**ENTREGA 1: ARQUITECTURA DE NUBE Y SISTEMAS DISTRIBUIDOS**

**FUNCIONAMIENTO DE UN SISTEMA P2P**

**REALIZADO POR**

**SANTIAGO MAYA HORTA**

**MATIAS ARANGO RUIZ**

**DOCENTE**

**ALVARO ENRIQUE OSPINA SANJUAN**

**UNIVERSIDAD PONTIFICIA BOLIVARIANA**

**Resumen Ejecutivo**

Se implementó un sistema P2P híbrido para compartición de archivos que combina un servidor de directorio centralizado (Maestro) con comunicación peer-to-peer directa. El sistema utiliza microservicios REST para consultas y descubrimiento, y gRPC para transferencias de archivos, demostrando comunicación efectiva entre procesos distribuidos con soporte de concurrencia y tolerancia a fallas.

**1. Objetivo y Marco teórico breve**

**Objetivo General:** Implementar un sistema P2P distribuido donde cada nodo/proceso contiene uno o más microservicios que soportan un sistema de compartición de archivos distribuido y descentralizado, empleando middleware REST y gRPC para comunicación entre componentes concurrentes.

**Marco Teórico:**

Los sistemas Peer-to-Peer (P2P) permiten que nodos con roles equivalentes compartan recursos sin depender exclusivamente de una autoridad centralizada. Existen principalmente dos tipos de arquitecturas P2P:

- Redes P2P no estructuradas: Utilizan técnicas como flooding o random walk para descubrimiento

- Redes P2P estructuradas: Emplean DHT (Distributed Hash Tables) como Chord o Kademlia

**Arquitectura Híbrida Implementada:**

El sistema implementa una red P2P híbrida que combina:

**- Componente centralizado**: Servidor Maestro que actúa como Directorio/Localización para bootstrap inicial

**- Componente descentralizado**: Red de peers amigos que permite consultas distribuidas directas

**-** **Middleware de comunicación**: REST (HTTP/JSON) para consultas y APIs públicas, gRPC (HTTP/2 + Protocol Buffers) para transferencias de archivos de alto rendimiento

Esta arquitectura permite el mejor balance entre eficiencia de descubrimiento (centralizado) y robustez distributiva (descentralizado), donde si el Maestro falla, los peers pueden continuar operando a través de su red de amigos.

**2. Descripción del servicio y problema abordado**

**Problemática Identificada:**

Los sistemas de compartición de archivos centralizados presentan puntos únicos de falla y limitaciones de escalabilidad. Los usuarios necesitan acceder a archivos distribuidos en múltiples nodos de una red sin depender completamente de un servidor central, pero requieren un mecanismo eficiente de descubrimiento inicial.

**Descripción del Servicio:**

El sistema resuelve el problema de descubrimiento, localización y transferencia de archivos en un entorno distribuido mediante:

1. **Gestión distribuida de recursos:** Cada peer mantiene su propio conjunto de archivos en un directorio local (`shared/`) y los registra automáticamente en el sistema
2. **Descubrimiento híbrido:**

* Centralizado vía Maestro para eficiencia inicial
* Descentralizado vía red de peers amigos para robustez

1. **Transferencias directas:** Los archivos se transfieren directamente entre peers usando gRPC, evitando cuellos de botella centralizados
2. **Sincronización automática:** El sistema actualiza automáticamente el directorio maestro tras cada operación de transferencia
3. **Tolerancia a fallas:** Si un peer no responde, el sistema puede localizar el archivo en otros peers; si el Maestro falla, los peers continúan operando via red de amigos

**Arquitectura de Microservicios:**

Cada nodo implementa múltiples microservicios especializados:

1. **Servicio de Registro (REST)**: Comunicación con Maestro para Bootstrap
2. **Servicio de Consulta (REST):** Búsqueda y listado de archivos locales y remotos
3. **Servicio de Descubrimiento (REST)**: Localización distribuida via peers amigos
4. **Servicio de Transferencia (gRPC):** Upload/Download de archivos con alta eficiencia
5. **Cliente P2P**: Interfaz unificada para interactuar con todos los servicios

**Beneficios del Diseño:**

1. Separación clara de responsabilidades entre microservicios
2. Escalabilidad horizontal mediante adición de nuevos peers
3. Comunicación eficiente con protocolos especializados (REST para metadata, gRPC para datos)
4. Robustez ante fallas individuales de componentes

**3. Arquitectura del sistema y diagramas**

**Diseño Arquitectónico General:**

El sistema implementa una arquitectura de microservicios distribuidos con comunicación híbrida REST/gRPC. La topología combina un servidor central para bootstrap con una red mallada de peers para operaciones distribuidas.

**Componentes Principales:**

**3.1 Servidor Maestro (Puerto 9000)**

**- Responsabilidad**: Directorio centralizado y registro de peers

**- Servicios:**

* Registro automático de peers al inicio (POST /register)
* Consulta de directorio global (GET /directory)
* Localización de archivos (GET /locate)
* Listado de peers activos (GET /peers)

**- Tecnología**: FastAPI + Uvicorn ASGI Server

**- Persistencia**: En memoria (diccionario Python)

**3.2 Nodos Peer (P1, P2, P3)**

Cada peer ejecuta \*\*microservicios concurrentes\*\* en procesos separados:

**a) Servidor REST (Puertos 9001, 9003, 9005):**

- Gestión de estado y metadatos del peer

- Consulta de archivos locales (/files)

- Búsqueda local de archivos (/search)

- Descubrimiento peer-to-peer sin Maestro (/locate)

- Información del peer (/info)

- Actualización manual del registro (/refresh)

**b) Servidor gRPC (Puertos 9002, 9004, 9006):**

- Transferencias directas de archivos (Upload/Download)

- Comunicación binaria optimizada con Protocol Buffers

- Concurrencia mediante ThreadPoolExecutor

- Actualización automática del Maestro post-transferencia

**c) Cliente P2P (client.py):**

- Interfaz unificada para operaciones distribuidas

- Soporte para descubrimiento vía Maestro o peer-to-peer

- Manejo de múltiples protocolos (REST + gRPC)

**3.3 Red de Peers Amigos**

Configuración de topología distribuida:

Peer 1: Amigos → P2 (primario), P3 (backup)

Peer 2: Amigos → P1 (primario), P3 (backup)

Peer 3: Amigos → P1 (primario), P2 (backup)

Esta red permite consultas distribuidas independientes del Maestro.

**Diagrama de Arquitectura:**

**Escala de tiempo

El contenido generado por IA puede ser incorrecto.**

**Patrones Arquitectónicos Implementados:**

1. Microservicios: Separación de responsabilidades por servicio
2. API Gateway: Maestro como punto de entrada centralizado
3. Service Discovery: Registro automático y dinámico de peers
4. Circuit Breaker: Fallback a peers amigos si Maestro no responde
5. Load Balancing: Distribución automática de carga entre peers con mismo archivo

**4. Especificación de protocolos y APIs**

**4.1 Protocolo REST (HTTP/JSON)**

**Servidor Maestro - Puerto 9000:**

| **Endpoint** | **Método** | **Descripción** | **Ejemplo de Respuesta** |
| --- | --- | --- | --- |
| / | GET | Estado del servidor maestro | { "message": "Servidor Maestro P2P", "status": "activo" } |
| /register | POST | Registro de un peer con sus archivos | { "status": "registered", "peer": "http://ip:port", "files": [...] } |
| /peers | GET | Lista de peers registrados | { "peers": [...] } |
| /locate | GET | Localización de un archivo específico | { "file": "nombre", "peers": [...] } |
| /directory | GET | Directorio global con todos los archivos | { "directory": {...} } |

**Peers - Puertos 9001, 9003, 9005:**

| **Endpoint** | **Método** | **Descripción** | **Ejemplo** |
| --- | --- | --- | --- |
| / | GET | Estado del peer | { "message": "Peer P1", "status": "activo" } |
| /files | GET | Lista de archivos locales | { "files": ["a.txt", "b.pdf"] } |
| /search | GET | Busca un archivo en el peer | { "file": "a.txt", "exists": true } |
| /locate | GET | Búsqueda distribuida vía amigos | { "file": "x", "peers": [...] } |
| /info | GET | Información completa del peer | { "id": "P1", "grpc\_port": 9002 } |

**4.2 Protocolo gRPC (Protocol Buffers)**

**Definición del servicio (files.proto):**

service FileService {

rpc Upload(UploadRequest) returns (UploadResponse);

rpc Download(DownloadRequest) returns (DownloadResponse);

}

**Implementación de servicios gRPC:**

1. Upload: Recibe archivo binario, lo guarda en directorio `shared/`, actualiza automáticamente registro en Maestro
2. Download: Lee archivo desde directorio local y lo retorna como stream de bytes
3. Concurrencia: ThreadPoolExecutor con hasta 10 workers concurrentes por peer
4. Error handling: Respuestas estructuradas con códigos de estado y mensajes descriptivos

**4.3 Algoritmos de Comunicación**

**Flujo de Registro de Peer:**

1. Peer inicia y lee configuración desde `config.json`
2. Escanea directorio `shared/` para obtener lista de archivos
3. Envía POST `/register` al Maestro con metadata del peer
4. Maestro actualiza directorio global y registro de peers
5. Peer queda disponible para consultas y transferencias

**Flujo de Descubrimiento:**

1. Centralizado: Cliente pregunta al Maestro (/locate)
2. Distribuido: Cliente pregunta a peers amigos (/locate)

**Flujo de Transferencia**

1. Localizar archivo (Maestro o peers)
2. Cliente conecta vía gRPC al peer propietario
3. Solicita archivo → lo recibe en binario
4. .Registro se sincroniza

**4.4 Manejo de Concurrencia y Fallas**

**Concurrencia:**

1. Servidor REST: Asíncrono vía FastAPI + Uvicorn
2. Servidor gRPC: ThreadPoolExecutor para múltiples transferencias simultáneas
3. Cliente: Soporte para múltiples conexiones paralelas

**Tolerancia a Fallas:**

1. Maestro no disponible: Peers usan red de amigos para descubrimiento
2. Peer no responde: Cliente intenta con siguiente peer en lista
3. Transferencia falla: Cliente reintenta con timeout configurable
4. Registro desactualizado: Endpoint /refresh para sincronización manual

**5. Algoritmos de participamiento y distribución**

**5.1 Estrategia de Descubrimiento Híbrido**

**Algoritmo de Registro Inicial (Bootstrap)**

1. Leer configuración (config.json)
2. .Escanear carpeta shared/ para inventario de archivos
3. Intentar registro en Maestro (POST /register)
4. Si éxito → Estado REGISTERED
5. .Si falla → Estado STANDALONE (funciona solo con peers amigos)
6. Inicia servidores REST y gRPC concurrentemente

**Descubrimiento de Archivos**

**Opciones disponibles:**

* Vía Maestro: Consulta rápida con /locate
* Vía Peer Amigo: El peer busca en su red primaria y secundaria.

**Lógica resumida:**

1. Intentar búsqueda en Maestro
2. Si falla → Buscar en peer amigo primario
3. Si no responde → Buscar en peer amigo de respaldo
4. Si tampoco responde → Reintentar con Maestro

**5.2 Algoritmo de Distribución de Carga**

**Distribución de Carga**

Para evitar sobrecargar a un único peer:

1. Se usa un algoritmo Round-Robin simple para elegir qué peer responderá.
2. Si un peer no responde, se prueba con el siguiente de la lista.

**5.3 Algoritmo de Consistencia Eventual**

**Consistencia Eventual**

Después de cada transferencia de archivo, el peer actualiza automáticamente su registro en el Maestro:

1. Guarda el archivo en shared/
2. Escanea el nuevo inventario
3. Envía actualización al Maestro (POST /register)
4. Si falla, se mantiene en operación (inconsistencia temporal)

**5.4 Topología de Red de Peers Amigos**

**Topología de Peers Amigos**

1. Cada peer tiene un amigo primario y un amigo de respaldo:

* Peer 1 → P2 (primario), P3 (backup)
* Peer 2 → P1 (primario), P3 (backup)
* Peer 3 → P1 (primario), P2 (backup)

**Ventajas:**

* Conectividad asegurada (máximo 2 saltos).
* Redundancia si un peer falla.
* Escalabilidad al agregar nuevos peers.

**5.5 Algoritmo de Tolerancia a Fallas**

**Tolerancia a Fallas**

El sistema implementa mecanismos automáticos de resiliencia para asegurar que la red P2P continúe funcionando, aunque alguno de sus componentes falle.

**a) Manejo de Fallas del Maestro**

Si el servidor Maestro deja de estar disponible, los peers pueden seguir operando gracias a la red de amigos.

Algoritmo Resilient File Search:

1. Intentar localizar archivo en el Maestro (/locate)
2. Si el Maestro no responde → consulta en peers amigos (primario → backup)
3. Si tampoco funciona → devolver error de “archivo no localizable”

CODIGO:

def resilient\_file\_search(filename):

try:

return maestro\_search(filename)

except MaestroUnavailableException:

print("Maestro no disponible, usando red P2P")

try:

return peer\_to\_peer\_search(filename)

except NetworkException:

print("Red P2P no disponible")

return []

**b) Reconexión Automática con Maestro**

Los peers intentan restablecer comunicación con el Maestro cada 30 segundos:

Algoritmo de Reconexión:

1. Verificar conectividad con Maestro
2. Si no está disponible → intentar reconexión
3. Si reconexión exitosa → sincronizar de nuevo el directorio de archivos

CODIGO:

def maintain\_maestro\_connection():

while system\_running:

if not maestro\_reachable():

attempt\_reconnection()

if reconnection\_successful():

sync\_file\_registry()

sleep(30)

**c) Escenarios de Falla Cubiertos**

Maestro no disponible:

* Los peers consultan su red de amigos.

Peer no responde:

* El cliente selecciona otro peer de la lista disponible.
* Transferencia falla en curso:
* Se reintenta automáticamente con un timeout configurable.

Registro desactualizado:

* Se puede forzar actualización manual con /refresh.

**5.6 Análisis de Complejidad**

**Complejidad de Operaciones:**

- Registro de peer: O(1) - inserción en diccionario

- Búsqueda via Maestro: O(1) - lookup directo en directorio

- Búsqueda via peers: O(k) donde k = número de peers amigos (típicamente k=2)

- Transferencia de archivo: O(n) donde n = tamaño del archivo

- Actualización de directorio: O(m) donde m = número de archivos del peer

**Escalabilidad:**

- Maestro: Maneja hasta ~1000 peers con hardware estándar

- Peers: Sin límite teórico, limitado por ancho de banda de red

- Red de amigos: Escala O(log n) con topología estructurada óptima

**6. Descripción del entorno de ejecución nativo**

**6.1 Stack Tecnológico Implementado**

Lenguaje Base: Python 3.8+

Gestión de dependencias: pip + requirements.txt

Entorno aislado: virtualenv (venv)

Frameworks y Librerías principales:

FastAPI (0.104+): Framework web para APIs REST

Uvicorn (0.24+): Servidor ASGI asíncrono

gRPC (1.74+): Comunicación binaria de alto rendimiento

Requests (2.31+): Cliente HTTP para microservicios REST

Pydantic (2.5+): Validación y serialización de datos

**6.2 Configuración Dinámica por Nodo**

**Archivo de configuración config.json por peer:**

{

"id": "P1",

"ip": "0.0.0.0",

"port": 9001,

"grpc\_port": 9002,

"shared\_dir": "shared",

"maestro\_url": "http://localhost:9000",

"friend\_peer\_primary": "http://localhost:9003",

"friend\_peer\_backup": "http://localhost:9005"

}

**Configuración del Servidor Maestro:**

{

"id": "Maestro",

"ip": "0.0.0.0",

"port": 9000

}

**6.3 Gestión de Procesos y Concurrencia**

Cada nodo ejecuta procesos paralelos para REST y gRPC.

def run\_peer(config\_path):

rest\_thread = Thread(target=start\_rest\_server, args=(config,))

grpc\_thread = Thread(target=start\_grpc\_server, args=(config,))

rest\_thread.start()

grpc\_thread.start()

rest\_thread.join()

grpc\_thread.join()

**Modelo de Concurrencia:**

* REST: Asíncrono (async/await con FastAPI + Uvicorn)
* gRPC: Multithread (ThreadPoolExecutor, hasta 10 workers)
* Cliente: Soporta múltiples conexiones paralelas

**Gestión de Estado (thread-safe):**

CONFIG = None

SHARED\_DIR = None

PEER\_BASE\_URL = None

MAESTRO\_URL = None

peers\_registry = {}

**6.4 Comandos de Ejecución**

**Ejecución Manual**

Maestro:

cd maestro

python -m uvicorn server\_maestro:app --host 0.0.0.0 --port 9000

**Peers:**

cd peer1

python run\_peer.py --config config.json

Cliente:

cd peer1

python client.py --action download --file archivo3.txt

**Ejecución Automatizada**

bash prueba\_completa.sh # Inicia todo el sistema

bash verificar\_sistema.sh # Verifica estado del sistema

**6.5 Gestión de Directorios y Archivos**

**Estructura de Directorios por Peer:**

peer1/

├── shared/ # Archivos compartidos

│ ├── archivo1.txt

│ └── downloaded\_\*

├── config.json # Configuración

├── server\_rest.py # Microservicio REST

├── server\_grpc.py # Microservicio gRPC

├── client.py # Cliente P2P

└── run\_peer.py # Launcher principal

**6.7 Requisitos de Sistema**

**Software Requerido:**

- Sistema Operativo: Linux/Windows/macOS

- Python 3.8 o superior

- pip (gestor de paquetes Python)

- curl (para pruebas HTTP opcionales)

**Puertos de Red Utilizados:**

- 9000: Servidor Maestro (REST)

- 9001, 9003, 9005: Servidores REST de peers

- 9002, 9004, 9006: Servidores gRPC de peers

**7. Pruebas y Análisis de Resultados**

**7.1 Metodología de Pruebas**

Se aplicaron pruebas a distintos niveles:

**a) Unitarias (microservicios):**

* Verificación de endpoints REST.
* Validación de servicios gRPC (Upload/Download).
* Pruebas de configuración y arranque.
* Manejo de errores controlados.

**b) Integración**:

* Registro automático de peers en Maestro.
* Descubrimiento de archivos (centralizado y distribuido).
* Transferencias gRPC end-to-end.
* Sincronización automática del directorio global.

**c) Concurrencia:**

* Transferencias simultáneas (5–10 concurrentes).
* Consultas paralelas al Maestro.
* Registro simultáneo de múltiples peers.

**d) Tolerancia a Fallas:**

* Falla del Maestro en ejecución.
* Desconexión de peer durante transferencia.
* Peers amigos inactivos.
* Recuperación automática de conexiones.

**7.2 Escenarios de Prueba Ejecutados**

**Escenario 1: Sistema básico funcional**

prueba\_completa.sh

**Escenario 2: Transferencias gRPC**

python client.py --action download --file archivo3.txt

python client.py --action upload --file test.txt --peer http://localhost:9003 --grpc-port 9004 --filepath local.txt

**Escenario 3: Tolerancia a Fallas**

pkill -f "server\_maestro"

curl "http://localhost:9005/locate?file=archivo1.txt"

**7.3 Análisis de Resultados**

**Fortalezas:**

* Alta disponibilidad (funciona sin Maestro).
* Escalabilidad horizontal.
* Consistencia automática del directorio.
* Robustez ante fallas.

**Limitaciones:**

* Maestro sin persistencia (solo RAM).
* Red de amigos estática (configurada manualmente).
* Sin autenticación de peers.
* Transferencias síncronas (sin streaming para archivos grandes).

**Cuellos de botella:**

* Maestro como punto único de falla inicial.
* ThreadPoolExecutor limitado a 10 transferencias.
* Serialización JSON pesada con muchos metadatos.
* Falta de notificaciones push (usa polling).

8**. Código Fuente en Repositorio y Video Demostración**

**8.1 Estructura del Repositorio**

P2P-Proyecto/

├── maestro/

│ ├── server\_maestro.py # Servidor central REST

│ └── config.json # Configuración del Maestro

├── peer1/, peer2/, peer3/ # Nodos P2P

│ ├── server\_rest.py # API REST del peer

│ ├── server\_grpc.py # Servidor gRPC

│ ├── client.py # Cliente CLI

│ ├── run\_peer.py # Launcher REST + gRPC

│ ├── config.json # Configuración del peer

│ └── shared/ # Archivos compartidos

├── common/

│ ├── grpc\_client.py # Cliente gRPC reutilizable

│ ├── files\_pb2.py # Clases generadas de Protocol Buffers

│ └── files\_pb2\_grpc.py # Servicios gRPC generados

├── proto/

│ └── files.proto # Definición de servicios gRPC

├── scripts/

│ ├── prueba\_completa.sh # Prueba automatizada

│ ├── verificar\_sistema.sh # Script de verificación

│ └── start\_component.sh # Lanzador de componentes

├── docs/

│ ├── README.md # Descripción del proyecto

│ └── README\_GUIA\_EJECUCION.md # Guía de instalación y uso

└── venv/ # Entorno virtual

**8.2 Comandos de Prueba**

**Registro y consulta REST:**

curl http://localhost:9000/peers | python -m json.tool

curl "http://localhost:9000/locate?file=archivo1.txt" | python -m json.tool

curl "http://localhost:9005/locate?file=archivo2.txt" | python -m json.tool

**Transferencias gRPC:**

python client.py --action download --file archivo1.txt

python client.py --action upload --file nuevo.txt --peer http://localhost:9003 --grpc-port 9004 --filepath local\_file.txt

**8.3 Video Demostración (10–15 minutos)**

Guion:

* Explicación breve de la arquitectura.
* Levantar Maestro + 3 peers.
* Registrar peers automáticamente.
* Búsqueda de archivo vía Maestro.
* Búsqueda distribuida vía peers amigos.
* Ejemplo de transferencia gRPC (upload + download).