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

REINGENIERÍA BACK-END SOYMOMO

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

MANUEL ALEJANDRO RAMÍREZ GUTIÉRREZ

PROFESOR GUÍA:
MARÍA CECILIA BASTARRICA PIÑEYRO

MIEMBROS DE LA COMISIÓN:
NELSON BALOIAN TATARYEN
ANDRÉS FARÍAS RIQUELME

SANTIAGO DE CHILE
2019

RESUMEN

SoyMomo es un reloj-celular-GPS para niños que permite a los padres -entre otras funciones- saber dónde se encuentran y poder hacer llamadas desde y hacia el reloj, a través de aplicaciones móviles. Estos relojes tienen un módulo GSM (2G) y se comunican a un servidor a través de sockets TCP.

El sistema se compone de un servidor desarrollado en Node.js y un sistema de Backend as a Service que almacena la base de datos. Debido al sostenido aumento de usuarios de éste, el servidor ha alcanzado el límite de capacidad, utilizando todos los recursos que estaba diseñado para usar. Este límite sin embargo, no es el límite del sistema, por lo tanto existían oportunidades de mejora.

Además, también se alcanzaba el límite de consultas por segundo en la base de datos, lo que provocaba la pérdida de información debido a que el servidor rechazaba las consultas sobre ese límite.

Se realizó una reingeniería del servidor, identificando los puntos más críticos de falla: diseño, rendimiento y optimización de consultas; y se reescribió el *software* utilizando el lenguaje Go, aplicando conceptos de paralelismo y concurrencia, con el fin de mejorar el rendimiento y crear un sistema escalable.

Para esto, se reestructuró el programa en secciones pequeñas de código capaces de ejecutarse de forma concurrente, con el fin de utilizar las bondades del *scheduler* lenguaje, que es capaz de repartir el trabajo entre todos los *cores* disponibles del procesador.

Además, se creó una sección central que se encarga de agrupar consultas a la base de datos y enviarlas en lotes, permitiendo atender a la misma cantidad de clientes con menos recursos.

Finalmente se pudo concluir que las mejoras planificadas cumplen con lo propuesto, realizando una utilización óptima de los recursos del sistema, al lograr atender un 19 % más de clientes con lel mismo número de consultas; eliminando la pérdida de información y rebajando el uso de CPU a menos del 20 %.

There is no dark side of the moon really,
matter of fact it's all dark.
(Pink Floyd - Eclipse)

Tabla de Contenido

Apéndices **A**

1. Introducción **1**
1.1. Contexto . . . 1
1.2. Motivación . . . 2
1.2.1. Infraestructura Original . . . 2
1.2.2. Problemas de la Infraestructura Original . . . 2
1.2.3. Expectativas de Crecimiento . . . 3
1.3. Objetivos . . . 4
1.4. Descripción del Trabajo Realizado . . . 4
1.5. Estructura del documento . . . 4

2. Marco Teórico **6**
2.1. Reingeniería de Software . . . 6
2.2. Escalabilidad . . . 6
2.3. Concurrencia y Paralelismo . . . 7
2.4. Comunicación entre procesos y sincronización . . . 8
2.5. Go . . . 9

3. Análisis de la infraestructura original **10**
3.1. Protocolo de comunicación . . . 10
3.2. Arquitectura del sistema . . . 11
3.2.1. Sistema BaaS (Parse Server) . . . 11
3.2.2. Servidor TCP . . . 11
3.2.3. Servidor HTTPS . . . 11
3.3. Evaluación de la infraestructura . . . 12
3.3.1. Porcentaje de uso de CPU . . . 12
3.3.2. Porcentaje de pérdida de información . . . 13
3.3.3. Tiempo de respuesta servidor TCP . . . 13
3.3.4. Tiempo de escritura y lectura en la DB . . . 14
3.3.5. Cantidad de peticiones a la DB por dispositivo . . . 15
3.4. Análisis . . . 16

4. Multithreading **17**
4.1. Multithreading en Node.js . . . 17
4.1.1. Primeros pasos . . . 17
4.1.2. Solución con child process y fork . . . 18

4.2. Multithreading en Go 19
4.2.1. Diseño del Software 20
4.2.2. Main y servidor TCP 20
4.2.3. Servidor HTTPS 21
4.2.4. *Fan-out* de comandos 22

5. Optimización de consultas a la base de datos 23
5.1. Análisis y categorización de las consultas 23
5.1.1. Comandos no optimizables 24
5.1.2. LK: Actualización del porcentaje de batería y conteo de pasos 24
5.1.3. TKQ: Solicitud de comandos pendientes 24
5.1.4. Otros comandos: confirmación de recepción 26
5.2. Refactorización por lotes 27

6. Resultados 29
6.1. Porcentaje de uso de CPU 29
6.2. Porcentaje de pérdida de información 30
6.3. Tiempo de respuesta servidor TCP 31
6.4. Tiempo de escritura y lectura en la DB 32
6.5. Cantidad de peticiones a la DB por dispositivo 32

Conclusión 33

Bibliografía 36

Apéndices 38

A. Protocolo de Comunicación de SoyMomo 38

B. Datos Uso de CPU 39

C. Datos Pérdida de Información y Cantidad de Consultas 44

D. Datos tiempo respuesta, lectura y escritura 49

Índice de Tablas

1.1. Duración del trabajo realizado 5

3.1. Cantidad de peticiones a la DB por dispositivo 16

6.1. Cantidad de peticiones a la DB por dispositivo, después del *batching* 33

Índice de Ilustraciones

1.1. Imágen promocional de SoyMomo 1
1.2. Arquitectura del servicio SoyMomo 3

3.1. Uso de CPU 12
3.2. Pérdida de Información 13
3.3. Tiempo de respuesta 14
3.4. Tiempo de lectura en DB 14
3.5. Tiempo de escritura en DB 15
3.6. Consultas promedio por dispositivo 15

4.1. Implementación de multithreading en Node.js 19
4.2. Uso de CPU al utilizar multithreading en Node.js 19
4.3. Diseño del software en Go 20

5.1. Consultas producidas por el comando LK 25
5.2. Consultas producidas por el comando TKQ 26
5.3. Diseño del sistema de *batching* 27
5.4. Diagrama de flujo del sistema de *batching* 28

6.1. Uso de CPU, servidor desarrollado en Go 29
6.2. Prueba de carga servidor desarrollado en Go 30
6.3. Porcentaje de pérdida de información, después del *batching* 31
6.4. Tiempo de respuesta servidor desarrollado en Go 31
6.5. Tiempo de escritura en la DB, después del *batching* 32
6.6. Tiempo de lectura en la DB, después del *batching* 32
6.7. Consultas promedio por dispositivo, después del *batching* 33

Capítulo 1

Introducción

1.1. Contexto

SoyMomo es un reloj-celular-GPS para niños que permite a los padres -entre otras funciones- saber dónde se encuentran y poder hacer llamadas desde y hacia el reloj, a través de apps disponibles para Android e iOS. Estos relojes tienen un módulo GSM (2G) y se comunican a un servidor a través de sockets TCP.

El desarrollo de este proyecto se inició en Agosto de 2016 por Sosmart Labs, empresa chilena de tecnología. Fue puesto en producción el 15 de diciembre de 2016, con la venta de los primeros relojes. Si bien el software y el hardware de los relojes es desarrollado en China, el desarrollo del servidor y de las aplicaciones móviles se ha llevado a cabo íntegramente en Chile.

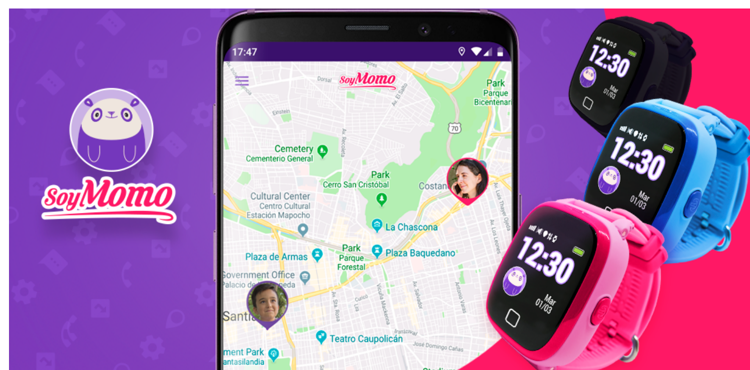

Figura 1.1: Imágen promocional de SoyMomo

1.2. Motivación

A medida que ha ido creciendo el proyecto, tanto en términos de las funcionalidades implementadas como en número de usuarios, se identificaron problemas en el back-end del proyecto. Con aproximadamente 3000 clientes activos, *peaks* de 1000 conexiones concurrentes (a la fecha 31/08/2017) y amplias proyecciones de crecimiento, la infraestructura tecnológica al momento de realizar este trabajo se tornaba insuficiente para el desarrollo de la empresa.

1.2.1. Infraestructura Original

La infraestructura tecnológica del back-end de la empresa consta de 2 partes: una plataforma de BaaS[1], manejada por un proveedor externo y un software que comprende un servidor HTTPS[2] y TCP.

- **BaaS:** Parse[3] (provisto por Back4app[4]) es una plataforma de código abierto desarrollada en Node.js. Incluye una API para manejar datos de forma relacional sobre MongoDB y además de servir como BD, permite correr código en la nube (Cloud Code), incluye un módulo de autenticación de usuarios y también contempla SDKs[5] para integrarlos en aplicaciones móviles para Android e iOS con métodos para realizar operaciones CRUD[6], Login/Registro de usuarios y llamadas a Cloud Code.
- **Servidor HTTPS/TCP:** Desarrollado desde cero en la empresa usando Javascript bajo el entorno Node.js. Recibe conexiones HTTPS y TCP a través de distintos puertos en el mismo proceso y actúa como adaptador entre la plataforma de BaaS (HTTPS) y los relojes (TCP). Está instalado en una máquina virtual en Microsoft Azure[7].

1.2.2. Problemas de la Infraestructura Original

Se ha identificado que el servidor (HTTPS+TCP) alcanzó su capacidad límite para servir las peticiones de los relojes. Debido a que Node.js funciona en un solo hilo de ejecución, se observa que se está ocupando el 100 % de uno de los cores de la CPU de la máquina virtual durante gran parte del día, mientras que los otros cores están sin usar. Lo anterior provoca que el servidor sea incapaz de procesar de forma óptima todas las solicitudes enviadas por los clientes.

Al alcanzar el límite en su capacidad de procesamiento, el servidor deja de aceptar conexiones de nuevos clientes, mientras que las solicitudes recibidas a través de las conexiones

[1] Back-end as a Service

[2] HTTP es un protocolo a nivel de la capa de aplicación, que usualmente usa TCP como protocolo a nivel de la capa de transporte.

[3] https://parseplatform.org

[4] https://www.back4app.com/

[5] Software Development Kit

[6] Create, Read, Update, Delete

[7] Especificaciones: 4 Cores, 14GB RAM, 8 Data disks, 12800 Max IOPS, 28 GB Local SSD

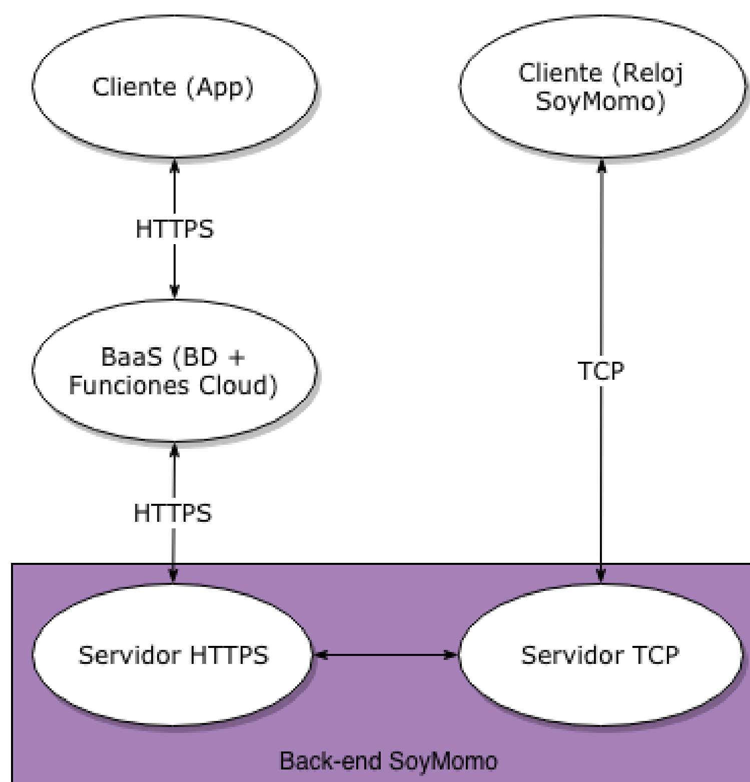

Figura 1.2: Arquitectura del servicio SoyMomo

existentes ven aumentados sus tiempos de respuesta.

Además, en los momentos de más tráfico, se alcanza el límite de operaciones por segundo permitidas por el sistema BaaS que maneja la base de datos. Estos *peaks* se alcanzan usualmente de lunes a viernes, alrededor de las 08:30 hrs. y a nuevamente en la tarde alrededor de las 17:00 hrs., horarios coincidentes con la entrada y salida de clases en la mayoría de los colegios.

Los problemas anteriores se traducen en un aumento significativo de los tiempos de respuesta de las peticiones y en algunos casos extremos, pérdida de información, deteriorando la calidad del servicio ofrecido. Al ser SoyMomo un producto enfocado a la seguridad, se hace crítico solucionar este problema, ya que la pérdida de información sensible podría marcar la diferencia en el bienestar de los usuarios.

1.2.3. Expectativas de Crecimiento

Debido a que SoyMomo aún es un producto nuevo y en proceso de expansión internacional, desde 2016 a la fecha (18/03/2019) se ha doblado cada año la cantidad de dispositivos vendidos.

Esto se traduce en aproximadamente 35 mil relojes SoyMomo, 50 mil usuarios (padres que utilizan la aplicación) y *peaks* de hasta 7 mil conexiones concurrentes en el servidor.

1.3. Objetivos

El objetivo general de este trabajo de memoria es desarrollar un nuevo servidor para el back-end de SoyMomo, que sea escalable y que permita utilizar de forma más eficiente los recursos del procesador y memoria, disminuyendo los tiempos de respuesta y evitando la pérdida de información.

Además, se busca mejorar el diseño y la implementación del software de manera que, con una mirada a largo plazo, pueda ser escalado horizontal y/o verticalmente ante un potencial aumento de clientes.

Para alcanzar las metas propuestas, se desprenden entonces los siguientes objetivos específicos:

- Reescribir el software existente para que pueda utilizar los recursos del sistema de manera eficiente, aplicando principios de paralelismo y concurrencia.
- Optimizar la cantidad de consultas realizadas a la base de datos para minimizar la pérdida de información.

1.4. Descripción del Trabajo Realizado

- Se realizó un análisis de la infraestructura inicial, midiendo los tiempos de respuesta del servidor y del sistema BaaS; la cantidad de consultas promedio por dispositivo, el porcentaje de pérdida de información y el porcentaje de uso de CPU del servidor.
- Se identificaron los puntos críticos de la infraestructura original.
- Se implementó *multithreading* en el *software* existente, replicando la ejecución del programa de forma independiente en múltiples *cores*, sin comunicación entre ellos; y luego refactorizando el código para que el mismo proceso se encargue de la comunicación entre *cores*.
- Se realizó la reingeniería del *software* y se reescribió usando el lenguaje de programación Go, utilizando *multithreading* para que pueda responder solicitudes de forma paralela.
- Se categorizaron las consultas realizadas al sistema BaaS y se agruparon en lotes (o *batches*), donde fuera factible, para reducir la cantidad total de consultas realizadas.
- Finalmente se volvió a tomar muestras de las métricas iniciales, además de realizar pruebas de esfuerzo del sistema, para confirmar empíricamente las mejoras teóricas de la solución propuesta.

1.5. Estructura del documento

El siguiente capítulo define el marco teórico sobre el cual se sustenta este trabajo de memoria. Posteriormente en el capítulo 3 se realiza un análisis de la infraestructura inicial, junto con las métricas que se buscan mejorar.

Tabla 1.1: Duración del trabajo realizado

Tarea	**Duración**
Análisis Inicial e identificación de puntos críticos	1 semana
Refactoring e implementación inicial en Node.js	1 semana
Reingeniería del *software* y nueva implementación en Go	3 semanas
Implementación del *batching* de consultas a la base de datos	1 semana
Análisis final y resultados	1 semana

El capitulo 4 presenta los pasos para el desarrollo del sistema con *multithreading* que aborda el problema del mal uso de recursos. En el capítulo 5 se evalúan las posibilidades de optimización de consultas a la base de datos y se propone una mejora.

En el capítulo 6, se muestran los resultados obtenidos al evaluar el nuevo *software*. Finalmente se presentan las conclusiones y el trabajo a futuro.

Capítulo 2

Marco Teórico

En este capítulo se introducen los conceptos involucrados en el trabajo de memoria realizado. También se describe como el lenguaje de programación escogido aplica estos conceptos para llegar a los objetivos planteados.

2.1. Reingeniería de Software

La reingeniería de software es la examinación, análisis y posterior modificación de un *software* existente para reconstruirlo en una nueva forma, junto con la implementación de este nuevo *software*.[7]

El objetivo de este proceso es entender el *software* existente en sus especificaciones, diseño, e implementación y posteriormente reimplementar este software, mejorando su funcionalidad, rendimiento o implementación misma.

En el caso particular de este trabajo de memoria, el trabajo se concentrará en el análisis y mejora del rendimiento e implementación del servidor de SoyMomo, con énfasis en la escalabilidad del software.

2.2. Escalabilidad

Se define la escalabilidad como la capacidad de un sistema para acomodar un número creciente de objetos, para procesar volúmenes crecientes de trabajo de forma apropiada y/o que sea susceptible a su crecimiento. Por el contrario, se dice que un sistema no es escalable cuando el costo adicional del aumento del trabajo o del tráfico hacia él es excesivo, o derechamente no puede funcionar cuando su carga de trabajo aumenta.[1]

Este costo se puede cuantificar de muchas formas, incluyendo tiempo de respuesta, espacio físico, memoria o dinero, entre otros.

Para analizar el *back-end* de SoyMomo, se hablará de escalabilidad vertical y horizontal[4]:

Se define la **escalabilidad vertical** como la capacidad de un sistema para mejorar su rendimiento a través de la inyección de recursos a un mismo nodo, por ejemplo, aumentando la cantidad de *cores* o de memoria RAM en una máquina virtual.

Se define la **escalabilidad horizontal** como la capacidad de un sistema para mejorar su rendimiento a través de la inyección de más nodos a éste, por ejemplo, agregando redundancia a un servidor web para poder servir más peticiones.

2.3. Concurrencia y Paralelismo

Se define **concurrencia** como la capacidad de un sistema para permitir que múltiples tareas distintas avancen en su ejecución dentro de un lapso determinado de tiempo.[2] Esto no necesariamente significa que las tareas se ejecuten en el mismo instante.

En sistemas con un solo procesador, por ejemplo, la concurrencia puede ser aplicada multiplexando el tiempo de ejecucción entre todas las tareas disponibles, tomándolas de forma secuencial o bajo algún criterio de *fairness*, ejecutando un subconjunto del total de instrucciones en el tiempo asignado a cada una de ellas por parte del *scheduler*.

El **paralelismo** (también nombrado por algunos autores como *verdadera* concurrencia), se define como la capacidad de un sistema para ejecutar más de una tarea de forma simultánea.[8]

Para poder usar esta capacidad, el sistema debe ser multiprocesador o multicore, y el software a ejecutar debe estar diseñado para ejecutarse de forma paralela. Por ejemplo: el servidor de SoyMomo está desarrollado en Node.js, el cual se ejecuta por defecto en un solo hilo o *thread*, por lo que a pesar de estar ejecutándose en una máquina con 4 *cores*, solo hace uso de uno, desperdiciando los demás recursos del sistema.

Al aumentar el grado de paralelismo, se espera conseguir mejoras en el rendimiento de un *software*. Sin embargo, esto depende no solo de la cantidad de procesadores, sino también de las instrucciones enviadas a cada uno de ellos.

Para poder cuantificar la mejora potencial en el rendimiento de un sistema al añadir más procesadores en paralelo a una aplicación que tiene componentes en serie y paralelos, se planteó la ley de Amdahl[8] en 1967, que establece una cota superior para esta medida.

Si S es el porcentaje de la aplicación que se ejecuta secuencialmente en un sistema con N cores, y el resto de la aplicación se ejecuta de forma paralela repartiendo el proceso de forma equitativa entre éstos, la fórmula se define de la forma siguiente:

$$mejora \leq \frac{1}{S + \frac{(1-S)}{N}}$$

Aplicando esta fórmula al caso particular de este trabajo de memoria, si se considera que salvo la inicialización y cierre de las respectivas conexiones para los puertos correspondientes a HTTPS y TCP, el resto del programa se puede ejecutar de forma paralela, y que disponemos de 4 *cores* en la máquina virtual, de acuerdo a la Ley de Amdahl podríamos alcanzar una mejora teórica en de un factor cercano a 4.

2.4. Comunicación entre procesos y sincronización

La sincronización[6] es un mecanismo que impone restricciones al orden de ejecución de los *threads* dentro de un sistema. A través de la sincronización es posible controlar el orden relativo de la ejecución de los *threads* y resuelve conflictos entre ellos que puedan causar consecuencias no deseadas.

En general la comunicación entre las componentes que están dentro de un sistema concurrente ocurre de 2 formas: a través de memoria compartida, o a través del envío de mensajes.

Los sistemas que utilizan memoria compartida para su comunicación deben implementar alguna forma de sincronización entre procesos, ya sean semáforos, *mutexes* o monitores, para coordinar entre los distintos *threads* que operan sobre un mismo sector de memoria. De no hacerlo, podrían obtenerse resultados inesperados en la ejecución del programa, producto de *data-races*.

Por el contrario, al comunicarse a través del envío de mensajes, la sincronización se da de forma implícita, ya que por definición un mensaje debe ser enviado antes de poder ser recibido.

Un caso particular de esta forma es la comunicación de procesos secuenciales (CSP[3] por su nombre en inglés),que corresponde a un lenguaje formal usado para describir ciertos patrones de interacción en sistemas concurrentes que se comunican a través de mensajes.

CSP permite describir sistemas en términos de componentes o procesos independientes que sólo se comunican entre ellos a través de mensajes a través de canales bidireccionales. Para que esta comunicación ocurra, un proceso debe nombrar a otro como emisor, y este segundo proceso debe nombrar al primero como receptor. En su forma más primitiva, esta comunicación es bloqueante, es decir, ninguno de los 2 procesos puede avanzar hasta que emisor y receptor hayan completado la comunicación.

Es por esta razón que la comunicación a través de mensajes es por definición *thread-safe*, ya que siempre hay un solo proceso que tiene acceso a una estructura compartida en un momento determinado.

2.5. Go

Go es un lenguaje de código abierto, compilado, concurrente, estáticamente tipado y con recolector de basura desarrollado por Google.

Uno de sus objetivos es facilitar la programación concurrente a través de la composición de funciones independientes de código secuencial, utilizando los principios propuestos en CSP[5][9].

Go mejora la implementación de CSP agregando canales unidireccionales y canales con *buffers*, lo que permite que el canal no se bloquee al escribir en él si es que hay espacio en el *buffer*.

Como unidad básica de la programación concurrente, Go utiliza *goroutines*. Una *goroutine* no es un *thread* en el sentido usual de la palabra, sino que corresponde a una función con su propia pila de tamaño dinámico multiplexada en un *thread* del sistema operativo.

Estas funciones son muy livianas; a modo de comparación, un *thread* de Java tiene una pila de tamaño fijo de 1MB, mientras que una *goroutine* tiene una pila variable, comenzando en un tamaño de 2KB. Además, cualquier función puede ejecutarse en una *goroutine* anteponiendo la *keyword go*. Es por esto que no es extraño tener miles o cientos de miles de *goroutines* ejecutándose de forma concurrente en programas desarrollados en Go.

Además, si una *goroutine* se bloquea, solo bloquea el *thread* en el que se está ejecutando, pero no afecta al resto de *goroutines*, las cuales pueden continuar su ejecución en alguno de los *threads* restantes.

La comunicación idiomática entre *goroutines* se realiza a través de canales, que son objetos de primera clase en el lenguaje (incluso es posible tener canales que transmiten otros canales). Para casos particulares, el lenguaje también soporta las estructuras tradicionales de comunicación a través de memoria compartida: punteros, *mutexes*, semáforos y monitores.

La ejecución de las *goroutines* es manejada de forma automática por el lenguaje, haciendo un uso eficiente de todos los recursos del sistema asignados al programa.

Capítulo 3

Análisis de la infraestructura original

Tal como se muestra en la figura 1.2.1, la arquitectura del *back-end* consta de 3 componentes principales: un servidor TCP y un servidor HTTPS funcionando en un mismo proceso, además de un sistema BaaS que contiene la base de datos. A continuación se analizarán en detalle estos componentes.

3.1. Protocolo de comunicación

Para poder comunicarse con el servidor, los relojes SoyMomo envían mensajes a través de un *socket* TCP. Esta conexión se mantiene abierta indefinidamente a través del uso de *keepalive*, sin necesidad de estar enviando mensajes constantemente. En el caso de perder la conexión, el servidor guarda los mensajes pendientes y los envía posteriormente una vez que la comunicación se recupere.

Si bien se podría argumentar que el protocolo UDP sería más apto ya que se envían mensajes individuales y no un *stream* de datos; al tratarse de dispositivos móviles, éstos no tienen una dirección IP fija o estable en el tiempo, lo que no garantiza que la comunicación desde el servidor hacia el reloj se garantice correctamente.

Los mensajes enviados y recibidos por el reloj obedecen al formato
[XX*YYYYYYYYYY*ZZZZ*CCCC], donde:

- XX es el código de fabricante (2 caracteres ASCII)
- YYYYYYYYYY es el número identificador del Momo (10 dígitos, formato decimal)
- ZZZZ es el largo del contenido CCCC (4 dígitos, formato hexadecimal)
- CCCC es el comando a ejecutar (número variable de caracteres ASCII, determinado por el largo ZZZZ)

El listado de comandos y su uso se muestra en el anexo A.

3.2. Arquitectura del sistema

3.2.1. Sistema BaaS (Parse Server)

Parse es un ecosistema BaaS de código abierto (a partir de 2016), orientado al desarrollo de aplicaciones móviles. Entre sus componentes se encuentra el Parse Server y múltiples SDK de cliente para diferentes plataformas (Android, iOS, Unity, Javascript, etc.). Este servidor está desarrollado en Node.js y Express (framework orientado a web), y provee una API REST para interactuar con una base de datos MongoDB. Además, permite la ejecución de código en la nube de forma serverless o FaaS a través del sistema Cloud Code. La implementación usada en este proyecto corresponde a la versión 2.3.7 del Parse Server, a través de Back4App: empresa que provee instancias de Parse Server y se encarga de su administración y hosting. Este sistema se comunica con las Apps a través de alguno de los SDKs de cliente y también con el servidor HTTP en respuesta a cambios en la base de datos (triggers) generados por las apps.

3.2.2. Servidor TCP

Desarrollado en Node.js, su función principal es actuar como adaptador entre el cliente (Momo) y las demás componentes del back-end. Entre sus funciones se encuentran: Proveer una interfaz para recibir y enviar información a través de sockets TCP. Esto se hace a través del módulo net del lenguaje que entrega una API para crear conexiones TCP asíncronas. Identificar cada socket abierto con el código único correspondiente a cada dispositivo cliente. Parsear las peticiones recibidas a través del socket TCP y enviarlas al BaaS a través del SDK de Parse. Parsear las peticiones recibidas desde el servidor HTTP y enviarlas a los clientes a través de sockets TCP. Realizar peticiones a la API de Geolocalización de Google. Ante una nueva conexión, verificar si existen cambios pendientes en la base de datos y enviarlos al cliente.

3.2.3. Servidor HTTPS

Desarrollado en Node.js y Express, su función principal es actuar como adaptador entre el sistema BaaS y el servidor TCP. Tanto el servidor HTTP como el servidor TCP funcionan en el mismo proceso con el objetivo de poder establecer comunicación entre ambos, por razones que se detallan a continuación. Entre sus funciones se encuentran: Recibir las peticiones del sistema BaaS dirigidas a algún cliente TCP. Identificar si el cliente está conectado al servidor. Si el cliente está conectado, enviar las peticiones al servidor TCP para ser reenviadas al cliente. Si el cliente no está conectado, enviar de vuelta las peticiones al sistema BaaS para ser guardadas en la base de datos, de modo de asegurar su posterior ejecución. Además, provee rutas para Webhooks relacionados a los sistemas de e-commerce y courier de la empresa, que no son relevantes para efectos de este informe.

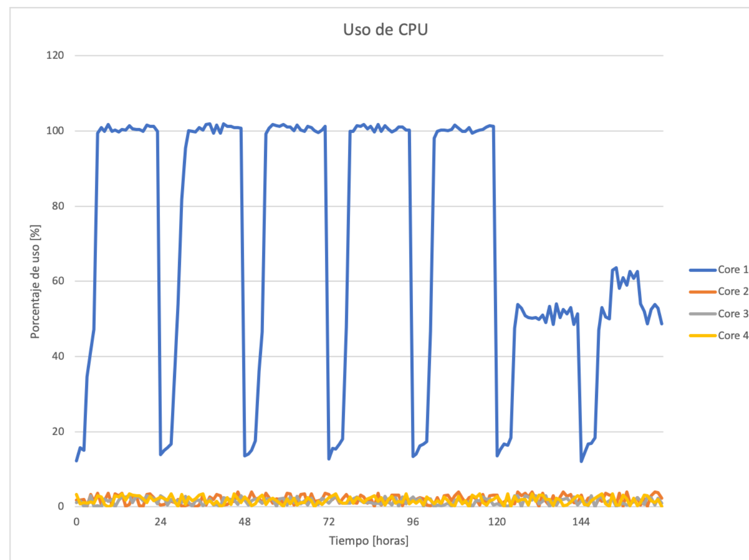

Figura 3.1: Uso de CPU

3.3. Evaluación de la infraestructura

Para evaluar la situación de base, se definieron 6 métricas: porcentaje de uso de CPU, tiempo de respuesta del servidor TCP, porcentaje de pérdida de información en la DB, tiempo de lectura en la DB, tiempo de escritura en la DB y cantidad de peticiones a la base de datos promedio por reloj. Estas mediciones se llevaron a cabo durante 7 días, entre las 00:00 hrs. del 02/10/2017 y las 23:59 hrs. del 08/10/2017, con el fin de monitorear una semana normal de uso. En ese tiempo, el servidor tuvo en promedio alrededor de 800 conexiones concurrentes, peaks de 1200 conexiones concurrentes, y aproximadamente 2200 clientes distintos a través de TCP. Para mantener la calidad de servicio en niveles controlados, el proceso del servidor era reiniciado todos los días a las 00:00 hrs. mediante el uso de pm2, un módulo que funciona como administrador de procesos para Node.js. Este módulo permite ejecutar tareas en formato cron. Esto permitía liberar la CPU y comenzar el proceso desde cero.

3.3.1. Porcentaje de uso de CPU

Para evaluar el porcentaje de uso de CPU se utilizó Keymetrics, solución de analytics y métricas integrada con pm2. La medición se realizó cada 1 segundo, siendo los resultados promediados cada una hora para su posterior análisis.

Se observa de la figura 3.1 que entre 6:00 y 7:00 el porcentaje de uso del *core 1* de la CPU comienza a crecer rápidamente, alcanzando el 100 % entre las 7:00 y 8:00 hrs de lunes a viernes, mientras que los otros *cores* no se encuentran trabajando, salvo por procesos inherentes al sistema operativo. Este porcentaje no decae durante el resto del día ya que se observa un fenómeno similar al *thrashing*, donde la memoria de un computador se encuentra en un estado constante de paginamiento, excluyendo el funcionamiento del programa en ejecución.

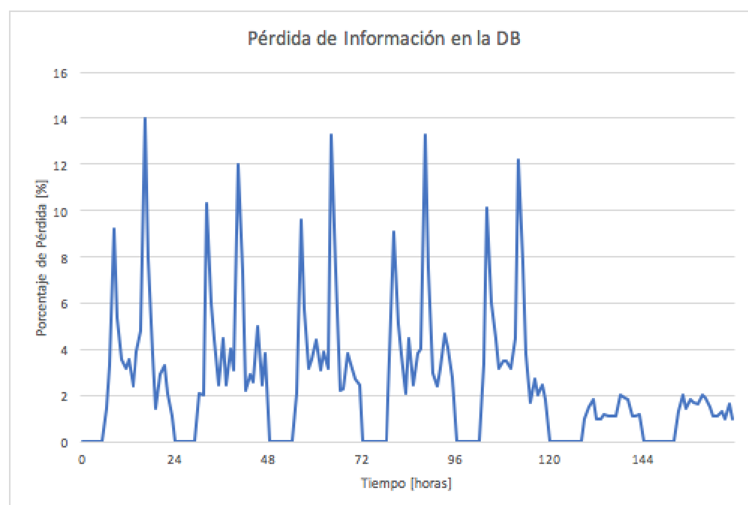

Figura 3.2: Pérdida de Información

En este caso, el recolector de basura del lenguaje se encuentra constantemente funcionando, lo que impide el avance fluido del proceso, lo que a su vez genera más trabajo para el recolector de basura, generando un loop infinito.

3.3.2. Porcentaje de pérdida de información

Para evaluar el porcentaje de pérdida de información se realizó un análisis histórico de los logs del servidor, cuantificando el número total de peticiones a la base de datos y el número de peticiones fallidas por pasar el límite de peticiones (error 155 de Parse). Peticiones fallidas por otros errores no fueron consideradas como pérdida de información debido a que su causa no tiene relación con el número de peticiones realizadas, sino más bien con errores de conexión y/o de programación de código. Estos valores fueron calculados cada hora durante el periodo de medición.

Es posible identificar en la figura 3.2 peaks de pérdida de información alrededor de las 8:30 hrs. y 17:00 hrs. en los días de semana, lo cual tiene relación con los horarios de entrada y salida del colegio de los niños, donde se realiza la mayor cantidad de peticiones. En otros horarios, se observa un menor porcentaje de pérdida, sin embargo este problema está presente todos los días de la semana.

3.3.3. Tiempo de respuesta servidor TCP

Para esta medición se simuló un cliente TCP en Node.js con un identificador falso para que realizara una consulta al servidor cada 1 segundo y se midió el RTT de cada consulta. Los resultados fueron agregados cada hora para un mejor análisis.

De los resultados mostrados en la figura 3.3 es posible observar una correlación entre el tiempo de respuesta y el aumento del porcentaje de uso de CPU, con un tiempo base de

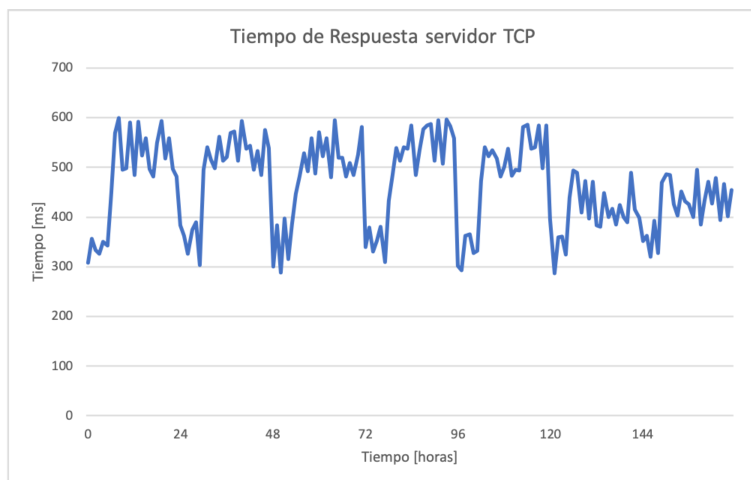

Figura 3.3: Tiempo de respuesta

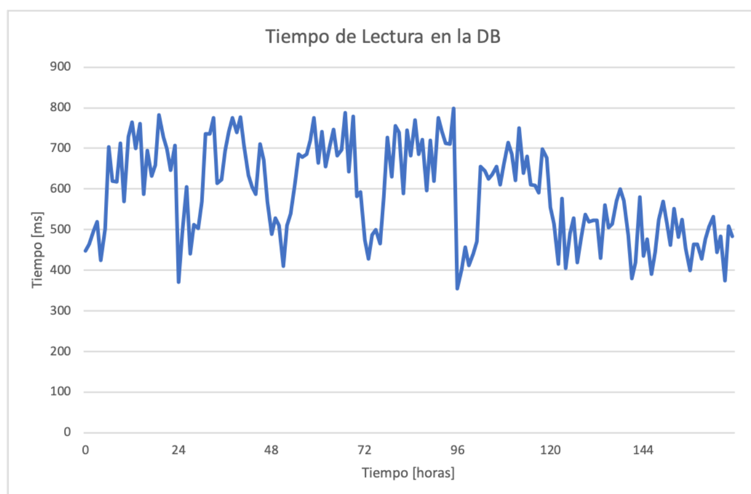

Figura 3.4: Tiempo de lectura en DB

aproximadamente 300 [ms] en condiciones óptimas.

3.3.4. Tiempo de escritura y lectura en la DB

Para estas mediciones se realizaron escrituras y lecturas cada 10 segundos en una tabla de la base de datos creada especialmente para este proceso, utilizando el SDK de Parse para Javascript. Se agregaron los resultados cada hora para mejor análisis.

En los resultados mostrados en las figuras 3.4 y 3.5 es posible observar un aumento en el tiempo de respuesta al realizar operaciones en la DB durante el día, con un tiempo base de aproximadamente 400 [ms] en condiciones óptimas, siendo las lecturas marginalmente más rápidas que las escrituras. Esto se produce debido al aumento en la cantidad de clientes conectados.

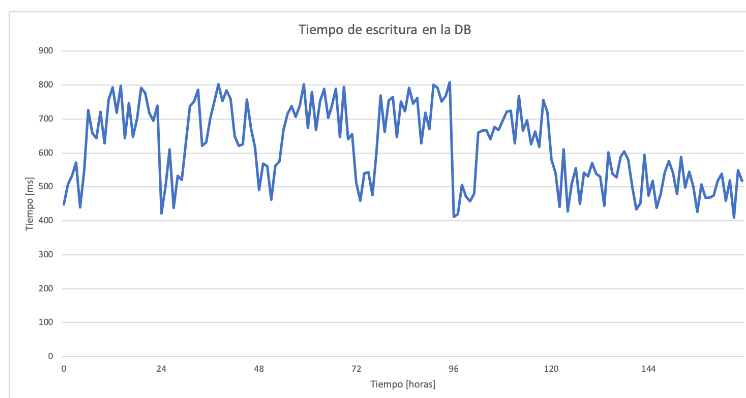

Figura 3.5: Tiempo de escritura en DB

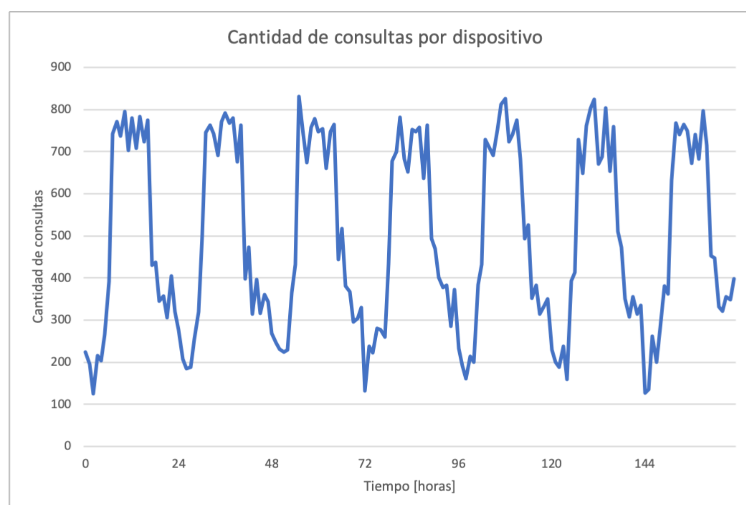

Figura 3.6: Consultas promedio por dispositivo

3.3.5. Cantidad de peticiones a la DB por dispositivo

A través del análisis de los logs, se agruparon por tipo los mensajes que envía cada dispositivo en un día, luego se asoció cada tipo de mensaje con la cantidad de consultas necesarias a la base de datos para completar la operación, según lo indicado en la tabla 3.1

En las peticiones que tienen más de una variante (por ejemplo: el comando TK puede usarse para confirmar la recepción de un mensaje, o para enviar un mensaje nuevo), se realizó un promedio de las consultas necesarias para cada caso, ponderado por la cantidad de ocurrencias, para simplificar el análisis posterior.

Además, hay comandos que no realizan ninguna consulta, simplemente se descartan, ya que son redundantes con otros.

Finalmente, se muestran los resultados promedio por hora en la figura 3.6.

Tabla 3.1: Cantidad de peticiones a la DB por dispositivo

Comando	**Cantidad de peticiones promedio**	**Comandos/día**
LK	4	687758
UD, UD2, UD3	4	431290
TKQ	1,34	3575052
AL	2	381290
TK	3	878
UPLOAD, SOS, LZ, PEDO, WALKTIME, SILENCETIME, FLOWER, REMIND, GSMANT, WHITELIST1, WHITELIST2, PHB, PHB2, PROFILE	2	1553

3.4. Análisis

De acuerdo a los resultados obtenidos, se pueden identificar 2 problemas principales: la pérdida de información, y el porcentaje de uso de la CPU. En una situación óptima, el porcentaje de pérdida de información debería ser cero y la carga de trabajo en la CPU debería repartirse entre todos los *cores* disponibles; solo llegando al 100 % en casos excepcionales y por períodos cortos de tiempo.

Capítulo 4

Multithreading

Uno de los problemas del sistema existente era que usaba el 100 % de uno de los 4 *cores* disponibles en el procesador, mientras que los 3 restantes no estaban trabajando.

Debido a que este problema fue encontrado en el ambiente de producción y el funcionamiento del sistema se hacía insostenible, se optó por separar la solución en partes: refactorizar el sistema existente para aplicar *multithreading* para resolver el problema a corto plazo; y volver a desarrollar el *software* con el úso de múltiples threads como foco del diseño con el objetivo de obtener un programa mantenible a largo plazo.

4.1. Multithreading en Node.js

Si bien Node.js es un entorno de ejecución de Javascript diseñado para ejecutarse en un solo hilo, existe soporte para ejecutar más de un hilo a la vez, tanto de forma nativa como a través de librerías externas.

4.1.1. Primeros pasos

Un primer acercamiento al multithreading fue usar pm2, la misma librería que nos permite reiniciar el servidor en intervalos de tiempo definidos. Esta librería también nos permite replicar un proceso tantas veces como queramos, además de hacer el balance de cargas entre las conexiones de forma automática entre todos los procesos disponibles.

Esta solución intermedia permitió solucionar parcialmente el problema, sin escribir ninguna línea de código. Se distribuyó la carga entre los 4 *cores* disponibles, pero se degradó la calidad del servicio para los usuarios, ya que aumentó considerablemente el tiempo que pasaba entre que un usuario enviaba un comando al reloj y que el comando era recibido por éste.

Esto se debe a que como el servidor actúa de adaptador entre los clientes que se conectan vía HTTPS y los relojes que se conectan vía TCP, es necesario que el proceso tenga conocimiento de ambos clientes para realizar la comunicación. Sin embargo, como el balance de cargas se realizaba de forma automática y no existía comunicación entre procesos, ocurría que la solicitud HTTPS llegaba a un proceso X, mientras que el socket estaba conectado a un proceso Y.

Lo anterior ocasionaba que la solicitud no llegara de forma inmediata, sino que el servidor asumía al reloj como desconectado y luego guardaba la petición en la DB. Posteriormente el reloj tenía que solicitar los comandos pendientes para poder proceder con la modificación.

4.1.2. Solución con child process y fork

Child process es un módulo que permite instanciar nuevos procesos desde un proceso existente en Node.js. Cada proceso instanciado a través de este módulo, se ejecuta en su propio entorno de ejecución y tiene su propia pila de memoria.

En particular, se ocupó la llamada a *fork()* del módulo *child_process*, que permite crear un nuevo proceso, manteniendo un canal de comunicación entre el proceso padre y el(los) proceso(s) hijo(s). A pesar de compartir el nombre, esta llamada a *fork()* no clona el proceso original como ocurre en *POSIX*[1].

Se separó el proceso original que se encargaba de hacer todas las funciones en 2 tipos de procesos: el proceso padre que contiene al servidor HTTPS y se encarga de enrutar las peticiones y procesos hijos que se encargan de mantener los *sockets* abiertos con los relojes para entregar los mensajes, tal como se muestra en la figura 4.1.

Al iniciarse, el proceso padre crea N-1 procesos hijos, donde N es el número total de *cores* del procesador, ocupando todos los recursos disponibles del sistema. Estos procesos son creados a través de un objeto encargado de realizar el *fan-out* de solicitudes hacia los respectivos procesos hijos. Para esto, este objeto guarda las referencias a los canales de comunicación de cada proceso hijo, además de mantener un diccionario con la relación reloj-proceso, de tal forma de enrutar los mensajes de forma correcta.

Posteriormente, una referencia a este objeto es entregada al servidor HTTPS para que al recibir una solicitud, esta sea enrutada al proceso hijo correspondiente para ser luego enviada al reloj a través del *socket*. Si el reloj al que la petición está dirigida no está conectado, se envía al proceso con menos carga.

Los procesos hijos por su parte, se encargan de mantener las conexiones TCP y de notificar al proceso padre cuando un reloj se conecta o se desconecta a través de los canales establecidos entre procesos. Además, están encargados de realizar las operaciones que correspondan en la base de datos de acuerdo a los mensajes recibidos desde los relojes y enviar los mensajes recibidos desde los canales.

[1] http://man7.org/linux/man-pages/man2/fork.2.html

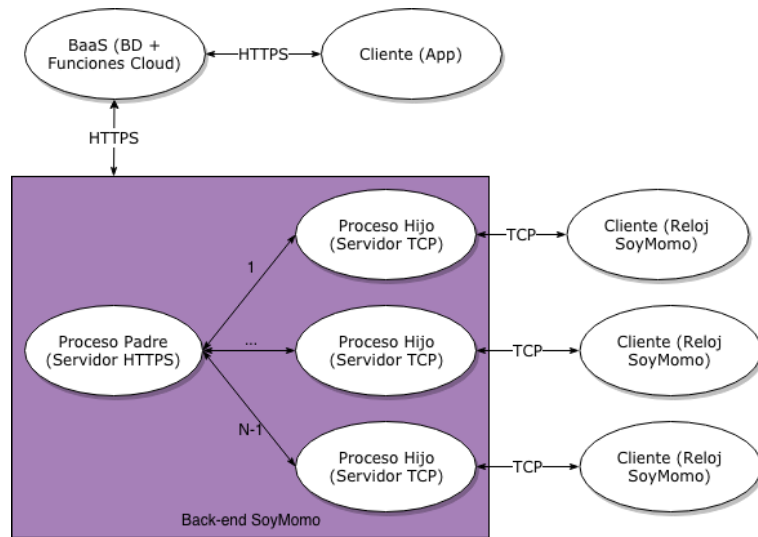

Figura 4.1: Implementación de multithreading en Node.js

Esto permitió mejorar tanto el uso de la CPU como la calidad del servicio que se había perdido con la solución inicial, sin embargo, se observa en la figura 4.2 que el *core* que ejecutaba el proceso padre realizaba mucho menos trabajo que los procesos hijos, por lo que aún no se alcanzaba una óptima utilización de recursos.

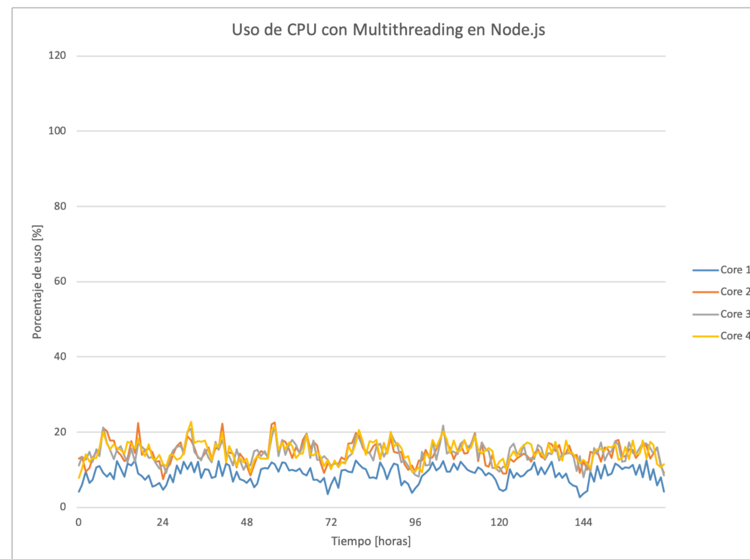

Figura 4.2: Uso de CPU al utilizar multithreading en Node.js

4.2. Multithreading en Go

Uno de los problemas que tenía la solución anterior era que no ocupaba de forma eficiente los recursos, ya que incluso al implementar las mejoras con *multithreading*, existía un thread que realizaba considerablemente menos trabajo que los demás. En *go*, el lenguaje es el encar-

gado de asignar de forma eficiente la ejecución de las distintas *goroutines* a uno de los *cores* disponibles en el sistema[2].

Luego, el problema se reduce a descomponer el programa en *goroutines* de forma eficiente y a establecer de forma correcta la comunicación entre ellas.

4.2.1. Diseño del Software

La solución anterior, separaba el problema en 2 partes: servidor HTTPS y servidor TCP. Usaba la comunicación ya establecida entre los distintos *threads* para poder pasar mensajes de un lado a otro. En el caso de *go*, la comunicación entre *goroutines* debe realizarse de forma manual a través del uso de canales que provee el lenguaje.

Para separar las funciones de cada *goroutine*, se agregó una componente adicional al diseño, encargada de realizar el *fan-out* de las solicitudes entrantes a través de HTTPS y derivarlas a las *goroutines* que correspondan a cada *socket*. Esta comunicación se realiza a través de canales especializados para cada propósito.

El diseño del software se ejemplifica en la figura 4.3.

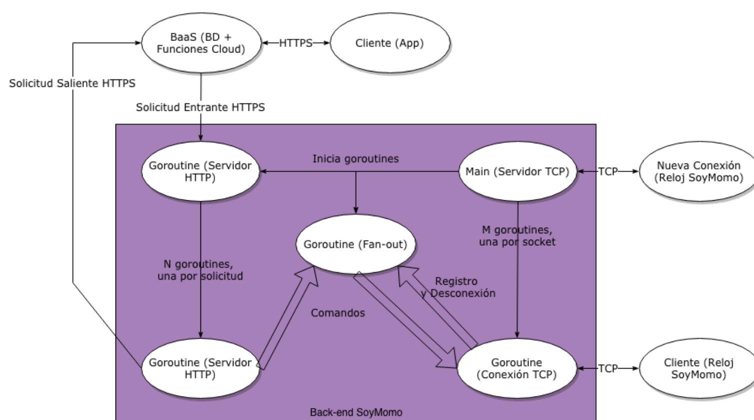

Figura 4.3: Diseño del software en Go

4.2.2. Main y servidor TCP

En muchos lenguajes de programación, para iniciar un programa es necesario tener una función principal a ejecutar, llamada *main()*. *Go* no es la excepción.

En este caso, la función *main()* se encarga de inicializar los canales de comunicación entre las componentes, y posteriormente las *goroutines* que recibirán esos canales como argumentos para ser utilizados durante la ejecución del programa.

[2]Es posible asignar una cantidad fija de *cores* a un programa en *go*, sin embargo, por defecto se utilizan todos los *cores* disponibles

Se crean los siguientes canales de comunicación:

- **Canal de entrada**: Canal unidireccional que transmite comandos con un *buffer* de hasta 100 comandos.
- **Canal de registro**: Canal unidireccional que transmite *structs* del tipo Registro, que contienen el identificador de un reloj en formato *string*, además de un canal que recibe comandos, con un *buffer* de hasta 100 Registros.
- **Canal de desconexión:** Canal unidireccional que transmite el identificador de un reloj en formato *string*, con un *buffer* de hasta 100 *strings*.

Luego se inicializan las siguientes *goroutines*:

- Se crea una *goroutine* correspondiente al Servidor HTTPS.
- Se crea una *goroutine* que realizará el *fan-out* de comandos entre el servidor HTTPS y los *sockets* TCP.
- Se utiliza la *goroutine* correspondiente a la función *main()* como servidor TCP.

El servidor TCP queda en un ciclo infinito escuchando conexiones nuevas. Luego, para cada conexión recibida, crea una *goroutine* que recibe los canales de registro y desconexión.

Esta *goroutine* inicializa un contador de *timeouts* y crea un canal que recibe comandos, el que posteriormente será enviado a la *goroutine* encargada de realizar el *fan-out* de comandos en el *struct* de Registro.

Luego se entra en un loop infinito que realiza los siguientes pasos:

1. Se intenta leer desde el *socket* con un *timeout* de 1 segundo.
    - Si se obtuvo un mensaje se realizan los siguientes pasos:
        (a) Se parsea el mensaje, creando un nuevo comando.
        (b) Si la conexión no estaba registrada, se crea el *struct* Registro con el canal creado anteriormente y el identificador obtenido desde el comando y se envía a través del canal de registro a la *goroutine* encargada del *fan-out*.
        (c) Se ejecuta el comando en una nueva *goroutine*.
    - Si se alcanzó el *timeout* antes de obtener un mensaje, se aumenta en 1 el contador. Si alcanza 300, se envía el identificador a la *goroutine* encargada del *fan-out* a través del canal de desconexión y se cierra el socket, para posteriormente terminar la *goroutine*
2. Si hay algún mensaje en el canal que recibe comandos desde la *goroutine* encargada del *fan-out*, se reciben y se ejecutan hasta que el canal esté vacío.

4.2.3. Servidor HTTPS

La *goroutine* que ejecuta el servidor HTTPS recibe como parámetro el canal de entrada, que nos permitirá transmitir los comandos recibidos hacia la *goroutine* encargada de realizar el *fan-out*.

Se utilizó la implementación estandar de *go* dentro de una clausura para poder capturar la referencia al canal de entrada. Esta implementación recibe como parámetro un enrutador que a su vez contiene las rutas del servicio junto con una función a ejecutar cada vez que se reciba una solicitud dirigida a esa ruta. Para cada ruta, esta función será ejecutada en una *goroutine* independiente, de tal forma que el servidor pueda responder a múltiples peticiones de forma concurrente.

Se definió una ruta que permite recibir comandos desde el sistema BaaS, cuya función lee y parsea los comandos recibidos y los envía a la *goroutine* encargada de realizar el *fan-out* a través del canal destinado para ello y luego responde con un código 200[3] al cliente que realizó la solicitud.

4.2.4. *Fan-out* de comandos

Esta *goroutine* es la encargada de despachar los comandos a los distintos destinatarios. Se inicializa recibiendo los canales de entrada, registro y desconexión, que permitirán la comunicación entre las distintas *goroutines*.

Para poder enviar los comandos a los destinatarios correctos, mantiene un diccionario con llaves de tipo *string*, correspondientes a los identificadores de los relojes y valores de tipo *chan* (canales en *go*), que corresponde al canal de comunicación con la *goroutine* correspondiente a ese dispositivo.

Una vez inicializado el diccionario, entra en un ciclo infinito que escucha en los canales de entrada, registro y desconexión, realizando las siguientes acciones dependiendo del canal donde reciba el mensaje:

- **Canal de entrada:** al recibir un mensaje, se busca en el diccionario si existe el dispositivo de destino.
  - Si el dispositivo existe, se obtiene el canal y se envía el comando a través de él.
  - Si el dispositivo no existe, se crea una nueva *goroutine* que guarda los comandos en la base de datos, para ser recuperados por el dispositivo en el futuro.
- **Canal de registro:** al recibir un mensaje, se crea una nueva entrada en el diccionario, con los datos recibidos.
- **Canal de desconexión:** al recibir un mensaje, se elimina la entrada correspondiente en el diccionario.

[3]En el protocolo HTTP, el código 200 indica que la solicitud se realizó correctamente.

Capítulo 5

Optimización de consultas a la base de datos

Otra de las oportunidades de mejora del software es reducir la cantidad de consultas realizadas a la base de datos, con el fin de reducir la pérdida de información debido al límite de consultas por segundo.

En el software original, las consultas se realizan de forma individual, sin embargo el sistema BaaS permite agrupar consultas en lotes de hasta 50 consultas, siempre y cuando correspondan a operaciones de creación, modificación o eliminación de registros.

Estos lotes no están limitados a un tipo de operación en específico, ni a una tabla determinada. Por ejemplo, es posible crear un registro en una tabla A y eliminar otro de una tabla B en un mismo lote de consultas. Las lecturas, por el contrario, deben realizarse de manera individual por cada tabla de la que se requiera extraer información.

El objetivo es identificar cuales consultas pueden ser postergadas hasta que se junte un lote de tamaño máximo o haya transcurrido un límite de tiempo apropiado para enviar el lote obtenido hasta entonces.

Estas consultas serán agrupadas en una cola, de tal manera que se procesen en orden de llegada o FIFO[1] por sus siglas en inglés.

5.1. Análisis y categorización de las consultas

Se analizará, por cada comando enviado por el reloj, la secuencia de operaciones sobre la base de datos que se realiza y cada operación se asignará a una de cuatro categorías:

- **Categoría 1:** Corresponde a una lectura y no se puede agrupar, por diseño del sistema.
- **Categoría 2:** No es posible postergar esta consulta, ya que degradaría la calidad del

[1] First in, first out

servicio.

- **Categoría 3:** No es posible agrupar esta consulta, ya que su resultado se utiliza en una consulta posterior.
- **Categoría 4:** Es posible postergar esta consulta para ser agrupada.

5.1.1. Comandos no optimizables

Debido a la naturaleza del producto y servicio ofrecidos, cuyo objetivo es entregar seguridad a los padres respecto de la ubicación de sus hijos, existen 3 comandos para los cuales no es posible postergar ninguna petición sin afectar la calidad del servicio:

- **AL:** Envía una alerta desde el reloj al celular del padre. Pueden corresponder a una señal de SOS (petición de ayuda), enviada de forma manual por el niño; o a alertas generadas de forma automática que indican que el dispositivo tiene batería baja o que el niño se sacó el reloj.
- **UD, UD2, UD3:** Envían datos de la posición del reloj. Esta posición posteriormente es comparada con la posición anterior para saber si el reloj entró o salió a alguna zona segura (por ejemplo, casa o colegio) predefinida por el usuario. En el caso de cumplirse esta condición, una notificación es enviada al celular del padre, avisándole de la situación.
- **TK:** Envía un mensaje de voz desde el reloj hacia el celular del padre. Al ser éste un tipo de mensajería instantánea, no es razonable postergar su procesamiento.

5.1.2. LK: Actualización del porcentaje de batería y conteo de pasos

El comando LK envía el porcentaje de batería restante en el reloj y también la cantidad de pasos dados por el niño durante el día, hasta el momento de la actualización. En respuesta, el servidor realiza una serie de operaciones sobre la base de datos: obtener el registro Reloj, actualizar el porcentaje de batería en el registro, buscar el conteo de pasos correspondiente al día, y modificarlo o crearlo en el caso que no exista. Este flujo se muestra en la figura 5.1.

Para la secuencia de operaciones producida por este comando, solo es posible postergar la actualización del conteo de pasos en el caso que exista o la creación de la relación entre el conteo de pasos y el reloj, en el caso de que no exista.

5.1.3. TKQ: Solicitud de comandos pendientes

El comando TKQ es enviado regularmente por el reloj cada 1 minuto, para solicitar los comandos que puedan estar pendientes si es que el reloj se desconectó del servidor en algún momento. Para esto, se realiza una operación de lectura en la base de datos, se envían los comandos obtenidos al reloj, y para cada uno de éstos, se elimina o se marca como enviado

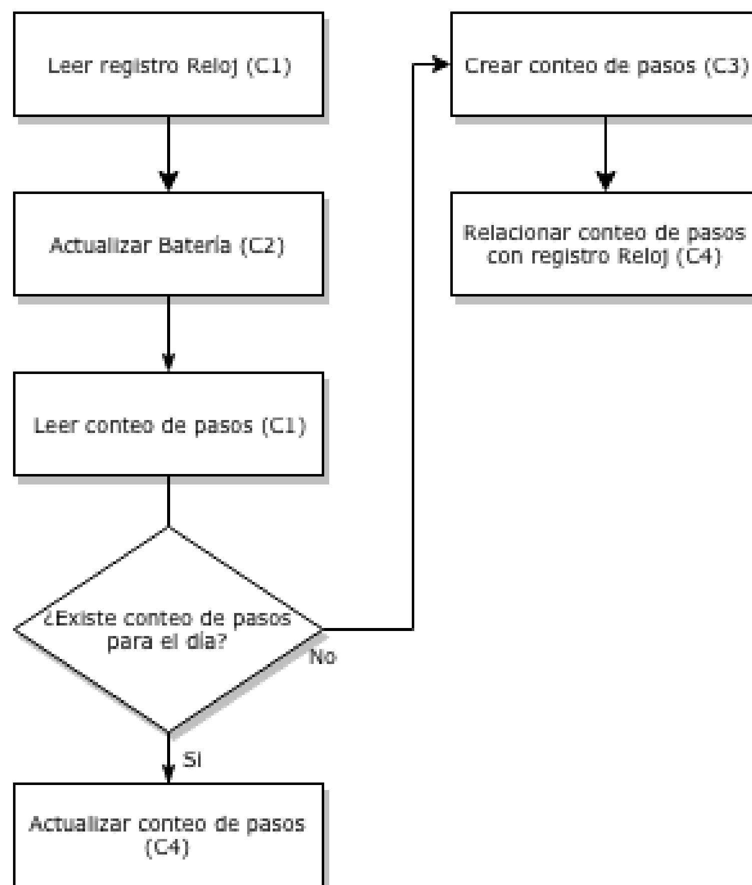

Figura 5.1: Consultas producidas por el comando LK

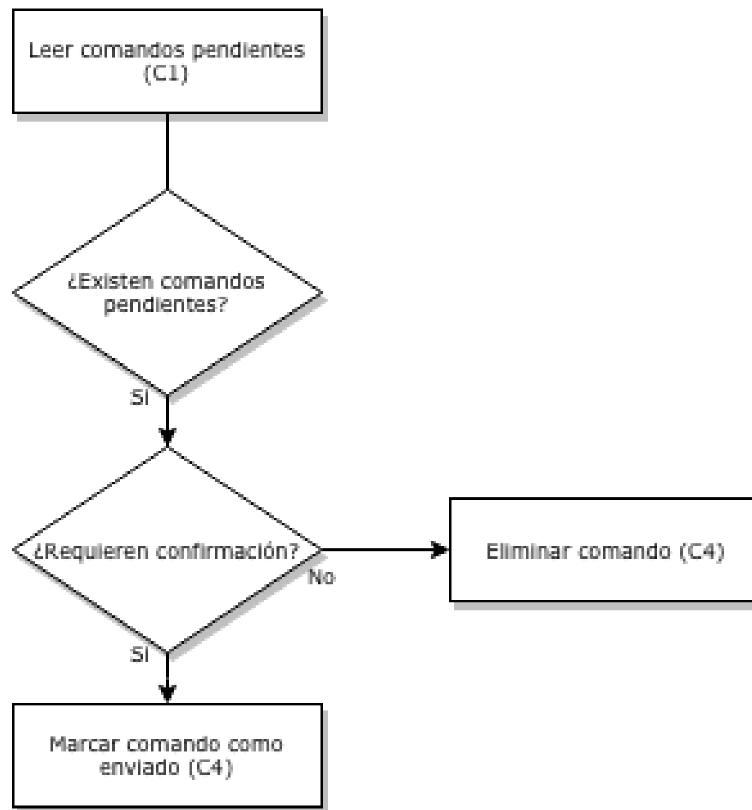

Figura 5.2: Consultas producidas por el comando TKQ

en la base de datos, dependiendo si el comando requiere confirmación de recepción o no. El flujo se muestra en la figura 5.2.

En este caso, hay posibilidad de mejora al agrupar y postergar todas las modificaciones y/o eliminaciones en un mismo lote, siempre y cuando la operación se realice antes de 1 minuto, ya que luego de este tiempo el reloj solicitará nuevamente los comandos pendientes y podrían enviarse de forma duplicada.

5.1.4. Otros comandos: confirmación de recepción

Para los comandos restantes, el reloj envía una confirmación cuando fue recibido correctamente, lo que genera una operación de lectura en la base de datos para el respectivo comando marcado como enviado anteriormente y luego se procede a su eliminación.

Si bien la eliminación de estos comandos parece un buen candidato a postergación, al no haber certeza de cuando se realizará la próxima solicitud de comandos pendientes, es posible que se solicite nuevamente un comando antes de haber sido eliminado, causando operaciones duplicadas en el reloj. Es por esto que no se realizó una optimización en estos comandos.

5.2. Refactorización por lotes

Luego de haber identificado cuales consultas son posibles de postergar y posteriormente agrupar por lotes, es necesario desarrollar un sistema que se encargue de agrupar y ejecutar estas consultas.

Se definió una cola de tamaño 50, que es la cantidad máxima de peticiones que se pueden enviar simultáneamente al sistema BaaS, además de un tiempo máximo de 10 segundos que puede esperar una petición antes de ser enviada. De esta forma, si la cola no se llena en esos 10 segundos, se enviarán todas las solicitudes que estén pendientes hasta que se cumpla el límite de tiempo.

En la práctica, el 95 % de las solicitudes se envía antes de 3 segundos y el 100 % se envía antes de 5 segundos, sin embargo, es importante establecer un límite para cubrir todos los casos de borde.

Debido a que múltiples relojes necesitan encolar solicitudes y las escrituras en una cola no son *thread-safe*, se decidió una *goroutine* encargada de realizar las operaciones de lectura y escritura sobre la cola, además de crear nuevas *goroutines* para realizar las solicitudes.

Para realizar la comunicación entre las *goroutines* correspondientes a cada reloj, y la *goroutine* encargada de manipular la cola, se usó un canal unidireccional con un *buffer* de 100 elementos, para evitar el bloqueo de las *goroutines* de los relojes al escribir sobre el canal. Es importante recordar que los canales en *go* si son *thread-safe*, por lo que es posible escribir en ellos de forma concurrente sin problemas con la integridad de los datos. Un diagrama del sistema se muestra en la figura 5.3.

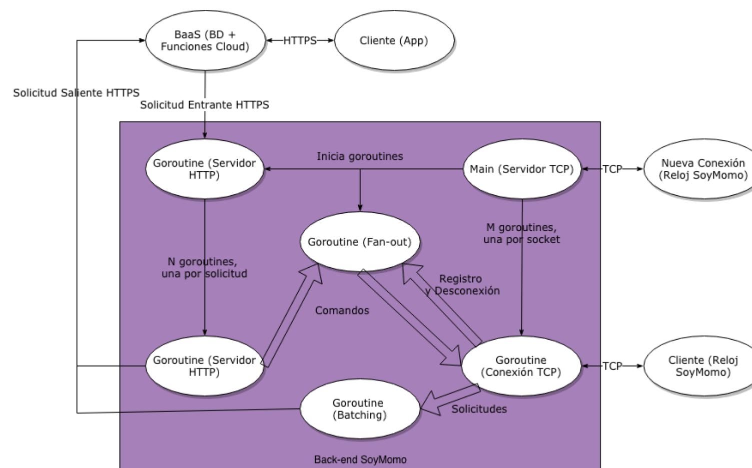

Figura 5.3: Diseño del sistema de *batching*

Luego, para implementar el límite de tiempo, se utilizó un timer. Esta estructura crea un canal a través del cual recibe un evento una vez que se cumple el tiempo.

Finalmente como tenemos 2 canales por los cuales recibimos eventos distintos, se utiliza la expresión *select* del lenguaje, que nos permite esperar la recepción de mensajes desde

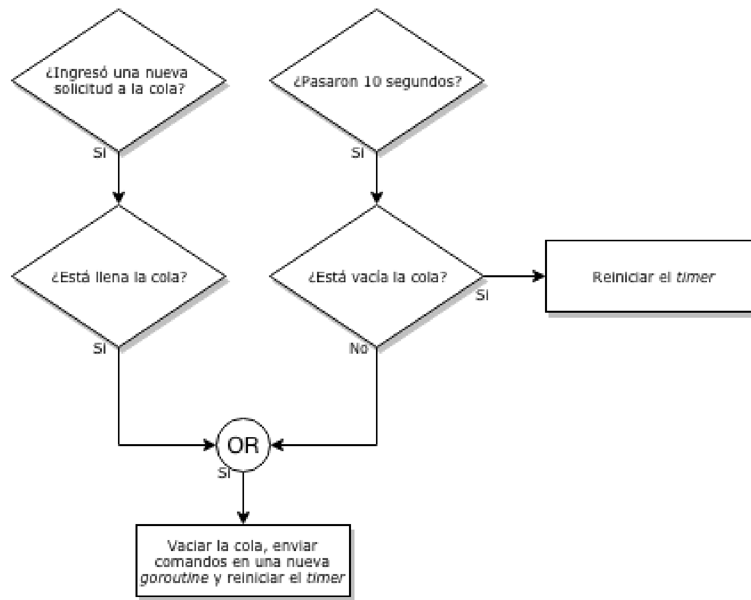

Figura 5.4: Diagrama de flujo del sistema de *batching*

múltiples canales de forma paralela, ejecutando una sección de código con el resultado del primer canal que se encuentre listo para recibir un mensaje.

Para realizar las consultas de la cola, se creará una función que recibirá una lista de consultas y que se encargará de construir y enviar la solicitud POST hacia el sistema BaaS.

Al recibir un mensaje del canal del *timer*, si la cola está vacía, éste se reiniciará. Por el contrario, si la cola tiene elementos, se vaciará la cola y se ejecutará la función en una nueva *goroutine*. Al recibir una nueva solicitud a través del otro canal, si la cola está llena, se vaciará y se ejecutará la función nombrada anteriormente, para posteriormente encolar el nuevo elemento. Si la cola aún tiene espacio, simplemente se encolará la nueva consulta, hasta que se cumpla alguna de las 2 condiciones. Esto se ejemplifica en la figura 5.4.

En este caso, como necesitamos iterar sobre los canales de forma constante, se ubica la expresión *select* dentro de un loop infinito. A diferencia de otros lenguajes, esto no bloquea el procesador, sino que el *scheduler* se encarga de despertar la *goroutine* solo cuando uno de los canales esté listo para recibir un mensaje.

Capítulo 6

Resultados

A continuación se analizarán los resultados obtenidos repitiendo las mediciones iniciales sobre la nueva versión del sistema. Los experimentos se realizaron con 1000 clientes conectados al sistema, replicando la situación inicial, de forma de poder comparar fácilmente los resultados.

6.1. Porcentaje de uso de CPU

Se observa de la figura 6.1 que se utilizan los 4 cores de manera equilibrada, rondando el 13 % de uso de CPU en promedio, y cercano al 20 % en los horarios punta.

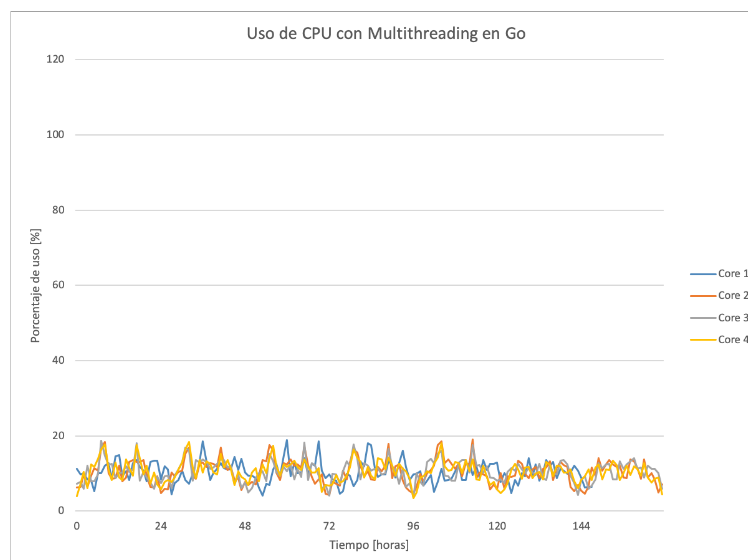

Figura 6.1: Uso de CPU, servidor desarrollado en Go

Además, se realizó una prueba de carga con 10000 clientes simulados, que representan un

crecimiento de 10 veces respecto de la situación original, obteniendo un uso de CPU cercano al 60 %.

Se muestra también el estado de la situación actual, al 17/12/2018, con 2500 clientes en promedio y *peaks* de hasta 4000 clientes. En este caso se observa una mayor distinción en los horarios punta, debido a que hay una mayor varianza entre el número de clientes.

Estos resultados se muestran en la figura 6.2, donde se promedió el uso de todos los *cores* del sistema.

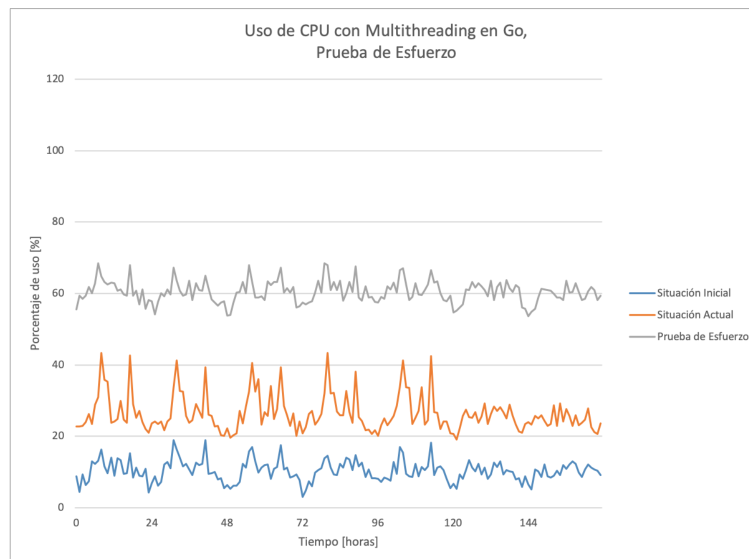

Figura 6.2: Prueba de carga servidor desarrollado en Go

6.2. Porcentaje de pérdida de información

El porcentaje de pérdida de información se redujo a cero. Durante la semana de prueba no se alcanzó el límite de peticiones en ningún momento.

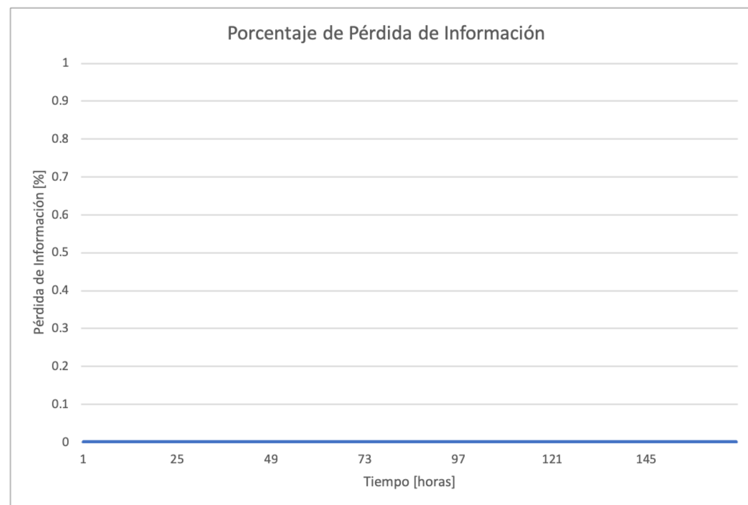

Figura 6.3: Porcentaje de pérdida de información, después del *batching*

6.3. Tiempo de respuesta servidor TCP

Se observa en la figura 6.4 que el tiempo de respuesta del servidor TCP disminuyó considerablemente, hasta en un 50 % en los horarios de mayor trafico. Además, presenta mucho menos varianza, oscilando alrededor de los 300[ms]. La mayor parte del tiempo de respuesta corresponde a la latencia introducida por la distancia entre cliente y servidor, el cual se encuentra alojado en EEUU.

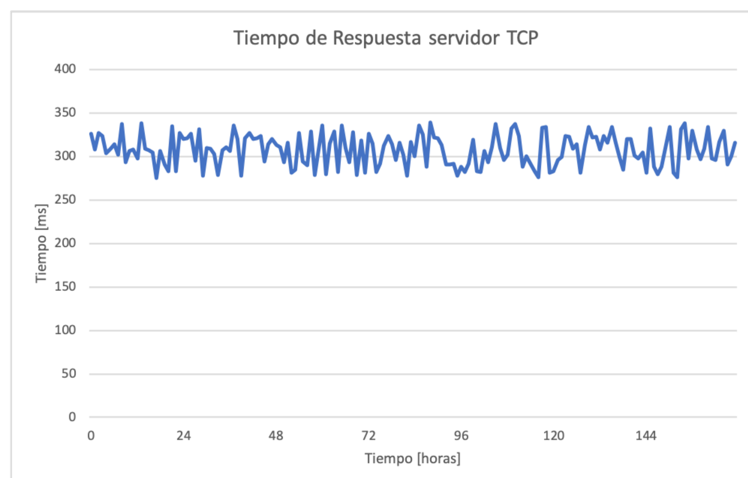

Figura 6.4: Tiempo de respuesta servidor desarrollado en Go

6.4. Tiempo de escritura y lectura en la DB

En las figuras 6.6 y 6.5 se muestran los resultados de los tiempos de lectura y escritura en la base de datos, respectivamente.

No se observa una mejora significativa, ya que si bien al implementar el sistema de *batching* se realizan menos consultas a la API, en general la cantidad de operaciones que realiza la base de datos se mantiene constante.

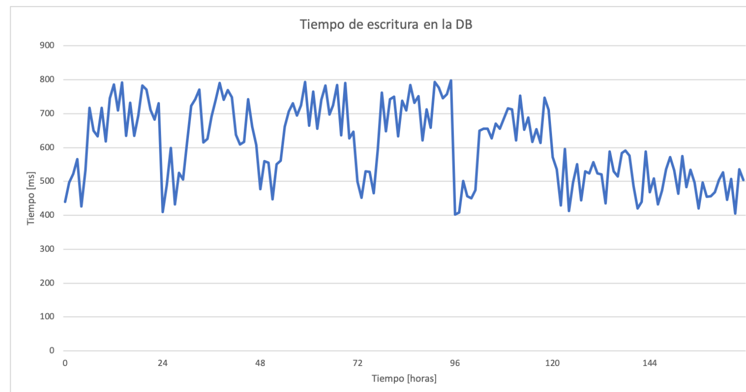

Figura 6.5: Tiempo de escritura en la DB, después del *batching*

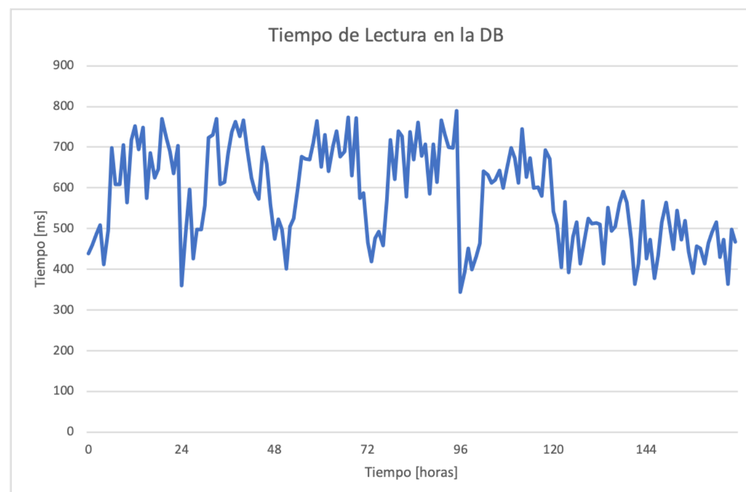

Figura 6.6: Tiempo de lectura en la DB, después del *batching*

6.5. Cantidad de peticiones a la DB por dispositivo

Se observa en la figura 6.7 que la cantidad de consultas por dispositivo disminuyó en un 16 %, de aproximadamente 10 mil consultas al día a cerca de 8400. Este resultado nos permite atender a un 19 % más de clientes con los mismos recursos.

Tabla 6.1: Cantidad de peticiones a la DB por dispositivo, después del *batching*

Comando	**Cantidad de peticiones promedio**	**Comandos/día**
LK	3,25	673942
UD, UD2, UD3	4	446209
TKQ	1,09	3767931
AL	2	361740
TK	3	799
UPLOAD, SOS, LZ, PEDO, WALKTIME, SILENCETIME, FLOWER, REMIND, GSMANT, WHITELIST1, WHITELIST2, PHB, PHB2, PROFILE	2	1632

El detalle de las peticiones realizadas por dispositivo se encuentra en la tabla 6.1.

Figura 6.7: Consultas promedio por dispositivo, después del *batching*

Conclusión

Este trabajo de memoria se centró en la reingeniería de un sistema ya existente y en funcionamiento, con el objetivo de mejorar su rendimiento y lograr la escalabilidad del mismo.

Se afrontó el análisis de la situación original, identificando métricas que permitieron, al final de este trabajo, cuantificar las mejoras obtenidas tras la modificación del sistema.

Además se abordó la reingeniería del software desde distintas aristas, incluyendo diseño de *software*, *profiling* y optimización de la cantidad de operaciones realizadas sobre un sistema externo.

El nuevo diseño del *software*, implementado en Go, permite que el sistema escale horizontalmente sin necesidad de hacer cambios al código, ya que automáticamente utiliza todos los recursos que se le asignan a él, distribuyendo el trabajo de todas las *goroutines* de manera eficiente a entre todos los *cores* disponibles en el procesador. Esto representa una mejora significativa al compararlo con la solución original, que utilizaba solo uno de los cores del procesador; y también respecto de la solución intermedia, que utiliza un número de threads definidos en el código del programa, además de no utilizar de forma eficiente todos los *cores* asignados a éste.

El rendimiento del sistema también fue mejorado por el nuevo *software*, que además de poseer las ventajas de un lenguaje compilado, versus la implementación anterior en un lenguaje interpretado, ya no sufre de problemas de alcanzar el 100 % de uso de CPU en ninguno de sus *cores*, incluso al hacerlo funcionar con 10 veces la carga original. Esto efectivamente resultó en una mejora en la calidad de servicio, ya que se disminuyeron los tiempos de respuesta hacia los clientes y tampoco quedan clientes sin atender.

Se optimizaron además las consultas realizadas a la base de datos en el sistema BaaS, logrando eliminar la pérdida de información para el número de clientes original. Si bien no es posible atender la cantidad de clientes actual[1] con el límite original de 100 consultas/segundo, este trabajo nos permitió atender un 19 % más de clientes que antes de la optimización, con el mismo límite de consultas.

En general los resultados obtenidos son satisfactorios. Se logró eliminar los problemas de diseño y rendimiento encontrados en el software inicial, además de aumentar la cantidad de clientes atendidos con los mismos recursos. Si bien, aumentaron los costos de mantención del

[1]Límite de 300 consultas/segundo. 2500 clientes promedio, *peaks* de 4000 clientes, a la fecha 17/12/2018.

sistema BaaS, este aumento no fue tan significativo como lo hubiera sido sin las optimizaciones realizadas.

Como trabajo a futuro se contempla escalar horizontalmente el sistema a través del uso de contenedores y un orquestador (por ejemplo, Kubernetes[2]), con la finalidad de introducir redundancia en el sistema. Para esto además será necesario introducir un balanceador de cargas con una base de datos en memoria (por ejemplo, Redis[3]) que se encargue de rutear de forma correcta las solicitudes HTTPS a la instancia del servidor que tenga el socket del reloj correspondiente.

[2]https://kubernetes.io
[3]https://redis.io

Bibliografía

[1] André B. Bondi. "Characteristics of Scalability and Their Impact on Performance". En: *Proceedings of the 2nd International Workshop on Software and Performance* (2000).

[2] Rance Cleaveland y Scott A. Smolka. "Strategic Directions in Concurrency Research". En: *ACM Computing Surveys* 28.4 (1996).

[3] C. A. R. Hoare. "Communicating Sequential Processes". En: *Commun. ACM* (1978).

[4] M. Michael y col. "Scale-up x Scale-out: A Case Study using Nutch/Lucene". En: *2007 IEEE International Parallel and Distributed Processing Symposium* (2007).

[5] Rob Pike. *Go at Google: Language Design in the Service of Software Engineering*. 2012. URL: `https://talks.golang.org/2012/splash.article`.

[6] M. M. Rahman. "Process synchronization in multiprocessor and multi-core processor." En: (2012).

[7] Linda H. Rosenberg y Lawrence E. Hyatt. "Software Re-engineering". En: *Software Assurance Technology Center* (1996).

[8] Abraham Silberschatz, Peter B. Galvin y Greg Gagne. *Operating System Concepts*. 9th. Wiley, 2012.

[9] Peiyi Tang. "Multi-Core Parallel Programming in Go". En: (2010).

.

Apéndices

Apéndice A

Protocolo de Comunicación de SoyMomo

Comando	**Descripción**
TKQ, TKQ2	Solicita los comandos pendientes.
LK	Envía el conteo de pasos y el porcentaje de batería del reloj.
UD, UD2, UD3	Envían la ubicación del reloj.
AL	Envía una alarma, puede ser por batería baja, uso del botón SOS o por el sensor que indica si el niño se quitó el reloj.
GSMANT	Confirmación de que recibió una modificación al parámetro de contestado automático.
LZ	Confirmación de que recibió una modificación al idioma o a la zona horaria.
CR	Confirmación de que recibió una solicitud de actualización de posición.
MESSAGE	Confirmación de que recibió un mensaje de texto.
PROFILE	Confirmación de que recibió una modificación al parámetro de sonido o vibración.
POWEROFF	Confirmación de que recibió un comando de apagar.
REMIND	Confirmación de que recibió una modificación a las alarmas guardadas en el reloj.
SILENCETIME	Confirmación de que recibió una modificación a los horarios de bloqueo en el reloj.
SOS, WHITELIST1, WHITELIST2, PHB, PHB2	Confirmación de que recibió alguna modificación en el listado de contactos del reloj, puede ser contacto de emergencia, contactos permitidos de salida o contactos permitidos de entrada.
FIND	Confirmación de que recibió un comando de hacer sonar.
UPLOAD	Confirmación de que recibió un comando para modificar la frecuencia del GPS.
FLOWER	Confirmación de que recibió un corazón (sistema de recompensas).
TK	Envía un mensaje de voz o confirmación de que recibió un mensaje de voz, dependiendo del parámetro que lo acompaña.
PEDO, WALKTIME	Confirmación de que recibió una modificación a la configuración del podómetro (contador de pasos).

Apéndice B

Datos Uso de CPU

Hora	%Uso CPU	% Uso CPU1	% Uso CPU2	% Uso CPU3	% Uso CPU4
0	12,5	6,2	10,5	10,6	8
1	14,5	6,8	8,4	10	10,2
2	16,7	6,6	12,6	10,3	11,4
3	34,4	9,6	13,6	12,8	13,4
4	40	10,6	11,4	12,3	12,6
5	47,9	12,2	17,8	13	13,7
6	99,8	10,5	13,7	17,2	15,3
7	101,2	8,6	19	22,1	22,8
8	101,6	9,4	20	18,1	23
9	101,6	11,8	17,2	12,9	13,8
10	99,6	12,4	12,4	13	17,8
11	99,8	9	12	15	12,6
12	100,4	10,8	17,2	17,3	12,3
13	100,4	7,9	15,6	12,9	17
14	101,5	9,9	15,3	14,6	13,7
15	101,8	10,7	12,1	14,8	13,1
16	99,5	9,5	12,9	16,5	14,1
17	101	9,1	19,7	23	19,3
18	100,3	8,9	16,6	15,3	16,1
19	100,5	6,8	14,3	13,9	15,4
20	101,4	5,8	13	12	11

Hora	%Uso CPU	% Uso CPU1	% Uso CPU2	% Uso CPU3	% Uso CPU4
21	100,8	8,2	15,2	12,3	10,8
22	102	5,1	10,5	13,5	13,9
23	101,7	6,8	13,8	13,7	13,9
24	12,7	5,1	8	9,3	8,6
25	15,9	8,3	8,5	9,6	8,8
26	15,4	7,3	10,3	11,4	11,1
27	16,8	6,4	15	10,5	10,4
28	36,1	11,3	15,5	13,9	12,1
29	52,2	10,8	14,9	14,1	13,8
30	81,8	12,2	12,9	16,5	18
31	94,1	10	22,5	18,2	20,5
32	101,5	10	20,3	22,9	21,5
33	101,1	7,5	15,4	16,8	14,5
34	101,5	12	17	12,3	16,7
35	100,9	11,9	17,5	16,9	16,2
36	101,7	8,8	15,5	12,8	16,1
37	101,2	8,6	17,4	12,4	12,9
38	100	11,2	16,1	15,7	17,4
39	101,6	11,7	13	17,5	14,2
40	101,8	10,3	15,8	14,1	12,6
41	99,5	11,6	20,5	17,1	18,6
42	101,5	8,7	12,5	15,5	14,5
43	99,6	11,1	17,2	12,5	14,4
44	100,9	9,6	12,8	13,6	12,6
45	101,5	6,2	12,7	15,2	13,4
46	100,3	5,5	14,2	11,3	10,7
47	99,9	3,4	9,6	9,8	14
48	12,6	5,2	11	11,5	8,7
49	15	8	14	12,2	9,1
50	15,4	8,4	11,1	11	15
51	17,8	6,4	14,3	13,8	14,1
52	36,7	9,8	13,6	14,3	11,2
53	46,8	10	13	12,4	15,1
54	98,1	11,9	13,5	12,2	16,2
55	101,1	11,7	22,4	17	22,9
56	102	11,3	19,1	19,9	17,9
57	101,5	9,7	14,4	17,2	17,1
58	99,9	11,2	15,1	13	16,5
59	101,6	7,5	18	14,8	17,2
60	100,3	9,3	17	15,4	14,9

Hora	%Uso CPU	% Uso CPU1	% Uso CPU2	% Uso CPU3	% Uso CPU4
61	101,5	12,4	12,6	12,9	14
62	99,9	11,7	12,1	17,3	13,2
63	101,9	12,4	17,3	13,8	17,3
64	101,9	11,6	17,3	13,9	15,5
65	100,5	7,7	23	19,8	21,2
66	99,5	10,1	16,8	16,6	15
67	100,1	7,9	17,3	15,7	12,6
68	101,5	6,4	15,4	12,7	13,3
69	101,6	8,5	10,7	15,6	14,6
70	99,8	4,3	10,7	9,6	12,6
71	100,5	7	13,4	10	12,5
72	13,3	3,6	8,4	10,1	9,4
73	15,6	4	12	13,6	10,2
74	15,9	4,9	9,9	9,9	10,4
75	17,3	7,3	14,5	12,5	14,1
76	18,6	8,8	12	16,3	15,3
77	46,1	7,7	17,5	15,2	13,6
78	98,8	9,5	16,5	12,2	13,7
79	100,9	11,8	20	18,2	20,4
80	101,8	7,8	21,2	20,5	21,9
81	100,8	11,5	14,8	16,1	14,1
82	100,7	11	13,7	17,8	16
83	102	9,7	13,9	17,8	18
84	99,8	10,1	17	15,1	17,7
85	99,7	9,9	15,6	16,2	17,6
86	101,5	10,6	16,3	12,6	14,4
87	100,3	8,1	18	14,7	13,9
88	99,7	11,1	13,7	15,6	16,8
89	101,4	9,9	22,1	19,4	17,1
90	99,6	10,7	16,2	12,6	14,7
91	100,8	10,2	16,5	17	14,8
92	100,9	10,3	11,2	17	11,2
93	101	7,2	10,8	11,4	13,5
94	100,5	4,7	9,9	9,4	12,5
95	101,6	5,7	13,6	12,5	12
96	13,9	2,6	12,7	10,6	12,1
97	14,6	6,9	9,5	13,3	9,3
98	15,6	8,8	9,6	11,9	13,5
99	16,4	9,8	14,3	15,6	12,1
100	17,4	8,9	16,4	16,5	16,1

Hora	%Uso CPU	% Uso CPU1	% Uso CPU2	% Uso CPU3	% Uso CPU4
101	47,1	9,3	13,6	12,1	14,8
102	98	12,5	14,8	14,6	15,2
103	100,8	10,1	20,4	18,2	21,3
104	100,9	8,1	21,7	17,1	22,8
105	100,4	12,1	13,3	16,8	12
106	99,9	8,6	12,6	16,3	13,2
107	101,7	10	13,9	13,9	17,9
108	99,6	8,1	16,5	13,3	13,5
109	101,6	8,1	12,9	14,7	15,7
110	101,3	12,2	16,5	17,4	13,2
111	100,6	8,7	12,6	17,9	16,5
112	100,2	12,4	14,5	15,9	15,5
113	100,2	12,1	17,5	18,7	18
114	100,4	9,1	15,1	17,5	13,4
115	101,3	10,3	12,8	17,7	15,8
116	101,5	8,1	11,6	11,6	13
117	101,8	8,5	12,3	14,7	15,4
118	100,1	3,5	10,9	15	10
119	101,7	4,1	9,5	9,7	11,5
120	13,8	4	9,8	8,6	12,1
121	15,8	4,4	10,9	10,2	10,7
122	16,4	7,2	10,9	10,4	11,2
123	17,8	9,5	10,6	12,5	12,9
124	18,9	7,1	15,1	13,7	12,4
125	46,4	8,7	16,3	17,3	15,4
126	53	8,8	13,3	12,3	15,7
127	51	11,3	17	14,2	16,2
128	50,2	9,5	15,7	13,7	17,3
129	53,3	11,5	12	16,5	16,9
130	50,9	8,2	14,2	16,2	14,4
131	48,8	10,4	17,3	14,6	14,4
132	49,9	9,4	15,9	16,8	13,7
133	51,9	7,8	13,2	13,1	17,4
134	48,3	10,1	15,6	13,7	16,3
135	52,4	7,5	18	17,8	13,4
136	53,6	7,7	15,6	14,4	13,8
137	50,3	9,3	15,6	13,3	16,2
138	52,1	10,1	17,1	15,1	13,3
139	50	11,1	15,9	12,9	12,9
140	52,8	8,7	13,6	16,6	12,6

Hora	%Uso CPU	% Uso CPU1	% Uso CPU2	% Uso CPU3	% Uso CPU4
141	53,6	7,9	11,8	13,8	14,8
142	50,2	4,2	10,6	11	13,5
143	48,7	5	13,6	10,8	10,7
144	13	5,5	7,3	9,6	8,4
145	14,3	4,1	8	10,1	12,5
146	15,4	4,8	10	11,5	9,3
147	16,1	6,4	10,1	11,4	11,7
148	17,8	8,9	14,2	12,2	16,6
149	46,5	9,6	16,7	13,6	13,2
150	50,2	9,5	12,7	15,3	15,3
151	50,9	9,7	16,2	17,6	17,2
152	51,8	7,7	13,6	16,4	12,5
153	61,9	8,1	17,7	15,4	16,9
154	60,6	11,8	15,3	12,8	15,7
155	58,1	11,6	17	17,3	17,6
156	58,7	8,8	15,8	13,2	15,3
157	61,8	11,9	13,6	14	12,1
158	61,7	7,9	17,9	13,1	15,4
159	58,2	7,8	13	17,2	17,6
160	60,8	8,2	17,3	16,5	16,1
161	55,1	9,2	15,2	17,2	15,8
162	53,5	8,1	14,4	17,2	16
163	52,7	10,2	14,6	14,3	12,5
164	48,1	9,2	15,2	14,7	15
165	52,4	5,5	15,5	14,3	14,9
166	51,6	6,6	12,9	13,6	14
167	48,6	3,7	12,3	11,5	12

Apéndice C

Datos Pérdida de Información y Cantidad de Consultas

Hora	% Pérdida Antes	% Pérdida Después	Consultas Antes	Consultas Después
0	0	0	225	199
1	0	0	195	171
2	0	0	126	106
3	0	0	216	185
4	0	0	204	174
5	0	0	267	234
6	2,8	0	392	335
7	3,5	0	743	646
8	9,4	0	772	671
9	5	0	738	642
10	3,4	0	796	686
11	3,8	0	703	558
12	4,7	0	780	566
13	3,9	0	708	595
14	2	0	784	670
15	4,4	0	724	641
16	13,3	0	774	679
17	7,6	0	431	378
18	2,8	0	437	367
19	1,1	0	345	292
20	2,9	0	358	314

Hora	% Pérdida Antes	% Pérdida Después	Consultas Antes	Consultas Después
21	4,5	0	307	269
22	2,6	0	405	340
23	2	0	320	276
24	0	0	277	235
25	0	0	208	181
26	0	0	185	162
27	0	0	189	163
28	0	0	255	220
29	0	0	319	268
30	2,9	0	500	427
31	2	0	745	659
32	10,3	0	763	647
33	5,3	0	743	624
34	3,6	0	692	612
35	2,6	0	771	676
36	3,6	0	791	682
37	2,6	0	768	645
38	3,8	0	780	678
39	4,6	0	676	568
40	12,1	0	763	675
41	7,2	0	399	350
42	2,9	0	473	404
43	1,6	0	315	276
44	2,8	0	396	350
45	3,9	0	317	278
46	2	0	360	310
47	4,7	0	343	288
48	0	0	269	232
49	0	0	247	215
50	0	0	231	194
51	0	0	225	194
52	0	0	230	202
53	0	0	366	324
54	0	0	433	364
55	2,9	0	831	698
56	10,1	0	740	655
57	6,5	0	675	597
58	3,6	0	758	642
59	2,6	0	779	666
60	4,2	0	747	633

Hora	% Pérdida Antes	% Pérdida Después	Consultas Antes	Consultas Después
61	2	0	754	639
62	2,1	0	661	585
63	3,8	0	747	628
64	13,8	0	764	670
65	7	0	445	374
66	3,5	0	518	443
67	2,1	0	381	320
68	2	0	368	309
69	3,3	0	296	262
70	2,2	0	304	267
71	2,4	0	330	287
72	0	0	133	116
73	0	0	238	202
74	0	0	222	195
75	0	0	280	248
76	0	0	278	242
77	0	0	260	222
78	0	0	432	369
79	2,5	0	678	600
80	10,3	0	699	608
81	5,1	0	782	657
82	3,5	0	683	579
83	3,7	0	652	562
84	4,4	0	753	649
85	4	0	748	662
86	3	0	757	636
87	4,7	0	637	564
88	14	0	763	647
89	8,1	0	494	419
90	3,3	0	469	411
91	2,7	0	402	344
92	2	0	378	329
93	3,2	0	382	329
94	3,2	0	286	253
95	1,8	0	373	316
96	0	0	233	201
97	0	0	192	167
98	0	0	162	136
99	0	0	214	186
100	0	0	201	170

Hora	% Pérdida Antes	% Pérdida Después	Consultas Antes	Consultas Después
101	0	0	385	326
102	0	0	432	363
103	3,7	0	729	613
104	9,6	0	708	600
105	6,6	0	691	586
106	4,5	0	752	632
107	2,2	0	812	688
108	4,9	0	826	694
109	2,8	0	723	634
110	2,9	0	741	628
111	5	0	775	686
112	12,4	0	683	589
113	8,5	0	494	437
114	2,1	0	526	442
115	1,4	0	353	304
116	2,3	0	383	336
117	1,6	0	314	271
118	1,2	0	334	283
119	1,6	0	350	299
120	0	0	229	201
121	0	0	200	171
122	0	0	188	165
123	0	0	238	203
124	0	0	160	142
125	0	0	393	342
126	0	0	414	363
127	0	0	728	639
128	0	0	649	574
129	1,5	0	762	651
130	1	0	804	676
131	1,7	0	825	693
132	1,2	0	670	593
133	1	0	688	593
134	1	0	803	686
135	1,7	0	653	553
136	1,9	0	760	655
137	1	0	510	447
138	1,4	0	474	416
139	1,9	0	351	300
140	1,9	0	308	268

Hora	% Pérdida Antes	% Pérdida Después	Consultas Antes	Consultas Después
141	1,8	0	355	309
142	1,4	0	315	279
143	1,1	0	335	294
144	0	0	128	109
145	0	0	135	118
146	0	0	262	226
147	0	0	201	173
148	0	0	280	248
149	0	0	381	326
150	0	0	363	316
151	0	0	633	550
152	0	0	768	656
153	1,4	0	740	627
154	1,3	0	765	648
155	1,2	0	750	664
156	1,6	0	673	575
157	1,1	0	740	649
158	1,1	0	683	594
159	1,7	0	797	699
160	1,4	0	715	627
161	1,4	0	453	397
162	1,6	0	447	396
163	2	0	331	278
164	1,5	0	322	273
165	1,9	0	355	309
166	2	0	348	308
167	1,9	0	398	340

Apéndice D

Datos tiempo respuesta, lectura y escritura

Hora	Respuesta A	Escritura A	Lectura A	Respuesta D	Escritura D	Lectura D
0	308	449	448	326	440	439
1	357	507	464	308	497	459
2	333	533	496	327	522	487
3	326	572	520	324	566	508
4	351	440	425	304	427	412
5	343	547	501	309	533	495
6	448	726	704	314	717	699
7	570	659	620	302	650	609
8	599	643	617	337	633	609
9	495	722	713	293	717	706
10	498	629	569	306	618	564
11	590	756	728	308	745	717
12	484	793	764	298	786	751
13	592	719	700	338	709	695
14	524	797	760	309	792	748
15	558	643	588	307	634	575
16	497	747	694	305	732	686
17	481	648	632	275	634	624
18	550	701	659	306	695	647
19	593	791	782	291	782	770
20	518	777	729	283	771	723

A: Antes, D: Después, datos en [ms]

Hora	Respuesta A	Escritura A	Lectura A	Respuesta D	Escritura D	Lectura D
21	558	718	699	335	711	690
22	497	694	646	283	682	635
23	482	740	708	327	731	703
24	384	422	370	320	410	361
25	362	501	490	321	489	477
26	326	610	606	326	599	596
27	374	439	440	295	433	426
28	389	533	513	331	525	498
29	303	521	503	278	506	498
30	496	630	569	310	616	556
31	541	738	736	309	723	723
32	515	752	736	303	743	730
33	499	786	776	279	771	769
34	562	621	615	307	615	609
35	513	631	623	311	625	614
36	521	702	694	306	691	685
37	569	752	742	336	739	737
38	572	802	775	320	790	763
39	514	753	740	278	741	727
40	593	784	777	321	769	766
41	537	759	705	327	748	697
42	544	649	634	320	638	625
43	495	621	606	321	609	591
44	533	625	588	324	617	573
45	485	758	711	294	743	700
46	576	674	672	314	662	659
47	539	617	570	320	608	560
48	301	491	489	313	477	474
49	384	569	529	311	560	523
50	288	561	510	293	555	497
51	397	463	411	316	448	402
52	315	562	510	281	550	505
53	386	575	539	285	561	524
54	447	668	615	327	661	603
55	482	716	686	294	707	677
56	528	738	679	290	731	671
57	493	707	685	329	695	670
58	558	740	719	279	725	710
59	487	802	775	307	793	765
60	571	673	665	336	664	652

Hora	Respuesta A	Escritura A	Lectura A	Respuesta D	Escritura D	Lectura D
61	522	780	741	280	765	731
62	558	667	656	315	656	641
63	480	753	708	329	741	702
64	595	789	747	282	783	740
65	519	704	682	336	698	677
66	520	738	696	310	724	690
67	481	789	788	293	784	773
68	509	646	643	328	636	630
69	485	795	779	279	790	772
70	525	641	582	318	627	574
71	581	656	593	281	646	588
72	340	513	474	326	500	466
73	379	459	428	315	452	420
74	330	540	487	282	530	476
75	352	543	499	292	528	492
76	381	476	465	312	465	458
77	310	604	581	324	594	567
78	433	770	726	314	762	717
79	485	662	631	296	648	622
80	539	755	755	316	743	740
81	513	765	740	302	750	727
82	541	646	589	278	633	578
83	537	751	745	317	738	738
84	585	723	682	300	709	670
85	485	791	770	336	784	760
86	535	746	685	325	732	679
87	577	762	721	288	752	708
88	585	629	597	339	621	585
89	587	719	719	322	713	708
90	513	671	620	321	658	615
91	595	800	776	313	793	767
92	508	792	741	291	777	730
93	596	751	713	291	745	700
94	583	768	710	292	757	698
95	559	808	799	278	797	790
96	302	411	355	288	403	344
97	293	420	401	282	408	393
98	362	506	457	292	501	452
99	366	472	412	319	457	399
100	328	458	440	283	450	432

Hora	Respuesta A	Escritura A	Lectura A	Respuesta D	Escritura D	Lectura D
101	332	481	471	282	474	464
102	473	660	655	306	649	641
103	541	666	645	293	656	632
104	523	667	624	311	656	613
105	535	640	635	337	627	620
106	518	676	655	310	670	642
107	481	668	611	296	656	599
108	502	694	661	302	685	647
109	537	722	714	332	715	699
110	483	724	686	337	713	674
111	495	629	622	324	621	612
112	494	768	750	288	753	745
113	581	666	639	300	652	627
114	586	696	681	291	689	673
115	537	626	610	283	617	600
116	540	663	608	276	654	602
117	584	618	590	333	613	581
118	498	756	699	334	747	692
119	585	720	676	281	711	671
120	396	580	555	283	572	542
121	287	542	514	296	535	508
122	359	442	415	299	430	404
123	361	610	576	324	596	566
124	324	428	405	323	413	392
125	439	510	491	309	497	480
126	494	555	529	314	550	516
127	490	450	420	281	445	414
128	409	542	487	313	530	475
129	472	531	538	334	524	524
130	397	570	519	322	556	513
131	471	539	523	323	524	514
132	383	530	523	308	521	510
133	381	445	429	324	436	414
134	448	602	561	316	588	552
135	400	539	505	334	530	495
136	417	528	514	316	515	505
137	385	588	572	299	583	562
138	425	605	599	285	591	591
139	400	581	572	320	576	564
140	389	499	487	320	487	474

Hora	Respuesta A	Escritura A	Lectura A	Respuesta D	Escritura D	Lectura D
141	490	434	379	301	421	364
142	415	452	419	298	440	414
143	398	594	581	305	588	567
144	352	474	436	281	469	427
145	362	518	477	332	509	472
146	320	438	390	288	433	378
147	393	481	444	280	473	434
148	328	543	524	288	535	515
149	470	576	569	309	571	564
150	486	541	517	334	533	507
151	485	479	462	281	464	450
152	426	588	552	276	575	545
153	403	498	481	331	484	473
154	451	545	524	338	534	519
155	432	501	453	298	496	442
156	426	426	399	330	421	391
157	400	507	463	308	497	457
158	496	469	463	297	455	452
159	385	468	428	309	456	414
160	433	474	476	334	469	463
161	471	518	506	298	504	491
162	427	538	531	296	527	516
163	478	459	444	317	446	429
164	394	519	484	330	507	472
165	466	410	374	291	405	363
166	401	549	508	300	536	498
167	454	518	483	316	504	468