PEC1. Sistemas distribuidos.

Kepa Sarasola Bengoetxea

1. Explicar el concepto de transparencia en sistemas distribuidos. Proporcionar ejemplos de transparencia de acceso y transparencia de escala:

La transparencia en un sistema distribuido se define como la ocultación a los usuarios y a los programadores de aplicaciones que usan los diferentes recursos del sistema, de la separación de los componentes del mismo, de modo que el sistema se percibe como un todo, un único sistema y no como una colección de componentes independientes.

El Manual de Referencia ANSA [ANSA 1989] y la Organización Internacional para la Normalización del modelo de referencia para el procesamiento distribuido abierto (RM-ODP) [ISO 1992] identifican ocho formas de transparencia:

- **Transparencia de acceso:** permite que los usuarios locales y remotos puedan realizar accesos a los recursos del sistema usando las mismas operaciones, independientemente de que estos recursos sean locales o remotos.

- **Transparencia de movilidad:** permite el acceso a los recursos del sistema sin tener conocimiento sobre la ubicación física o de la red en la que están situadas estas. También se refiere a que el sistema debe de mantenerse igualmente consistente independientemente del lugar desde el que el usuario o programa haga referencia al sistema.

- **Transparencia de concurrencia:** permite que varios procesos puedan operar de manera simultanea usando recursos compartidos por el sistema sin que por ello se sufran interrupciones o interferencias.

- **Transparencia de replicación:** permite el uso de varias instancias replicadas de los recursos de sistema con el objetivo de aumentar el rendimiento y la fiabilidad del sistema sin que exista un conocimiento previo por parte de los usuarios o programadores.

- **Transparencia de fallos:** permite la ocultación de errores a usuarios y programas que están usando recursos del sistema para que estos puedan completar sus tareas a pesar de errores o fallos que puedan darse en componentes de hardware o software del sistema.

- **Transparencia de movilidad:** permite la movilidad de los recursos y los clientes dentro de un sistema sin que esto afecte al funcionamiento del sistema y por lo tanto al funcionamiento de los programas y usuarios que aloja o que usan este.

- **Transparencia de rendimiento:** permite que el sistema pueda reconfigurarse para aumentar el rendimiento de esta en función de las cargas que ha de soportar, sin que por ello puedan verse afectados los usuarios y programas del sistema.

- **Transparencia de escala:** permite que el sistema y las aplicaciones puedan expandirse en escala sin que para ello haya que realizar cambios en la estructura del sistema o en los algoritmos de la aplicación.

Ejemplos de transparencia de acceso

En el libro “Distributed Systems: Concepts and Design” se comentan varios ejemplos de transparencia de acceso; como una interfaz gráfica de usuario para un sistema de ficheros o una API para el acceso a archivos que funcionaría de la misma manera en archivos locales y remotos.

A estos ejemplos añadiría los siguientes:

- El Interface de acceso a una impresora, que puede ser local o remota, pero que en todo caso usará la misma interface de comunicación con el usuario.

- Acceso a una base de datos, el acceso a base de datos, sea a una base de datos local o remota, incuso puede estar distribuida en varios servidores, cuando se hace de la misma manera, sin tener que diferenciar la manera de acceder a estas en función de cuales, como y donde estén. En este caso estaremos ante un ejemplo de transparencia de acceso.

Ejemplos de transparencia de escala

La transparencia de escala oculta el aumento de los recursos de sistema de manera horizontal o vertical al usuario o programas que usan estos recursos del sistema. Ejemplos:

- Aumento de la capacidad de almacenamiento de un sistema pasando este de un disco físico a un almacenamiento en la nube. La transparencia de escala permitirá al usuario usar el mismo interface o API de comunicación para las operaciones de almacenamiento, lectura, etc.

- En el caso de un escalado horizontal, es decir, cuando el crecimiento se basa en la agregación de nuevos nodos, la transparencia de escala permitirá al usuario usar los recursos del sistema independientemente del número de nodos que compongan este.

2. Explicar las diferencias entre el escalado vertical y horizontal en sistemas distribuidos con ejemplos.

La escalabilidad es la propiedad de un sistema, una red o un proceso, que indica su habilidad para reaccionar y adaptarse sin perder calidad, o bien manejar el crecimiento continuo de trabajo de manera fluida.

En general, también se podría definir como la capacidad del sistema informático de cambiar su tamaño o configuración para adaptarse a las circunstancias cambiantes, podemos decir pues que un sistema es escalable si conserva su efectividad ante cambios considerables en cuanto al número de recursos y al número de usuarios o peticiones que recibe.

La escabilidad puede darse en forma vertical y/o horizontal, la diferencia entre ambas es que mientras que el escalado vertical añade más recursos a un nodo del sistema, la escalabilidad horizontal añade más nodos al sistema, en ambos casos, el objetivo es que el rendimiento general del sistema mejore.

Escalado vertical

Como hemos dicho el objetivo del escalado vertical es aumentar la potencia o la capacidad de uno ovarios nodos del sistema. Ejemplos de escalado vertical son el aumento de la capacidad de disco, memoria RAM o del número de CPUs de un nodo del sistema o incluso la sustitución de este por otro más potente.

Escalado horizontal

En el caso del escalado horizontal, esta se basa en la modularidad del sistema, y la agregación se consigue aumentando el número de nodos del sistema, es decir, añadiendo equipos a este.

Un ejemplo de escalado horizontal sería el de añadir nuevos equipos a un sistema con el objetivo de pasar de un único servidor a pasar a un sistema con varios servidores y un balanceador de carga que se encargue de distribuir la carga de trabajo entre estos.

3. Comparar modelos de comunicación síncrona y asíncrona con diferentes ejemplos reales. Explica cuándo usar cada uno de ellos. ¿Qué modelo es más escalable? ¿Por qué?

La comunicación síncrona y asíncrona son dos maneras de intercambiar información entre un emisor y un receptor, cuya diferencia básica es la simultaneidad con la que se envía y se recibe el mensaje.

La comunicación síncrona es aquella en la que el emisor y el receptor del mensaje coinciden en el tiempo, mientras que mediante la comunicación asíncrona el emisor y el receptor utilizan el mismo medio de comunicación pero en tiempos diferentes.

En términos computacionales podemos decir que mientras la comunicación asíncrona no espera una respuesta inmediata del receptor, la comunicación síncrona requiere que las partes estén disponibles y puedan comunicarse en tiempo real.

Podemos señalar como ejemplos de comunicación síncrona:

- Una video conferencia, una llamada telefónica o una conversación presencial entre dos o más personas.
- Una página web que descarga toda la información que necesita en el mismo momento en que se recibe la petición en el servidor. Por ejemplo:
    - Stackoverflow, al realizar una búsqueda y pasar de una página a otra se vuelve a recargar todo el contenido de la página. La comunicación es síncrona, ya que cuando se hace la petición al servidor, el cliente se queda a la espera de la respuesta para poder dibujar la página web en el navegador.

Mientras que en el caso de la comunicación asíncrona, los ejemplos son:

- El correo tradicional, el correo electrónico, los mensajes y discursiones en un foro de debate, y las aplicaciones de chat como Whatsapp, Telegram, etc.
- Una página web con arquitectura SPA (Simple Page Application) que va realizando peticiones de información al servidor mediante AJAX según va necesitando esta. Podemos encontrar ejemplos de páginas web muy populares que usan el modelo asíncrono para incorporar nueva información a sus páginas:
    - Muchas de las redes sociales: Twitter, Instagram, Facebook,….
    - El buscador duckduckgo.com incorpora un botón al final de la página “Más resultados” que realiza una llamada AJAX, utilizando una comunicación asíncrona, para añadir más resultados a la página ya cargada anteriormente.

Podemos observar las características de ambos sistemas en la siguiente tabla:

Comunicación síncrona	Comunicación asíncrona
• El emisor y el receptor deben de estar activados y en linea. • El emisor espera a recibir una respuesta del receptor antes de continuar. • El emisor no puede continuar hasta recibir la respuesta, podemos decir que se bloquea.	• El receptor puede no estar activo, una vez activado recibirá la petición y la responderá. • El emisor deposita su petición en una cola de mensajes y no espera una respuesta inmediata del receptor • El emisor puede seguir procesando su cola de tareas mientras llega la respuesta del receptor.

Uso

Sobre la conveniencia del uso de una u otra, dependerá de la necesidad concreta que se pretenda resolver, pero en general diría que aunque la comunicación síncrona es más rápida y más sencilla de gestionar técnicamente, la comunicación asíncrona tiene la ventaja de poder gestionar de una manera más eficaz los recursos del sistema, ya que estos serán requeridos solamente los que sean necesarios y en el momento en que sean necesarios. Como inconveniente del sistema de comunicación asíncrona conviene destacar que los tiempos de espera pueden aumentar y así como que hay que tener una correcta gestión de los errores que se puedan producir.

Escalado

La comunicación asíncrona es más escalable ya que no es bloqueante como la comunicación síncrona, en este último caso el *thread* que hace la petición ha de esperar hasta que reciba la respuesta, lo cual supone un desperdicio de recursos, ya que mientras espera no puede realizar otras tareas. La comunicación asíncrona permite un mejor uso de los recursos del sistema, aumentando el rendimiento de esta y la posibilidad de responder a un mayor volumen de trabajo, por lo tanto es más escalable.

Como contrapartida cabe señalar que los sistemas síncronos son más rápidos y seguros, ya que en caso de que se dé algún error un sistema asíncrono es muy probable que el receptor no tenga constancia de este, existiendo la posibilidad de que se de la propagación del error.

4. Leer el siguiente artículo y responder a las preguntas: Ray: A Distributed Framework for Emerging AI Applications

https://arxiv.org/pdf/1712.05889.pdf

a) Explica las principales ideas presentadas en este trabajo.

En este trabajo se realiza una justificación, presentación y evaluación del proyecto Ray, presentando este como una alternativa optima para el desarrollo de sistemas de *aprendizaje reforzado* (RL). Tras realizar una presentación del estado del arte actual sobre la expansión de técnicas complejas de *Inteligencia Artificial* (IA) y del *aprendizaje automático* (ML), el artículo presenta lo que considera que es y será el futuro de las aplicaciones de la IA, el *aprendizaje reforzado* (RL) y realiza una propuesta de arquitectura distribuida para llevar a cabo de una manera óptima dicha técnica.

El RL tiene como objetivo aprender a realizar acciones de una manera continua y en un contexto cambiante. El objetivo es aprender una política y un mapeo constante del contexto que posibilite el desempeño a lo largo del tiempo de una actividad concreta, p.e. pilotar un dron.

En este contexto el trabajo evalúa que las soluciones actuales utilizadas sobre todo en entornos de aprendizaje supervisado, no pueden responder al reto de la RL, ya que este debe de trabajar con un grano fino de cálculos que le permitan representar acciones en milisegundos e interactuar con el mundo real. Por lo tanto llega a la conclusión de que se necesita un marco de cálculo dinámico que maneje millones de tareas heterogéneas por segundo en latencias de milisegundos.

Según sus autores, un framework válido para las aplicaciones RL debe proporcionar un soporte eficiente para el entrenamiento, el servicio y la simulación. Aunque en teoría se podría utilizar una combinación de los frameworks utilizados actualmente para solucionar estas necesidades, en la práctica, el movimiento de datos entre los frameworks se volvería demasiado grande y la latencia entre los diferentes sistemas harían inviable esa vía.

Ray es una solución que se presenta como la unificación de dos soluciones existentes actualmente de manera separada, la abstracción de programación paralela de tareas y una abstracción de autor. Ray surge con el objetivo de satisfacer los siguientes requisitos:

- Realización de cálculos heterogéneos y detallados
- Modelo de cálculo flexible: trabajando con cálculos con y sin estado, para abarcar simulaciones de grado fino y a su vez poder implementar servidores de parámetros.
- Ejecución dinámica: el orden en que se determinan los cálculos no siempre se conoce de antemano, por ejemplo algunos resultados de un cálculo pueden determinar cálculos futuros.

Ray ofrece la posibilidad de poder realizar millones de tareas por segundo permitiendo la integración con sistemas de simulación y frameworks de aprendizaje profundo. Es un proyecto de código abierto, desarrollado por la Universidad de California. Se integra completamente con el entorno Python y es fácil de instalar ejecutando *pip install ray*.

b) Describe la arquitectura distribuida, los modelos de comunicación y los mecanismos de tolerancia a fallos.

Arquitectura

La arquitectura de Ray comprende una capa de aplicación que implementa la API y una capa de sistema que proporciona alta escalabilidad y tolerancia a fallos.

Capa de la aplicación: consta de 3 tipos de procesos:

- Driver: Un proceso que ejecuta el programa de usuario.
- Worker: Un proceso sin estado que ejecuta tareas (funciones remotas) invocadas por un controlador (driver) u otro worker. Cuando se declara una función remota, la función se publica automáticamente para todos los workers. El worker ejecuta tareas en serie.
- Actor: es un proceso con estado que ejecuta, solo los métodos que expone. Un worker o un controlador crean una instancia explicita de un actor. Los actores ejecutan métodos en serie.

Capa de sistema: consta de 2 componentes principales:

- **Almacén de control global (GCS):** mantiene todo el estado de control del sistema, la razón principal de su existencia es mantener la tolerancia a fallos y la baja latencia para un sistema que puede generar millones de tareas por segundo.

  Con el objetivo de mantener una latencia baja en el sistema, se almacenan los metadatos del objeto en el CGS en vez de en el programador, desacoplando completamente el envío de tareas de la programación de estas.

  La separación entre el programador y el CGS permite que todos los componentes del sistema sean autónomos, simplificando la tolerancia a fallos, además facilita escalar el almacén de objetos distribuidos y el programador de manera independiente, pudiendo añadir al sistema más instancias de GCS o de programadores globales de manera autónoma.

- **Programador distribuido ascendente:** Es un programador jerárquico de dos niveles que consta de un programador global y programadores locales por nodo. Para no sobrecargar el planificador global, las tareas creadas en un nodo se envían primero al planificador del nodo, y en caso de que este no pueda atenderlas, se envían al programador global.

  El programador global estudia la carga de cada nodo y selecciona el nodo que proporciona el menor tiempo de espera estimado para la realización de la tarea. Si el programador global se

convierte en un cuello de botella, podemos añadir más replicas compartiendo la misma información a traves del CGS.

Almacén de objetos distribuidos en memoria
Para aumentar la rapidez del sistema se ha implementado un sistema de almacenamiento distribuido en memoria para almacenar las entradas y salidas de cada tarea o cálculo sin estado. Este almacén de objetos se ha implementado en cada nodo, permitiendo compartir datos entre tareas dentro del mismo nodo. Así, la replicación de nodos elimina el cuello de botella debido a los objetos de datos calientes y minimiza el tiempo de ejecución de la tarea ya que solo escribe o lee datos desde o hacia la memoria local.

El almacenamiento de objetos se limita a datos inmutables, evitando la necesidad de protocolos de consistencia complejos y simplificando el soporte para la tolerancia a fallos. En caso de fallo del nodo, se recupera cualquier objeto para la re-ejecución del linaje, esta recuperación del objeto se efectúa mediante el GCS que rastrea tanto las tareas sin estado como los actores con estado durante la ejecución inicial.

El almacén de datos no admite objetos distribuidos, cada objeto sólo encaja en un nodo. Los objetos complejos se pueden implementar a nivel de aplicación como colecciones de futuros.

Modelo de Comunicación

En cuanto al modelo de comunicación es importante subrayar que Ray separa al programador de la ruta crítica, es decir, se separan el GCS y el programador, de esta manera se desacoplan el envío de las tareas con la programación de estas.

Estamos pues ante un modelo de comunicación asíncrona, ya que como hemos indicado en el apartado a) los objetos que han de realizar los cálculos trabajan con compromisos a futuro, esto significa que las tareas son programadas, pero que el programador no se queda esperando al resultado de la operación, más bien se produce una especie de contrato que normalmente se traducirá en un Id pero sin valor y que se rellenará una vez que se realice el cálculo. Este sistema puede producir errores que como veremos más adelante se intentan subsanar mediante la re-programación de tareas

La separación de los componentes del sistema permite que todos los componentes del sistema sean independientes y puedan escalar de manera diferente según las necesidades del sistema

Mecanismos de tolerancia a fallos

Respecto de los mecanismos de tolerancia a fallos, la separación del GCS y del programador facilitan enormemente la re-planificación de tareas que hayan sido erróneas o que no se hayan podido realizar de una manera satisfactoria, ya que toda la información necesaria para poder volver a realizar estas se encuentra en el GCS, sin la necesidad de la participación del programador del sistema.

c) Proporciona tu propia perspectiva sobre el artículo e intenta encontrar las limitaciones del modelo propuesto.

He encontrado la propuesta del artículo muy interesante ya que el modelo propuesto permite la distribución del sistema en tres niveles que pueden ser escalados de manera independiente.

- El almacén de control global donde se guardan los meta-datos de los objetos.
- La programación de las tareas.
- Capa de aplicación, lo nodos.

Además se ha buscado la manera de optimizar los nodos mediante el uso del programador local y el almacén de objetos distribuidos, que tienen como objetivo realizar la mayor parte de las operaciones posibles dentro del mismo nodo utilizando para ello sistemas de replicación.

Respecto a la escalabilidad del modelo, la propuesta basada en un separación de tareas reflejada en la arquitectura mediante el almacén de control global y el programador distribuido de dos niveles parece que un sistema de una alta escalabilidad al estar diferenciados el almacén de control y el programador, pudiendo escalar estos de manera independiente.

En el informe los autores hablan de una escalabilidad lineal de hasta 1.8 millones de tareas por segundo, bajo mi punto de vista la única limitación entonces será la posibilidad de inversión para poder escalar la herramienta hasta lograr el limite.

5. Mira el siguiente video: How we've scaled Dropbox

https://www.youtube.com/watch?v=PE4gwstWhmc

a) Describe las principales ideas presentadas en este video.

El video describe los principales modelos de arquitectura que Dropbox desarrollo a medida de que fue creciendo entre los años 2007 y 2012, donde se paso de una arquitectura basada en un único servidor a un sistema distribuido y escalable basado en 3 capas:

- El servidor de bloques (blockserver) y el servidor de ficheros alojado en Amazon S3.
- Un servidor de metadatos apoyado en memcache y en una base de datos MySql.
- Notification server que se utiliza para notificar al usuario de cambios que haya podido haber en los ficheros que tiene alojados en el sistema.

En el video se subrayan 4 ideas respecto a las necesidades que trata de cubrir el sistema distribuido ideado por Dropbox:

- El sistema es usado por más de 10 millones de usuarios que alojan más de cientos de millones de archivos, por lo que ha de responder a una alta concurrencia.
- Se trata de una producto en el que la relación lectura/escritura es muy cercana a 1, esto significa que las operaciones de escritura son muchas en comparación con otro tipo de productos.
- Es indispensable mantener la consistencia e integridad de los datos, no se pueden permitir el lujo de perder un archivo de un usuario.
- Trabajan con un único producto, almacenamiento y sincronización de ficheros, no hay diversificación del producto por lo tanto para ser competitivos ha de realizar su función bien.

Vemos pues que los requerimientos a los que trata de responder el sistema son muy exigentes.

b) Describe la arquitectura distribuida, los modelos de comunicación y las técnicas de escalado empleadas.

Arquitectura

Aunque en el video se nos muestran diferentes etapas en la arquitectura del proyecto, ya en 2008 se nos muestra una primera fase de lo que posteriormente se desarrollará hasta 2012. Vemos que la propuesta de 2008 que ya está preparada para ser escalada de manera independiente en cada una de sus capas, es la siguiente:

- El servidor de bloques (*blockserver*) y el servidor de ficheros alojado en Amazon S3.
- Un servidor de metadatos (*metaserver*) y una servidor de base de datos MySql. Al que posteriormente se le incorpora Memcache.
- El Notification server que se utiliza para notificar al usuario de cambios que haya podido haber en los ficheros que tiene alojados en el sistema.
- Balanceadores de carga que distribuyen el tráfico entre la pila de *metaserver* de las peticiones que llegaban desde los clientes y desde el *blockserver*

Esta propuesta evoluciona hasta el año 2012 añadiendo más nodos en cada uno de los apartados, pero básicamente se mantiene tal y como se diseño en 2008.

Se observa que la arquitectura distribuida ha separado e independizado, el sistema de ficheros y su servidor de bloques, el acceso a la base de datos y ha creado un servidor de notificaciones cuya labor es la de notificar a los usuarios de los cambios que hayan habido en sus diferentes carpetas y ficheros.

La separación de tareas permite el crecimiento y la distribución de la infraestructura separando las funciones de estas y la infraestructura necesaria para responder a cada una de ellas.

Comunicación

Respecto a los modelos de comunicación empleados en la arquitectura, me gustaría subrayar dos tipos de modelos usados:

- Comunicación síncrona/asíncrona. El modelo trabaja con una comunicación asíncrona, ya que cuando el cliente lanza la petición, el sistema trata esta petición y acepta un compromiso en cuanto a la devolución de la información requerida. Para cada petición, cada uno de los componentes del sistema no va bloqueando el thread que trata la petición hasta que recibe la respuesta, sino que existe un compromiso de respuesta que libera el thread hasta que vuelve a ser necesitado para procesar la respuesta
- Comunicación pull/push y el notification server, envío de notificaciones. Observamos como en le caso de Dropbox, es necesario implementar el tratamiento de la comunicación *push*, es decir el tratamiento de la comunicación desde el servidor hacia el cliente para poder notificar a cada uno de sus clientes de los cambios que han podido suceder en sus carpetas y ficheros durante el tiempo que no han estado conectados.

  El modelo de comunicación *pull/push*, nos indica desde quien y hacia quien se realiza la comunicación. En el caso de la comunicación *pull*, quien inicia la comunicación es el cliente, este tipo de comunicación ocurre en la mayoría de los servicios, tambien en Dropbox. En cuanto a la comunicación *push*, que se implementa en esta arquitectura mediante el *notification server*, la comunicación se da desde el servidor hacia el cliente. El objetivo de la comunicación push en este caso es la de notificar cambios al cliente.

Podemos decir que la comunicación push es más escalable, ya que esta realiza comunicaciones solamente en el caso de que sea necesario, en este caso solamente cuando haya habido un cambio en algún fichero. Al contrario las comunicaciones desde el cliente o *pull* pueden saturar la infraestructura con consultas que no devuelvan ninguna información añadida al cliente.

Técnicas de escalado

En cuanto a la técnica de escalado, subrayar la evolución de la arquitectura de Dropbox es de 1 a n, es decir pasa de ser una arquitectura basada en una única maquina a tener la opción de escalar cada uno de los componentes que forman el sistema manera autónoma y sin limite en cuanto al número, siendo ciertamente el límite su capacidad de inversión.

La técnica usada ha sido la de distribuir o separar cada una de las funciones necesarias para llevar a cabo el objetivo del sistema y añadir elementos que aceleren la respuesta de cada una de las partes. Concretamente el sistema se ha dividido en:

- La gestión de bloques y ficheros, cuya representación en la arquitectura son el *blockserver* y el servidor de ficheros alojado en Amazon S3.
- La gestión de datos, representada en el *metaserver,* un servidor de base de datos *MySql* y *Memcache*, éste último con el objetivo de acelerar la respuesta.
- La comunicación push al cliente mediante el *notification server*

Cada una de estas partes pueden ser escaladas de manera autónoma, es decir, tal y como se vayan produciendo cuellos de botella estos se pueden solucionar añadiendo más nodos justamente en la parte en la que se tenga el problema.

c) Proporciona tu propia perspectiva sobre la arquitectura de Dropbox e intenta encontrar las limitaciones de este modelo. ¿Es factible en 10 años con redes 6G?

Desde mi prespectiva de estudiante, me parece que la arquitectura distribuida que se nos presenta en el video, soluciona las necesidades que Dropbox tenía en 2012, al igual que los modelos de arquitectura anteriores solucionaban las necesidades de años anteriores. Me refiero a que es una arquitectura que ha ido evolucionando para dar respuesta al número de clientes que tenía el servicio.

Esto puede tener mucha lógica ya que en el discurso de Kevin Modzelewski algo que se repite mucho es la relación clientes-arquitectura-empleados, es decir desde el principio las diferentes arquitecturas presentadas vienen contextualizadas al número de clientes y al número de trabajadores, y supongo que a la capacidad de inversión, que la compañía tenia en ese momento.

El modelo de arquitectura es pues un modelo que va creciendo y cambiando de forma y componentes sobre la marcha con el objetivo de dar un buen servicio al número de clientes y a la demanda existentes en el momento con los recursos disponibles.

No estamos pues ante un caso en el que se construye un servicio y una arquitectura que de respuesta a ese servicio ideal o sin limitaciones. Concretamente en el caso expuesto, creo que el esquema del 2012 no varía sustancialmente del planteado en 2007 (con 0 usuarios y 3 empleados), diría que es el mismo esquema llevado a un esquema distribuido y escalable. En la linea de lo comentado anteriormente creo que el esquema inicial ha sido evolucionado para que pueda responder a las necesidades de 2012, pero pienso que se la estructura principal es la de los inicios.

Respecto a si la arquitectura planteada puede ser viable a 10 años y en un contexto con redes 6G, diría que me parece muy complicado, ya que las redes 6G aumentarán de manera exponencial la capacidad de comunicación de los clientes hacia el sistema y como consecuencia de ello el número

de solicitudes a los que tendrá que atender este. Creo que el número de equipos que serían necesarios para poder mantener las exigencias del producto-cliente serían tantas que veo complicado tener una capacidad de inversión tan grande como para poder responder estas necesidades.

El mayor problema que veo en el planteamiento realizado es que en un contexto con técnologias 6G aunque no aumenten mucho el número de usuarios de Dropbox, y con ello sus ingresos, las necesidades derivadas del aumento exponencial de la capacidad de comunicación por parte del cliente y de las peticiones que esto creará pueden poner en cuestión tanto la arquitectura presentada, como la base de crecimiento clientes-arquitectura-empleados que se plantean. En definitiva diría que este modelo no es viable en un entorno con redes 6G.

Referencias

Transparencia

Coulouris, G., Dollimore, J., Kindberg, T., & Blair, G. (2012). 1.5.7 Challenges: Transparency. En Distributed Systems: Concepts and Design (págs. 39-41). Essex: Pearson.

https://es.slideshare.net/lddt/transparencia-51701435

http://sistemas-distribuidos-unerg.blogspot.com/2008/10/caractersticas-principales-de-los.html

Escalabilidad horizontal y vertical

https://es.wikipedia.org/wiki/Escalabilidad

Comunicación síncrona y asincróna

https://searchapparchitecture.techtarget.com/tip/Synchronous-vs-asynchronous-communication-The-differences (Joel Shore)

Modelo Ray

“Ray: A Distributed Framework for Emerging AI Applications”. Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, Ion Stoica University of California, Berkeley

https://ray.io/

Modelo Dropbox

https://www.dropbox.com/es/business/trust/security/architecture