TECNOLÓGICA DEL ORIENTE
INSTITUCIÓN DE EDUCACIÓN SUPERIOR

SISTEMAS DISTRIBUTIVOS

COMPARACIÓN ENTRE RPC Y REST

COMPARACIÓN ENTRE RPC Y REST

En los sistemas distribuidos modernos, uno de los aspectos fundamentales para garantizar la interoperabilidad entre componentes es la forma en que estos se comunican. Dos de los modelos más utilizados para este propósito son Remote Procedure Call (RPC) y Representational State Transfer (REST). Ambos permiten la invocación de funcionalidades a través de redes, pero se basan en principios arquitectónicos profundamente distintos (Muñoz Escoí, 2013).

Mientras que RPC busca emular la llamada a una función remota como si fuera local, REST promueve un modelo basado en recursos y operaciones estandarizadas mediante HTTP. Comprender sus diferencias estructurales, filosóficas y prácticas es esencial para elegir la solución más adecuada en un entorno distribuido.

Fundamentos de cada modelo

RPC (Remote Procedure Call):

- Modelo centrado en procedimientos.
- El cliente invoca una función en el servidor (por nombre y argumentos).
- Utiliza diferentes protocolos de transporte (como TCP, HTTP, HTTP/2).
- Permite una abstracción fuerte: el código cliente “cree” que llama a una función local.
- Puede ser implementado mediante XML-RPC, JSON-RPC, gRPC, entre otros.

REST (Representational State Transfer):

- Modelo centrado en recursos.
- La interacción se realiza mediante operaciones estándar (GET, POST, PUT, DELETE).
- Utiliza HTTP como base de comunicación.
- Requiere que el cliente conozca la estructura de la API y los recursos disponibles.
- Representa datos típicamente en formatos JSON o XML.

Tabla 1. Comparación detallada por criterios claves

Criterio	RPC	REST
Modelo.	Orientado a procedimientos o métodos.	Orientado a recursos y representaciones.
Transporte.	Puede usar HTTP, TCP, HTTP/2 (especialmente en gRPC).	Utiliza HTTP estándar.
Verbos utilizados.	Nombres de funciones definidos por el desarrollador.	Verbos HTTP (GET, POST, PUT, DELETE).
Formato de datos.	Personalizado: Protobuf (gRPC), XML (XML-RPC), JSON (JSON-RPC).	JSON, XML, YAML, etc.
Acoplamiento.	Más fuerte entre cliente y servidor.	Menor acoplamiento.

Criterio	RPC	REST
Contratos/Interfaces.	Definidos formalmente en .proto (gRPC) o WSDL (SOAP).	Generalmente documentado (OpenAPI, Swagger).
Semántica de errores.	Personalizada o específica del protocolo.	Basada en códigos de estado HTTP (200, 404, 500...).
Estado del servidor.	Generalmente sin estado (stateless), pero puede ser con estado.	Requiere ser sin estado (stateless).
Complejidad.	Mayor en configuración inicial, menor en consumo.	Menor complejidad inicial, uso más explícito.
Rendimiento	Alta eficiencia (especialmente gRPC con HTTP/2).	Eficiente, pero no tan optimizado como gRPC en grandes volúmenes.
Ideal para.	Comunicación entre microservicios, alta velocidad, streaming.	APIs públicas, CRUD sobre recursos, simplicidad.

Ejemplos

Ejemplo con RPC (gRPC):

Servicio definido con Protocol Buffers:

Figura 1. Protocol Buffers

```
// archivo: calculadora.proto
syntax = "proto3";

service Calculadora {
  rpc Sumar (Operacion) returns (Resultado);
}

message Operacion {
  int32 a = 1;
  int32 b = 2;
}

message Resultado {
  int32 valor = 1;
}
```

Llamada desde el cliente:

Figura 2. Cliente

```
respuesta = stub.Sumar(Operacion(a=10, b=20))
print(respuesta.valor)  # Resultado: 30
```

La llamada se comporta como una función local, pero la operación ocurre en el servidor.

Ejemplo equivalente en REST:

URL del recurso: POST /api/calculadora/sumar.

Cuerpo de la solicitud (JSON):

Figura 3. Body POST sumar

```
{
  "a": 10,
  "b": 20
}
```

Respuesta (JSON):

Figura 4. Respuesta Json

```
{
  "valor": 30
}
```

El cliente debe conocer la URL, el método HTTP y el formato del mensaje.

Elección entre RPC y REST, según el contexto

La elección entre RPC y REST, no debe basarse en preferencias personales, sino en el contexto técnico y arquitectónico. Algunos escenarios típicos:

- **REST es ideal para:**
  - Aplicaciones CRUD simples.
  - APIs abiertas o públicas.
  - Integraciones con terceros y desarrollo web tradicional.
  - Ecosistemas donde los clientes cambian con frecuencia.
- **RPC (especialmente gRPC) es ideal para:**
  - Comunicación entre microservicios con alto rendimiento.
  - Servicios internos que requieren streaming bidireccional.
  - Arquitecturas fuertemente tipadas y controladas.
  - Escenarios donde se prioriza eficiencia y velocidad.

Convergencias y coexistencia

En muchos entornos modernos, REST y RPC coexisten en la misma arquitectura. Por ejemplo:

- Una API REST pública para aplicaciones móviles y web.
- Una capa interna de microservicios comunicándose vía gRPC para mayor eficiencia.
- Una puerta de enlace (API Gateway) que traduce entre ambos modelos.

Esta coexistencia demuestra que no se trata de enfoques mutuamente excluyentes, sino de herramientas complementarias que deben emplearse con criterio técnico.