Informe Técnico - Proyecto P2P: Comunicación entre procesos mediante REST y gRPC

Estudiante: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ ID: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# 1. Objetivo y Marco teórico breve

Objetivo: Diseñar e implementar una versión inicial de un sistema P2P que permita la compartición de índices de archivos entre peers usando microservicios. El sistema debe emplear API REST y gRPC para comunicación entre peers, soportar concurrencia en los microservicios y utilizar un archivo de configuración dinámico en el bootstrap de cada nodo.

Marco teórico: Los sistemas P2P (peer-to-peer) permiten a nodos con roles equivalentes compartir recursos sin depender exclusivamente de un servidor centralizado. Existen redes P2P estructuradas (por ejemplo, DHTs) y no estructuradas (servidor de directorio, flooding, etc.). En este proyecto se propone una red no estructurada con un peer maestro que actúa como Directorio/Localización para facilitar el descubrimiento de recursos; la transferencia real de datos se realiza directamente entre peers (modelo partial-centralized). Se emplean dos middlewares: REST (HTTP/JSON) para consultas y APIs públicas y gRPC (HTTP/2, protobuf) para operaciones de baja latencia y transferencia simulada (ECO/DUMMY).

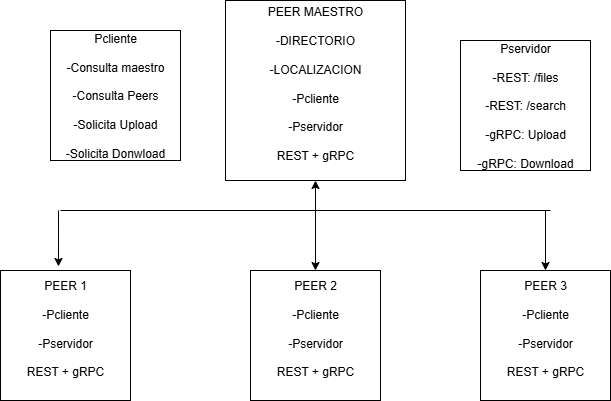
# 2. Descripción del servicio y problema abordado

Descripción: El servicio resuelve el problema de descubrimiento y consulta de archivos distribuidos entre varios peers. Cada peer expone microservicios PServidor (REST + gRPC) que: 1) listan archivos en un directorio configurado, 2) responden consultas de localización, 3) proveen endpoints ECO/DUMMY para upload y download simulados, y 4) soportan concurrencia. Además, cada peer incluye un módulo PCliente para consultar otros peers y registrarse ante un peer maestro (Directorio). El objetivo es demostrar consultas distribuidas, concurrencia y comunicación híbrida REST/gRPC sin necesidad de transferir realmente los archivos (solo index/URIs y ECOs).

# 3. Arquitectura del sistema y diagramas

Arquitectura propuesta:  
- Peer Maestro: mantiene un directorio de peers y localizaciones; expone REST para consultas de directorio y gRPC para señales rápidas.  
- Peers (P1..PN): cada peer ejecuta varios microservicios (PServidor) y un cliente (PCliente). Los microservicios se dividen por responsabilidad: localización, listado de archivos, ECO upload/download.  
- Comunicación: solicitudes de descubrimiento y metadatos via REST; operaciones ECO/DUMMY via gRPC y también REST para compatibilidad.  
  
Flujos principales:  
1) Bootstrap: un peer arranca, lee su archivo de configuración y contacta a su peer amigo (o al Peer Maestro) para registrarse.  
2) Consulta de recursos: un cliente en P2 solicita a Peer Maestro (o a un peer conocido) la ubicación del archivo. Maestro devuelve URI del peer propietario.  
3) Transferencia simulada: P2 contacta directamente a P3 (peer propietario) para invocar el ECO/DUMMY de download o upload (gRPC/REST).

Diagrama de componentes (diseño del usuario):



# 4. Especificación de protocolos y APIs

- REST API (HTTP/JSON):  
 \* /register (POST): registra peer en Peer Maestro. Payload: {peer\_id, ip, port\_rest, port\_grpc, directory\_listing\_url}  
 \* /peers (GET): obtiene listado de peers registrados en Maestro.  
 \* /files (GET): lista de archivos públicos en un peer (expone en cada peer PServidor).  
 \* /search?name=<file> (GET): consulta distribuida por nombre (puede invocar al Maestro o a peers conocidos).  
  
- gRPC (Protobuf):  
 Services:  
 \* DirectoryService: RegisterPeer(RegisterRequest) -> RegisterResponse; GetPeers(GetPeersRequest) -> GetPeersResponse  
 \* FileService: ListFiles(ListFilesRequest) -> ListFilesResponse  
 \* TransferService: DummyUpload(UploadRequest) -> UploadResponse; DummyDownload(DownloadRequest) -> DownloadResponse  
 Mensajes incluyen metadatos de archivo (name, size, uri) y status codes.

# 5. Algoritmos de particionamiento y distribución

Para la versión inicial se propone un esquema simple de Directorio centralizado (Peer Maestro) que indexa qué peer tiene qué archivos. Alternativas (opciones de mejora):  
- Flooding/Query: las consultas se propagan a un conjunto limitado de peers (TTL) para búsquedas sin Maestro.  
- DHT (Chord/Kademlia): para una versión estructurada y escalable, usar hashing consistente para distribuir la responsabilidad de índices.  
  
Estrategias implementadas:  
- Indexación local: cada peer genera su propio índice leyendo el directorio configurado en el bootstrap y lo expone via API.  
- Registro en Maestro: al arrancar, peer envía su índice mínimo (lista de nombres + URIs) o solo metadatos básicos para no cargar la red.  
- Consistencia eventual: las actualizaciones son notificadas al Maestro; en caso de fallo, otros peers pueden continuar con información ligeramente desactualizada.

# 6. Descripción del entorno de ejecución nativo o en Docker

Entorno recomendado:  
- Desarrollo local: ejecutar cada peer en contenedores Docker individuales (imagen basada en Python/Node/Go según la implementación elegida). Cada contenedor expone dos puertos: REST (HTTP/JSON) y gRPC (HTTP/2). El directorio compartido se monta como volumen en el contenedor.  
- Producción/Pruebas distribuidas: desplegar cada peer como VM o contenedor en AWS Academy. Usar Docker Compose para pruebas locales con 3-4 peers y un Maestro.  
  
Archivo de configuración (ejemplo YAML/JSON por peer):  
{  
 "peer\_id": "P1",  
 "ip": "0.0.0.0",  
 "port\_rest": 5000,  
 "port\_grpc": 50051,  
 "directory": "/data/shared",  
 "peer\_friend\_primary": "http://10.0.0.5:5000",  
 "peer\_friend\_secondary": "http://10.0.0.6:5000"  
}

# 7. Pruebas y Análisis de resultados

Plan de pruebas (recomendado):  
1) Pruebas unitarias de cada microservicio (endpoints REST y stubs gRPC).  
2) Prueba de integración local con Docker Compose: levantar 3 peers + Maestro; validar registro, consulta y ECO/DUMMY.  
3) Pruebas de concurrencia: realizar múltiples requests simultáneos hacia el endpoint /files y TransferService.  
4) Pruebas de tolerancia a falla: apagar uno de los peers y verificar que consultas sigan funcionando con otros peers y que el Maestro actualiza su directorio.  
  
Métricas a recolectar:  
- Latencia en consultas (ms)  
- Tiempo de respuesta de gRPC para ECO/DUMMY  
- Número de consultas exitosas vs fallidas  
  
Resultados esperados (ejemplo simulado):  
- Registro de 4 peers en Maestro en < 2s promedio.  
- Listado de archivos entregado en < 150 ms por REST en condiciones locales.  
- gRPC ECO/DUMMY responde < 80 ms en el mismo host.

# 8. Plan de desarrollo (victorias tempranas → final)

Sprint 0 (Setup) - 1 día:  
- Definir stack (p. ej. Python Flask + gRPC with grpcio OR Node Express + @grpc/grpc-js)  
- Crear repositorio y esqueleto de proyecto.  
  
Sprint 1 (MVP) - 3 días:  
- Implementar microservicio de listado de archivos (REST) y configuración Bootstrap.  
- Implementar registro básico al Peer Maestro (/register).  
  
Sprint 2 - 4 días:  
- Implementar gRPC TransferService con DUMMY upload/download.  
- Soporte de concurrencia (threading/async) y pruebas locales.  
  
Sprint 3 - 3 días:  
- Despliegue con Docker Compose, ejecutar 3 peers + Maestro.  
- Recolección de logs y pruebas de carga.  
  
Entrega final - Documentación y video demostración (2 días).

# 9. Código fuente en repositorio y video demostración

Estructura sugerida del repositorio GitHub:  
- /maestro: código del Peer Maestro (REST + gRPC)  
- /peer\_template: plantilla para peers (Dockerfile, entrypoint, config.sample.json)  
- /docs: diagramas, informe técnico (este Word/PDF), guías de despliegue  
- /scripts: docker-compose.yml, scripts de bootstrap y pruebas  
  
Video demostración (10-15 min):  
- Mostrar arquitectura y configuraciones.  
- Levantar 3 peers + Maestro y demostrar: registro, búsqueda de archivo y transferencia simulada.

# Anexos: ejemplos de payloads y comandos de prueba

Ejemplo: Register (REST POST /register)  
{ "peer\_id":"P2", "ip":"10.0.0.7", "port\_rest":5002, "port\_grpc":50052, "directory":["fileA.txt","fileB.pdf"] }  
  
cURL ejemplo para listar archivos (REST):  
curl http://localhost:5002/files  
  
gRPC: ejemplo de llamar DummyDownload via cliente stub:  
client.DummyDownload(DownloadRequest(filename='fileA.txt'))

# Notas finales y recomendaciones

Recomendaciones:  
- Mantener separación clara entre microservicios para facilitar pruebas independientes.  
- Usar TLS para gRPC en despliegues reales.  
- Implementar mecanismo de reintento/backoff al contactar peers y al actualizar el Maestro.  
- Registrar logs (structured logging) y métricas (Prometheus) para evaluación en AWS.  
  
Con esto se cumple el entregable 1 (Informe técnico). Para completar la entrega recomendamos seguir los puntos descritos y anexar el código fuente y video conforme a lo solicitado en el enunciado.