UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

ESTUDIO DE FACTIBILIDAD TÉCNICA DEL USO DE BASES DE DATOS KEY-VALUE-STORE EN SERVIDORES DE NOMBRES DNS

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

DARÍO ANDHAEL PALMA CARRASCO

PROFESOR GUÍA:
JAVIER BUSTOS JIMÉNEZ

MIEMBROS DE LA COMISIÓN:
ALEJANDRO HEVIA ANGULO
CESAR GUERRERO SALDIVIA

SANTIAGO DE CHILE
2020

Resumen

El servicio de resolución de nombres o sistema de nombres de dominio es una columna vertebral del internet. Se utiliza por todo tipo de usuarios, diariamente, y trillones de veces. Uno de los elementos clave son los servidores de nombre DNS quienes mantienen un registro de pares dominio y valores. Este servicio esta bajo una demanda creciente, dado el aumento de servicios en la nube (o Cloud) y creciente número de dispositivos conectados a internet.

El sistema DNS es distribuido sin un único punto de falla. Cada servidor de nombre dueño de un dominio o zona posee registros de la forma llave valor. Por otro lado existen bases de datos distribuidas que también son llave-valor, llamadas Key-Value-Stores. Entonces ¿se podrá utilizar bases de datos KVS para responder consultas DNS? ¿Cómo podemos compararlas?

Para responder estas preguntas se escogieron tres servidores de nombre y tres bases de datos KVS. Se diseño un programa intermedio que permiten a las bases de datos distribuidas responder como un servidor básico DNS. Se creó un nuevo benchmark basado en el estado del arte, que permite medir la capacidad de respuesta, memoria, tiempo de respuesta, uso de procesador y respuesta a fallas.

Utilizando ambas herramientas desarrolladas se hicieron pruebas de desempeño y se descubrió que Redis es la KVS que se acerca más al rendimiento de un servidor DNS, tanto en memoria como en capacidad de respuesta. No obstante su rendimiento tiene bastante espacio para mejorar y acercarse a servidores de nombre tradicionales como Bind.

Para mis abuelos que me acompañan y los que ya no

Agradecimientos

Quisiera agradecer primero que todo a mi profesor guía Javier por la ayuda que me ha dado más allá de la memoria, y por su puesto por la infraestructura a la que me dio acceso para poder llevar a cabo las pruebas. A Nicolás y Juan Pablo quienes me ayudaron pasar por el plan común con risas y alegría. A mis tres padres quienes son un ejemplo para mi y me inspiran a hacer lo correcto. A mis amores que ya no están, pero que me inspiraron a no rendirme durante esos años 2015-2019 que fueron tan difíciles. A mis amigos de siempre, Carlos, Jaime, Sebastián, Francisco, Cristóbal, Pablo, Martín que siempre están ahí y que no me abandonan, sin importar si estoy en Chile, China o Francia.

A Eduardo, Franco, Stefano, Perry, Rodrigo, Lucas, Tomás, Gabriel y mis amigos del DCC con los cuales no perdí el contacto. Todos los docentes del departamento de computación con los que tuve clases, quienes me formaron más que en lo académico.

A la comunidad internacional que me formó en el extranjero y en Chile. A aquellos compatriotas que me mantuvieron a flote en Francia, Bastián, Rubén, Esteban y Felipe, mas mis amigos mexicanos José Sosa, Carlos, Tiffani, Sofía.

Mi equipo del LEIT Darío, Isaías, Victoria, Fernanda y la directora de la biblioteca central Rosa Leal. Muchos más hicieron ameno este largo viaje, desde muchos continentes, cada uno tuvo su rol crucial para que pudiera llegar hasta aquí

Por último quisiera agradecer a Rosa M, a la psicóloga de bienestar estudiantil y a todos aquellos que decidieron olvidar pero que yo nunca olvidaré.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Capítulo 1

Introducción

1.1. Antecedentes generales

Desde 1983 el sistema de nombres de dominio o DNS se ha convertido en un componente critico del internet. Ya en 2012, Google llevaba 3 años de operación con su servicio DNS público, en el cual recibían más de 70 millones de peticiones al día[6]. Esto muestra que el servicio DNS posee bases de datos con mucha demanda, procesando millones de operaciones de lectura y escritura, de forma totalmente distribuida con miles de máquinas. Las tareas principales de estas bases de datos son:

- Mantener un subconjunto de nombres de dominio e IPs con aquellas más utilizadas en su red, y delegar las resoluciones de los demás a otros servidores de internet.
- Asociar direcciones IP (Internet Protocol) con los dominios y sub-dominios para los cuales el servidor DNS tiene autoridad.
- Tener alta disponibilidad y rapidez de respuesta.
- Ser resistente a particiones de red.

1.2. Motivación

Dada la naturaleza de los datos manejados por el sistema DNS, este se puede ver como una base de datos distribuida Key-Value-Store. Este tipo de base de datos NoSQL (No sólo SQL) podría cumplir el mismo rol que las tradicionales, entonces ¿cómo se compararán ambas bases de datos? ¿podremos obtener mejor rendimiento y velocidad de respuesta con una BDD Key Value? ¿cual manejará el particionamiento de la información más eficientemente?

Si se encontrase algún punto de mejora esto impactaría un elemento clave del internet, ya sea en mayor disponibilidad, resistencia ante particiones, etc. Dado que los dos tipos de bases de datos fueron creados en contextos distintos no hay expectativas sobre los resultados, pero claramente los sistemas clásicos ofrecen funcionalidades orientadas a la seguridad que quizás las KVS no posean.

1.3. Objetivos

El objetivo general de esta memoria es diseñar un software que permita comparar el rendimiento de los servidores de nombre DNS con bases de datos distribuidas de tipo Key-Value-Store. Con esto determinar si se pueden utilizar estos últimos para responder peticiones DNS y descubrir si son una opción con ventajas sobre los sistemas actuales.

1.4. Metodología de trabajo

El plan de trabajo definido para alcanzar los objetivos es el siguiente:

1. Estudiar los requerimientos operativos, bases de datos disponible, servidores DNS y escoger 6 en total.
2. Estudiar las herramientas de medición y comparación o *benchmark* disponibles.
3. Recolectar datos de registros DNS para hacer pruebas.
4. Diseñar e implementar un intermediario que permita a las bases de datos Key-Value-Store responder peticiones DNS.
5. Diseñar e implementar un benchmark para realizar pruebas a ambos grupos.
6. Probar el rendimiento en un entorno de pruebas.
7. Escribir el documento de memoria y discutir resultados.

1.5. Organización de la memoria

El informe está organizado en 6 capítulos principales más un glosario y un anexo. Primero la introducción en el capítulo 1.

En el capítulo 2 se estudia el estado del arte, comenzando por los servidores y bases de datos disponibles. Luego si existen herramientas de comparación para cada grupo y una que compare ambos al mismo tiempo. Después los requerimientos operacionales para un servidor DNS.

En el capítulo 3 se presenta la relevancia del problema y los objetivos del trabajo.

En el capítulo 4 se muestra el diseño de la solución, considerando detalles de implementación y de operación. Además de cómo se obtuvieron datos para realizar pruebas de estrés.

En el capítulo 5 se presentan y discuten los resultados de pruebas de estrés con dos conjuntos de datos.

En el capítulo 6 se concluye y comentan las posibles mejoras como trabajo futuro.

Capítulo 2

Estado del Arte

En este capítulo se plasma una investigación bibliográfica realizada acerca de los temas relativos al trabajo, explicando el rol e importancia de cada uno.

2.1. Conceptos de red y DNS

Los servidores DNS funcionan manejando los diferentes niveles de un dominio. En el esquema (2.1) se muestra que el dominio *google.com* se divide en tres niveles. En el mundo existen 13 grupos de servidores DNS a nivel root, que se adjudican la gestión de diferentes dominios a Top-Level. Esto se llama manejar una zona. A continuación se muestra un ejemplo de jerarquía de dominio.

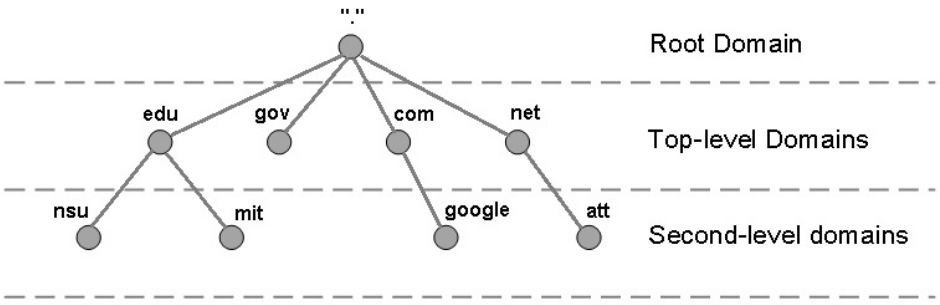

Figura 2.1: Ejemplo de jerarquía de dominios

Los servidores de nombre conocen la respuesta a los dominios de su zona. Una zona en el figura sería por ejemplo *., .edu, .gov*.

Existen servidores DNS a nivel root, TLD y de zonas a niveles más bajos. Además existen servicios de DNS llamados resolvers, quienes se dedican a consultar servidores dueños de zonas o Autoritativos y registran temporalmente los valores. En la figura (2.2) se muestra como un cliente consulta a un Resolver por el dominio *www.google.com*.

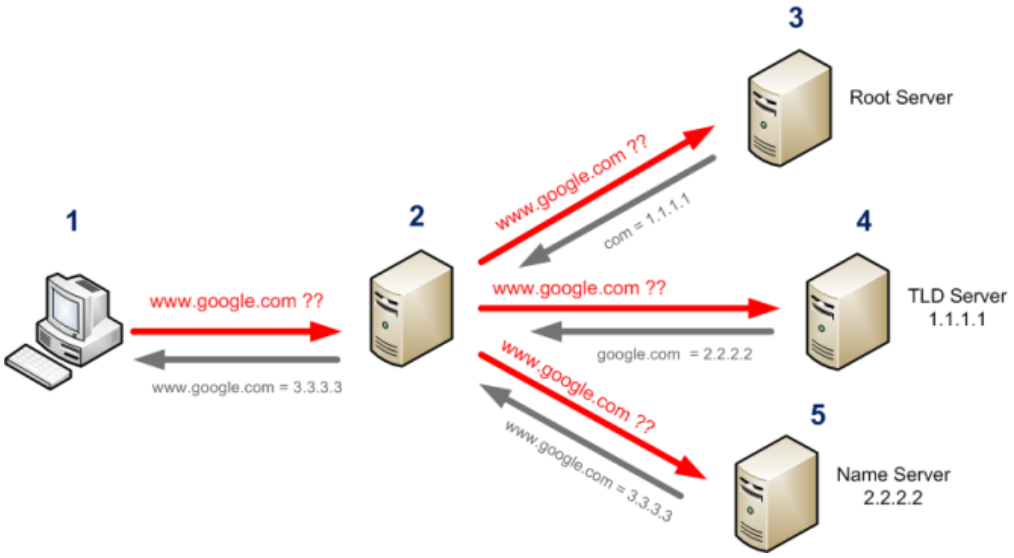

Figura 2.2: Ejemplo de como funciona una consulta recursiva DNS.

Existen servidores de nombre DNS que funcionan como Resolver, servidor Autoritativo o ambos.

DNS funciona principalmente mediante el protocolo de transporte de red UDP. Este protocolo permite comunicación sin crear una conexión entre ambas partes lo cual agiliza el proceso de mensaje-respuesta. También se utiliza TCP pero como alternativa.

2.2. Conceptos: ACID, CAP y BASE

Para entender las diferencias entre las bases de datos tradicionales y las bases de datos sobre sistemas distribuidos es necesario describir cuales son los paradigmas de construcción de una base de datos y conceptos de teoría misma.

El diseño de bases de datos ha progresado mucho durante los últimos 40 años. Theo Harder y Andreas Reuter introdujeron en su artículo sobre el diseño de transacciones para BDD[26] los principios ACID. Estos fueron los fundamentos para asegurar transacciones seguras, y significa lo siguiente:

- Atomicidad: cada modificación a la base de datos debe operar bajo un principio de todo o nada. Esto significa que si una parte de una operación compleja falla entonces toda la operación debe ser abortada.
- Consistencia: sólo se permitirá que los datos que no rompan la consistencia ni las reglas de la BDD. Cada transacción lleva el sistema de un estado consistente a otro estado consistente.
- Aislamiento: las operaciones que se ejecuten en paralelo no pueden tener un impacto una sobre la otra. Esto evita que se lean datos intermedios y creen estados inconsistentes.
- Durabilidad: los cambios registrados sobre el sistema no pueden perderse. Esto se logra mediante mecanismos de registros adicionales, copias de seguridad y cualquier herramienta que permita reconstruir el estado de la base de datos a pesar de fallas de software

o hardware.

Las bases de datos Relacionales fueron el paradigma dominante junto a SQL, hasta que en los años 2000 un nuevo paradigma de diseño e interacción se gestó: las bases de datos no relacionales o no sólo relacionales NoSQL. Sistemas de bases de datos distribuidos fueron creados pero éstos se alejaron de las garantías ACID.

Se creó una nueva caracterización: BASE o Básicamente disponible, Soft State o estado permisivo sin garantías de consistencia y eventualmente consistente. Además para describir el funcionamiento de estos sistemas distribuidos Brewster hizo la conjetura o teorema CAP[37]. Este significa: consistencia, disponibilidad y resistencia a particiones. Ésta postula que sólo pueden escogerse dos de estos tres puntos al diseñar un sistema distribuido.

La descripción de CAP y BASE suele aplicar sobre los sistemas distribuidos, pero cada unidad dentro de éste suele tener seguridad ACID de transacciones.

2.3. Tecnologías: servidores de nombre DNS y bases de datos

En esta sección se revisan servidores de nombre autoritativos. Luego se habla de bases de datos no relacionales, Key-Value-Store, Wide-Column-Store y particularidades sobre Redis, Etcd y Cassandra.

2.3.1. Servidores de nombre DNS

Si bien estos servicios existen desde finales de los ochenta y principios de los noventa hoy en día existen cerca de 13 Servidores de nombre para DNS. Unos proporcionan servicios con soporte para resolución de nombres recursiva, otros para resolución autoritativa y otros para ambos.

Nos interesa estudiar aquellos servidores que tengan autoridad sobre nombres de dominio y que sean Open Source para poder realizar pruebas. Entre estos servidores podemos citar:

1. Bind: la más antigua y con una gran presencia en la internet fue desarrollada por la Universidad de Berkeley, de ahí su nombre Berkeley Internet Name Domain (BIND), y hoy en día es mantenida por la comunidad Open Source y la Internet Systems Consortium[14]. Bind9 es la última versión oficial del servidor de nombre y fue lanzada el año 2000, siendo un gran cambio desde la versión anterior desarrollada en 1997[16]. Dentro de sus capacidades puede mantener autoridad sobre los dominios que gestiona o delegar de forma recursiva las peticiones. Es el servidor DNS más utilizado [38]. En su sitio podemos encontrar todas las normas de la IETF que cumple, y que se necesitan para un servidor robusto[33].
2. Knot: uno de los más recientes. Solo permite actuar como servidor autoritativo, se especializa en carga rápida de zonas[1] [41].

[1]Una zona es una porción de un dominio delegado a una entidad legal, organización o compañía para su mantención. En el área técnica corresponde a una porción de un dominio que permite control granular de

3. Power DNS: permite resolver peticiones con autoridad[2] y de forma recursiva, con manejo de carga por máquina.

4. Unbound y NSD: al primero se le considera más moderno y compacto que Bind dada su implementación modular pero actúa principalmente como un resolver[3]. El segundo es un servidor autoritativo que tiene uso diario en los root server[4]. Ambos desarrollados por NLnet Labs [42]. Es importante indicar que NSD es utililizado en 4 root servers (D,H,L y K)[5].

Existen otros diez, que son menos populares en la industria. Entre ellas YADIFA que se ocupa del TLD[6] *.eu* y también MaraDNS que se utiliza para manejar zonas estáticas pequeñas (para cambiar algún registro hay que reiniciar MaraDNS). Cabe destacar que existe un servidor llamado **Redins**[17] que utiliza *Redis* como base de datos, lo que se tomará en cuenta para en sección (c4.2.1).

Los servidores de nombre almacenan su información mediante archivos conocidos como **master zone files**, descritos en el RFC1035, que son validados antes de su uso para responder consultas.

Power DNS permite utilizar bases de datos para gestionar los archivos y además tienen una implementación con el formato de BIND. La mayoría[22] son BDD SQL y existen guías para poder desarrollar conectores personalizados. La lógica de toma de decisiones y catálogo de nombres de dominio esta separada en el **PacketHandler** lo cual permite tener conexiones independientes a la base de datos. En otras palabras el Backend no piensa. Esto muestra que algunas implementaciones han considerado delegar el manejo de información a bases de datos tradicionales.

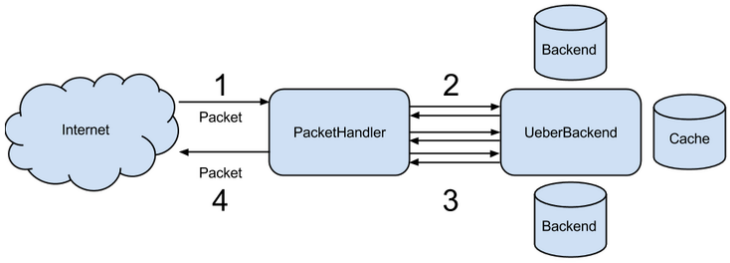

Figura 2.3: Manejo de backend de PowerDNS[44].

respuestas DNS. Por ejemplo una zona puede ser para el dominio ejemplo.cl y sub dominios www.ejemplo.cl, dev.ejemplo.cl, remote.ejemplo.cl el subconjunto [ejemplo.cl,www.ejemplo.cl, dev.ejemplo.cl] .

[2]Servidores de nombre autoritativos: son dueños de una zona y conocen las valores de las respuestas a los dominios de la zona.

[3]Un *Resolver* es un servidor DNS que delega la mayoría de las peticiones a otros servidores que tienen las respuestas. Implementan un cache para guardar datos temporales de las respuestas y así acelerar el tiempo de respuesta a los clientes

[4]Servidor que responde las consultas DNS y redirige a los servidores TLD

[5]https://d.root-servers.org/history.html

[6]Top Level Domain: servidores DNS que se ocupan de resolver consultas inmediatamente inferiores al root server. Por ejemplo: eu, fr, cl.

2.3.2. Key-Value-Stores y bases de datos

Estos sistemas almacenan llaves y valores como dice su nombre. Estos valores pueden ser cualquier cosa, desde estructuras de datos hasta sólo bytes brutos. Las KVS se utilizan principalmente como un cache[7] a nivel de aplicaciones de sistema, como para aplicaciones web. Para esto se privilegia manejar los datos en memoria principal, lo cuál acelera el acceso a los datos. Esto hace que no todos los KVS mantengan alguna forma de persistencia de datos por lo cual no podemos llamarles bases de datos.

Las Key-Value-Store a pesar de intuitivamente ser similares a una tabla de hash pueden implementarse con diferentes enfoques. Un ejemplo de los diferentes enfoques es PebbleDB la que creada como una modificación de HyperLevelDB, y utiliza una estructura de datos llamada Fragmented Log-Structured Merge Trees para mejorar el desempeño en operaciones de escritura [23].

Otros tipos de bases de datos permiten hacer consultas anidadas de llaves y valores pero en principio pueden utilizarse como Key-Value-Stores. Por ejemplo utilizar Cassandra, una base de datos basada en columnas, con una sola columna emularía el comportamiento de una BDD Key Value. Por esto en la sección de solución se incluirán algunas bases de datos que no son estrictamente Key Value.

Actualmente las Key Value Store más populares son[19]:

1. Redis[45]: se auto denomina como data-structure store dado que no sólo almacena valores planos dada una llave, si no que permite realizar operaciones con más complejidad sobre estructuras de datos como listas, conjuntos, entre otros. Se le considera CP por el teorema CAP.
2. Amazon DynamoDB: key value y document value[8] son sus paradigmas de datos. Es de naturaleza distribuida en los servicios web de Amazon. Es de licencia propietaria por lo cual no se podrá utilizar en las pruebas. Se le considera CA por el teorema CAP.
3. Microsoft Azure Cosmos DB: es una base de datos multiparadigma, incluyendo soporte para grafos. Es de uso comercial bajo los servicios de Microsoft Azure. Microsoft permite que la consistencia sea configurable.
4. Memcached: es una Key Value Store que funciona puramente en memoria principal, lo cual no la hace resistente a fallas de sistema. No posee replicación. Es utilizada principalmente para manejar sesiones web.
5. Hazelcast: consiste en una KVS y document value store escrita en Java con un enfoque fuerte en redundancia de datos geográfica.
6. Etcd: es un KVS distribuido que se utiliza en Kubernets[9] y otros servicios. Tiene replicación total entre nodos y soporta consultas simples.

A continuación se revisa con un poco más de detalle tres bases de datos dado que se

[7]Los caches se han utilizado como memoria intermedia de acceso rápido entre el procesador o CPU y la memoria principal RAM[7]

[8]Los document value store utilizan una id para indexar documentos u objetos principalmente en format JSON, los cuales tienen formato libre. Uno de los ejemplos clásicos es MongoDB.

[9]Una plaforma para manejar servicios en contenedores que facilita configuración y automatización, en particular en replicación y escabilidad

utilizan en este trabajo.

Redis como base de datos

Redis es un caché en memoria principal, lo cual lo aleja de la categoría de bases de datos. Sin embargo Redis cuenta con diferentes opciones para asegurar persistencia de datos. En particular dos modos de operación:

- Append Only File: corresponde a un listado de los comandos procesados por Redis, con los cuales se puede reconstruir la información almacenada. Esto se guarda a cada segundo por lo cual datos en esta ventana de tiempo pueden perderse.
- RDB snapshot: es una versión comprimida de los datos de memoria principal. Esto se guarda cada ciertas operaciones y cada cierto tiempo, por ejemplo se comprueba que hayan 1000 inserciones de datos cada 5 minutos y si así fue se crea una copia de la memoria. Esta forma es la que resulta en menos interrupciones de sistema pero también tiene ventanas de tiempo más grandes para que datos se pierdan.

Desde la versión 4 estos dos modos pueden combinarse para recuperar más rápido los datos, con menos memoria en disco y más fiabilidad.

Su sistema de replicación y *clusterización* cuenta con un sistema de maestros que manejan las operaciones de escritura y con un grupo de esclavos que reciben copias periódicas. Si un maestro falla, un esclavo asume su rol en el cluster.

Es importante destacar que los datos son repartidos entre los nodos maestros, por lo cual ningún nodo cuenta con todos los datos.

Etcd

Etcd es una KVS de código abierto, que provee la base para sistemas distribuidos como kubernetes[10]. Tiene las siguientes propiedades[11]:

- Replicación completa: cada nodo cuenta con todos los datos del cluster.
- Consistente: cada operación de lectura entrega la escritura más reciente entre las diferentes instancias.
- Disponibilidad: dada la replicación si un nodo falla, los datos no se pierden.

Sus datos son manejados principalmente en disco por lo cual se recomienda que opere con discos de estado sólido.

Wide-Column-Store y Cassandra

Además de las Key-value-store hay otras bases de datos llamadas wide-column-store que consisten en una llave o columna principal con un número dinámico de columnas secundarias. Por esto se les llama a veces key-value-stores en dos dimensiones[20]. Una de ellas es Cassandra

[10]Sistema de orquestación de contenedores en la nube. Sitio oficial https://kubernetes.io/

[11]Se omiten ciertas propiedades no relevantes.

que fue originalmente creada por Facebook para manejar grandes volumenes de datos en un sistema distribuido [36].

El diseño de Cassandra fue hecho con los siguientes objetivos[12]:

- Replicación de datos con múltiples nodos maestros.
- Crecimiento lineal de la capacidad de operaciones lectura/escritura con cada nodo adicional.
- Manejo en línea de carga.
- Esquema de datos flexible.
- Consultas orientadas a llaves de partición.

Como es una base de datos wide-column su enfoque no es relacional, si no que es ajustado a consultas[10]. Las tablas que se construyen están indexadas por una llave primaria y una llave de particionamiento, que permite repartir los datos entre nodos del cluster. No existen operaciones *JOIN*[13] porque se busca tener que acceder a los datos una vez mediante estas llaves. Esto lleva a un proceso de desnormalización de tablas y a tener que replicar datos. Ésto le permite a Cassandra tener tiempos de lectura rápidos.

2.4. Tecnologías: Sistemas remotos y PCAP

En esta sección en primer lugar se revisa Ansible como orquestador de sistemas. Luego se estudian herramientas para crear y capturar paquetes de red.

2.4.1. Ansible y manejo de sistemas remotos

Ansible es un gestor de configuraciones desarrollado por la compañía Red Hat el cual permite proveer, configurar, hacer pruebas y levantar aplicaciones e infraestructura con sus propias instrucciones o código[2].

Posee varias funcionalidades:

- realizar operaciones sobre sistemas remotos sin necesidad de instalar Ansible en ellos.
- conexión a través de SSH.
- compatibilidad con sistemas Unix y Windows.
- cualquier máquina con Ansible puede orquestar nodos, lo cual se aleja de un enfoque centralizado proclive a fallos.
- integración con contenedores y servicios en la nube (o Cloud).

Los archivos de configuración consisten en texto en formato YAML[14] lo cual les otorga simplicidad. Los archivos de configuración se escriben en texto plano como se muestra[2] a continuación:

[12]Se omiten todos los objetivos y se muestran los más relevantes.

[13]Una operación JOIN en base de datos relacionales corresponde a unir dos tablas dada la correspondencia de valores en ambas. Por ejemplo una tabla usuario que contiene nombre y cumpleaños, a una tabla departamento que contiene dueño y precio. Estas tablas pueden unir nombre del usuario con dueño de departamento.

[14]YAML Ain't Markup Language es un estándar de escritura de datos enfocado en legibilidad.

```
[webservers]
www1.example.com
www2.example.com

[dbservers]
db0.example.com
db1.example.com
```

Existe una gran cantidad de módulos que realizan tareas sobre servicios específicos. Por ejemplo el módulo **unarchive** utiliza los programas que tiene el sistema para comprimir y descomprimir archivos sin necesidad de configuración manual, así el usuario se enfoca sólo en la tarea y parámetros particulares.

Los usos principales en la industria para un orquestador son:

- Proveer sistemas a máquinas remotas.
- Manejar configuraciones de sistemas existentes.
- Despliegue de aplicaciones.
- Automatización y seguridad

Su competencia directa es Puppet en el mundo de código abierto, el cual tiene una pequeña ventaja en el mercado de gestionadores de configuraciones, 19 % Ansible contra 22 % de Puppet. Es una de las herramientas con más experiencia pues comenzaron a operar el 2005. Según la consultora de seguridad informática Upguard[47] Puppet se volvió lenta y complaciente, intentando que los usuarios utilizaran licencias comerciales de soporte, lo cual creó un espacio para que Red Hat con Ansible pudieran llevarse una porción del mercado significativa, del 26 % en un momento. Esto gracias a la flexibilidad de Ansible y el soporte rápido que entregó Red Hat.

2.4.2. PCAP y herramientas para paquetes de red

Packet Capture, o captura de paquetes abreviado PCAP es una API que captura paquetes de red en las capas 2 a 7 del modelo OSI[15]. En otras palabras permite capturar desde la capa de datos, la capa anterior a la física, hasta la capa de aplicación.

Esta herramienta se utiliza principalmente para monitoreo de redes y en algunos casos maliciosos captura de información. Con un grupo de paquetes capturado y una herramienta de análisis se pueden diagnosticar problemas de red, identificar ataques o amenazas, uso de banda ancha, etc.

Cada sistema de captura, y sistema operativo cuenta con diferentes tipos de archivos y librerías de código[35]:

- Pcap: el formato estándar de captura de paquetes.

[15]Open Systems Interconnection model es el estándar de comunicación por computadoras y telecomunicaciones. Cuenta con sus respectivas normas ISO/EIC-7498

- Libpcap: una librería de código abierto escrita en C/C++ diseñada para sistemas Unix como Linux y MacOS.
- Winpcap: otra librería y formato diseñada en particular para sistemas Windows, pero descontinuada.
- Pcapng: formato de *nueva generación* que permite añadir metadatos a los paquetes como comentarios, tiempo de captura, estadísticas, etc. Al ser nuevo tiene problemas de compatibilidad con otras herramientas.
- Npcap: la librería para Windows que reemplaza a Winpcap.

A continuación se presentan tres herramientas con sus diferentes usos.

WireShark

Wireshark es el programa para capturar y analizar tráfico de red más popular del mundo y soporta los paquetes mencionados en la sección anterior tanto para sistemas Unix como Windows. Es utilizado por agencias gubernamentales, empresas a nivel comercial y no comercial y organismo educacionales. Posee herramientas de visualización integradas además de una línea de comandos. Su desarrollo funciona en base a contribuciones de expertos en redes computacionales de todo el mundo y es la continuación de un proyecto de 1998 llamado Ethereal por Gerald Combs[48].

El software cuenta con una licencia GNU General Public License version 2 o bien GPLv2, lo cual permite su uso no comercial (no como parte de un producto).

Scapy

Scapy es una herramienta de manipulación de paquetes de red originalmente escrita en Python por Philippe Biondi[4]. Sus principales usos son la creación, falsificación y decodificación de paquetes. Además permite enviar, capturar, escanear y emparejar paquetes. Tiene una interfaz con la librería libpcap y se conecta con varios programas externos, entre ellos Python, Gnuplot, Graphviz, etc.

Antes de Scapy la manipulación de diferentes capas debía realizarse a encadenando múltiples programas, lo cual hacía difícil y limitada forjar paquetes[5]. Unificando estas funcionalidades se pueden crear rápidamente un conjunto de paquetes sintéticos.

Tcpreplay

Tcpreplay es un conjunto de utilidades para Unix y Windows para editar y reenviar tráfico de red capturado previamente mediante herramientas como Wireshark o creados por otras como Scapy. En particular permite clasificar el tráfico como cliente o servidor, reescribir las capas 2 a 4 del modelo OSI y enviar el tráfico por la red a dispositivos como switches, routers, firewalls. Es utilizada por numerosos firewalls a nivel empresarial, educacional y de investigación[46].

2.4.3. Golang y goroutines

Go o Golang es un lenguaje de programación desarrollado por Google, de código abierto. Al momento de ser diseñado, lenguajes como Java y C++ eran los más usados para escribir aplicaciones para servidores en Google. Estos incurrían en mucha repetición por lo que sus ingenieros migraron a lenguajes más fluidos como Python, con un costo en eficiencia y seguridad de ejecución. Go fue concebido para unir lo mejor de ambos mundos.[24]

Fue diseñado con lo siguientes principios:

- Concurrencia con sub rutinas y canales.
- Revisión estática de tipos, durante la compilación.
- Integración de interfaces para tipos.
- Declaración una vez, sin archivos de cabecera (header files).
- Recolección de basura.
- Simplicidad de escritura.

Concurrencia y la programación con múltiples núcleos tiene una reputación de ser difícil. Esto puede que sea por los enfoques que lenguajes como C le dio a herramientas como los *mutex*, manejo de memoria, *semáforos* y los problemas asociados como *Data Races*, *Memory Leaks*, etc. Go soluciona esto proporcionando Procesos de Comunicación Secuencial o CSP en inglés. A este le llaman **Goroutines**[16].

Las goroutines otorgan concurrencia simplificada y mediante el uso de *canales* pueden enviar datos entre sí y el hilo principal. Estas rutinas no son hilos de ejecución adicionales, sin embargo pueden distribuirse entre diferentes hilos, lo cual permite que sean más baratas en términos de uso de memoria (uso del stack) que los pthreads. Los canales estan diseñados para evitar *Data Races* al usar memoria compartida y las rutinas son manejadas de forma transparente para evitar bloquear el programa principal. Por último la memoria de cada rutina es dinámica, comenzando con unos pocos Kilobytes y adaptándose en base a la necesidad.

2.5. Benchmarks y medición

A continuación se presentan benchmark para servidores DNS y BDDs distribuidas, donde se utilizaron principalmente como métricas los tiempos de respuesta de las bases de datos y los servidores DNS. En particular para los sistemas distribuidos la métrica utilizada fue operaciones por segundo que se realizan concurrentemente y su latencia en milisegundos.

2.5.1. Knot Benchmarking Projet para DNS

Diferentes servidores han sido diseñados como competidores de BIND, entre ellos Knot DNS, un servidor sólo de respuestas autoritativas. Es Knot quien desarrolló un proyecto para hacer pruebas: Knot DNS Benchmarking Projet[40].

[16]https://golangbot.com/goroutines/

Este benchmark está escrito en Bash y cuenta con diferentes módulos de pruebas. El módulo principal mide tiempo de respuesta utilizando tcpreplay para enviar diferentes volúmenes de consultas. Esto se hace utilizando al menos dos servidores, uno que actúa como cliente y otro que responde como servidor DNS (DUT o Device Under Test). Su diseño está orientado a sobrecargar con consultas al DUT para llevar a sus límites la configuración de red y la máquina en si.

Cuenta con diferentes herramientas para crear conjuntos de datos sintéticos. Estos corresponden a archivos de configuración para Knot, Bind, NSD y PowerDNS además de diferentes archivos de zonas DNS. Estos últimos son zonas a nivel de TLD, zonas con DNSSEC, zonas no TLD y diferentes combinaciones cambiando volumen de Resource Records[17].

Además realiza un análisis de red automático para reescribir los paquetes PCAP generados y que lleguen al servidor de destino.

Los resultados publicados[18] fueron favorables para Knot posicionándolo casi siempre en los primeros dos lugares cuando la carga de consultas por segundo era muy alta.

La herramienta necesita que los servidores DNS sean compilados por cada prueba, y no provee información para determinar cuales son los requisitos de cada servidor, ni los paquetes a nivel sistema operativo para utilizar el benchmark. Además dado que el código está escrito mayoritariamente en Bash Script su legibilidad, alcance de variables y requisitos de funcionamiento requieren mucho esfuerzo para entender.

2.5.2. Benchmarks para bases de datos distribuidas

En la literatura se han realizado diferentes pruebas a sistemas distribuidos y a key-value-stores. Inicialmente para las bases de datos ACID se realizaron pruebas para comparar sistemas con benchmarks como por ejemplo las auditadas por **TPC**[19]. Este consiste en una prueba lineal de escrituras y lecturas con múltiples transacciones, pero para la gran mayoría de las BDD KV la linealidad de las operaciones no está garantizada dada la naturaleza distribuida de los sistemas en si.

Como no existía un sistema de comparación uniforme entre los diferentes servicios en la nube que utilizaban bases de datos sin transacciones ACID (en su lugar **BASE**) en Duke University se diseñó una nueva metodología (o Framework) de prueba llamada YCSB[18] (Yahoo Cloud Serving Benchmark). Este rápidamente se convirtió en el estándar de prueba con datos sintéticos generados con diferentes distribuciones. EL 2017 Edwin et al.[8] desarrollaron un sistema para simular operaciones reales mediante registros históricos y mostraron que las pruebas anteriores generaban sobre estimaciones de rendimiento de un 15 %. Discuten en su publicación que durante la generación de datos sintéticos para hacer pruebas se produce una mayor localidad temporal de estos (datos), lo que se traduce en una mejora de

[17] Un Resource Record corresponde a un registro de datos DNS, que posee la llave o dominio, seguida de sus valores como TTL, IP (si aplica), tipo de registro, clase de registro, entre otros.

[18] Publicados en https://www.knot-dns.cz/benchmark/

[19] Transaction Performance Processing Counsel es una organización compuesta por diferentes compañías tecnológicas que se unieron originalmente en el esfuerzo de crear pruebas uniformes de rendimiento para bases de datos, sistemas de información, etc.

rendimiento gracias al cache de las bases de datos. Así vemos que es importante probar con cargas de peticiones reales de datos.

YCSB propone al momento de diseñar las pruebas para los sistemas distribuidos que existen 4 niveles de variables a medir:

1. Rendimiento: como cambia el tiempo de respuesta a medida que la base de datos es sometida a cantidades de estrés en aumento.
2. Escalabilidad: el impacto de agregar más máquinas al sistema distribuido.
3. Disponibilidad: como cambian la respuestas y su tiempo cuando se presentan fallos en las máquinas, los cuales pueden ser de red, hardware, o del sistema mismo.
4. Replicación: evaluar la consistencia de las respuestas y el estado de la base de datos.

El *Framework* mismo posee soluciones para los dos primeros puntos, pero propone formas de realizar pruebas para el punto 3 y el punto 4.

Del diseño del framework YSCB se pueden ver las siguientes decisiones: existe un número fijo de hilos de ejecución dedicados a mantener una conexión estable con la base de datos mediante TCP[20] en una capa de interfaz a la base de datos y la carga de peticiones se reparte entre hilos adicionales desde la capa de carga.

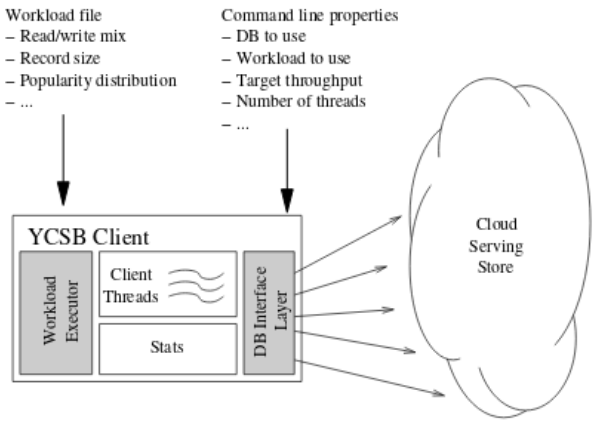

Figura 2.4: Arquitectura del cliente del framework YCSB[18]

2.6. Requerimientos para Servidores DNS

La Internet Systems Consortium mantiene una lista[15] de todos los RFC relevantes para los servidores DNS. Estos describen desde los tipos de datos a gestionar, sugerencias de operación y diseño, requisitos de seguridad, administración de zonas, entre otras.

[20]Transmission Control Protocol: protocolo de red para establecer conexiones estables.

2.6.1. Operación

La principal operación que ejecutan los DNS es entregar información sobre sus zonas, en particular sus registros. Existen diferentes problemáticas asociadas como el actualizar valores, comprobar la veracidad de éstos, transferir zonas. Estos han sido descritos en los RFC 4986, 4641, 4471, 4033 para DNSSEC[21], 1995 y 5936 para transferencia de zonas IXFR y AXFR y RFC 2136 para DNS Update.

Se permite realizar operaciones automáticas de actualización de información mediante la petición DNS UPDATE[29]. Sin embargo para que esto funcione la IP del cliente debe tener permisos para la zona deseada y en algunos casos tener un Token de autenticación que es evaluado en una capa adicional de seguridad, la ACL (Access Control Layer) dependiendo del servidor.

Se recalca que aún cuando DNS es distribuido, según el RFC1035 las operaciones de actualización para un registro particular, de lectura concurrente, entre otras deben ocurrir con aislamiento para un mismo servidor DNS. Es decir se respetan ciertas propiedades ACID en los servidores individualmente.

2.6.2. Arquitectura

Según la implementación recomendada por el RFC1035[34] las bases de datos de servidores DNS debiesen poseer la siguiente estructura:

- Un catálogo con la lista de zonas disponibles y un puntero a la estructura de datos que posea los datos respectivos.
- Estructuras de datos para cada zona por cada *name server*.
- Un cache o múltiples caches para cada clase de RR (Resource Record).

Las estructuras sugeridas para el catálogo y las zonas son árboles con los nombres de cada dominio. En particular se considera que el catálogo será principalmente estático y manejado por el administrador del sistema de forma manual[28]. Dado que se trata de árboles con punteros al archivo con la información de la zona, realizar actualizaciones de los registros se puede hacer simplemente cambiando la dirección del puntero.

2.6.3. Resource Records

Originalmente el RFC1035[34] define 16 Resource Record Types, de éstos los más relevantes son:

- A, dirección IP.
- NS, nombre de un servidor autoritativo.
- CNAME, nombre canónico o alias de dominio.
- SOA, marca el inicio de autoridad sobre una zona.

[21] Son cerca de 10 RFC sobre operación, validación, prevención de ataques, etc. Se muestran los principales.

- PTR, apunta a un nombre de dominio.
- HINFO, información de una máquina.
- MX, dominio de Mail Exchange o servidor de correo.
- TXT, texto plano multi propósito.

Posteriormente se agregó el RR AAAA para direcciones IPv6, DNAME para manejo de alias, entre otros. Estos son los más utilizados a nivel usuario final.

2.6.4. Seguridad

DNS por si solo no es seguro pues la seguridad no fue una consideración al momento de su concepción[27]. La única forma de saber si la respuesta viene del servidor dueño de la zona es verificar la dirección IP, lo cual con herramientas como Scapy u otras es fácilmente falsificado. Por esto ingenieros de la IETF crearon las extensiones de seguridad para DNS.

A grandes rasgos DNSSEC utiliza firmas digitales que están basadas en criptografía asimétrica, es decir, con una llave pública y una llave privada. Sin detallar el protocolo en si las extensiones agregan:

1. Verificación de la autenticidad de los datos.
2. Protección de integridad de datos que permite a los resolver saber si la respuesta recibida fue modificada en el trayecto.

Existen dos partes importantes en el proceso, la firma de las zonas por parte de los servidores y la verificación por parte del cliente, el cual usualmente es un servidor DNS Resolver que verifica la integridad de los datos recibidos los entrega al usuario final.

Capítulo 3

Problema

En este capítulo se presentan los objetivos de la memoria. Primero se presentan estadísticas sobre la demanda del servicio DNS, luego una breve caracterización de DNS como base de datos distribuida para finalmente introducir una oportunidad de explorar soluciones alternativas y los objetivos de la memoria.

3.1. Demanda de dominios y tráfico DNS

En Chile la cantidad de dominios inscritos se ha duplicado en los últimos 10 años. En el siguiente gráfico proporcionado por NIC Chile se muestra la tendencia de crecimiento:

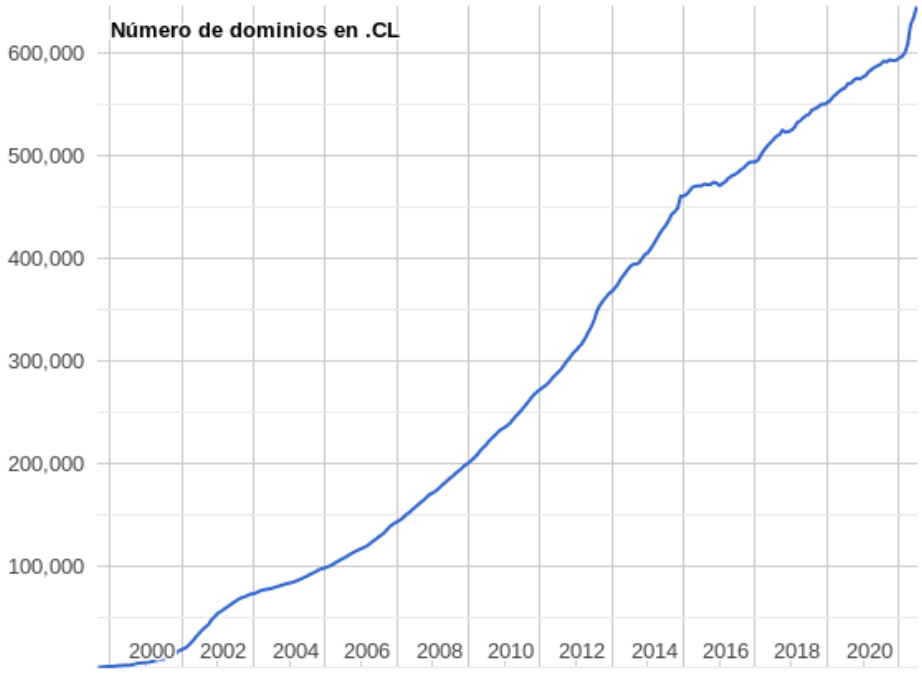

Figura 3.1: Crecimiento mensual del número de nombres de dominio en .CL, desde septiembre de 1997[11]

De éstos el año 2017 sólo un 77 % contaba con un servidor de nombre asociado, cerca de 417.000 dominios. De éstos mismos sólo un 0,02 % contaba con una configuración de DNSSEC correcta[12]. Estas estadísticas son a nivel de Top Level Domain.

A más alto nivel, es decir los servidores Root se puede ver una situación global del tráfico[1].

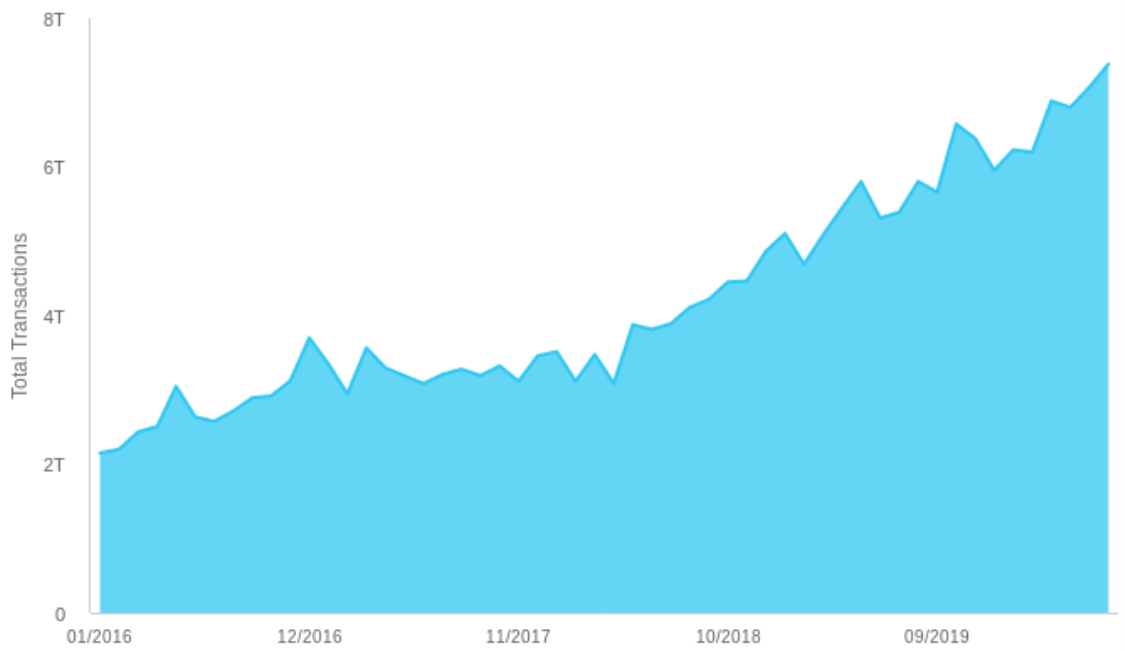

Figura 3.2: Transacciones DNS, consultas y respuestas, a nivel de servidores root desde 2016 a inicios de 2020.

[1]https://www.akamai.com/es/es/why-akamai/dns-trends-and-traffic.jsp

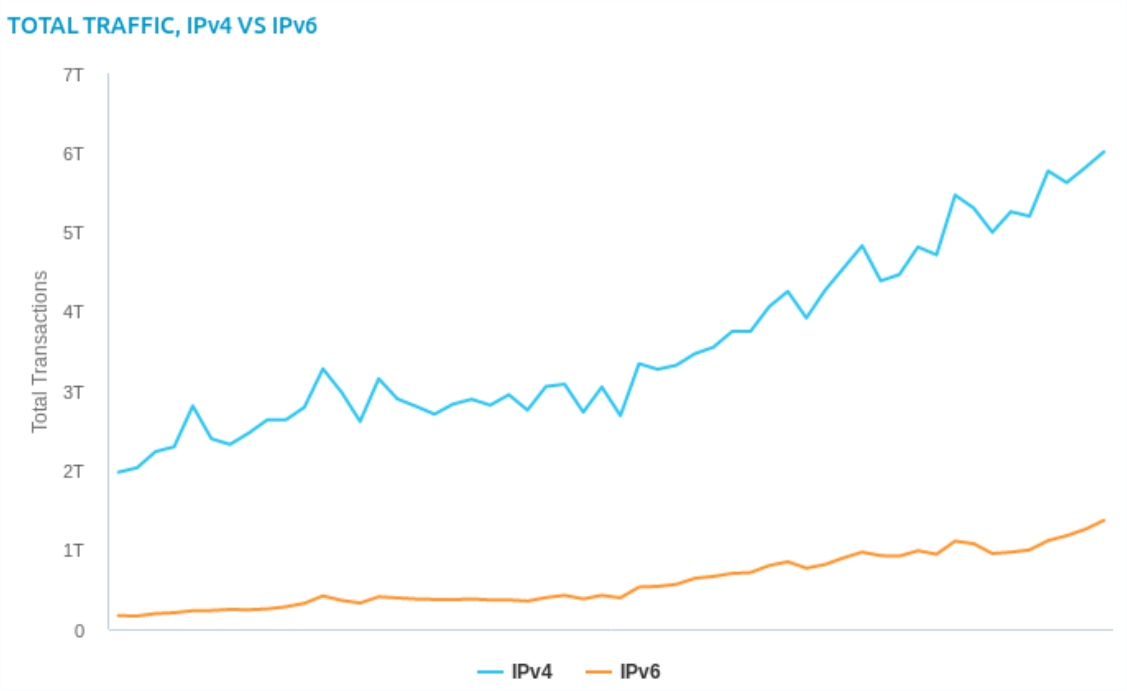

Figura 3.3: Comparación de tráfico por protocolo IP a nivel root server.

En la figura (3.2) se ve que para el conjunto de 9 servidores root el tráfico aumentó en un 200 % en 5 años, a cerca de 8 trillones de paquetes. Y como se ve en la figura (3.3) IPv6 aún no ha tomado un papel protagónico en el internet.

Akamai muestra[1] que el promedio de **consultas diarias es de 13.3 billones** por grupo de servidores DNS[2]. Esto toma en cuenta redundancia geográfica y múltiples máquinas.

Se podría estimar que el volumen de consultas es aún mayor al agregar los servicios de resolver que son quienes consultan directamente a los servidores indicados. Se ve que manejar un gran número de consultas es relevante para este servicio que actúa como columna vertebral del internet.

[2]Los servidores DNS a nivel Root conforman 13 grupos designados por letras del alfabeto, A,B,C, etc. Diferentes organizaciones como universidades, centros de investigación mantienen y coordinan las responsabilidades de cada grupo.

3.2. DNS, distribuido y llave-valor

El sistema de nombres de dominio es jerárquico, descentralizado y distribuido lo cual le permite ser resistente a ataques de denegación de servicio, particiones de red, y fallas de máquinas individuales. Podemos modelar este sistema como un árbol de nodos que delegan sus sub-dominios a otros servidores. Cada dominio representa a su vez una llave y su valor puede ser, texto, otro dominio, una dirección IP, etc.

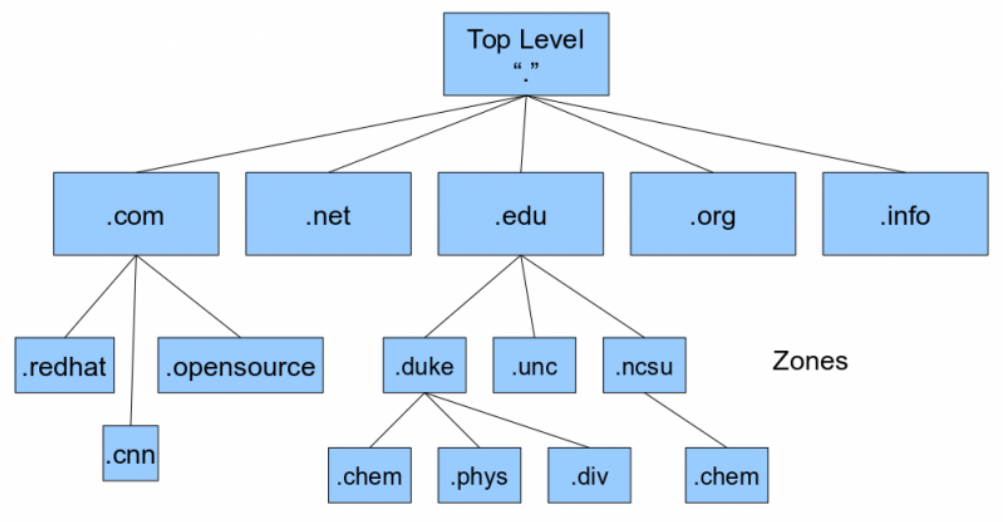

Figura 3.4: Representación del sistema DNS.

Considerando DNS una gran base de datos distribuida, con un modelo llave valor, se puede preguntar si utilizar bases de datos Key-Value-Store o similares para responder peticiones es factible. Como se vio en el estado del arte, algunos servidores de nombre permiten utilizar bases de datos para almacenar los registros RR, ¿será posible comparar ambos sistemas para llegar a uno más confiable o más rápido?

Como las bases de datos key-value-store por si solas no delegan peticiones a otros servicios, éstas son comparables a los servidores DNS autoritativos que responden con valores o errores (NXDOMAIN, REFUSED, entre otros).

3.3. Objetivo

3.3.1. Objetivo General

El objetivo general para este trabajo es comparar el rendimiento en cuanto a manejo de consultas por segundo de los servidores de nombre de dominio tradicionales contra las bases de datos distribuidas de forma Key-Value-Store. Esto para determinar si se pueden utilizar estos últimos para responder peticiones DNS y descubrir si son una opción con ventajas sobre los sistemas actuales.

3.3.2. Objetivos Específicos

En particular el objeto de estudio serán los servidores autoritativos y bases de datos distribuidas NoSQL con comportamiento Key-Value-Store. Las funcionalidades de respuesta recursivas[3] ni DNSSEC serán objeto de estudio.

Para esto se estudiará lo siguiente:

1. Estudiar y definir cuales son los requerimientos básicos de una base de datos para las peticiones de DNS.
2. Estudiar la estructura de datos base que posee cada servidor/BDD y comparar complejidad.
3. Determinar cuales son los rangos operacionales de consultas tratadas por un servidor para definir cuales serán las cargas de datos a consultar y modificar en las pruebas de estrés.
4. Diseñar y realizar pruebas con un pequeño sistema distribuido para estudiar los comportamientos en una red.
5. Medir y comparar los desempeños de ambas categorías.
6. Determinar si es factible utilizar bases de datos key-value como remplazo a sistemas DNS tradicionales basado en los requerimientos determinados anteriormente.

[3]Cuando el servidor DNS no posee autoridad para responder sobre un dominio en particular se recurre a otros servidores, que pueden estar especificados en los registros del servidor o pueden ser delegados a un *resolver*

Capítulo 4

Solución

En este capítulo se presenta la solución encontrada para cumplir con los objetivos del capítulo 3. Así mismo se presentan los diferentes problemas y soluciones encontradas para realizar la comparación de sistemas. La solución tiene distintos componentes por lo que se entregan secciones en este capítulo para la selección de base de datos y servidores DNS, interacción de KVS con consultas DNS, obtención de datos para realizar pruebas y finalmente creación de un nuevo benchmark.

4.1. Bases de datos y servidores a considerar

Respecto a los servidores DNS se decidió optar por los más populares por lo cual se incluye a Bind, Knot y NSD. A continuación se presenta la elección de KVS, notas sobre las estructuras de datos principales de cada programa y sobre su lenguaje de programación.

4.1.1. Bases de datos Key Value Store

Para seleccionar bases de datos se necesita que posean una estructura nativa de replicación de datos, ya sea por clusterización parcial o completa, una API por red para interactuar con la base de datos y persistencia. A continuación se enumeran las bases escogidas:

1. Redis: al ser la más popular posee clientes en muchos lenguajes de programación, una API bien documentada, soporte para diferentes estructuras de datos de forma nativa y diferentes formas de persistencia de datos. Posee repliación parcial entre nodos. Funciona en memoria principal.
2. Apache Cassandra: replicación, distribución de llaves, confiabilidad ante fallos, libertad de esquema de datos, replicación parcial, popular y con buena documentación. Obtiene una porción configurable de la memoria principal disponible para operar y guarda sus datos a memoria secundaria.
3. Etcd: escrita en Go, replicación total de datos, con un modelo de datos simple y utiliza el protocolo Raft[43] para entregar consistencia a los datos escritos. Permite hacer copias del estado actual o snapshots y utiliza una combinación de las memorias disponibles.

Las bases de datos escogidas proporcionan suficiente variedad de consistencia eventual (o no), facilidad de particionamiento o sharding[1] en sistemas distribuidos y si permiten funcionar en memoria principal o no.

4.1.2. Estructuras de datos

A continuación se revisan las estructuras de datos que utilizan los servidores de nombre seleccionados y las bases de datos distribuidas (a nivel de nodo local). De ser posible se menciona la complejidad de su estructura de datos principal.

En particular para los servidores de nombre se revisa la estructura para almacenar las zonas que gestionan (otras estructuras se describen en la sección 2.6.2).

Bind y NSD: Black Red Trees

Estos servidores utilizan una estructura de datos llamada **Red Black Tree**[39][2] (NSD también puede utilizar Radix Trees[3]). Un RBT es un árbol de búsqueda binaria donde la mayoría de las operaciones toman un orden de $O(h)$ donde h es la altura del árbol. En un árbol normal desbalanceado la peor altura podría ser n y entonces las operaciones $O(n)$. Por esto un Red black tree se encarga de realizar un operaciones de rotación para mantener una altura de $log(n)$. Así logra realizar sus operaciones de búsqueda, inserción y borrado con un costo de $O(log(n))$.

Knot y Tries

Knot utiliza un Trie para albergar sus zonas y responder consultas[4]. Un Trie es una estructura de datos que se utiliza principalmente para almacenar texto, encontrar prefijos, entre otros, pero su aplicación general es almacenar llaves de cualquier tipo. El costo de búsqueda e inserción son $O(M)$ donde largo de la llave M sería dominio en nuestro caso.

Cassandra Memtables y SSTables

Cassandra mantiene sus datos en memoria principal en una Memtable, que funciona como un cache que guarda sus datos como filas y cada fila con sus datos de columna (RowColumnData). Estos objetos funcionan como un grupo de llave/valor pero como Cassandra permite tener columnas que almacenan colecciones (listas, tablas de hash, conjuntos), las operaciones de inserción y búsqueda no son siempre de orden constante.

Cuando existe una cantidad de operaciones de escritura, un llamado a sincronizar o la memoria se esta acabando Cassandra almacena sus datos en una Sorted Strings Table (SSTable). Este archivo es inmutable hacia el pasado, sólo se pueden agregar datos.

[1]Sharding es el proceso de realizar particionamiento horizontal de una base de datos, por ejemplo todas las filas con id mayor a cinco mil son trasladadas a otra instancia de la BDD.

[2]Bind define su base de datos en https://gitlab.isc.org/isc-projects/bind9/-/blob/main/lib/dns/rbtdb.c como explica Tony Finch en https://www.dns.cam.ac.uk/news/2018-01-12-bind-bug.html que rbtdb significa **Red Black Tree Database**

[3]NSD define sus usos de estructuras de datos en https://github.com/NLnetLabs/nsd/blob/master/namedb.h

[4]En particular el código fuente lo define en https://gitlab.nic.cz/knot/knot-dns/-/blob/master/src/knot/zone/zonedb.h#L107

Etcd y B+Trees

Etcd utiliza un B+Tree o un árbol de m nodos con persistencia en disco[3]. Esta estructura de datos consiste en un árbol donde cada nodo tiene dos o más hijos, y los nodos contienen sólo llaves, no valores. Para llegar a un valor se busca un llave, se desciende por el árbol hasta llegar al nivel más profundo, donde se encuentran.

Los B+Trees permiten recuperar rápidamente los valores almacenados en memoria secundaria, dado que se reducen las operaciones de acceso a ella. Además las llaves en cada nodo están ordenadas, lo cual Etcd utiliza para almacenar el número de versión de sus Keys a nivel base de datos.

En términos de complejidad un B+Tree, en promedio es de O(logn) para búsqueda, inserción y borrado.

Redis y Hash Table

Redis significa *REmote DIctionary Server* pues utiliza una tabla de Hash para acceder a objetos intermedios llamados **robj**[5]. Los Robj contienen a su vez un puntero hacia su valor que puede ser otra estructura de datos como una lista, un conjunto, una tabla de hash, y las demás que Redis aloja. Por esto las operaciones de búsqueda a primer nivel en Redis son de orden constante $O(1)$ mientras no existan colisiones para las llaves. Dependiendo de la sub estructura de datos del robj se agrega un costo adicional.

4.1.3. Nota sobre las implementaciones

Es importante mencionar los lenguajes de programación en los cuales están implementados los servidores de nombre y las bases de datos:

- Redis, Bind, Knot y NSD están escritos en C.
- Cassandra está escrita en Java.
- Etcd está escrito en Go/Golang.

C es un lenguaje de programación simple que fue creado en 1972. Es parte del código del kernel[6] de Linux, de bases de datos como MySQL, PostgreSQL, Oracle Database y MS SQL Server.

Isaac Gouy en Debian[25] desarrollo diferentes benchmarks para comparar lenguajes de programación con pruebas sobre estructuras de datos, algoritmos, entre otros. Comenta que la mejor forma de comparar rendimiento de lenguajes es con aplicaciones reales[7] pues otros benchmarks con programas simples se prestan a pequeñas optimizaciones por lenguaje de programación. no obstante creó su benchmark para entregar **nociones** sobre las diferencias en rendimiento entre lenguajes.

[5]Se explica principalmente en el README.md https://github.com/redis/redis, pero se puede ver directamente en https://github.com/redis/redis/blob/11cd983d58199b6ac7fa54049734457bd767a0b5/src/db.c#L55

[6]Programa que maneja las tareas del sistema operativo, dispositivos, etc. Es el núcleo, o la base del mismo.

[7]Ver en Why Toy Programs? en https://benchmarksgame-team.pages.debian.net/benchmarksgame/why-measure-toy-benchmark-programs.html

En particular desde las pruebas del benchmark de Isaac Gouy se rescata que Go es más lento que C, Go es más rápido que Java y Java más lento que C++, donde se compara Java con C++ por ambos ser orientado a objetos. Existen además comparaciones en el uso de memoria, Java utiliza por ejemplo más del doble que C++ y por lo general más que Go. C se mantiene como el que utiliza menos memoria entre los 3. Esto puede entregar una noción del rendimiento esperado de los servidores de nombre y KVS escogidas.

4.2. Key-Value-Stores como servidor DNS

4.2.1. ¿Cómo responder peticiones DNS?

Dado que las bases de datos no poseen una interfaz que responda a peticiones DNS directamente es necesario escribir un módulo de comunicación por cada servidor DNS para cada base de datos, o bien crear un cliente común que implemente ciertas funcionalidades de un servidor DNS. Las ventajas y desventajas de cada una de las dos alternativas:

1. Crear módulos intermediarios entre servidores DNS y BDD:
   - Permite aprovechar la implementación original.
   - Al utilizar el código fuente tenemos no hay que re-implementar todas las recomendaciones de los RFC.
   - Requiere un estudio y comprensión profunda del código fuente.
   - El resultado permite realizar **actualizaciones por zonas**
2. Crear un cliente con implementaciones parciales DNS que actúe como receptor de peticiones DNS y capa de conexión a las bases de datos.
   - Da libertad de diseño e implementación.
   - Requiere implementar lo mínimo para realizar mediciones y pruebas.
   - La robustez depende del tiempo disponible para implementar funcionalidades.
   - Es necesario escoger cuales RFC implementar.
   - Permite hacer **actualizaciones individuales** de RRs por nombre en lugar de archivos de zona completos.

Para poder realizar el punto 1 dados los servidores escogidos y el código fuente disponible existe una complicación mayor, toda la lógica de selección de nombres, catálogo, zonas y manejo de archivos maestros no está separada para manejar un *backend*. En otras palabras la arquitectura fue pensada para manejar (en lo posible) toda la información en memoria primaria, sin necesidad de perder tiempo conectándose a una base de datos, minimizando además el tiempo de búsqueda en disco. De elegir esta implementación es necesario realizar cambios profundos en el código. Existen soluciones para BIND como DLZ[21] que permiten gestionar zonas con bases de datos, pero en particular DLZ es relativamente antigua y sin mantención.

En el caso del punto 2 se debe hacer una implementación uniforme para ciertos RFC, con manejo de múltiples hilos de ejecución, conexiones a clientes remotos, a las bases de datos en la nube, entre otros. De elegir esta implementación no será posible implementar todas las exigencias operacionales de la IETF relacionadas a seguridad[30][31][32], lo cual podría otorgar una ventaja en tiempo de respuesta.

Se decide entonces crear un cliente pues cumple con las condiciones requeridas para este trabajo y disminuye la complejidad de implementación.

4.2.2. Prueba preliminar con NodeJS y Redis

Se creó un cliente para Redis con NodeJS, Docker y los módulos de NPM *native-dns, dnsd*. Dnsd permite codificar y decodificar mensajes DNS, además de escuchar de forma asíncrona en un puerto específico conexiones UDP. Native-dns permite hacer consultas DNS a un *resolver* remoto.

Por cada petición ejecutada con la herramienta de red *Dig* el servidor DNS recibe la petición, la decodifica y pide una conexión a Redis para completar la consulta, en caso de una respuesta negativa traspasa la pregunta a un *resolver* (en particular el *resolver* de Google de IP 8.8.8.8).

A diferencia de esfuerzos anteriores para crear un serializador DNS, las librerías fueron de uso simple y otorgan funcionalidades bastante completas. El esfuerzo consistió en encontrar las librerías y completar el cliente básico. Tomó una semana implementar el cliente (sólo responde a peticiones de RRs de tipo A) gracias al código disponible.

NodeJS tiene ventajas al manejar de forma asíncrona las peticiones de forma nativa. Sin embargo, para poder crear un cliente con el menor sacrificio de desempeño y con tiempos de respuesta comparables a los otros servidores DNS se hizo una versión del cliente en Go.

4.2.3. Arquitectura de software y módulos para el nuevo cliente

Examinando el diseño de la herramienta YCSB[18] (Yahoo Cloud Serving Benchmarking) se pueden rescatar las siguientes decisiones: existe un número fijo de hilos de ejecución dedicados a mantener una conexión estable con la base de datos mediante TCP[8] en una capa de interfaz a la base de datos y la carga de peticiones se reparte entre hilos adicionales desde la capa de carga.

El nuevo cliente cuenta con una arquitectura similar a la de la figura 2.4 y la figura 2.3. Esto consiste en manejar tres capas: conexiones UDP, balanceo de peticiones, módulos de interacción para cada base de datos.

[8]Transmission Control Protocol: protocolo de red para establecer conexiones estables.

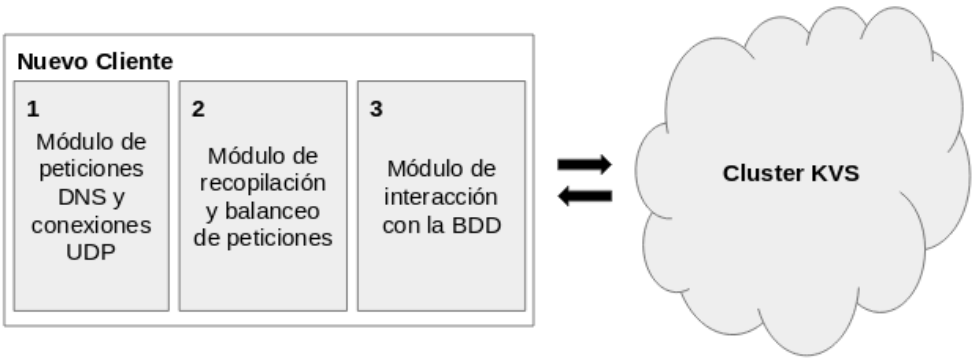

Figura 4.1: Arquitectura del nuevo cliente

Las responsabilidades de los módulos son:

- Módulo 1: manejar los paquetes UDP en base a las consultas recibidas.
- Módulo 2: realizar la recopilación de resultados, transcribir códigos de respuesta, utilizar el módulo 3 para interactuar con la base de datos.
- Módulo 3: gestionar una conexión estable con la base de datos, con conexiones a uno o varios nodos del cluster[9].

Desde este punto en adelante en este trabajo los módulos a nivel arquitectura son referidos como M1,M2 y M3.

4.2.4. Implementación: Kvs Dns con Golang

Repositorio en https://github.com/dario617/goKvsDns

Con la arquitectura anterior definida se escogió el lenguaje de programación Go o Golang dado su manejo simple de concurrencia y paralelismo, además de su rendimiento en aplicaciones para servidores. Se construyó el cliente intermediario y se le llamó Kvs Dns. A continuación se presenta como se implementaron los módulos.

Módulo 1: Servidor DNS

Escribir un servidor DNS desde cero, que además cumpla con los requisitos de la IETF requiere de un equipo de ingenieros y muchísimo tiempo de desarrollo. Sin embargo en los repositorios de código abierto existen diferentes librerías que permiten manipular paquetes DNS, hacer firmas digitales sobre zonas, leer archivos de zonas, etc. Uno de estas es **dns**[10] de **Miek Gieben** que proporciona módulos de Go que cumplen los requisitos descritos arriba, y todos los RFC de la IETF relacionados. Este repositorio es popular, mantenido y con una

[9]Conjunto de computadoras funcionando en sincronía
[10]Repositorio: https://github.com/miekg/dns

API simple. Cuenta con un servidor base que escucha peticiones DNS y permite proporcionar una función para responder.

Con esta librería como base basta crear una función que redirija el manejo de las respuestas a los módulos intermediarios. De esta forma el trabajo recae en el manejo de respuestas, cantidad de hilos, conexión con las bases de datos y la librería se encarga de escuchar, desempacar paquetes, codificar respuestas y enviar.

M1 entonces permite recibir la configuración inicial, es decir: base de datos, direcciones del cluster, cantidad de hilos a utilizar, verbosidad de logs y el puerto a escuchar.

Módulo 2: Driver intermediario

Este conjunto de módulos de Go se llaman drivers intermediarios pues su rol es conectar las peticiones a las bases de datos. Para que el código fuese extensible se escribió una *interface*, de forma que agregar nuevos drivers fuese sencillo.

```
type DBDriver interface {
    MakeQuery(m *dns.Msg) int
    UploadRR(line string) error
    HandleFile(location string, replace bool)
    ConnectDB(ips []string)
    Disconnect()
    Handle(w dns.ResponseWriter, r *dns.Msg)
}
```

De los métodos más relevantes tenemos:

- MakeQuery: toma un mensaje dns y ejecuta una consulta utilizando la conexión de cada driver. Esto permite que la implementación sea independiente para cada base de datos.
- Handle: este método se entrega directamente al M1 para ser llamado cada vez que se reciba un mensaje DNS. Entonces para cada KVS se pueden implementar de forma completa o parcial cada respuesta posible al mensaje.
- Connect y Disconnect son métodos totalmente dependientes del driver de conexión. Esta capa de abstracción separa el M1.

Por ejemplo para Redis se utiliza un driver de conexión externo (M3).

```
type RedisKVS struct {
    client *redis.ClusterClient
    Print  bool
}
//...
func (r *RedisKVS) ConnectDB(ips []string) {
    // []string{":7000", ":7001", ":7002", ":7003", ":7004", ":7005"}
    rdb := redis.NewClusterClient(&redis.ClusterOptions{
        Addrs: ips,
    })
    r.client = rdb
}
//...
```

```
// Metodos adicionales
// ...
```

Vemos que el *struct RedisKVS* que implementa la interface *DBDriver* utiliza un driver externo con alias *redis* para crear su conexión. Otras estructuras como la utilizada en Etcd cuentan con métodos adicionales que permiten verificar cuales datos son ingresados a la base de datos. Cada driver debe interactuar con un módulo de M3 y además contar con un esquema de datos que sea adecuado.

Sobre las respuestas vacías. Si el servidor no es dueño del dominio, por lo que no puede entregar respuestas autoritivas, se implementa lo requerido en el RFC1035[34] que especifica que se debe retornar el código de error 5 **REFUSED**.

Módulo 3: Driver de conexión

Conectar las bases de datos al cliente funciona bajo diferentes drivers de código abierto. Los drivers funcionan como módulos de Go externos que son integrados al M2 de la arquitectura. Los drivers utilizados son:

1. **gocql**[11].
2. **go-redis/redis**[12].
3. **etcd/clientv3**[13].

Todos los drivers manejan de forma independiente las conexiones TCP, búsqueda de respuestas entre nodos de los cluster, pérdidas de conexión y re-conexión. Asimismo, cuentan con una API que realiza operaciones sobre las bases de datos.

4.2.5. Esquemas de datos

Para definir el esquema de datos se definen primero cuales son los Resource Records a utilizar. Considerando que el objetivo principal de las pruebas de rendimiento es el tiempo de respuesta, no se requiere considerar todas, por lo que siguiendo el RFC1035 se consideran entonces: A, NS, CNAME, SOA, PTR, HINFO, MX, TXT y AAAA.

La mayoría de los RR tiene la siguiente estructura:

```
Key             RType     Class     TTL     Value
// Por ejemplo
dominio.cl      A         IN        600     192.168.0.231
```

donde cada campo tiene la siguiente definición:

- Key: suele ser el dominio consultado, excepto cuando se realizan preguntas inversas, es decir dada una IP consultar por el dominio.
- RType: valor numérico que indica cual es el tipo de RR.

[11]Repositorio: https://github.com/gocql/gocql
[12]Repositorio: https://github.com/go-redis/redis
[13]Repositorio: https://github.com/etcd-io/etcd/tree/master/clientv3

- Class: clase de RR, pero para este trabajo todos son de tipo INET abreviado IN.
- TTL: tiempo de validez antes de volver a consultar por el registro.
- Value: normalmente una IP o dominio. Para los TXT esto es un texto plano de formato libre. Para los SOA, MX y HINFO son particulares.

Sólo puede existir un valor SOA por cada zona registrada en el servidor DNS y debe existir al menos un registro NS con su correspondiente A o AAAA. Un dominio puede contar con uno o más registros de tipo A, MX, NS y suele ser el caso. Estas son las consideraciones para los esquemas.

Cassandra

Para Cassandra, como se describió en el estado del arte, no se cuenta con un modelo relacional si no orientado a consultas. Esto resulta ser beneficioso pues sólo contamos con una llave primaria, el dominio o IP.

Los esquemas de la base de datos consisten en la llave primaria que es el dominio (o IP), y la llave de partición que es una llave al azar obtenida con una función de hash. Esto tiene la siguiente ventaja:

- Al consultar por un dominio podemos preguntar por la llave primaria, por ejemplo el dominio "*ejemplo.cl*"que cuenta con una o más valores indexados por diferentes llaves de partición.
- Nos permite agregar más de un valor por dominio sin complicar las columnas con listas ni conjuntos.

Considerando estas decisiones se construyeron las tablas para cada tipo de RR.

Redis

En Redis colocar más de un valor para una llave no puede implementarse como en Cassandra pero Redis cuenta con diferentes estructuras de datos. Para cumplir las mismas funciones que Cassandra se utilizan las listas y los conjuntos (set).

Las llaves se forman como *"Dominio + : + RType"*. Los valores son almacenados como texto plano y consideran todo lo que no sea la llave. Para los valores únicos como SOA se utiliza una simple llave valor. Para A, MX, NS, etc se utiliza un conjunto lo cual evita que se repitan valores. Para TXT se utiliza una lista y se agrega el texto a ella.

No tiene sentido almacenar los valores como bytes puros pues se trata de texto y no se gana más espacio. Almacenar texto facilita a Redis realizar comparaciones.

Etcd

Etcd no cuenta con ninguna estructura de datos integrada en los valores. Por esto almacenar múltiples valores para una llave única es posible pero debe implementarse manualmente en el cliente. Es esto lo que se realizó a nivel de M2. El modelo de datos es idéntico al de Redis.

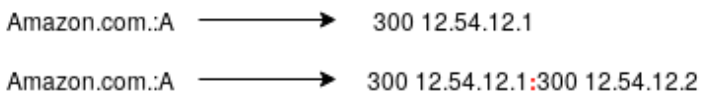

Figura 4.2: Etcd en Kvs Dns: principio de funcionamiento

Lo que se hace es ingresar valores únicos a una lista de valores. Esto representa un conjunto o *set*. El ejemplo muestra dos RR de tipo A para el dominio *amazon.com*, cada valor se guarda como *TTL Value* en este caso.

4.2.6. Integración con Ansible

Para poder conectar las bases de datos de forma automática se creo un archivo Makefile que en conjunto con Ansible realizan las siguientes tareas:

1. Instalar los requisitos para utilizar las bases de datos. Por ejemplo Cassandra utiliza Java 8.
2. Descargar las BDD, descomprimir los archivos, mover a su posición correcta.
3. Compilar de ser necesario.
4. Preparar el esquema de ser necesario.
5. Levantar el servicio en segundo plano
6. Lanzar Kvs Dns en la máquina maestra y conectar al cluster

Se consideró que la instalación del cliente más su base de datos debe ser lo más limpia posible, para lo cual la instalación del software se realiza a nivel usuario, en una carpeta llamada *workspace*(configurable).

Ansible permite agrupar las tareas en roles, y roles en playbooks. Por esto se escribieron 2 playbooks por cada servidor, uno para instalar y lanzar la BDD y otro para apagarla.

Como las librerías se instalan a nivel sistema fue necesario otorgar permiso de administrador.

4.2.7. Configuración de bases de datos

Considerando un cluster de 3 servidores cada base de datos opera con configuraciones diferentes tanto en cantidad de nodos como de puertos disponibles por máquina. En esta sección se muestran las diferencias por cada BDD.

Cassandra

Cassandra al momento de definir los *keyspaces* necesita que se defina replicación. Para que haya consenso el número debe ser impar por lo cual como base se cuenta con un cluster de 3 bases de datos. Estas se comunican entre si para repartir llaves y para esto se configura un puerto, normalmente el 7000.

La persistencia de datos no necesita configuración adicional, pero como se vio en la definición de conceptos CAP la consistencia es eventual entre bases de datos.

Redis

Redis no cuenta con persistencia absoluta de datos, si no con estrategias para minimizar su pérdida. En este caso se utilizó la combinación de Append-Only-File y Snapshot RDB. Así para cada servidor Redis se puede perder a lo más 1 segundo de datos nuevos (a nivel local). No obstante las operaciones realizadas con mayor frecuencia son de lectura. Las operaciones de escritura se realizarían sólo al actualizar una zona.

Para un cluster se necesitan como mínimo 6 servidores, para poder repartir claves y obtener consenso. De éstos la mitad corresponden a nodos maestros, quienes realizan operaciones de escritura y nodos esclavos, quienes reciben datos desde el maestro. Redis cuenta con una utilidad CLI para que los nodos distribuyan sus roles entre máquinas, evitando tener dos nodos maestros en la misma.

Etcd

Este es la única base de datos que no permite ser daemonizada[14], por lo cual un proceso debe quedar a cargo de forma permanente. Para esto se debe crear un usuario asignado a mantener el proceso activo y instalar como servicio en *systemd*.

4.2.8. Persistencia y carga de datos

Para asegurar que los Resource Records no dependan exclusivamente de la integridad del cluster KVS se siguió la estrategia de los servidores tradicionales, es decir, contar con archivos de zona con los RR definidos.

Los archivos de zona son recuperados por una utilidad desarrollada para Kvs Dns que lee cada zona, revisa que posean ciertas características básicas como la presencia de un registro SOA, y ingresa los datos al cluster utilizando los driver del M2 y M3. Los datos son ingresados a través de la red.

[14]Los Daemons son procesos que corren en segundo plano, sin supervisión directa de un usuario.

4.3. Resource Records y datasets

Para probar cuantas consultas DNS sería capaz de contestar cada servidor se necesita que haya variedad de consultas y su respectiva variedad de zonas. Knot Benchmark opta por utilizar sólo datos sintéticos, distribuidos de forma uniforme entre zonas. Esto claramente difiere de un escenario de uso real de un servidor donde, hay dominios que concentran registros como por ejemplo un servicio de máquinas virtuales. A continuación se muestran dos dataset a considerar.

4.3.1. Datos reales

NIC Labs cuenta con un conjunto de registros RR de consultas DNS, pero dada la pandemia del COVID-19 no se pudo acceder a ellos remotamente. Por esto se buscaron datasets de RR en el sitio de la organización CAIDA (Center for Applied Internet Data Analysis)[15], tanto de zonas disponibles como de consultas. Se encontró un dataset de donde se estudiaba la topología de la red[9] realizando consultas por dominios FQD[16]. Conformado por diferentes archivos que contienen, consultas y respuestas realizadas en un mes a diferentes dominios.

Al utilizar los datos de CAIDA de diferentes años se obtuvieron dominios y resultados parciales, pues algunos dominios dejaron de existir y la respuesta guardada correspondía a un error. Por esto se creó un programa en Go que utilizando la librería DNS para Kvs Dns realizó consultas durante 3 semanas para obtener la mayor cantidad de registros posibles para cada dominio. Dado que el programa era ligero se utilizó un Raspberry Pi[17] para recopilar la información faltante. Se obtuvieron en total 2,4 millones de Resource Records.

Archivos de zona

Como se necesitan archivos maestros de zonas para los servidores DNS se escribió un programa para agrupar los registros en una estructura de árbol y rescatar los registros A, SOA, AAAA, TXT y HINFO. Como los datos obtenidos fueron incompletos se completaron las zonas con datos sintéticos para SOA y NS. Esto no cambia la distribución de datos rescatada de forma significativa.

La selección fue reducida dado que los servidores tradicionales exigen la existencia de valores A para todos los dominios, manejo correcto DNAME en archivos de zona, entre otros. De esta forma se simplificó el programa de conversión.

Distribución de RR por zonas recuperadas

Para visualizar la distribución RR por zona de datos recopilados se presenta el siguiente histograma con las primeras 500 zonas más densas.

[15] https://www.caida.org/home/

[16] Fully-Qualified-Domain son dominios que apuntan a una IP, es decir, no forman parte de un Alias por ejemplo.

[17] Raspberry Pi es una computadora multiuso miniaturizada de bajo costo creada en el Reino Unido.

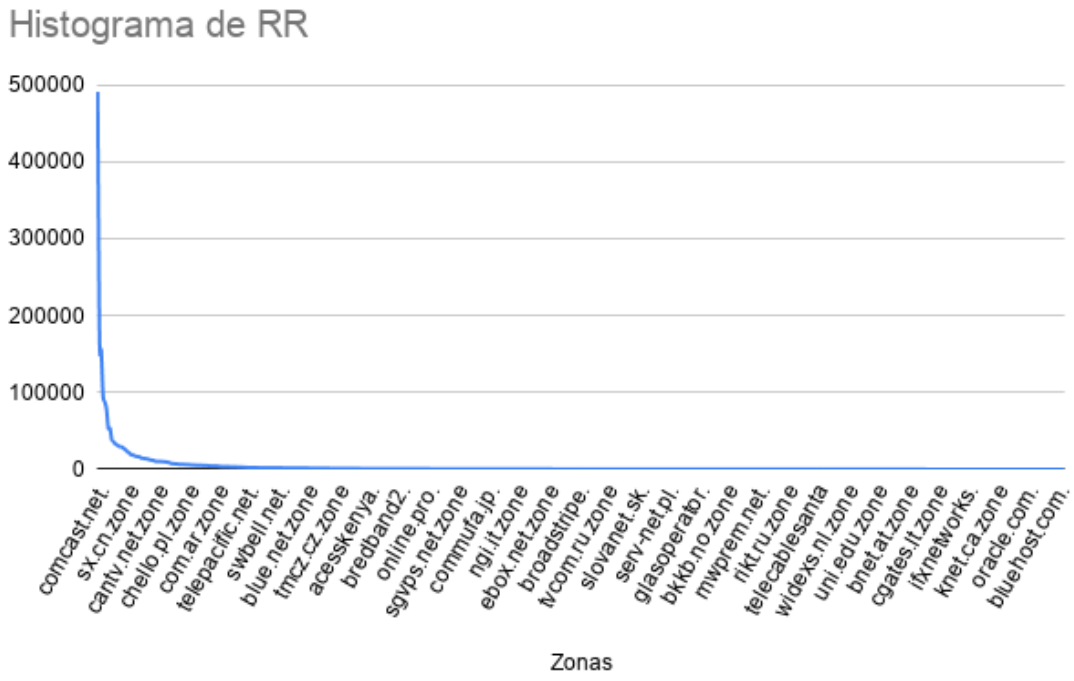

Figura 4.3: Histograma de datos recopilados

Cualquier subconjunto de datos al azar que se obtenga tendrá una distribución no uniforme, en comparación con los datos sintéticos.

4.3.2. Generar datos sintéticos

Knot Benchmark proporciona diferentes herramientas y en particular programas para crear zonas con palabras al azar. Esto es mejor que crear zonas con dominios que incluyan texto totalmente aleatorio. Las zonas generadas cuentan con 10 RR en total. Las zonas sintéticas se utilizan durante las pruebas del nuevo benchmark.

4.4. Kvs Dns Benchmark

Repositorio en https://github.com/dario617/kvs-dns-benchmarking

Para realizar pruebas con Kvs Dns y los servidores de nombres se construyó un nuevo benchmark orientado a sistemas distribuidos. Se basa en gran parte sobre Knot Benchmark pero se realizaron adiciones significativas las cuales se describen en esta sección.

4.4.1. Variables y métricas

Se consideraron relevantes los siguientes datos a recolectar por cada máquina:

- Interrupciones de sistema para recolectar datos de memoria secundaria. Basta medir el valor IRQ y IOWAIT de la utilidad *mpstat*.
- Uso del CPU a nivel usuario, sistema, etc. Basta obtener los datos desde programas como *top* o *mpstat*.
- Uso de la memoria principal considerando RAM y Swap. Para esto se puede utilizar *free -m* en consola.
- Tiempo de respuesta para consultas. Utilizando la utilidad *Dig* recuperando el RTT (round time trip).
- Uso de red como tamaño de paquetes UDP, bytes recibidos y enviados. Leyendo directamente */proc/net/dev/* para la interface de red utilizada.
- Porcentaje de respuesta reportado. Esto es reportado por si mismo como servidor. Se mide calculando la diferencia entre paquetes recibidos y enviados desde */proc/net/dev/*.
- Porcentaje de respuesta observado. Respuestas recibidas por el usuario. Esto se mide utilizando Wireshark y contando los paquetes de respuesta.

Las métricas pueden ser medidas siempre y cuando el servicio esté activo.

Además existen variables a considerar como:

- Dataset a utilizar.
- Cantidad de RR gestionados/almacenados.
- Carga de paquetes por segundo.
- Fallas de sistema.

Todos estos datos/métricas y variables son combinados en el diseño de pruebas.

4.4.2. Diseño de pruebas

Considerando que el objetivo es comparar las bases de datos KVS como sistema distribuido contra los servidores tradicionales se opta por utilizar 4 máquinas. Una como cliente que realiza consultas y otras 3 como servidor DNS. Así existe una cantidad suficiente para los KVS que requieren consenso y el escenario se asemeja a un escenario real con varios servidores redundantes con las mismas zonas.

Se definen 3 pruebas generales:

1. Medición de rendimiento y estadísticas bajo una demanda alta de transacciones por segundo.

2. Medición de tiempo de respuesta en condiciones de poca demanda.

3. El primer punto pero realizando una falla controlada a uno de los servidores.

La primera prueba se realiza para diferentes dataset, tanto en volumen de datos alojados como tipo (sintético y real). Cada servidor puede recibir un tráfico cada vez más intenso y se puede estudiar la evolución del rendimiento.

La segunda prueba nos permite estudiar cuanto afecta el protocolo de intercambio de información entre servidores KVS al tiempo final de respuesta. Esto es un factor a considerar dado que el cliente Kvs Dns no alberga información y debe repetir la consulta a su base de datos. Por otro lado, sin condiciones de alta demanda se puede estudiar cuanta es la carga del servidor sobre la máquina que lo hospeda.

La tercera prueba permite simular que el proceso falle, si esto afecta a la red y como afecta a las consultas. Dado que no se pueden realizar mediciones dentro de una máquina que falla se considera mejor eliminar el proceso del servidor DNS / Base de datos.

4.4.3. Orquestación con Ansible

Ansible permite agrupar tareas en roles, roles en playbooks. Cada rol se asigna a un grupo de máquinas y estos grupos cuentan con variables de configuración y variables descubiertas durante un proceso de escaneo. Variables que Ansible descubre son por ejemplo la configuración de red, dispositivos, núcleos del procesador, sistema operativo, entre otros.

Se definen dos grupos de operación: el/los clientes o *requesters* y los servidores.

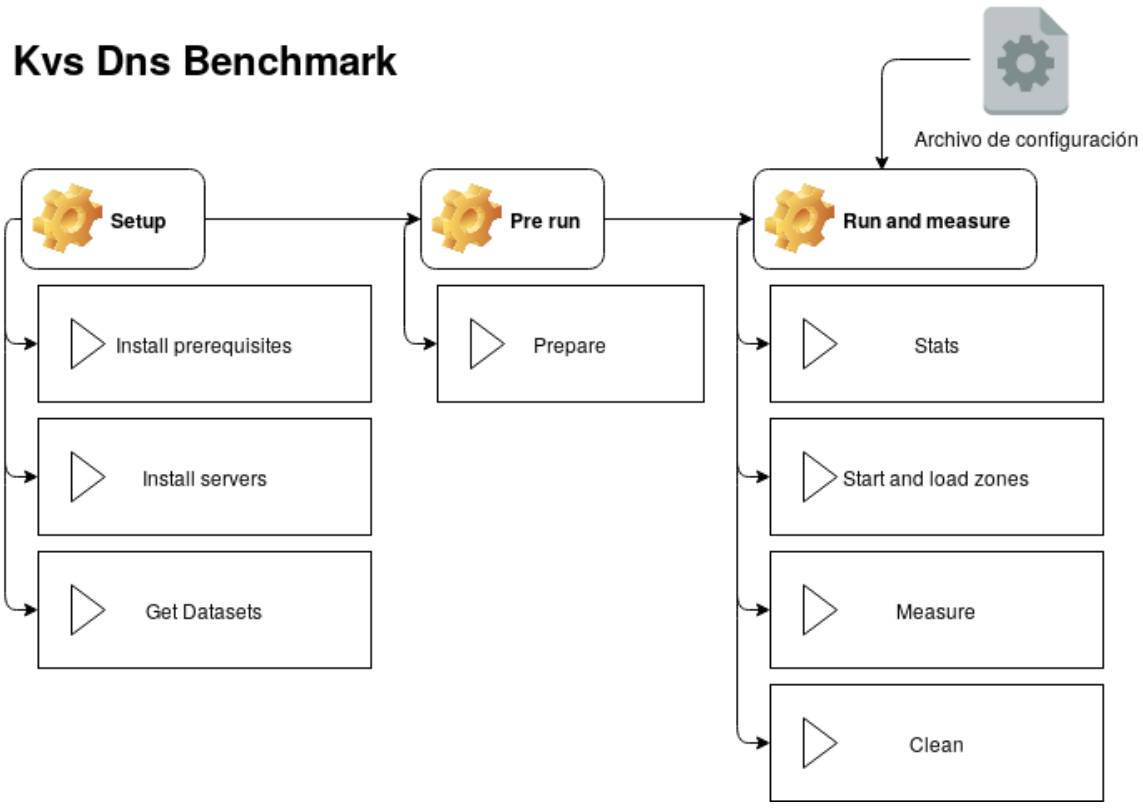

Figura 4.4: Esquema de funcionamiento general de Kvs Dns Benchmark.

Los engranajes representan playbooks y los triángulos roles. Como vemos en la figura (4.4) hay tres playbook principales que son utilizados de izquierda a derecha. En detalle:

1. Setup se encarga de preparar las máquinas del *requester* y los servidores, instalando todo lo necesario para operar y compilar el código fuente de ser necesario para los servidores DNS y las bases de datos. Finalmente el rol *Get Datasets* descarga los datos desde un archivo remoto y herramientas para crear archivos PCAP.
2. Pre run crea archivos de configuración para los servidores DNS, crea un conjunto de consultas para ambos dataset y con esto dos archivos PCAP con mensajes UDP preparados.
3. Run and measure lee un archivo de configuración adicional para ejecutar las pruebas para un servidor DNS/KVS en específico. Captura estadísticas del sistema antes de cargar las zonas. Carga las zonas o guarda los RR en las bases de datos (si no fue hecho antes). Realiza las tres pruebas descritas. Recupera valores del sistema con los servidores encendidos y libera los recursos en preparación para la siguiente ronda de pruebas.

El archivo de configuración exporta variables de entorno con el nombre de la base de datos/servidor de nombre y el dataset a utilizar. Por esto el benchmark cuenta con 12 rondas de pruebas en total.

En particular para el rol Measure es relevante que: se envían paquetes durante 10 segundos con un flujo constante, las mediciones para la prueba 1 son desde dos puntos de vista. El

primero es leer **/proc/net/dev** donde cada máquina registra número de bytes y paquetes recibidos, además de paquetes no procesados ya sea porque la cola está llena o que llegaron corruptos. El segundo desde el *requester* con Wireshark quien lee los paquetes que salen y entran revisando las IP de origen y destino durante un periodo de 40 segundos. El periodo a considerar es amplio porque es de interés ver si los servidores simplemente no pudieron contestar o si se demoraron de forma excesiva.

4.4.4. Compilación, servidores y configuración

Durante los pasos de preparación los servidores se configuran sin límite de memoria, utilizando los núcleos disponibles de sistema. Las librerías de las cuales dependen son diferentes para distribuciones de Linux basadas en Debian y en Red Hat. Esto se toma en cuenta en el playbook de *Setup*. No se consideró Windows Server como máquina para realizar pruebas.

Las máquinas necesitan acceso de administrador para instalar todas las librerías necesarias, además de una conexión SSH sin contraseña. Esto se configura fácilmente a través de la CLI de SSH y con un uso de SUDO.

Para aprovechar al máximo la naturaleza distribuida de Kvs Dns, en lugar de levantar un sólo cliente que concentre el tráfico de red y que mantenga conexiones estables con el sistema distribuido KVS se levanta uno por cada máquina. Entonces cada máquina tiene acceso a completo al cluster KVS para realizar sus consultas. Esto también permite que en caso de que uno de los nodos del sistema falle, por ejemplo un nodo maestro de Redis, la máquina pueda seguir consultando a los servidores vecinos.

RRL y limitación de respuesta

Los servidores de nombre cuenta con una utilidad llamada Response Rate Limiting que se creó para contrarrestar ataques de amplificación[18]. Esto consiste en limitar la capacidad de respuesta del servidor autoritativo si se detectan volúmenes grandes consultas maliciosas, lo cual significa que están dirigidas hacia el mismo cliente y son probablemente falsificadas. Esto es precisamente lo que se hace con el benchmark, por lo cual esta funcionalidad de seguridad se encuentra desactivada en los archivos de configuración de Bind, Knot y NSD.

4.4.5. Envío y creación de paquetes

Para la creación de paquetes se necesita contar con la información desde la capa Ethernet hasta mensaje DNS. Utilizando Scapy los valores de cada capa son asignados de forma manual al momento de crear los paquetes UDP. Estos consideran en particular la dirección MAC de las máquinas de destino, para la cual es necesario determinar cual de las interfaces de red interactúa directamente con el *requester*. Luego se ingresan las IP, puertos y el mensaje mismo.

[18]La ISC explica en este artículo https://kb.isc.org/docs/aa-00897

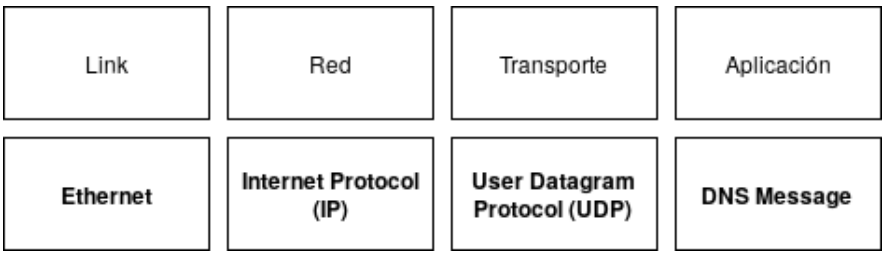

Figura 4.5: Diagrama simplificado de capas a manipular con Scapy.

Con los archivos PCAP listos para un número fijo de consultas diferentes el paso siguiente es enviarlos a través de la red. Para esto se utiliza Tcpreplay que permite manejar definir el período de transmisión y velocidad para un conjunto de paquetes.

En Knot Benchmark los paquetes creados contienen valores estáticos vacíos para la MAC e direcciones IP. Estos son remplazados durante la ejecución de cada test con Tcprewrite. En este benchmark los paquetes cuentan con valores fijos correctos, por lo cual este paso es innecesario.

4.4.6. Configuración de red

El caso ideal para medir el desempeño de un servidor es conectar la máquina cliente directamente a él y así reducir lo más posible variables como el uso de red. Sin embargo en escenarios reales existe al menos un Router, Switch, y servidores intermedios. Cloudflare utiliza Anycast[19] a nivel de red WAN y LAN[13] para distribuir la carga al azar a sus servidores DNS, y NIC Chile opera de forma similar. En nuestro caso no es necesario configurar un router para realizar Anycast. Enviar paquetes puede hacerse de forma secuencial a cada servidor por simplicidad.

Así se considera el siguiente escenario:

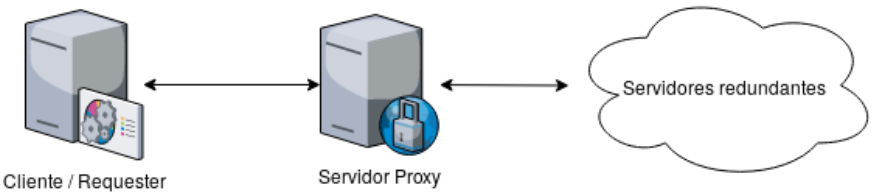

Figura 4.6: Topografía general de red.

donde existe un intermediario de red que distribuye la carga a servidores.

El benchmark desarrollado posee una variable para enviar los paquetes a través de un servidor Proxy o máquina intermediaria. Esto se hace escribiendo la dirección MAC de dicha máquina, y a nivel IP colocando las IP de destino de la red interna. Se hace así pues el cliente

[19] Anycast es un método de ruteo y direccionamiento de red donde una sola dirección tiene múltiples caminos posibles o internamente más de una ruta.

afuera de la red no conoce las direcciones MAC de la red privada, ni cuantas máquinas existen en principio.

4.4.7. Experiencia de usuario

Ansible proporciona información detallada durante la ejecución de tareas y playbooks, lo que permite saber en tiempo real como se ejecutan las pruebas desde en cual paso va hasta cuanto tiempo ha tomado cada uno. Esto junto a un archivo de texto *README.md* detallando cuales variables modificar, como las direcciones de las máquinas, flujo de paquetes, cantidad de registros a considerar y hospedar por servidor permite que se prueben diferentes escenarios sin necesidad de modificar directamente el código.

El programa en si funciona con Bash por lo cual no es necesario interactuar directamente con Ansible.

4.4.8. Herramientas adicionales

Para facilitar operaciones de limpieza se crearon dos Playbooks adicionales.

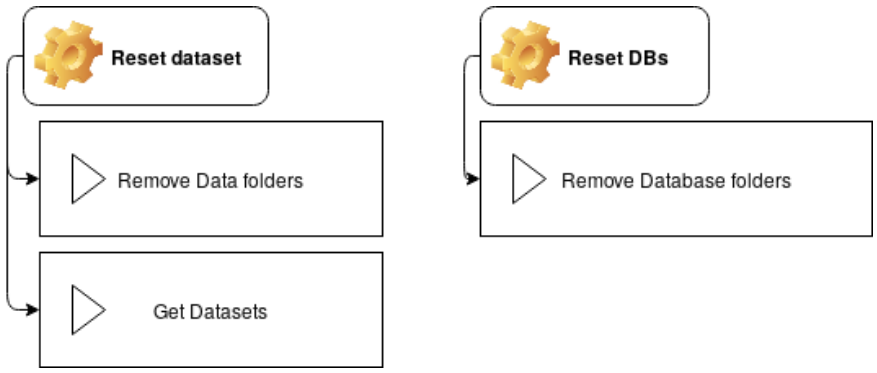

Figura 4.7: Herramientas adicionales para Kvs Dns Benchmark

Para hacer cambios sobre las zonas, ya sea para recrear zonas sintéticas, agregar más o expandir el programa para recuperar zonas desde los RR recuperados se tiene el Playbook *Reset Dataset*. Para limpiar la memoria secundaria de las máquinas eliminando los datos almacenados por las bases de datos se creó *Reset DBs*. Ambos playbook funcionan bajo un programa escrito en Bash que evita que el usuario interactúe directamente con Ansible.

Finalmente se agrega una herramienta escrita en Python que recolecta los datos y los copia de forma ordenada a un archivo CSV, que puede ser leído por diferentes herramientas de análisis como Excel, Pandas, etc.

4.5. Objetivos: Kvs Dns y el nuevo Benchmark

Creando el cliente Kvs Dns se logra conectar DNS con las bases de datos Key-Value-Store. El cliente esta diseñado para responder consultas básicas, que nos permiten medir tiempo de respuesta y disponibilidad, sin ser un programa pesado que consuma muchos recursos.

Con el benchmark desarrollado se separan los procesos de compilación, obtención y generación de datos, ejecucion de pruebas y limpieza de sistema. La compilación se realiza antes de las pruebas y no por cada una. El estado del becnhmark se muestra de forma transparente al usuario y él nunca interactúa directamente con Ansible dado que el programa principal es interactivo mediante Bash. Este programa permite realizar las pruebas de comparación que se necesitan.

Existe una disyuntiva, el discernir si los límites son a causa del programa o de las limitaciones de su entorno de prueba. Para esto es importante revisar las métricas de uso del CPU y memoria. Sin embargo esto no se hace automáticamente desde el benchmark construido. Debe realizarse como análisis manual.

Capítulo 5

Validación y Resultados

En este capítulo se presenta el escenario de pruebas, los resultados de pruebas con el benchmark de Knot, los resultados del nuevo benchmark para dos tipos de conjuntos de datos y finalmente una síntesis de los resultados obtenidos.

5.1. Entorno de pruebas final

Para poder probar tanto el funcionamiento de las bases de datos, el cliente intermediario Kvs Dns y posteriormente los benchmarks, se cuenta con la siguiente configuración de red:

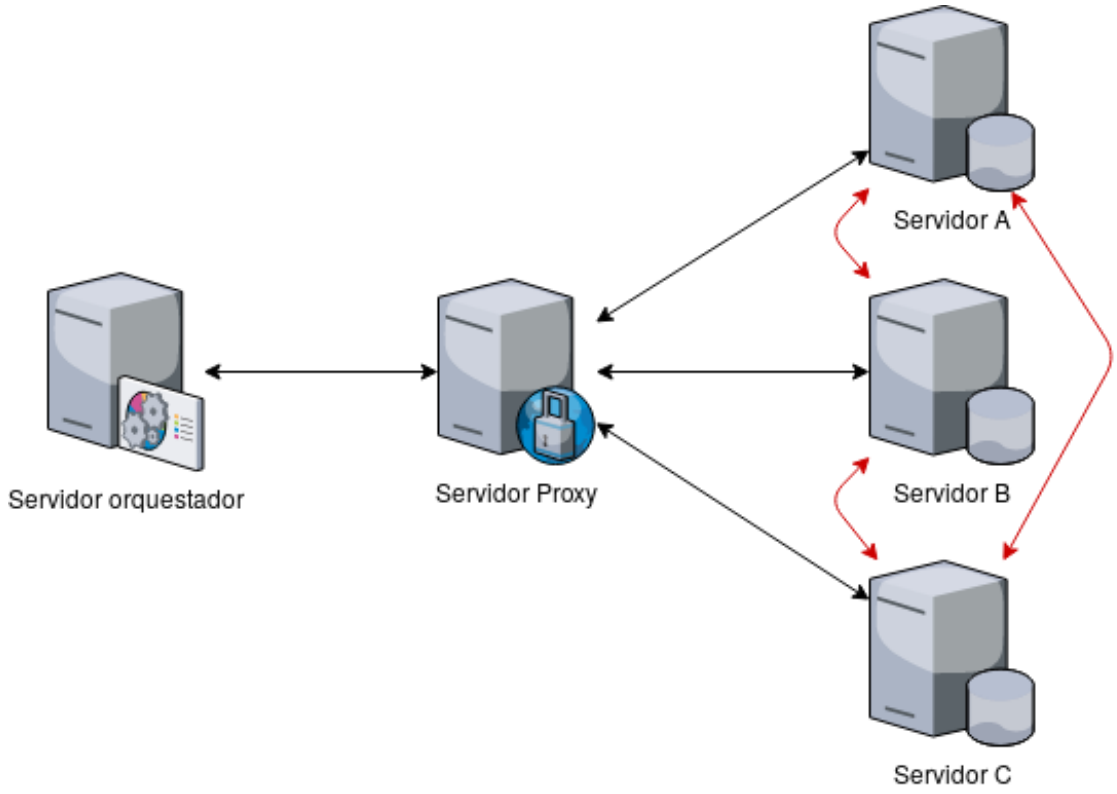

Figura 5.1: Configuración de red para entorno de pruebas

Las líneas negras representan el tráfico con el mundo exterior, y las rojas el tráfico dentro de la red privada. Estas últimas en el caso de las KVS son el tráfico para llegar a consensos, repartir llaves, etc. En el caso de los servidores de nombre es inexistente.

La tabla (5.1) presenta una tabla con los datos sobre el poder de procesamiento de cada máquina relevante. Vemos que sólo la máquina externa a la red cuenta con un poder de procesamiento mayor, y los servidores A a C poseen las mismas capacidades. Estos son donde residen las bases de datos y servidores DNS.

Máquina	Procesador	Núcleos/Hilos	RAM	Swap
S. Orquestador	Intel(R) Xeon(R) CPU E5520 @2.27GHz	4/8	11.6GB	200MB
S. A	Intel(R) Atom CPU D510 @1.66GHz	2/4	2GB	2GB
S. B	Intel(R) Atom CPU 330 @1.60GHz	2/4	2GB	2GB
S. C	Intel(R) Atom CPU 330 @1.60GHz	2/4	2GB	2GB

Tabla 5.1: Procesadores y memoria de servidores de entorno de prueba.

Respecto a los sistemas operativos el servidor orquestador tiene un sistema operativo CentOS 7, el servidor A y B Debian 9, el servidor C Debian 10 que se considera menos estable que la versión anterior.

5.1.1. Consideraciones sobre los dataset

Se utilizaron 6 dataset para realizar pruebas de estrés.

Número de RR	Número de zonas	Tipo de datos
100.000	10.000	Sintéticos
100.000	1500	Reales
200.000	20.000	Sintéticos
200.000	1500	Reales
2.000.000	200.000	Sintéticos
2.000.000	15.000	Reales

Tabla 5.2: Dataset utilizados para pruebas de estrés.

Cada dataset contiene RR diferentes se construyó seleccionando valores al azar de los datos reales recuperados y generando un nuevo conjunto. Se considera poco realista que un sólo servidor maneje más de un millón de RR, para esto existe delegación de zonas. No obstante se realizaron pruebas a modo de verificar las tendencias de los otros dos escenarios con cantidad de zonas más realista.

5.1.2. Consideraciones sobre las pruebas

Para las pruebas se realizó el siguiente volumen de consultas en total:

Prueba	Inicio QPS	Paso QPS	Tope QPS	Datasets
Módulo de respuestas	3000	2000	30000	100k RR
Módulo de fallos	3000	2000	25000	100k RR
Módulo de respuestas	6000	4000	50000	200k RR
Módulo de fallos	6000	4000	38000	200k RR
Módulo de respuestas	5000	10000	100000	2M RR
Módulo de fallos	5000	10000	35000	2M RR

Tabla 5.3: Datasets y volúmenes de datos por prueba.

El módulo de respuestas considera la medición de evolución de memoria, cpu, y capacidad de respuesta por cada máquina. El módulo de fallos ejecuta el anterior pero apaga un proceso en un servidor al azar. Para este se usan menos puntos de medición pues los datos del módulo anterior más unos datos adicionales son suficientes para encontrar una tendencia.

Volúmenes de consultas

El benchmark funciona enviando desde un cliente orquestador o *requester* volumenes grandes de consultas a un grupo de servidores. Desde el cliente se envían por ejemplo 30.000 consultas que equivalen a 10.000 consultas por servidor.

En particular se distingue entre las mediciones auto-reportadas de flujo menor y las reportadas por wireshark de flujo total.

5.2. Knot benchmark

Para determinar cuales eran los niveles de operación o valores esperados de rendimiento bajo cargas pesadas (mayores a 14.000 QPS) se utilizaron los servidores B y C para probar el Benchmark original.

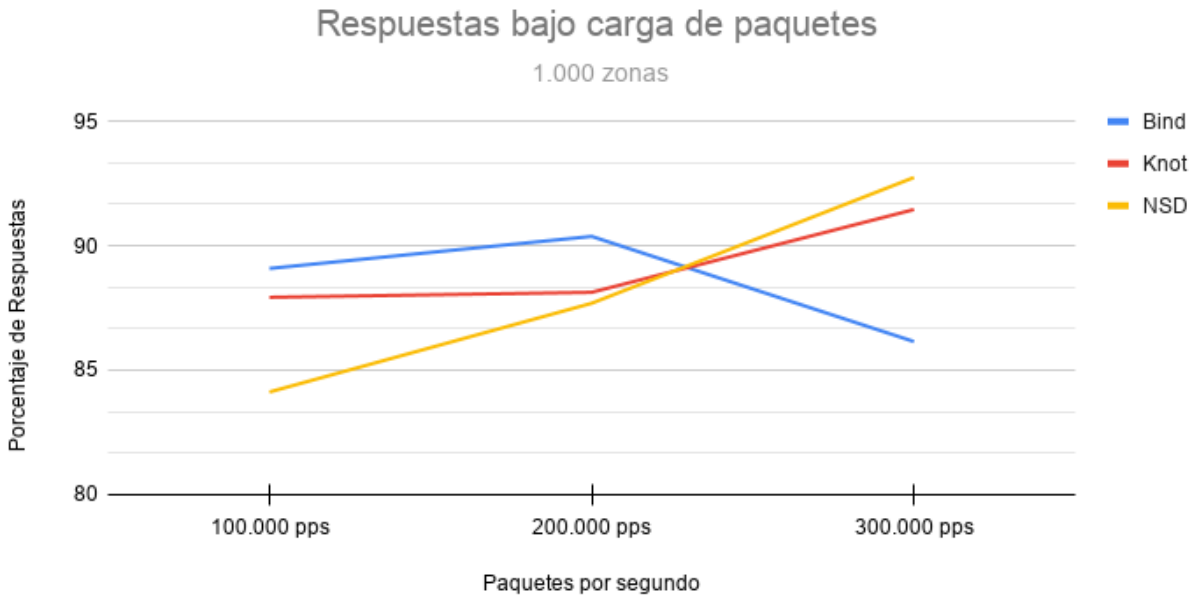

Figura 5.2: Porcentaje de respuestas bajo cargas de consultas por segundo, para Bind, Knot y NSD.

Los resultados son contradictorios puesto que bajo las cargas de la figura (5.2) el benchmark reportó que cada servidor respondió cerca de **6 a 5** consultas por segundo. Esto es una fracción minúscula de la carga enviada por tcpreplay. A pesar de repetir el experimento los valores reportados no cambiaron significativamente por lo que no se pudo estudiar la capacidad individual de estos servidores con este benchmark.

En particular el benchmark reportó que cada servidor recibió cerca de 6-8 consultas por segundo, no 100.000 a 300.000. El problema puede estar en el cálculo de los valores finales con la herramienta *bc*, el tiempo de espera al terminar la operación de reenvío, cierre premeditado de las conexiones SSH, límite de capacidad de la red alcanzado o alguna interferencia.

5.3. Kvs Dns Benchmark

En esta sección se presentan los resultados obtenidos de las pruebas de rendimiento del benchmark desarrollado. Gráficos adicionales se encuentran en el anexo (Anexo).

5.3.1. Uso de memoria inicial

Al cargar las zonas por cada servidor y dejar los procesos en segundo plano se observa lo siguiente para ambos dataset. En este caso se cargan 200.000 RR distribuidos en 20.000 zonas para los datos sintéticos y 1500 zonas para los datos reales recuperados.

Caso de estudio: Datasets de 200.000 RR

Se muestra el uso de memoria, por cada servidor, para cada máquina en las siguientes figuras.

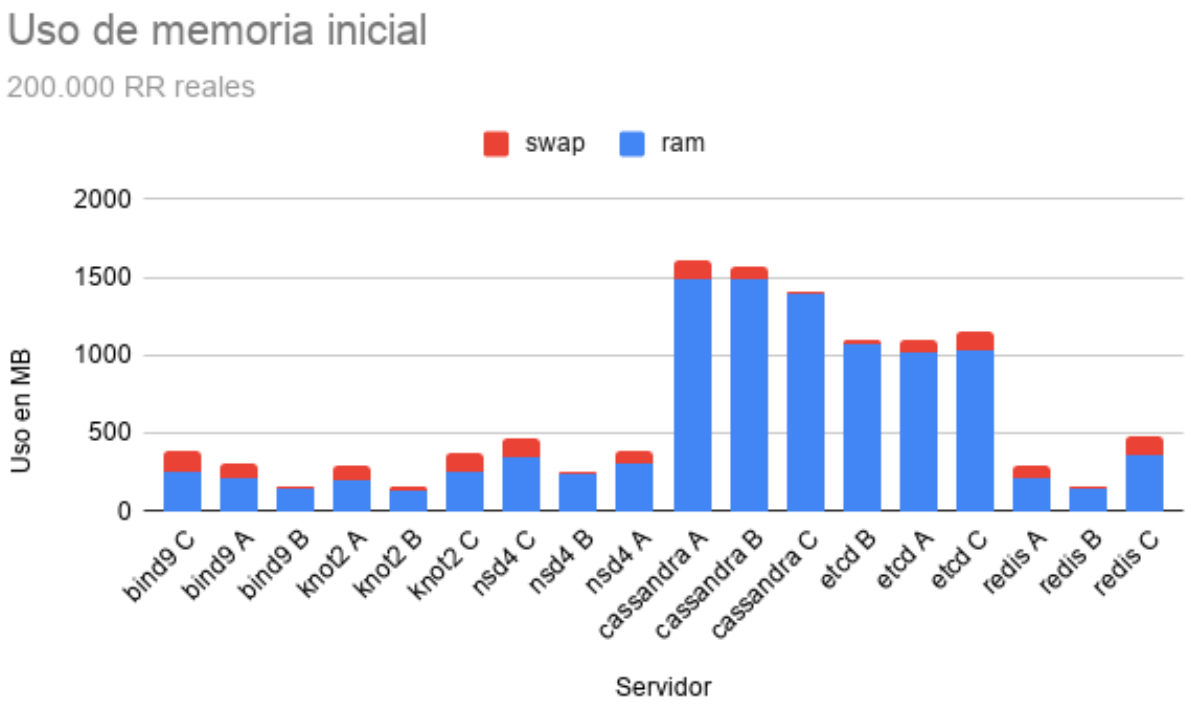

Figura 5.3: Uso de memoria inicial por cada servidor DNS con 200.000 RR reales.

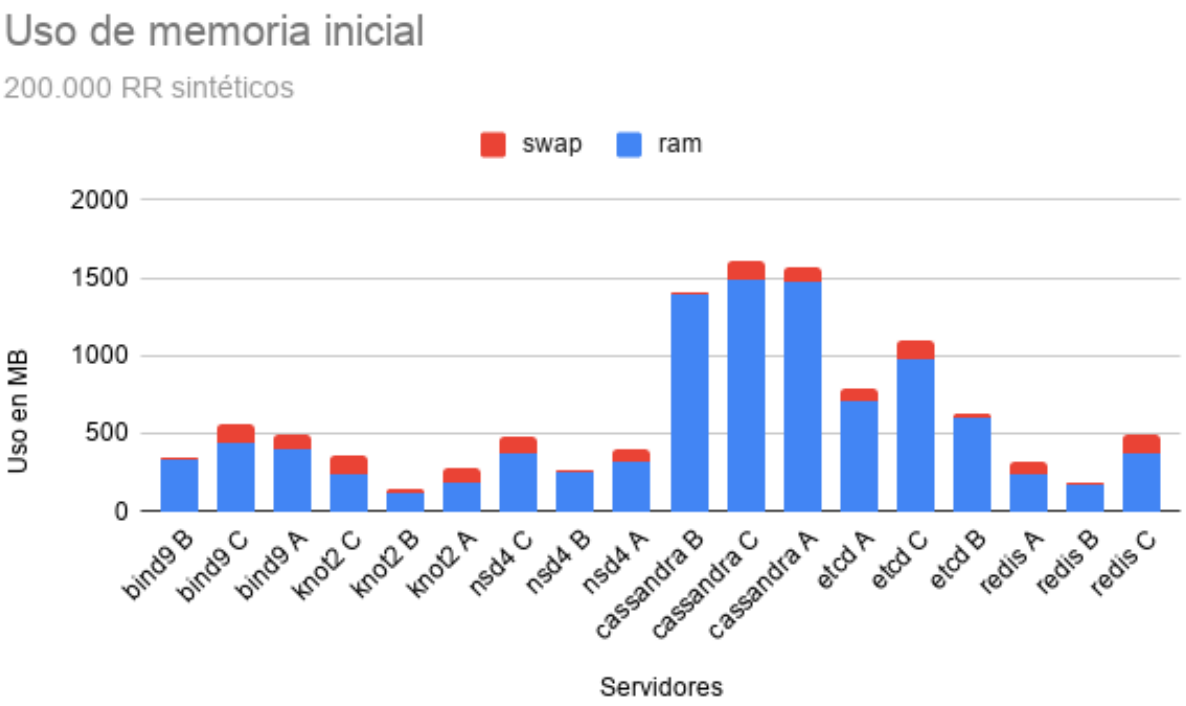

Figura 5.4: Uso de memoria inicial por cada servidor DNS con 200.000 RR sintéticos.

Se ve en la figuras (5.3) y (5.4) que el uso de memoria de Cassandra y Etcd es muy alto en comparación a los otros servidores. Cassandra permite configurar cuanta memoria reservar de forma preventiva, para evitar que otros procesos se la adjudiquen, de hecho su uso base suele ser la mitad de la memoria disponible.

Además los valores de Swap es probable que sean residuales pues aún no se ha alcanzado la saturación de memoria en 2.048 MB (2GB).

5.3.2. Tiempo de respuesta

Se midió utilizando *dig* el tiempo de respuesta de cada conjunto de servidores, con 1000 muestras para cada uno.

Caso de estudio: Dataset de 100.000 RR

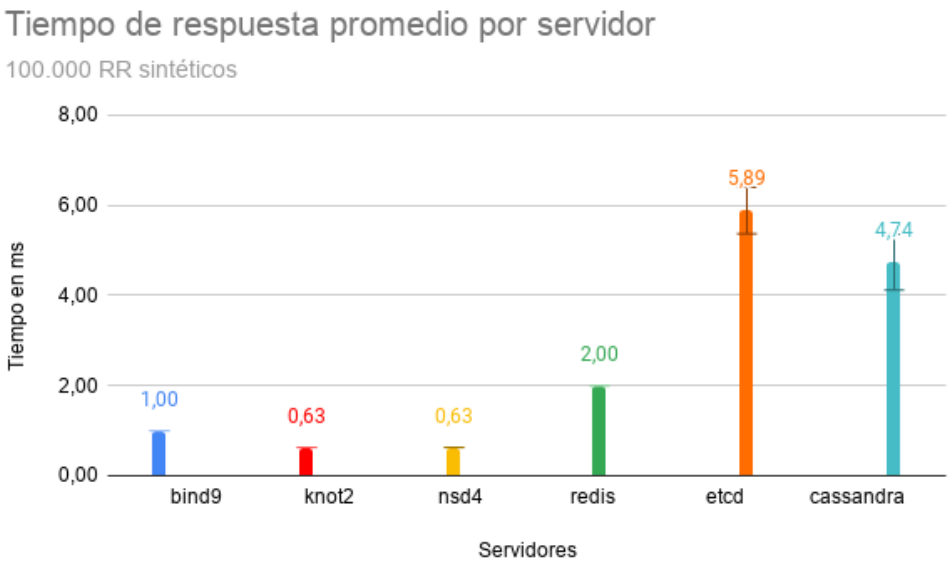

Figura 5.5: Tiempo de respuesta promedio por cada servidor DNS con 100.000 RR sintéticos.

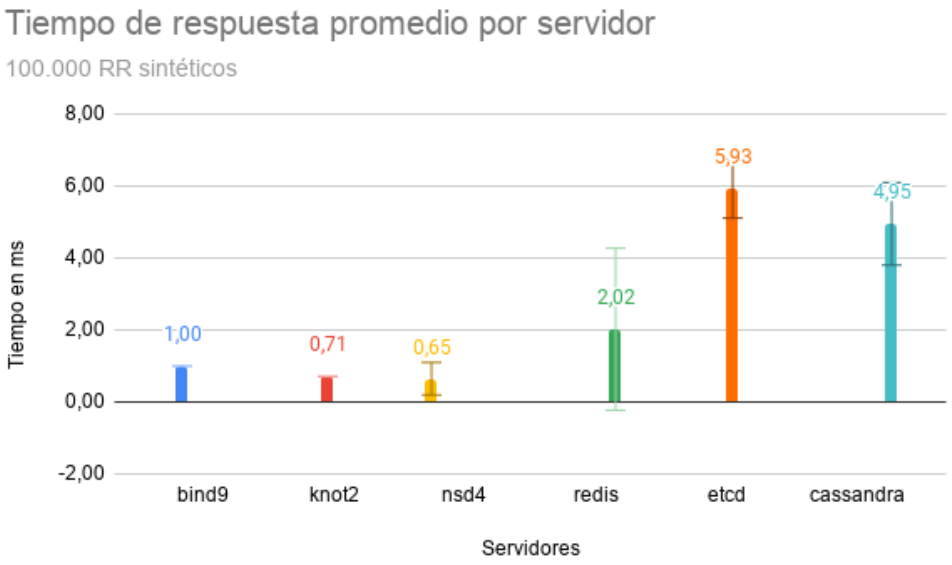

Figura 5.6: Tiempo de respuesta promedio por cada servidor DNS con 100.000 RR reales.

Se observa en ambas figuras que Redis es el más cercano en términos de velocidad de respuesta a los servidores tradicionales. Esto puede ser dado que Redis está diseñado como un

caché. Cassandra en cambio está diseñado para tener alta disponibilidad, no necesariamente en aplicaciones de tiempo real.

Caso de estudio: Dataset de 200.000 RR

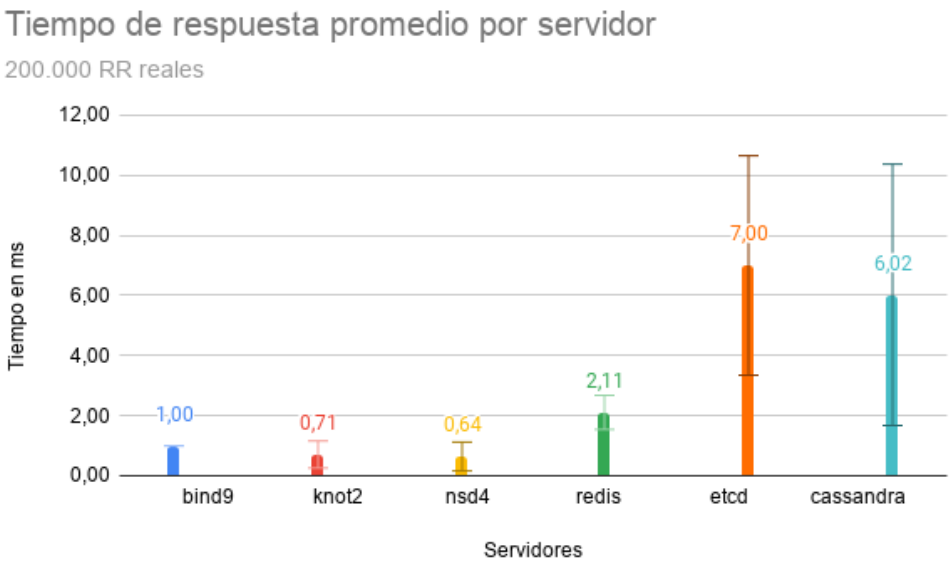

Figura 5.7: Tiempo de respuesta promedio por cada servidor DNS con 200.000 RR reales.

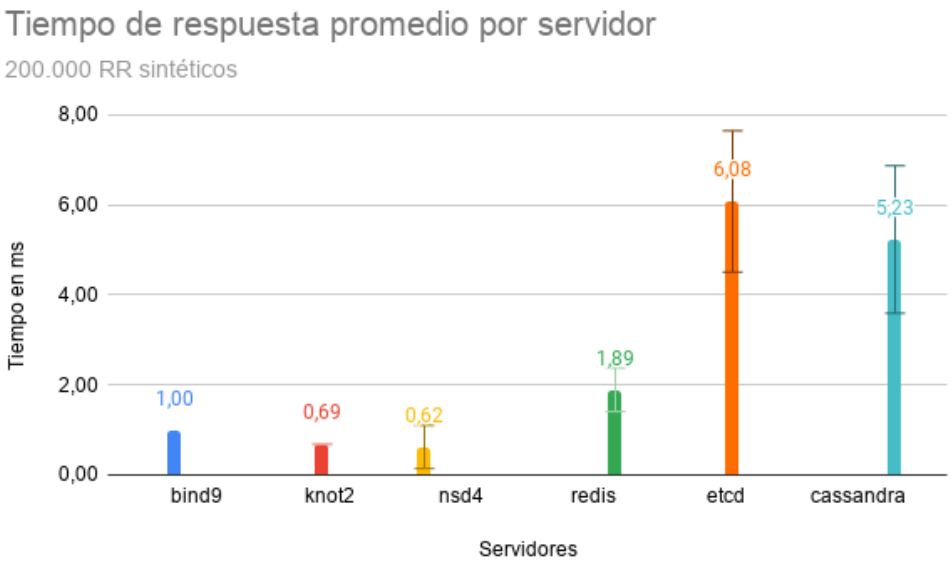

Figura 5.8: Tiempo de respuesta promedio por cada servidor DNS con 200.000 RR sintéticos.

A diferencia del caso con menos datos, la variación de los tiempos de respuesta es mayor. Esta diferencia es más evidente con los datos reales. Sin embargo el tiempo de respuesta en si no cambia significativamente.

5.3.3. Evolución de uso de la memoria

A continuación se muestran las diferentes tendencias de uso de memoria a medida que cada servidor se somete a una demanda creciente de consultas.

Caso de estudio: Dataset de 200.000 RR reales

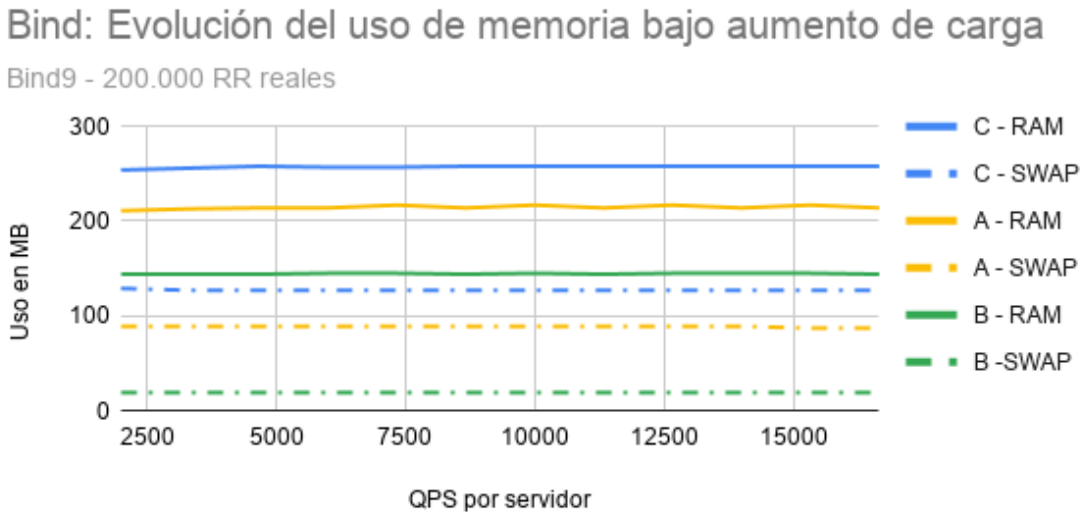

Figura 5.9: Evolución del uso de memoria para Bind y 200.000 RR reales.

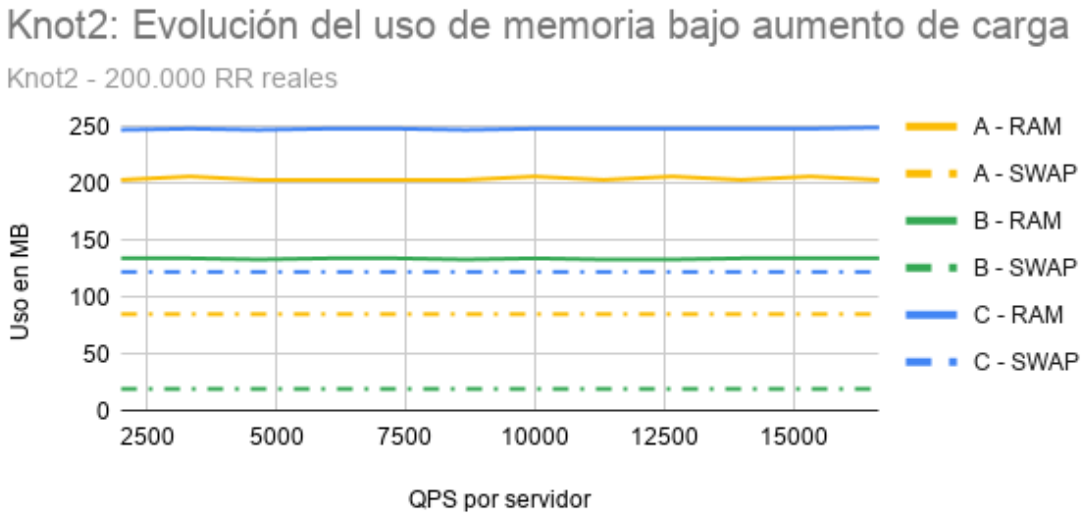

Figura 5.10: Evolución del uso de memoria para knot y 200.000 RR reales.

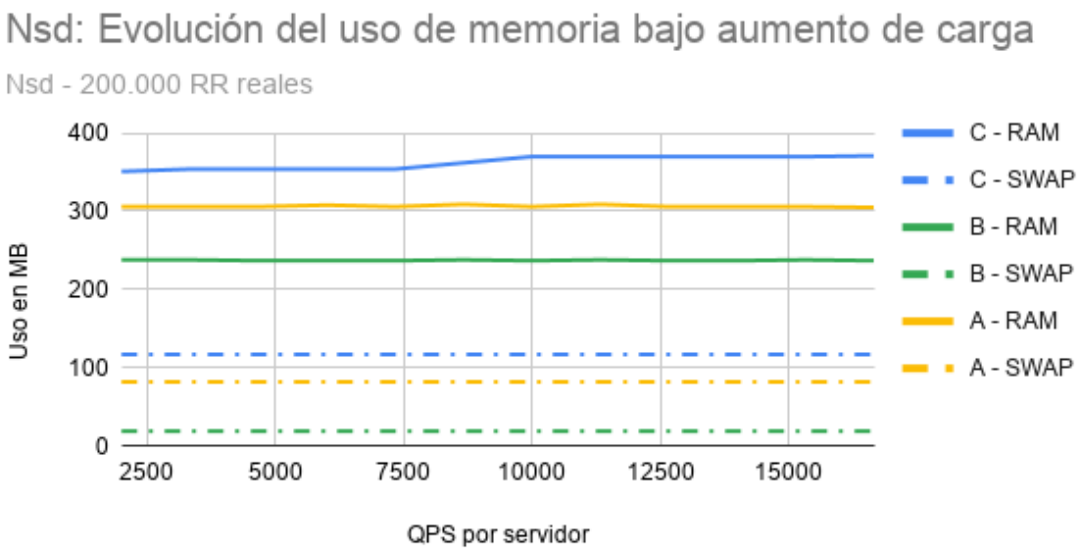

Figura 5.11: Evolución del uso de memoria para NSD y 200.000 RR reales.

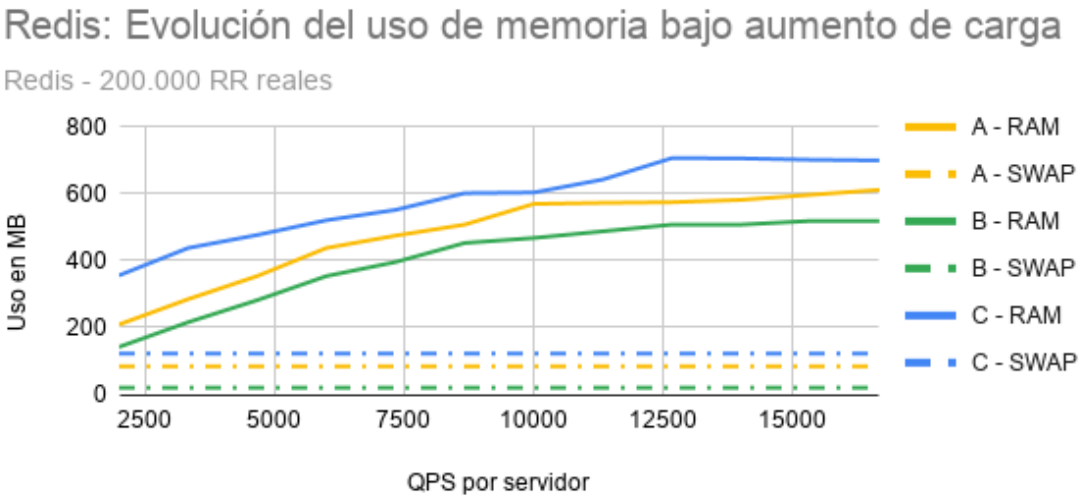

Figura 5.12: Evolución del uso de memoria para Redis y 200.000 RR reales.

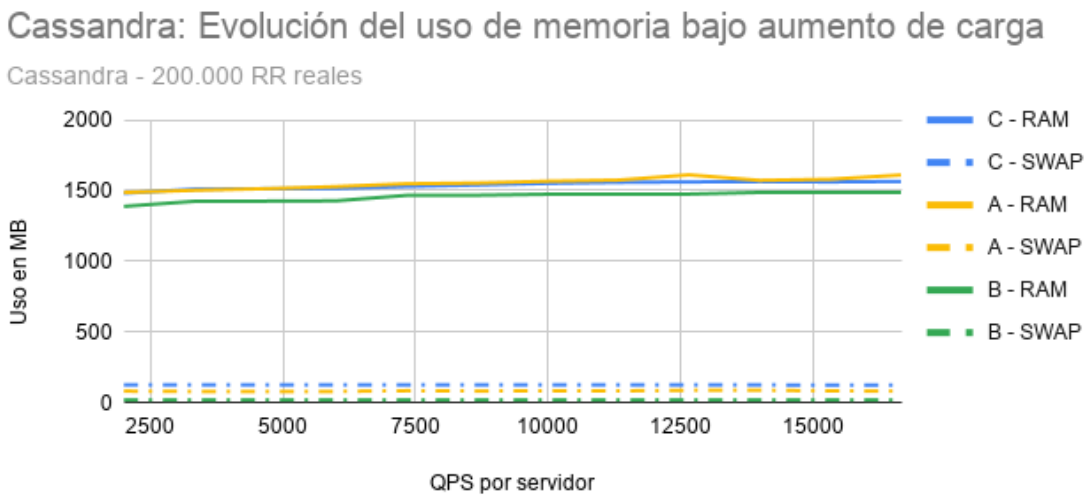

Figura 5.13: Evolución del uso de memoria para Cassandra y 200.000 RR reales.

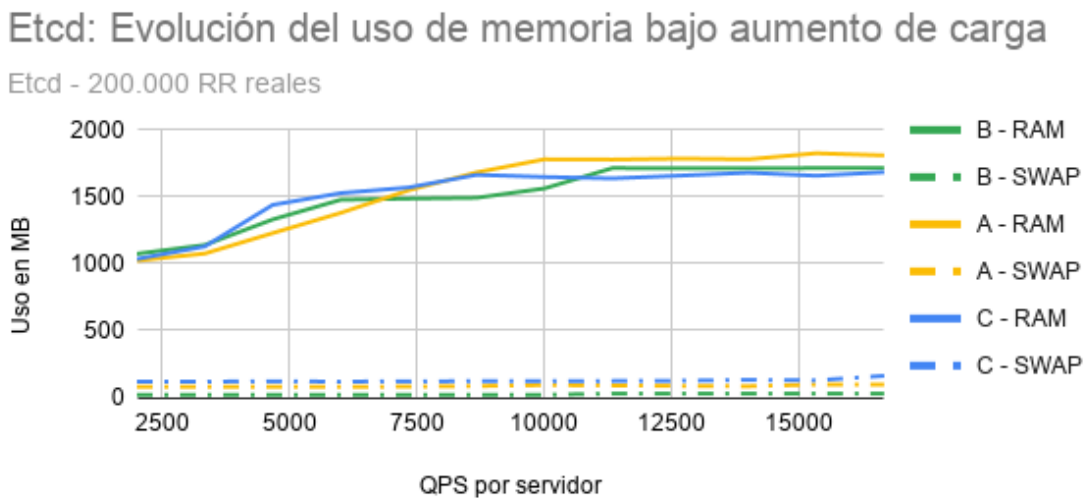

Figura 5.14: Evolución del uso de memoria para Etcd y 200.000 RR reales.

Se observa que los servidores tradicionales no se ven afectados en su uso de memoria por la demanda. Sin embargo Redis y Etcd si con Etcd casi utilizando todos los recursos de sistema ya al final de la prueba. Cassandra mantiene una pequeña tendencia de crecimiento pero su uso de memoria sigue cercano al inicial.

Cierto uso de la memoria se atribuye también al cliente Kvs Dns, en particular en etcd donde para entregar respuestas múltiples se necesita realizar copias del *string* entregado por la BDD.

Caso de estudio: Dataset de 200.000 RR sintéticos

Los resultados son muy similares, sólo se presentarán resultados representativos.

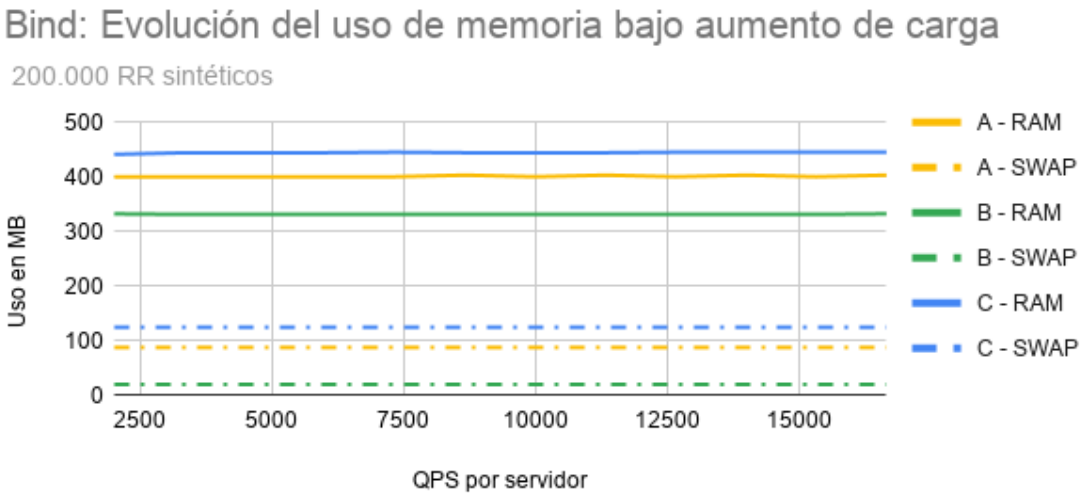

Figura 5.15: Evolución del uso de memoria para Bind y 200.000 RR sintéticos.

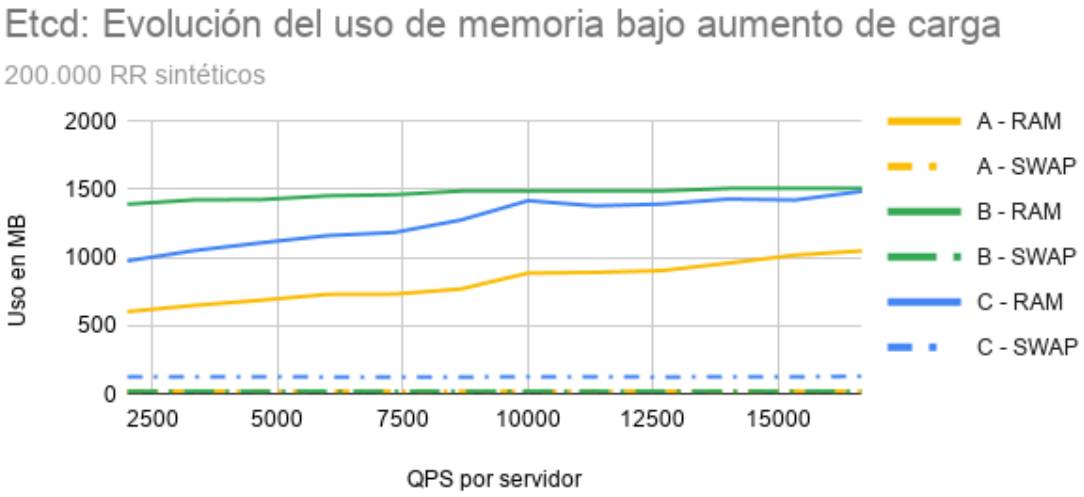

Figura 5.16: Evolución del uso de memoria para Etcd y 200.000 RR sintéticos.

Las tendencias se repiten para ambos grupos, sin embargo se ve que en comparación con los gráficos anteriores hay una diferencia en el uso de memoria. Esto se comparará a continuación.

Comparación de uso de memoria de ambos dataset de 200.000 RR

Para comparar el uso de memoria se obtiene un promedio de las 3 máquinas para cada servidor y se estudia la diferencia de uso de memoria.

En particular la diferencia parece ser constante por lo cual, se obtiene del promedio de esta para diferentes cargas de transacciones. El resultado es el siguiente.

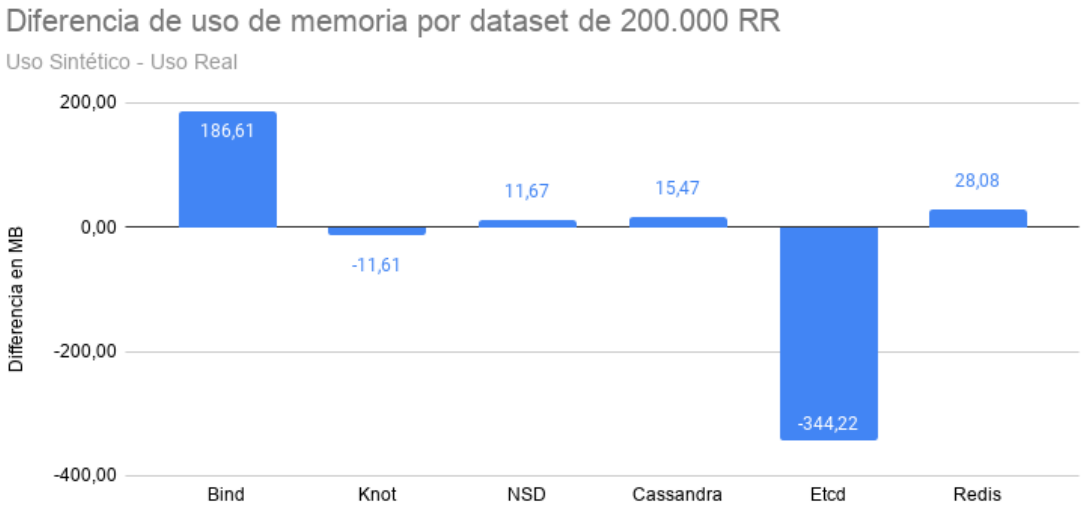

Figura 5.17: Diferencia de uso de memoria por dataset de 200.000 RR y servidor.

Las diferencias orientadas hacia arriba indican que el servidor utiliza más memoria con datos sintéticos. Por el contrario aquellas orientadas hacia arriba indican un mayor uso de memoria con datos con distribuciones reales.

Dada la implementación del manejo de llaves con Etcd y Kvs Dns sus valores son de mayor longitud cuando los registros RR se concentran en una zona. Esto explica que con datos reales su uso de memoria sea pronunciadamente mayor.

En el caso de Bind se observa que si se aumenta el número de zonas su uso de memoria aumenta.

Comparación de usos de memoria entre 100.000 y 200.000 RR

Figura 5.18: Evolución del uso de memoria para Bind y 100.000 RR sintéticos.

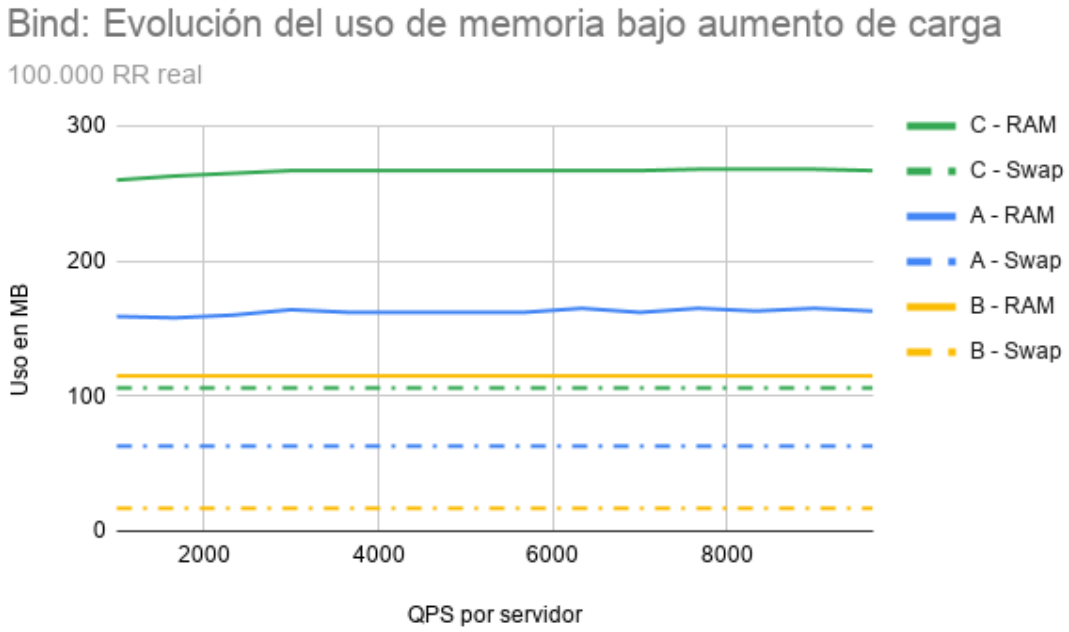

Figura 5.19: Evolución del uso de memoria para Bind y 100.000 RR reales.

En los gráficos de memoria de Bind con diferentes datos se ve la misma tendencia, a más zonas se utilizan más recursos de memoria. En el caso de Cassandra la memoria permanece

constante en 1,5 GB aproximadamente por su gran reserva inicial.

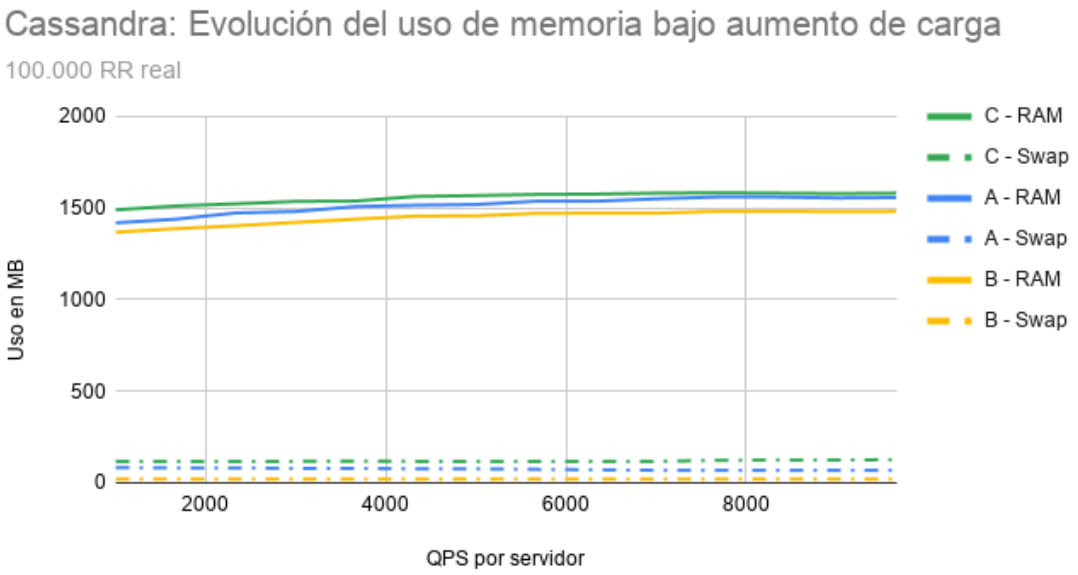

Figura 5.20: Evolución del uso de memoria para Cassandra y 100.000 RR reales.

5.3.4. Evolución de uso del CPU

En esta sección se muestra el uso de CPU a medida que se satura la red con consultas DNS por segundo. A continuación se explican las variables contenidas en los gráficos:

- usr: porcentaje del CPU funcionando a nivel aplicación de usuario.
- sys: porcentaje del CPU funcionando a nivel de sistema.
- iowait: porcentaje del tiempo que el CPU esperó mientras hizo una petición a memoria secundaria.

Los valores de las variables en un caso ideal van desde 0 a 1, o bien 0 % a 100 %. Números mayores indican una sobre carga en el núcleo. Los valores indicados en los gráficos corresponden al promedio de los CPU disponibles por cada máquina (2 por cada una).

Caso de estudio: Dataset de 200.000 RR reales con *mpstat*

Las mediciones realizadas con *mpstat* entregaron los siguientes resultados.

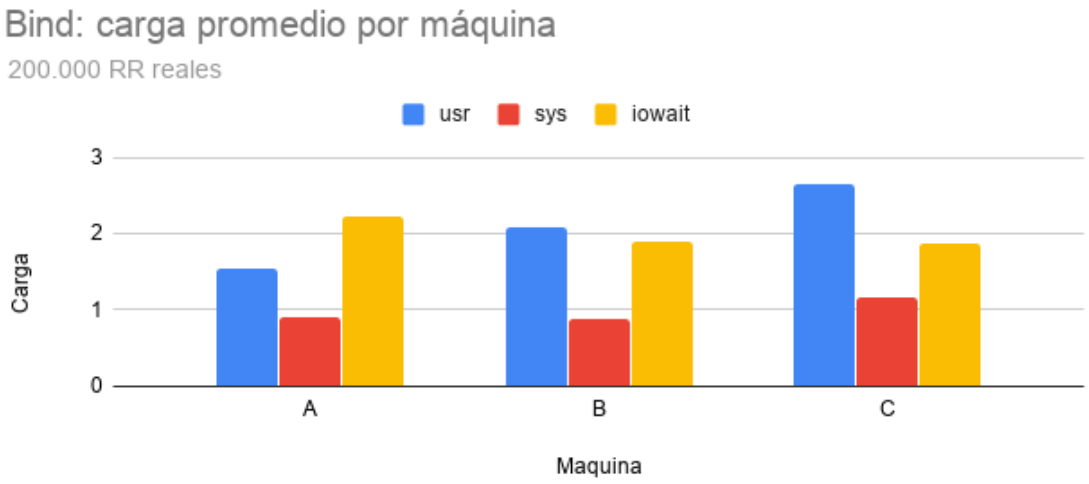

Figura 5.21: Carga promedio por máquina con Bind y 200.000 RR reales.

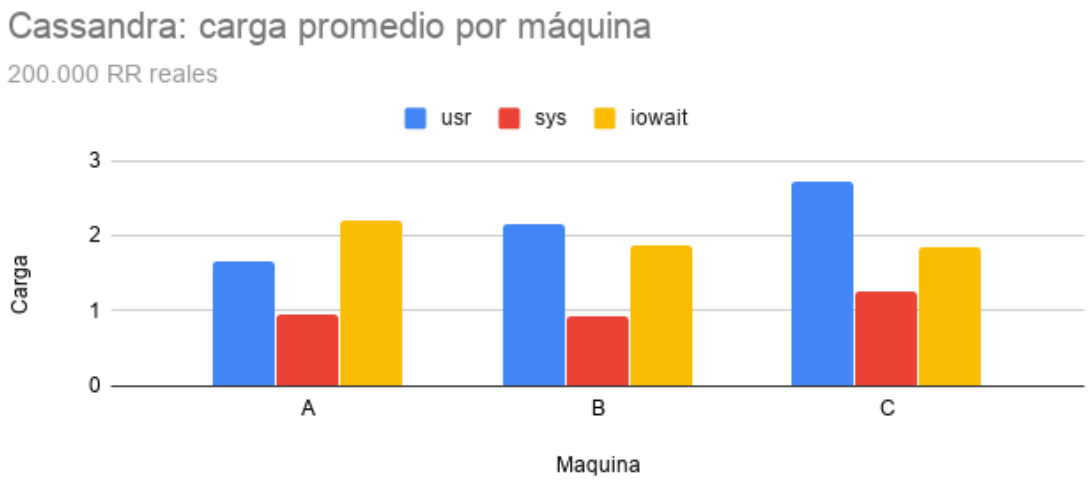

Figura 5.22: Carga promedio por máquina con Cassandra y 200.000 RR reales.

Si se revisa el uso para Cassandra y Bind se observa que tanto el uso de CPU e interrupciones IO son muy similares. Esto se repite para los demás servidores y parece estar asociado a la máquina en particular, más que al servidor DNS.

Por otro lado la tendencia de uso parece ser de cambios muy pequeños. Es decir que efectivamente la carga aumenta pero, en el orden de un 5 % cuando el servidor ya está saturando el CPU al doble de su capacidad como mínimo. La única carga mayor ocurre en Redis que a diferencia de los demás tiene una carga de 280 % en comparación con 263 % de los otros.

El uso de CPU con tendencia estática al momento de medición se ve en la siguiente figura, que es representativa de los resultados obtenidos.

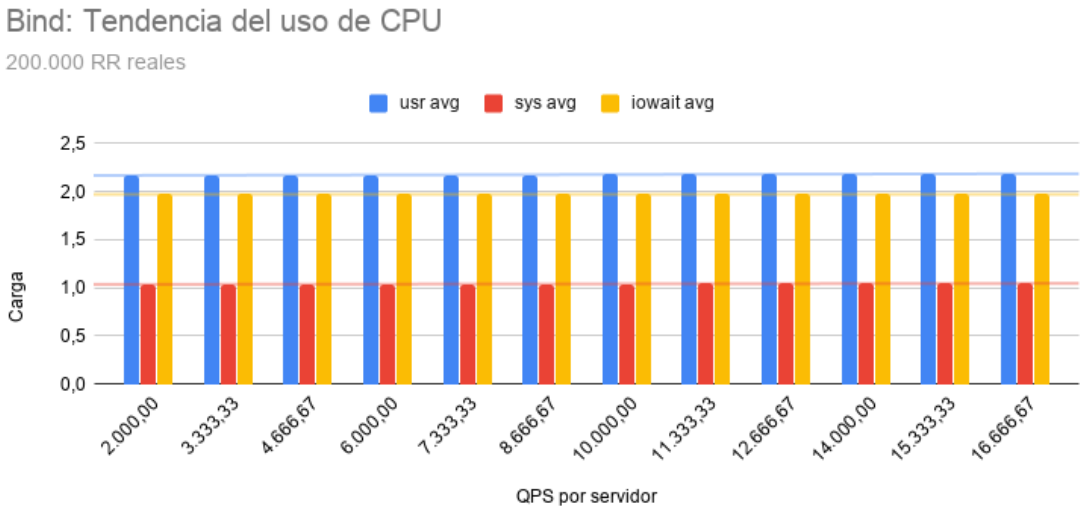

Figura 5.23: Tendecia del uso de CPU para bind entre sus 3 máquinas, con 200.000 RR reales.

Es probable que **exista un desfase** entre los valores medidos y el peak real de uso. Las mediciones se realizaron 5 segundos después de enviado el tráfico de 10 segundos.

Caso de estudio: Dataset de 100.000 RR reales con */proc/loadavg*

El pseudo sistema de datos */proc* contiene información del sistema en tiempo real. A partir de éste se midieron los siguientes datos.

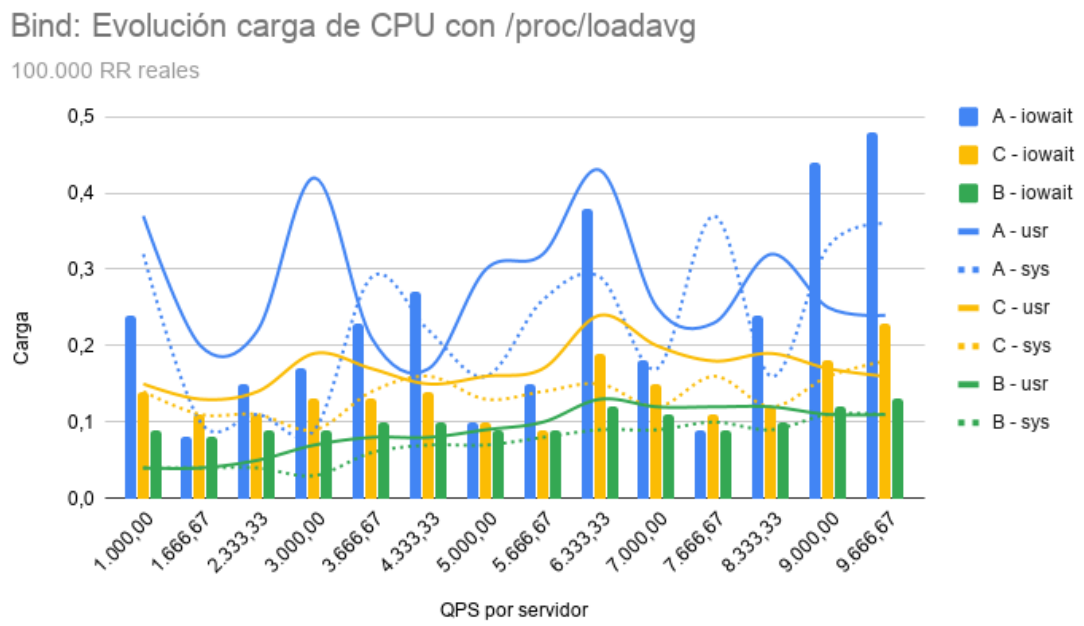

Figura 5.24: Tendencia de la carga de CPU para Bind entre sus 3 máquinas, con 100.000 RR reales.

Para Bind no parece haber una tendencia clara, incluso las mediciones parecen ruido. Lo que se mantiene constante es que la máquina C mantiene una carga superior a lo largo de las pruebas y métricas de la figura (5.24. Esto también sucede para NSD y Knot.

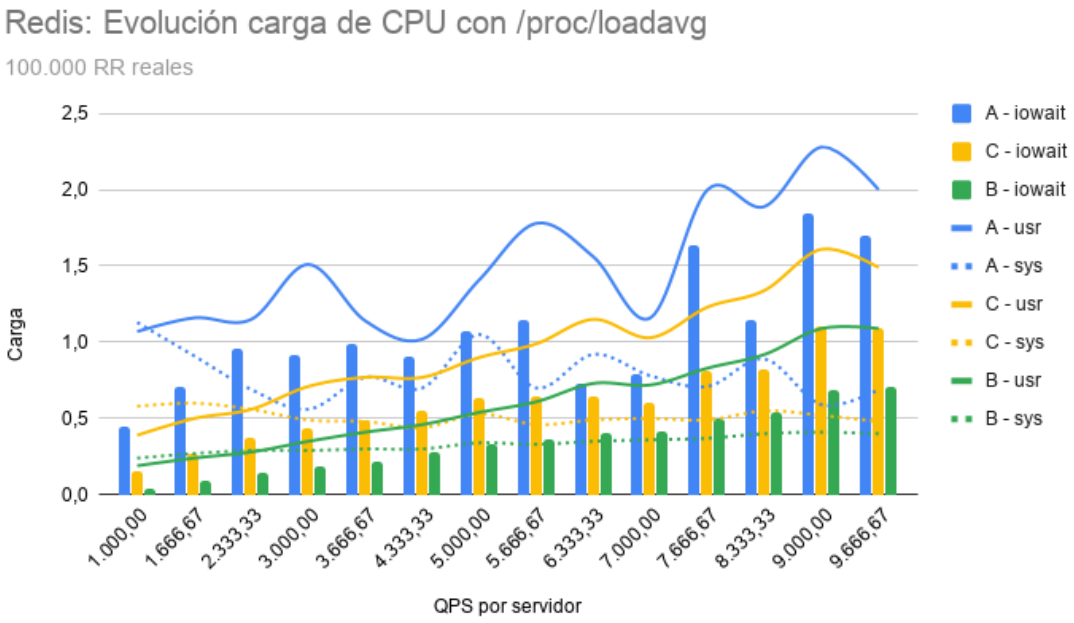

Figura 5.25: Tendencia de la carga de CPU para Redis entre sus 3 máquinas, con 100.000 RR reales.

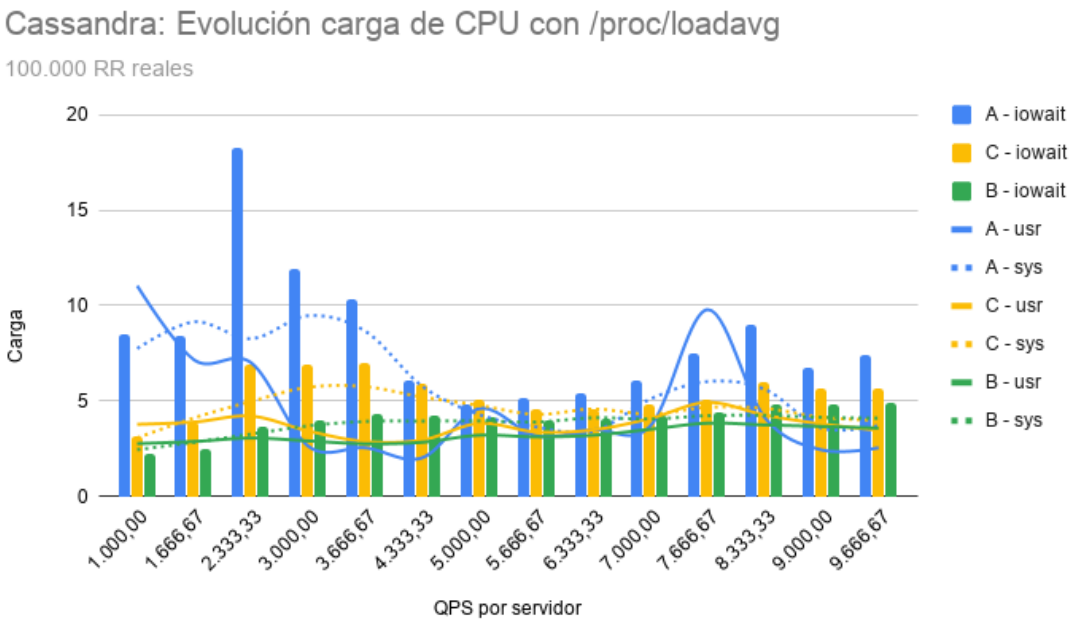

Figura 5.26: Tendecia de la carga de CPU para Cassandra entre sus 3 máquinas, con 100.000 RR reales.

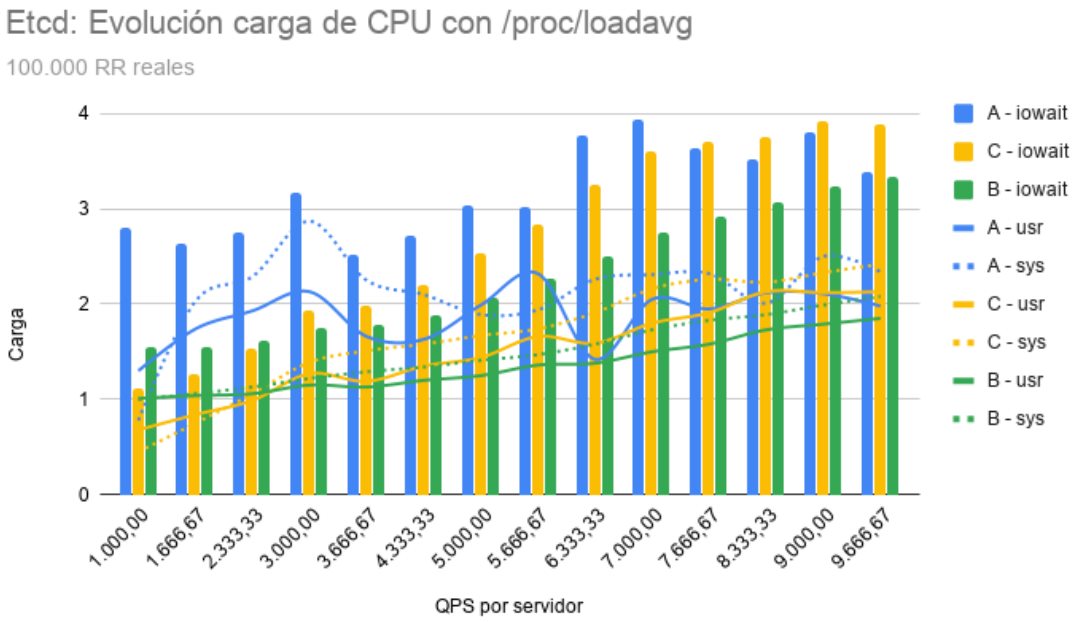

Figura 5.27: Tendecia de la carga de CPU para Etcd entre sus 3 máquinas, con 100.000 RR reales.

Redis, Cassandra y Etcd utilizan más recursos del CPU a medida que las pruebas son más pesadas. En particular Cassandra utiliza más recursos y sobrecarga la cola de tareas, esto puede ser debido a la implementación y el lenguaje Java.

Una explicación para el acceso a disco es que cada cliente Kvs Dns escribe un log de errores e información cada cierto tiempo. Lo mismo ocurre con Etcd.

5.3.5. Porcentaje de respuesta

Para medir el porcentaje de respuesta se separan las mediciones en categoría de valores auto-reportados por cada servidor/máquina y en el que recibe las respuestas.

Los casos más representativos son las mediciones de 200.000 RR. Los demás se encuentran en el anexo (Anexo).

Valores auto-reportados: Dataset de 200.000 RR sintéticos

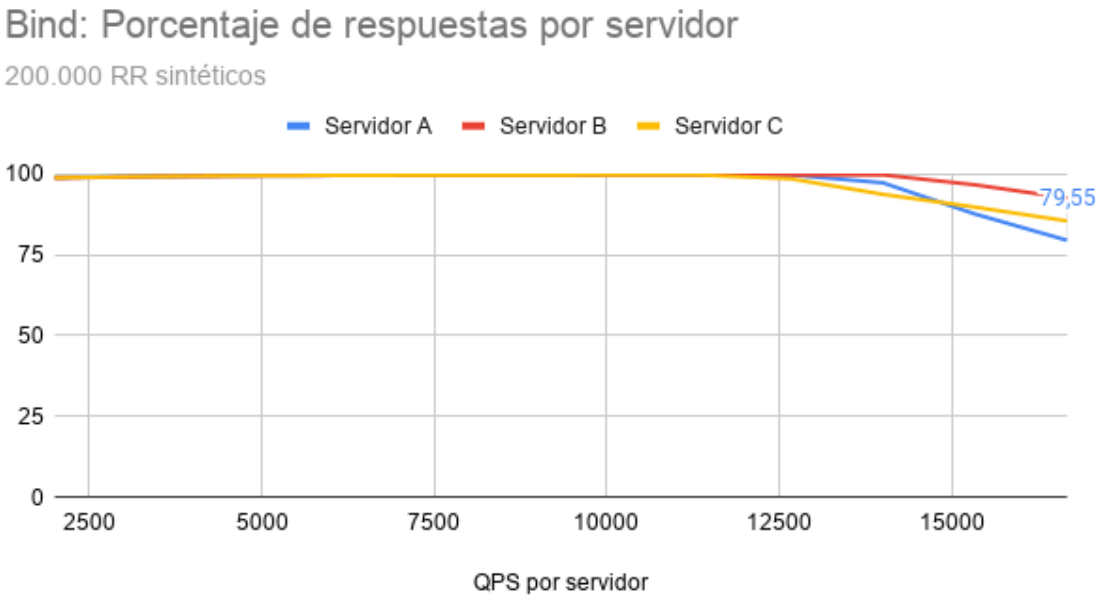

Figura 5.28: Porcentaje de respuestas bind entre sus 3 máquinas, con 200.000 RR sintéticos.

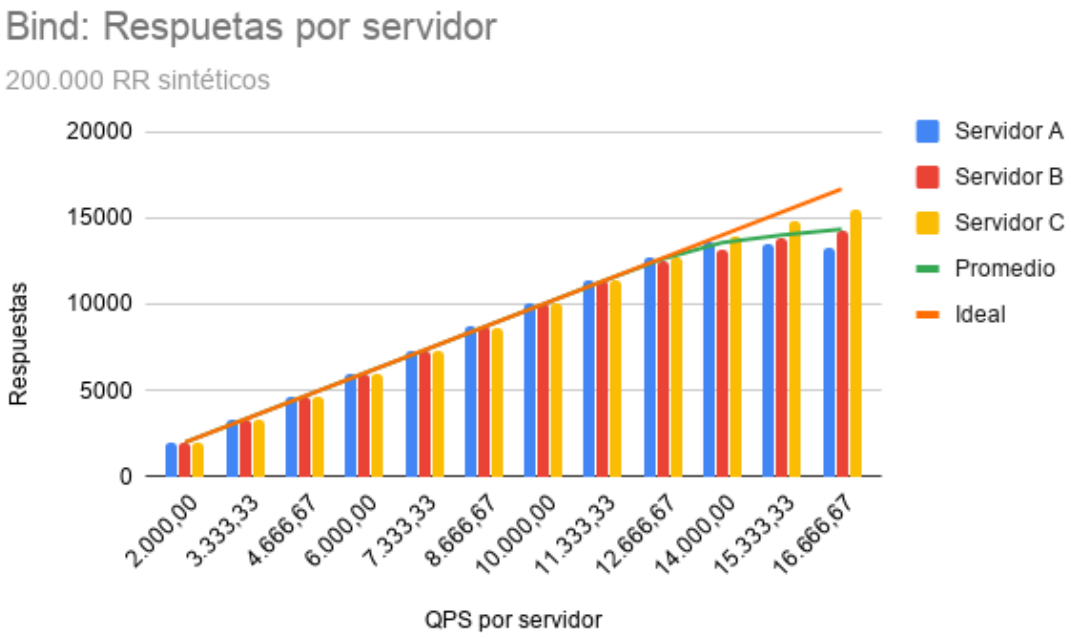

Figura 5.29: Respuestas bind entre sus 3 máquinas, con 200.000 RR sintéticos.

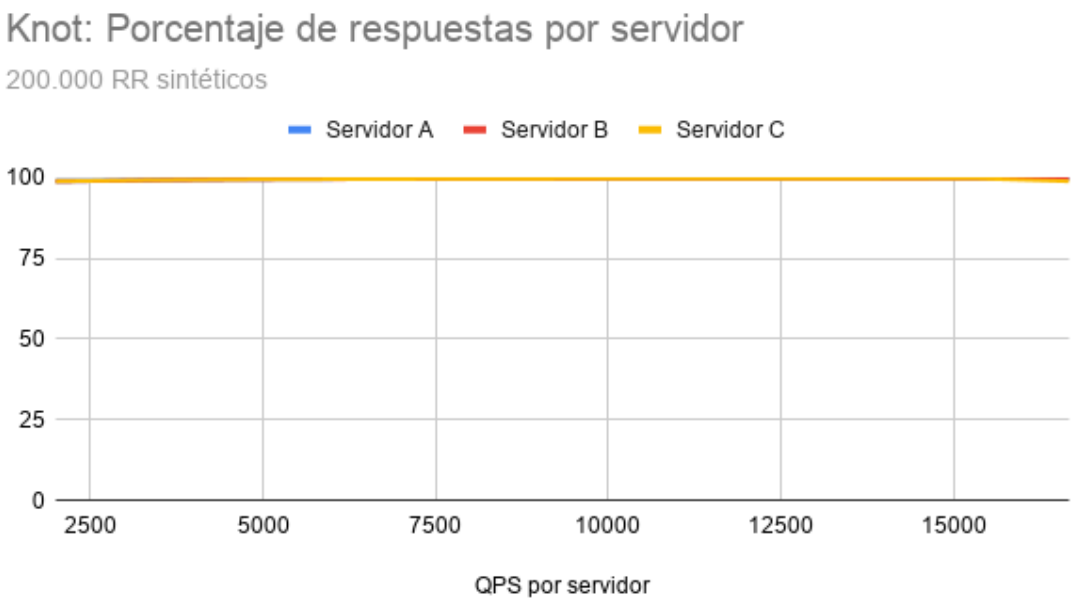

Figura 5.30: Porcentaje de respuestas knot entre sus 3 máquinas, con 200.000 RR sintéticos.

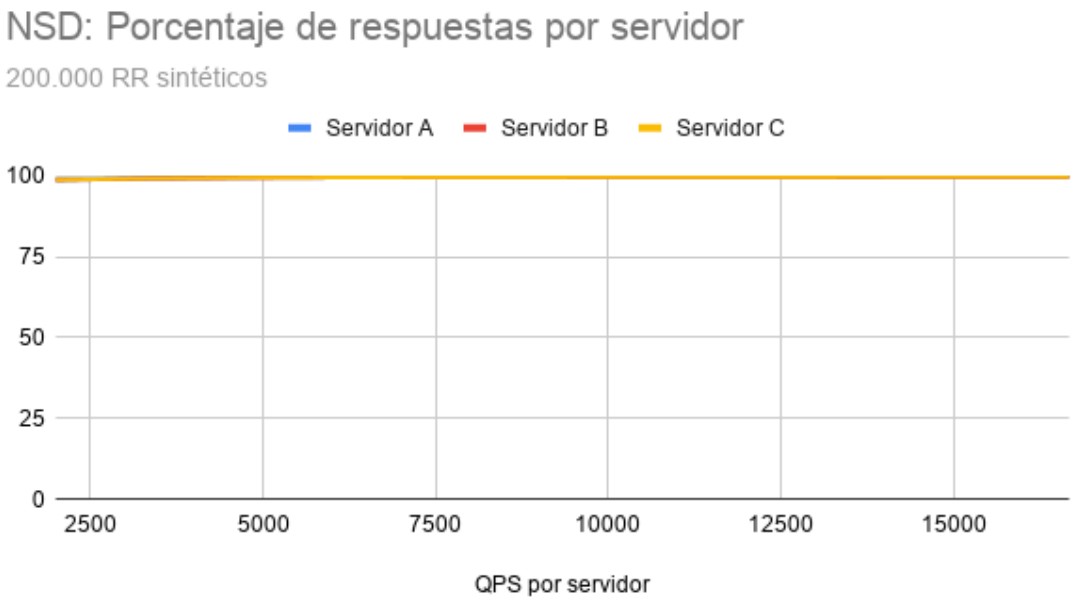

Figura 5.31: Porcentaje de respuestas NSD entre sus 3 máquinas, con 200.000 RR sintéticos.

Knot y NSD muestran el mismo comportamiento.

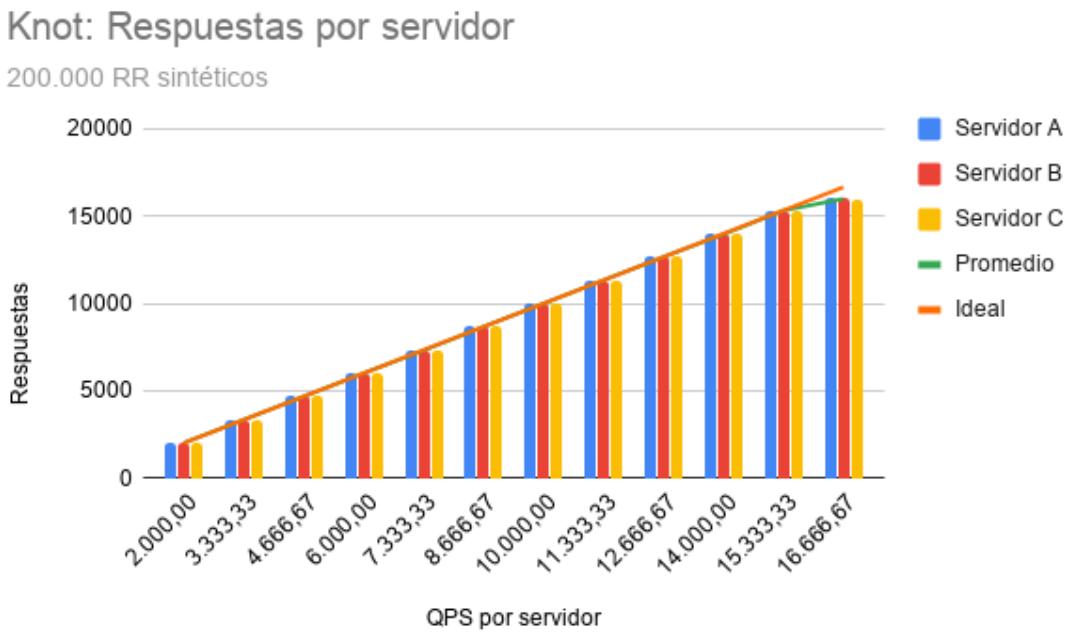

Figura 5.32: Respuestas Knot entre sus 3 máquinas, con 200.000 RR sintéticos.

Los gráficos de respuesta para Bind y Knot muestran que ambos alcanzan cierta capacidad máxima de respuesta (para esta configuración) más allá de 14.000 consultas por segundo.

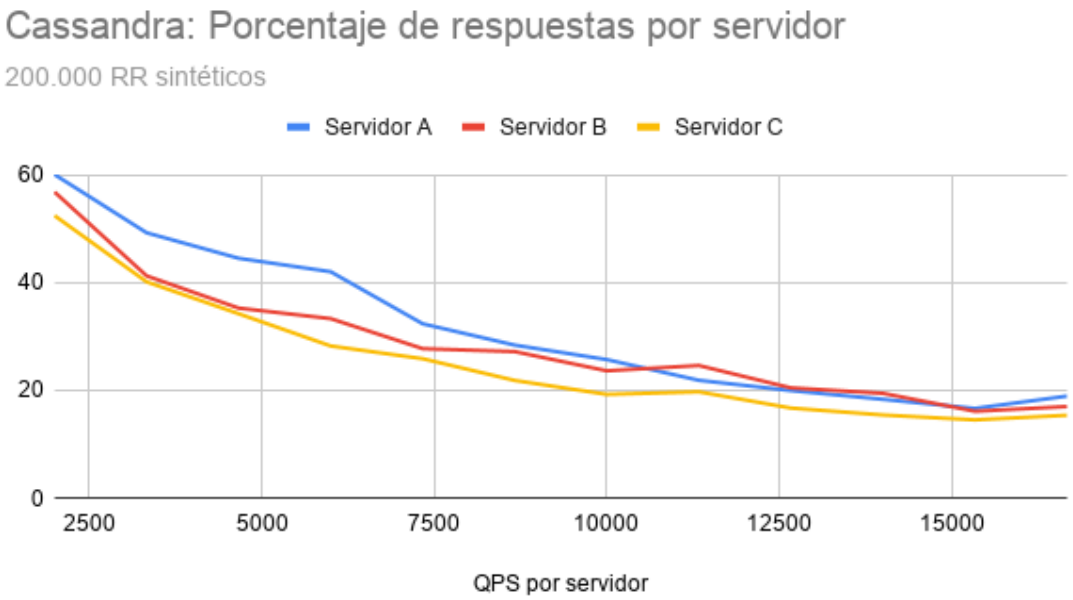

Figura 5.33: Porcentaje de respuestas Cassandra entre sus 3 máquinas, con 200.000 RR sintéticos.

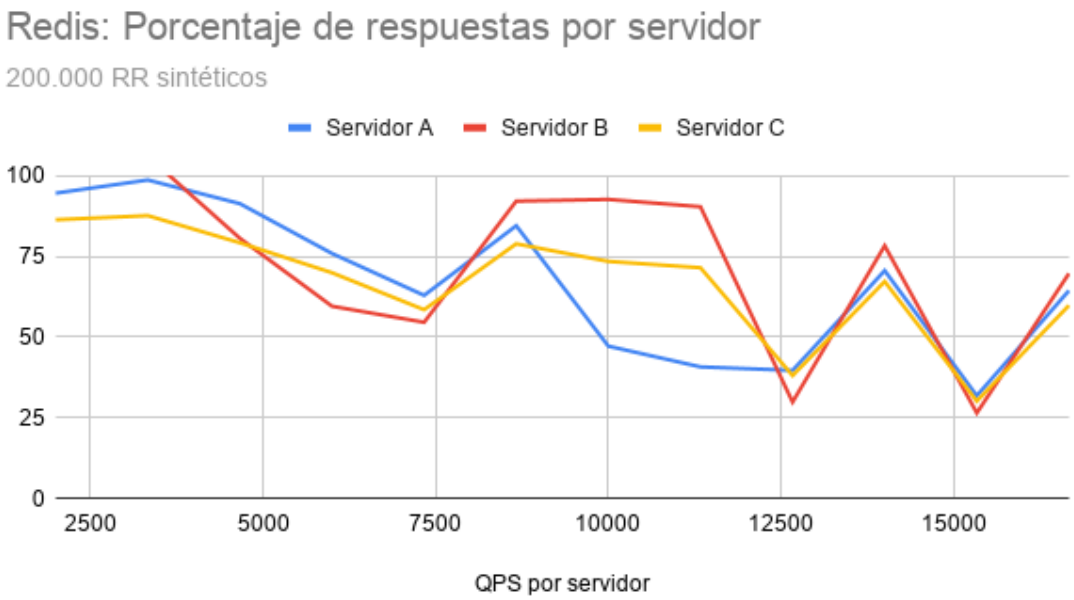

Figura 5.34: Porcentaje de respuestas Redis entre sus 3 máquinas, con 200.000 RR sintéticos.

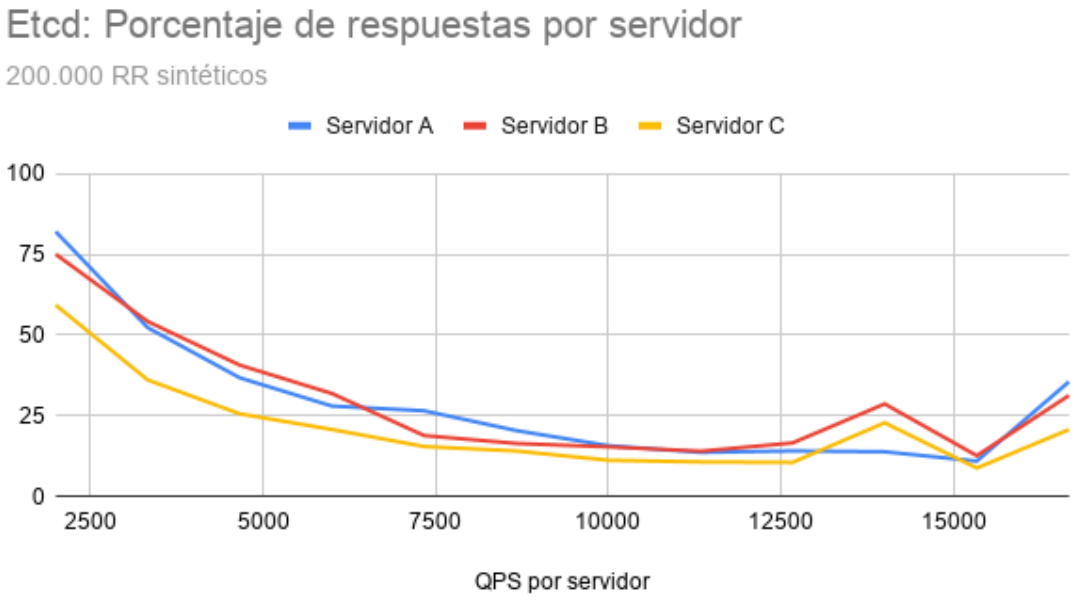

Figura 5.35: Porcentaje de respuestas Etcd entre sus 3 máquinas, con 200.000 RR sintéticos.

A partir de los gráficos de esta sección se puede observar que los servidores tradicionales logran responder la mayor cantidad de consultas. Después sigue Redis, luego Etcd y finalmente Cassandra.

Valores auto-reportados: Dataset de 200.000 RR reales

En el caso de Bind, Knot y NSD los resultados fueron consistentes, respondiendo el mismo porcentaje por carga. El gráfico siguiente es representativo de los servidores normales.

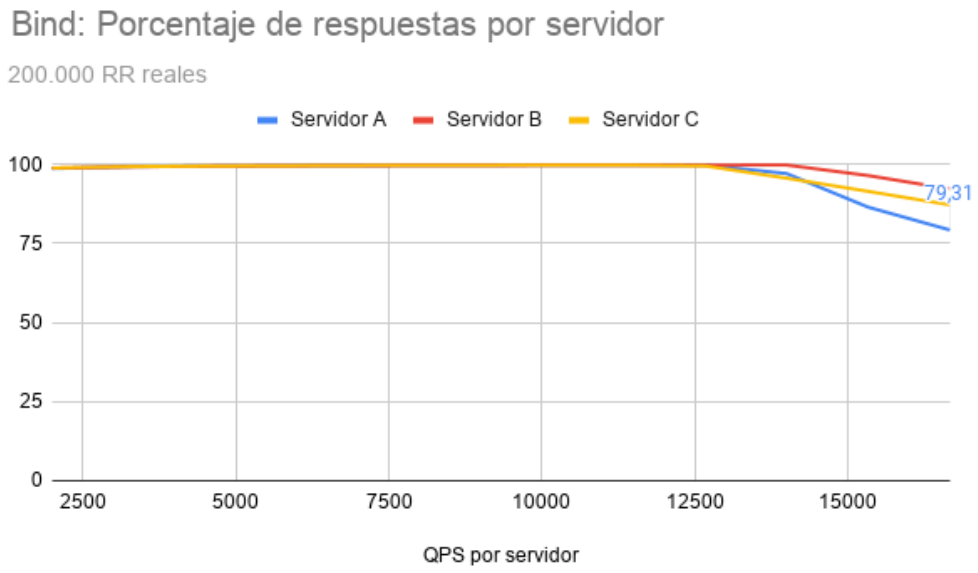

Figura 5.36: Porcentaje de respuestas bind entre sus 3 máquinas, con 200.000 RR reales.

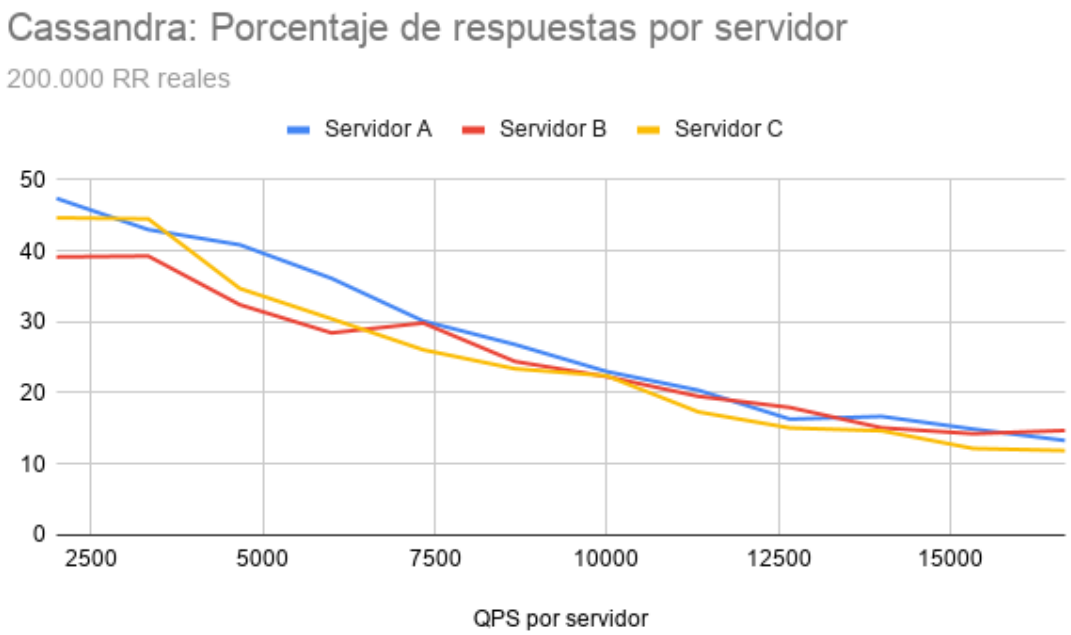

Figura 5.37: Porcentaje de respuestas Cassandra entre sus 3 máquinas, con 200.000 RR reales.

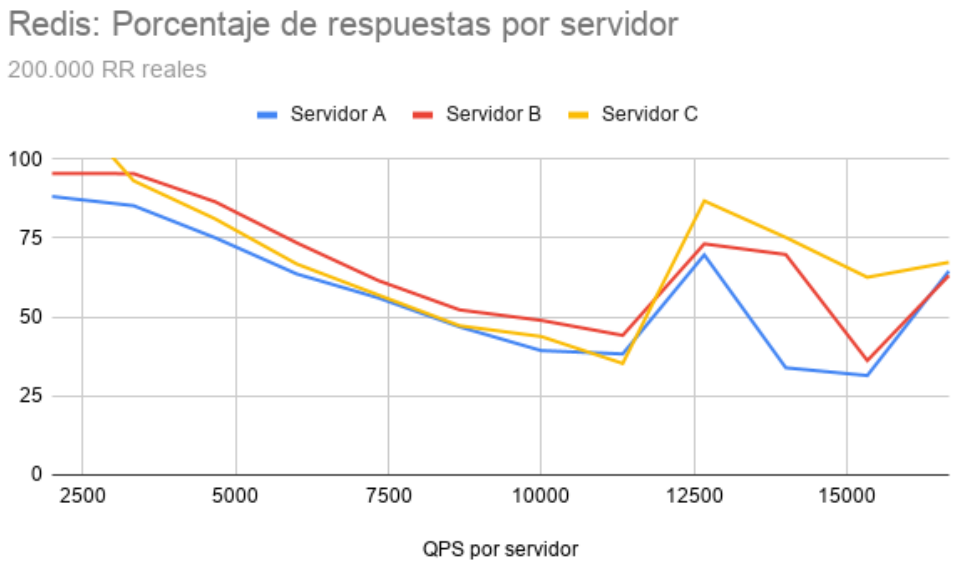

Figura 5.38: Porcentaje de respuestas Redis entre sus 3 máquinas, con 200.000 RR reales.

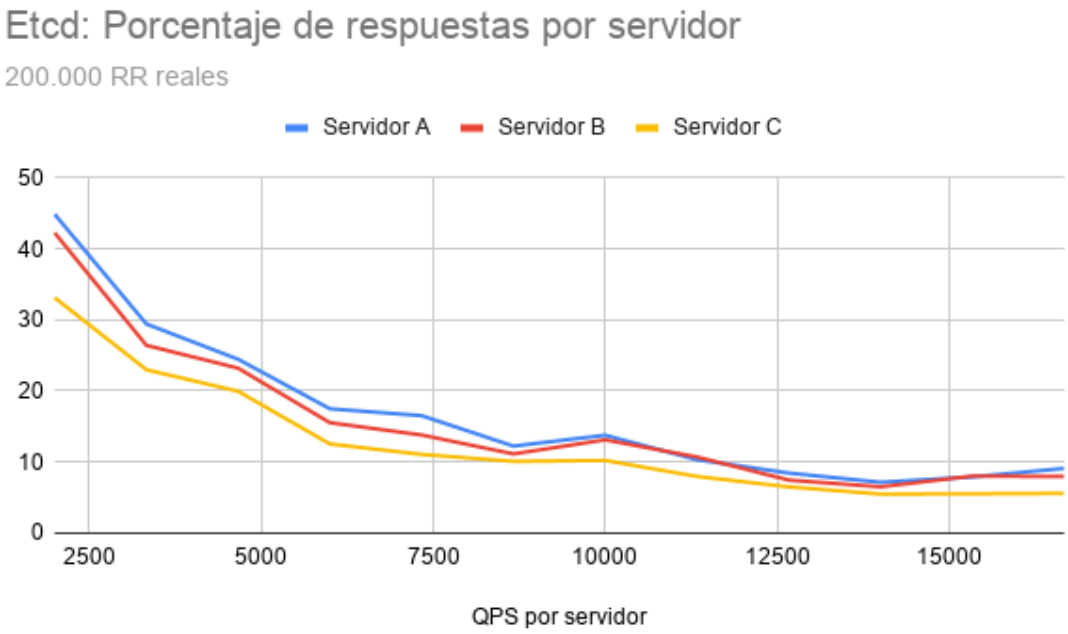

Figura 5.39: Porcentaje de respuestas Etcd entre sus 3 máquinas, con 200.000 RR reales.

En el caso de Etcd y Cassandra se ve que se pierde cerca de un 20 % inicial de la capacidad de respuesta en comparación con los datos sintéticos. Redis pierde su capacidad de respuesta a mitad de la prueba.

Porcentajes medidos por Wireshark: Dataset de 200.000 RR sintéticos

Figura 5.40: Porcentaje de respuestas recibidas por el *requester* medido por Wireshark y con datos sintéticos

Porcentajes medidos por Wireshark: Dataset de 200.000 RR reales

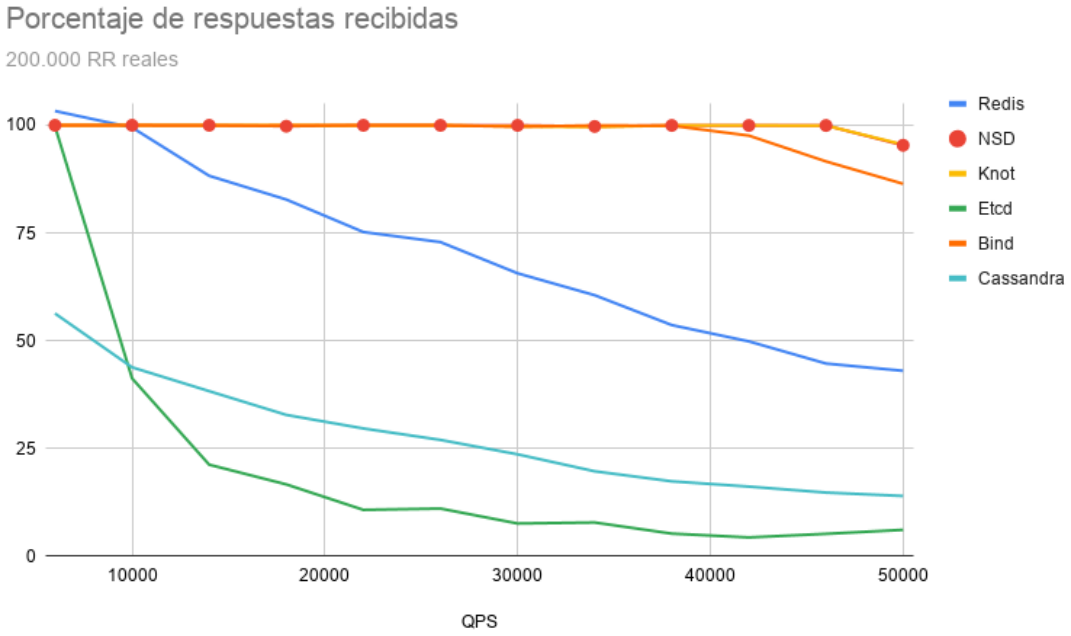

Figura 5.41: Porcentaje de respuestas recibidas por el *requester* medido por Wireshark y con datos reales

Wireshark mide el conjunto de respuestas recibidas por el cluster DNS. Vemos que Redis se mantiene arriba de Cassandra y Etcd pero ahora este último es el con menos capacidad de transacciones por segundo.

NSD y Knot tienen mayor capacidad que Bind según los resultados.

Síntesis de resultados para 200.000 RR

Como se observa en las figuras (5.40) y (5.41) el desempeño de los servidores DNS parece ser consistente entre ambas pruebas. En cambio para Etcd y Cassandra el la cantidad de datos por llave es un factor importante. En el caso de Cassandra se deben ir a buscar en más nodos y para Etcd una llave puede almacenar texto muy largo, lo cual en ambos casos vuelve la operación más cara y así bajan las transacciones por segundo. Redis mantiene cierto nivel de consistencia.

Resultados adicionales

El porcentaje de respuesta para los dataset de 100.000 RR y para algunos resultados con 2.000.000 de RR se encuentran en el anexo (cA)(cB). Se rescata que la cantidad de paquetes que puede responder cada programa alcanza un límite. Bind cerca de 15.000 qps, Knot y NSD cerca de 18.500 qps. Redis se comporta de forma errática, Cassandra alcanza 2.500 qps y Etcd 300 qps.

5.3.6. Tamaño promedio de paquetes

En esta sección es de interés determinar que sucede con el tráfico entre servidores. Esto es válido para las KVS y aporta información sobre su desempeño en general. En particular ver si el tráfico es puramente consultas y respuestas DNS.

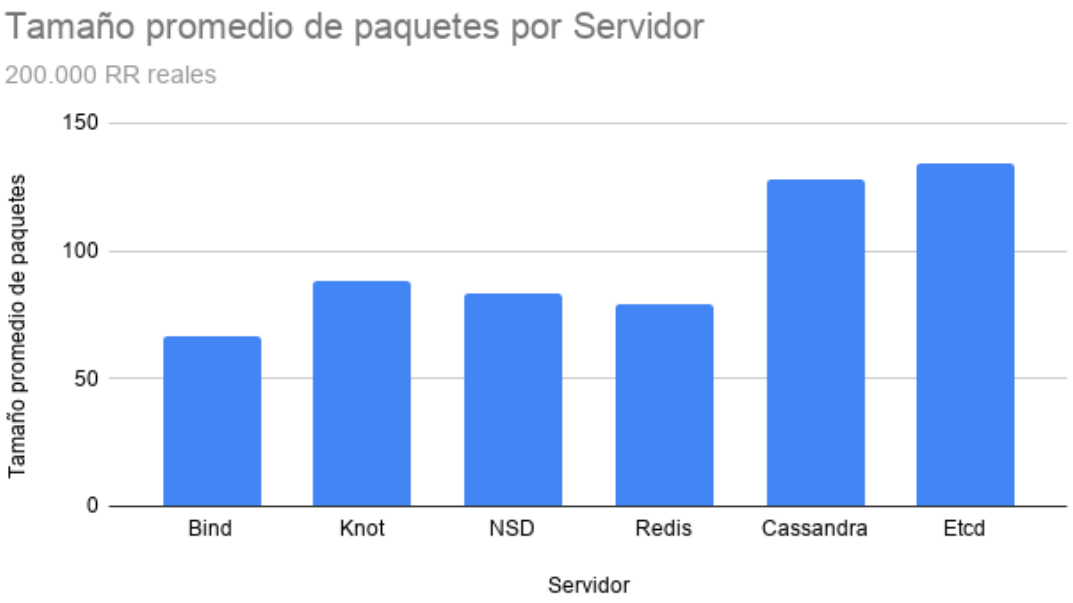

Figura 5.42: Tamaño promedio de paquetes enviados por la interface de red principal.

Se ve que el tamaño aumenta para Cassandra y Etcd por sobre los demás. Esto se debe a que existe tráfico adicional de red generado por estas bases de datos. El cual puede ser para confirmar la validez de un valor, conseguir valores no locales, tráfico de coordinación del cluster, etc.

Estas mediciones no consideran la etapa de adición de registros (subir las zonas a Kvs Dns), la cual haría que los valores aumentasen para las KVS.

5.3.7. Porcentaje de respuestas con una falla controlada

Los valores que se presentan en las figuras continuación fueron reportados por el *requester*.

Caso de estudio: Dataset de 200.000 RR reales auto-reportado

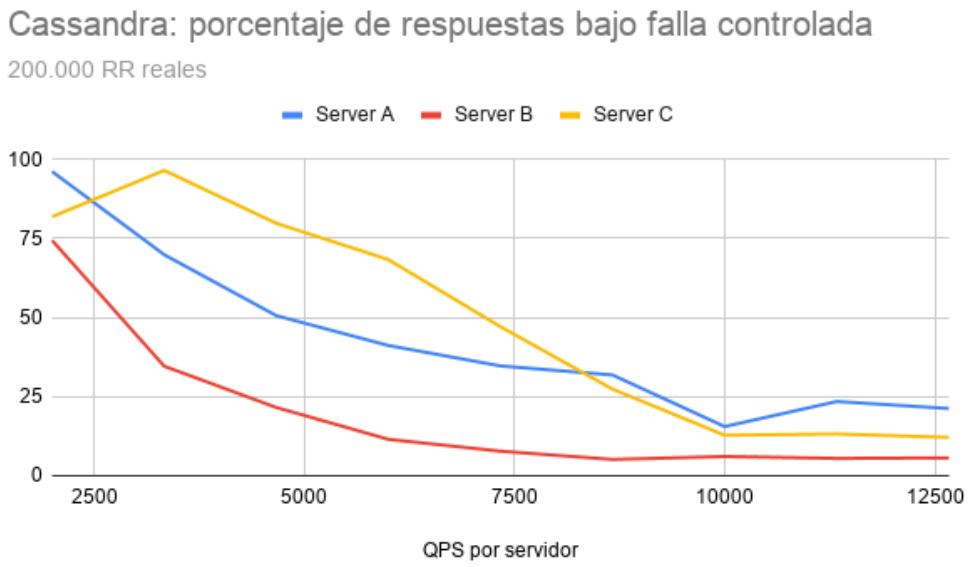

Figura 5.43: Porcentaje de respuestas bajo falla controlada para Cassandra, por servidor, para 200.000 RR reales.

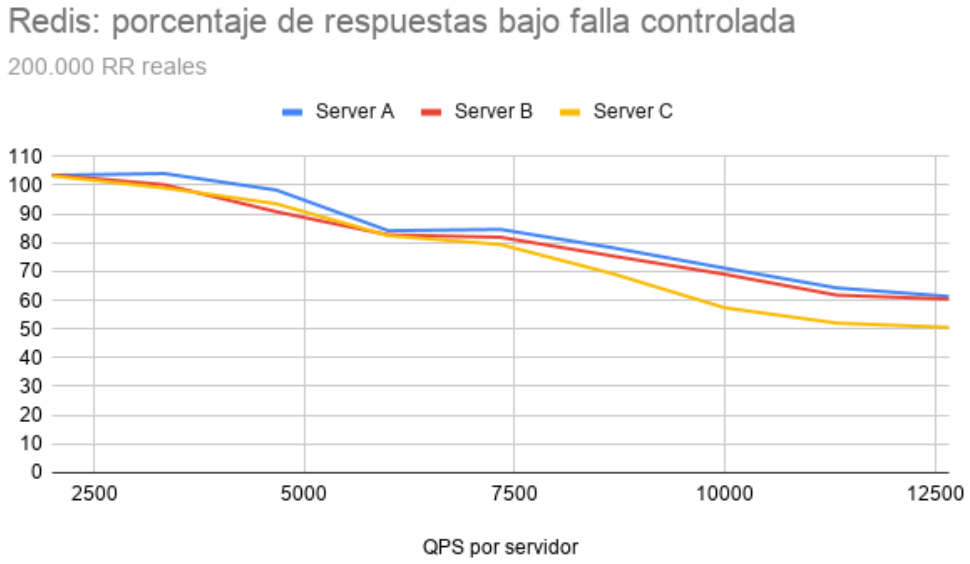

Figura 5.44: Porcentaje de respuestas bajo falla controlada para redis, por servidor, para 200.000 RR reales.

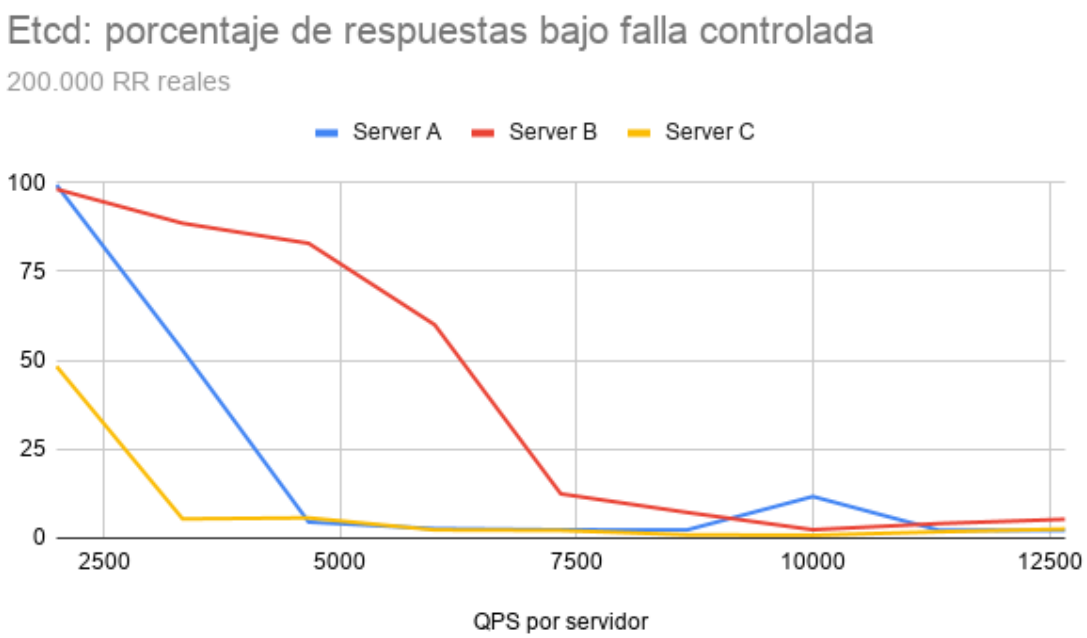

Figura 5.45: Porcentaje de respuestas bajo falla controlada para Etcd, por servidor, para 200.000 RR reales.

Knot representa las tendencias de Bind y NSD.

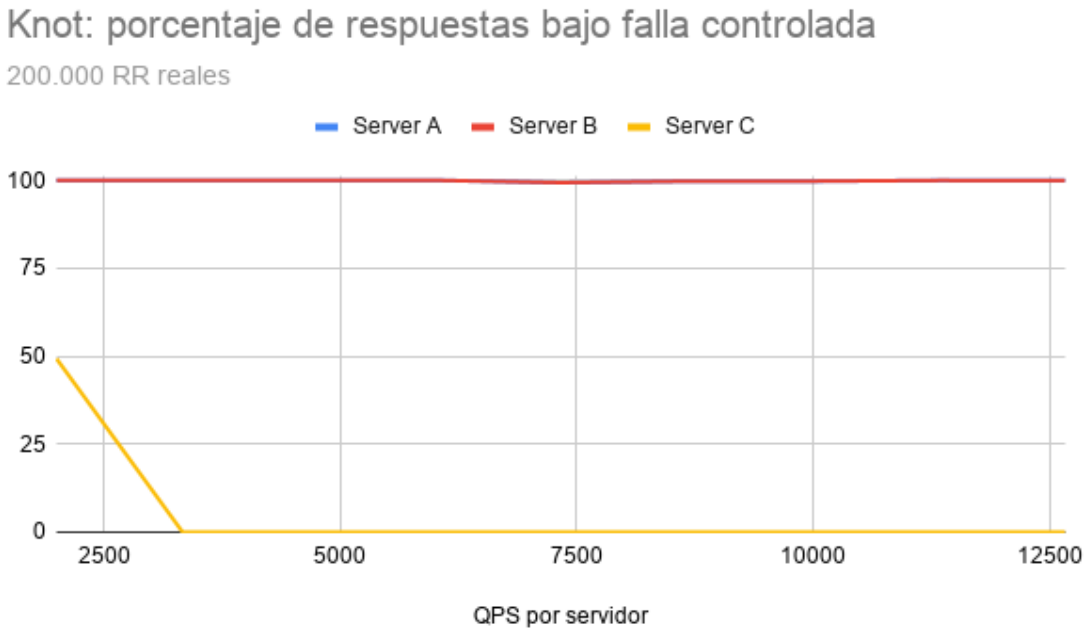

Figura 5.46: Porcentaje de respuestas bajo falla controlada para Knot, por servidor, para 200.000 RR reales.

Al eliminar uno de los nodos de las bases de datos KVS el cliente Kvs Dns aún puede consultar a los nodos vecinos. Por esto sus máquinas aún responden. En particular para Redis al contar con 6 nodos hay menos posibilidades de que una falla de software inhabilite el sistema.

Para los servidores DNS tradicionales es claro que si se elimina uno de los procesos el servidor dejará de responder. Es precisamente esto lo que sucede con Bind, Knot y NSD donde su capacidad de respuesta disminuye en un 33 %.

Caso de estudio: Dataset de 200.000 RR sintéticos medido por Wireshark

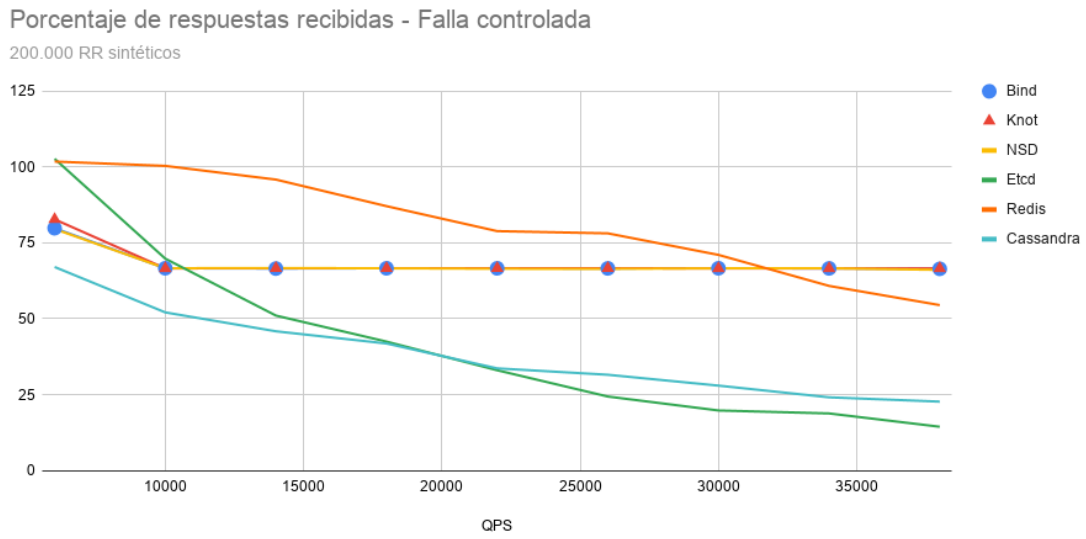

Figura 5.47: Porcentaje de respuestas bajo falla controlada por servidor, para 200.000 RR sintéticos.

Como se observó en el caso de estudio anterior los servidores DNS tradicionales ven su capacidad de respuesta cortada a un tercio. El resto de comportamientos reportados es consistente con las mediciones auto-reportadas con datos reales, donde se ve que Redis se mantiene resistente ante fallas a un nodo.

5.3.8. Comparación de porcentaje de respuestas normal y con fallas para 200.000 RR

Los siguientes datos de las figuras fueron medidos con Wireshark desde el *requester*.

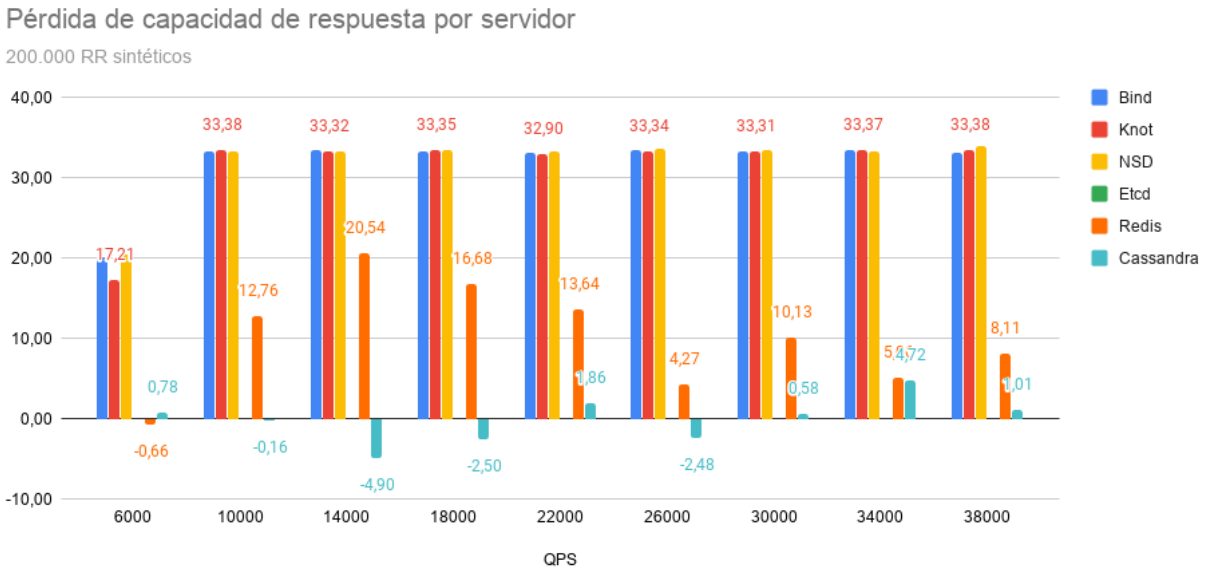

Figura 5.48: Pérdida de capacidad de respuesta por servidor, para 200.000 RR sintéticos.

Con este gráfico se observa cuanta capacidad de respuesta pierde el cluster en conjunto frente a la falla de uno de sus procesos. Se observa que para Etcd y Cassandra no hay mucha diferencia y que por otro lado Redis no tiene una pérdida constante.

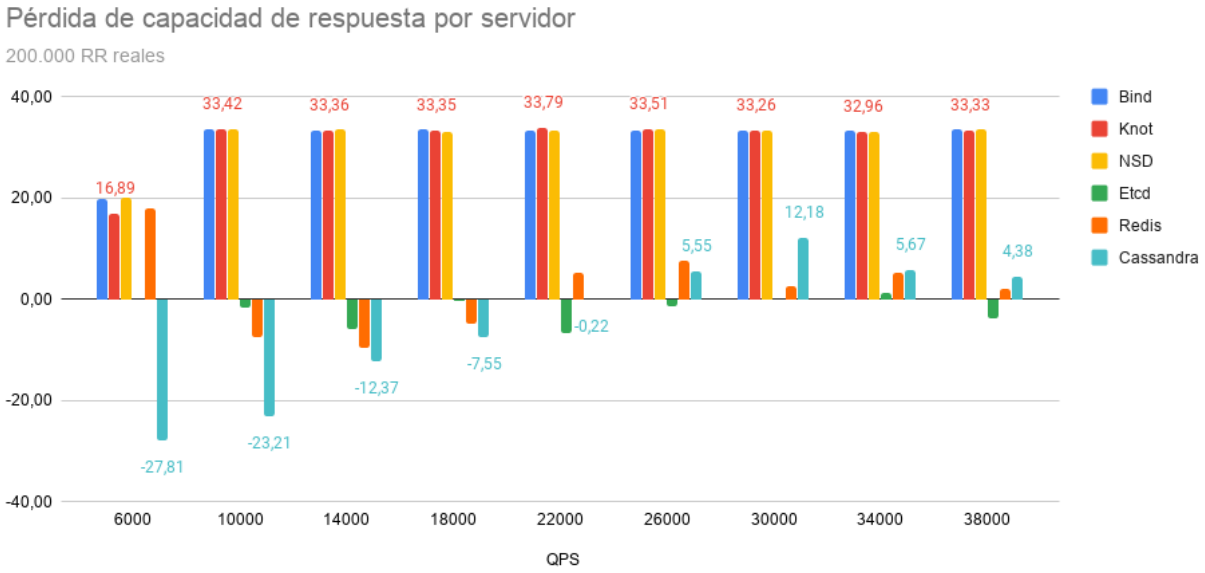

Figura 5.49: Pérdida de capacidad de respuesta por servidor, para 200.000 RR reales.

Sin embargo en el escenario con datos reales se puede ver que Cassandra gana capacidad al eliminar un nodo y Redis se mantiene con rendimientos similares.

5.3.9. Tiempo de carga de datos por servidor

Para cargar los datos los servidores tradicionales leen su archivo de configuración y los archivos de zona. En cambio para las bases de datos Kvs se necesita interactuar con un cliente. En este caso se utilizó la herramienta para subir zonas de Kvs Dns y los resultados fueron los siguientes:

Servidor	Tiempo para subir datos
Bind	3 - 8 minutos
Knot	3 - 8 minutos
NSD	3 - 8 minutos
Redis	20 - 40 minutos
Cassandra	45 - 70 minutos
Etcd	150 minutos y más

Tabla 5.4: Orden de Tiempos de carga de datos para 200.000 RR

Estas grandes diferencias ocurren debido a que la utilidad para subir zonas de Kvs Dns lee los archivos de forma local y los envía a los tres nodos de cada cluster KVS por protocolo TCP.

En particular para Etcd se realiza en la misma utilidad de subida operaciones de conjuntos de forma local, es decir si se sube un dominio con 4 registros A se inserta 4 veces pero se recupera también 4 el valor de la llave. Esto ocurre porque se verifica que no existan valores repetidos en la llave y para crear el nuevo *conjunto* uniendo los valores únicos en un sólo valor. Esto no ocurre en Redis que realiza estas operaciones a nivel de servidor.

5.3.10. Resultados para 2.000.000 RR en cada servidor

Adicionalmente se realizaron pruebas para exigir al máximo los sistemas y máquinas disponibles, con 2 millones de RR tanto sintéticos como de los datos reales recuperados. Algunos resultados se muestran en el anexo (cB)

Las tendencias se mantienen en su mayoría, en particular **Etcd falla** y sobrecarga el sistema. Bind, Knot, Redis y NSD trabajaron a los **límites del sistema** utilizando entre 1GB y 2GB de RAM y 100 MB de swap. Cassandra utilizó 200 MB en promedio de swap y mantuvo su uso de memoria.

5.4. Síntesis de resultados

Los servidores Knot, Bind y NSD superan con mucha ventaja, respecto al porcentaje de respuestas, al cliente Kvs Dns y las bases de datos distribuidas bajo una carga normal. Además como se vio que su uso de memoria es consistentemente menor que el de las bases de datos distribuidas. Bind parece utilizar más memoria a medida que los archivos de zona aumentan y también tiene una baja de rendimiento cerca de las 15.000 consultas por segundo.

La única base de datos distribuida que se acerca al rendimiento tanto en memoria como en capacidad de respuesta es Redis, probablemente gracias a su implementación en memoria con múltiples tablas de hash. Sin embargo queda muy en desventaja en comparación a los servidores DNS a medida que el tráfico de red aumenta.

Cassandra no funciona bien en tiempo real para las cargas a la cual se le somete en este experimento. Tampoco Etcd y dado que los valores son procesados por el cliente Kvs Dns entonces el uso de memoria crece, dado que Go utiliza un recolector de basura para liberar memoria.

Etcd como mantiene sus valores en disco, gracias al B+Tree, tiene la peor capacidad de respuesta en tiempo real para la mayoría de las pruebas.

La ventaja de las bases de datos distribuidas en comparación con los servidores DNS es que para fallar y dejar de responder múltiples procesos deben fallar al mismo tiempo. Así logran responder aún cuando uno de los nodos del cluster se detiene, pero con sus pobres capacidades de respuesta.

Es difícil distinguir para las KVS cual papel juega el cliente intermediario en los tiempos de respuesta. Además como éste se compone de drivers externos ellos también tienen un papel en los tiempos de respuesta. Por ejemplo el driver de Redis si no encuentra una llave en un nodo pregunta a uno hasta encontrarla. Se debe considerar también que el cliente no posee un cache local por lo cual siempre consulta a su base de datos.

Capítulo 6

Conclusión y Trabajo futuro

En este capítulo se concluye sobre el objetivo general de la memoria. Después se discuten resultados particulares y puntos de mejora como trabajo futuro.

6.1. Conclusión

Al término de este trabajo se logró cumplir el objetivo principal de comparar servidores de nombre DNS con bases de datos Key-Value-Store.

Para esto se creó un intermediario que permite realizar consultas DNS a las KVS, el cual utiliza concurrencia con Go y diferentes programas de conexión a las bases de datos distribuidas.

Además se creó un nuevo benchmark para medir el rendimiento de servidores de nombre, con herramientas modernas como Ansible, Wireshark y PCAP. Este benchmark es una mejora en comparación a Knot Benchmark dado que se permite medir al mismo tiempo, uso de CPU, memoria, consultas por segundo desde los servidores y respuestas recibidas desde el cliente orquestador. Al estar escrito con YAML para Ansible es más mantenible que el código original escrito puramente en Bash.

Se seleccionaron los servidores DNS más populares,y tres tipos distintos de bases de datos distribuidas key-value. Resultó que Redis es un buen candidato para manejar una base de datos distribuida que responda a consultas DNS. Esto por su diseño como un caché rápido, su bajo uso en memoria primaria y estructuras de datos integradas. Además se verificó que Bind, Knot y Nsd poseen un rendimiento similar y competitivo entre sí que no se ve afectado por la distribución de datos en zonas.

Sin embargo no se logró obtener los resultados originales de Knot benchmark, ni un resultado similar al utilizarlo.

Uno de los grandes aprendizajes de este proyecto es utilizar múltiples sistemas diferentes sobre dos tipos de sistema operativo (Debian y CentOS). Además de crear programas con Golang siguiendo el estilo de código estándar.

6.2. Trabajo futuro

6.2.1. Mejora de cliente Kvs Dns

El cliente fue diseñado con tres partes principales o módulos. Estos no definen límites de cuantas goroutines definir por lo cual la carga puede crecer mucho si el CPU no es suficientemente rápido.

Dado que Kvs Dns utiliza diferentes herramientas externas es difícil realizar test unitarios. Estos no fueron agregados por limitaciones de tiempo y priorización de tareas.

Además para que cumpla con todas las especificaciones de los RFC se requiere profundizar su desarrollo.

6.2.2. Medición de uso del CPU

Se podría experimentar con un proceso en paralelo que mida constantemente el uso del CPU mientras ser ejecutan las pruebas, para que luego otro programa rescate sólo los valores relevantes. Esto puede proporcionar mediciones más precisas dado que el diseño actual mide sólo después de enviar las consultas.

6.2.3. Medición del uso de memoria

Expandir el análisis para indicar cuanta memoria utiliza el servidor en total, pero también cada proceso que lo compone. Esto sería particularmente útil para determinar si la base de datos distribuida o el cliente intermediario utiliza más memoria.

6.2.4. Extender el Kvs Dns Benchmark para probar más servidores, sistema operativo, etc

El benchmark fue construido utilizando Ansible de base. Esto permite que agregar tareas sea sencillo por lo cual agregar por ejemplo librerías para cada máquina, o soporte para versiones futuras de Go es fácil. Lo que significa más trabajo es probar en tiempo real tomando en cuenta las condiciones de uso reales.

Para poder usar el benchmark con sistemas como Windows server se deben replantear ciertas herramientas para medir el uso de CPU y memoria dado que estas confían en sistemas Unix. Se debe además de adaptar las instrucciones de terminal.

6.2.5. Mejorar tiempos de subida de datos

La utilidad de Kvs Dns para subir datos a clusters KVS fue escrita considerando que los módulos de driver de conexión con la BDD implementarían esto utilizando sus librerías especializadas. No obstante es probable que utilizando directamente el CLI de cada base de datos de forma local se pueda acelerar el poblado de cada una. Para esto es necesario que exista un programa que verifique la correctitud de la zona y luego lo traspase a la serie de instrucciones específicas.

6.2.6. Explorar ventajas de utilizar RR como unidad

Actualmente se utilizan zonas para manejar los Resource Records. Sin embargo al utilizar una BDD KVS se puede trabajar sólo con RR como unidad atómica para el mantenimiento del servidor de nombre.

Siglas

ACID Atomicidad, Consistencia, Aislamiento, Durabilidad. 4, 5, 13, 15

ACL Access Control Layer. 15

API Aplication Programming Interface. 10, 22, 28, 29

BASE Basicamente disponible, Estado Permisivo (Soft State), Eventualmente consistente. 5, 13

BDD Bases de datos. 1, 4, 6, 7, 13, 23, 25, 31, 77, 78

CAP Consistencia, Disponibilidad, Resistencia a particiones. 5, 7

CLI Command line interface. 32, 38, 77

CSV Comma Separated Values. 40

DNS Domain Name System. viii, 1–6, 12–16, 19–22, 25–28, 30, 33, 35–39, 41, 43, 46–49, 56, 57, 68, 72, 75, 76

DNSSEC DNS Security Extensions. 13, 15, 16, 18

DUT Device Under Test. 13

FQD Fuly Qualified Domain. 33

GNU GNU is not Unix. 11

IETF Internet Engineering Task Force. 5, 16, 25, 27

IP Internet Protocol. 1, 26, 29, 30, 33, 38, 39

IRQ Interruption Request. 35

KVS Key-Value-Store. 1, 7, 8, 22, 28, 32, 35–38, 43, 69, 72, 74–76, 78

MX Mail Exchange. 16, 29, 30

NoSQL Not only SQL. 1, 21

OSI Open Systems Interconnection model. 10, 11

PCAP Packet Capture. 10, 13, 37, 39, 76

QPS Queries per second. 44, 45

RAM Random Access Memory. 35, 74

RFC Request for comments. 14, 15, 25, 27

RR Resource Records. 15, 16, 20, 25, 26, 29–35, 37, 40, 43, 46, 54, 74

RTT Round Trip Time. 35

SOA Start of Authority. 15, 29, 30, 32, 33

SQL Structured Query Language. 1, 5, 6

SSH Secure Shell. 9, 38

TCP Transmission Control Protocol. 4, 14, 74

TLD Top Level Domain. 3, 6, 13

TTL Time to Live. 13, 30, 31

UDP User Datagram Protocol. 4, 26, 27, 35, 37, 38

YCSB Yahoo Cloud Serving Benchmark. viii, 13, 14, 26

Glosario

anycast Anycast es un método de ruteo y direccionamiento de red donde una sola dirección tiene múltiples caminos posibles o internamente más de una ruta.. 39

backend Corresponde al código que se utiliza del lado de un servidor, pero también puede referirse al servidor mismo.. 25

benchmark Estándar o serie de estándares utilizado como punto de referencia para evaluar rendimiento u otro nivel de calidad.. i, 12, 42, 44, 45

cache Es un componente de hardware o software que almacena datos con el objetivo de que peticiones futuras puedan ser servidas más rápido. Estos datos suelen ser el resultado de un cómputo realizado o una copia de un valor guardado en una memoria de más difícil acceso.. 7

cluster Sistema computadoras interconectadas que trabajan en conjunto, para manejar bases de datos, calcular valores, etc. Desde afuera parecen un solo sistema.. 8, 69

dirección MAC Media Access Control es un identificador único asignado a un controlador de interfaz de red. Es usado para identificar a máquinas dentro de una red.. 38

dominios a Top-Level Corresponden a por ejemplo el .CL, .COM, .MX, .ORG. Leyendo de derecha a izquierda un dominio (FQD) es el primero que se lee.. 3

firmas digitales Es un esquema criptográfico que se utiliza para verificar la autenticidad de un mensaje, es decir verificar que este fue escrito por el autor.. 16

framework Es una colección de programas creados para cumplir una tarea en conjunto, que permiten a los programadores escribir código más facilmente y de forma estructurada.. 13

Key-Value-Store Son base de datos que modela sus datos como pares llave valor. Puede tener o no persistencia de datos.. i, 1

master zone files Archivo que describe una zona. Contiene al menos un registro SOA, un registro NS y un registro A.. 6

nube (o Cloud) La nube es un término utilizado para describir una red de servidores a nivel global, cada uno con su función única. La nube no es una entidad física, si no una red de máquinas remotas alrededor del mundo que se encuentran interconectadas para operar como un ecosistema.. i, 9

open Source Open Source software son programas que tienen su código disponible para que el público lo lea, y pueda modificarlo. Su uso comercial está normado bajo diferentes licencias.. 5

resolver Un servidor dns que no es dueño de una zona (necesariamente), si no que se encarga de consultar a otros servidores y guardar temporalmente sus respuestas para contestar consultas de clientes.. 3

servidor Proxy Un servidor proxy actúa como una compuerta entre dos puntos de una red, por ejemplo un usuario y el internet. Proveen funcionalidades como agregar capas de seguridad, privacidad, registro de tráficos, etc.. 39

servidores DNS a nivel root Manejan la primera línea de servidores dns que conocen los dominios de nivel más alto o TLD. Son 13 grupos de servidores en el mundo.. 3

sharding Sharding es el proceso de realizar particionamiento horizontal de una base de datos, por ejemplo todas las filas con id mayor a cinco mil son trasladadas a otra instancia de la base de datos.. 23

sistema de nombres de dominio El sistema de nombres de dominio maneja la asociación dominio con su dirección IP. Esto permite a los usuarios de internet memorizar dominios en lugar de direcciones para acceder a recursos en la red.. i, 1

swap El kernel de linux utiliza espacio en memoria secundaria (normalmente un disco) para guardar datos de memoria principal y liberar espacio en ella. Esto se hace en un espacio reservado llamado SWAP.. 35

zona Una zona corresponde a un dominio y sub-dominios. Por ejemplo google.com es la zona y www.google.com, mail.google.com, ns1.google.com y ns2.google.com, delegated.google.com forman parte de ella.. 3

Bibliografía

[1] Akamai. Dns trends and traffic. https://www.akamai.com/es/es/why-akamai/dns-trends-and-traffic.jsp, Mayo 2020. [Consulta en Agosto, 2020].

[2] Red Hat Ansible. How ansible works. https://www.ansible.com/overview/how-ansible-works. [Consulta en Junio, 2020].

[3] Etcd Authors. Data model - physical view. https://etcd.io/docs/v3.4.0/learning/data_model/, Junio 2020. [Consulta en Agosto, 2020].

[4] Philipe Biondy. Credits. https://scapy.readthedocs.io/en/latest/backmatter.html. [Consulta en Agosto, 2020].

[5] Philipe Biondy. Network packet forgery with scapy. https://scapy.net/conf/scapy_pacsec05.pdf. [Consulta en Agosto, 2020].

[6] Google Official Blog. Google public dns: 70 billion requests a day and counting, March 2012. [Consulta en Diciembre, 2019].

[7] Jamesm Linux Journal Bottomley. Understanding cache. https://www.linuxjournal.com/article/7105. [Consulta en Marzo, 2020].

[8] E. F. Boza, C. San-Lucas, C. L. Abad, and J. A. Viteri. Benchmarking key-value stores via trace replay. In *2017 IEEE International Conference on Cloud Engineering (IC2E)*, pages 183–189, April 2017.

[9] CAIDA. Ipv4 routed /24 dns names dataset. https://www.caida.org/data/active/ipv4_dnsnames_dataset.xml, Julio 2020. [Consulta en Agosto, 2020].

[10] Apache Cassandra. Query driven modeling. https://cassandra.apache.org/doc/latest/data_modeling/intro.html#query-driven-modeling. [Consulta en Agosto, 2020].

[11] NIC Chile. Estadísticas. https://www.nic.cl/estadisticas/index.html. [Consulta en Abril, 2020].

[12] NIC Chile. Memoria conmemorativa 30 aniversario del .cl. https://www.nic.cl/acerca/memoria30/index.html, Sept 2017. [Consulta en Agosto, 2020].

[13] Matthew Prince Cloudflare. Load balancing without load balancers. https://blog.cloudflare.com/cloudflares-architecture-eliminating-single-p/, Junio 2013. [Consulta en Agosto, 2020].

[14] Internet Systems Consortium. Bind implementation. https://gitlab.isc.org/isc-projects/bind9, Sept 2019. [Consulta en Septiembre, 2019].

[15] Internet Systems Consortium. Dns related rfcs. https://www.bind9.net/rfc, Sept 2019. [Consulta en Octubre, 2019].

[16] Internet Systems Consortium. History of bind. https://www.isc.org/bindhistory/, August 2020. [Consulta en Agosto, 2020].

[17] Redins contributors. Redins github repository. https://github.com/arvancloud/redins. [Consulta en Agosto, 2020].

[18] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[19] DB-Engines. Popularity of key value stores. https://db-engines.com/en/ranking/key-value+store. [Consulta en Abril, 2020].

[20] DB-Engines. Wide column stores. https://db-engines.com/en/article/Wide+Column+Stores. [Consulta en Abril, 2020].

[21] Bind DLZ. Home. http://bind-dlz.sourceforge.net/.

[22] Power DNS. Backends. https://doc.powerdns.com/authoritative/backends/index.html, December 2019. [Consulta en Diciembre, 2019].

[23] Pandian Raju et al. Pebblesdb: Building key-value stores usingfragmented log-structured merge trees. https://www.cs.utexas.edu/~vijay/papers/sosp17-pebblesdb.pdf, May 2018.

[24] Golang. Frequently asked questions. https://golang.org/doc/faq. [Consulta en Agosto, 2020].

[25] Isaac Gouy. The computer language benchmarks game. https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/go-gcc.html, Junio 2020. [Consulta en Agosto, 2020].

[26] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15:287–317, 1983.

[27] ICANN. Dnssec - what is it and why is it important? https://www.icann.org/resources/pages/dnssec-what-is-it-why-important-2019-03-05-en. [Consulta en Agosto, 2020].

[28] IETF. Domain names - concepts and facilities, December 1987.

[29] IETF. Dynamic updates in the domain name system (dns update), April 1997.

[30] IETF. Ietf rfc 4033 - dns security introduction and requirements, December 2005.

[31] IETF. Ietf rfc 4034 - resource records for the dns security extensions, December 2005.

[32] IETF. Ietf rfc 4035 - protocol modifications for the dns security extensions, December 2005.

[33] IETF. Domain name system operations (dnsop). `https://datatracker.ietf.org/wg/dnsop/documents/`, Dec 2019. [Consulta en Diciembre, 2019].

[34] IETF. Domain names - implementation and specification. `https://www.ietf.org/rfc/rfc1035.txt`, Sept 2019. [Consulta en Octubre, 2019].

[35] Tim Keary. Pcap guide: Packet capture. `https://www.comparitech.com/net-admin/pcap-guide/`. [Consulta en Julio, 2020].

[36] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[37] A. B. M. Moniruzzaman and Syed Akhter Hossain. Nosql database: New era of databases for big data analytics - classification, characteristics and comparison. *CoRR*, abs/1307.0191, 2013.

[38] Don Moore. Dns server survey. `http://mydns.bboy.net./survey/`, Dec 2004. [Consulta en Diciembre, 2019].

[39] John Morris. Data structures and algorithms: Red black trees. `https://www.cs.auckland.ac.nz/software/AlgAnim/red_black.html`, Junio 1998. [Consulta en Agosto, 2020].

[40] CZ NIC. Benchmark de servidores dns autoritativos. `https://www.knot-dns.cz/benchmark/`, Sept 2019. [Consulta en Septiembre, 2019].

[41] CZ NIC. Knot dns. `https://www.knot-dns.cz/`, Sept 2019. [Consulta en Septiembre, 2019].

[42] NLnet. Página principal de nlnet labs. `https://nlnetlabs.nl/`, Sept 2019. [Consulta en Septiembre, 2019].

[43] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.

[44] PowerDNS. What is a powerdns backend? and how do i make it send an nxdomain? `https://blog.powerdns.com/2015/06/23/what-is-a-powerdns-backend-and-how-do-i-make-it-send-an-nxdomain/`, De-

cember 2015. [Consulta en Diciembre, 2019].

[45] Redis. Introducción a redis. https://redis.io/topics/introduction, Sept 2019. [Consulta en Diciembre, 2019].

[46] Tcpreplay. Tcpreplay overview. http://tcpreplay.appneta.com/wiki/overview.html. [Consulta en Agosto, 2020].

[47] Upguard. Ansible vs puppet. https://www.upguard.com/blog/ansible-puppet. [Consulta en Agosto, 2020].

[48] Wireshark. About. https://www.wireshark.org/. [Consulta en Agosto, 2020].

Anexo

En este anexo se muestran gráficos adicionales, complementarios al capítulo 5. Se pueden revisar tendencias de casos omitidos en el trabajo principal.

A . Anexo A Resultados adicionales para 100.000

A .1. Uso de memoria

Datos sintéticos

Ver evolución de Bind con datos sintéticos en figura (5.18).

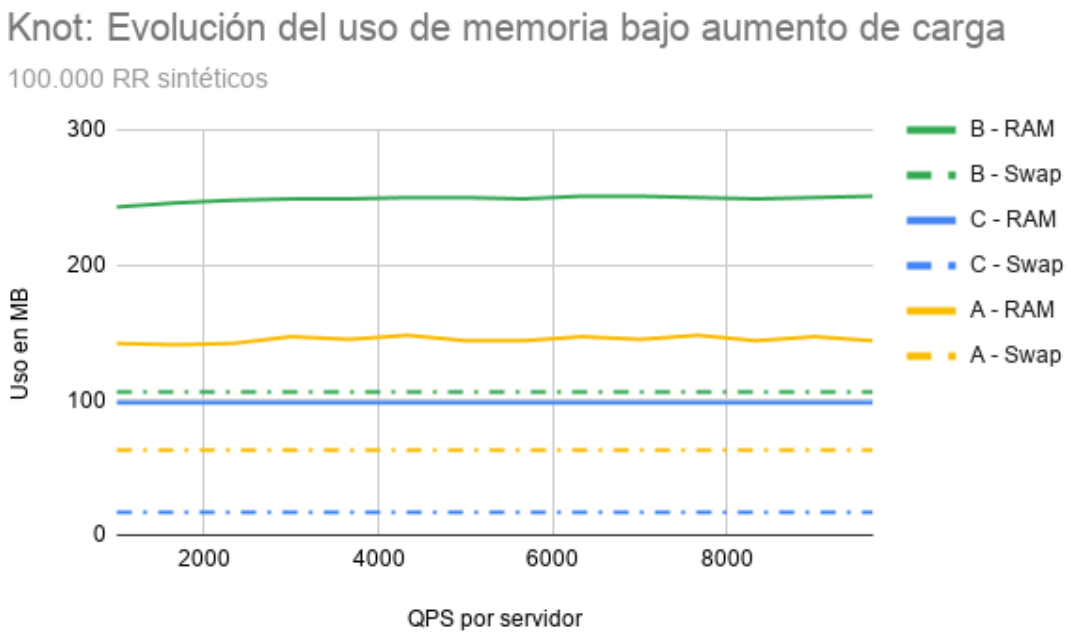

Figura 7.1: Evolución del uso de memoria para Knot y 100.000 RR sintéticos en hosting.

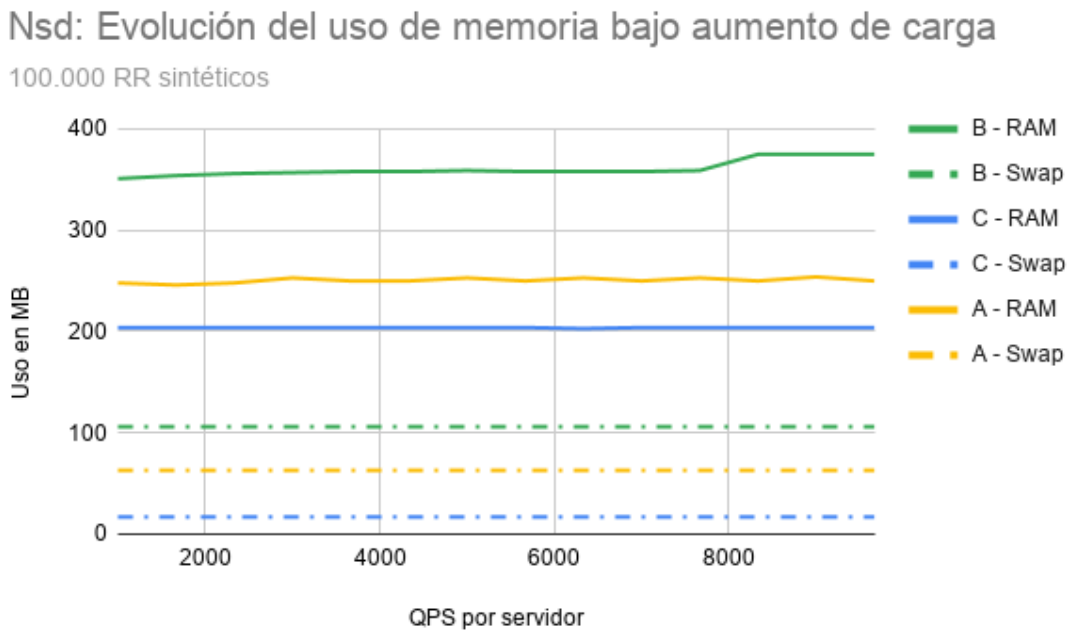

Figura 7.2: Evolución del uso de memoria para Nsd y 100.000 RR sintéticos en hosting.

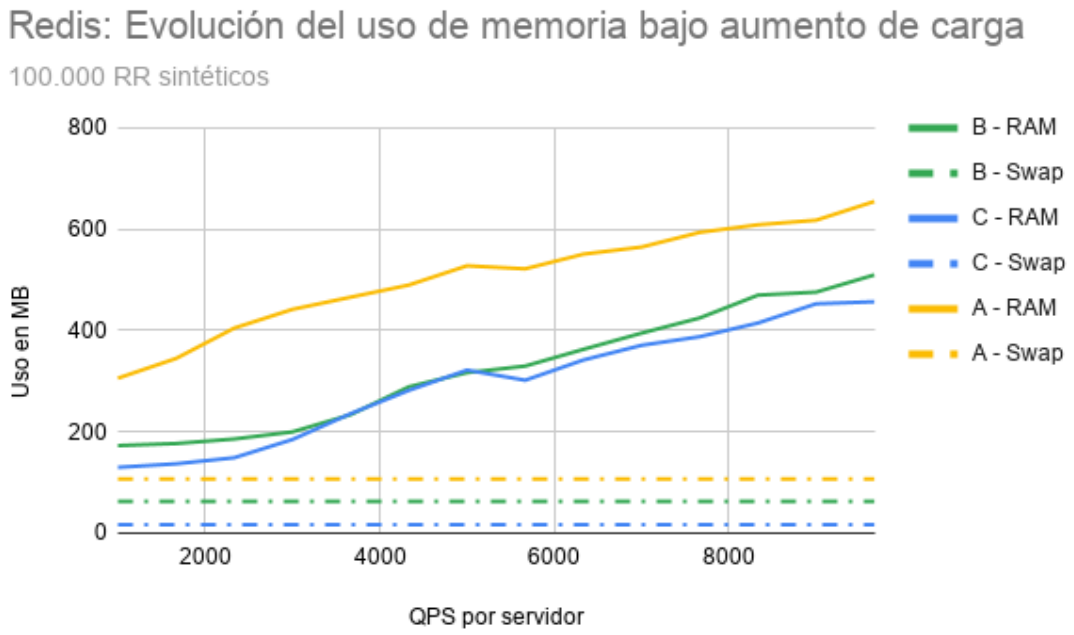

Figura 7.3: Evolución del uso de memoria para Redis y 100.000 RR sintéticos en hosting.

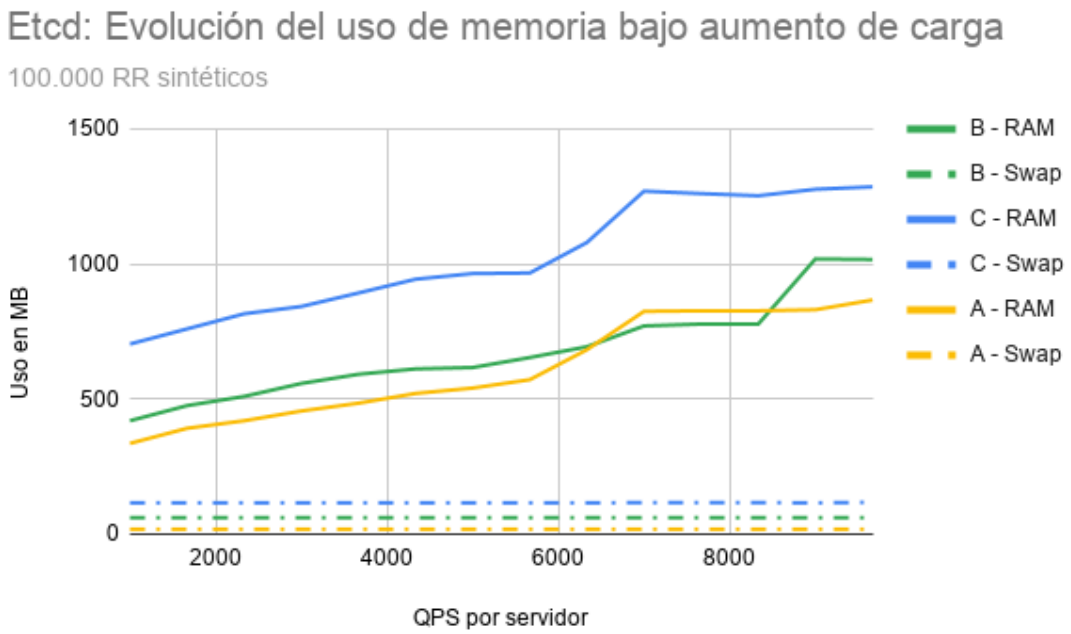

Figura 7.4: Evolución del uso de memoria para Etcd y 100.000 RR sintéticos en hosting.

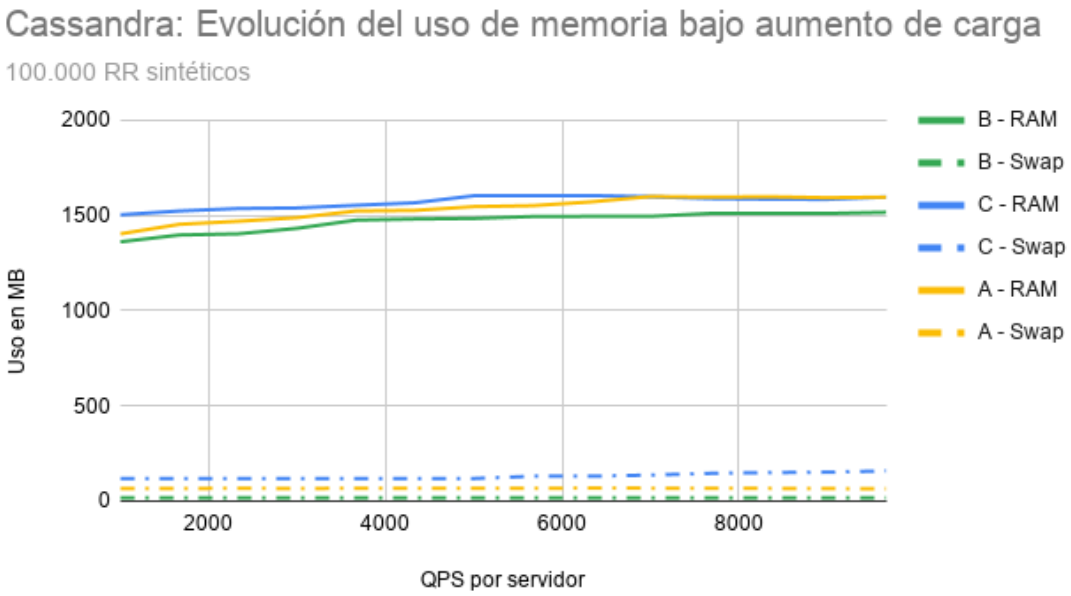

Figura 7.5: Evolución del uso de memoria para Cassandra y 100.000 RR sintéticos en hosting.

Datos reales

Ver evolución de Bind con datos reales en figura (5.19).

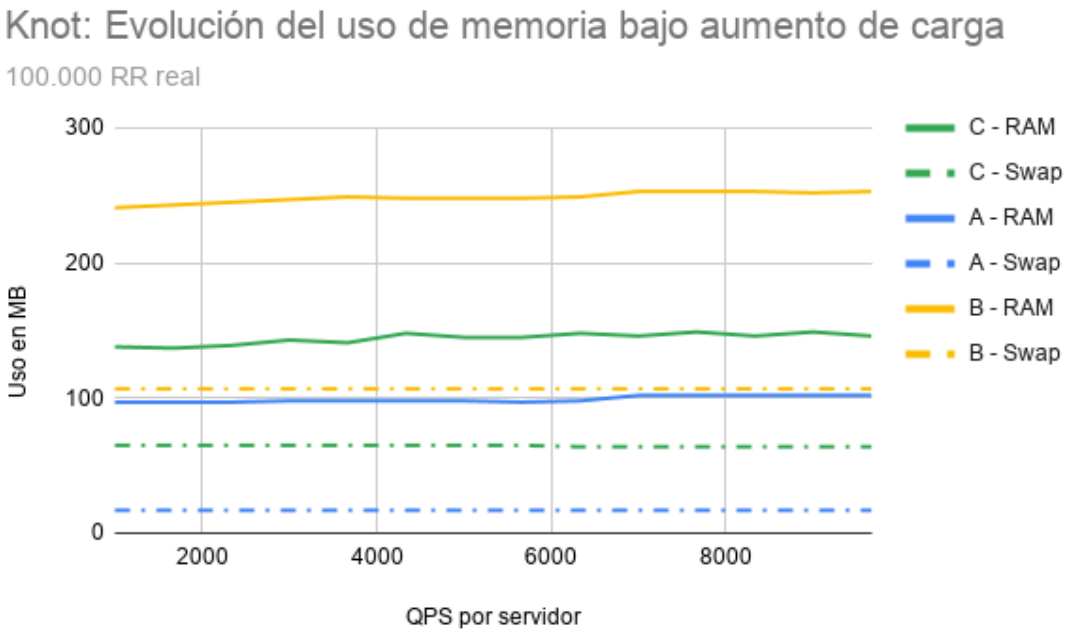

Figura 7.6: Evolución del uso de memoria para Knot y 100.000 RR reales en hosting.

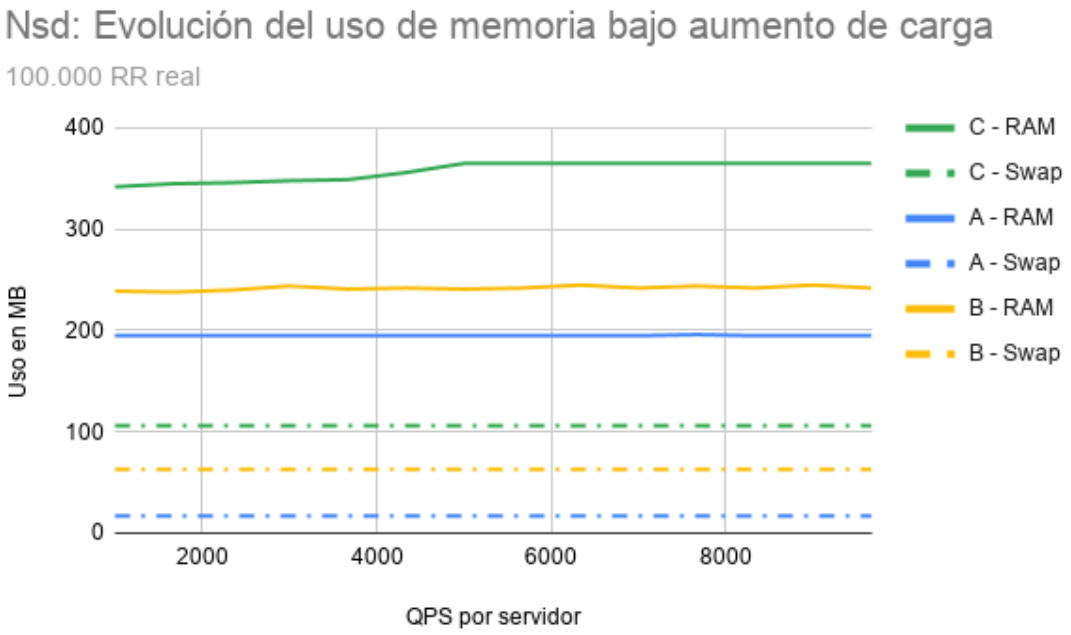

Figura 7.7: Evolución del uso de memoria para Nsd y 100.000 RR reales en hosting.

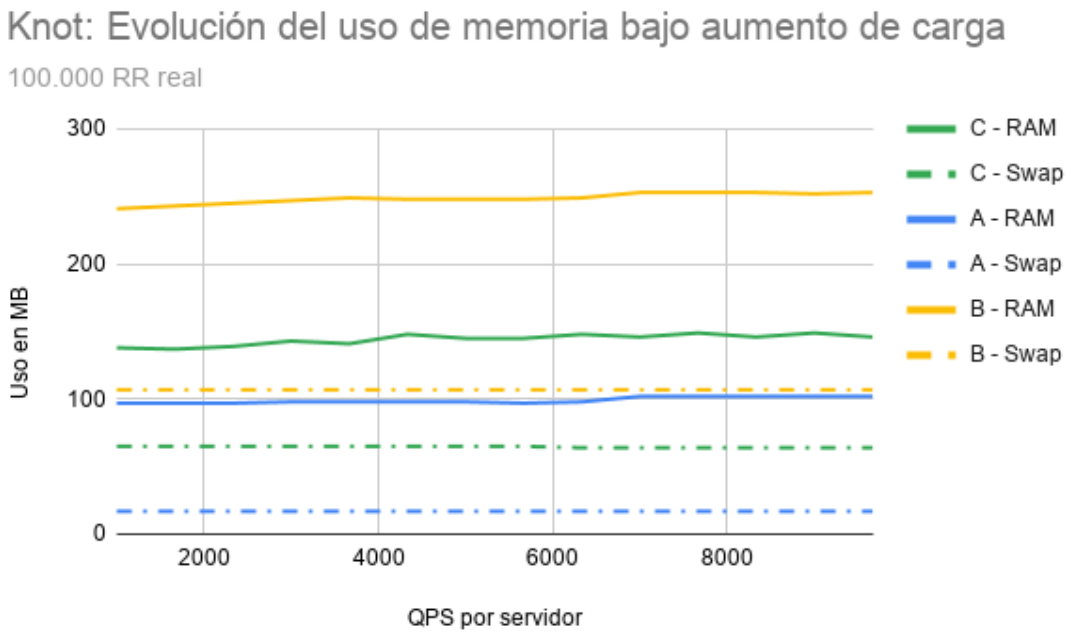

Figura 7.8: Evolución del uso de memoria para Knot y 100.000 RR reales en hosting.

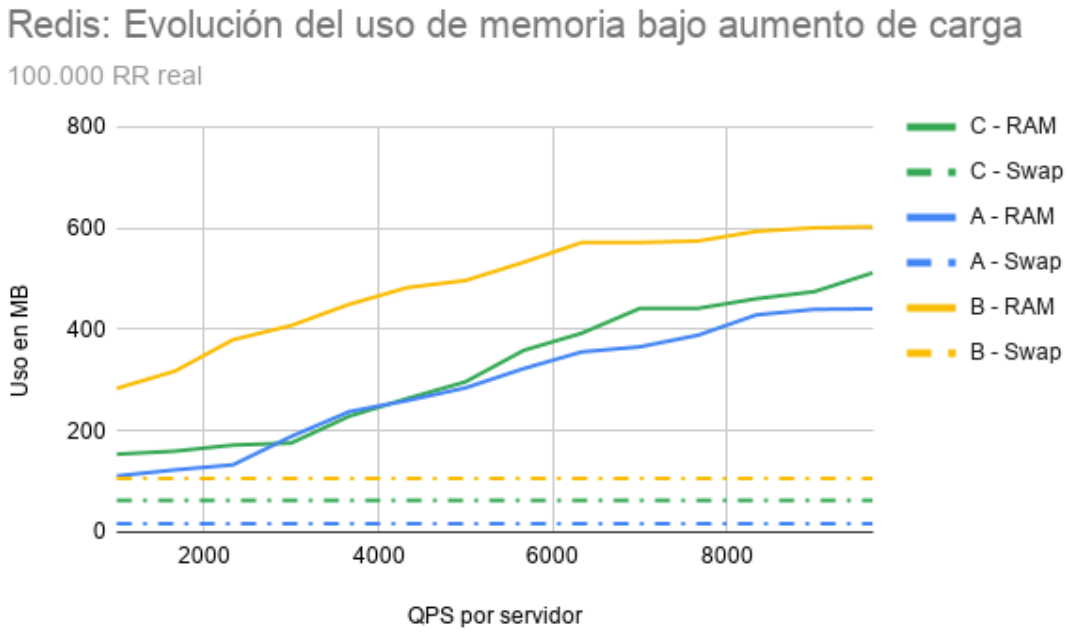

Figura 7.9: Evolución del uso de memoria para Redis y 100.000 RR reales en hosting.

Ver evolución de Cassandra con datos reales en figura (5.20).

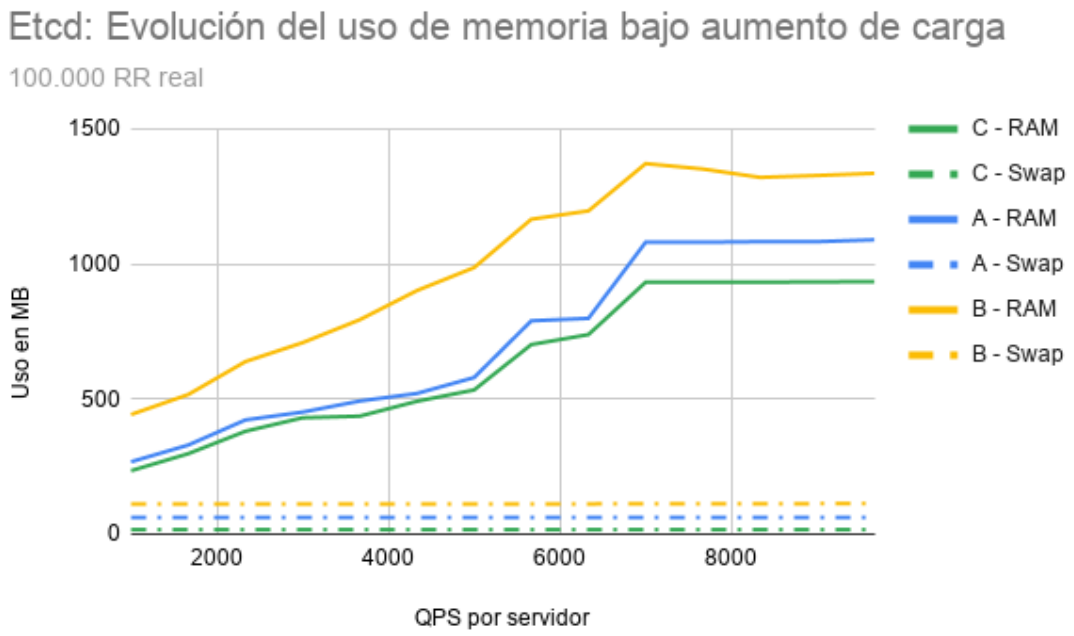

Figura 7.10: Evolución del uso de memoria para Etcd y 100.000 RR reales en hosting.

A .2. Uso de CPU

Carga de CPU medida con *proc/loadavg* para datos sintéticos

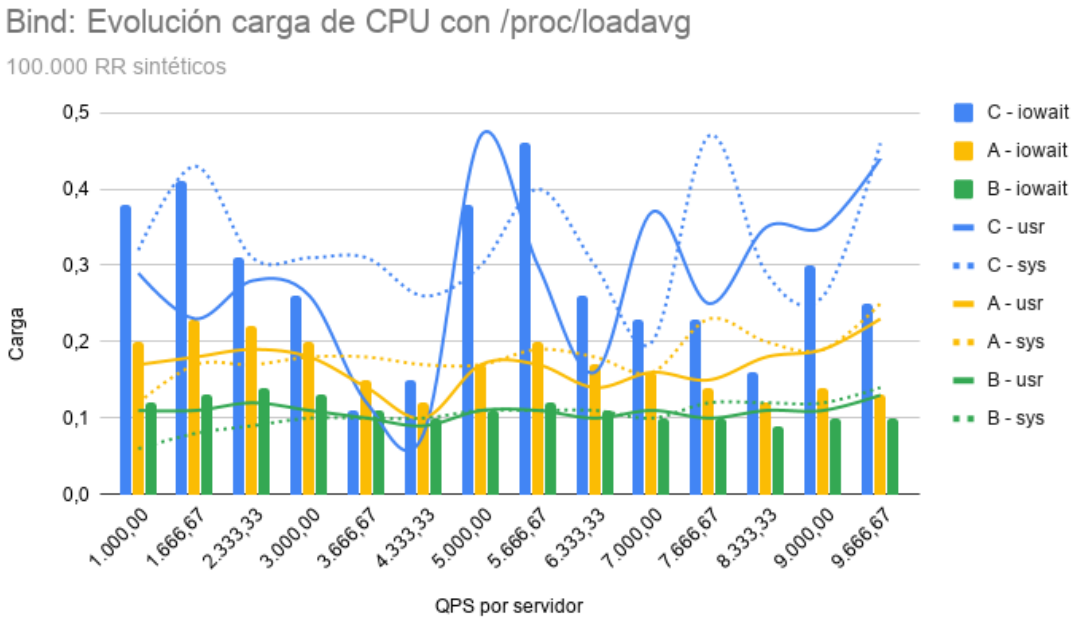

Figura 7.11: Tendencia de la carga de CPU para Bind entre sus 3 máquinas, con 100.000 RR sintéticos en hosting.

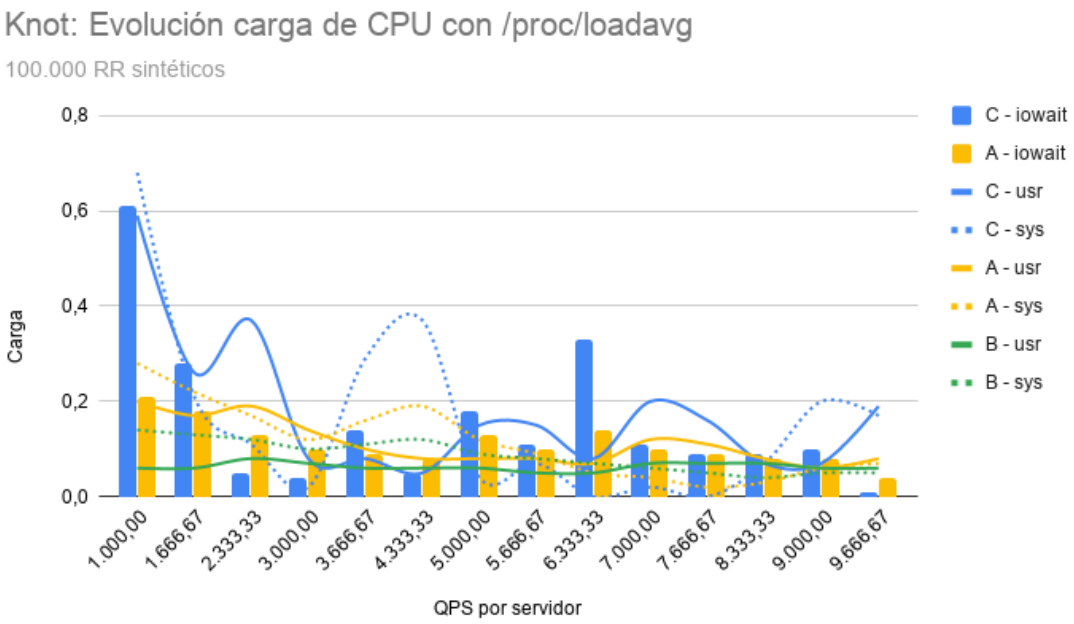

Figura 7.12: Tendencia de la carga de CPU para Knot entre sus 3 máquinas, con 100.000 RR sintéticos en hosting.

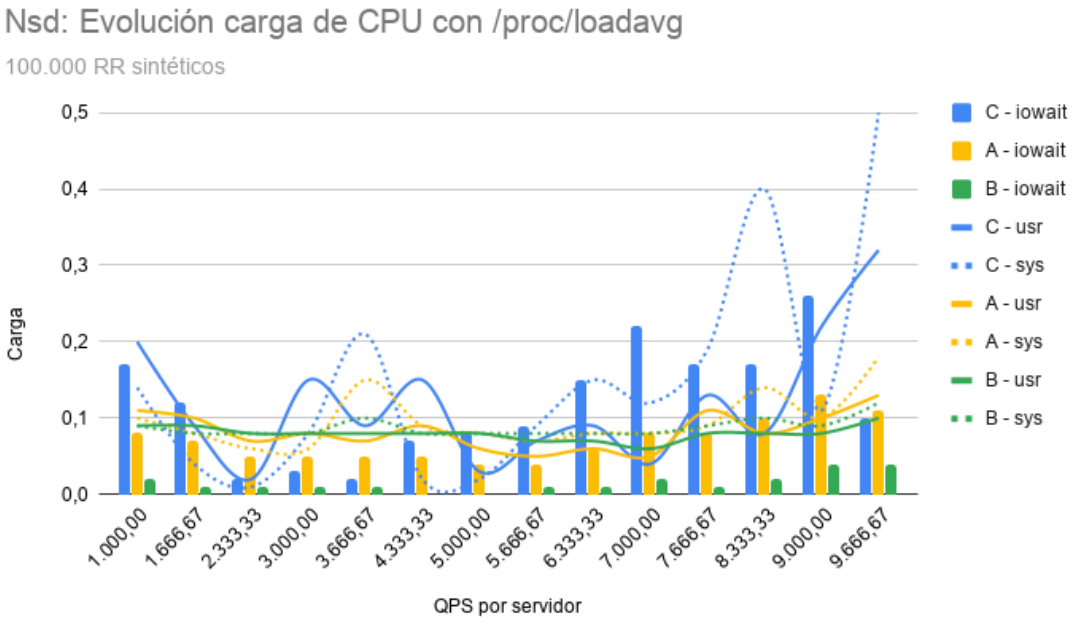

Figura 7.13: Tendencia de la carga de CPU para Nsd entre sus 3 máquinas, con 100.000 RR sintéticos en hosting.

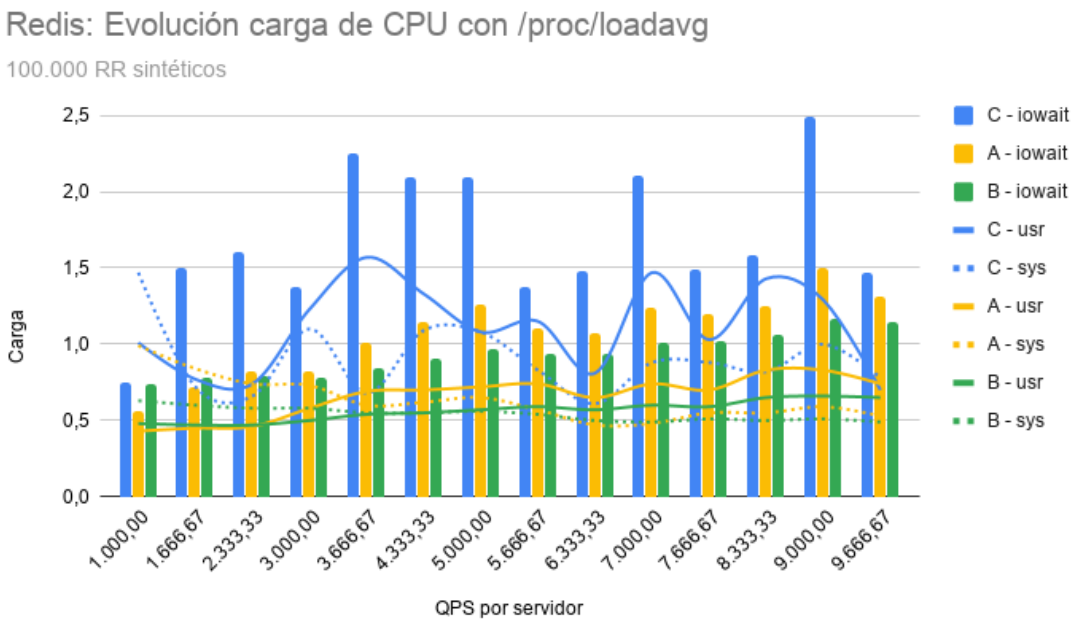

Figura 7.14: Tendencia de la carga de CPU para Redis entre sus 3 máquinas, con 100.000 RR sintéticos en hosting.

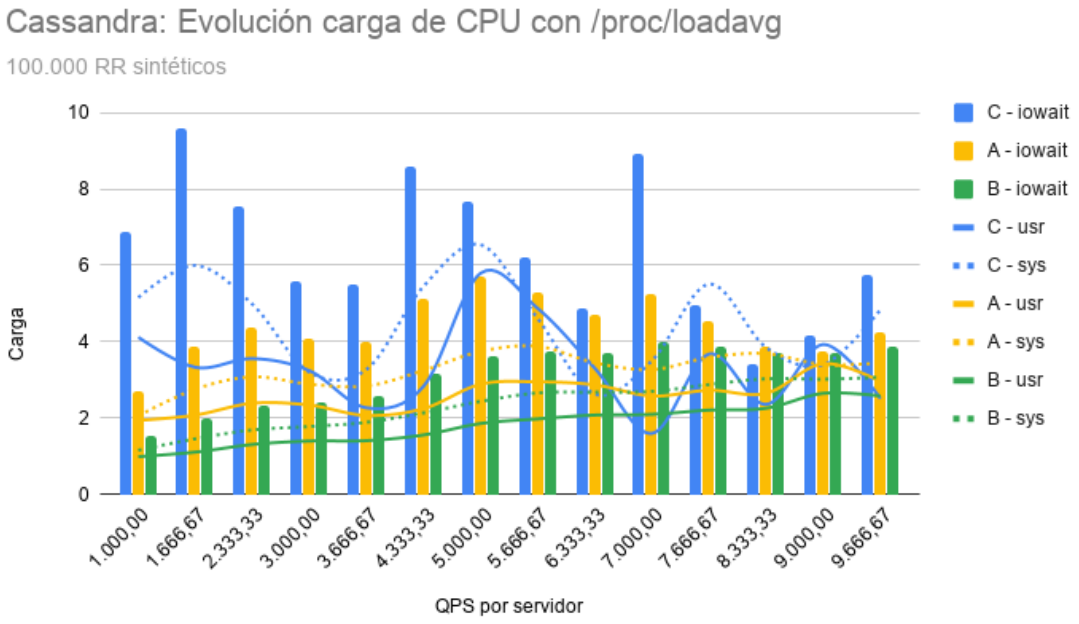

Figura 7.15: Tendencia de la carga de CPU para Cassandra entre sus 3 máquinas, con 100.000 RR sintéticos en hosting.

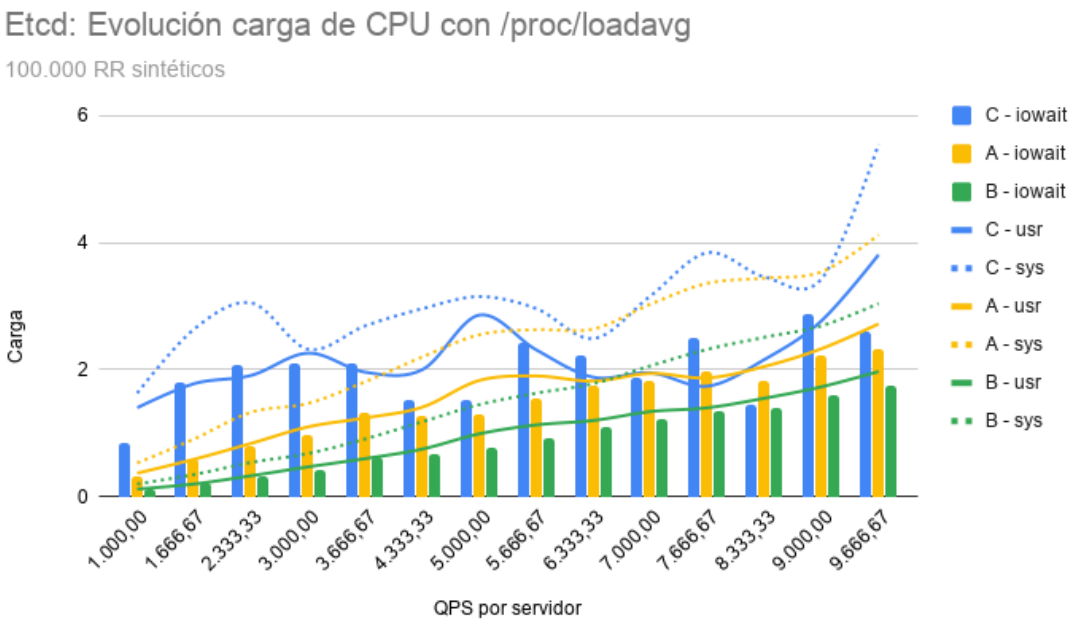

Figura 7.16: Tendencia de la carga de CPU para Etcd entre sus 3 máquinas, con 100.000 RR sintéticos en hosting.

Carga de CPU medida con */proc/loadavg* para datos reales

Ver evolución de Bind con datos reales en figura (5.24).

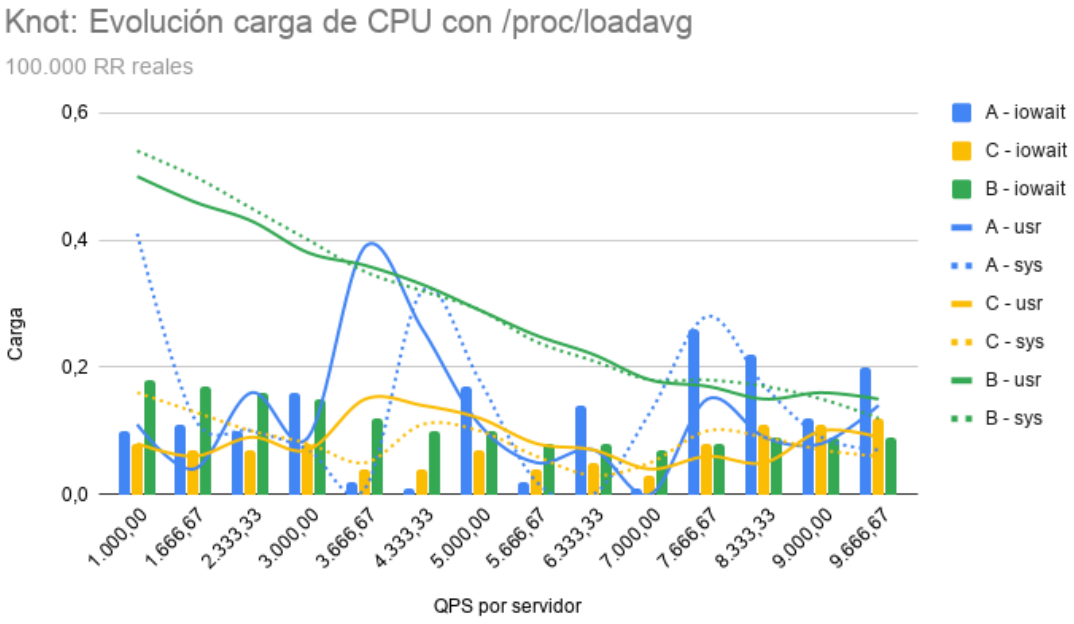

Figura 7.17: Tendencia de la carga de CPU para Knot entre sus 3 máquinas, con 100.000 RR reales en hosting.

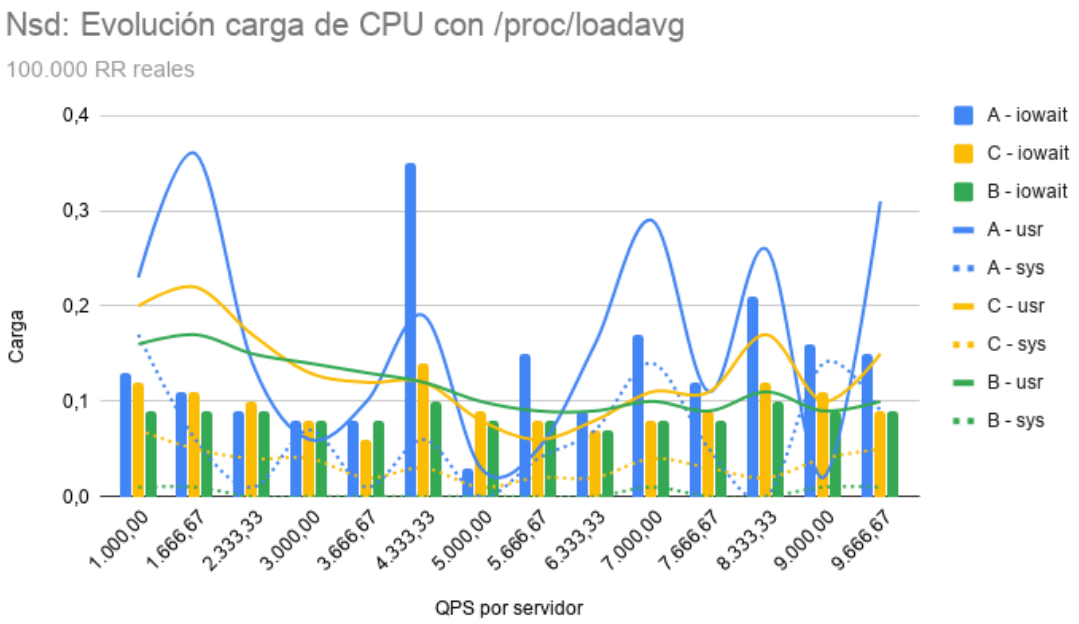

Figura 7.18: Tendencia de la carga de CPU para Nsd entre sus 3 máquinas, con 100.000 RR reales en hosting.

Ver evolución de Redis con datos reales en figura (5.25).

Ver evolución de Cassandra con datos reales en figura (5.26).

Ver evolución de Etcd con datos reales en figura (5.27).

B . Anexo B Resultados para 2.000.000

B .1. Uso de memoria

Datos sintéticos

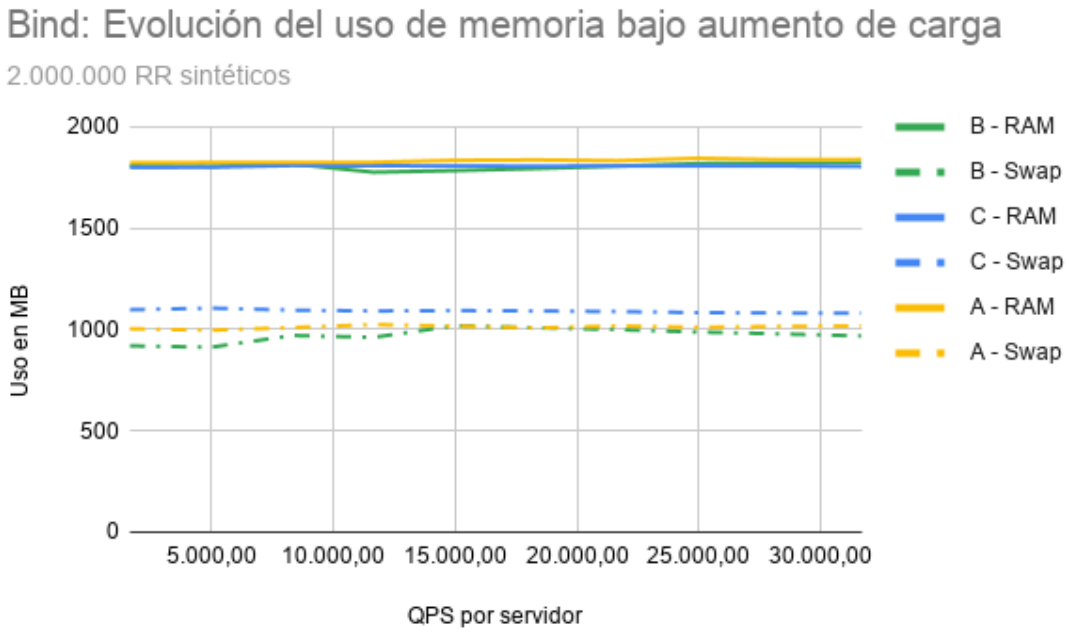

Figura 7.19: Evolución del uso de memoria para Bind y 2.000.000 RR sintéticos en hosting.

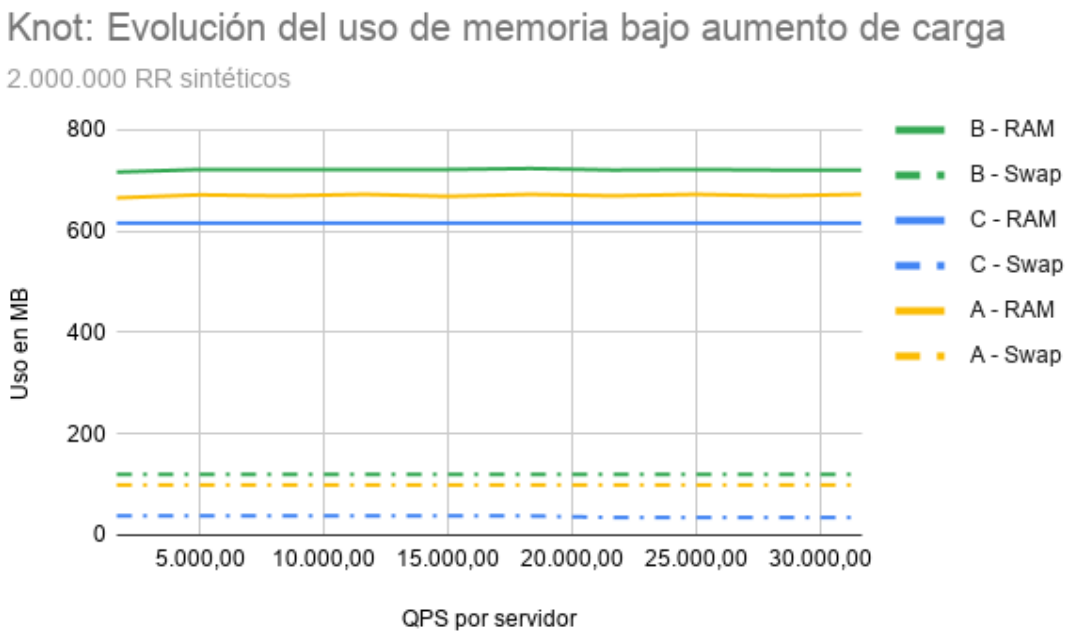

Figura 7.20: Evolución del uso de memoria para Knot y 2.000.000 RR sintéticos en hosting.

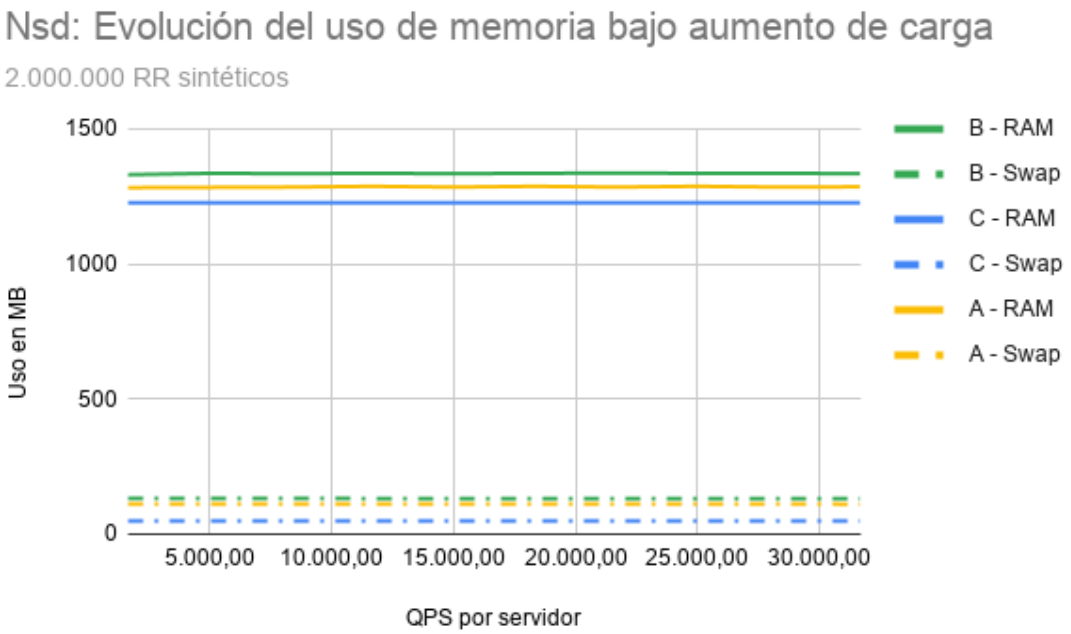

Figura 7.21: Evolución del uso de memoria para Nsd y 2.000.000 RR sintéticos en hosting.

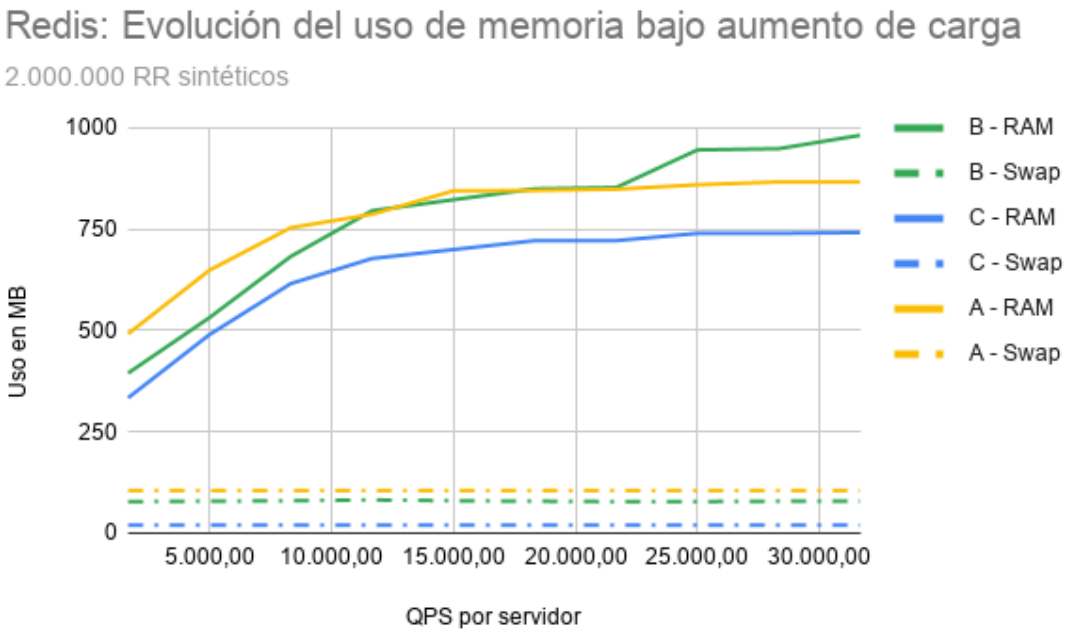

Figura 7.22: Evolución del uso de memoria para Redis y 2.000.000 RR sintéticos en hosting.

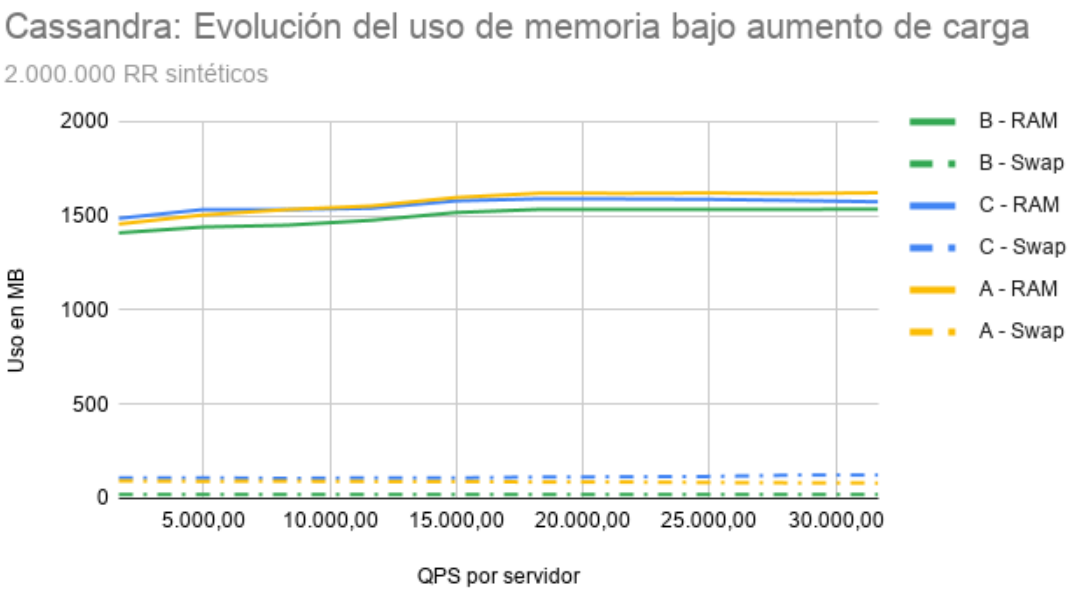

Figura 7.23: Evolución del uso de memoria para Cassandra y 2.000.000 RR sintéticos en hosting.

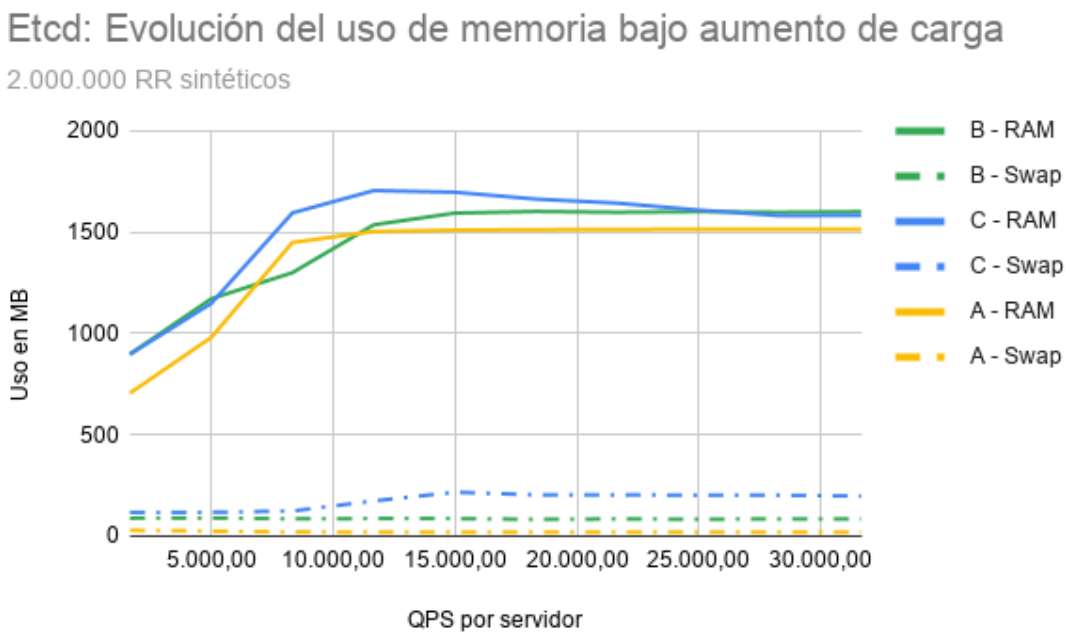

Figura 7.24: Evolución del uso de memoria para Etcd y 2.000.000 RR sintéticos en hosting.

Datos reales

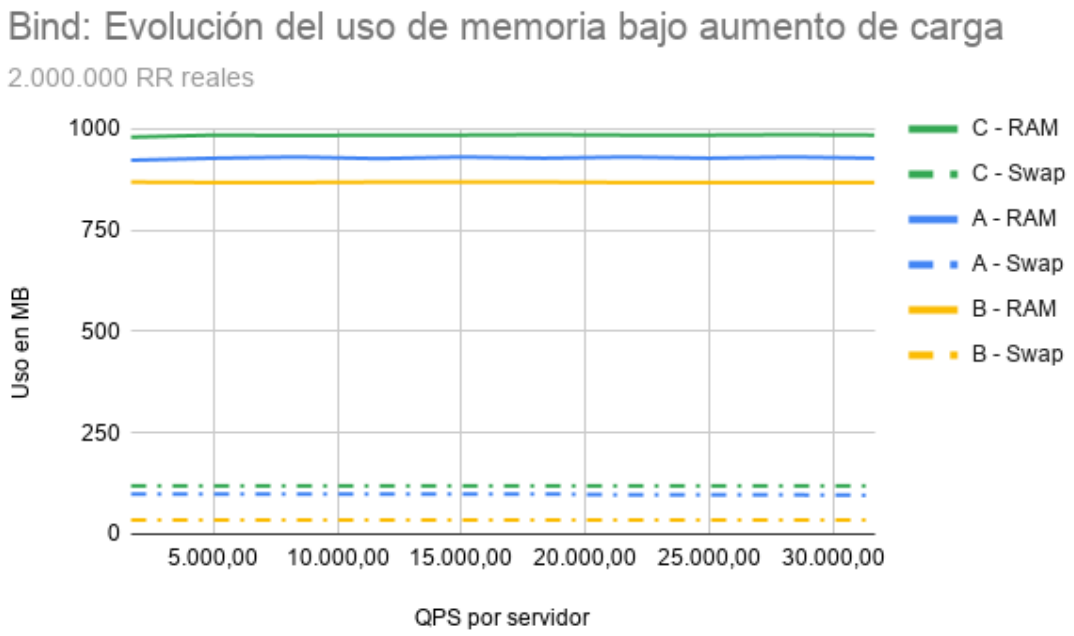

Figura 7.25: Evolución del uso de memoria para Bind y 2.000.000 RR reales en hosting.

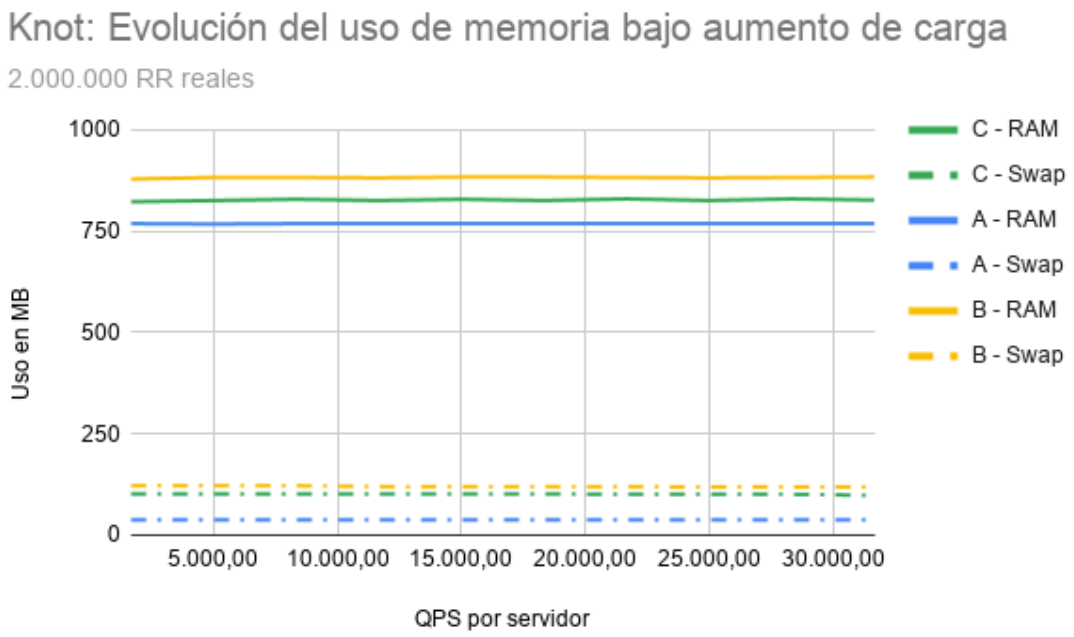

Figura 7.26: Evolución del uso de memoria para Knot y 2.000.000 RR reales en hosting.

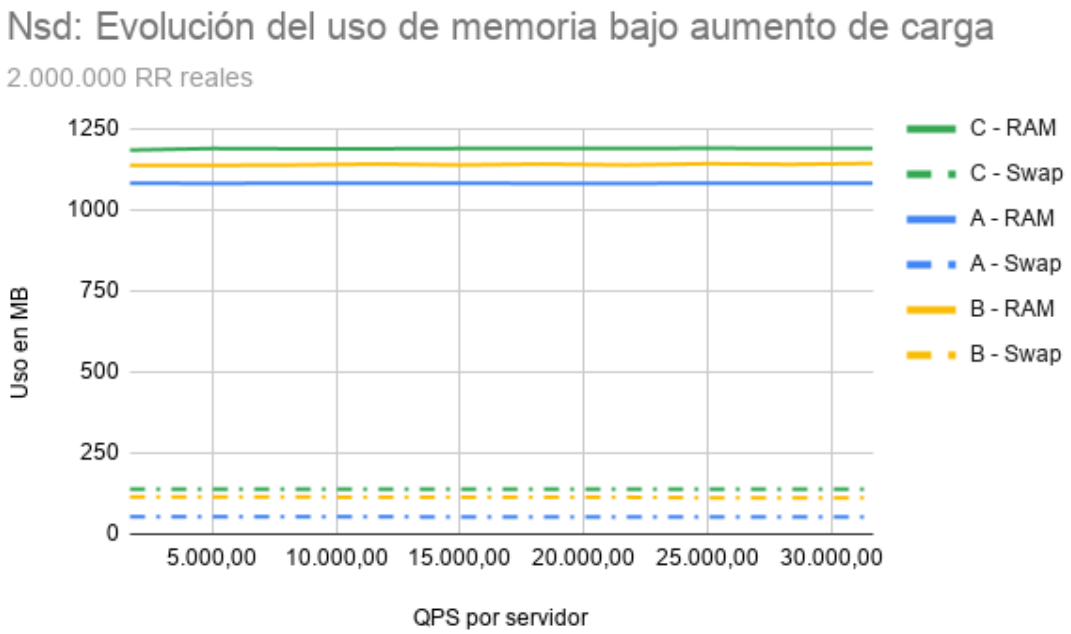

Figura 7.27: Evolución del uso de memoria para Nsd y 2.000.000 RR reales en hosting.

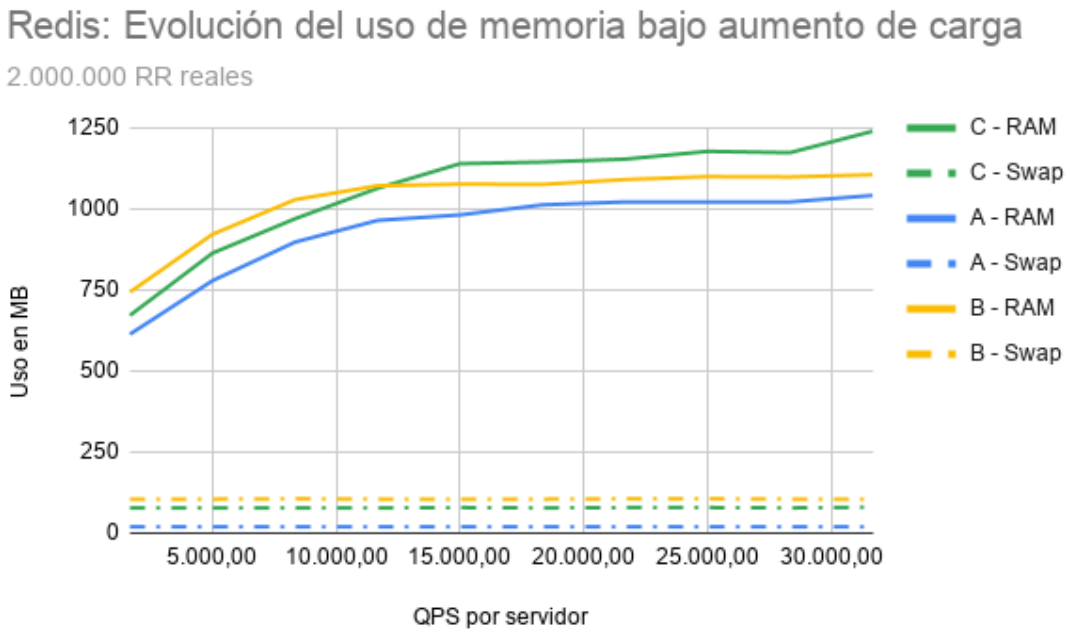

Figura 7.28: Evolución del uso de memoria para Redis y 2.000.000 RR reales en hosting.

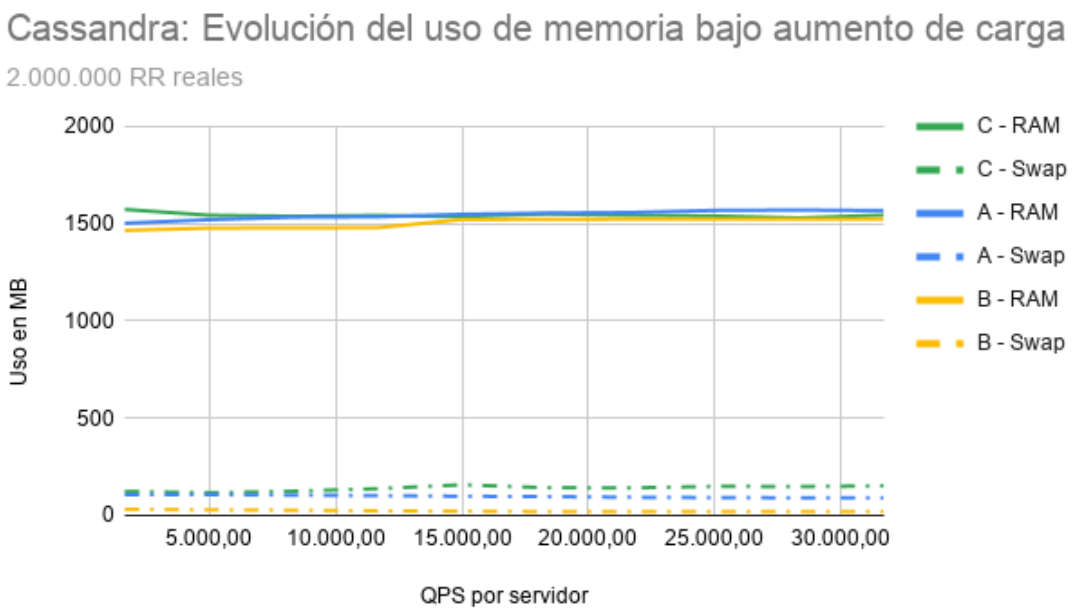

Figura 7.29: Evolución del uso de memoria para Cassandra y 2.000.000 RR reales en hosting.

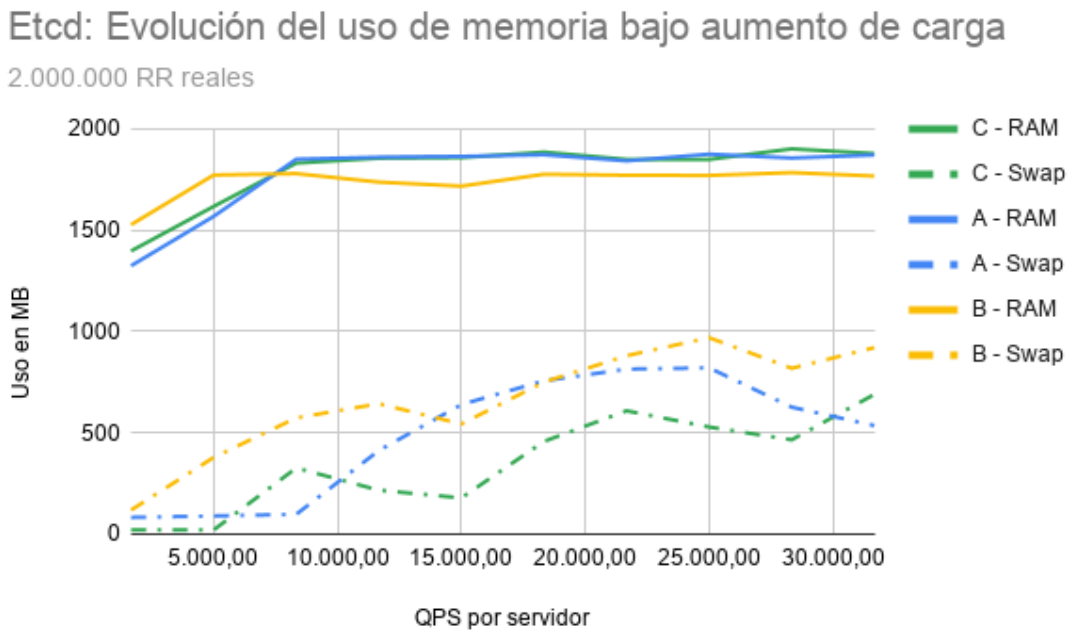

Figura 7.30: Evolución del uso de memoria para Etcd y 2.000.000 RR reales en hosting.

B .2. Cantidad de respuestas con datos reales

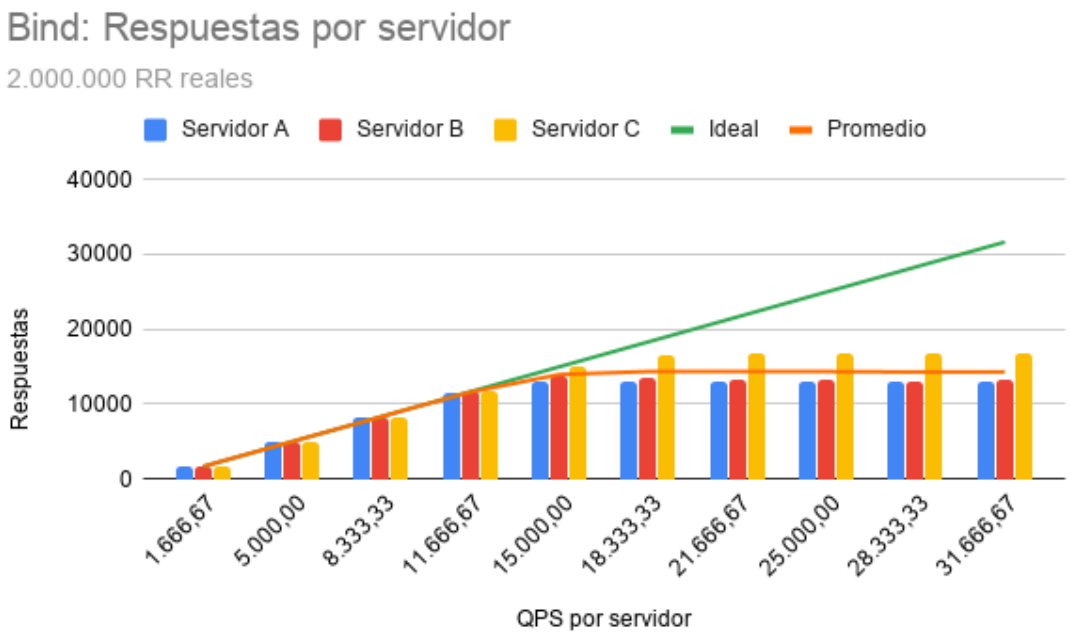

Figura 7.31: Respuestas por servidor para Bind, con 2.000.000 RR reales.

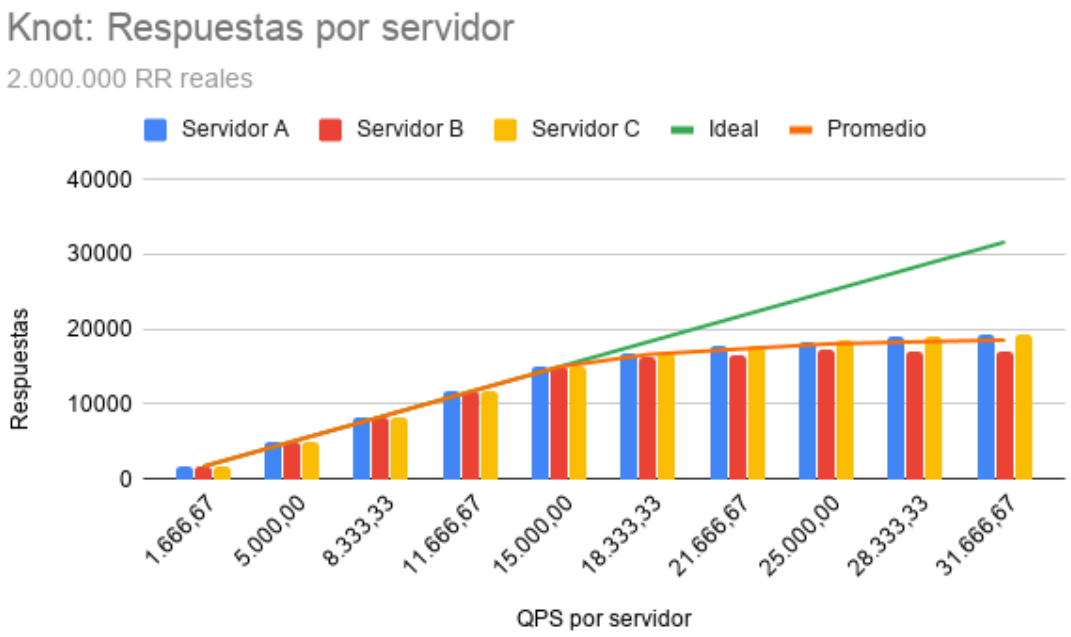

Figura 7.32: Respuestas por servidor para Knot, con 2.000.000 RR reales.

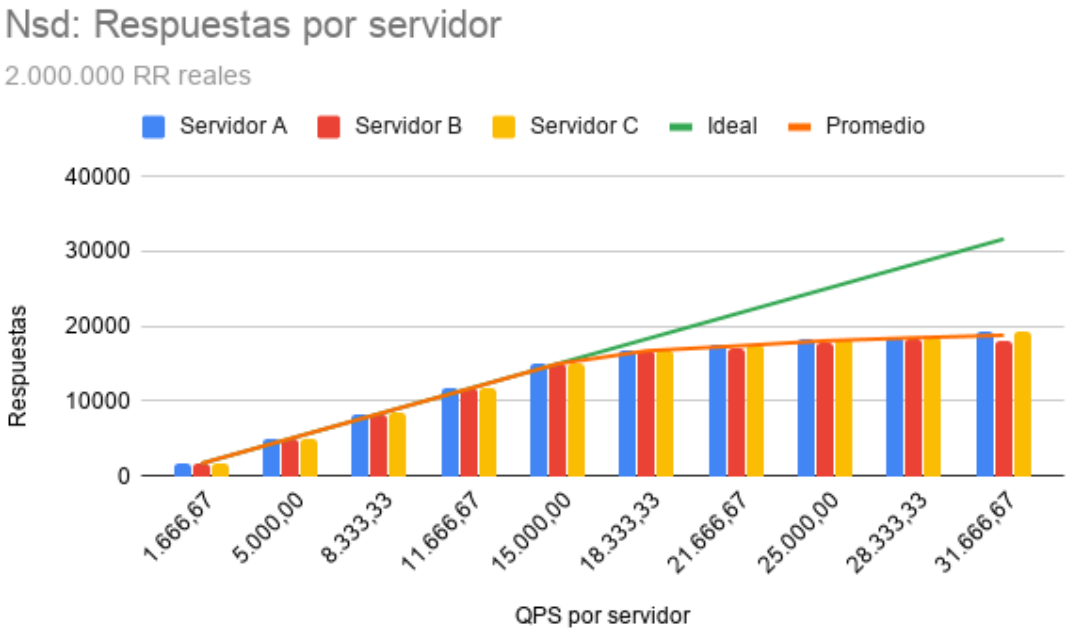

Figura 7.33: Respuestas por servidor para Nsd, con 2.000.000 RR reales.

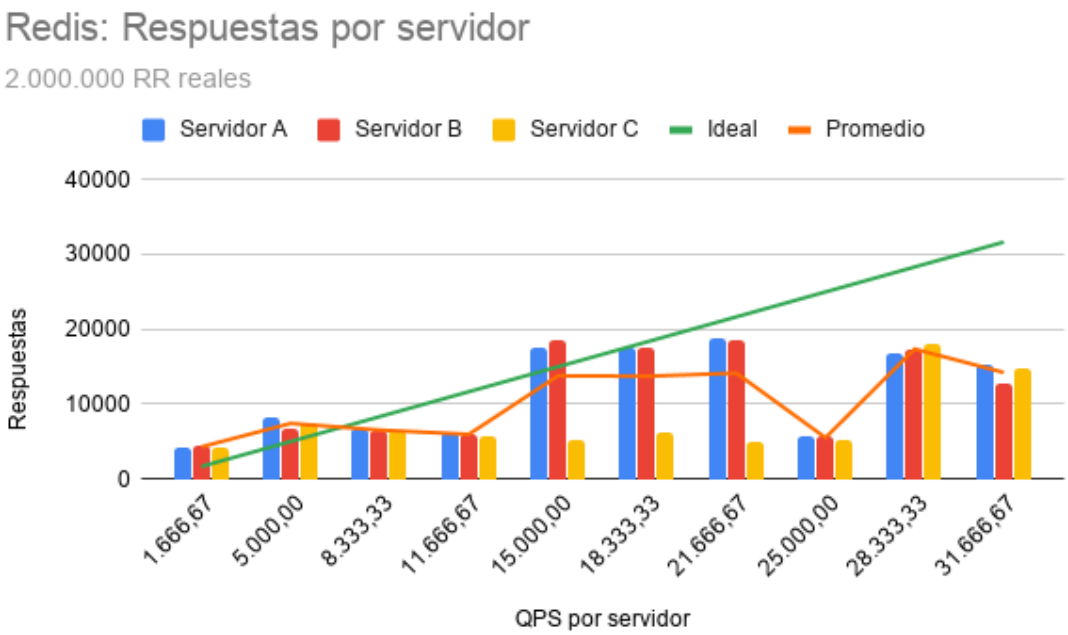

Figura 7.34: Respuestas por servidor para Nsd, con 2.000.000 RR reales.

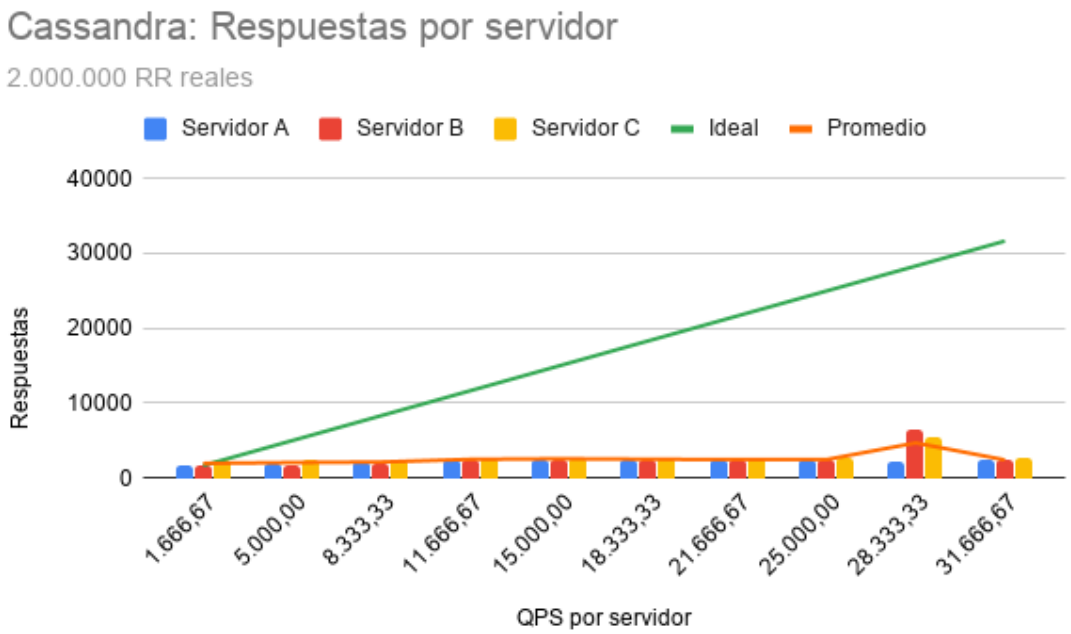

Figura 7.35: Respuestas por servidor para Cassandra, con 2.000.000 RR reales.

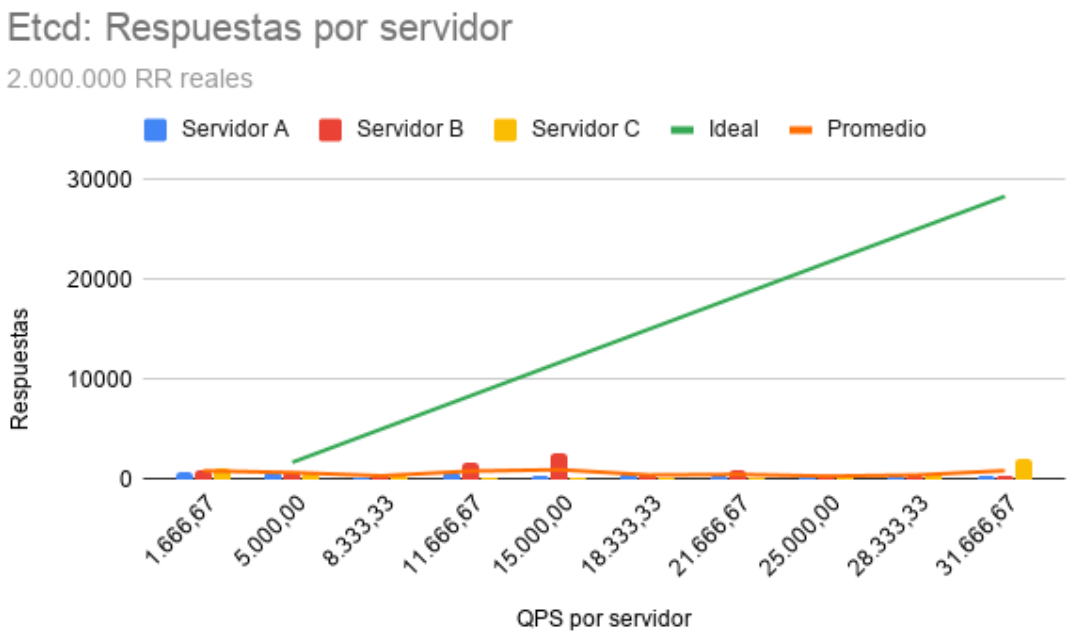

Figura 7.36: Respuestas por servidor para Etcd, con 2.000.000 RR reales.