Taller de programación III (75.61), FIUBA

TP1: Contador de visitas y pruebas de carga
Informe de pruebas de carga

Adrián Barreal

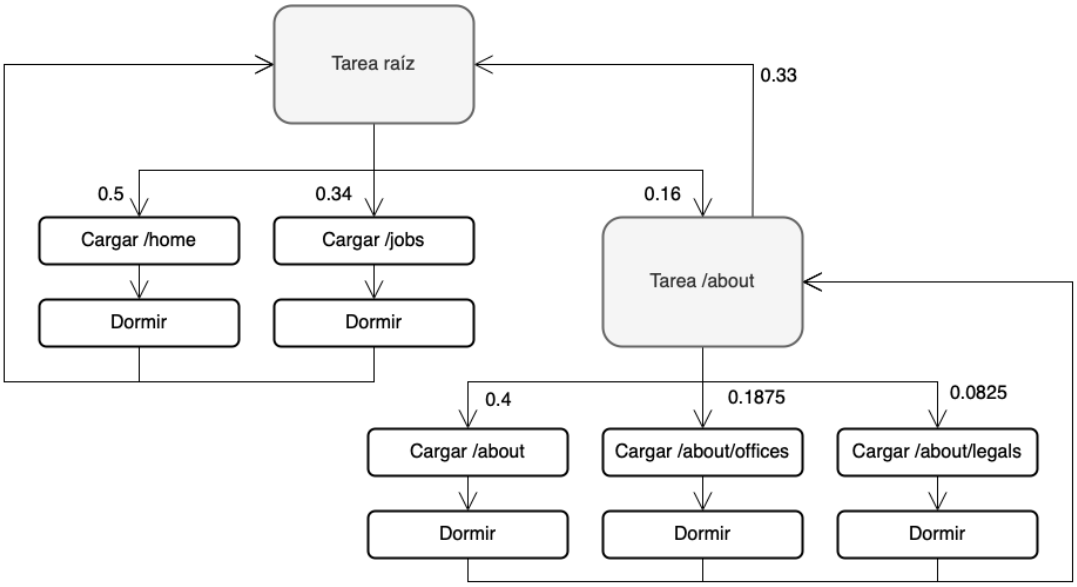

Figura 1: Diagrama de estados que describe el comportamiento del usuario modelo. Los números representan probabilidad de transición.

1. Introducción

Este documento detalla las pruebas de carga desarrolladas sobre el sitio web institucional cuya estructura fue detallada en el documento de diseño adjunto. El modelo del usuario para las pruebas de carga y la estructura general de las mismas se detalla en la sección 2. Se ejecutó una prueba inicial (sección 3) para determinar las capacidades del sistema en su versión original, con los parámetros propuestos en el documento de diseño. En base a los resultados obtenidos se propusieron mejoras para aumentar el rendimiento del sistema, comentadas en la misma sección. Luego se ejecutó una segunda prueba para evaluar la efectividad de las mejoras, obteniendo resultados positivos (sección 4). Algunas conclusiones generales se presentan en la sección 5.

2. Diseño y ejecución de las pruebas

Las pruebas de carga fueron desarrolladas usando Locust[1], un framework para Python diseñado específicamente para la tarea. Cada instancia de prueba simula una cantidad creciente en el tiempo de usuarios concurrentes navegando el sitio web en forma simultánea. El modelo de comportamiento de un usuario se implementó como un script para Locust y se describe como un diagrama de estado en la figura 1. Los usuarios comienzan en el estado inicial "tarea raíz" y pueden cambiar de estado siguiendo las aristas del grafo. Los números o pesos que acompañan las aristas representan la probabilidad de

[1] `https://locust.io`

transición. Una arista sin peso tiene probabilidad 1 de ejecutarse. Además de las tareas instantáneas "tarea raíz" y "tarea `/about`" que son solo conceptuales y no están respaldadas por interacción concreta con la aplicación, hay otros dos tipos de tarea: cargar y dormir, que se describen a continuación:

- Cargar recurso: Se accede por HTTP al recurso indicado, descargando el contenido HTML. Para emular más adecuadamente el comportamiento del browser, si es la primera vez que el usuario accede a la ruta se procede adicionalmente a extraer y a solicitar todos los recursos estáticos listados en el HTML: imágenes, archivos CSS y archivos JavaScript.
- Dormir. Se emula el comportamiento de un usuario que pasa una cantidad aleatoria de tiempo en la página a la que accedió, por ejemplo leyendo. Cada página tiene configurados períodos distintos de inactividad (e.g. aquellos pocos usuarios que accedan a la sección de legales podrían pasar una cantidad de tiempo relativamente elevada leyendo el contenido de esa página).

Las probabilidades de transición fueron definidas en forma más o menos arbitraria y deberían idealmente surgir de observaciones reales en los datos.

3. Prueba inicial

Los principales parámetros que impactan en el rendimiento del sistema, con sus valores iniciales propuestos en el documento de diseño, son los siguientes:

- Cantidad máxima de instancias del contenedor del sitio web: 2.
- Cantidad máxima de instancias de la API: 2.
- Cantidad máxima de instancias del contador de visitas: 2.
- Cantidad de shards por contador: 100.
- Todas las instancias fueron provistas de 1 CPU y 1 GB de RAM.

Esta primero prueba evalúa el rendimiento del sistema en estas condiciones iniciales.

3.1. Resultados experimentales

El gráfico generado por Locust de la cantidad de usuarios en función del tiempo se presenta en la figura 2. El gráfico de la cantidad de pedidos por segundo que generan estos usuarios, así como la cantidad de pedidos que resultan en errores, se muestra en la figura 3. Se observa que, llegada una cierta cantidad de pedidos por segundo, la aplicación comienza a retornar errores. En la figura 4 puede observarse también un deterioro significativo en el tiempo de respuesta, llegando eventualmente a que al menos el 95 % de los pedidos realizados resultan en timeout.

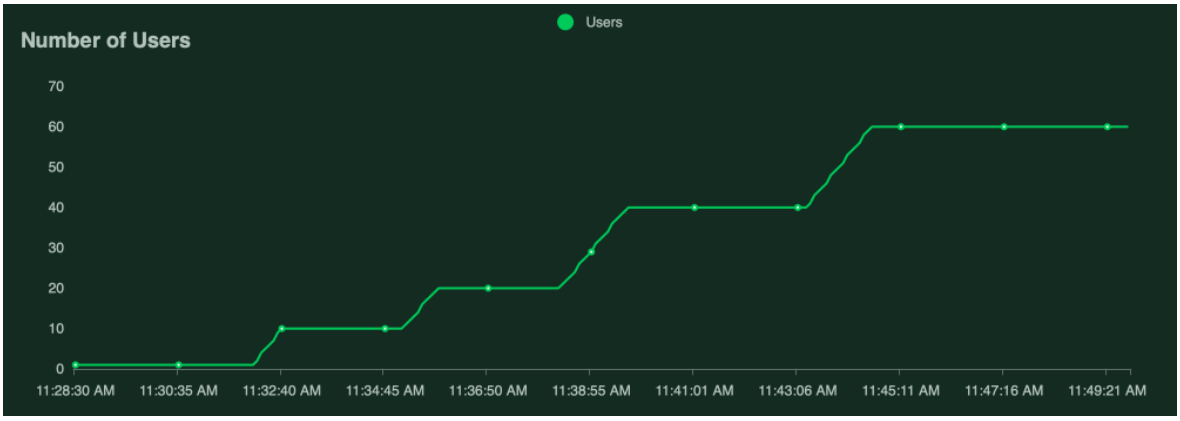

Figura 2: Prueba #1, cantidad de usuarios en función del tiempo.

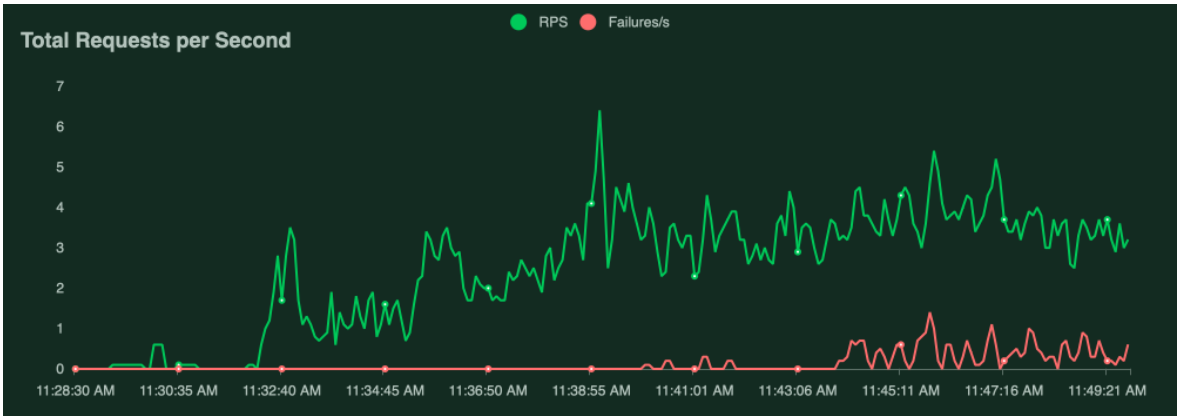

Figura 3: Prueba #1, pedidos por segundo y errores por segundo.

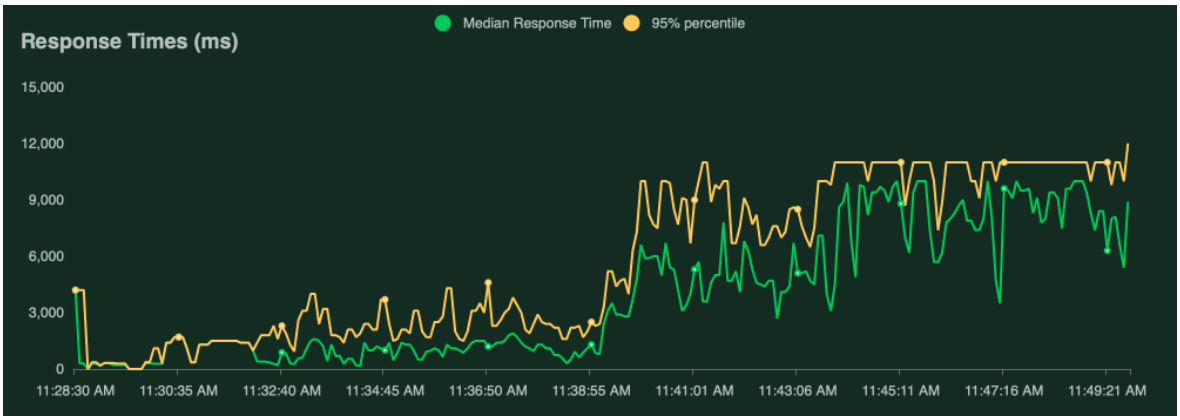

Figura 4: Prueba #1, tiempo de respuesta del servidor.

El progreso irregular en la figura 2 se debe a que la intención inicial era duplicar la cantidad de usuarios en cada paso; al observar el rendimiento deteriorado con 40 usuarios, no obstante, se decidió reducir la diferencia hasta la siguiente meseta.

3.2. Análisis de métricas y resultados

En los gráficos de las figuras 2, 3, 4 se distinguen vagamente tres regiones: una región de operación normal (con aproximadamente 20 usuarios concurrentes), una región de operación con respuesta notablemente deteriorada pero con una tasa de errores relativamente baja (con aproximadamente 40 usuarios concurrentes) y una región en la que la respuesta del sistema se ve significativamente deteriorada al punto en que hay una cantidad significativa de fallos por segundo por causa de timeout.

Los errores observados se tratan exclusivamente de errores 429. Este es un error que Cloud Run retorna cuando no hay suficientes instancias disponibles para gestionar un pedido dado. Esto quiere decir que en algún punto la aplicación web de cara al usuario no tiene recursos computacionales suficientes para procesar el pedido entrante. En la figura 5 se observa efectivamente que el consumo de CPU de dicho container comienza a subir hasta llegar a 100 % en visible concordancia con el aumento en el tiempo de respuesta. Nótese que la carga en la API permanece relativamente baja a lo largo de todo el experimento.

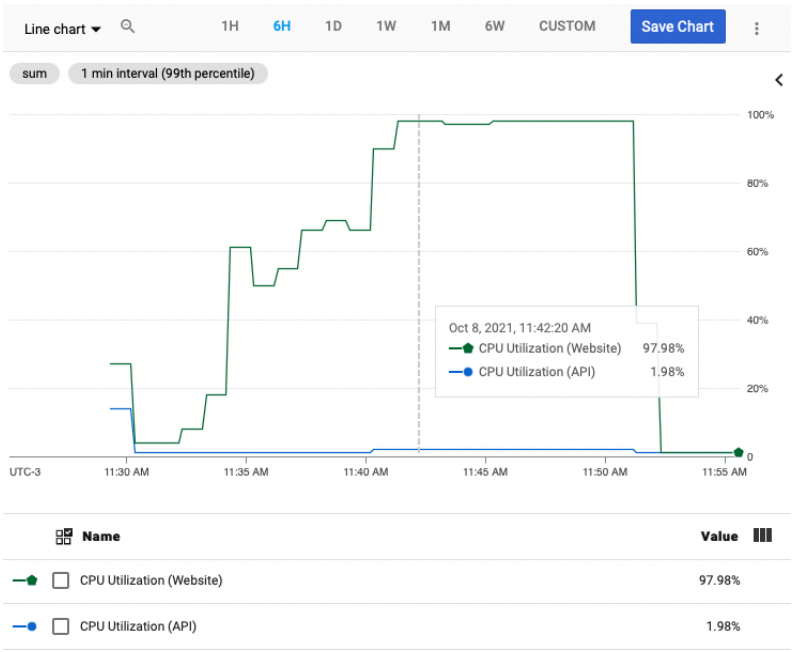

Figura 5: Prueba #1, consumpo de CPU en Cloud Run.

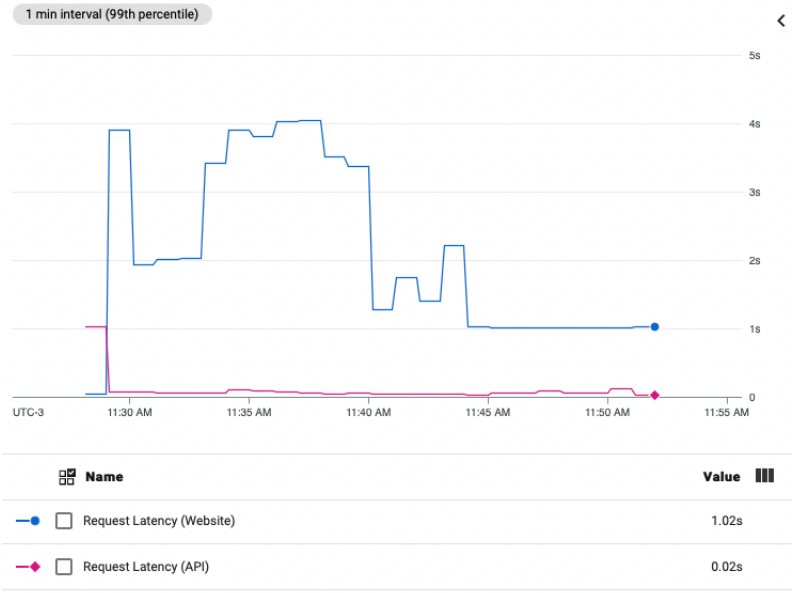

Figura 6: Prueba #1, tiempo de respuesta de Cloud Run.

Los gráficos del tiempo de respuesta del contenedor de cara al usuario y de la API se detallan en la figura 6. Lo importante a observar en este gráfico es lo relativamente poco que tarda la API en responder. Esto indica que el cuello de botella no está en la API.

***Nota**: Para el trabajo práctico se incorporó un busy wait al código de algunos controladores del sitio web para simular tareas de carga y procesamiento de datos de varias centenas de milisegundos. Un modelo más adecuado hubiese sido utilizar un sleep bloqueante ya que simularía más adecuadamente el comportamiento de un servidor web que se queda bloqueado esperando la respuesta de, por ejemplo, una base de datos. El busy wait, por otro lado, ocupa la CPU por fracciones importantes de segundo por cada pedido, lo que limita significativamente la cantidad de pedidos por unidad de tiempo que pueden ser atendidos. Dado el busy wait como se implementó, no debería ser posible atender más que unos pocos pedidos por segundo por instancia.*

Se observa por otro lado, en las figuras 7 y 8, que el contador de visitas y Firestore no tienen dificultad gestionando la cantidad de pedidos entrantes. En la primera se observa que los mensajes entrantes a Pub/Sub son inmediatamente despachados y rápidamente confirmados, sin ningún tipo de acumulación. En la segunda se observa que la cantidad de escrituras por segundo a Firestore no llegan a ser 3; dada la cantidad de shards por contador en uso (100), la probabilidad de colisión en la selección del shard en un segundo dado es diminuta.

Figura 7: Prueba #1, mensajes recibidos, despachados y confirmados en Pub/Sub.

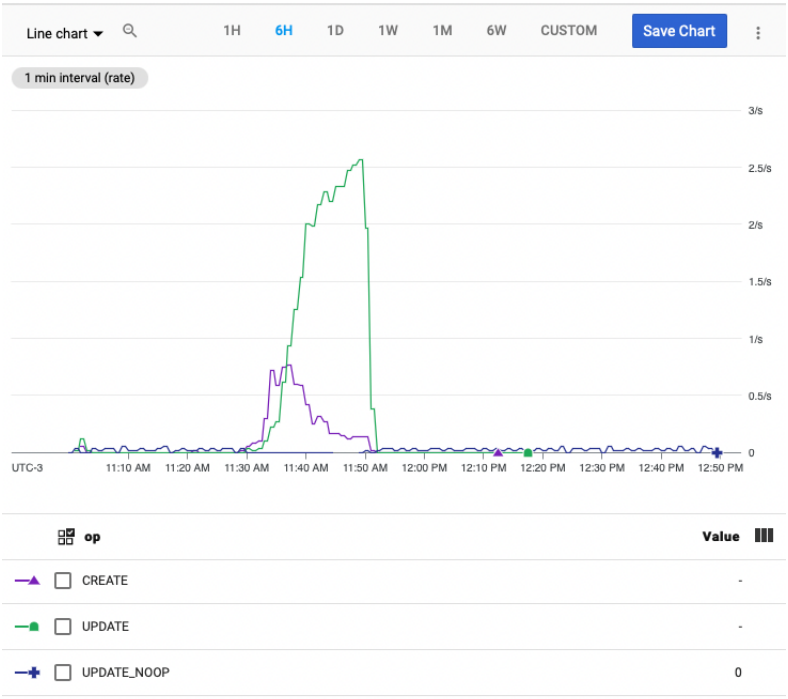

Figura 8: Prueba #1, escrituras por segundo a Firestore.

3.3. Mejoras propuestas

De las observaciones previas se concluye que para mejorar la capacidad del sistema de tolerar carga es necesario poner foco en mejorar la capacidad del servidor web de cara al usuario. Las mejoras propuestas son las siguientes:

1. Optimizar el código del contenedor que sirve el contenido web. Esta alternativa no implica costos por mes adicionales de infraestructura.
2. Aumentar la cantidad de instancias del contenedor que sirve el contenido web. Esta opción tendrá un costo económico asociado más allá del costo de desarrollo.
3. Delegar la responsabilidad de servir archivos estáticos a un servicio especializado (e.g. nginx, Google Cloud Storage). Los pedidos por archivos estáticos iniciados por el browser podrían llegar prácticamente en simultáneo, lo cual presentaría un problema observadas las limitaciones de throughput que afectan al servidor web.
4. Disminuir la cantidad máxima de instancias de la API, vista la gran diferencia de consumo de procesador entre los servicios. Se observó en las métricas de Cloud Run que incluso cuando una es suficiente para atender la totalidad de los pedidos en forma eficiente hay una fracción importante del tiempo en la que hay una instancia en estado “idle” que fue levantada preventivamente y rápidamente determinada innecesaria por Cloud Run.

No se propone disminuir la cantidad de shards por contador ni de instancias del contador de visitas sin antes evaluar qué impacto tendrán las mejoras propuestas en la carga sobre el sub-sistema de conteo de visitas.

4. Prueba #2

Habiendo implementado las mejoras propuestas en la sección 3.3 se procedió a ejecutar las pruebas de carga nuevamente, obteniendo mejores resultados esta vez. Los parámetros del sistema fueron en esta instancia los siguientes:

- Cantidad máxima de instancias del contenedor del sitio web: 6 (tres veces más que antes).
- Cantidad máxima de instancias de la API: 1 (una instancia menos que antes).
- Cantidad máxima de instancias del contador de visitas: 2 (sin cambios).
- Cantidad de shards por contador: 100 (sin cambios).
- Todas las instancias fueron provistas de 1 CPU y 1 GB de RAM (sin cambios).
- Si bien no se incorporó un proxy reverso para delegar el servicio de archivos estáticos, se simuló la característica eliminando los pedidos del script de Locust.

***Nota**: Para simular mejoras en el código del contenedor web se redujo el busy wait a la mitad y se agregó en cambio un sleep bloqueante por el total de la cantidad reducida. Si bien el tiempo de respuesta para un pedido individual debería ser el mismo, el hecho de que el sleep sea bloqueante debería tener un impacto positivo en el throughput. En un contexto general esto simularía, por ejemplo, la delegación de cómputo a una base de datos bien provisionada mediante queries más complejos. Esta mejora se evaluó individualmente en una prueba adicional no documentada y se comprobó que el sistema pasa a ser capaz de tolerar 40 usuarios sin deterioro de rendimiento como ocurría inicialmente.*

4.1. Resultados experimentales

Gráficos análogos a los presentados en la sección 3.1 se presentan en las figuras 9, 10, 11. Se observa que la cantidad de pedidos por segundo llega aproximadamente a 12 antes de empezar a observarse errores y tiempos de respuesta deteriorados: entre 3 y 4 veces lo que se obtuvo originalmente.

4.2. Análisis de métricas y resultados

Se observa que la mediana del tiempo de respuesta permanece aproximadamente constante en apariencia, lo cual podría indicar que el tiempo de respuesta ante pedidos individuales no se degrada y la limitación sigue estando en el throughput debido a que

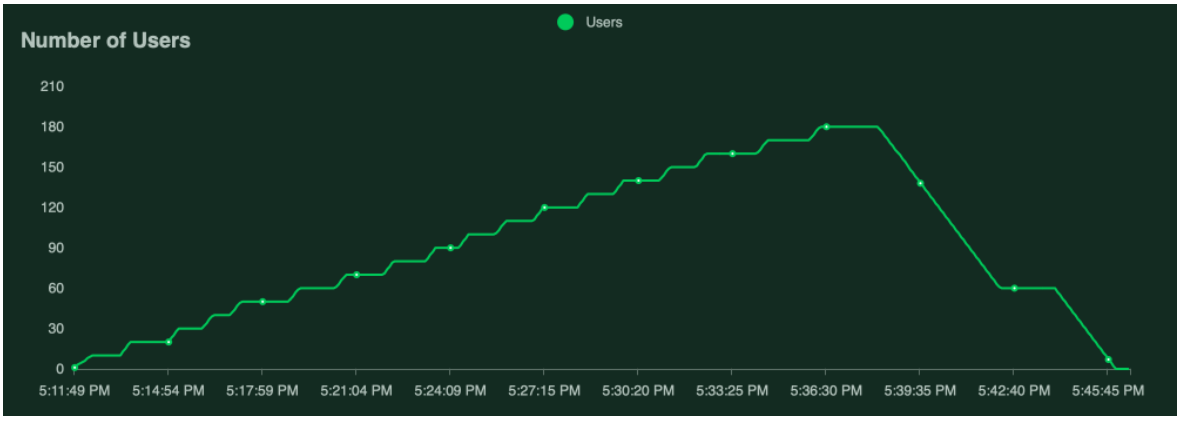

Figura 9: Prueba #2, cantidad de usuarios en función del tiempo.

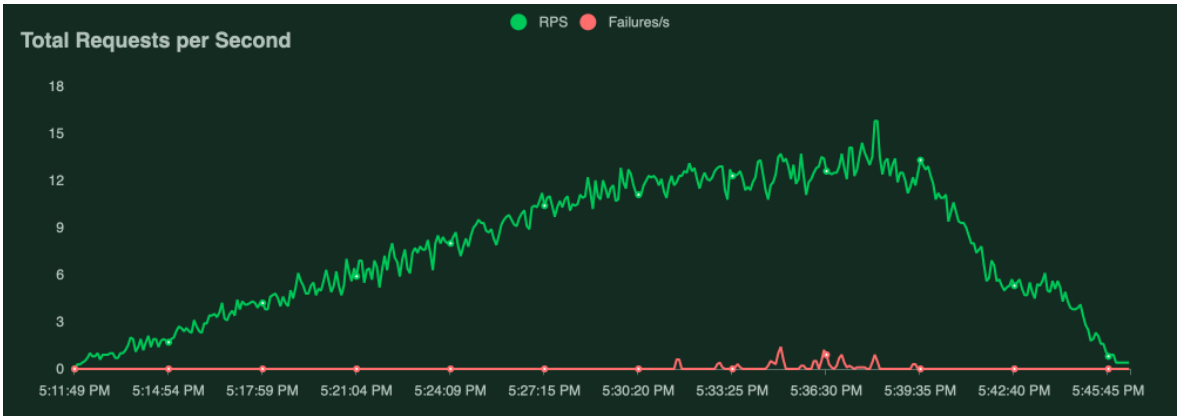

Figura 10: Prueba #2, pedidos por segundo y errores por segundo.

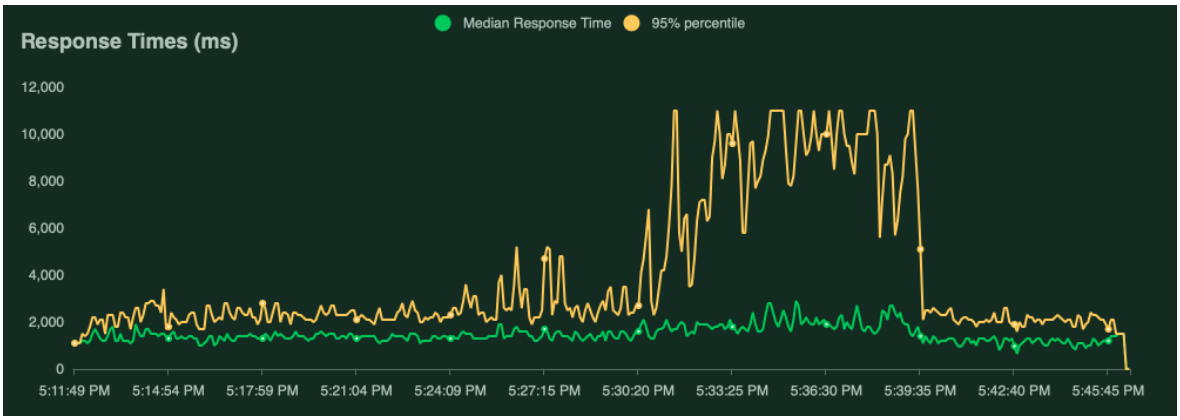

Figura 11: Prueba #2, tiempo de respuesta del servidor.

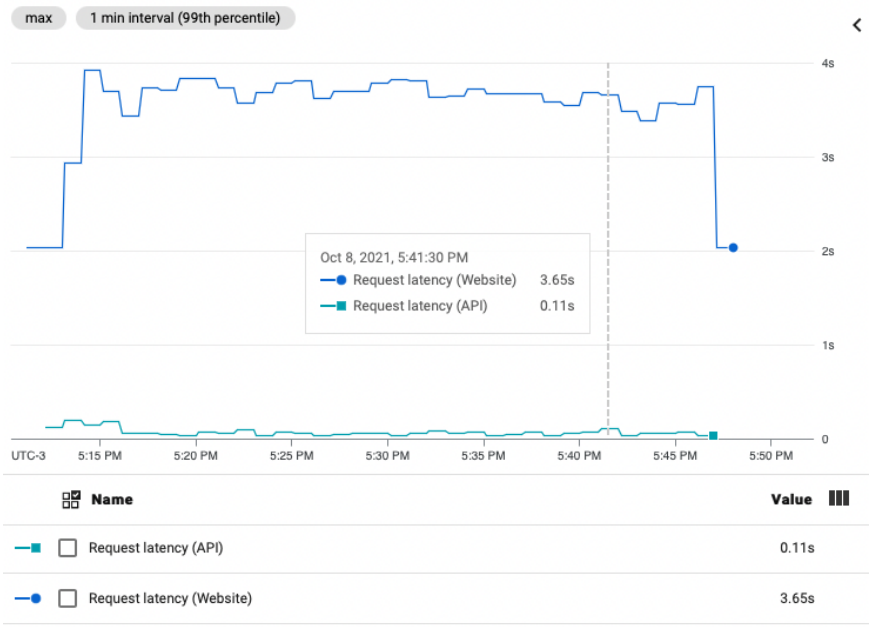

Figura 12: Prueba #2, tiempo de respuesta registrado por Cloud Run.

no hay instancias suficientes para atender la cantidad de pedidos entrantes. Estos pedidos quedan posiblemente acolados en Cloud Run hasta que eventualmente hay instancias libres para procesarlos. Aquellos pedidos que no llegan a ser atendidos antes del tiempo de timeout son descartados por Cloud Run.

Efectivamente, en la figura 12 se observa que según Cloud Run el tiempo máximo de respuesta del servidor web de cara al usuario no llega a superar los 4 segundos en ningún momento durante el experimento. Se sigue observando también que la API responde en forma prácticamente inmediata. Visto que la aplicación no realiza ningún otro pedido a servicio que pudiera convertirse en un cuello de botella (despreciando el tiempo envío de mensajes a Pub/Sub), la limitación actual parecería seguir estando en el tiempo de procesador consumido atendiendo cada pedido individual. Nuevamente, si se deseara seguir mejorando el throughput habría que aumentar la cantidad de instancias en el contenedor del sitio web, u optimizar el algoritmo aún más para hacer uso más eficiente del tiempo de CPU con el que se cuenta.

Resta determinar cómo el aumento en el throughput impacta al sub-sistema de conteo de visitas. La figura 13 muestra lo que se ve desde el punto de entrada en Pub/Sub. Se observa que el contador tal como está configurado es todavía capaz de atender todos los pedidos entrantes sin acumulación de mensajes. En la figura 14 se observa que la cantidad de escrituras por segundo a Firestore, si bien aumentó, sigue siendo pequeña en relación a la cantidad de shards por contador.

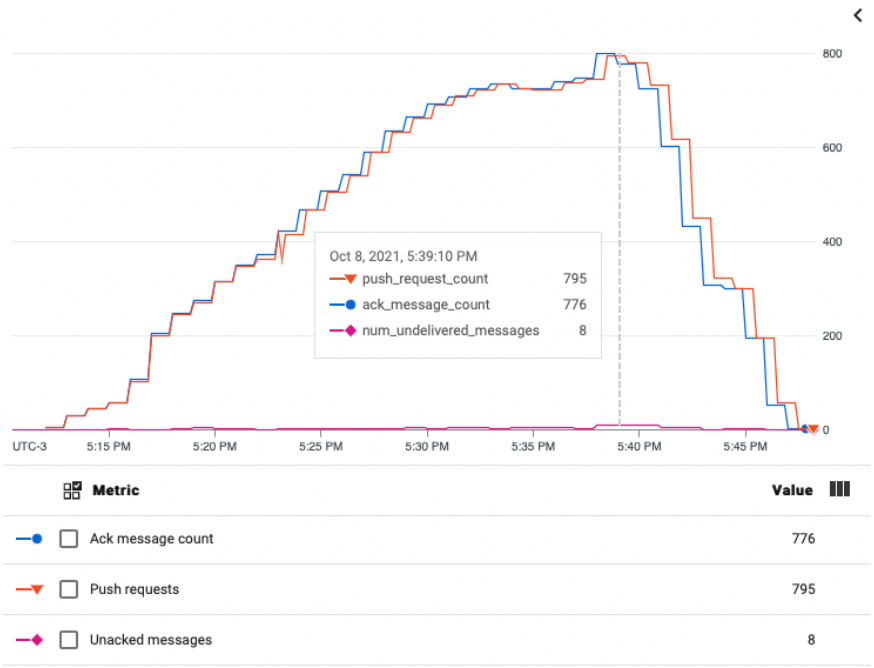

Figura 13: Prueba #2, mensajes recibidos, despachados y confirmados en Pub/Sub.

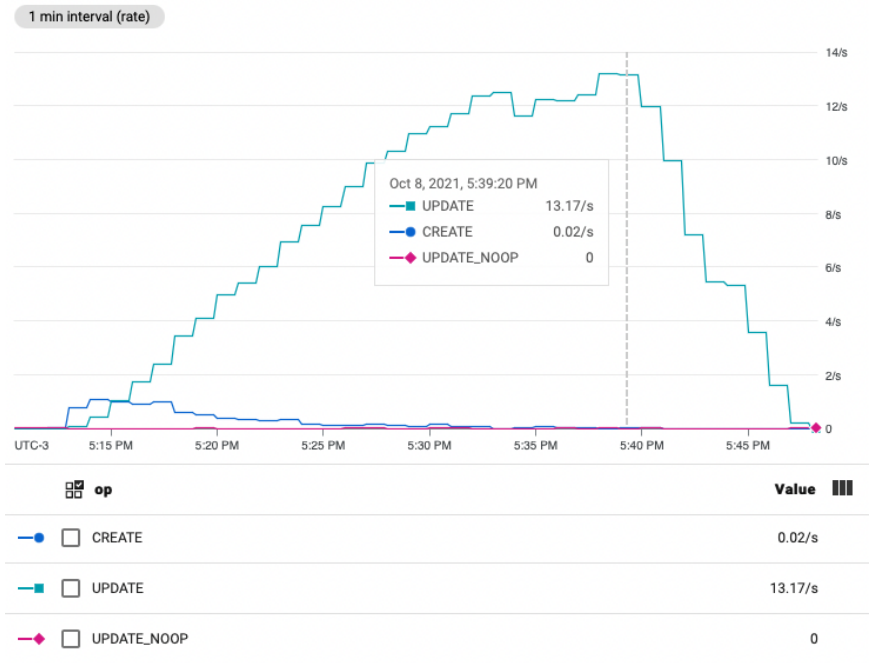

Figura 14: Prueba #2, escrituras por segundo a Firestore.

5. Conclusiones

Este experimento permite observar claramente la diferencia entre tiempo de respuesta y throughput. Independientemente del tiempo que se tarde en responder a un pedido individual, si la mayor parte de ese tiempo el proceso permanece bloqueado esperando la respuesta de servicios de back-end la instancia tendrá ciclos de procesador disponibles para atender a una mayor cantidad de clientes en simultáneo. De esto se desprende la importancia de utilizar algoritmos óptimos, de cachear adecuadamente para evitar procesamiento innecesario, de despachar tareas asíncronas a través de colas si es admisible procesar datos en forma diferida, y de evitar repetir tareas de procesamiento si el resultado puede ser compartido, entre otras optimizaciones.

A posteriori se observa que los resultados iniciales podrían haber sido predichos con moderada certeza en forma teórica. Si se conoce el tiempo de procesamiento promedio para cada endpoint, y se conoce el porcentaje de pedidos para cada endpoint, se puede calcular una media ponderada del tiempo de procesamiento por pedido. Razonablemente, si el tiempo promedio de procesamiento por pedido es T milisegundos, se puede estimar la cantidad de pedidos por segundo que puede procesar una instancia con un solo CPU como $1000/T$.