PORTADA

ÍNDICE

1 RESUMEN

*¿En qué ha consistido el trabajo?  
¿Qué se ha intentado conseguir?  
¿Por qué tiene sentido el trabajo?*

2 INTRODUCCIÓN Y OBJETIVOS

2.1 Introducción

*Introducción a java 21, Project loom y los virtual threads ¿por qué suponen una revolución los virtual threads? ¿qué había antes de los VT? Esencialmente planteamiento del problema*

Introducción: El pasado septiembre de 2022, se introdujo una nueva herramienta en Java 19 los llamados *Virtual Threads,* o hilos virtuales, como *preview feature*. A mediados de 2023, los Virtual Threads se incluyeron en el lanzamiento de java 21. Los Virtual Threads tienen origen en el Proyecto Loom, una iniciativa open-source cuyo objetivo es conseguir un modelo de concurrencia ligero y de alto rendimiento para el lenguaje de programación Java.

Los hilos de plataforma, o hilos del sistema operativo, han sido, durante mucho tiempo, la base sobre la que se han construido la gran mayoría de aplicaciones Java. Cada declaración en cada método se ejecuta dentro de un hilo, y como Java el multi-hilo, múltiples hilos de ejecución suceden al mismo tiempo. Generalmente, las aplicaciones web gestionan múltiples usuarios de manera concurrente. Por lo que tiene sentido dedicar un hilo a cada petición o usuario. Este enfoque de asignar un hilo por solicitud es sencillo de comprender, implementar, depurar y analizar, ya que utiliza la unidad de concurrencia de Java, el hilo de plataforma, para representar la unidad de concurrencia de la aplicación. Este enfoque está limitado por el número de hilos de plataforma del sistema operativo. Al ser un número limitado, las aplicaciones tienen una capacidad limitada de responder peticiones de usuario simultáneas sin que su rendimiento se vea afectado.

A parte de la limitación del número de hilos de plataforma disponibles, el estilo de un hilo por petición presenta otro problema potencial. En muchos casos, las aplicaciones web dependen de operaciones de entrada-salida durante su ejecución, ya sea uso de bases de datos, servicios de mensajería o comunicación con otras aplicaciones. Las aplicaciones que necesitan de muchas operaciones de entrada-salida bloquean el hilo de plataforma en el que se están ejecutando. De esta forma, el procesador se encuentra ocioso durante largos periodos de tiempo.

De este problema nacieron los modelos de programación concurrente, que buscaban exprimir el uso del hardware al máximo. Para conseguir su objetivo, estos modelos querían limitar los tiempos de espera del procesador durante las operaciones de entrada salida, y dedicar ese tiempo a responder nuevas peticiones. Este modelo en el que una misma petición puede ser ejecutada por distintos hilos durante su ciclo de vida presenta dos grandes problemas. En primer lugar, los cambios de contexto, en los que se guardan los estados de los hilos y se comienzan a ejecutar otros, son operaciones costosas que requieren tiempo de procesamiento no dedicado a responder peticiones de usuario. En segundo lugar, la implementación de estos modelos de programación implica un aumento significativo en la complejidad del código y mayor dificultad a la hora de depurar posibles errores en el código.

Los hilos virtuales, nacen con la intención de resolver el problema de la limitación en el número de hilos de plataforma disponibles, sin renunciar a la comodidad del estilo de un hilo por petición. Como las aplicaciones Java no se ejecutan directamente sobre el sistema operativo, sino sobre la Máquina Virtual de Java, o JVM, Java ha podido romper la relación 1:1 que existía entre los hilos de ejecución y los hilos de plataforma. Se puede decir que un hilo virtual es una instancia de java.lang.Thread que no está atado a un hilo de plataforma en concreto.

De esta forma, se puede mantener el estilo de un hilo por petición usando hilos virtuales. Sin embargo, los hilos virtuales solo consumen recursos de los hilos de plataforma en los momentos en los que haya que realizar operaciones que usen el procesador. Cuando el código que se ejecuta en un hilo virtual hace una operación de entrada/salida bloqueante (usando la API de Java), automáticamente se suspende la ejecución de hilo virtual hasta que se pueda reanudar más adelante. De cara a los desarrolladores, los hilos virtuales son hilos baratos de crear y virtualmente infinitos.

2.2 Objetivos

*El objetivo del proyecto es analizar la complejidad de cara al uso de los VT con diversas tecnologías y entender como afecta su uso al rendimiento de aplicaciones web java.*

3 TECNOLOGÍAS

3.1 Frameworks

*Spring Boot y Quarkus*

3.2 Bases de datos

*Mongodb y mysql*

3.3 Modelos de programación

*imperativa y reactiva*3.4 Virtual Threads

*Casos críticos como el del bloque synchronized*

*¿En que casos tienen sentido y en cuales no?*

*¿Cómo se usan en Quarkus y en SB?*

*En Quarkus no son compatibles con las librerías reactivas: por qué  
¿Qué framework parece más preparado para los VT?*

*… bibliografía*

4 TEST DE CARGA

*¿Qué tecnologías se han usado para el desarrollo de los test de carga?*

*¿En que consistían los test de carga y por qué?*

*¿Sobre que infraestructura se han ejecutado los test de carga? Docker limitado a 2 cores y 500MB*

5 ANÁLISIS DE LOS RESULTADOS

5.1 Sin VT vs con VT

5.2 Imperativo vs Reactivo

5.3 Spring Boot vs Quarkus

6 CONLUSIONES Y TRABAJOS FUTUROS

6.1 Conclusiones

*¿En qué casos tienen sentido (y en cuáles no) el uso de los VT?  
¿Por qué a veces el uso de los VT empeora el rendimiento?  
¿Cuál es la mejor combinación de framework-arquitectura de bd-modelo de programación y uso de vt?  
¿Los vt merecen la pena para lo que cuesta implementarlos?*

6.2 Trabajos futuros

*Implementación de otros endpoints en el que la app haga de proxy y se bloquee de manera artificial para comprobar el rendimiento de los VT en ese caso.  
Uso de otras bases de datos/frameworks*

*Comparar uso de VT contra modelos de programación concurrente “clásicos”*

7 BIBLIOGRAFÍA