración en entorno distribuido** | **15%** | - Ejecución del DFS en al menos 3 nodos distribuidos o contenedores - Subida y descarga de archivos sin errores (salvo caídas de nodos) - Evidencia de distribución de bloques en los DataNodes - **Si un DataNode falla**: el sistema debe informar el error correctamente y continuar operativo para otros archivos |
| --- | --- | --- |

| **6. Video e informe final** | **10%** | - Video (10–15 min) claro mostrando la ejecución del sistema - Informe técnico completo (objetivos, arquitectura, implementación, pruebas, resultados) - Repositorio GitHub bien documentado |
| --- | --- | --- |

**Recursos Sugeridos**

* **Sistema operativo Linux**
* **Docker:** para simular clúster local distribuido
* **FastAPI / Flask:** para APIs de comunicación entre nodos
* **gRPC:** para alternativas eficientes a HTTP REST
* **Apache kafka o rabbitmq** para comunicaciones asincrónicas
* **Bases de datos,** para almacenamiento temporal