Санкт-Петербургский Политехнический Университет Петра Великого

Институт Компьютерных наук и технологий

Кафедра компьютерных систем и программных технологий

Отчет по созданию сетевому API

Предмет: Высоконагруженные вычисления

Студент: Мбеле Оссийи Люк П.

Гр. № 3540901/01501

Санкт-Петербург 2021

# Цель работы

Целью работы является создание сетевого API

# Задачи

1. В своём Java package ru.mail.polis.service.<username> реализовать интерфейс Service и поддерживать следующий HTTP REST API протокол:

* **HTTP GET /v0/entity?id=<ID>** -- получить данные по ключу <ID>. Возвращает 200 OK и данные или 404 Not Found.
* **HTTP PUT /v0/entity?id=<ID>** -- создать/перезаписать (upsert) данные по ключу <ID>. Возвращает 201 Created.
* **HTTP DELETE /v0/entity?id=<ID>** -- удалить данные по ключу <ID>. Возвращает 202 Accepted.
* Возвращайте реализацию интерфейса в ServiceFactory.

1. Провести нагрузочное тестирование с помощью wrk в одно соединение:

* **PUT** запросами на стабильной нагрузке (wrk должен обеспечивать заданный с помощью -R rate запросов)
* **GET** запросами на стабильной нагрузке по наполненной БД

1. Отпрофилировать приложение (CPU и alloc) под PUT и GET нагрузкой с помощью async-profiler. Приложите SVG-файлы FlameGraph cpu/alloc для PUT/GET нагрузки.

Структура проекта

В ходе разработки данной реализации был выбран язык программирования Java версия 11 и библиотека one-nio для создания высокопроизводительных Java-серверов.

Работа была выполнена на виртуальной машине под ОС Kali Linux 2021.1.

Были установлены утилиты wrk2 для проведения нагрузочного тестирования и async-profiler для отпрофилирования.

В качестве среды разработки использовалась IntelliJ Idea.

# Проведение нагрузочного тестирования с помощью wrk в одно соединение

Для проведения нагрузочного тестирования были созданы скрипты (lua) для обеих методов и выбраны следующие параметры :

Text

Description automatically generated -t (thread) : 1, -c (connection) : 1, -d (duration) : 5m -R (Rate): 2000

A picture containing text, electronics

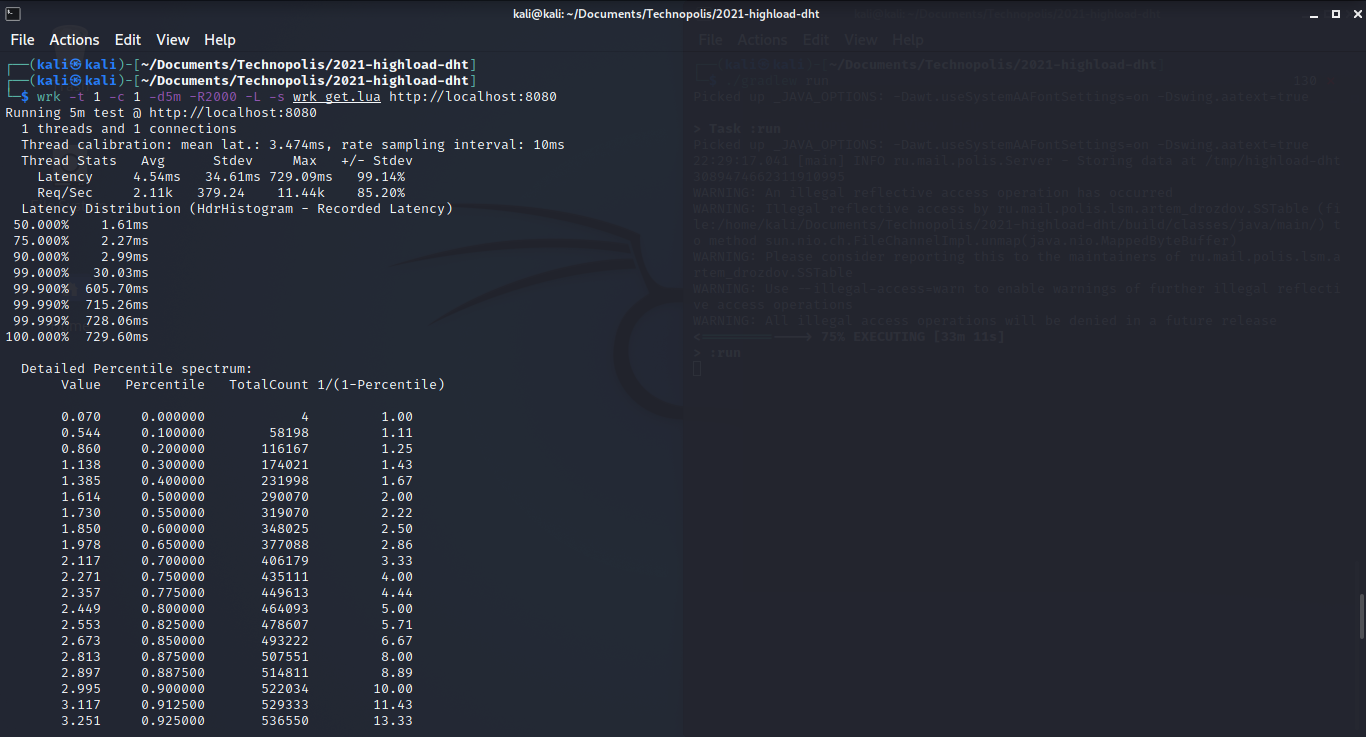
Description automatically generatedText

Description automatically generated *Рис 1 : PUT запросами на стабильной нагрузке*

Приведенное выше используется 1 поток и 1 соединение в течение 5 минут для метода *PUT*. Wrk2 смогло создать *599995* запросов за 5 минут и передать *38.34* МБ. В среднем получаем *1999.99* запросов в секунду, и передача данных в секунду : *130.86* KB.

Насчет задержки получили среднее значение : *4.40* ms. В нашем случае среднее значение задержки не очень велико. За 50% выполняемых запросов у нас было задержка 1.76 ms и за 100% 600 ms. Отмечаем, что, когда почти все запросы выполняются (99%) задержка очень быстро растет.

Req/sec : 2.00K представляет среднее число запросов, полученных за 1 поток



A picture containing text, electronics, computer

Description automatically generatedGraphical user interface, text

Description automatically generated*Рис 2 : GET запросами на стабильной нагрузке*

Приведенное выше используется 1 поток и 1 соединение в течение 5 минут для метода GET на наполненной базе данных. Wrk смогло создать *599999* запроса за *5* минут и получить *38.34* МБ как у метода *PUT*. Получили одинаковые результат как у метода PUT насчет запросов в секунду, и передачи данных в секунду.

Насчет задержки получили среднее значение : 4.50 ms. Среднее значение задержки почти как у метода PUT. Задержка при загрузке страницы с ресурсами оптимально, когда она составляет около двух секунд или меньше.

За 50% выполняемых запросов у нас было задержка 1.61 ms и за 100% 729.60 ms. Отмечаем, что, когда почти все запросы выполняются (99%) задержка очень быстро растет.

Req/sec : 2.11K представляет среднее число запросов, полученных за 1 поток

В итоге результаты представлены с помощью утилита wrk2 для нагрузочного тестирования для методов PUT и GET практически совпадают, особых различий нет, почти ***2000*** запросов/с были выполнены.

# Отпрофилирование приложения (CPU и alloc) под PUT и GET нагрузкой с помощью async-profiler

## Анализ производительности процессора

Chart

Description automatically generated with medium confidence *Рис 3 : График использования процессора методом PUT*

Text

Description automatically generated

*Рис 4 : Применение async-profiler (метод GET (аналогично сделано для метода PUT)*

Сначала мы проанализировали использование ЦП в течение 10 секунд, указывая PID нашего сервера.

Результирующий график пламени показывает код Java зеленым цветом, C++ красным, а пути системного кода – оранжевым.

На этом рисунке можно видеть, что метод *SelectorThread.run* из библиотеки **one-nio** использует больше всего ЦП (100%) среди вызовов основного метода. На нем надо аккуратно обратить внимание поскольку они прослушивают входящие сетевые подключения и записывают данные по медленным каналам. Они выполняют весь выбор объектов для принятых активных подключений.

Мы можем обобщить работу с кодом java (зеленым светом) на следующих этапах :

С помощью *SelectorThread* мы прослушиваем соединения, а затем наступает момент установления сеанса между клиентом и сервером, он занимает практически 90% ЦП, затем мы обрабатываем запросы.

Потом во время этапов отправки и записи ответов библиотекой *one.nio* (занимающей около 83% ЦП) выполняются соответственно и параллельно процессы чтения пути *v0/entity* (что приводит к вызову трех наших методов **GET, PUT и DELETE**) и внесения данных ( **PUT**) (1.46%). Можно вывести что наша реализация вообще не использует практически ЦП.

Наконец, в период выполнения функций write модуля Session one.nio (83% ЦП) параллельно происходит вставка данных в базу данных (DAO - 1.46%).

Далее продолжается процесс отправки данных с помощью метода send (занимающего значительную часть ЦП), а все этапы, связанные с обработкой пакетов в соответствии с моделью TCP / IP, занимают, в свою очередь, большую часть работы ЦП. Кроме того, можно также отметить очень низкое использование мусорного коллектора (0.73%). Производительность сборщика мусора определяется не количеством мертвых объектов, а скорее количеством живых. Чем больше объектов умирает, тем быстрее происходит сборка мусора.

Chart, bar chart

Description automatically generated*Рис 5 : График использования процессора методом GET*

Chart

Description automatically generated

*Рис 6 : Подробный график использования процессора методом GET*

У нас практически одинаковый последовательность событий как у метода **PUT** и такая же низкая загрузка ЦП по-нашему реализацию (5%), однако после проверки наличия ключа в базе данных выполняется непрерывный вызов методов *next* и *Peekingiterator.next*, что интерпретируется как существование нескольких ключей в нашем диапазоне ключей, установленном в методе ***GET*** из реализации. Выше мы отмечаем преобразование ключа в тип ByteBuffer, то что важно всякий раз, когда нам нужно выполнить быстрый низкоуровневый ввод-вывод, реализовать протокол TCP/IP или создать базу данных (СУБД).

Отмечаем также использование (низкое) *epoll\_wait* который ждет событий ввода/вывода в файловом дескрипторе *epoll*

## Анализ кучи памяти

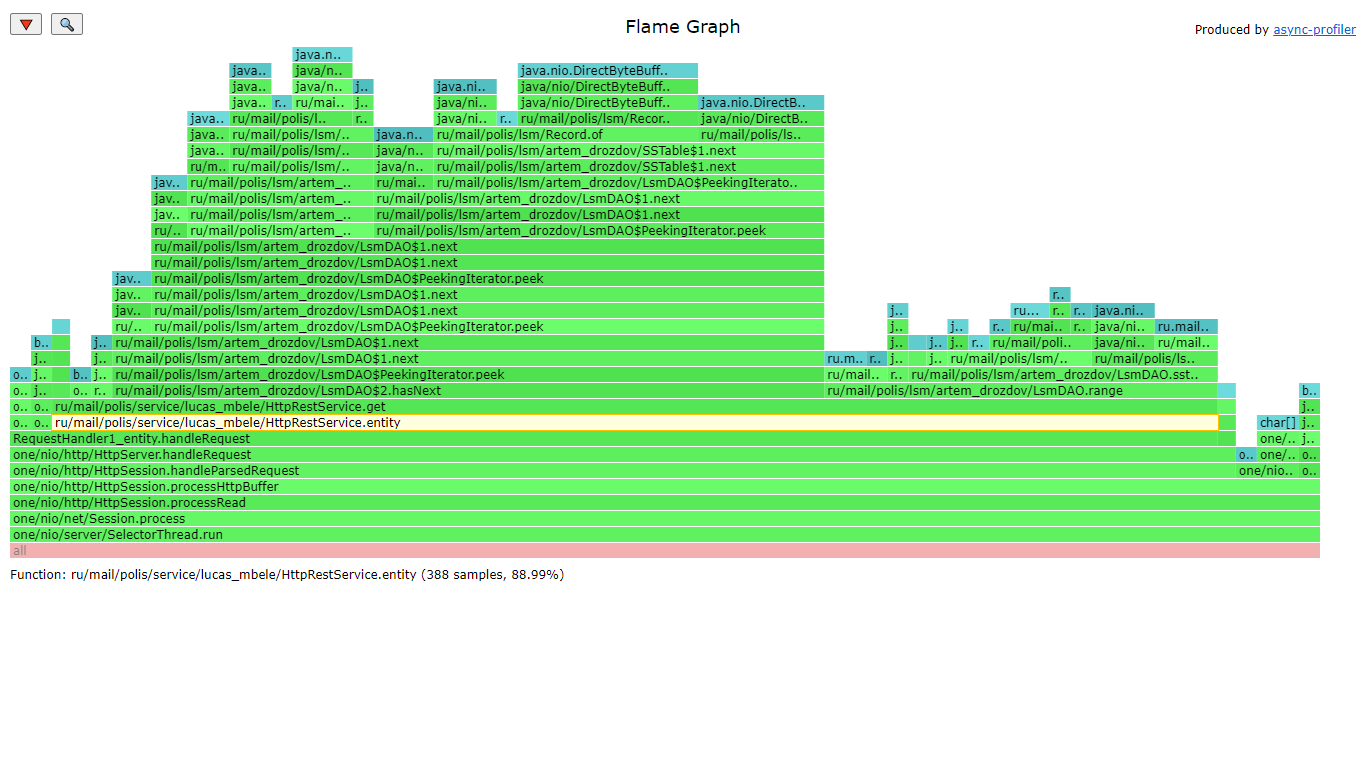
Graphical user interface

Description automatically generated

*Рис 7 : График выделения памяти методом PUT*

Процессы управления подключением (*SelectorThread*), сеансом (*HttpSession*), запросом (*HandleParseRequest*) занимают наибольшее место в памяти.

Метод *Put* нашей реализации занимает значительное место в памяти (34%). Это можно объяснить тем, что вызовы данного метода, выполняемые wrk, повторяются во время запросов вставки данных. Следует также отметить, что чтение в кодировке UTF-8 при преобразовании данных также занимает значительное место в памяти. В верхней части нашего графика мы можем отметить использование типов *byte, char, String, Response*, которые в основном соответствуют типам переменных, возвращаемых нашими функциями.



*Рис 7 : График выделения памяти методом GET*

Как у метода *PUT* процессы управления подключением (*SelectorThread*), сеансом (*HttpSession*), запросом (*HandleParseRequest*) занимают наибольшее место в памяти.

Но в отличии от метода *PUT,* здесь память чаще всего используется (88%) в процессе поиска ключей в заданном диапазоне ключей (LSMDAO.range и LsmDAO$2.hasNext, LsmDAO$PeekingIterator.peek, LsmDAO$1.next) и соответствующих значений, данных для этих ключей присутствующих в наполненной базе данных.