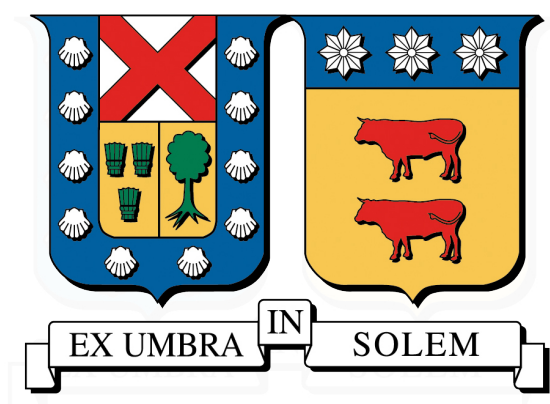


UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
VALPARAÍSO - CHILE



**ANÁLISIS Y EVALUACIÓN
DE LA TRANSFERENCIA DE MENSAJES
EN ARQUITECTURAS DE MICROSERVICIOS**

BRIAN ALEJANDRO VIDAL CASTILLO

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERÍA DE EJECUCIÓN EN INFORMÁTICA

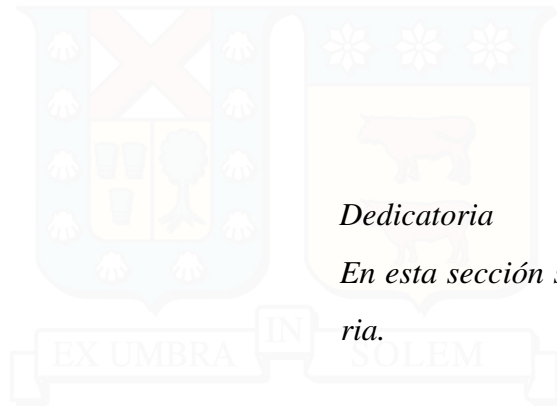
PROFESOR GUÍA : RAÚL MONGE ANDWANTER
PROFESOR CORREFERENTE : HERNÁN ASTUDILLO ROJAS

NOVIEMBRE 2016

Agradecimientos

En esta sección escribiré los agradecimientos.





Dedicatoria

En esta sección se incluirá la dedicatoria.

RESUMEN EJECUTIVO

Las Arquitecturas de Microservicios proponen la separación de grandes sistemas en pequeños componentes de responsabilidad reducida que poseen su propio almacén de datos, una interfaz de comunicación bien definida y son desarrollados por equipos pequeños e independientes. Un sistema de mediana envergadura suele poseer decenas de microservicios, los que requieren un entorno computacional adaptable a la carga del sistema y estrategias de comunicación entre ellos. Cada mensaje que viaja de un microservicio a otro debe ser serializado en el origen, transportado a través de la red, ruteado y deserializado en destino. La elección de tecnologías y estrategias para realizar dichas operaciones es vital para el perfecto desempeño del sistema y de los programadores, cuyos criterios dependen de la naturaleza del problema. Sin embargo, la estabilización del mercado de herramientas permite generar una matriz que ayude a los arquitectos de software a escoger las herramientas que han probado ser aptas en la industria y alejarse de aquellas combinaciones que han sido insuficientes o sobre complejizadas. La correcta elección puede acelerar el desarrollo de software, optimizar recursos humanos y mejorar la comunicación entre equipos, además de permitir la introducción de nuevos programadores de manera controlada, aislada y fomentar la innovación dentro de las empresas.

Palabras Claves: Arquitecturas de Microservicios, serialización binaria, protocolos binarios, desarrollo de software basado en mensajes

ABSTRACT

Micro-services Architectures advocates the separation (shift from monoliths to smaller units...) of big systems to tiny components with single responsibility, isolated databases, a single communication interface; usually developed by small and independent teams. A medium-sized system usually consists of dozens microservices, which require a computationally adaptable environment and communication strategies between them. Each message that travels from one microservice to another must be serialized on origin, transported through the network, routed and deserialized on destination. The choice of technologies and strategies to fulfill those operations is vital to the optimal performance of the system and the programmers. The criteria to choose depends of the nature of the solution. Nevertheless the stabilization of tools allows the construction of an architectural matrix that can be used by software architects to choose the tools that have been proven on the industry and stay away from the combinations that have been insufficient or overly complex. The right choice can accelerate software development, optimize human resources and improve communication between teams, it also allows the introduction of new programmers in a controlled and isolated way, fostering innovation inside the organization.

Keywords. Micro-services Architectures, binary serialization, binary protocols, message-driven development

GLOSARIO

Canal de mensajes: Es el medio por el cual un productor de mensajes y un consumidor se comunican

Mensaje: Unidad básica en la construcción de arquitecturas basadas en mensajes? microservicios? No son necesarios los microservicios, pero sí el paso de mensajes. Microservicios sólo dramatizan esta situación. Ahí la utilidad. Un mensaje es una estructura de datos definida, sin comportamiento propio que es intercambiado entre componentes de un sistema.

Microservicio: Componente de un sistema. Posee una única responsabilidad y almacén propio de datos. Toda comunicación es a través de una interfaz bien definida.

Arquitectura de microservicios: Patrón de arquitectura de software donde los sistemas son construidos por numerosos componentes responsables de una única tarea en lugar de un gran monolito con múltiples responsabilidades.

Programación asíncrona: Capacidad de un sistema de entregar un contenedor para un dato que eventualmente estará disponible

Reactividad: Capacidad de un sistema para responder ante la eventual disponibilidad de un dato

Arquitecturas basadas en mensajes: Patrones de arquitectura de software donde la comunicación es a través del paso de estructuras de datos conocidas tanto por el origen y el destino.

Serialización: Proceso reversible por el cual una estructura de datos es transformada a una representación transportable por la red

Deserialización: Proceso por el cual una estructura de datos es transformada desde su representación a su estado original

Protocolos de transporte: Conjunto de reglas por las cuales una estructura de datos es negociada, entregada o rechazada (ELABORAR)

The difference being that messages are directed, events are not — a message has a clear addressable recipient while an event just happen for others (0-N) to observe it.

Message oriented applications by default come with removal of shared memory. Publisher and subscriber don't need to share a memory space. On the other hand, all the other features (i.e. order, method name coupling and such) are not necessities.



Índice de Contenidos

0.0.1. Contexto:Motivo	XI
1. Capítulo 1	1
1.1. Definición del Problema	1
1.2. Objetivos	3
1.2.1. Objetivo General:	3
1.2.2. Objetivos Específicos:	4
1.3. Retrospectiva Histórica	4
1.3.1. Patrones	4
1.3.2. Arquitecturas	5
2. Capítulo 2	6
2.1. Marco Teórico	6
2.2. Protocolos de Transferencia de Datos	6
2.2.1. Protocolos Basados en Texto	8
2.2.1.1. STOMP	8
2.2.1.2. HTTP 1.1	9
2.2.1.3. ZMTP	10
2.2.2. Protocolos Binarios	11
2.2.2.1. AMQP	12
2.2.2.2. MQTT	14
2.2.2.3. HTTP2	15
2.2.2.4. CoAP	16
2.2.2.5. Apache Thrift	23
2.3. Serialización	23
2.3.1. Serialización Basada en Texto	23
2.3.1.1. XML	24
2.3.1.2. JSON	24
2.3.1.3. YAML	24
2.3.2. Serialización Binaria	24
2.3.2.1. Protocol Buffers	25
2.3.2.2. Avro	25
2.3.2.3. Thrift	25
2.4. Patrones de Transferencia de Datos Entre Procesos	25
2.4.1. Transferencia Síncrona	25
2.4.1.1. Petición-Respuesta	25

2.4.2. Transferencia Asíncrona	25
2.4.2.1. Uno a uno	25
2.4.2.2. Uno a muchos	25
3. Capítulo 3	26
3.1. Diseño y Construcción de las Pruebas	26
3.2. Metodología	26
3.2.1. Procedimiento	27
3.2.2. Instrumentos	27
3.2.2.1. Software	27
3.2.3. Estrategia de análisis	28
3.2.4.	28
3.3. Métricas	28
4. Capítulo 4	29
4.1. Mediciones	29
5. Capítulo 5	30
5.1. Validación y Análisis de los Resultados	30
5.2. Conclusiones	30
Bibliografía	32
A. ANEXO 1	33

Índice de Tablas



Índice de Figuras



INTRODUCCIÓN

0.0.1. Contexto:Motivo

Justo después del resumen inicia la introducción, en la cual el tesista ubica al lector en el contexto mundial, nacional, así como señala la importancia del tema que se ha seleccionado en el área de su disciplina y así mismo guía al lector hacia la estructura de la tesis narrando lo que encontrará en casa uno de los capítulos del documento.

La rama de la Informática que se dedica al diseño e implementación de sistemas distribuidos ha tenido una década sin tregua.

Desde la masificación de los navegadores web y los dispositivos móviles, se ha trabajado incesablemente por entregar mejores herramientas a quienes implementan servicios complejos, distribuidos en distintas regiones geográficas; sistemas que deben mantenerse en coordinación y coherencia. La práctica más común, hace un par de años, era desarrollar enormes sistemas con grandes almacenes de datos y alojarlos en costosa infraestructura.

Latinoamérica no puede darse esos lujos [1].

La revolución cultural informática que ha traído el uso de contenedores se ha reflejado en la forma de construir aplicaciones distribuidas; hoy ya no diseñamos monolitos y hemos adoptamos la cultura de construir un gran número de servicios informáticos con tareas muy específicas que, en conjunto, entregan un gran nivel de resiliencia, elasticidad y por sobre todo, claridad mental a quienes los desarrollan.

Sin embargo, tal cantidad de servicios, cuyo número siempre crece (tal como nos indica la experiencia) se ven enfrentados a una disyuntiva; por un lado requieren de un fuerte contrato acerca de sus responsabilidades y las tareas que son capaces de recibir (nuestros mensajes) y por otro lado demandan un entorno que les permita iterar rápidamente sin acoplarlos demasiado al sistema como un todo.

Muchas de las necesidades de estos pequeños pero numerosos servicios han sido satisfechas por gigantes tecnológicos que saben de primera fuente lo complejo que resulta orquestarlos, distribuirlos, mantenerlos operativos, permitirles fallar elegantemente y más importante aún: mantener una tasa creciente de innovación. Sin embargo, muy poco se ha

estudiado acerca de la forma y la lengua en que se comunican.

La principal barrera en la implementación de soluciones ha sido hasta ahora la enorme diversidad de herramientas, los lenguajes de programación y especificidad en cada solución. Este trabajo busca transformar esta debilidad en una oportunidad para los arquitectos de software [2].

Hoy en día el método más utilizado en la comunicación inter-sistemas es a través de texto plano, a nivel de transporte vía HTTP o a través de protocolos binarios que transportan estructuras codificadas en... texto plano, lo que ha dado a luz a diversas técnicas de programación defensiva y, sin ir más lejos, pérdida de productividad en los desarrolladores que deben preocuparse por la “forma” que traen los datos que necesitan en lugar de simplemente utilizarlos [3].

Y es allí donde radica la oportunidad de abrazar la diversidad de lenguajes de programación y protocolos de transporte, utilizando datos estructurados o semi estructurados en un dialecto neutral (a los lenguajes de programación).

A pesar de no ser el tema de estudio en esta memoria, las diferencias en rendimiento computacional (considerando cada tecnología por separado) son abismantes [4] [4.1].

Entonces, ¿Por qué seguimos comunicando sistemas a través de texto? ¿No existen acaso convenciones que nos ayuden a diseñar tales sistemas? Y más importante aún ¿Por qué estamos usando el camino más lento y endeble para construir sistemas igual de endebles?

En el desarrollo de esta memoria propongo desarrollar argumentos que motiven a los arquitectos de software a utilizar protocolos binarios para transportar estructuras de datos bien definidas, versionables, además de utilizar herramientas de generación de código para automatizar estas tareas, así como también formas de integrar estos procesos en la cadena de producción de software, para potenciar la integración continua.

A la vez, esta memoria pretende ser la antesala al fenómeno del Internet de las Cosas en Latinoamérica, cuyo núcleo es la comunicación entre pequeños sistemas [5].

1 | Capítulo 1

FOCO: Estudiar cómo se comportan los protocolos de transferencia actuales en el escalado de microservicios cuando se agregan servicios nuevos y, a la vez, aumentan las peticiones.

<http://blog.wix.engineering/2015/07/14/building-a-scalable-and-resilient-architecture/>

1.1. Definición del Problema

La infraestructura necesaria para el funcionamiento de internet es demasiado costosa en términos económicos, ambientales, climáticos, **MÁS ASPECTOS**

My First Law of Distributed Object Design: Don't distribute your objects.

- Martin Fowler, *Patterns of Enterprise Application Architecture*

Los sistemas que se mantienen relevantes se ven enfrentados a complejidad en la modularización de sus componentes, falta de resiliencia, ineficiencias en recursos humanos y un alto costo de planificación y desarrollo.

A lo largo de los años, diversas patrones de programación y arquitecturas de software han demostrado lo complejo que es crear sistemas distribuidos confiables. Aún más, distribuir un sistema que inicialmente fue creado como una única entidad genera nuevas necesidades del sistema que antes no existían (COMPLEJIDAD ASOCIADA).

Los protocolos de internet están basados principalmente en TCP, a pesar de existir mecanismos más ágiles y bien adoptados como UDP o exóticos como OSI **REFERENCIA**. Una de las razones para este panorama es la comprobación de entrega de paquetes de datos

que ofrece TCP y la posibilidad de utilizar tecnologías fundacionales existentes para crear tecnologías más sofisticadas. Luego, es deseable generar arquitecturas de software cuyo principal mecanismo de transporte sea TCP.

Es deseable también un protocolo que esté mejor destinado a un patrón como REST (Representational State Transfer, Transferencia de Estado Representacional) que HTTP como sucede en la actualidad, donde los clientes de un sistema no poseen acceso a los eventos que notifiquen los cambios de estado en estas entidades. Es rescatable el amplio uso del patrón, lo que ha entregado a la comunidad un conjunto mínimo de normas sobre las cuales actuar ante entidades de datos, resultando de librerías informáticas de gran calidad, gran participación de individuos en proyectos de código abierto y curvas de aprendizaje menos sinuosas y menos elevadas.

Éticamente, es correcto utilizar protocolos que sean entendibles por humanos, de tal forma que la información en tránsito pueda ser auditada por la ciudadanía, en caso de ser necesario. Sin embargo, el flujo de información, una vez ésta ingresa en un sistema, está libre de tales restricciones, dando cabida al uso holgado de transportes y codificaciones binarios, cuando suponen una ventaja competitiva o mejora en rendimiento.

Y si consideramos recursos humanos, hay varios aspectos relevantes que quedan al descubierto; por un lado, las posibilidades de hardware empujan a los lenguajes y librerías informáticas a introducir elementos primitivos que permitan la programación paralela, para utilizar los núcleos disponibles, al mismo tiempo que permitan la construcción de sistemas distribuidos. Por otro lado, están los equipos de desarrollo, que históricamente han sido organismos en crecimiento constante, encargados cada vez de sistemas más difíciles de manejar. Todo esto conlleva complejidad y altos costos.

Relevancia del problema

Consolidar las arquitecturas de microservicios y crear un par de combinaciones eficientes, sólidas y SIMPLES de transporte y serialización.

Por qué este problema es relevante para su contexto local e institucional?

Evaluar contemporáneamente un paisaje tan cambiante en esta década, como lo es la

programación distribuida, es muy relevante cuando Internet es sólo la antesala de el Internet de las cosas, que ya posee implementaciones para nichos muy particulares (como el Internet de los Automóviles) o movimientos altamente políticos y sociales, como son las redes en malla de libre acceso (como el caso de Hyperbórea).

- Requerimientos de arquitecturas modernas
 - Larga vida del software
 - Priman atributos de calidad (no-funcionales)
 - Retrasar diseño (hasta que los problemas existan)
 - Cambios!
 - Modularidad para automatización (build, test, deploy)
 - Reflejar la estructura organizacional
- ¿Qué arquitecturas se ven afectadas?
- ¿Cuál es la influencia del panorama actual de Internet? (navegadores, HTTP, APIs)
- ¿Cuáles son los desafíos que trae IoT en este campo? (tostadores con conectividad)
- ¿Qué otros transportes han sido efectivos en solucionar el problema?
- ¿Cuáles son los puntos más dolorosos para la industria hoy?
- ¿Cuáles serán mañana?

1.2. Objetivos

1.2.1. Objetivo General:

Evaluar el comportamiento de los principales protocolos de transferencia de mensajes **Ser consistente con el nombre del tema** cuando una arquitectura de microservicios se ve enfrentada a la necesidad de escalar sus operaciones tanto en inclusión de nuevos servicios tanto como un mayor afluente de peticiones. El resultado principal es la elaboración de

un conjunto de estrategias para arquitectos de software donde se relacione el dominio del problema, su arquitectura subyacente, aplicaciones y tecnologías óptimas para llevar a cabo la solución.

1.2.2. Objetivos Específicos:

Concretamente, en el marco de la creación de mensajes que deben ser transportados, comprendidos y ejecutados por varios sistemas, el trabajo espera:

- Generar una retrospectiva histórica de protocolos usados en la industria
- Estudiar y aplicar técnicas de serialización neutral en diversos lenguajes de programación cooperantes
- Establecer estrategias para la elección de transporte y serialización
- Crear una matriz arquitectural de herramientas neutrales a los lenguajes de programación
- Establecer criterios para la elección de protocolos textuales o binarios
- Contrastar la serialización binaria y la basada en texto plano
- Descubrir potenciales beneficios en la optimización de recursos humanos

1.3. Retrospectiva Histórica

1.3.1. Patrones

- RPC
- CORBA (OMG 2000)

1.3.2. Arquitecturas

- SOA **PATRÓN**
- Monolitos
- Microservicios
- Sistemas auto-contenidos

acerca de microservicios

A communications protocol is a system of digital message formats and rules for exchanging those messages in or between computing systems and in telecommunications

More informally, a protocol is an agreement between two computer systems on how they will talk to each other.

2 | Capítulo 2

2.1. Marco Teórico

MANTENERSE OBJETIVO EN ESTE CAPÍTULO

INCLUIR NÚMERO DE RFC EN CADA PROTOCOLO

Lo que se va a analizar

- Protocolos
- Serialización
- Comunicación entre procesos

2.2. Protocolos de Transferencia de Datos

Para que un dato sea transportado desde un punto a otro necesita tener una forma determinada para ser apto en el mecanismo de transporte, además debe tener una estructura replicable por ambas partes de tal manera que el dato recibido sea una copia fiel del original y sea entendido por ambas partes.

Cuando se trata de la transferencia de datos en arquitecturas de microservicios se debe considerar que el fundamento de todos estos mecanismos es TCP, que pone a disposición el concepto de puertos, para utilizar más de un protocolo por dirección IP; garantiza también el orden e integridad de los grupos de bytes en tránsito, por lo cual, los protocolos pueden construir instrumentos más sofisticados usando estos cimientos.

Un Protocolo de Transferencia de Datos se compone tanto de comandos (esto es un conjunto de reglas que deben ser seguidas en un orden determinado) que son utilizadas

en iniciar, negociar, mantener y terminar la conexión entre los puntos involucrados; se compone además de una estructura bien definida y replicable para los datos que han de ser transferidos. Dichas estructuras suelen incluir metadatos, como delimitadores de inicio y término de un dato o indicadores de la longitud de éstos mismos. La presencia, ausencia y combinaciones de estos indicadores y su emplazamiento, dan origen a innumerables estructuras posibles (formas de serialización).

Las formas de serialización son elementos intercambiables (y reemplazables) en los protocolos y muy raramente son parte de los protocolos mismos. Esto permite una evolución continua de las arquitecturas y la experimentación con nuevos elementos para optimizar el funcionamiento de los sistemas. De no ser así, esta memoria no sería posible.

MONO: PROTOCOLO = COMANDOS + SERIALIZACION

Configuraciones comunes en los protocolos de transferencia de datos incluyen métodos de compresión, indicadores configurables de fin de transferencia, indicación del formato de serialización, semánticas de orden y distribución, mecanismos de sincronización, entre otros. Otros parámetros menos usados junto a TCP son las rutinas de detección y corrección de errores.

Es necesario hacer la distinción entre los protocolos de transporte y los protocolos de transmisión de datos. Los primeros son aquellos que determinan la manera en que viajan **bytes** a través de las redes (contemplan a TCP y UDP); los segundos determinan las maneras en que son transportadas, embaladas y abiertas las **estructuras de datos** formadas por bytes. Estos últimos son usados por las aplicaciones en la intercomunicación. Un término usado en lengua inglesa es *wire protocol*, cuya traducción literal puede ser protocolo de cable.

Los protocolos de transferencia de datos pueden ser estructurados como comandos de texto o como una secuencia de bytes (también llamado modo binario). La elección de un protocolo basado en texto o uno binario es una decisión arquitectural fundamental, que engloba implicancias en el rendimiento del sistema, complejidad para los desarrolladores de aplicaciones e incluso implicancias a nivel ético y de recursos humanos.

Será la última década en el campo de la informática el foco generacional de esta memoria, de donde se analizan los protocolos relevantes para el desarrollo de software de

ésta y la próxima oleada de tecnología. El campo de observación será la evolución temporal que se produce cuando un proyecto nace del revisitar ideas anteriores, como el caso de HTTP/1.1 y HTTP2.

2.2.1. Protocolos Basados en Texto

Cuando los comandos que componen el guión que intercambiarán los sistemas están escritos en lenguajes humanos, estamos en la presencia de protocolos basados en texto, esto es una serie de instrucciones legibles por personas y que deben ser traducidos a formas interpretables por máquinas antes de ser interpretadas por el hardware **OJO ACÁ**. Estos protocolos tienen la característica de que no necesitan herramientas de software específicas y su funcionamiento puede ser analizado con técnicas forenses.

La incidencia de los protocolos de transferencia de datos basados en texto es enorme en la actualidad (tan dramática incluso, que se podría metaforizar que son el combustible de lo que hoy conocemos por Internet).

De esta categoría de protocolos, serán estudiados tres de ellos:

- STOMP
- HTTP
- ZMTP

Esta elección responde a la necesidad de evaluarlos en distintos modelos de comunicación (cliente-servidor, suscripción, punto-a-punto, etc) con el fin de que las mediciones a obtener sean útiles incluso en modelos nuevos.

2.2.1.1. STOMP

STOMP (*Simple/Streaming Text Oriented Messaging Protocol*, Protocolo Sencillo de Mensajes Orientado a Texto). Sus mensajes se componen de un encabezado con propiedades y un cuerpo. Los principios del diseño del protocolo se basan en generar mensajes sencillos y ampliamente inter-operables.

STOMP no maneja colas ni temas; usa una semántica de envío (SEND) con un texto de destino. El intermediario (broker) debe transformar este texto de forma interna en un tema, cola o intercambio. Los consumidores pueden suscribirse a esos destinos. Dado que los destinos no están definidos en la especificación, queda a merced de la implementación el tipo de destino. No siempre es posible cambiar de implementación de manera directa.

Sin embargo, STOMP es sencillo y liviano, con un gran número de librerías en diversos lenguajes. Posee también semánticas transaccionales. Esfuerzos actuales han permitido utilizar el protocolo a través de WebSockets lo que abre la posibilidad de llegar a los navegadores web, dispositivos móviles y aplicaciones en tiempo real.

2.2.1.2. HTTP 1.1

HTTP (*Hypertext Transfer Protocol*, Protocolo de Transferencia de Hipertexto) fue diseñado para sistemas de medios distribuidos y colaborativos. Ha sido usado desde 1990 en la iniciativa de la red amplia mundial (World-Wide Web). La primera versión, conocida como HTTP/0.9, era un protocolo simple para la transferencia de datos en bruto a través de internet. Su iteración posterior, HTTP/1.0 mejoró el protocolo al permitir que los mensajes fuesen codificados usando MIME-types; añadiendo metadatos sobre la información transferida y añadiendo modificaciones en la semántica de petición-respuesta. Sin embargo, HTTP/1.0 poseía falencias; no estaba diseñado para permitir intermediarios (proxies), caché, conexiones persistentes o hosts virtuales.

Los mensajes tienen la siguiente estructura:

- **Línea inicial** Termina con retorno de carro y un salto de línea. Su estructura depende de si se trata de una petición o una respuesta
 - Para las peticiones: la acción requerida por el servidor (método de petición) seguido de la URL del recurso y la versión HTTP que soporta el cliente
 - Para respuestas: La versión del HTTP usado seguido del código de respuesta (que indica que ha pasado con la petición seguido de la URL del recurso) y de la frase asociada a dicho retorno.

- **Cabeceras** Son metadatos y terminan con una línea en blanco. Estas cabeceras le dan gran flexibilidad al protocolo
- **Cuerpo** Es opcional. Su presencia depende de la línea anterior del mensaje y del tipo de recurso al que hace referencia la URL. Típicamente tiene los datos que se intercambian cliente y servidor. Por ejemplo para una petición podría contener ciertos datos que se quieren enviar al servidor para que los procese. Para una respuesta podría incluir los datos que el cliente ha solicitado.

Ejemplo

Petición

```
GET /index.html HTTP/1.1
Host: www.example.com
User-Agent: nombre-cliente
Referer: www.google.com
User-Agent: Mozilla/5.0
Connection: keep-alive
[Línea en blanco]
```

Respuesta

```
HTTP/1.1 200 OK
Date: Fri, 31 Dec 2003 23:59:59 GMT
Content-Type: text/html
Content-Length: 1221
```

```
<html>
</html>
```

2.2.1.3. ZMQ

ZMQ (ZeroMQ Message Transport Protocol) es un protocolo para la capa de transporte para intercambiar mensajes entre dos pares conectados a una capa de transporte para

intercambiar mensajes. Se trata de un protocolo de par a par.

Una conexión típica involucra

- Ambos pares acuerdan la versión y los mecanismos de seguridad de la conexión al enviarse mensajes y eligiendo continuar la conexión o bien cerrándola
- Se acuerda el mecanismo de seguridad intercambiando cero o más comandos. Si el acuerdo es exitoso, los pares continúan la discusión, de lo contrario uno o ambos cierran la conexión.
- Cada par envía al otro metadatos sobre la conexión a modo de último comando. Los pares comprueban los metadatos y deciden si seguir o cerrar la conexión.
- Cada par queda listo para intercambiar mensajes con el otro. Cualquiera de ellos puede cerrar la conexión en cualquier momento.

2.2.2. Protocolos Binarios

DEFINICION

Un protocolo binario es aquel diseñado para ser leído por máquinas o software en lugar de personas. Estos protocolos tienen la característica de ser sucintos, lo que se traduce en una potencial ventaja en rendimiento tanto en transmisión como interpretación,

Dichos protocolos requieren herramientas específicas para ayudar en el desarrollo y la detección de problemas. A diferencia de los protocolos basados en texto, estas no pueden ser usadas para más de un protocolo.

<http://martin.kleppmann.com/2012/12/05/schema-evolution-in-avro-protocol-buffers.html>

- AMQP
- MQTT
- CoAP
- Thrift
- HTTP2

2.2.2.1. AMQP

AMQP (*Advanced Message Queuing Protocol*, Protocolo Avanzado de Mensajería basado en Colas) fue diseñado como un reemplazo abierto frente a soluciones propietarias de mensajería. Dos de las razones más potentes para la adopción de AMQP han sido confiabilidad e interoperabilidad.

El protocolo incluye una gran cantidad de características de mensajería, incluyendo colas persistentes, mensajería de publicación-suscripción basada en tópicos, ruteo flexible, transacciones y seguridad. Los mensajes pueden ser entregados de forma directa, por tema e incluso basándose en las cabeceras.

Existe una gran flexibilidad para ajustar el funcionamiento del protocolo; es posible controlar el acceso a las colas de mensajes y manejar su profundidad, entre otras cosas.

Todo el protocolo se diseñó desde el comienzo para permitir la interoperabilidad entre implementadores.

BNF

```

amqp                = protocol-header *amqp-unit
protocol-header     = literal-AMQP protocol-id protocol-version
literal-AMQP        = %d65.77.81.80          ; "AMQP"
protocol-id         = %d0                    ; Must be 0
protocol-version    = %d0.9.1                ; 0-9-1
method              = method-frame [ content ]
method-frame        = %d1 frame-properties method-payload frame-end
frame-properties    = channel payload-size
channel             = short-uint              ; Non-zero
payload-size        = long-uint
method-payload      = class-id method-id *amqp-field
class-id            = %x00.01-%xFF.FF
method-id           = %x00.01-%xFF.FF
amqp-field          = BIT / OCTET
                    / short-uint / long-uint / long-long-uint

```

```

/ short-string / long-string
/ timestamp
/ field-table
short-uint      = 2*OCTET
long-uint       = 4*OCTET
long-long-uint  = 8*OCTET
short-string    = OCTET *string-char      ; length + content
string-char     = %x01 .. %xFF
long-string     = long-uint *OCTET        ; length + content
timestamp       = long-long-uint          ; 64-bit POSIX
field-table     = long-uint *field-value-pair
field-value-pair = field-name field-value
field-name      = short-string
field-value     = 't' boolean
/ 'b' short-short-int
/ 'B' short-short-uint
/ 'U' short-int
/ 'u' short-uint
/ 'I' long-int
/ 'i' long-uint
/ 'L' long-long-int
/ 'l' long-long-uint
/ 'f' float
/ 'd' double
/ 'D' decimal-value
/ 's' short-string
/ 'S' long-string
/ 'A' field-array
/ 'T' timestamp
/ 'F' field-table

```

	/ 'V'	; no field
boolean	= OCTET	; 0 = FALSE, else TRUE
short-short-int	= OCTET	
short-short-uint	= OCTET	
short-int	= 2*OCTET	
long-int	= 4*OCTET	
long-long-int	= 8*OCTET	
float	= 4*OCTET	; IEEE-754
double	= 8*OCTET	; rfc1832 XDR double
decimal-value	= scale long-uint	
scale	= OCTET	; number of decimal digits
field-array	= long-int *field-value	; array of values
frame-end	= %xCE	
content	= %d2 content-header *content-body	
content-header	= frame-properties header-payload frame-end	
header-payload	= content-class content-weight content-body-size property-flags property-list	
content-class	= OCTET	
content-weight	= %x00	
content-body-size	= long-long-uint	
property-flags	= 15*BIT %b0 / 15*BIT %b1 property-flags	
property-list	= *amqp-field	
content-body	= %d3 frame-properties body-payload frame-end	
body-payload	= *OCTET	
heartbeat	= %d8 %d0 %d0 frame-end	

2.2.2.2. MQTT

MQTT (*Message Queue Telemetry Transport*, Cola de Mensajes para Transporte Telerimétrico), fue desarrollado originalmente por IBM junto a otros actores de la industria y liberado años más tarde a la comunidad de código abierto. Actualmente está siendo

desarrollado por el consorcio OASIS.

Los principios de diseño y metas de MQTT son mucho más simples que AMQP: entregar un modelo de publicación-suscripción sin colas (a pesar del nombre), donde el énfasis está en el intercambio de información entre dispositivos de recursos limitados, altas latencias y bajos anchos de banda.

- Sólo cinco métodos
- No es posible agregar metadatos a los mensajes
- Encabezados comprimidos

Soporta QoS 1, 2 y 3

Facebook usa el protocolo para sus servicios de mensajería ("Facebook Messenger") con gran éxito REFERENCIA

2.2.2.3. HTTP2

ORIGEN <– de dónde, quién

Características Binary Protocol HTTP/2 is a binary protocol. This means that it is much more efficient on the wire, however as a result it is no longer human readable without using tools to decode the protocol.

Multiplexing HTTP/2 supports multiplexing several streams over a single connection. This means that a client can send multiple requests on the same connection, and the server can respond in whatever order the responses become available.

Header Compression HTTP requests and responses generally include a large number of redundant headers. HTTP/2 uses HPACK header compression to greatly compress headers.

Server Push

Server push allows a server to send additional cacheable resources to the client that the client has not explicitly asked for. The allows the server to anticipate the resources the client will request next and send them eagerly, which saves a round trip.

Protocolo Binario Binary protocols are more efficient to parse, more compact “on the wire”, and most importantly, they are much less error-prone, compared to textual

protocols like HTTP/1.x, because they often have a number of affordances to “help” with things like whitespace handling, capitalization, line endings, blank lines and so on ¹.

HTTP/2 was developed by the IETF’s HTTP Working Group, which maintains the HTTP protocol. It’s made up of a number of HTTP implementers, users, network operators and HTTP experts.

DISEÑO <– para qué

- is binary, instead of textual
- is fully multiplexed, instead of ordered and blocking; can therefore use one connection for parallelism
- uses header compression to reduce overhead
- allows servers to “push” responses proactively into client caches

ESPECIFICACIÓN <– principalmente ventajas

MÁQUINAS DE ESTADO <– mono

QoS <– 1,2,3 QUIÉN LO USA <– farándula

2.2.2.4. CoAP

ORIGEN <– de dónde, quién

DISEÑO <– para qué

ESPECIFICACIÓN <– principalmente ventajas

MÁQUINAS DE ESTADO <– mono

QoS <– 1,2,3 QUIÉN <– farándula

BNF

```
; -----
; | RFC5234 |
; -----
```

¹IEFT Working Group <https://http2.github.io/faq/>

ALPHA	=	%x41-5A / %x61-7A	
			; A-Z / a-z
BIT	=	"0" / "1"	
CHAR	=	%x01-7F	
			; any 7-bit US-ASCII character,
			; excluding NUL
CR	=	%x0D	
			; carriage return
CRLF	=	CR LF	
			; Internet standard newline
CTL	=	%x00-1F / %x7F	
			; controls
DIGIT	=	%x30-39	
			; 0-9
DQUOTE	=	%x22	
			; " (Double Quote)
HEXDIG	=	DIGIT / "A" / "B" / "C" / "D" / "E" / "F"	
HTAB	=	%x09	
			; horizontal tab
LF	=	%x0A	
			; linefeed
LWSP	=	*(WSP / CRLF WSP)	
			; Use of this linear-white-space rule
			; permits lines containing only white
			; space that are no longer legal in
			; mail headers and have caused
			; interoperability problems in other
			; contexts.
			; Do not use when defining mail

```

                                ; headers and use with caution in
                                ; other contexts.

OCTET      = %x00-FF
                                ; 8 bits of data

SP         = %x20
VCHAR      = %x21-7E
                                ; visible (printing) characters

WSP        = SP / HTAB
                                ; white space

; -----
; | RFC3986 |
; -----

URI        = scheme ":" hier-part [ "?" query ] [ "#" fragment ]
hier-part  = "//" authority path-abempty
            / path-absolute
            / path-rootless
            / path-empty

URI-reference = URI / relative-ref

absolute-URI = scheme ":" hier-part [ "?" query ]

relative-ref = relative-part [ "?" query ] [ "#" fragment ]

relative-part = "//" authority path-abempty
              / path-absolute
              / path-noscheme
              / path-empty

```

```

scheme      = ALPHA *( ALPHA / DIGIT / "+" / "-" / "." )

authority   = [ userinfo "@" ] host [ ":" port ]
userinfo    = *( unreserved / pct-encoded / sub-delims / ":" )
host        = IP-literal / IPv4address / reg-name
port        = *DIGIT

IP-literal  = "[" ( IPv6address / IPvFuture  ) "]"

IPvFuture   = "v" 1*HEXDIG "." 1*( unreserved / sub-delims / ":" )

IPv6address =
    6( h16 ":" ) ls32
    /
    "::" 5( h16 ":" ) ls32
    / [
        h16 ] "::" 4( h16 ":" ) ls32
    / [ *1( h16 ":" ) h16 ] "::" 3( h16 ":" ) ls32
    / [ *2( h16 ":" ) h16 ] "::" 2( h16 ":" ) ls32
    / [ *3( h16 ":" ) h16 ] "::"   h16 ":"   ls32
    / [ *4( h16 ":" ) h16 ] "::"                        ls32
    / [ *5( h16 ":" ) h16 ] "::"                        h16
    / [ *6( h16 ":" ) h16 ] "::"

h16         = 1*4HEXDIG
ls32        = ( h16 ":" h16 ) / IPv4address
IPv4address = dec-octet "." dec-octet "." dec-octet "." dec-octet
dec-octet   = DIGIT                    ; 0-9
             / %x31-39 DIGIT           ; 10-99
             / "1" 2DIGIT              ; 100-199
             / "2" %x30-34 DIGIT       ; 200-249
             / "25" %x30-35            ; 250-255

```



```

reg-name      = *( unreserved / pct-encoded / sub-delims )

path          = path-abempty    ; begins with "/" or is empty
               / path-absolute  ; begins with "/" but not "/"
               / path-noscheme  ; begins with a non-colon segment
               / path-rootless  ; begins with a segment
               / path-empty     ; zero characters

path-abempty  = *( "/" segment )
path-absolute = "/" [ segment-nz *( "/" segment ) ]
path-noscheme = segment-nz-nc *( "/" segment )
path-rootless = segment-nz *( "/" segment )
path-empty   = 0<pchar>

segment       = *pchar
segment-nz    = 1*pchar
segment-nz-nc = 1*( unreserved / pct-encoded / sub-delims / "@" )
               ; non-zero-length segment without any colon ":"

pchar         = unreserved / pct-encoded / sub-delims / ":" / "@"

query         = *( pchar / "/" / "?" )

fragment      = *( pchar / "/" / "?" )

pct-encoded   = "%" HEXDIG HEXDIG

unreserved    = ALPHA / DIGIT / "-" / "." / "_" / "~"
reserved      = gen-delims / sub-delims

```

```

gen-delims    = ":" / "/" / "?" / "#" / "[" / "]" / "@"
sub-delims    = "!" / "$" / "&" / "'" / "(" / ")"
               / "*" / "+" / "," / ";" / "="

```

```

; -----
; | RFC7252 |
; -----

```

```

coap-URI = "coap:" "//" host [ ":" port ] path-abempty [ "?" query ]
coaps-URI = "coaps:" "//" host [ ":" port ] path-abempty [ "?" query ]

```

```

; -----
; | RFC6690 |
; -----

```

```

Link          = link-value-list
link-value-list = [ link-value *[ "," link-value ] ]
link-value     = "<" URI-Reference ">" *( ";" link-param )
link-param     = ( ( "rel" "=" relation-types )
                   / ( "anchor" "=" DQUOTE URI-Reference DQUOTE )
                   / ( "rev" "=" relation-types )
                   / ( "hreflang" "=" Language-Tag )
                   / ( "media" "=" ( MediaDesc
                                   / ( DQUOTE MediaDesc DQUOTE ) ) )
                   / ( "title" "=" quoted-string )
                   / ( "title*" "=" ext-value )
                   / ( "type" "=" ( media-type / quoted-mt ) )
                   / ( "rt" "=" relation-types )
                   / ( "if" "=" relation-types )

```

```

        / ( "sz" "=" cardinal )
        / ( link-extension ) )
link-extension = ( parmname [ "=" ( ptoken / quoted-string ) ] )
                / ( ext-name-star "=" ext-value )
ext-name-star  = parmname "*" ; reserved for RFC-2231-profiled
                ; extensions. Whitespace NOT
                ; allowed in between.
ptoken         = 1*ptokenchar
ptokenchar     = "!" / "#" / "$" / "%" / "&" / "'" / "("
                / ")" / "*" / "+" / "-" / "." / "/" / DIGIT
                / ":" / "<" / "=" / ">" / "?" / "@" / ALPHA
                / "[" / "]" / "^" / "_" / "`" / "{" / "|"
                / "}" / "~"
media-type     = type-name "/" subtype-name
quoted-mt      = DQUOTE media-type DQUOTE
relation-types = relation-type
                / DQUOTE relation-type *( 1*SP relation-type ) DQUOTE
relation-type  = reg-rel-type / ext-rel-type
reg-rel-type   = LOALPHA *( LOALPHA / DIGIT / "." / "-" )
ext-rel-type   = URI
cardinal       = "0" / ( %x31-39 *DIGIT )
LOALPHA        = %x61-7A ; a-z
;quoted-string = <defined in [RFC2616]>
;URI           = <defined in [RFC3986]>
;URI-Reference = <defined in [RFC3986]>
;type-name     = <defined in [RFC4288]>
;subtype-name  = <defined in [RFC4288]>
;MediaDesc     = <defined in [W3C.HTML.4.01]>
;Language-Tag  = <defined in [RFC5646]>
;ext-value     = <defined in [RFC5987]>

```

```

;parmname          = <defined in [RFC5987]>

; -----
; | draft-ietf-core-http-mapping-12 |
; -----

s  = "coap" / "coaps" ; from [RFC7252], Sections 6.1 and 6.2
hp = host [":" port]  ; from [RFC3986], Sections 3.2.2 and 3.2.3
p  = path-abempty     ; from [RFC3986], Section 3.3
q  = query             ; from [RFC3986], Section 3.4
qq = [ "?" query ]    ; qq is empty if and only if 'query' is empty
tu = coap-URI / coaps-URI ; from [RFC7252], Section 6.1 and 6.2

```

2.2.2.5. Apache Thrift

Thrift es un proyecto que engloba un protocolo binario orientado a flujos, un lenguaje de definición de interfaces, un conjunto de mecanismos de serialización y herramientas para la generación de código. A diferencia del resto de los proyectos estudiados acá, Thrift intenta entregar una solución completa en la transferencia de mensajes.

Fue desarrollado inicialmente en Facebook para el desarrollo escalable de servicios que abarcan múltiples lenguajes de programación. Más tarde, fue donado a la fundación Apache quien es el actual gobernante del proyecto.

2.3. Serialización

2.3.1. Serialización Basada en Texto

DEFINIR

- XML
- JSON

2.3.1.1. XML

XML (Extensible Markup Language, Lenguaje de Marcado Extensible) es un lenguaje basado en etiquetas que define un conjunto de reglas para la codificación de documentos en un formato que a la vez legible por humanos y máquinas. La especificación es gobernada por el Consorcio Mundial de la Web (W3C, World Wide Web Consortium).

Los principios de diseño de XML enfatizan la simplicidad, generalización y usabilidad a través de todo Internet. El formato se basa netamente en texto, con soporte Unicode para permitir el uso amplio de lenguajes humanos.

The design goals of XML emphasize simplicity, generality, and usability across the Internet.[5] It is a textual data format with strong support via Unicode for different human languages. Although the design of XML focuses on documents, the language is widely used for the representation of arbitrary data structures[6] such as those used in web services.

Several schema systems exist to aid in the definition of XML-based languages, while programmers have developed many application programming interfaces (APIs) to aid the processing of XML data.

2.3.1.2. JSON

2.3.1.3. YAML

2.3.2. Serialización Binaria

DEFINIR

- Protocol Buffers
- Avro
- Thrift

	Uno a uno	Uno a muchos
Síncrono	Petición-respuesta	
Asíncrono	Notificación Productor-consumidor	Petición-respuesta asíncrona Productor asíncrono

2.3.2.1. Protocol Buffers

2.3.2.2. Avro

2.3.2.3. Thrift

- BinaryProtocol
- CompactProtocol

2.4. Patrones de Transferencia de Datos Entre Procesos

2.4.1. Transferencia Síncrona

2.4.1.1. Petición-Respuesta

2.4.2. Transferencia Asíncrona

2.4.2.1. Uno a uno

Notificación Productor-Consumidor

2.4.2.2. Uno a muchos

Petición/Respuesta asíncrona (push) Productor asíncrono (suscripción)

3 | Capítulo 3

3.1. Diseño y Construcción de las Pruebas

Para evaluar el rendimiento, complejidad y escalabilidad de cada combinación de protocolo de transferencia, formato de serialización y patrón de comunicación entre procesos, se diseñarán dos arquitecturas de software; una enfocada en operaciones transaccionales y otra con menos garantías.

3.2. Metodología

Mido escalabilidad con peticiones bidireccionales.

Establecer un criterio para diferenciar la manera en que se distribuyen las peticiones dentro del sistema; argumentar que la naturaleza de la arquitectura tiene grandes repercusiones en la utilidad de los resultados de esta memoria; aún así, es posible generar mediciones que sirvan de aproximación.

Usar una matriz de 3x3 (o R3) con un eje que indique densidad de la arquitectura, el otro indica peticiones y el último indica tipo de distribución (aleatoria, euclidiana, constante)

MODELOS DE DISTRIBUCIÓN

El uso de modelos de distribución permitir comprobar la solidez de los resultados cuando cambia la arquitectura o demostrar que no son corroborables. Es también una medida para ir controlando los experimentos.

Tomar tiempo hasta el primer byte útil, latencias aportadas por el protocolo usando el mismo broker, mismos servidores, mismos servicios en una misma arquitectura; sólo medir lo que aporta el protocolo y las serializaciones cuando el resto se mantiene constante.

- Aleatorio
- Euclidiano
- Constante (acá hay que crear una forma de generar un patrón aleatorio que se mantenga durante toda la medición)
- Levantar máquinas
- instalar software
- escoger apps
- programar gateway
- programar scheduler de tareas
- alguna forma de variar los parámetros

3.2.1. Procedimiento

Una vez implementadas las arquitecturas, **brokers,**

3.2.2. Instrumentos

<http://www.dell.com/us/business/p/poweredge-r920/pd>

- CPU
- RAM
- Red
- Disco

3.2.2.1. Software

Para realizar las mediciones se utilizará

- Typhoon
- Apache jMeter

3.2.3. Estrategia de análisis

3.2.4.

3.3. Métricas

Rendimiento

- Mensajes por segundo
- Operaciones por segundo
- Overhead de transporte
- Overhead de serialización
- Latencia hasta el primer byte del cuerpo
- Latencia hasta el primer byte del cuerpo

Operacionales

- Infraestructura
- Complejidad percibida
- Líneas de código
- Tareas automatizables
- Tareas automatizables

Documentación

- Estándares
- Librerías disponibles
- Lenguajes soportados

4 | Capítulo 4

4.1. Mediciones

INCLUIR ACÁ LOS RESULTADOS DE LAS MEDICIONES
INCLUIR ANÁLISIS INICIAL DE LOS RESULTADOS

5 | Capítulo 5

5.1. Validación y Análisis de los Resultados

Recomendaciones generales:

- *Presenta los resultados interpretando los datos a la luz del marco teórico planteado, dependiendo de la pregunta de investigación y los objetivos de la misma.*
- *Recuerda que los datos no son los resultados. Los resultados vienen del análisis de los datos para encontrarles sentido. Se apoyan en los datos, pero son interpretaciones del estudiante que requieren de argumentación para que tengan sentido. Además, se debe presentar la respuesta que el estudiante da a la pregunta de investigación, con la argumentación que lo lleva a dar esta respuesta.*

5.2. Conclusiones

La función principal de las conclusiones es exponer los hallazgos, observaciones y posibles retos que se desprendan de la investigación realizada. Esta sección nunca puede, como parte formal, ser más extensa que la investigación misma. Una buena sección de conclusiones expondrá de manera puntual y ordenada una síntesis de la investigación realizada a partir de los resultados obtenidos en relación con la hipótesis que dio inicio al trabajo. Al exponer los resultados, debe poder apreciarse si se trata de resultados esperados, si difieren de la expectativas, si son parciales o insuficientes. Es decir, los resultados deben referenciarse con el punto de partida de la investigación. Esto hace que, entre la

sección de introducción y la de las conclusiones, haya un vínculo evidente.

Es importante considerar que las conclusiones no constituyen un resumen de los capítulos o partes de la tesis. En esta sección tampoco es apropiado introducir nuevos elementos teóricos ni hay lugar para retomar el objeto de estudio desde otras perspectivas.

La parte más valiosa de las conclusiones es la reflexión que se deriva de los resultados obtenidos. No es suficiente listar los resultados. A partir de ellos hay que elaborar observaciones y, sobre todo, nuevas preguntas para futuras investigaciones que permitan aportar en la construcción del conocimiento.

Bibliografía



A | ANEXO 1

