Curso
Herramientas de Computación en la Nube

Clase 5

Febrero 14 de 2026

RUSTFS

Almacenamiento de Objetos tipo S3

Profesor

Alvaro Mauricio Montenegro Díaz, Ph.D.
Universidad de la Sabana

Febrero 2026


RUSTFS
ALMACENAMIENTO DE OBJETOS TIPO S3

1. Introducción General

En una implementación privada, RustFS puede ocupar el rol del almacenamiento S3 mostrado en este diagrama, actuando como backend compatible con la API sin depender de la infraestructura de AWS.

RustFS es un sistema de almacenamiento de objetos distribuido, de alto rendimiento, desarrollado en el lenguaje Rust.

Se publica bajo licencia **Apache 2.0**, es de código abierto y totalmente compatible con la API de Amazon S3.

RustFS combina:

- Seguridad de memoria sin garbage collector
- Concurrencia segura
- Rendimiento cercano a C/C++
- Arquitectura descentralizada
- Diseño cloud-native

Se posiciona en el mismo espacio arquitectónico que:

- MinIO
- Ceph
- Amazon S3 (a nivel de API)

Su objetivo es ofrecer una alternativa técnicamente sólida, empresarial y moderna para almacenamiento de objetos en nubes privadas, híbridas o entornos distribuidos.

El siguiente es un diagrama conceptual macro que ubica a RustFS dentro del ecosistema completo de datos

Compute
↓
Orquestación (K8s)
↓
RustFS (Object Storage)
↓
Infraestructura física

2. ¿Qué es RustFS en términos arquitectónicos?

RustFS implementa **Object Storage**.

No es:

- Un sistema de archivos POSIX tradicional
- Un motor de base de datos SQL
- Un NAS convencional

Es:

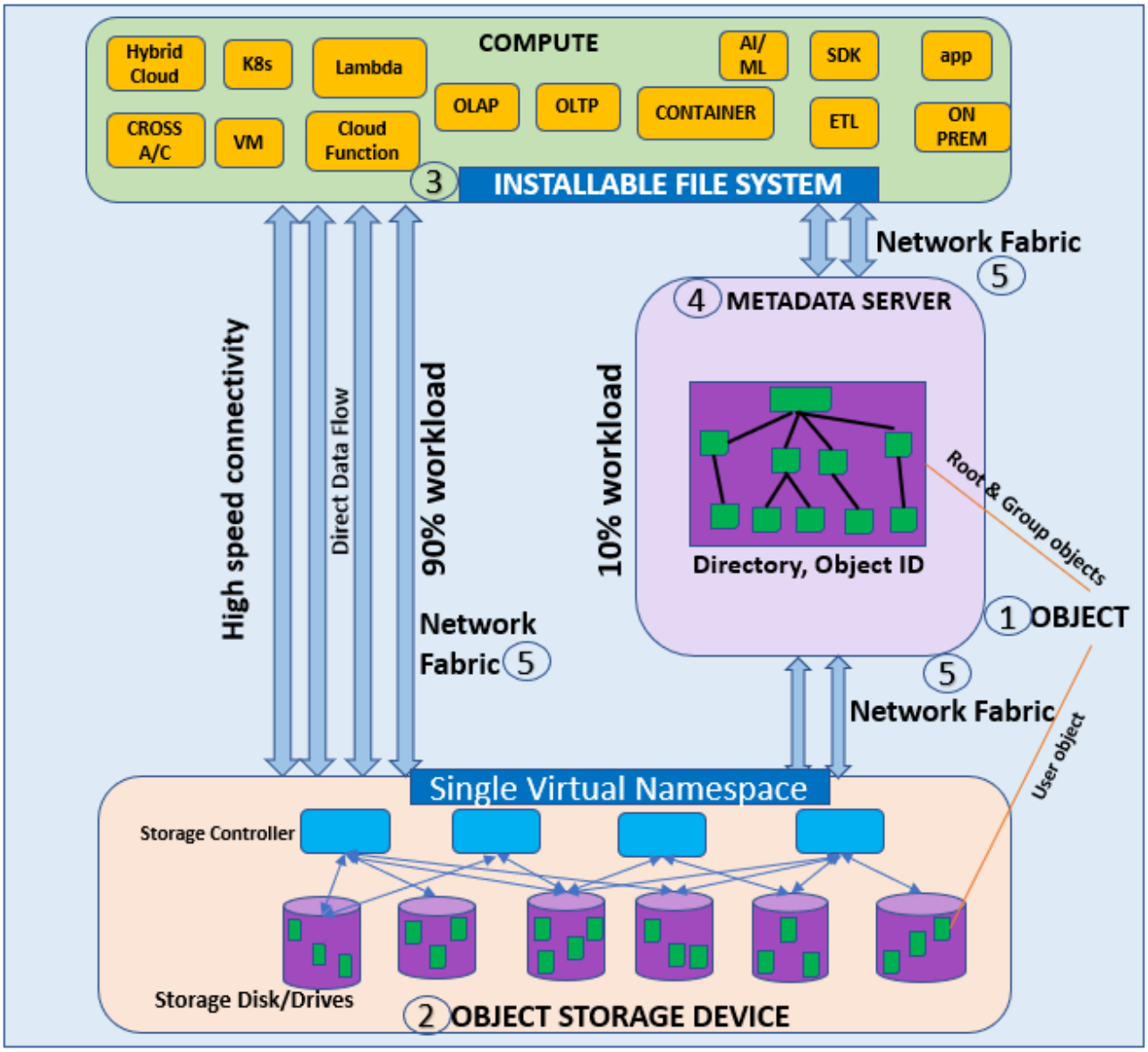
- Un sistema basado en objetos inmutables
- Organizado en buckets
- Accesible vía HTTP(S)
- Diseñado para escala horizontal

En una arquitectura cloud moderna se ubica en la capa:

```
Aplicaciones
      ↓
Servicios de Datos
      ↓
Object Storage  ← RustFS
      ↓
Discos / NVMe / Cluster
```

Es la capa fundamental de persistencia para sistemas de datos, IA, backups y plataformas distribuidas.

RustFS se ubica en la capa de Object Storage dentro de una arquitectura híbrida moderna. El siguiente diagrama ilustra el lugar estructural donde opera: como backend de persistencia para cargas OLTP, OLAP, ML y microservicios.



Arquitectura de un sistema hibrido moderno en la nube

3. Características Técnicas Principales

Compatibilidad S3

Total interoperabilidad con el ecosistema S3: big data, data lakes, backups, procesamiento multimedia y pipelines analíticos.

Arquitectura distribuida

Escala horizontal mediante múltiples nodos interconectados.

Alto rendimiento

Benchmarks reportan:

- Lectura: hasta 323 GB/s
- Escritura: hasta 183 GB/s

El rendimiento depende de red y almacenamiento subyacente (NVMe, SSD, etc.).

Seguridad por diseño

Gracias a Rust:

- No hay use-after-free
- No data races
- Seguridad de memoria garantizada en tiempo de compilación

Multiplataforma

- Linux
- macOS
- Windows

Cloud-native

Optimizado para:

- Docker
- Kubernetes
- Infraestructura contenerizada

Extensible

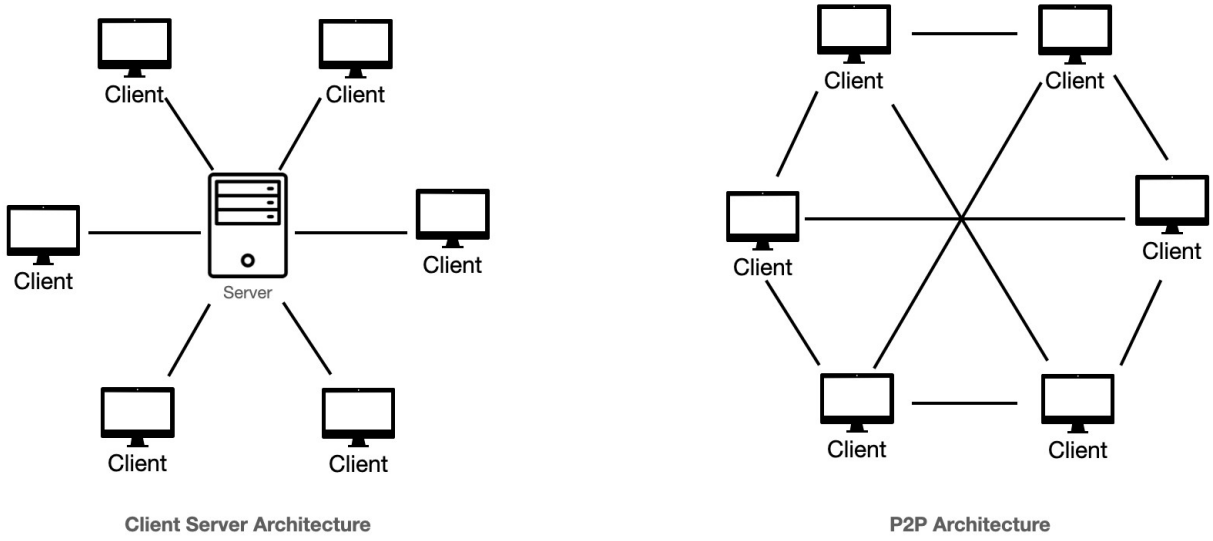
Arquitectura de plugins y personalización profunda.

4. Arquitectura de RustFS

4.1. Arquitectura Peer to Peer (P2P)

En **RustFS**, “peer-to-peer” significa que **todos los nodos del clúster son equivalentes** (“peers”): cada nodo puede **recibir solicitudes S3**, **participar en lectura/escritura**, **mantener la lógica de**

distribución, y **cooperar** para almacenar/recuperar objetos. No existe un **nodo maestro** permanente ni una separación rígida tipo *metadata node vs data node*. La coordinación se hace **de manera distribuida**.



Arquitectura P2P

Esta comparación ilustra cómo RustFS elimina el nodo central de coordinación, distribuyendo la responsabilidad entre todos los peers.

4.2.

4.3. Diferencia con arquitecturas “clásicas” (master/metadata/data)

En sistemas distribuidos tradicionales es común ver:

- **Nodo maestro (master/manager)**: coordina, asigna trabajo, controla el “estado del clúster”.
- **Nodos de metadatos**: guardan el mapa “objeto → ubicación”.
- **Nodos de datos**: guardan bytes.

Problemas típicos:

- **SPOF** (single point of failure) si el master no está altamente redundado.
- **Hotspot de metadatos** (cuello de botella) si muchas operaciones requieren consultar/actualizar metadatos centralizados.
- **Complejidad operativa**: más roles, más componentes, más dependencias.

En **P2P**, el objetivo es que:

- No haya “un cerebro” único.
- La carga se reparta.
- El clúster degrade de forma más suave ante fallos.

4.4. ¿Cómo opera un clúster P2P cuando llega una petición S3?

Pensemos en un `PUT Object` (subir un objeto):

1. **Cualquier nodo puede ser “front door”.** El cliente (Python/boto3, aws cli, etc.) le pega a cualquier endpoint del clúster (o a un balanceador que reparte).
2. **Selección de destino (placement).** El nodo que recibe la petición decide **en qué “set/conjunto”** cae el objeto (según la forma en que RustFS particiona el clúster).
    - Regla clave: ***un objeto se almacena dentro de un único set***.
3. **Distribución de fragmentos o réplicas.** Dependiendo del esquema (replicación o erasure coding), ese nodo:
    - Divide el objeto en partes (chunks) o
    - Crea réplicas
    - Y coordina el envío a múltiples discos/nodos dentro del set.
4. **Commit distribuido (confirmación).** El sistema confirma la operación cuando se cumple la condición de durabilidad requerida:
    - “Se escribieron suficientes fragmentos”
    - “Se alcanzó quórum”
    - “Se escribió en N réplicas”
5. **Read-after-write**

    Tras confirmar, cualquier lectura posterior debe ver ese objeto como disponible.

En un `GET Object`:

- El nodo que recibe el request identifica el set donde vive el objeto y recupera los fragmentos necesarios para reconstruirlo (si hay erasure coding) o lee la réplica adecuada.

4.5. ¿Dónde viven los metadatos en un diseño P2P?

En P2P no desaparecen los metadatos; lo que cambia es **cómo se gestionan**:

- En lugar de un “servidor de metadatos” central, los metadatos suelen estar:
    - **distribuidos** (por ejemplo, con información local por set)
    - **replicados** (para evitar pérdida)
    - y diseñados para que la operación sea “autocontenida” dentro del set que aloja el objeto.

En RustFS, la idea central (según su documentación de arquitectura) es evitar un componente único crítico para el mapeo de ubicación, reduciendo el riesgo de que la “pérdida de metadatos” comprometa la integridad o recuperabilidad del dato.

4.6. Ventajas prácticas de P2P en almacenamiento de objetos

a) Menos puntos únicos de fallo (SPOF). Si un nodo cae, el clúster sigue atendiendo desde otros nodos.

b) Escalamiento horizontal más natural. Añadir nodos incrementa:

- capacidad total
- throughput agregado
- potencial de paralelismo I/O

c) Menos cuellos de botella de control-plane. Al no depender de un master central para cada operación, se reduce el riesgo de “atascar” la coordinación.

d)Operación más simple (idealmente). Menos roles diferentes. Menos “piezas” obligatorias.

4.7. Trade-offs reales (lo que P2P te exige)

P2P no es magia; cambia el tipo de problemas:

a) Consenso / coordinación distribuida. Aunque no haya master, igual necesitas coordinación:

- membresía del clúster (quién está vivo)
- reintentos
- tolerancia a particiones de red

b) **Consistencia y quórums.** Garantías como read-after-write requieren reglas claras:

- cuándo confirmas escritura
- cómo resuelves lecturas si un nodo está caído
- cómo evitas “lecturas viejas” durante fallos

c) **Rebalanceo y expansión.** Cuando se agregan discos/nodos, debes decidir:

- ¿muevo objetos viejos (rebalanceo) o solo ubico nuevos objetos en el nuevo set?
- ¿cómo mantengo uniformidad en la distribución?

d) **Debugging más difícil.** Los bugs distribuidos son más complejos: timeouts, particiones, estados intermedios.

4.8. P2P aplicado a los “sets” (conjuntos) de RustFS

- **Unidad** = disco físico
- **Set** = grupo fijo de unidades (distribuidas en distintos nodos)
- **Objeto** = vive dentro de un set
- **Clúster** = se divide en varios sets

Esto es una forma práctica de P2P:

- Cada set funciona como un “microdominio” donde el dato y su protección (replicas/EC) se resuelven localmente.
- Distribuir unidades de un set entre nodos reduce el riesgo de perder un objeto por caída de un solo nodo.

4.9. ¿Qué significa para integración cloud

Clientes: no necesitan saber P2P; solo ven un endpoint S3.

Balanceador: puede repartir a cualquier nodo.

Kubernetes:

- replicas/pods de RustFS como peers
- service (ClusterIP/LoadBalancer/Ingress) como punto de entrada
- health checks para sacar peers caídos

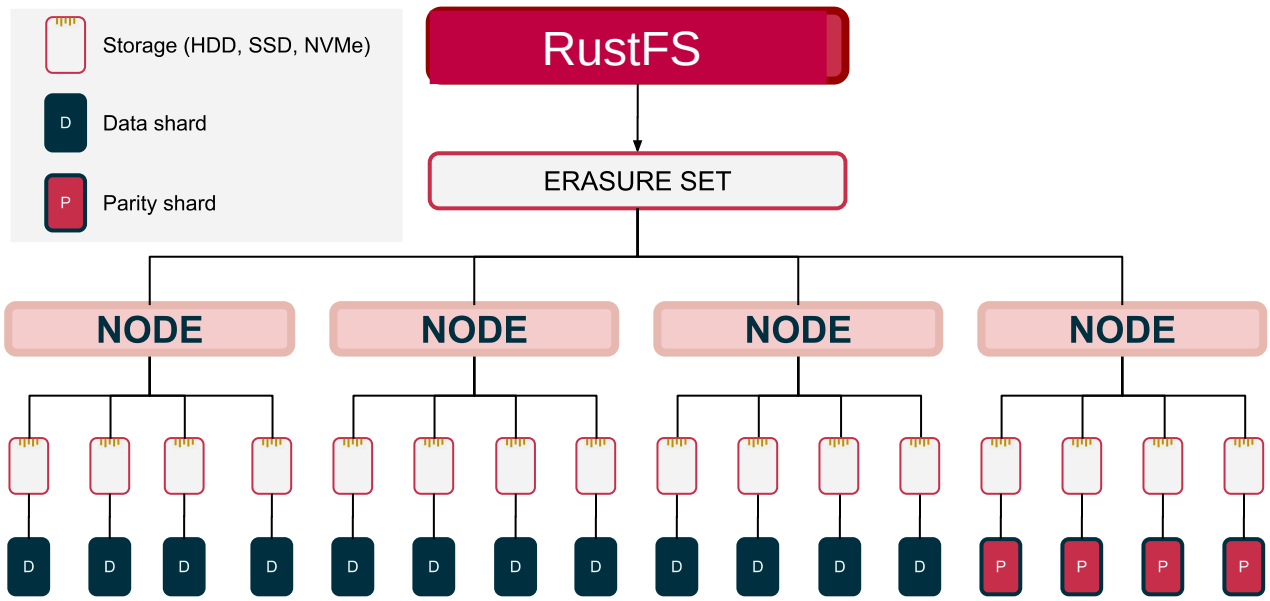
El P2P te permite:

- escalar por replicas/nodos
- tolerar fallas sin “líder único” rígido

RustFS adopta una arquitectura **peer-to-peer descentralizada**:

- No existen nodos maestros
- No existen nodos exclusivos de metadatos
- Todos los nodos son iguales

Esto elimina puntos únicos de fallo y simplifica la operación.



Arquitectura de un cluster en RustFS

4.10. Mini-ejemplo mental

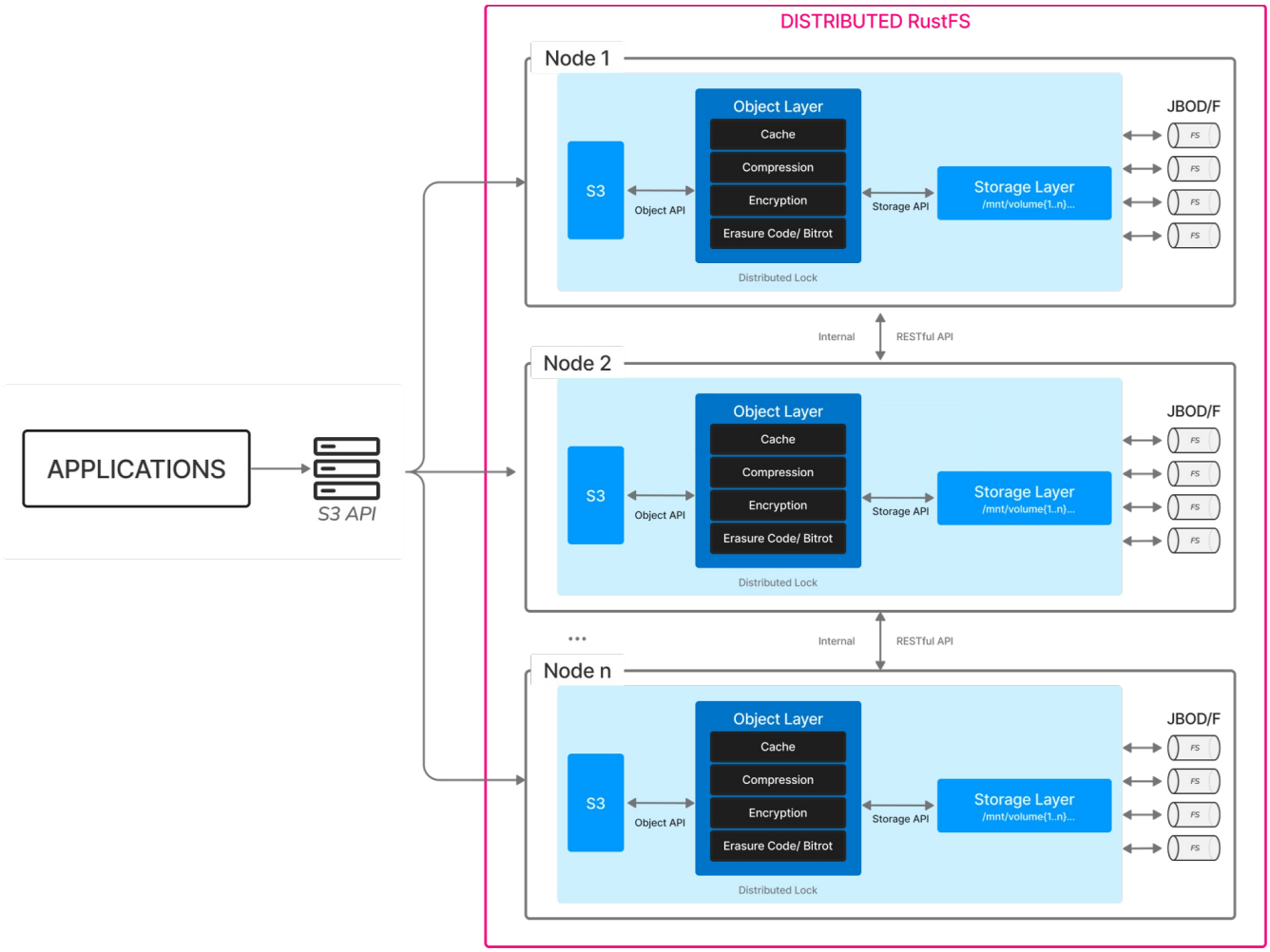
Imagina 8 nodos, cada uno con 4 discos. El clúster define sets de 16 discos (distribuidos en varios nodos).

- Subes `video.mp4`
- El nodo A recibe el PUT
- Decide que `video.mp4` va al set #3
- Divide el archivo en fragmentos + paridad

- Escribe en varios discos de distintos nodos
- Confirma cuando tiene suficientes fragmentos para garantizar durabilidad

Luego, un GET que llegue al nodo F puede reconstruirlo leyendo fragmentos del set Eso es P2P: **cualquier peer recibe**, **el peer coordina**, **los peers cooperan**, **no dependes de un master único**.

4.11. Componentes estructurales



Arquitectura RustFS

1. API Gateway S3-compatible
2. Motor de almacenamiento
3. Gestión de metadatos distribuida
4. Mecanismo de replicación o erasure coding
5. Sistema de autenticación y control de acceso

6. Cifrado en tránsito y en reposo

5. Modelo de Consistencia

RustFS implementa **read-after-write consistency**:

Después de una escritura exitosa, cualquier lectura posterior verá inmediatamente la versión actualizada del objeto.

Esto aplica tanto en modo single-node como distribuido.

Perfecto. Voy a redactarte una sección **con nivel de curso de arquitectura de sistemas distribuidos**, lista para insertar en el documento, y luego te indico exactamente dónde ubicarla.

6. RustFS y el Teorema CAP

En sistemas distribuidos, el **Teorema CAP**, formulado por Eric Brewer, establece que un sistema no puede garantizar simultáneamente:

- **C — Consistencia (Consistency).** Todas las lecturas reflejan la escritura más reciente.
- **A — Disponibilidad (Availability).** Cada solicitud recibe una respuesta (aunque no necesariamente la más reciente).
- **P — Tolerancia a particiones (Partition tolerance).** El sistema continúa operando incluso si hay fallas de comunicación entre nodos.

En entornos reales distribuidos (como RustFS), la **tolerancia a particiones es obligatoria**, ya que las redes pueden fallar. Por lo tanto, el diseño debe elegir entre priorizar consistencia o disponibilidad durante una partición.

6.1. Posicionamiento de RustFS frente a CAP

RustFS implementa:

- Consistencia fuerte (read-after-write consistency)
- Tolerancia a fallos mediante replicación o erasure coding
- Confirmación de escritura basada en quórum

Esto implica que, ante una partición severa de red:

- Puede limitar temporalmente la disponibilidad

- Pero evita lecturas inconsistentes o estados divergentes

En términos prácticos:

RustFS prioriza **consistencia + tolerancia a particiones (CP)** sobre disponibilidad irrestricta.

Esta decisión es coherente con sistemas de almacenamiento empresarial donde:

- La pérdida de consistencia es inaceptable
- La durabilidad y coherencia del dato son críticas
- Se requiere previsibilidad operativa

6.2. Implicaciones arquitectónicas

1. Las escrituras requieren confirmación de suficientes nodos (quórum).
2. Durante fallos de red, algunos nodos pueden rechazar operaciones para preservar consistencia.
3. El sistema evita “split-brain” mediante reglas de coordinación distribuida.

Para arquitectos cloud, esto significa que RustFS es apropiado para:

- Sistemas financieros
- Pipelines de datos críticos
- Infraestructura de auditoría
- Almacenamiento de modelos de IA

donde la coherencia del dato es prioritaria sobre la disponibilidad absoluta.

7. Seguridad y Cumplimiento en RustFS

El almacenamiento de objetos en entornos empresariales no es solamente un problema de rendimiento, sino de **seguridad, aislamiento y cumplimiento normativo**.

RustFS aborda la seguridad en múltiples capas:

7.1. Seguridad por construcción (Rust)

El uso del lenguaje Rust proporciona:

- Eliminación de *use-after-free*
- Eliminación de *data races*
- Garantías de memoria segura en tiempo de compilación

- Reducción significativa de vulnerabilidades clásicas

Esto reduce la superficie de ataque en la infraestructura base.

7.2. Seguridad en tránsito

- Cifrado mediante TLS
- Protección contra interceptación
- Integración con certificados empresariales

Todo tráfico S3 puede operar sobre HTTPS.

7.3. Seguridad en reposo

RustFS puede operar sobre:

- Discos cifrados
- Infraestructura con cifrado a nivel de volumen
- Políticas de protección de datos sensibles

En arquitecturas empresariales se recomienda:

- Cifrado AES a nivel de objeto o volumen
- Rotación de claves
- Gestión segura de credenciales

7.4. Control de acceso (IAM-like)

Al ser compatible con la API S3, RustFS puede implementar:

- Políticas de acceso por bucket
- Control granular por usuario
- Restricciones por prefijo
- Roles y permisos

Esto permite:

- Multi-tenant seguro
- Aislamiento por proyecto
- Segmentación organizacional

7.5. Versionado y protección contra borrado

En entornos empresariales es crítico evitar pérdida accidental o maliciosa de datos.

Mecanismos recomendados:

- Versionado de objetos
- Object locking (modo WORM)
- Políticas de retención
- Auditoría de accesos

Esto es especialmente relevante para:

- Cumplimiento regulatorio
- Evidencia digital
- Entornos financieros o legales

7.6. Seguridad en arquitectura P2P

El diseño peer-to-peer implica:

- No existe un nodo maestro comprometible
- No hay un único punto de ataque estructural
- La caída de un nodo no compromete el sistema completo

La descentralización mejora resiliencia frente a ataques dirigidos.

8. Conceptos Clave

Objeto

Unidad fundamental de almacenamiento: archivo, flujo de bytes o dato no estructurado.

Bucket

Contenedor lógico de objetos. Aislamiento por namespace.

Unidad

Disco físico que almacena datos.

Conjunto (Set)

Grupo de unidades donde se almacena un objeto. Un objeto vive en un único conjunto.

Principios de diseño:

- Un clúster se divide en múltiples conjuntos.
- Las unidades de un conjunto deben distribuirse entre nodos distintos.
- El sistema calcula automáticamente la configuración según escala.

9. Arquitectura Interna Avanzada

Característica	Impacto Arquitectónico
Rust (memory safety)	Infraestructura más robusta
Async runtime	Alto throughput
Zero-cost abstractions	Eficiencia comparable a C/C++
S3 API compatible	Integración inmediata
Descentralización	Eliminación de SPOF
Diseño minimalista	Operación simplificada

RustFS se inspira en la filosofía arquitectónica de MinIO: simplicidad, rendimiento y confiabilidad.

10. Lugar en una Arquitectura Cloud Moderna

RustFS actúa como backend de almacenamiento para:

- Data Lakes
- Sistemas de IA
- Entrenamiento de modelos
- Backups empresariales
- ETL pipelines
- Microservicios
- Plataformas serverless

Ejemplo:

```
Frontend (Next.js)
Backend (FastAPI)
Workers (Python)
        ↓
     RustFS
        ↓
Cluster de discos
```

En Kubernetes:

```
Kubernetes
├── API Services
├── ETL
├── ML Training
├── Vector DB
└── RustFS Cluster
```

11. Interfaces de Uso

11.1. API HTTP (S3)

Ejemplo conceptual:

```
PUT /bucket/archivo.parquet
GET /bucket/archivo.parquet
```

Compatible con SDK AWS y librerías S3.

11.2. CLI

Puede utilizarse:

- aws cli
- mc (MinIO Client)
- Clientes S3

Ejemplo:

```
aws s3 cp archivo.csv s3://mi-bucket/
```

11.3. Uso desde Python

Compatible mediante:

- boto3
- s3fs
- aiobotocore

Ejemplo conceptual:

```
import boto3

s3 = boto3.client(
    "s3",
    endpoint_url="http://localhost:9000",
    aws_access_key_id="key",
    aws_secret_access_key="secret"
```

```
)
s3.upload_file("datos.parquet", "mi-bucket", "datos.parquet")
```

Crítico para:

- Pandas
- Apache Spark
- Airflow
- Pipelines ML
- Entrenamiento distribuido

11.4. Interfaz Web

Dependiendo de la implementación:

- Gestión de buckets
- Gestión de usuarios
- Políticas de acceso
- Visualización de objetos

12. Integración en Ecosistemas Cloud

Kubernetes

- StatefulSets
- Persistent Volumes
- Helm
- Operadores

Seguridad

- OIDC
- JWT
- OAuth2
- TLS

Data Platforms

- Apache Spark
- Trino
- DuckDB
- Airbyte
- Airflow

AI / ML

- Datasets
- Model artifacts
- Checkpoints
- Versionado

13. Comparación Estratégica

Sistema	Lenguaje	Enfoque
MinIO	Go	Object storage ligero
Ceph	C++	Infraestructura masiva
Amazon S3	Propietario	Cloud pública
RustFS	Rust	Cloud-native seguro

RustFS apunta a entornos donde:

- La seguridad es crítica
- El rendimiento es prioritario
- Se requiere arquitectura descentralizada
- Se busca reducir superficie de vulnerabilidad

14. Rol en Arquitectura Lakehouse

Modelo:

```
Raw → Bronze → Silver → Gold
```

RustFS puede almacenar:

- Raw (JSON, CSV)
- Bronze (Parquet)
- Silver (Parquet refinado)
- Artefactos ML
- Evidencias auditables
- Logs estructurados

Es ideal para pipelines modernos de datos.

15. Casos de Uso Avanzados

- Data Lake distribuido
- Backend multi-tenant
- Infraestructura de backup inmutable
- Edge storage
- Plataformas de auditoría
- Sistemas de IA a gran escala

16. Implicaciones Arquitectónicas Estratégicas

RustFS representa una tendencia en infraestructura moderna: sistemas distribuidos minimalistas, seguros por construcción y diseñados para operar en entornos cloud-native.

En comparación con arquitecturas complejas basadas en nodos maestros y planos de metadatos centralizados, RustFS reduce la superficie de ataque, simplifica el despliegue y mejora la previsibilidad operacional.

Para arquitectos de sistemas, la decisión de adoptar RustFS no es solamente técnica sino estratégica: implica optar por descentralización, menor complejidad operativa y mayor alineación con principios de resiliencia distribuida.

17. Resumen Ejecutivo

Perfecto. Entonces lo cerramos **sobre RustFS específicamente**, no sobre S3 en abstracto.

Aquí tienes una versión más precisa, con foco en Rust, P2P y arquitectura distribuida:

18. Síntesis Conceptual

RustFS no es simplemente “otro almacenamiento S3-compatible”. Es una expresión arquitectónica de una tendencia más profunda en infraestructura moderna:

- descentralización,
- simplicidad estructural,
- seguridad por construcción,
- y alineación con principios cloud-native.

No existe un sistema de almacenamiento universalmente superior. Existen sistemas coherentes con un modelo mental de arquitectura.

RustFS es óptimo cuando:

- la durabilidad es crítica,
- la consistencia no puede negociarse,
- el sistema debe escalar horizontalmente,
- se busca eliminar puntos únicos de fallo,
- la seguridad de memoria es estratégica,
- y la complejidad operativa debe mantenerse bajo control.

Su diseño peer-to-peer elimina la dependencia de nodos maestros. Su compatibilidad S3 permite integración inmediata con el ecosistema de datos. Su implementación en Rust reduce clases completas de vulnerabilidades estructurales.

La decisión arquitectónica clave no es únicamente “usar RustFS”, sino comprender:

- qué modelo de consistencia se adopta,
- qué trade-offs frente al teorema CAP se asumen,
- cómo se distribuye la responsabilidad entre almacenamiento y cómputo,
- y qué nivel de resiliencia se requiere.

RustFS representa una arquitectura que privilegia coherencia y previsibilidad sobre complejidad innecesaria.

19. Cierre de la lección

RustFS no debe entenderse solo como un producto open-source, sino como un caso de estudio de arquitectura distribuida moderna. A través de RustFS hemos visto:

- cómo opera un sistema peer-to-peer real,
- cómo se implementa consistencia fuerte en almacenamiento distribuido,
- cómo se diseñan conjuntos y erasure sets,
- cómo se posiciona el almacenamiento en una arquitectura cloud,
- y cómo la seguridad puede incorporarse desde el lenguaje mismo.

Comprender RustFS es comprender cómo se construye infraestructura distribuida robusta en la actualidad.

Quien entiende este modelo puede evaluar, diseñar y criticar arquitecturas cloud con criterio técnico real.

20. Diccionario Técnico Avanzado

Perfecto.
Lo rehacemos correctamente. Profesional. Jerárquico. Académico.
Este sí es un **Diccionario Técnico Avanzado estructurado**, organizado por dominios conceptuales, no solo alfabético.

Ubicación recomendada en tu documento:

15. Diccionario Técnico Avanzado – Arquitectura RustFS

---

21. Diccionario Técnico Avanzado

Arquitectura Distribuida y RustFS

A. Fundamentos de Sistemas Distribuidos

CAP Theorem

Principio formulado por Eric Brewer que establece que un sistema distribuido no puede garantizar simultáneamente Consistencia (C), Disponibilidad (A) y Tolerancia a particiones (P). En entornos reales, la tolerancia a particiones es obligatoria, por lo que el diseño implica un trade-off entre consistencia y disponibilidad.

Consistencia Fuerte (Strong Consistency)

Modelo donde todas las lecturas reflejan la escritura más reciente confirmada.

Read-after-Write Consistency

Garantía específica donde una lectura posterior a una escritura exitosa devuelve inmediatamente el valor actualizado.

Disponibilidad (Availability)

Capacidad del sistema de responder a solicitudes incluso ante fallos parciales.

Tolerancia a Particiones (Partition Tolerance)

Capacidad del sistema de seguir operando cuando existen fallos de comunicación entre nodos.

Quórum

Número mínimo de nodos o fragmentos necesarios para confirmar una operación como válida o exitosa.

Split-Brain

Situación en la que dos subconjuntos de un sistema distribuido creen ser la instancia principal debido a una partición de red.

Durabilidad

Probabilidad estadística de que un dato no se pierda a lo largo del tiempo, incluso ante fallos múltiples.

Throughput

Cantidad de datos procesados por unidad de tiempo.

Latencia

Tiempo que tarda una operación en completarse.

B. Modelo de Almacenamiento de Objetos

Object Storage

Modelo de almacenamiento donde los datos se organizan como objetos independientes identificados por clave, no como bloques ni archivos jerárquicos tradicionales.

Objeto

Unidad fundamental compuesta por:

- datos binarios,
- metadatos,
- identificador único (key).

Bucket

Contenedor lógico que agrupa objetos dentro de un namespace.

Namespace

Espacio lógico de nombres que organiza buckets y objetos.

Inmutabilidad

Propiedad por la cual un objeto no se modifica; cualquier cambio genera una nueva versión.

Versionado (Versioning)

Mecanismo que permite mantener múltiples versiones de un objeto.

Object Lock (WORM)

Modelo “Write Once Read Many” que impide modificar o eliminar un objeto durante un período definido.

C. Arquitectura Distribuida de RustFS

Peer-to-Peer (P2P)

Arquitectura descentralizada donde todos los nodos son equivalentes y no existe un nodo maestro permanente.

Nodo

Instancia del sistema que participa en el clúster distribuido.

Clúster

Conjunto de nodos cooperando para ofrecer almacenamiento distribuido.

Unidad

Disco físico que almacena fragmentos de objetos.

Set (Conjunto)

Grupo fijo de unidades donde se almacenan los fragmentos de un objeto.

Erasure Coding (EC)

Técnica que divide un objeto en k fragmentos de datos y m fragmentos de paridad, permitiendo reconstrucción ante fallos.

Erasure Set

Conjunto específico de discos donde se aplica el esquema de erasure coding para un objeto.

Replicación

Estrategia de protección que almacena copias completas del objeto en múltiples nodos.

Fragmentación

Proceso de dividir un objeto en múltiples partes para distribución.

Placement Strategy

Algoritmo que determina en qué set o nodos se almacenará un objeto.

Rebalanceo

Proceso de redistribución de datos cuando se agregan o eliminan nodos/discos.

Cluster Membership

Mecanismo que define qué nodos están activos y participan en el clúster.

D. Arquitectura Interna y Operación

Control Plane

Capa encargada de la coordinación del estado del sistema, membresía y decisiones de distribución.

Data Plane

Ruta por donde fluyen los datos reales (lecturas y escrituras de objetos).

API S3

Interfaz HTTP estándar compatible con Amazon S3 para operaciones sobre buckets y objetos.

PUT Object

Operación S3 que carga un objeto en un bucket.

GET Object

Operación S3 que recupera un objeto almacenado.

Sharding

División lógica de datos para distribuir carga.

Load Balancer

Componente que distribuye solicitudes entre múltiples nodos del clúster.

Multi-Tenant

Arquitectura que permite aislar múltiples clientes o proyectos dentro del mismo sistema.E. Seguridad y Cumplimiento

TLS (Transport Layer Security)

Protocolo criptográfico que protege datos en tránsito.

Cifrado en Reposo

Protección criptográfica de datos almacenados en disco.

IAM (Identity and Access Management)

Sistema de control de acceso basado en usuarios, roles y políticas.

Políticas de Acceso

Reglas que definen qué operaciones puede realizar un usuario sobre un bucket u objeto.

Superficie de Ataque

Conjunto de puntos potenciales de vulnerabilidad en un sistema.

F. Conceptos Estratégicos

Cloud-Native

Diseñado para operar en entornos contenerizados y orquestados (ej. Kubernetes).

Descentralización

Arquitectura que elimina puntos únicos de fallo estructural.

SPOF (Single Point of Failure)

Componente cuya falla detiene el sistema completo.

Trade-off Arquitectónico

Decisión consciente de priorizar ciertas propiedades del sistema sobre otras.

22. Bibliografía

- RustFS Documentation – Architecture. https://docs.rustfs.com/concepts/architecture.html
- Amazon S3 API Reference. https://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html
- Kleppmann, M. (2017). *Designing Data-Intensive Applications*.
- Burns, B., Beda, J., & Hightower, K. (2019). *Kubernetes: Up and Running*.
- Sam Newman (2021). *Building Microservices*.
- Brewer, E. “CAP Twelve Years Later”
- Ghemawat et al. (Google File System)

- Amazon Dynamo Paper
- MinIO architecture docs (comparativa)
- Rust Book (para el componente memory safety)