

**FACULTAD DE INGENIERÍA  
Y ARQUITECTURA**

**DESARROLLO DE UNA LIBRERÍA DE SIMULACIÓN POR EVENTOS DISCRETOS EN LENGUAJE RUST BASADA EN EL USO DE GENERADORES PARA SIMULACIONES ORIENTADAS A PROCESOS**

Trabajo de memoria para obtener el título de:

**INGENIERO CIVIL EN COMPUTACIÓN E INFORMÁTICA.**

|  |  |  |
| --- | --- | --- |
|  | Alumno: | Kevin Frez Parra |
|  | Profesor Patrocinante: | Mauricio Oyarzun Silva |

IQUIQUE – CHILE  
2022

ÍNDICE DE MATERIAS

[1. Diseño 1](#_Toc93092034)

[1.1. Introducción 1](#_Toc93092035)

[1.2. Diagramas del diseño 1](#_Toc93092036)

[1.3. Explicación de la API 6](#_Toc93092037)

[1.4. Diseño de experimentos 9](#_Toc93092038)

[2. Implementación 13](#_Toc93092039)

[2.1. Introducción 13](#_Toc93092040)

[2.2. Especificaciones de Hardware y Software 13](#_Toc93092041)

[2.3. Experimento verificación de estructuras de la librería. 13](#_Toc93092042)

[Resultados 18](#_Toc93092043)

[2.4. Experimento modelos de operaciones inusuales 18](#_Toc93092044)

[Caso 1. Todos los generadores en *Passivate*. 19](#_Toc93092045)

[Caso 2. Un generador en Passivate hace Hold o Passivate. 20](#_Toc93092046)

[Caso 3. Un generador en *Active* recibe un *Activate.* 20](#_Toc93092047)

[2.5. Experimento modelo de productor – consumidor con buffer limitado 20](#_Toc93092048)

[Caso 1. Único productor, único consumidor: 21](#_Toc93092049)

[Caso 2: Múltiple productor – múltiple consumidor 25](#_Toc93092050)

[2.6. Experimento Motor de búsqueda 29](#_Toc93092051)

[3. Anexos 32](#_Toc93092052)

ÍNDICE DE TABLAS

[Tabla 1.1: Agrupación de estructuras de la librería entre públicas y privadas 4](#_Toc93092025)

[Tabla 2.1: Valores de configuración del modelo único productor – único consumidor 22](#_Toc93092026)

[Tabla 2.2: Resultados modelo único productor – único consumidor con estrategia 1 23](#_Toc93092027)

[Tabla 2.3: Resultados modelo único productor – único consumidor con estrategia 2 24](#_Toc93092028)

[Tabla 2.4: Primer resultado del modelo múltiple productor - múltiple consumidor 27](#_Toc93092029)

[Tabla 2.5: Segundo resultado del modelo múltiple productor - múltiple consumidor 28](#_Toc93092030)

[Tabla 2.6: Tercer resultado del modelo múltiple productor - múltiple consumidor 28](#_Toc93092031)

[Tabla 2.7: Configuraciones variables de cada ejecución del modelo de motor de búsqueda 30](#_Toc93092032)

[Tabla 2.8: Tasa de trabajo de los generadores 31](#_Toc93092033)

ÍNDICE DE FIGURAS

[Figura 1.1: Diagrama de clases de la librería de simulación. 6](#_Toc93091938)

[Figura 1.2 - Diagrama de secuencias de la configuración de una simulación en la librería. 7](#_Toc93091939)

[Figura 1.3 - Diagrama de secuencias de una simulación en ejecución 9](#_Toc93091940)

[Figura 1.4: Modelado del proceso de un motor de búsqueda 11](#_Toc93091941)

[Figura 2.1: Código de uno de las pruebas de Scheduler. 14](#_Toc93091942)

[Figura 2.2: Test de Container, múltiples funciones pueden ser ingresadas 15](#_Toc93091943)

[Figura 2.3: Test de Container, generadores pueden ser resumidos 16](#_Toc93091944)

[Figura 2.4: Test de State, ingresar, modificar y eliminar datos de State 17](#_Toc93091945)

[Figura 2.5: Resultado de la llamada al comando cargo test 18](#_Toc93091946)

[Figura 2.6: Modelo de simulación donde todos los generadores entran en Passivate 19](#_Toc93091947)

[Figura 2.7: Generador recibiendo un Activate estando activo. 20](#_Toc93091948)

[Figura 2.8: Ejecución del modelo único productor - único consumidor 21](#_Toc93091949)

[Figura 2.9: JSON de configuración del modelo múltiple productor - múltiple consumidor 26](#_Toc93091950)

[Figura 2.10: Archivo de configuración del modelo de motor de búsqueda 29](#_Toc93091951)

[Figura 2.11: Rendimiento del modelo en base a la configuración 30](#_Toc93091952)

[Figura 3.1: Resultado de ejecutar el modelo de motor de búsqueda con la configuración ID = 1 de la Tabla 2.8 32](#_Toc93091953)

[Figura 3.2: Resultado de ejecutar el modelo de motor de búsqueda con la configuración ID = 2 de la Tabla 2.8 33](#_Toc93091954)

[Figura 3.3: Resultado de ejecutar el modelo de motor de búsqueda con la configuración ID = 3 de la Tabla 2.8 34](#_Toc93091955)

[Figura 3.4: Resultado de ejecutar el modelo de motor de búsqueda con la configuración ID = 4 de la Tabla 2.8 35](#_Toc93091956)

[Figura 3.5: Resultado de ejecutar el modelo de motor de búsqueda con la configuración ID = 5 de la Tabla 2.8 36](#_Toc93091957)

[Figura 3.6: Resultado de ejecutar el modelo de motor de búsqueda con la configuración ID = 6 de la Tabla 2.8 37](#_Toc93091958)

[Figura 3.7: Resultado de ejecutar el modelo de motor de búsqueda con la configuración ID = 7 de la Tabla 2.8 38](#_Toc93091959)

[Figura 3.8: Resultado de ejecutar el modelo de motor de búsqueda con la configuración ID = 8 de la Tabla 2.8 39](#_Toc93091960)

[Figura 3.9: Resultado de ejecutar el modelo de motor de búsqueda con la configuración ID = 9 de la Tabla 2.8 40](#_Toc93091961)

[Figura 3.10: Resultado de ejecutar el modelo de motor de búsqueda con la configuración ID = 10 de la Tabla 2.8 41](#_Toc93091962)

[Figura 3.11: Resultado de ejecutar el modelo de motor de búsqueda con la configuración ID = 11 de la Tabla 2.8 42](#_Toc93091963)

[Figura 3.12: Resultado de ejecutar el modelo de motor de búsqueda con la configuración ID = 12 de la Tabla 2.8 43](#_Toc93091964)

[Figura 3.13: Resultado de ejecutar el modelo de motor de búsqueda con la configuración ID = 13 de la Tabla 2.8 44](#_Toc93091965)

[Figura 3.14: Resultado de ejecutar el modelo de motor de búsqueda con la configuración ID = 14 de la Tabla 2.8 45](#_Toc93091966)

[Figura 3.15: Resultado de ejecutar el modelo de motor de búsqueda con la configuración ID = 15 de la Tabla 2.8 46](#_Toc93091967)

[Figura 3.16: Resultado de ejecutar el modelo de motor de búsqueda con la configuración ID = 16 de la Tabla 2.8 47](#_Toc93091968)

[Figura 3.17: Resultado de ejecutar el modelo de motor de búsqueda con la configuración ID = 17 de la Tabla 2.8 48](#_Toc93091969)

[Figura 3.18: Resultado de ejecutar el modelo de motor de búsqueda con la configuración ID = 18 de la Tabla 2.8 49](#_Toc93091970)

[Figura 3.19: Resultado de ejecutar el modelo de motor de búsqueda con la configuración ID = 19 de la Tabla 2.8 50](#_Toc93091971)

[Figura 3.20: Resultado de ejecutar el modelo de motor de búsqueda con la configuración ID = 20 de la Tabla 2.8 51](#_Toc93091972)

# Diseño

## Introducción

En el siguiente capitulo se entra en detalle sobre el diseño de la librería, las clases involucradas, su función dentro de la librería como interactúan con sus métodos. Además, se incluye una explicación sobre la configuración y ejecución de un modelo de simulación utilizando la librería. Se procederá a mostrar y explicar los experimentos a realizar para verificar y validar el correcto diseño e implementación de la librería. Finalmente se entregará una explicación respecto a las decisiones y filosofía elegidas respecto a la flexibilidad del diseño de la librería.

## Diagramas del diseño

La librería está compuesta de varias estructuras con responsabilidades definidas donde las principales se componen en una única estructura la cual se vuelve el punto de contacto entre el usuario y la simulación. Se explica las estructuras y su función en la librería a continuación.

*Container*:

* Encargada de contener los generadores escritos por los usuarios.
* Crea una *Key* que se asocia al generador ingresado.
* Para acceder a un generador se requiere usar la *Key* asociada.
* Mantiene guardado el estado actual del generador (*Passivated*, *Active*).

*Scheduler*

* Encargada de guardar los *EventEntry* asociados a los generadores.
* Mantiene el tiempo actual de la simulación.
* Cuando un *EventEntry* es retirado de esta estructura avanza el tiempo de simulación.
* Siempre devuelve el *EventEntry* más cercano al tiempo actual de la simulación.

*EventEntry*

* Representa un evento discreto en la simulación.
* Compuesto por
  + Una duración que representa el tiempo de simulación donde el evento ocurre.
  + Una *Key* asociada al generador que emite el evento.

*State*

* Encargada de guardar los datos que 2 o más generadores desean compartir o tener acceso.
* Cada generador es un ambiente aislado, esta estructura provee la capacidad de compartir data.
* Ingresar un dato produce una *StateKey* asociada al dato ingresado.
* Para extraer un dato de esta estructura es necesario utilizar el *StateKey* que tiene asociado.

*ResumeWith*

* Encargada de guardar el dato con el cual se reanuda la ejecución de un generador, si es que así el usuario lo define.
* Necesaria para permitir la simulación de modelos complicados.

*Key*

* Estructura que permite identificar a un generador.
* Es usada en
  + *Container* para obtener acceso al generador asociado.
  + *Scheduler* para asociar un EventEntry con el generador que lo emite.

*StateKey*

* Estructura que permite identificar un tipo ingresado al *State* para su extracción.
* Posee internamente información adicional del dato ingresado junto con una id.

*ComponentState*

* Enumeración con dos variantes
  + *Passivated*: Indica que el generador ha devuelto un *Action::Passivate* y no debe volverse a ingresar al *Scheduler* hasta que algún otro generador realice un *Action::Activate(Key)* con la *Key* de este generador.
  + *Active*: Indica que el generador se encuentra actualmente activo, es decir existe un evento asociado a este generador en el *Scheduler*.

*GeneratorState*

* Enumeración con dos variantes
  + *Yielded(Action)*
  + *Complete(())*
* Proveniente de la dependencia que permite usar generadores en Rust.
* Cuando un generador pausa su ejecución lo hace devolviendo uno de los valores de *Action* mientras que si termina su ejecución completamente no devuelve nada.

Action

* Enumeración con tres variantes
  + *Hold(duración)*: Indica que el generador desea reanudar su ejecución en el tiempo de simulación actual + duración.
  + *Passivate*: Indica que el generador desea pausar su ejecución hasta que algún otro generador lo active.
  + *Activate(Key)*: Indica que el generador actual desea reanudar la ejecución del generador asociado con la *Key* entregada.
* Es un error:
  + Recibir un *Hold* o *Passivate* si el generador se encuentra en estado *Passivated*.
  + Recibir un *Activate* si el generador se encuentra en estado *Active*.
* En caso de error el programa se detendrá forzosamente debido a que el modelo esta incorrecto.

*Co*

* Estructura proveniente de una dependencia de la librería y es parte de la *API* publica de la librería.
* Provee las capacidades de pausar y reanudar la ejecución de los generadores.
* No incluido en el diagrama de clases debido a que es un detalle de implementación

*Component*

* Es una interfaz que define el comportamiento que debe tener un generador.
* Sirve para unificar todos los generadores en un único tipo, esencial para el funcionamiento de *Container*.
* Si no existiera esta interfaz, no sería posible definir un contenedor que guarde todos los posibles generadores que un usuario pueda crear.
  + Este tipo unificado es conocido como *Box<dyn Component<R>>.*

*Simulation*

* Estructura diseñada para ser el punto de contacto del usuario de la librería.
* Es la composición de
  + *Container.*
  + *Scheduler.*
  + *State.*
  + *ResumeWith.*
* Provee todas las capacidades para ejecutar el modelo de simulación del usuario.
* La gran mayoría de sus métodos públicos son llamadas a los métodos de las estructuras de las cuales está compuesto.

Si bien se explica cada estructura y su funcionalidad no se espera que un usuario interactúe con la totalidad de estructuras definidas anteriormente. La Tabla 1.1 agrupa las estructuras en dos grupos: públicas y privadas en el primer grupo se encuentran las estructuras que el usuario es capaz de interactuar mientras que las del segundo grupo son aquellas que el usuario no tiene forma de acceder directamente. Las privadas ocultan detalles de implementación mientras que las públicas dan una interfaz de uso al usuario.

Tabla 1.1: Agrupación de estructuras de la librería entre públicas y privadas

|  |  |
| --- | --- |
| Visibilidad | Estructura |
| Publica | *Simulation*  *ComponentState*  *Co*  *State*  *Key*  *StateKey*  *Action* |
| Privada | *Container*  *Scheduler*  *ResumeWith*  *EventEntry* |

Si bien *Key* y *StateKey* son estructuras publicas ambas tienen un constructor privado lo que significa que el usuario no puede crear una de estas estructuras por su cuenta ya que están asociadas a otros datos debido a esto es responsabilidad de la librería crear estas estructuras, pero una vez creadas se entregan al usuario que puede utilizarla de manera que desee, por ejemplo, como parámetro de métodos que requieren de dichas estructuras para poder utilizarse.

Las estructuras privadas en cambio el usuario no tiene forma de acceder debido a que un mal uso de sus métodos causaría problemas con la ejecución de la simulación que pueden ir desde la ejecución incorrecta de una simulación con un posterior resultado erróneo o el cierre forzoso de la ejecución del programa al encontrarse una situación que teóricamente se consideraría imposible. Por esto *Simulation* que es una estructura publica encapsula estas estructuras privadas y provee de métodos públicos que llaman internamente a los métodos de estas estructuras privadas de tal forma que los errores mencionados anteriormente no puedan ocurrir.

Un diagrama de las estructuras principales de la librería se puede ver a continuación en la Figura 1.1

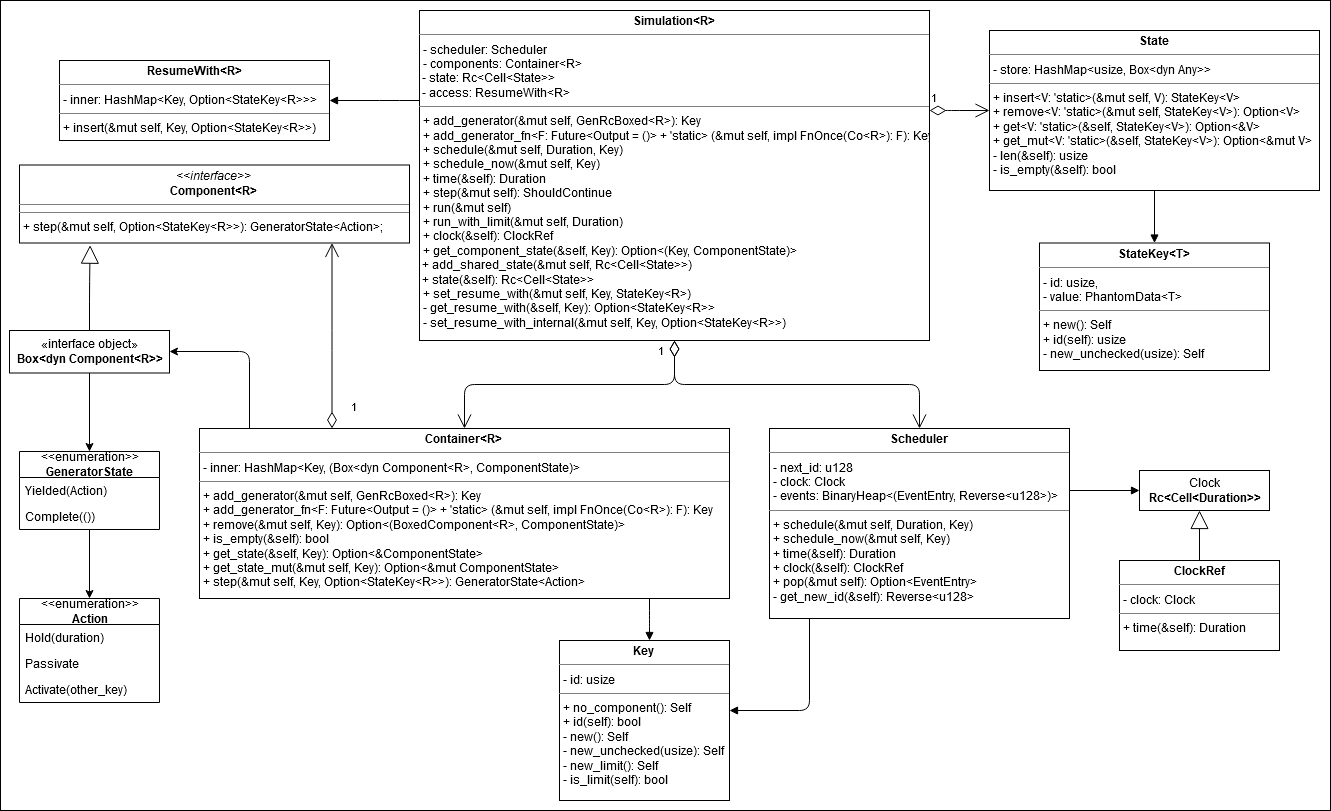


Figura 1.1: Diagrama de clases de la librería de simulación.

## Explicación de la API

Para ejecutar modelos de simulación la librería requiere que se programen las entidades que participaran en la simulación en funciones separadas, cada entidad se vuelve un entorno aislado con su propia data disponible sin acceso al de las demás entidades, si se desea compartir y manipular datos desde varias entidades se puede utilizar guardar los datos en el *State* y entregar las *StateKey* a las respectivas funciones como parámetros de entrada. El proceso de ingresar las entidades, conectarlas y agendarlas se puede ver como un diagrama de secuencias en la Figura 1.2.

Diagrama

Descripción generada automáticamente

Figura 1.2 - Diagrama de secuencias de la configuración de una simulación en la librería.

Las funciones que representan entidades se introducen a la simulación con el método *add\_generator\_fn()* de *Simulation* que internamente transformara la función en un generador y lo guardara en *Container* devolviendo una *Key* asociada al generador.

Si se desea que una entidad active otra entidad que se encuentra en *Passivate* es necesario entregar como parámetro la *Key* de la entidad y como estas no existen al momento de crear la función si no al momento de que ingresan a la estructura *Simulation* no es posible proveer dicha *Key* como parámetro de entrada de los generadores sin embargo el usuario puede utilizar el método *set\_resume\_with()* con las *Key* de las entidades con las que desea interactuar para que al momento de reanudar su ejecución reciba las *Key* a través de la funcionalidad de los generadores de poder ser reanudados con un valor adicional.

Un último paso antes de iniciar la simulación es definir cuando los generadores empezaran a ejecutarse, esto se realiza con los métodos de *Simulación* (que internamente llaman a los de *Scheduler* del mismo nombre) *schedule()* y *schedule\_now().* El orden de llamadas a estas funciones importa, cuando un evento asociado a un generador es agendado en el *Scheduler* este recibe una id auto incremental, si varios generadores buscan ejecutarse en el mismo tiempo de simulación el *Scheduler* devolverá aquel que tenga una id menor de esta forma el usuario puede definir con certeza que entidad ejecutara primero su lógica aun si varias entidades desean ejecutarse en el mismo tiempo de simulación, con este paso completado se termina el proceso de configuración de la simulación pudiendo así continuar con la ejecución del mismo.

Una forma de iniciar una simulación es con el método *run\_with\_limit()* de la estructura *Simulation*, el método inicia un ciclo con dos condiciones de termino la primera es que no existen más eventos en el *Scheduler*, la segunda condición es si evento retornado es el uno especial denominado limite que solo es creado con el método *run\_with\_limit()* una única vez, si ninguna de las condiciones se cumple la ejecución del ciclo continua de forma normal llamando al método *step()* de *Container* con los datos extraídos del evento, dicho método extraerá el generador asociado con la *Key* ingresada, reanudara su ejecución 1 paso, es decir hasta que vuelva a pausarse, devolviendo un todo el camino atrás hasta *Simulation* que revisara el valor de *GeneratorState*, si es *Complete(x)* significa que el generador termino su ejecución devolviendo *x* y no debe volver a ser resumido, en caso de serlo el programa terminara forzosamente con un error, y si en cambio recibe *Yielded(Action)* procederá a revisar el valor de *Action* el cual puede ser tanto *Hold(duration)*, *Passivate* o *Activate(other\_key)*, en caso de un *Hold* simplemente se vuelve a agendar el generador utilizando el método *schedule()* de *Scheduler*. En caso de *Passivate* se solicita el *ComponentState* del generador a *Container* con su método *get\_state\_mut()* y se modifica a *ComponentState::Passivated*, en caso de que el generador ya estuviera en *Passivated* el programa termina forzosamente con un error, un caso parecido ocurre cuando Action tiene valor *Activate* allí se obtiene el *ComponentState* del generador, se revisa que el estado no sea *Active*, se termina el programa con error en caso de serlo, se actualiza el estado a *Active* para finalmente agendar de forma inmediata al generador actual y al generador activado, al realizar cualquiera de estas acciones vuelve al inicio del ciclo continuando la ejecución de la simulación. La situación descrita anteriormente se puede observar en el diagrama de secuencias de la Figura 1.3

Diagrama

Descripción generada automáticamente

Figura 1.3 - Diagrama de secuencias de una simulación en ejecución

## Diseño de experimentos

Se contemplan realizar 4 experimentos para verificar y validar el diseño, implementación y desempeño de la librería, estos son:

* Verificación de estructuras de la librería.
* Modelos de operaciones inusuales
* Productor – Consumidor con buffer limitado.
* Motor de búsqueda.

La verificación de estructuras de la librería consiste en pruebas unitarias para asegurar de que el comportamiento en solitario de cada estructura es el deseado.

Modelos de operaciones inusuales consiste en la creación y ejecución de modelos realizando operaciones raras pero validas o totalmente invalidas asegurando de que ante tales situaciones la librería responde como corresponde.

El experimento de Productor – Consumidor es en general un modelo sencillo utilizado principalmente para verificar que una librería cumple con las especificaciones básicas de un simulador computacional, dentro de este experimento se consideran 2 variaciones de productores consumidores siendo estas:

* Un productor – Un consumidor.
* Múltiples productores – Múltiple consumidores.

Con estos experimentos se verifica la capacidad de la librería de:

* Permitir a dos o más generadores compartir un recurso.
* Asegurar el orden correcto de la ejecución de los eventos.
* Definir valores arbitrarios para la ejecución de los generadores sin cambiar drásticamente la lógica de estos.
* Recolectar estadísticas sobre los resultados obtenidos.

En este experimento se añade la condición de que el buffer donde los productores dejan el recurso y los consumidores lo retiran sea limitado para así poder definir el comportamiento que deben tener a la hora de encontrarse un buffer lleno o vacío en caso de productores o consumidores respectivamente.

El experimento de motor de búsqueda consiste en realizar una simulación en base a un modelo de motor de búsqueda, dicho modelo consiste en 4 entidades: *Query Generator Service*, *Front Service*, *Cache Service* y por último el *Inverted Index Service*. Estas entidades interactúan de manera específica y cumplen un determinado rol en la simulación la relación entre las entidades y como se comunican entre sí se puede ver en la Figura 1.4

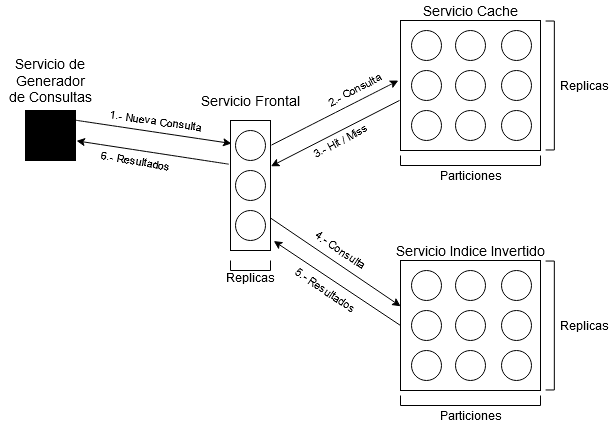


Figura 1.4: Modelado del proceso de un motor de búsqueda

*Query Generator Service* es una entidad que representa a una persona ingresando una consulta al motor de búsqueda, su labor es emitir consultas con tal de imitar el tráfico que tienen estos servicios.

*Front Service* es la entidad que simula ser la página que recibe la consulta del usuario y la envía al servicio correspondiente para realizar la búsqueda entregándole al usuario los resultados obtenidos.

*Cache Service* es el modelado de un sistema cache funcional de un motor de búsqueda, tiene tiempos de respuestas de magnitudes menores que *el Inverted Index Service,* pero de tamaño limitado que impiden guardar todas las consultas posibles que un usuario pueda realizar.

*Inverted Index Service* representa la estructura y funcionalidad de un índice invertido de búsqueda, cuando una consulta no se encuentra en el cache.

Figura 1.4 Muestra como cada servicio está dividido en particiones y replicas, las implicaciones de esto son las siguientes:

* Todas las réplicas pertenecen a una partición.
  + Existe mínimo al menos una partición y una réplica por servicio
  + *Query Generator Service* puede considerarse como un servicio con una partición y una réplica.
* Todas las réplicas en una partición comparten la misma cola de consultas.
  + Esto implica que cada partición tiene una cola distinta de consultas.
* Cada replica trabaja en una consulta distinta.
  + Ninguna replica trabaja en la misma consulta que otra replica ya haya empezado a trabajar.
  + Si una réplica del *Front Service* envía una consulta al *Cache Service*, cuando este entregue su resultado la réplica que envió la consulta debe ser la que procese dichos resultados.
  + Una excepción es el *Inverted Index Service* debido al procedimiento que debe seguir.

La simulación inicia con *Query Generator Service* el cual crea y envía consultas al *Front Service* este recibe la consulta, la redirige al *Cache Service* el cual devuelve *Hit* o *Miss.* Recibir un *Hit* indica que la consulta se encontraba en *Cache* y todo lo que el *Front Service* debe hacer es retornar los resultados del cache, en cambio obtener un *Miss* indica lo contrario, la consulta no se encontró en el *Cache Service* por lo que una búsqueda a fondo es requerida, en este caso el *Front Service* solicita al *Inverted Index Service* que busque la consulta en su directorio de documentos, debido a que los índices invertidos tienden a contener grandes volúmenes de datos el trabajo de buscar en ellos se distribuye en varios equipos por esto cuando llega una consulta a este servicio se reparte a todas las réplicas de una partición las cuales buscaran en un sector determinado recopilando los resultados que obtengan de allí devuelta al *Front Service* este al recibir los resultados de todas las réplicas realizara un procedimiento de ranking y reordenara los resultados en base a cual le será más importante al usuario que realizo la consulta una vez termine devolverá el resultado obtenido terminando el proceso.

# Implementación

## Introducción

Este capítulo está dedicado a mostrar todo lo relacionado a los experimentos, las especificaciones con la cual fueron ejecutados, el código encargado de verificar el experimento, los valores de configuración a los cuales fue sometidos y los resultados obtenidos.

## Especificaciones de Hardware y Software

Todos los experimentos se realizan con las siguientes especificaciones:

Hardware:

* Notebook OMEN by HP 15-ce0xx.
* Intel Core i5-7300HQ.
* 16 GB DDR4 2400 MHz.
* 1000 GB SSD KINGSTON SA400S3796G.

Software:

* Windows 10 Home Edition 64-bits (Kernel version 19043).
* Rust versión 1.57 estable obtenido desde Rustup (Herramienta oficial de distribución del lenguaje)
* Compilador y herramientas de C++ obtenidos desde Visual Studio Community Edition 2021.

## Experimento verificación de estructuras de la librería.

El experimento que más veces se ha realizado a lo largo del desarrollo de la librería. Cada estructura mayor (aquellas que cumplen una funcionalidad importante en la librería) tiene un set definido de pruebas que debe completar con éxito para decir que este experimento es exitoso, cada cambio introducido a la librería deberá de ser verificado, es decir, por cada cambio se ejecutaran todas las pruebas de las cuales ninguna puede fallar. Las estructuras testeadas son las siguientes:

* Scheduler
* Container
* State
* Simulation

Estas 4 estructuras poseen estructuras asociadas las cuales son testeadas en una o más de las estructuras mencionadas anteriormente.

*Scheduler*: Los test de esta estructura están centrados principalmente en verificar la implementación correcta de la inserción y extracción de eventos, para asegurar que los eventos emitidos por los generadores sean ejecutados en el momento correcto en el orden correcto de lo contrario la simulación puede fallar de forma inesperada o producir un resultado incorrecto sin dar indicios de ello. Figura 2.1 muestra el código de uno de los test de *Scheduler* para comprobar que eventos sin importar cuando son introducidos siempre son extraídos en el orden correcto.

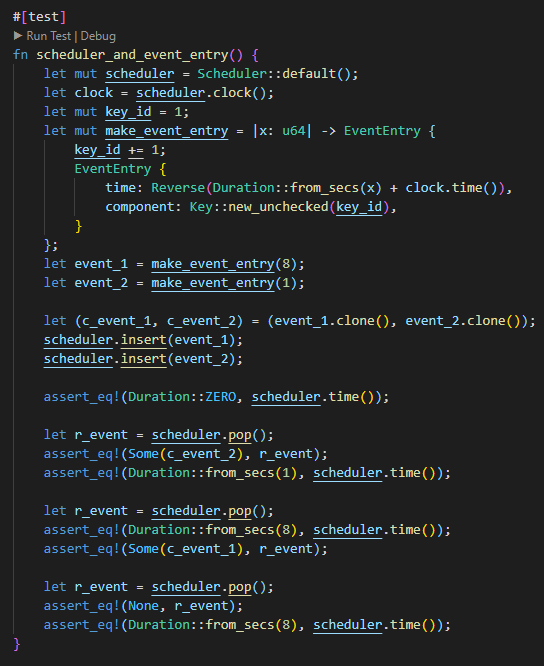


Figura 2.1: Código de uno de las pruebas de Scheduler.

La mayoría de las pruebas no emiten texto por su cuenta, en cambio utilizan los macros *assert, assert\_eq* o *assert\_ne* para asegurar que el output correcto es conseguido con respecto a un determinado input, en caso de que no coincidan el test indicara fracaso.

*Container*: Igual que en el *Scheduler* los test están centrados en verificar que la implementación de la estructura entregue los resultados que se le esperan a la hora de utilizarla, estas pruebas consisten en crear distintas funciones con distinto cuerpo y convertirlos en generadores, ingresarlos a la estructura, generar las llaves correspondientes, verificar que con dichas llaves se puedan acceder a los generadores de manera correcta, por ultimo determinar si la ejecución, pausa y reanudación de los generadores ocurre de forma correcta, en la Figura 2.2 se muestra una prueba que utiliza assert\_eq para verificar que los valores obtenidos son los deseados.

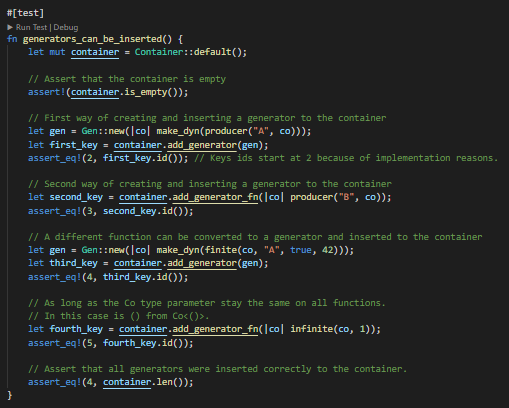


Figura 2.2: Test de Container, múltiples funciones pueden ser ingresadas

Figura 2.3 muestra un test que no contiene *assert* de ningún tipo en cambio llama a una función de container, lo que hace dicha función no es de importancia, lo importante es que después de que una condición se cumpla no puede volver a ser llamada y en caso de hacerlo el test debe fallar de forma similar a un *assert* no cumplido.

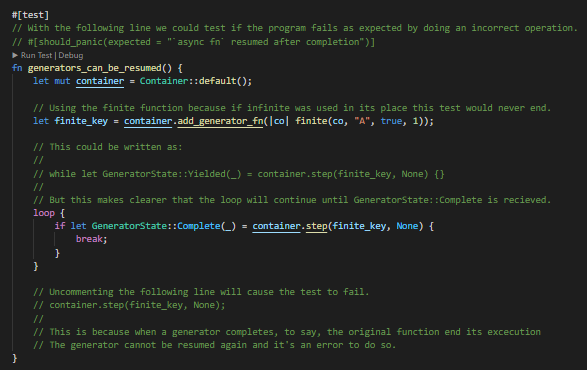


Figura 2.3: Test de Container, generadores pueden ser resumidos

*State*:No muy diferente de Scheduler, esta estructura tiene su funcionalidad definida y las pruebas que se le realizan se usan para comprobar de que los inputs ingresados generan los outputs deseados. Figura 2.4Figura 2.4 contiene dos pruebas, una encargada de verificar que al ingresar un valor se produzca una *Key* (más específicamente una *StateKey*) que puede usarse para obtener acceso al valor ingresado, de igual manera la segunda prueba verifica una situación similar al primer test, se crea el State sin ningún valor dentro, se ingresa un valor y se modifica el valor estando dentro del State utilizando la *Key* correspondiente para luego extraer el valor y verificar de que el valor de verdad ha sido cambiado con éxito, si el valor no cambiara, la llave no correspondiera al valor ingresado o no se pudiera extraer los datos ingresados, los test fallarían indicando un error en la implementación de la estructura.



Figura 2.4: Test de State, ingresar, modificar y eliminar datos de State

*Simulation*: Por su cuenta esta estructura no cuenta con un set de test como las demás, debido a que constituye casi la totalidad de la librería, habría que repetir los test de las otras estructuras en esta, además habría que incluir pruebas de integración para verificar que las estructuras siguen produciendo los valores correctos cuando trabajan en equipo en lugar de en solitario. Para evitar la repetición, las pruebas de otras estructuras no se repiten en *Simulation* y los pruebas de integración se realizan desarrollando modelos de simulación como un usuario de la librería lo haría, los próximos experimentos sirven cumplirán la doble función de confirmar de que la librería funciona correctamente utilizando exclusivamente su interfaz publica y validar el uso de la librería como simulador computacional al desarrollar un modelo conocido y producir los resultados esperados ejecutando un modelo ya validado.

### Resultados

Figura 2.5 tiene el output del terminal que se obtiene con una llamada al comando *cargo test*. Dicho comando ejecutara todas las funciones marcadas con el atributo *#[test]* observado en la parte superior de las funciones de las figuras anteriores, además ejecutara como test el código escrito en la documentación de la librería junto con el código que se encuentre en una carpeta */tests* en el directorio del proyecto y el código que se encuentre en la carpeta */examples*.

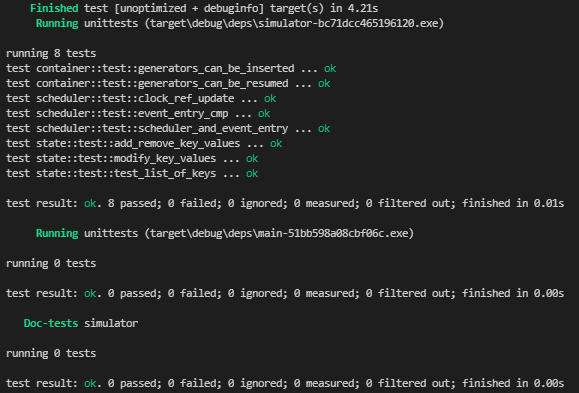


Figura 2.5: Resultado de la llamada al comando cargo test

## Experimento modelos de operaciones inusuales

Para todos los procedimientos existen inputs que causan situaciones que, aunque inusuales hay que manejar de manera correcta, este experimento consiste en realizar pequeños modelos diseñados específicamente para ejecutar dichas situaciones y comprobar de que la librería responda de forma adecuada, es decir, si la situación es inusual pero valida la ejecuta con normalidad en cambio si la situación es invalida debe detener la simulación con un error.

### Caso 1. Todos los generadores en *Passivate*.

Una situación parecida a un *deadlock* puede ocurrir cuando un generador entra en *Passivate* esperando recibir en algún momento en el futuro un *Activate*, sin embargo, el generador que debe mandar tal señal ya se encuentra en *Activate* causando que la simulación continue sin realizar trabajo hasta el infinito o hasta que el tiempo límite definido para correr la simulación sea alcanzado. Figura 2.6 muestra el código completo de cómo es dicho modelo y el resultado de su ejecución.

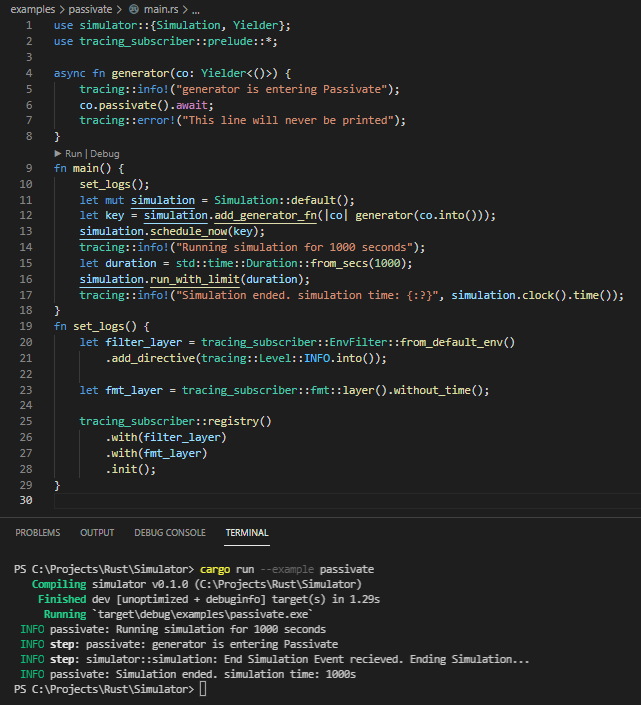


Figura 2.6: Modelo de simulación donde todos los generadores entran en Passivate

### Caso 2. Un generador en Passivate hace Hold o Passivate.

Este experimento probo ser una situación incapaz de ser modelada debido a que el usuario no tiene acceso al estado de los generadores, todos empiezan en *ComponentState::Active* y desde ahí pueden mantener registro del estado actual del generador, sin embargo, no pueden modificar directamente dicho estado y cuando un generador cambia a *ComponentState::Passivated* es porque de verdad realizo un *Passivate* y no volverá a ejecutarse hasta que reciba un *Activate*. A pesar de todo la librería que mantiene registro de los estados actuales de los generadores realiza una revisión y si detecta que un generador en *Passivated* recibe cualquier otra acción en lugar de *Activate* detiene la simulación con error.

### Caso 3. Un generador en *Active* recibe un *Activate.*

Una situación fácil de caer en que para evitarla se requiere que el usuario mantenga registro de los estados actuales de los generadores, actualizándolos cuando sea debido y haciendo las revisiones correspondientes para evitar esta situación. En caso de ocurrir la librería detendrá el programa con un error. Figura 2.7 muestra el código de tal situación excluyendo las importaciones y la definición de la función *set\_logs()* que es la misma que se puede ver en la Figura 2.6.

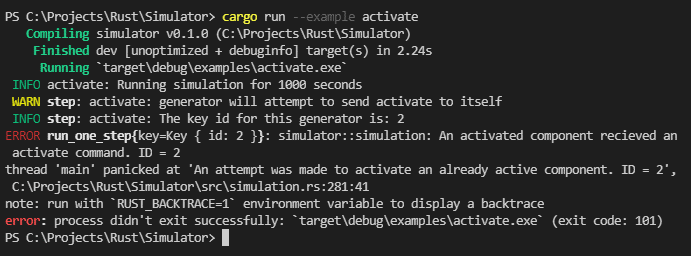


Figura 2.7: Generador recibiendo un Activate estando activo.

## Experimento modelo de productor – consumidor con buffer limitado

Probablemente uno de los modelos más conocidos para enseñar acerca de la simulación computacional. Para este experimento se prepararon dos casos de estudio, ambos con dos estrategias para decidir qué hacer en caso de que el buffer se encuentre lleno o vacío, para el productor y consumidor respectivamente.

### Caso 1. Único productor, único consumidor:

Para este caso de estudio se preparó un modelo de simulación de un productor y un consumidor, el productor agrega elementos al buffer y el consumidor los elimina, ambos se demoran una cantidad determinada de tiempo para realizar esta acción y pueden agregar o eliminar varios elementos en una sola acción. **¡Error! No se encuentra el origen de la referencia.¡Error! No se encuentra el origen de la referencia.** muestra la ejecución de dicho modelo.

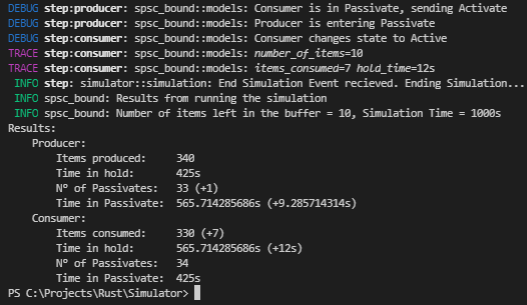


Figura 2.8: Ejecución del modelo único productor - único consumidor

Además, se definen dos estrategias que definen cuando se realiza el chequeo de *Passivated* y su posterior *Activate*.

* Estrategia 1: Cuando el buffer se encuentre lleno, en el caso del productor, o vacío, para el consumidor, revisaran si el otro generador se encuentra en *Passivated* y en caso de estarlo le mandan un *Activate* para finalmente ejecutar un *Passivate*.
* Estrategia 2: La revisión de *Passivated* se realiza al inicio de cada acción tomada por los generadores, en caso de que el otro se encuentre en *Passivated* se procede a enviar el correspondiente *Activate* y continua su ejecución de forma normal, si el buffer se encuentra lleno (para el productor) o vacío (para el consumidor) se procede a realizar un *Passivate*.

Para este caso de estudio se prepararon 7 archivos de configuración que definen los valores a los cuales se someten los generadores. Tabla 2.1 enseña de forma resumida dichos archivos.

Tabla 2.1: Valores de configuración del modelo único productor – único consumidor

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Simulación | | Productor | | | Consumidor | |
| N° | Limite (segundos) | Max Ítems | Ítems | Hold (segundos) | Ítems | Hold (segundos) |
| 1 | 1000s | 16 | 1 | 7s | 1 | 8s |
| 2 | 1000s | 16 | 1 | 15s | 1 | 12s |
| 3 | 1000s | 10 | 4 | 2s | 2 | 4s |
| 4 | 1000s | 150 | 30 | 200s | 20 | 150s |
| 5 | 1000s | 16 | 5 | 10s | 8 | 5s |
| 6 | 1000s | 81 | 9 | 1s | 2 | 12s |
| 7 | 1000s | 10 | 4 | 5s | 7 | 12s |

La tabla consiste de 3 columnas

* Simulación contiene las configuraciones respectivas de la simulación en si
  + En este caso el tiempo máximo a simular.
* Productor define
  + El valor máximo de ítems que puede haber en el buffer
  + Cuantos ítems produce por cada acción que toma
  + Cuanto tiempo se demora en añadir todos los ítems al buffer.
* Consumidor especifica
  + Cuantos ítems elimina del buffer por cada ciclo
  + Cuanto tiempo se demora en eliminar todos los ítems.

Un detalle a tener en cuenta es que, si el buffer no tiene la suficiente capacidad para llevar a cabo toda la acción, la realizara de forma parcial calculando el tiempo que le demoraría en hacerlo, por ejemplo, si el productor desea añadir 4 ítems y se demora 10 segundos, pero el buffer solo tiene espacio para 2, este agregara los 2 ítems y hará un Hold por 5 segundos.

La Tabla 2.2 y contienen los resultados de ejecutar su modelo con los valores de configuración de la Tabla 2.1 aplicando la estrategia 1.

Tabla 2.2: Resultados modelo único productor – único consumidor con estrategia 1

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Simulación | | Productor | | | | Consumidor | | | |
| N° | Límite (segundos) | N° de ítems | Hold (segundos) | N° de pasivaste | Passivate (segundos) | N° de ítems | Hold (segundos) | N° de passivates | Passivate (segundos) |
| 1 | 1000s | 69 (+1) | 483s (+7s) | 4 | 512s | 64 | 512s | 4 (+1) | 448s (+40s) |
| 2 | 1000s | 41 (+1) | 615s (+15s) | 2 | 384s | 32 | 384s | 2 (+1) | 480s (+136s) |
| 3 | 1000s | 400 | 200s | 39 (+1) | 780s (+20) | 398 (+2) | 769s (+4s) | 40 | 200s |
| 4 | 1000s | 120 (+30) | 800s (+200s) | 0 | 0s | 0 | 0s | 0 (+1) | 0s (+1000s) |
| 5 | 1000s | 384 | 768s | 23 (+1) | 230s (+2s) | 368 (+8) | 230s (+5s) | 24 | 768s |
| 6 | 1000s | 243 | 27s | 2 (+1) | 972s (+1s) | 162 (+2) | 972s (+12s) | 3 | 27s |
| 7 | 1000s | 340 | 425s | 33 (+1) | 565,71s (+9,29s) | 330 (+7) | 565,71s (+12s) | 34 | 425s |

Tabla 2.3 en cambio contiene los resultados de ejecutar el modelo con los valores de configuración aplicando la estrategia 2.

Tabla 2.3: Resultados modelo único productor – único consumidor con estrategia 2

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Simulación | | Productor | | | | Consumidor | | | |
| N° | Límite (segundos) | N° de ítems | Hold (segundos) | N° de pasivaste | Passivate (segundos) | N° de ítems | Hold (segundos) | N° de passivates | Passivate (segundos) |
| 1 | 1000s | 139 (+1) | 973s (+7s) | 26 | 26s | 124 (+1) | 992s (+8s) | 1 | 7s |
| 2 | 1000s | 66 (+1) | 990s (+15s) | 0 | 0s | 65 (+1) | 780s (+12s) | 66 | 210s |
| 3 | 1000s | 508 | 254s | 249 (+1) | 745s (+1s) | 498 (+2) | 996s (+4s) | 1 | 2s |
| 4 | 1000s | 150 (+30) | 1000s (200s) | 0 | 0s | 100 (+20) | 750s (+150s) | 1 | 200s |
| 5 | 1000s | 500 | 1000s | 0 | 0s | 495 | 309,38s | 99 (+1) | 683,75s (+6,88s) |
| 6 | 1000s | 247 | 27,44s | 83 (+1) | 696,78s (+2,78s) | 166 (+2) | 996s (+12s) | 1 | 1s |
| 7 | 1000s | 584 (+4) | 730s (+5s) | 82 | 266,50s | 578 (+4) | 990,86s (+6.86s) | 1 | 5s |

### Caso 2: Múltiple productor – múltiple consumidor

La lógica de los generadores es casi la misma, si pueden producir o consumir elementos, modificaran la lista, si no pueden entraran en *Passivated*, lo que cambia en este experimento del anterior es que ahora existe más de un productor y consumidor por lo que la lógica para mandar un *Activate* es un poco diferente. Primero que todo debemos diferenciar entre productores y consumidores luego a los productores se les dará acceso a todos los consumidores en *Passivated* y viceversa creando dos colas donde el primer generador que entra en *Passivated* será el primero en ser extraído, de esta forma se aplica una estrategia de mesa redonda, donde siempre se despierta un generador de forma distribuida.

Otra diferencia con el caso anterior deriva del hecho de que los productores y consumidores ya no están limitados a 1 esto implica que debe existir la capacidad de soportar un numero arbitrario de productores y consumidores, cada uno con sus propios valores de configuración. Figura 2.9 muestra un archivo de configuración para este modelo en formato JSON

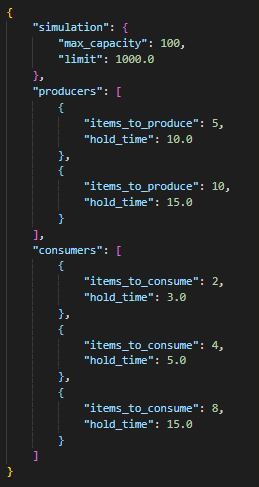


Figura 2.9: JSON de configuración del modelo múltiple productor - múltiple consumidor

Es un archivo JSON con 3 campos, “simulation”, “producers” y “consumers”.

* “simulation” por su parte contiene dos valores, “max\_capacity” que representa la capacidad máxima del buffer y “limit” que define el límite de tiempo de la simulación.
* “producers” es una lista donde sus elementos son un agregado de “ítems\_to\_produce” expresando cuantos ítems busca agregar el productor al buffer y “hold\_time” que especifica cuanto tiempo demora el productor en agregar los elementos al buffer
* “consumers” también es una lista definiendo donde sus elementos definen “ítems\_to\_consume” y “hold\_time” que cumplen una función similar a los definidos en “producers”

Explicando con palabras el archivo de configuración dice lo siguiente:

La simulación tendrá un buffer con tamaño máximo de 100 elementos y simulará a lo máximo 10000 segundos, contará con dos productores el primero producirá 15 elementos en intervalos de 30 segundos, el segundo producirá 40 elementos en intervalos de 25 segundos, por último, se tiene 4 consumidores consumiendo 4 elementos cada 10 segundos, 30 elementos cada 40 segundos, 10 elementos cada 20 segundos y 50 elementos cada 8 segundos.

Al realizar la simulación con los valores de la Figura 2.9 se obtienen los valores descritos en la Tabla 2.4.

Tabla 2.4: Primer resultado del modelo múltiple productor - múltiple consumidor

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Categoría | Ítems | Hold (segundos) | Passivates | Passivated Time (segundos) | Activates Enviados |
| Productor 1 | 500 | 1000s | 0 | 0s | 99 |
| Productor 2 | 660 (+10) | 990s (+15s) | 0 | 0s | 66 |
| Consumidor 1 | 332 | 498s | 66 (+1) | 496,5s (+5,5s) | 0 |
| Consumidor 2 | 333 | 416,25s | 83 (+1) | 578,75s (+5s) | 0 |
| Consumidor 3 | 497 (+8) | 931,88s (+15s) | 16 | 60s | 0 |

Todos los resultados de este modelo compartirán las columnas y variarán en el número de filas. Para mantener las cosas simples, solo se realizarán 3 simulaciones donde se agregará ya sea productores y/o consumidores al archivo de configuración base y la configuración ya existente no será alterada.

Para la siguiente ejecución se agrega un productor con los valores “ítems\_to\_produce” = 15 y “hold\_time” = 16.5.

Los valores obtenidos se pueden ver en la Tabla 2.5

Tabla 2.5: Segundo resultado del modelo múltiple productor - múltiple consumidor

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Categoría | Ítems | Hold (segundos) | Passivates | Passivated Time (segundos) | Activates Enviados |
| Productor 1 | 495 (+5) | 990s (+10s) | 0 | 0s | 0 |
| Productor 2 | 660 (+10) | 990s (+15s) | 0 | 0s | 1 |
| Productor 3 | 900 (+15) | 990s (+16,5s) | 0 | 0s | 0 |
| Consumidor 1 | 665 (+2) | 997,5s (+3s) | 0 | 0s | 0 |
| Consumidor 2 | 796 (+4) | 995s (+5s) | 0 | 0s | 0 |
| Consumidor 3 | 553 (+8) | 999,38s (+15s) | 1 | 0s | 0 |

Para la última ejecución de este experimento se agregará ambos un productor y un consumidor con los valores “ítems\_to\_produce” = 40, “hold\_time” = 180.0 y “ítems\_to\_consume” = 55, “hold\_time” = 202.0. Los resultados obtenidos se pueden ver en la Tabla 2.6.

Tabla 2.6: Tercer resultado del modelo múltiple productor - múltiple consumidor

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Categoría | Ítems | Hold (segundos) | Passivates | Passivated Time (segundos) | Activates Enviados |
| Productor 1 | 495 (+5) | 990s (+10s) | 0 | 0s | 0 |
| Productor 2 | 660 (+10) | 990s (+15s) | 0 | 0s | 1 |
| Productor 3 | 900 (+15) | 990s (+16,5s) | 0 | 0s | 0 |
| Consumidor 1 | 900 (+15) | 990s (+16,5s) | 0 | 0s | 0 |
| Consumidor 2 | 665 (+2) | 997,5s (+3s) | 0 | 0s | 0 |
| Consumidor 3 | 796 (+4) | 995s (+5s) | 0 | 0s | 0 |
| Consumidor 4 | 553 (+8) | 999,38s (+15s) | 1 | 0s | 0 |

## Experimento Motor de búsqueda

El experimento anterior consistió de dos tipos de generadores denominados productor y consumidor en el cual dependiendo del caso variaba la cantidad de productores y consumidores, pero la cantidad de tipos distintos de generadores seguía siendo de dos. Para este experimento se necesitan 4 tipos de generadores: *Query Generator*, *Front Service*, *Cache Service*, *Inverted Index Service* que se comunican como lo indica la Figura 1.4. Estos 4 tipos de generadores forman el modelado de un motor de búsqueda que se puede usar para estimar el performance de un motor real bajo circunstancias similares a la de la simulación.

El archivo de configuración encargado de definir los valores a los cuales se someterá este modelo de simulación se puede observar en la Figura 2.10. El archivo se encuentra en formato JSON y los valores en su determinan el tiempo que demora cada acción de los servicios, la semilla para los valores aleatorios, la cantidad de particiones – replicas, entre otros datos.

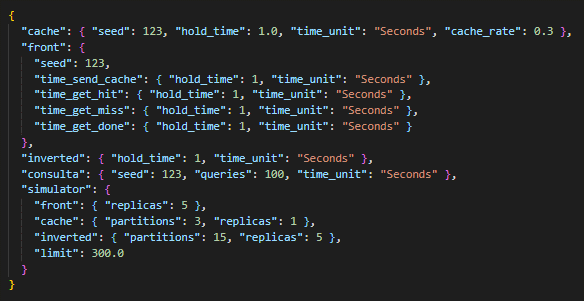


Figura 2.10: Archivo de configuración del modelo de motor de búsqueda

Los siguientes valores se mantienen constantes durante todas las ejecuciones del modelo: “consulta queries” = 1000. Todas las “seed” = 123. Todos los “time\_unit” = “Seconds”. “limit” = 300.0. “cache hold\_time” = 0.0110460. Los valores que en cambio varían entre una configuración y otra se especifican junto con sus valores en la Tabla 2.7.

La tabla define una ID para asociar los resultados con las configuraciones especificadas. Configuración define la cantidad de <FS Replicas, CS Particiones, CS Replicas, IIS Particiones, IIS Replicas> respectivamente que al sumarse generan el valor en la columna de Nro. de máquinas. El rendimiento de dichos valores se observa en la Figura 2.11 que calcula la cantidad de consultas manejadas por segundo.

Tabla 2.7: Configuraciones variables de cada ejecución del modelo de motor de búsqueda

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ID | Nro. de Maquinas | Configuración | Hold Time de FS (segundos) | Hold Time de IIS (segundos) | Tasa de hit CS |
| 1 | 115 | <12,3,1,10,10> | 0.00660711 | 0.10036 | 0.290252 |
| 2 | 118 | <15,3,1,10,10> | 0.00660518 | 0.100364 | 0.290252 |
| 3 | 118 | <13,5,1,10,10> | 0.00656825 | 0.10069 | 0.295845 |
| 4 | 122 | <15,7,1,10,10> | 0.00654679 | 0.100839 | 0.299441 |
| 5 | 138 | <15,3,1,15,8> | 0.00660668 | 0.066873 | 0.290252 |
| 6 | 145 | <18,7,1,12,10> | 0.00654422 | 0.0839778 | 0.299441 |
| 7 | 167 | <12,5,1,15,10> | 0.00656803 | 0.0671184 | 0.295845 |
| 8 | 175 | <20,5,1,15,10> | 0.0065646 | 0.0670875 | 0.295845 |
| 9 | 176 | <20,6,1,15,10> | 0.00655313 | 0.0671009 | 0.298042 |
| 10 | 189 | <22,7,1,16,10> | 0.00654636 | 0.0629947 | 0.299441 |
| 11 | 193 | <23,5,1,15,11> | 0.00656589 | 0.0670869 | 0.295845 |
| 12 | 200 | <24,6,1,17,10> | 0.00655807 | 0.0592222 | 0.298042 |
| 13 | 214 | <27,7,1,12,15> | 0.00654165 | 0.0839762 | 0.299441 |
| 14 | 215 | <30,10,1,25,7> | 0.00652358 | 0.0403762 | 0.302837 |
| 15 | 220 | <30,10,1,20,9> | 0.00652101 | 0.0504786 | 0.302837 |
| 16 | 220 | <30,10,1,18,10> | 0.00652187 | 0.0560872 | 0.302837 |
| 17 | 223 | <33,10,1,20,9> | 0.00652358 | 0.0504916 | 0.302837 |
| 18 | 226 | <36,15,1,25,7> | 0.00649784 | 0.0405203 | 0.309628 |
| 19 | 229 | <22,7,1,20,10> | 0.0065453 | 0.0503781 | 0.299441 |
| 20 | 240 | <40,20,1,30,6> | 0.00646348 | 0.0338921 | 0.312625 |

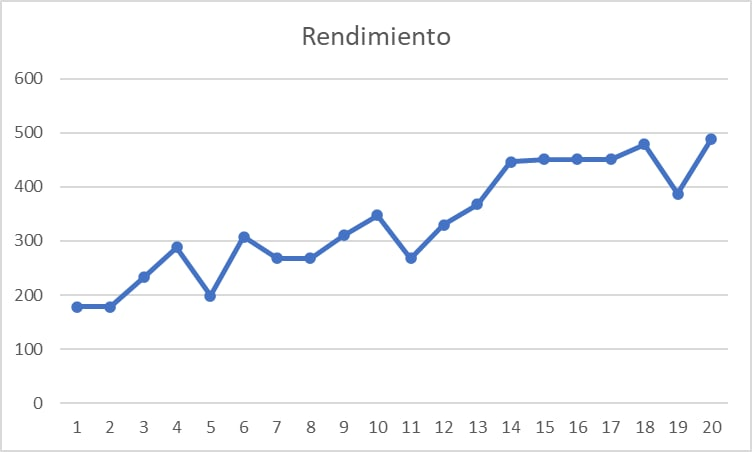


Figura 2.11: Rendimiento del modelo en base a la configuración

Otra medida de interés de estudio del modelo es determinar qué tan ocupados estuvieron los generadores en base a las condiciones aplicadas. Tabla 2.8 contiene dicha información, la cual clasifica las distintas etapas por la cual pasa la consulta con el respectivo generador.

Tabla 2.8: Tasa de trabajo de los generadores

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ID | % FS TOTAL | % FS USER | % FS HIT | % FS MISS | % FS COMPLETE | % CS TOTAL | % CS HIT | % CS MISS | % IIS |
| 1 | 75,44 | 55,00 | 4,34 | 10,61 | 5,49 | 99,99 | 29,02 | 70,97 | 99,9 |
| 2 | 60,33 | 43,99 | 3,47 | 8,49 | 4,39 | 99,99 | 29,02 | 70,97 | 99 ,96 |
| 3 | 78,36 | 50,47 | 6,75 | 16,12 | 5,02 | 99,99 | 29,50 | 70,49 | 99,97 |
| 4 | 75,58 | 43,60 | 8,25 | 19,40 | 4,33 | 99,98 | 29,84 | 70,15 | 99,97 |
| 5 | 61,23 | 44,00 | 3,47 | 8,49 | 5,27 | 99,99 | 29,02 | 70,97 | 99,98 |
| 6 | 63,68 | 36,32 | 6,87 | 16,16 | 4,33 | 99,98 | 29,84 | 70,15 | 99,97 |
| 7 | 86,78 | 54,68 | 7,31 | 17,46 | 7,34 | 99,97 | 29,49 | 70,49 | 89,98 |
| 8 | 52,05 | 32,79 | 4,38 | 10,47 | 4,40 | 99,99 | 29,49 | 70,49 | 89,98 |
| 9 | 55,41 | 32,73 | 5,29 | 12,51 | 4,88 | 99,99 | 29,71 | 70,28 | 99,98 |
| 10 | 53,30 | 29,72 | 5,63 | 13,23 | 4,72 | 99,99 | 29,84 | 70,15 | 99,97 |
| 11 | 45,26 | 28,52 | 3,81 | 9,11 | 3,83 | 99,99 | 29,50 | 70,49 | 81,80 |
| 12 | 46.75 | 27,30 | 4,41 | 10,43 | 4,61 | 99,99 | 29,71 | 70,28 | 99,98 |
| 13 | 43,88 | 24,20 | 4,58 | 10,77 | 4,33 | 99,98 | 29,84 | 70,15 | 99,97 |
| 14 | 45,15 | 21,72 | 5,93 | 13,73 | 3,77 | 99,88 | 30,15 | 69,73 | 99,98 |
| 15 | 45,24 | 21,71 | 5,93 | 13,72 | 3,88 | 99,88 | 30,15 | 69,73 | 99,98 |
| 16 | 45,25 | 21,72 | 5,93 | 13,72 | 3,88 | 99,87 | 30,15 | 69,73 | 99,98 |
| 17 | 41,14 | 19,75 | 5,39 | 12,48 | 3,52 | 99,87 | 30,15 | 69,73 | 99,98 |
| 18 | 39,18 | 18,03 | 5,52 | 12,51 | 3,12 | 73,56 | 22,54 | 51,02 | 99,95 |
| 19 | 54,48 | 29,72 | 5,63 | 13,23 | 5,91 | 99,98 | 29,84 | 70,15 | 99,98 |
| 20 | 35,14 | 16,14 | 5,03 | 11,11 | 2,86 | 55,17 | 17,21 | 37,96 | 99,99 |

## Comparativa de performance con librerías similares

Dentro del ecosistema de Rust existe una librería con funcionalidades similares y otra que aunque no provee la funcionalidad es posible emularla al costo de exponencialmente más código por parte del modelador que si usara esta o la otra librería. Dichas librerías son Desim y Simrs (Preguntar: Desim y Simrs deberían haber sido mencionadas en el capítulo de Antecedentes ¿debería hacer una pequeña introducción aquí o simplemente las menciono y dejo una explicación más detallada de ambas librerías en aquella sección?)

Este experimento consiste en comparar el rendimiento de la librería con otra similar en el lenguaje, se compararán los resultados obtenidos y las conclusiones obtenidas.

### Único Productor – Único Consumidor

Para este experimento se realizó un modelo simple de un único productor y consumidor, ambos modifican un contador, el productor agrega unidades al contador y el consumidor las elimina.

Condiciones del experimento: Se simulan 50000 segundos, el productor genera 1 unidad cada 2 segundos con un máximo de 15 unidades, si la cantidad actual es igual o superior a este limite el productor dejara de producir hasta ser activado nuevamente. El consumidor por su parte elimina 8 unidades cada 8 segundos y si no encuentra suficientes unidades para eliminar dejara de consumir hasta ser activado nuevamente. Para evitar el *deadlock* de que ambas entidades dejen de trabajar antes de realizar cualquier acción ambos el productor y consumidor revisaran si el otro se encuentra desactivado y enviaran un mensaje para activarlo.

Los resultados obtenidos fueron los siguientes

[Insertar los V*iolín Plot* de los resultados V1 de los benchmark]

Sin embargo, al estudiar el código de desim se descubrió que se podía mejorar el rendimiento cambiando el *HashMap<usize, (BoxedComponent, ComponentState)>* utilizado en *State* y *Container* por un *Vec<Option<(BoxedComponent, ComponentState)>>* el *usize* que asocia un componente a una id pasa a ser el índice en donde se encuentra el componente en el vector.

[Insertar los *Violin Plot* de los resultados V2 de los benchmark]

Como consecuencia de cambiar el *HashMap* por un *Vec* no es posible eliminar un componente del vector ya que invalidaría las *Key* emitidas previo a la eliminación y tampoco es posible actualizar los valores de las *Key* al nuevo índice sin implementar algún tipo de sincronización lo cual eliminaría cualquier ganancia de rendimiento obtenida, sin embargo es posible eliminar el componente sin comprometer el estado del *Vec*, cuando un generador es agregado al *Vec* este es guardado en la variante *Some* del *Option* y cuando se desee eliminar un componente, por que ya completo su ejecución y reanudar un generador completado causara que el programa haga un *crash*, se toma el generador del Some dejando un *None* en su lugar y se elimina el generador de esta forma se recupera la memoria usada por el generador sin embargo, se siguen haciendo *leak* el espacio ocupado por el *Option.*

Observaciones: La cantidad de código para hacer el modelo en las 3 librerías es más o menos similar, las tres librerías proveen un nivel similar de abstracción. Desim gana en performance, seguidos por Simrs y esta librería.

### Múltiples Productores – Múltiples Consumidores

Sasadasda

# Anexos

[Por Confirmar: Eliminar experimento del motor de búsqueda -> Implica eliminar estos anexos, de todas formas, estos anexos probablemente estén mal así que igual aunque no se eliminen probablemente deban ser modificados]

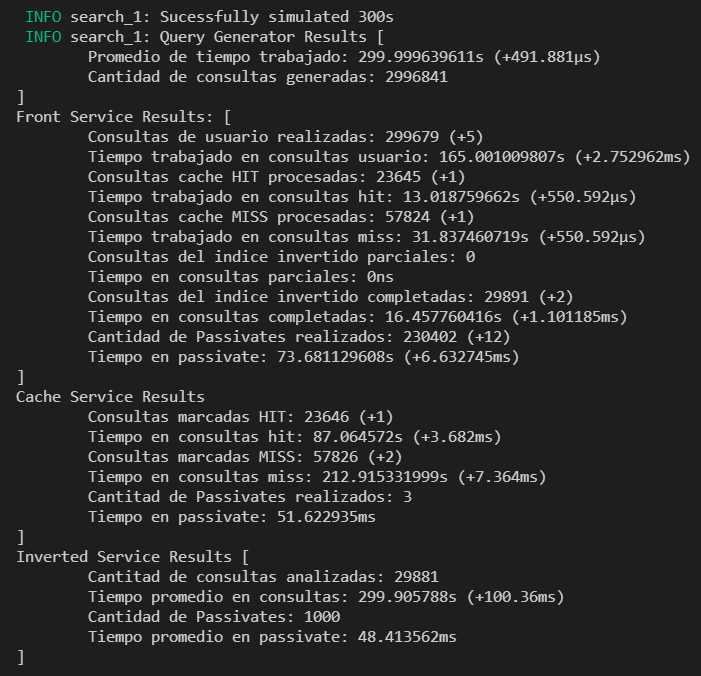


Figura 3.1: Resultado de ejecutar el modelo de motor de búsqueda con la configuración ID = 1 de la Tabla 2.8

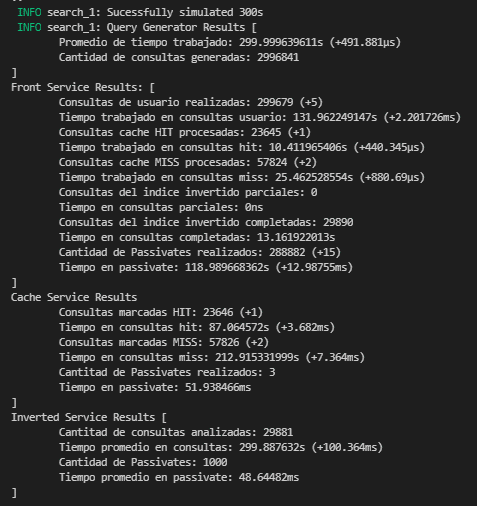


Figura 3.2: Resultado de ejecutar el modelo de motor de búsqueda con la configuración ID = 2 de la Tabla 2.8

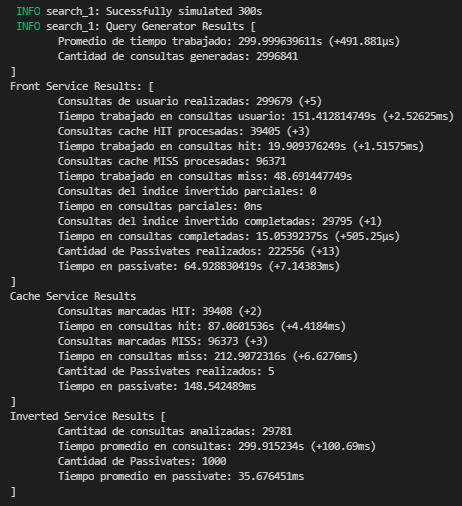


Figura 3.3: Resultado de ejecutar el modelo de motor de búsqueda con la configuración ID = 3 de la Tabla 2.8

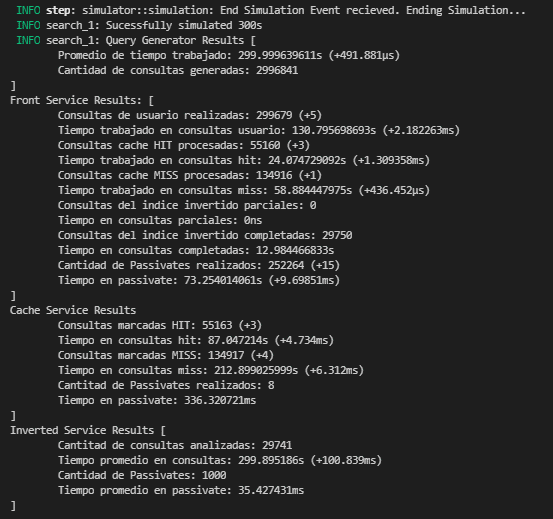


Figura 3.4: Resultado de ejecutar el modelo de motor de búsqueda con la configuración ID = 4 de la Tabla 2.8

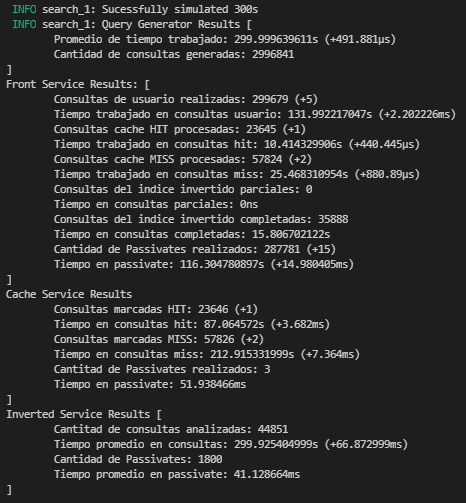


Figura 3.5: Resultado de ejecutar el modelo de motor de búsqueda con la configuración ID = 5 de la Tabla 2.8

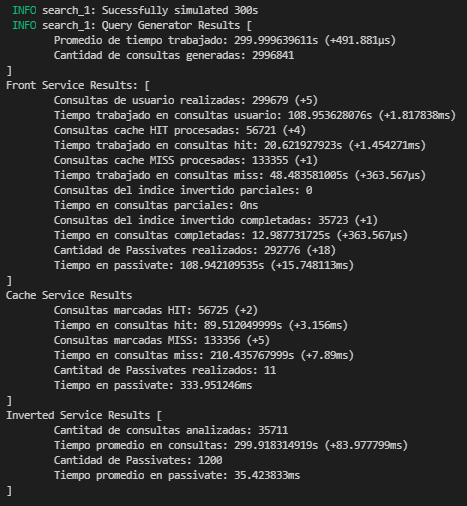


Figura 3.6: Resultado de ejecutar el modelo de motor de búsqueda con la configuración ID = 6 de la Tabla 2.8

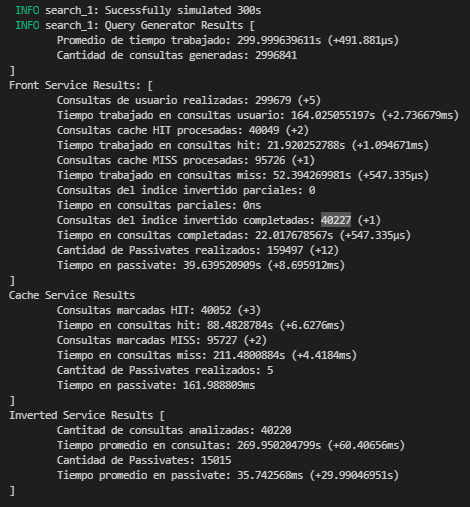


Figura 3.7: Resultado de ejecutar el modelo de motor de búsqueda con la configuración ID = 7 de la Tabla 2.8

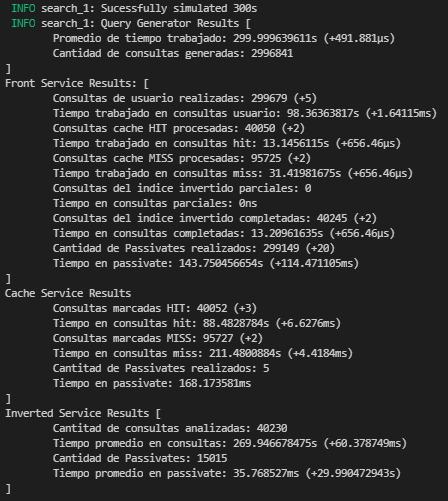


Figura 3.8: Resultado de ejecutar el modelo de motor de búsqueda con la configuración ID = 8 de la Tabla 2.8

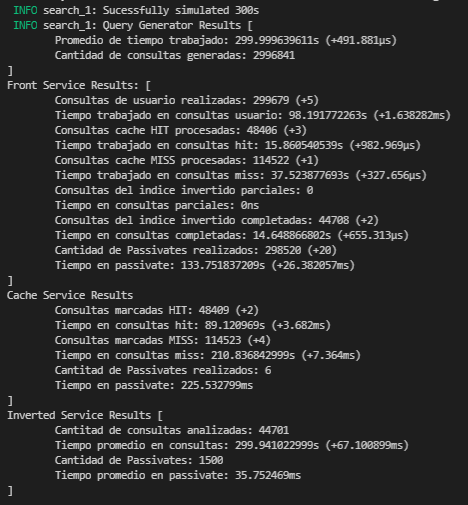


Figura 3.9: Resultado de ejecutar el modelo de motor de búsqueda con la configuración ID = 9 de la Tabla 2.8



Figura 3.10: Resultado de ejecutar el modelo de motor de búsqueda con la configuración ID = 10 de la Tabla 2.8

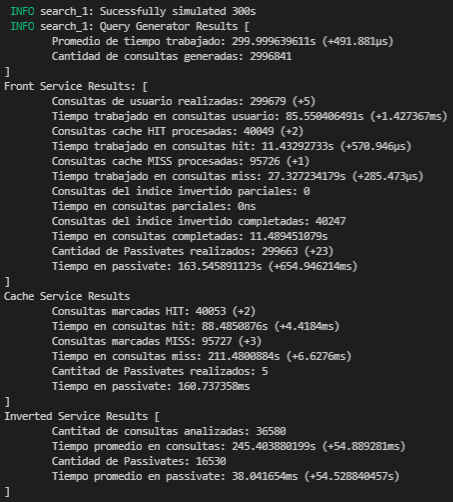


Figura 3.11: Resultado de ejecutar el modelo de motor de búsqueda con la configuración ID = 11 de la Tabla 2.8

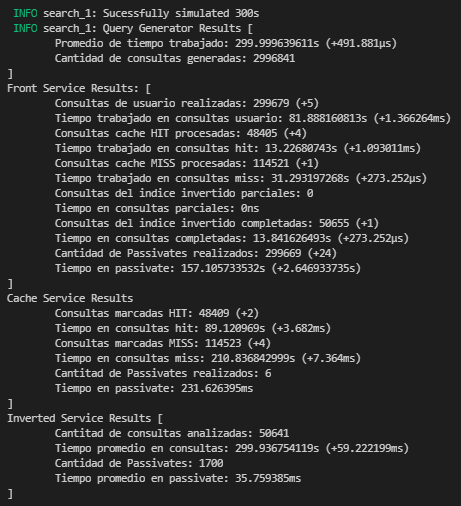


Figura 3.12: Resultado de ejecutar el modelo de motor de búsqueda con la configuración ID = 12 de la Tabla 2.8

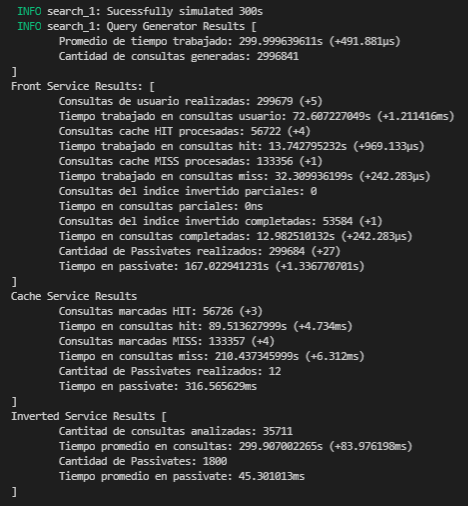


Figura 3.13: Resultado de ejecutar el modelo de motor de búsqueda con la configuración ID = 13 de la Tabla 2.8

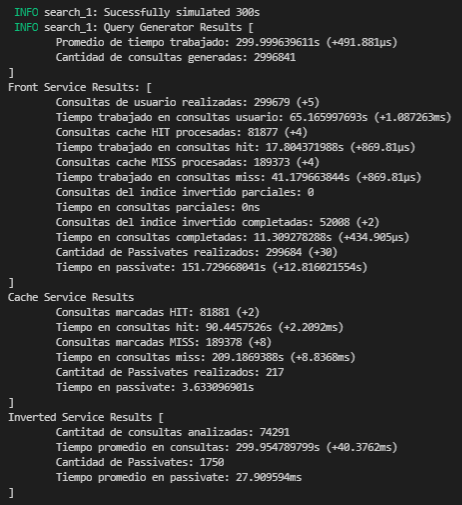


Figura 3.14: Resultado de ejecutar el modelo de motor de búsqueda con la configuración ID = 14 de la Tabla 2.8

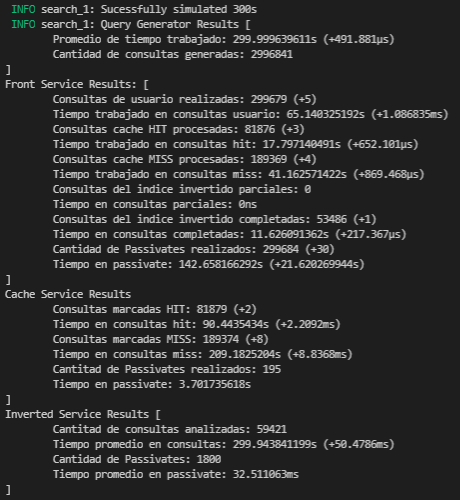


Figura 3.15: Resultado de ejecutar el modelo de motor de búsqueda con la configuración ID = 15 de la Tabla 2.8

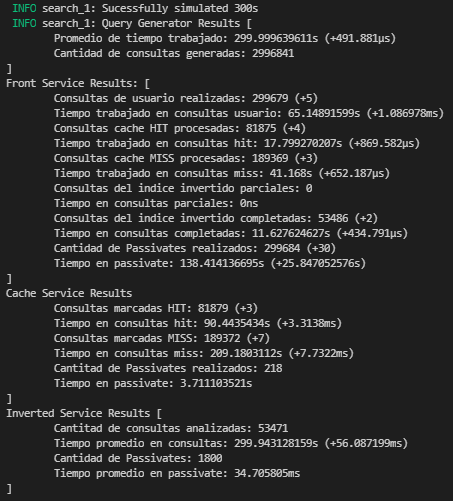


Figura 3.16: Resultado de ejecutar el modelo de motor de búsqueda con la configuración ID = 16 de la Tabla 2.8

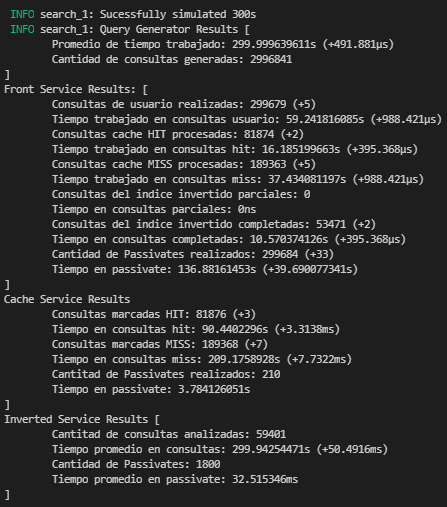


Figura 3.17: Resultado de ejecutar el modelo de motor de búsqueda con la configuración ID = 17 de la Tabla 2.8

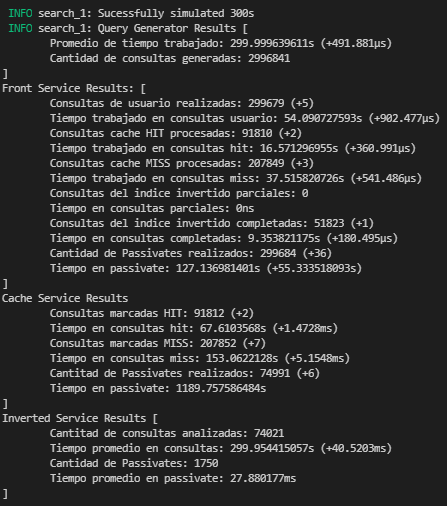


Figura 3.18: Resultado de ejecutar el modelo de motor de búsqueda con la configuración ID = 18 de la Tabla 2.8

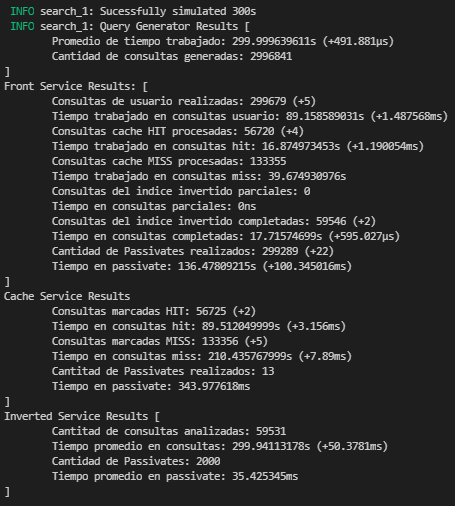


Figura 3.19: Resultado de ejecutar el modelo de motor de búsqueda con la configuración ID = 19 de la Tabla 2.8

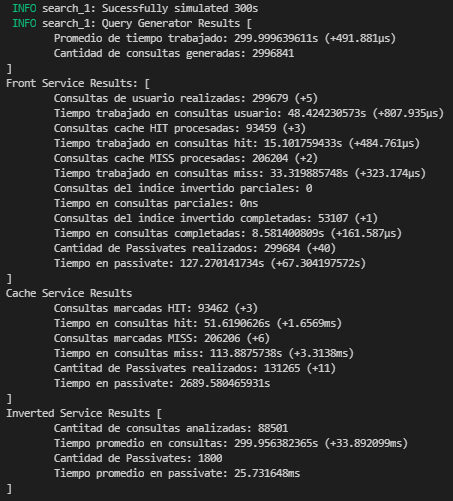


Figura 3.20: Resultado de ejecutar el modelo de motor de búsqueda con la configuración ID = 20 de la Tabla 2.8