<u>Lectura Certamen:</u> Optimizando la comunicación entre microservicios: Thrift, gRPC y HTTP/2

Protocolos de comunicación entre componentes de software. Invocar métodos de forma remota en una maquina -> definir protocolo de comunicaciones.

Nunca se habla de un protocolo concreto, son stacks que incluyen base de comunicación de red (TCP, UDP,…) hasta formato de los mensajes que se transfieren (binario, cifrado, textual…).

El mas popular actualmente JSON sobre REST sobre HTTP.

El foco sera uno mas fácil de leer que XML, mas racional en los métodos de protocolos que HTTP y cifrado.

No nos enfocaremos en requisitos de rendimientos críticos:

HTTP solo admite una petición por conexión.
JSON se codifica como texto (UTF-8). Tamaño 10 veces mayor que en binario (texto plano muy comprimible)

Actualmente (era de microservicios en plataformas de virtualización) una sola petición a un servicio puede ramificarse y extenderse a cierta cantidad de servicios auxiliares.

Se buscan desarrollar metodologías, componentes y protocolos que minimicen la latencia inherente a un Stack de microservicios distribuidos.

Alternativas:
1. Apache Thrift
2. gRPC
3. JSON sobre HTTP/2

Spoiler masivo: Thrift seguramente no nos interese salvo en el caso de aplicaciones legacy que ya lo estén usando; gRPC nos servirá cuando el rendimiento sea crítico, y JSON sobre HTTP/2 será nuestra opción menú del día para proyectos que, siendo más o menos complejos, no nos exijan exprimir hasta el último microsegundo de latencia.

Aspectos del stack a analizar antes de adopción:
1. Lenguajes soportados
2. Versionado/compatibilidad
3. Costes de adaptación formación e implementación
4. Costes de migración
5. Herramientas/documentación
6. Integración con frameworks existentes

1. Apache Thrift

Nació dentro de Facebook.
OpenSource descentralizado
Gestión basada en meritocracia
Desarrollo de código prevalente sobre documentación del mismo.
Stack completo de RPC (remote procedure call)
Lenguaje IDL (interface definition language)
Stubs de comunicación

Eficiencia: Ofrece varios formatos para los mensajes (binario y comprimido). Varios formatos de transporte (socket, frames…) Mayor que gRPC en cuanto a eficiencia. No ofrece solución para cachear respuestas idempotentes. Se puede usar elástico packetbeat (monitorización de carga). No esta documentado el balanceo de carga, hay que usar herramientas externas.

Lenguajes soportados: C, Common Lisp, C++, C#, D, Delphi, Erlang, Go, Haxe, Haskell, Java, JavaScript, node.js, OCaml, Perl, PHP, Python, Ruby y SmallTalk.

Documentación: Escasa y desorganizada.

Versionado y compatibilidad con evolutivos: compatibilidad se rompe en numero y tipo de parámetros, tipos de retorno con menos datos

Costes de adaptación y formación: Aprender IDL. Familiarizarse con APIs.

Costes de implementación: Como usa protocolo binario -> mas difícil depurar peticiones (no hay postran para thrift). Más simple que SOAP.

Costes de migración: si queremos usarlo en proyecto existente, los servicios y sus clientes deben ser implementados de nuevo.

Costes de mantenimiento: incremento de complejidad pero no tanta.

Herramientas de integración con frameworks existentes: plugin de macen para compilar esquemas. En Istio está soportado en vía el proxy de servicios Envoy. En herramientas de prueba, hay un CLI para probar llamadas.

2. gRPC
Origen en Google.
Se ha hecho OpenSource por partes.
Sigue siendo gestionado por google.
Se compone de un IDL
Formato binario de mensajes serializados
Protocol buffers que son mensajes que IDL gestionara para realizar las invocaciones remotas
Usa protocolo HTTP/2 para transporte.

Eficiencia: Mensajes se transmiten en formato ad hoc protocolé buffer en binario y comprimidos. Tamaño pequeño. Protocolo estándar HTTP/2 para transporte de mensajes. Se pueden cachear respuestas a peticiones repetidas en el cliente. Monitorear carga con interceptores. Balancea carga con muchas opciones. Cliente puede balancear carga por si mismo o hacerlo externamente (Envoy, Ribbon, Linkerd, o Istio)

Lenguajes soportados: compilador es ejecutable en C++. Java (6+, Android API 14+), Python, Objective-C, C++, Go, Ruby,JavaScript, PHP, Dart y C#. Soporta mensajes en formato Thrift.

Documentación: Documentación organizada.

Versionado y compatibilidad con evolutivos: compatibilidad se rompe en numero y tipo de parámetros, tipos de retorno con menos datos

Costes de adaptación y formación: Aprender IDL. Familiarizarse con APIs.

Costes de implementación: mas fluido comparado con API RESTful. Pero hay problema de depurar peticiones cuando están formateadas en binario comprimido.

Costes de migración: si queremos usarlo en proyecto existente hay que implementar de nuevo capa de transporte de servicios que queremos optimizar. Tambien adaptar codigo de data transfer objects.

Costes de mantenimiento: incremento de complejidad pero no tanta.

Herramientas de integración con frameworks existentes: Plugin maven y gradle. Plugin de análisis estatico de codigo. Transcodifcación de JSON a buffer (mantener API con endoints JSON y protobuff simultáneos). Grpc gateaway (plugin para generar endpoints RESTful automaticamente. Istio y service mesh. Spring.

3. REST sobre HTTP/2
Thrift y gRPC son frameworks RPC end to end. Usar JSON REST sobre HTTP/2 es opción atractiva (proporciona optimización sin tener que tocar código).
Desarrollar o actualizar aplicación usando practicas ya conocidas para generar servicios REST Protocolo de transporte HTTP/2.

Eficiencia: HTTP/2 convierte mensajes a binario comprimido, reduciendo el tamaño del JSON antes de viajar por la red. Soporta mantener un socket abierto con el servidor que enluta todos los mensajes minimizando el overhead de negociación HTTP. El server push. Se puede anticipar la siguiente petición y enviar respuesta antes (no es gratis). Monitoreo, cacheo y balanceo de carga ya están resueltas ene servidores HTTP.

Lenguajes soportados: mismo que con HTTP/1.1

Documentación: No hay un unico punto de referencia para obtener documentación. Pero hay mucho

Versionado y compatibilidad con evolutivos:

Costes de adaptación y formación: Pequeños retoques de código necesarios y de configurar servidor de apps. Coste menor a los dos anteriores. Si el servicio es nuevo el obverhead de los costes es minimo.

Costes de implementación: mismo que con HTTP/1.1

Costes de migración: se requiere servidor compatible con HTTP/2. Problema si cliente esta usando versión antigua que no podemos actualizar. Adaptar app JSON-REST a HTTP/2 es coste contenido en cuanto a codigo, la infraestructura puede necesitar actualizaciones.

Costes de mantenimiento: mismo que con HTTP/1.1

Herramientas de integración con frameworks existentes: postran no soporta HTTP/2.

Conclusion:
Usamos RPC completo o el tercer caso. En el primer caso habrá optimización máxima a coste medio y alto. En el segundo buena optimización a coste mínimo.

gRPC ventaja sobre Thrift.

Lectura Certamen: Amazon prime video migro de microservicios a arquitectura monolitica reduciendo costos

Videos a clientes de forma concurrente. Crearon herramienta para monitorear cada stream de. Video de los clientes. Identificar automaticamente calidad de stream y problemas como corrupción de datos o desincronizacion audio video. Repara problemas.

Antes tenia herramienta que veia calidad de video pero nunca se penso que fuese usado a gran escala. Notaron que habían cuello de botella, no se podían monitorear miles al mismo tiempo. Decidieron analizar arquitectura para evitar eso.

Antes tenia AWS step functions y lo mas costoso era control de flujo e información enviada a servicios. Movieron todos los componentes a unico proceso y mantener info de transferencia en memoria, simplificando logica de control de servicios. Amazon EC2 y ECS. Elastic compute cloud y elastic container service.