

ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH  
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN  
KHOA CÔNG NGHỆ PHẦN MỀM

NGUYỄN THANH HƯNG

KHÓA LUẬN TỐT NGHIỆP

XÂY DỰNG GIẢI PHÁP PHÂN PHỐI NỘI DUNG  
THÔNG QUA BỘ NHỚ ĐỆM

**Building a solution to enable disseminated content  
through caching**

CỬ NHÂN NGÀNH KỸ THUẬT PHẦN MỀM

TP. HỒ CHÍ MINH, 2025

**ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH**  
**TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN**  
**KHOA CÔNG NGHỆ PHẦN MỀM**

**NGUYỄN THANH HƯNG – 19521574**

**KHÓA LUẬN TỐT NGHIỆP**  
**XÂY DỰNG GIẢI PHÁP PHÂN PHỐI NỘI DUNG**  
**THÔNG QUA BỘ NHỚ ĐỆM**

**Building a solution to enable disseminated content**  
**through caching**

**CỬ NHÂN NGÀNH KỸ THUẬT PHẦN MỀM**

**GIẢNG VIÊN HƯỚNG DẪN**  
**THS. ĐINH NGUYỄN ANH DŨNG**

**TP. HỒ CHÍ MINH, 2025**

## **THÔNG TIN HỘI ĐỒNG CHẤM KHÓA LUẬN TỐT NGHIỆP**

Hội đồng chấm khóa luận tốt nghiệp, thành lập theo Quyết định số 79/QĐ-ĐHCNTT ngày 22 tháng 01 năm 2025 của Hiệu trưởng Trường Đại học Công nghệ Thông tin.

## LỜI CẢM ƠN

Lời đầu tiên, em xin chân thành cảm ơn Trường Đại Học Công Nghệ Thông Tin - Đại học Quốc gia Thành phố Hồ Chí Minh, cùng các thầy cô khoa Công nghệ phần mềm đã luôn giúp đỡ, tạo điều kiện cơ hội trong suốt quá trình học tập và nghiên cứu.

Em xin chân thành cảm ơn đến thầy ThS. Đinh Nguyễn Anh Dũng đã đồng hành, đưa ra những chỉ dẫn, cũng như ý kiến và kinh nghiệm của mình, để em hoàn thành đề tài khoá luận.

Trong suốt quá trình, em đã tìm hiểu thêm được nhiều kiến thức mới, qua đó có cơ hội học tập rèn luyện thêm. Tuy nhiên vẫn còn nhiều khó khăn và hạn chế trong việc tìm hiểu nên không thể tránh khỏi thiếu sót. Do đó, em mong nhận được sự góp ý, nhận xét từ hội đồng để hoàn thiện hơn.

Em xin chân thành cảm ơn!

Thành phố Hồ Chí Minh, ngày 08 tháng 02 năm 2025

Sinh viên thực hiện

Nguyễn Thanh Hưng

## MỤC LỤC

Chương 1.	MỞ ĐẦU .....	2
1.1	Đặt vấn đề .....	2
1.2	Hiện trạng công nghệ .....	4
1.3	Mục tiêu .....	6
1.4	Đối tượng và phạm vi nghiên cứu .....	6
Chương 2.	CƠ SỞ LÝ THUYẾT.....	7
2.1	Ngôn ngữ Java .....	8
2.2	Java Off-Heap và NIO .....	9
2.2.1	Vùng nhớ Off-Heap .....	9
2.2.2	Java Unsafe .....	10
2.2.3	FileChannel và MappedByteBuffer .....	10
2.2.4	Zero copy .....	12
2.3	VLQ (variable-length quantity) .....	15
2.3.1	VLQ. ....	15
2.3.2	LEB128 .....	15
2.3.3	Xử lý số nguyên âm .....	19
2.4	Thư viện Kryo.....	21
2.5	Thư viện Netty .....	23
2.6	Hash table .....	28
2.6.1	Hàm băm.....	28
2.6.2	Chaining.....	30
2.6.3	Open-addressing (OA).....	31

2.6.4	So sánh Chaining và Open-addressing .....	32
2.7	SLAB Allocation.....	33
2.8	Pull-based và push-based.....	37
Chương 3.	PHÂN TÍCH YÊU CẦU .....	39
3.1	Tính năng .....	39
3.2	Danh sách các Actor .....	39
3.3	Sơ đồ Use-case.....	40
3.4	Danh sách các Use-case .....	41
3.4.1	Use-case phát hành dữ liệu mới.....	41
3.4.2	Use-case rollback về version cũ.....	43
3.4.3	Use-case truy vấn dữ liệu.....	44
3.4.4	Use-case cập nhật trạng thái theo Producer .....	45
Chương 4.	THIẾT KẾ.....	46
4.1	Thiết kế kiến trúc .....	47
4.1.1	Producer - Consumer .....	47
4.2	Artifact.....	52
4.2.1	Tập Header.....	52
4.2.2	Tập Delta, ReverseDelta .....	53
4.2.3	Cách Rollback.....	54
4.3	Java Off-heap .....	56
4.4	Java NIO .....	57
4.5	LEB128 và Kryo .....	57
4.6	Netty.....	61

4.7 Hash-Table và SLAB .....	62
4.7.1 Lưu trữ dữ liệu .....	62
4.7.2 Đồng bộ trong đa luồng .....	66
Chương 5. TRIỂN KHAI ỨNG DỤNG .....	68
5.1 Producer .....	68
5.2 Consumer .....	69
Chương 6. Đánh giá giải pháp .....	70
6.1 Chuẩn bị môi trường .....	70
6.2 Triển khai .....	71
6.2.1 Triển khai NetflixHollow .....	71
6.2.2 Triển khai Cobra .....	71
6.3 Đánh giá hiệu năng .....	72
6.3.1 Sử dụng bộ nhớ .....	72
6.3.2 Sử dụng CPU .....	74
6.3.3 Hiệu năng .....	75
6.4 Đánh giá về tính linh hoạt .....	77
Chương 7. KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN .....	78
7.1 Kết luận .....	78
7.2 Hướng phát triển tương lai .....	78
Chương 8. TÀI LIỆU THAM KHẢO .....	79

## DANH MỤC HÌNH

Hình 1.1 Minh hoạ kiến trúc Adobe Manager .....	5
Hình 2.1 Quá trình biên dịch của Java.....	8
Hình 2.2 Bộ nhớ Heap và Off-heap trong Java .....	9
Hình 2.3 Sơ đồ tuần tự truyền file (không dùng zero-copy).....	12
Hình 2.4 Sơ đồ tuần tự truyền file (dùng zero-copy).....	13
Hình 2.5 Mô tả LEB128 biểu diễn số “1” .....	16
Hình 2.6 Mô tả cách LEB128 biểu diễn số “150” .....	17
Hình 2.7 Mô tả cấu trúc của Java S/D .....	21
Hình 2.8 Mô tả cấu trúc Kryo S/D.....	22
Hình 2.9 So sánh không gian lưu trữ Java và Kryo S/D.....	23
Hình 2.10 Minh hoạ Blocking I/O .....	24
Hình 2.11 Minh hoạ so sánh Blocking và Non-Blocking I/O.....	24
Hình 2.12 Minh hoạ Event-Loop trong Netty.....	26
Hình 2.13 Netty event-loop, Selector và Channel .....	26
Hình 2.14 Netty pipeline và handler .....	27
Hình 2.15 Mô tả Chaining Hash-Table .....	30
Hình 2.16 Minh hoạ phân mảnh bộ nhớ .....	33
Hình 2.17 Mô tả cấu trúc SLAB .....	34
Hình 2.18 Mô tả SLAB freelist.....	35
Hình 2.19 Minh hoạ SLAB re-balancing.....	36
Hình 2.20 Minh hoạ mô hình pull-based .....	37
Hình 2.21 Minh hoạ mô hình push-based.....	38
Hình 3.1 Sơ đồ Use-case của người dùng.....	40
Hình 3.2 Sơ đồ Use-case của Consumer.....	40
Hình 4.1 Các quá trình chính của phần mềm.....	46



Hình 4.2 Mô hình đơn giản của hệ thống .....	47
Hình 4.3 Sơ đồ tuần tự Producer – Consumer .....	48
Hình 4.4 Mô tả interface của các component liên quan cần triển khai.....	49
Hình 4.5 sơ đồ hoạt động Consumer .....	51
Hình 4.6 Minh hoạ rollback.....	55
Hình 4.7 Mô tả xử lý ghi Schema và Artifact.....	57
Hình 4.8 Mô tả xử lý đọc Artifact và Schema .....	57
Hình 4.9 Ví dụ định dạng bản ghi trong tệp Delta.....	59
Hình 4.10 Netty's pipeline của Producer .....	61
Hình 4.11 Quan hệ Hash-Table và Memory .....	64
Hình 4.12 Sơ đồ SLAB re-balancing.....	65
Hình 4.13 Sơ đồ lớp của Hash-Table và SLAB trong quản lý bộ nhớ .....	66
Hình 4.14 ReentrantLock khi kiểm tra phiên bản.....	66
Hình 4.15 Minh hoạ ReadWriteLock.....	67
Hình 5.1 Ví dụ khởi chạy Producer trên EC2.....	69
Hình 6.1 Mô hình triển khai Netflix Hollow .....	71
Hình 6.2 Mô hình triển khai Cobra.....	72
Hình 6.3 Đồ thị tiêu thụ bộ nhớ của Consumer .....	73
Hình 6.4 Đồ thị tiêu thụ bộ nhớ của Producer .....	74
Hình 6.5 Đồ thị mức phần trăm sử dụng CPU.....	75
Hình 6.6 Kết quả hiệu năng của K6.....	76

## DANH MỤC BẢNG

Bảng 2.1 So sánh hiệu năng FileChannel và FileOutputStream bằng JMH.....	12
Bảng 2.2 So sánh kết quả khi dùng zero-copy.....	14
Bảng 2.3 Cấu trúc 1 octet trong Base128 .....	15
Bảng 2.4 So sánh biểu diễn thông thường và VLQ số nguyên không âm 32-bit	18
Bảng 2.5 Giá trị mã hoá ZigZag đối số nguyên 32-bit .....	19
Bảng 2.6 So sánh Chaining và OA Hash-Table .....	32
Bảng 3.1 Danh sách các Use-case .....	41
Bảng 3.2 Use-case phát hành phiên bản mới.....	41
Bảng 3.3 Use-case rollback version.....	43
Bảng 3.4 Use-case truy vấn dữ liệu .....	44
Bảng 3.5 Use-case cập nhật trạng thái theo Producer.....	45
Bảng 4.1 Ví dụ cấu trúc tệp Header.....	53
Bảng 4.2 Cấu trúc tệp Delta.....	54
Bảng 4.3 Cấu trúc tệp ReverseDelta.....	54
Bảng 6.1 Cấu hình AWS EC2.....	70

## DANH MỤC TỪ VIẾT TẮT

API	Application Programming Interface
AWS	Amazon Web Services
CPU	Central Processing Unit
CSV	Comma-Separated Values
GC	Garbage collector
I/O	Input/Output
JMH	Java Microbenchmark Harness
JNI	Java native interface
JSON	Javascript Object Notation
LEB128	Little Endian Base 128
NCHFs	Non-Cryptographic Hash Functions
NIO	New Input/Output
OA	Open-Addressing
OS	Operating system
RAM	Random Access Memory
UC	Use-case
VLQ	Variable-length quantity
VUS	Virtual Users
XML	eXtensible Markup Language

## TÓM TẮT KHOÁ LUẬN

Đề tài “Xây dựng giải pháp phân phối nội dung thông qua bộ nhớ đệm” nhằm mục tiêu phát triển một thư viện theo mô hình Producer – Consumer, hỗ trợ các hệ thống có kiến trúc phân tán chia sẻ một tập dữ liệu, lưu trữ ở bộ nhớ đệm và khôi phục dữ liệu.

Đề tài tập trung vào giải quyết các vấn đề xoay quanh lưu trữ bộ nhớ đệm bao gồm vấn đề về hiệu năng khi lưu trữ trên bộ nhớ trong (RAM), tối ưu không gian lưu trữ, và tính nhất quán dữ liệu giữa các máy khác nhau.

Sản phẩm của đề tài này là một thư viện bằng ngôn ngữ lập trình Java, dùng để tích hợp vào các dự án, phần mềm có yêu cầu về tính sẵn sàng cao của dữ liệu như thương mại điện tử, truyền hình, quản lý v.v..

Khoá luận áp dụng các phương pháp như phân tích về yêu cầu, tìm hiểu các cơ sở lý thuyết, và thực nghiệm. Báo cáo này gồm 6 phần chính:

Chương 1: Mở đầu, hiện trạng công nghệ, mục đích, và phạm vi nghiên cứu.

Chương 2: Tập trung vào trình bày các cơ sở lý thuyết.

Chương 3: Phân tích yêu cầu.

Chương 4: Thiết kế phần mềm.

Chương 5: Triển khai ứng dụng.

Chương 6: Đánh giá giải pháp.

Chương 7: Kết luận, hướng phát triển tiếp theo.

Chương 8: Tài liệu tham khảo.

# Chương 1. MỞ ĐẦU

## 1.1 Đặt vấn đề

Với sự phát triển của Internet và nhu cầu người dùng tăng cao, kiến trúc phân tán đang ngày càng trở nên phổ biến, một vấn đề thường gặp trong kiến trúc phân tán là lưu trữ bộ nhớ đệm. Bộ nhớ đệm (caching) là một kỹ thuật giúp tăng tốc độ truy vấn dữ liệu, thông thường hạ tầng ở vùng biên thay vì phải gọi dữ liệu từ một kho dữ liệu tập trung sẽ chủ động caching dữ liệu để giảm độ trễ. Có 2 hướng tiếp cận phổ biến như sau:

- Sử dụng các phần mềm mã nguồn mở (Redis, Memcached, v.v.) để lưu trữ và truy cập dữ liệu, đây là kỹ thuật lưu trữ tập trung từ xa.
- Chuyển đổi dữ liệu sang các định dạng khác (như JSON, XML, v.v.) và chuyển đến các máy khách để xử lý và lưu vào RAM.

Lựa chọn một trong hai hướng tiếp cận trên đều có những khó khăn riêng:

- Đối với việc sử dụng lưu trữ tập trung, mặc dù cho phép người dùng phát triển tập dữ liệu vô hạn, song lại có các hạn chế như:
  - Sẽ có độ trễ và giới hạn băng thông khi yêu cầu một đối tượng.
  - Kho dữ liệu tập trung từ xa dễ bị ảnh hưởng bởi các yếu tố bên ngoài như môi trường, mạng, và không đáng tin cậy bằng kho dữ liệu cục bộ.
- Đối với việc sử dụng lưu trữ cục bộ:
  - Việc ánh xạ một tệp tin sang RAM làm tăng chi phí, có thể giảm hiệu năng ứng dụng.
  - Phân vùng nhớ đệm (Heap) trong các ngôn ngữ lập trình có thể trở nên lớn, ảnh hưởng lên hoạt động của trình thu gom rác (GC) [1], từ đó gây ảnh hưởng hiệu năng của ứng dụng.
  - Yêu cầu phải truyền nhiều dữ liệu hơn khi dùng các định dạng như JSON, XML.

- Một giải pháp khác là kết hợp cả 2 hướng đi trên – lưu trữ dữ liệu được dùng thường xuyên trên RAM và lưu trữ tập trung các dữ liệu ít truy cập, hướng tiếp cận này cũng có một vài khó khăn:
  - Tính nhất quán và đồng bộ dữ liệu giữa bộ nhớ RAM và kho dữ liệu từ xa không đáng tin cậy, đặc biệt trong kiến trúc microservices.
  - Tính toán kích thước cho vùng nhớ để cân bằng giữa việc lưu trữ ở RAM và độ trễ khi truy vấn kho dữ liệu từ xa là khó khăn.
  - Khi một máy chạy mới hoặc khởi động lại, dữ liệu thường xuyên truy cập trên máy đó không khả dụng, gia tăng đột biến yêu cầu truy xuất kho dữ liệu từ xa.
  - Việc giám sát, gỡ lỗi khó khăn.
  - Không thể áp dụng với các trường hợp yêu cầu tỉ lệ cache-hit 100%.

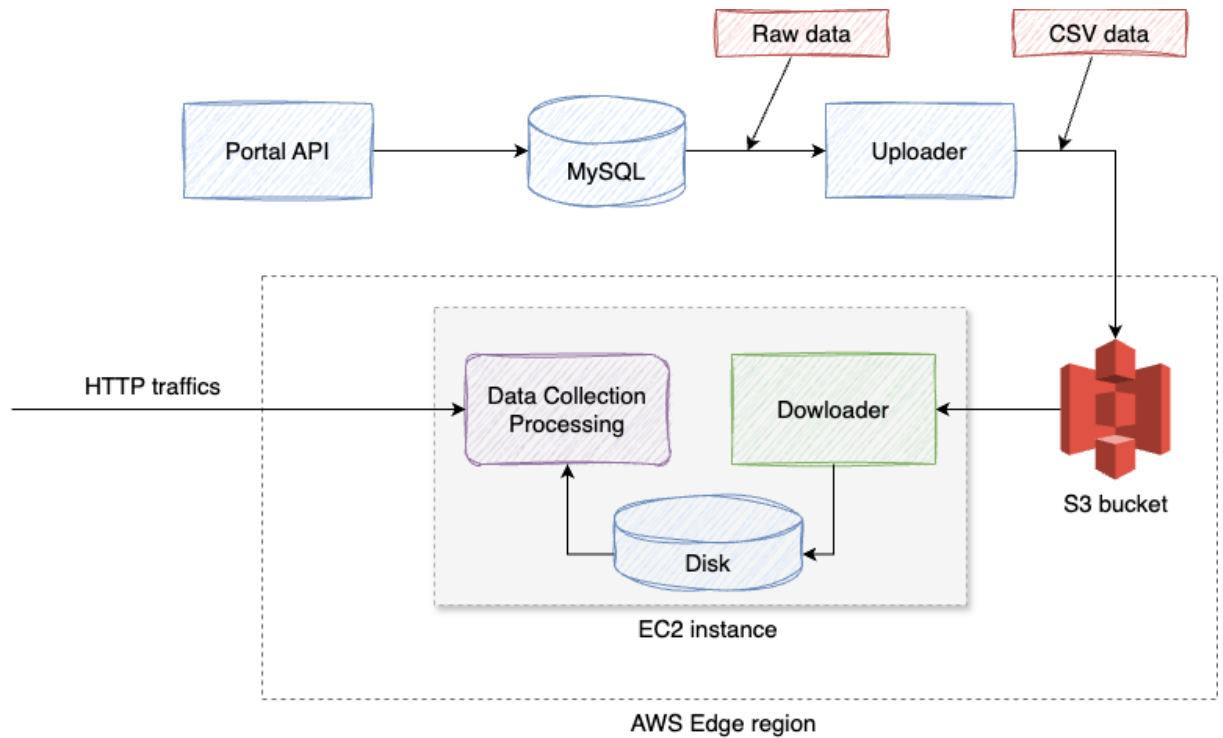
Ngoài ra, trong quá trình vận hành phần mềm đôi khi các nhóm kỹ sư có thể gặp các sự cố về dữ liệu như cập nhật nhầm 1 tập dữ liệu. Lấy ví dụ Tiki cần caching một tập dữ liệu cho sự kiện khuyến mãi ngày mai nhưng vì sự cố đã thực hiện việc cập nhật trước 1 ngày, trong trường hợp này Tiki cần một cơ chế để khôi phục lại dữ liệu được cache trước đó.

## 1.2 Hiện trạng công nghệ

Trong các hệ thống phân tán, Kafka là một giải pháp phổ biến trong việc truyền dữ liệu hoạt động dựa trên mô hình Producer – Consumer (nhà sản xuất - người tiêu dùng) [2], người dùng có thể nhập sự thay đổi dữ liệu và tự chủ động caching trên Consumer. Tuy nhiên, Kafka lại bị giới hạn về kích thước của mỗi gói tin (1 MB), điều này không phù hợp với yêu cầu thực tế khi các đối tượng cần cache có thể có kích thước lớn hơn.

Netflix Hollow (phát triển từ Zeno) là mã nguồn mở do Netflix phát triển, cho phép phân phối và caching dữ liệu ở Consumer, Producer sẽ gửi các tệp JSON đến Consumer để thực hiện cập nhật thay đổi. Nhược điểm là các thành phần kém linh động, Consumer giao tiếp trực tiếp với kho dữ liệu thay vì dùng interface, thêm vào đó dữ liệu được tuần tự hoá sang định dạng JSON làm tăng kích thước của một đối tượng trong tệp.

Trên thực tế, Adobe đã gặp một vấn đề tương tự trong quá trình vận hành nền tảng Adobe Manager [3]. Adobe Manager là một nền tảng có kiến trúc phân tán được triển khai trên hạ tầng AWS ở 8 khu vực khác nhau, Adobe Manager thực hiện việc lưu trữ metadata trong MySQL (standalone), tập dữ liệu này cần được chuyển đến các hạ tầng biên để phục vụ người. Adobe thực hiện trích xuất dữ liệu, chuyển sang định dạng CSV và lưu trữ ở AWS S3 (theo từng vùng).



Hình 1.1 Minh hoạ kiến trúc Adobe Manager

Kiến trúc này của Adobe Manager có một vài nhược điểm sau:

- Có độ trễ cao, tổng toàn bộ quá trình từ lúc dữ liệu được ghi vào MySQL đến khi được cache đến tất cả region mất khoảng 3 tiếng.
- Các tệp CSV được ánh xạ lên RAM mà không có cơ chế tối ưu có thể gây ảnh hưởng hiệu năng.
- Khi thêm mới các EC2 instance (bằng Kubernetes), tăng phí tổn khi triển khai thành phần Downloader (sidecar container).
- Gặp khó trong quá trình gỡ lỗi, xử lý và khôi phục dữ liệu.



### 1.3 Mục tiêu

Từ những vấn đề gặp trong thực tiễn được đề cập ở trên, nhóm em nhận thấy tính cần thiết của việc xây dựng một thư viện giúp phân phối dữ liệu hiệu quả, đặc biệt trong các hệ thống có kiến trúc phân tán. Mục tiêu của đề tài bao gồm:

- Thiết kế thư viện phân phối dữ liệu đáp ứng mô hình Producer – Consumer.
- Lưu trữ bộ nhớ đệm cục bộ bằng cặp khoá giá – giá trị (key – value).
- Tối ưu hoá tài nguyên như RAM, CPU.
- Dữ liệu có độ tin cậy cao, có khả năng khôi phục dữ liệu khi gặp sự cố.
- Cung cấp sự linh hoạt, người dùng có thể tự tích hợp, triển khai hệ thống riêng.

### 1.4 Đối tượng và phạm vi nghiên cứu

- Đối tượng:
  - Giao thức truyền tin (TCP) và cách tổ chức thông tin trong giao để truyền dữ liệu.
  - Các thuật toán và thư viện giúp lưu trữ dữ liệu trên RAM.
- Phạm vi nghiên cứu:
  - Phạm vi nội dung: tập trung phát triển thư viện theo mô hình Producer – Consumer chạy trên nền tảng JVM.
  - Phạm vi chức năng: cung cấp các tính năng cơ bản như:
    - Chính sửa dữ liệu ở Producer và truy xuất dữ liệu ở Consumer.
    - Khôi phục dữ liệu cũ.

## Chương 2. CƠ SỞ LÝ THUYẾT

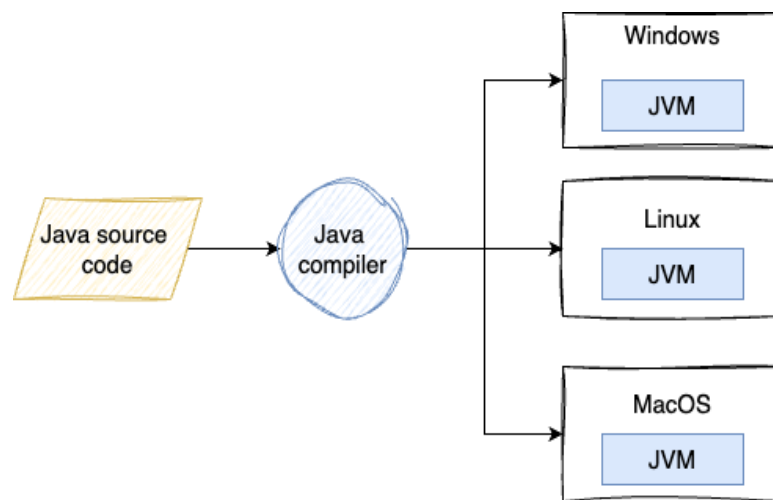
Dựa trên mục tiêu được đề ra, việc phát triển thư viện giúp phân phối và lưu trữ dữ liệu ở bộ nhớ đệm sẽ cần giải quyết các vấn đề sau:

- Tối ưu lưu trữ trên RAM, cần sử dụng những kỹ thuật để nén và tối ưu kích thước dữ liệu khi truyền qua Internet cũng như lưu trữ.
- Bên cạnh đó giảm thiểu chi phí hoạt động của GC.
- Chống phân mảnh bộ nhớ trên RAM.
- Cách tổ chức dữ liệu trong giao tiếp giữa Producer – Consumer.

## 2.1 Ngôn ngữ Java

Java là một ngôn ngữ lập trình hướng đối tượng, được phát triển bởi Sun Microsystem, Java đảm bảo tiêu chí quan trọng nhất trong quá trình phát triển của mình: viết một lần, dùng mọi nơi (Write once, run anywhere) [4].

Mã nguồn Java sẽ được biên dịch thành mã nhị phân (byte-code), sau đó được thông dịch sang mã máy bởi máy ảo JVM. Nhờ đặc tính này, Java không phụ thuộc quá nhiều vào hệ điều hành hay nền tảng [5].



Hình 2.1 Quá trình biên dịch của Java

## 2.2 Java Off-Heap và NIO

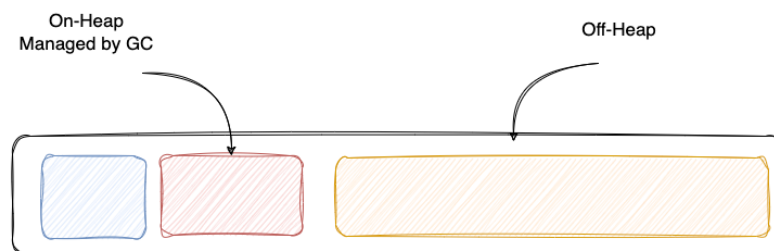
### 2.2.1 Vùng nhớ Off-Heap

Thông thường khi nhắc việc quản lý bộ nhớ của Java, ta sẽ nghĩ tới vùng nhớ Heap (On-Heap) và Stack [6]. Stack dùng để lưu các biến cục bộ trong một hàm, các giá trị nguyên thủy (primitive) và loại tham chiếu đến các đối tượng trong Heap. Heap sẽ là phân vùng chứa tất cả các loại dữ liệu còn lại.

Trong Java, bộ nhớ Heap được quản lý bởi trình thu gom rác (GC), khi GC hoạt động tất cả các tiến trình trong chương trình sẽ tạm dừng [1], vì lý do này nếu lưu tập dữ liệu trên bộ nhớ Heap thì hiệu năng sẽ bị ảnh hưởng, do chương trình sẽ thường xuyên bị dừng bởi GC.

Để giảm sự ảnh hưởng của GC tới hiệu năng, nhóm đã dùng giải pháp cấp phát bộ nhớ Off-Heap (được quản lý bởi OS). Không như bộ nhớ Heap, bộ nhớ Off-Heap lớn hơn On-Heap và không bị quản lý bởi GC, hiểu đơn giản Off-Heap chính là RAM và được quản lý bởi OS. Trong Java có 3 cách để thực hiện điều này:

- Nhúng code C/C++/Rust bằng Java Native Interface (JNI), tuy nhiên cách này mang lại nhiều rủi ro cũng như vấn đề đa nền tảng.
- Ủy thác cho OS (syscall), bằng cách sử dụng Java NIO API để cấp phát bộ nhớ.
- Sử dụng `java.misc.Unsafe` để cấp phát và ghi đọc bộ nhớ của OS.



Hình 2.2 Bộ nhớ Heap và Off-heap trong Java

Trong đó, cách đơn giản nhất là gọi hàm `ByteBuffer.allocateDirect(size)` của lớp `java.nio.ByteBuffer`, hàm này cho phép sử dụng một phân vùng nhớ trên RAM

và trả về một đối tượng `DirectByteBuffer` [7]. Hoặc sử dụng hàm `java.misc.Unsafe.allocate(size)`, hàm này trả về một giá trị số nguyên 64-bit (`long`) biểu diễn địa chỉ của vùng nhớ được cấp phát [8].

Vì Off-Heap là bộ nhớ OS, người dùng phải tự tay huỷ các đối tượng này, `java.misc.Unsafe` cung cấp hàm `freeMemory(address)` cho phép giải phóng vùng nhớ này. Nếu đối sử dụng Java NIO để cấp phát, người dùng có thể lấy được địa chỉ vùng nhớ thông qua đối tượng `DirectByteBuffer`.

### 2.2.2 Java Unsafe

Để xử lý đọc/ghi trên vùng nhớ Off-Heap (native memory) ta cần dùng lớp `sun.misc.Unsafe`, lớp `Unsafe` cung cấp các API để xử lý dữ liệu trực tiếp trên RAM, `Unsafe` cung cấp một vài tính năng như [9] [10]:

- Truy cập trực tiếp vào RAM, CPU, ...
- Tạo đối tượng không cần thông qua constructor, truy cập đối tượng private.
- Tạo ra các đối tượng, lớp ẩn danh mà không cần JVM xác minh.
- Quản lý bộ nhớ Off-Heap thủ công.
- Ngoài dùng `ByteBuffer` để cấp phát bộ nhớ Off-Heap, có thể dùng `Unsafe.allocate()` tuy nhiên hàm này trả về địa chỉ của bộ nhớ không phải một đối tượng `ByteBuffer`.

Từ phiên bản Java 21 trở đi, Java đã cung cấp FFM (Foreign Function and Memory) API, giúp thao tác với native memory.

### 2.2.3 FileChannel và MappedByteBuffer

`FileChannel` là một lớp được cung cấp bởi Java NIO (thư viện của Java), có thể dùng trong việc đọc/ghi dữ liệu của tệp thay cho cách truyền thống sử dụng `FileInputStream/FileOutputStream`, những lợi ích của `FileChannel`:

- Đọc/ghi ở một vị trí bất kỳ trong tệp.

- Tải một phần dữ liệu của tệp trực tiếp vào bộ nhớ, thay vì phải tải toàn bộ như `FileInputStream`.
- Chuyển dữ liệu nhanh hơn.
- Khoá một phân vùng của tệp.

Một lớp con của `ByteBuffer` trong NIO là `MappedByteBuffer`, lớp này cho phép ánh xạ một tệp trên ổ đĩa lên một vùng nhớ trong RAM. Khi đọc/ghi, dữ liệu sẽ đồng thời ở trên RAM và ổ cứng, việc đồng bộ hoá này được xử lý hiệu quả bởi hệ điều hành (`mmap` và `msync`) [11]. So sánh với việc dùng `FileOutputStream/FileInputStream` trong Java, về tổng quan `FileChannel` thực hiện các tác vụ ghi/đọc file hiệu quả hơn [12]:

a. Truy cập dữ liệu

- Với `FileOutputStream/FileInputStream`, việc đọc ghi dữ liệu được thực hiện tuần tự từ lúc đầu tệp đến cuối tệp.
- Trong khi `FileChannel` sử dụng một con trỏ cho phép người dùng đọc/ghi ở một vị trí bất kì.

b. Đa luồng

- `FileOutputStream/FileInputStream` không có xử lý đa luồng, nếu hai luồng (thread) cùng đọc/ghi một tệp, kết quả là không xác định. Cần sử dụng một cấu trúc dữ liệu khác để đồng bộ.
- `FileChannel` cung cấp `FileLock` hỗ trợ khoá một phân vùng của tệp, chặn vùng đó bị truy cập bởi các luồng khác.

### c. Hiệu năng

Sử dụng JMH so sánh hiệu năng, FileChannel sử dụng MappedByteBuffer thao tác trực tiếp trên bộ nhớ và ổ đĩa cho kết quả tốt hơn. Trong khi các tác vụ của FileOutputStream là blocking IO.

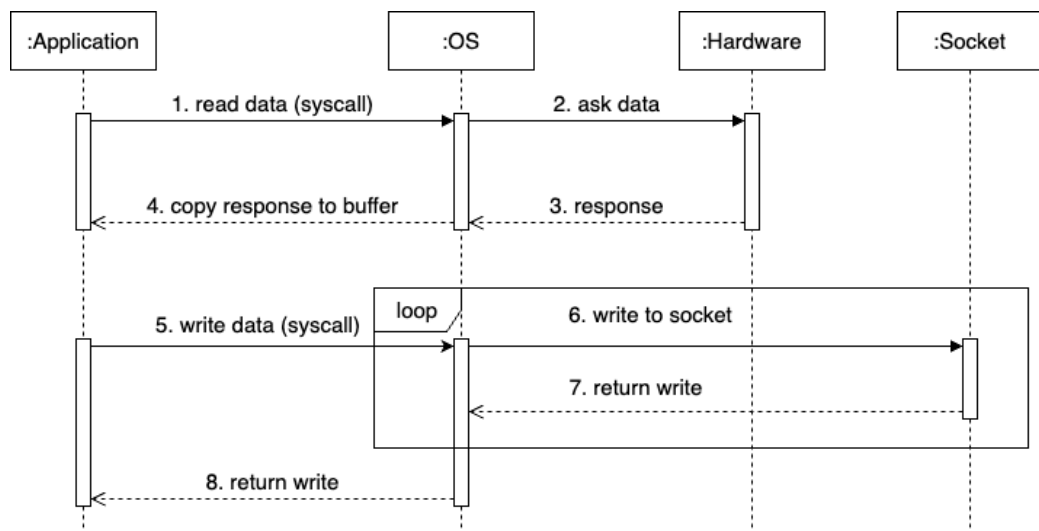
Bảng 2.1 So sánh hiệu năng FileChannel và FileOutputStream bằng JMH

Benchmark	Mode	Cnt	Score	Error	Units
fileChannel	avgt	5	341.414	$\pm 52.229$	ms/op
fileOutputStream	avgt	5	556.102	$\pm 91.512$	m/op

### 2.2.4 Zero copy

Xét tình huống cần đọc và truyền nội dung một tệp thông qua mạng hoặc đến một tiến trình khác, thông thường trong hướng tiếp cận truyền thống sẽ có mã giả như sau:

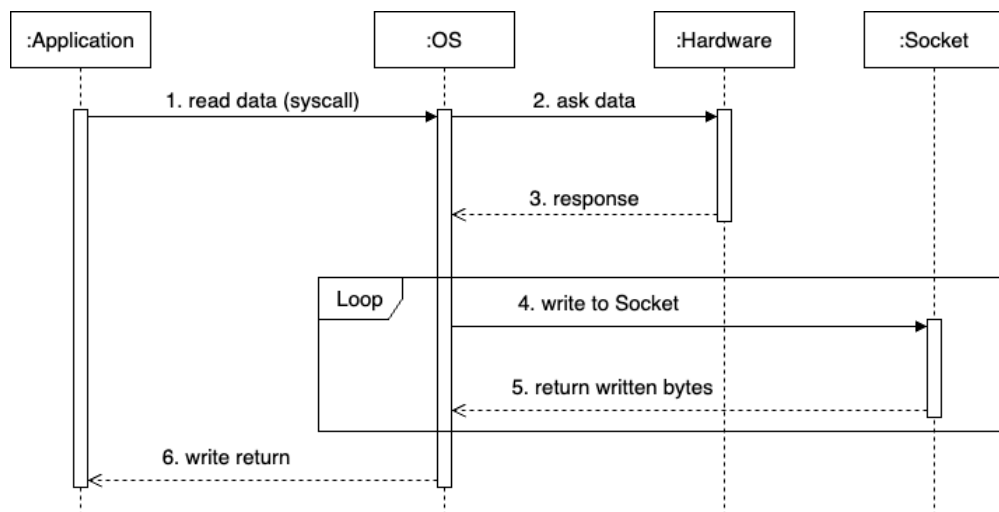
```
File.read(fileDesc, buffer, len);
Socket.send(socket, buffer, len);
```



Hình 2.3 Sơ đồ tuần tự truyền file (không dùng zero-copy)

- Có 4 lần chuyển đổi giữa Application và OS (switching-context): 2 lần gọi syscall từ ứng dụng (Application) và 2 lần trả lại từ OS.
- Trong đó có 2 lần sao chép dữ liệu giữa OS và Application ở bước 4 và 5, dữ liệu ở bước 4 và 5 là hoàn toàn giống nhau, tăng Application chỉ chứa dữ liệu trả về.

Zero-copy là một kỹ thuật giúp giảm số lượt switch-context, nhờ vào FileChannel trong Java NIO, người dùng sẽ uỷ thác thao tác ghi vào Socket cho OS thay vì nhận dữ liệu trả về [13]. Hình 6 minh hoạ kỹ thuật zero-copy.



Hình 2.4 Sơ đồ tuần tự truyền file (dùng zero-copy)

Số lần switch-context giữa OS và Application giảm xuống còn 2, OS chịu trách nhiệm ghi dữ liệu trực tiếp vào socket mà không cần thông qua Application như H.4

Bảng dưới đây so sánh kết quả khi dùng zero-copy và cách truyền thông, để thấy kỹ thuật zero-copy cho kết quả tốt hơn.



Bảng 2.2 So sánh kết quả khi dùng zero-copy

File size	Normal file transfer (ms)	transferTo (ms)
7MB	156	45
21MB	337	128
63MB	843	387
98MB	1320	617
200MB	2124	1150
350MB	3631	1762
700MB	13498	4422
1GB	18399	8537

## 2.3 VLQ (variable-length quantity)

### 2.3.1 VLQ.

Trong máy tính, các số nguyên sẽ được biểu diễn sang dạng nhị phân, đối với Java, kiểu dữ liệu integer (số nguyên) sẽ được biểu diễn bằng 4 octet (32-bits) và kiểu dữ liệu long được biểu diễn bằng 8 octet (64-bits). Với bit đầu tiên (MSB) dùng để đánh dấu là giá trị nguyên âm (MSB = 1) hoặc không âm (MSB = 0) bằng cách biểu diễn phân bù 2 [14].

Ví dụ:

Biểu diễn nhị phân của 10 (32-bit):

$$10 = 00000000\ 00000000\ 00000000\ 00001010$$

Biểu diễn nhị phân của -10 (32-bit):

$$-10 = 11111111\ 11111111\ 11111111\ 11110110$$

Trong ví dụ trên dễ thấy biểu diễn nhị phân của giá trị “10” có 3 octet không mang bất kỳ giá trị nào (đều là 0000 0000), điều này tăng chi phí khi truyền tải qua mạng cũng như không gian lưu trữ ở RAM. Để giải quyết vấn đề này, ta có thể dùng một kỹ thuật có tên là VLQ, VLQ (hay Base128) là một mã phổ quát (universal code) sử dụng một số lượng octet tùy ý để biểu diễn một số nguyên lớn hơn.

### 2.3.2 LEB128

Base128 là một định dạng dùng để nén dữ liệu bằng cách dùng từ 1 - 10 octet để biểu diễn 1 số nguyên không âm 64-bit với việc dùng MSB của mỗi octet để đánh dấu octet theo sau, 7 bit thấp hơn thể hiện giá trị của octet đó. LEB128 (Little-Endian Base128) giống Base128 nhưng thứ tự của octet là Little-Endian [15].

Cấu trúc của 1 octet trong Base128 (Bảng 3):

- Nếu A (MSB) là 0, thì đây là octet cuối cùng của 1 số nguyên, nếu A là 1, thì vẫn còn 1 octet nữa theo sau.
- B là một số 7-bit có giá trị nằm trong khoảng [0x00, 0x7f].

Bảng 2.3 Cấu trúc 1 octet trong Base128

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$
A	B						

Ở đề tài này, nhóm sẽ tập trung vào giải thuật LEB128, LEB128 được sử dụng trong định dạng DWARF debug file [16], cũng như được sử dụng rộng rãi trong LLVM’s Coverage Mapping Format [17], Google Protocol Buffer (Protobuf) [18], WebAssembly [19], v.v.. Mã giả của LEB128 như sau:

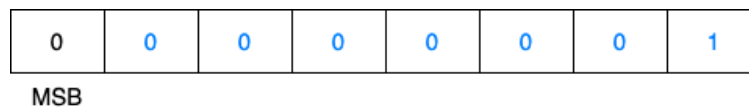
Mã hoá số nguyên không âm:

```
do {
    byte = low-order 7 bits of value
    value = value >> 7
    if value != 0
        set MSB of byte to 1
    emit byte to result
} while value != 0
```

Giải mã số nguyên không âm:

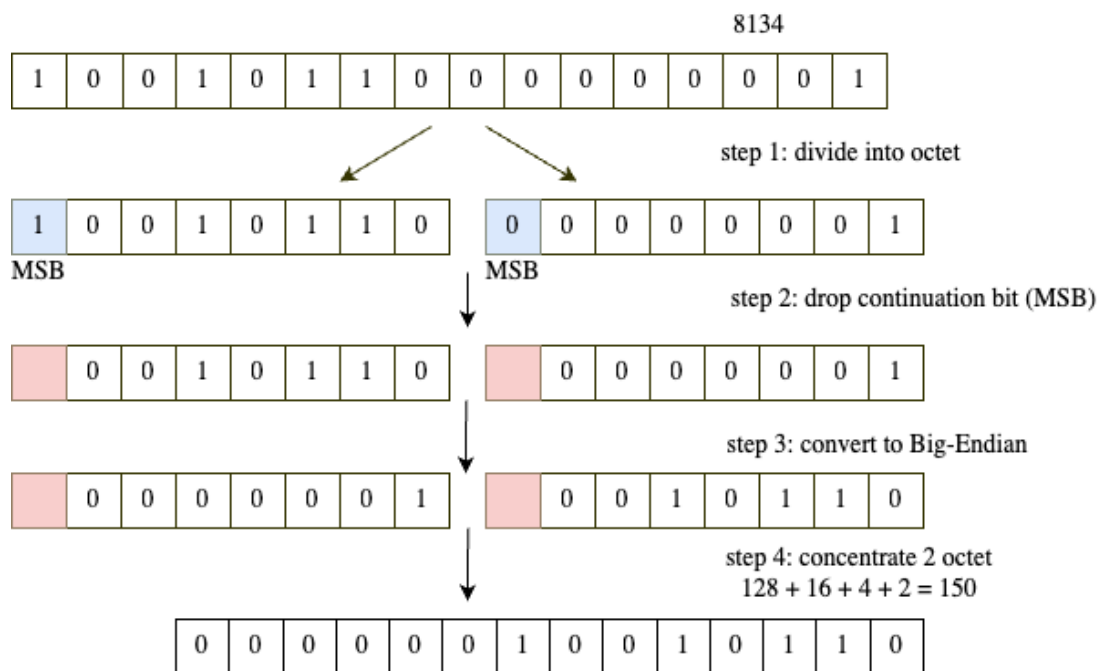
```
result = 0, shift = 0
while (true) {
    byte = low-order 7 bits of next byte
    result = result OR (byte << shift)
    if MSB of byte == 0
        break
    shift += 7
}
return result
```

Lấy ví dụ giá trị “1” có biểu diễn nhị phân tương ứng là “1”, chỉ cần dùng 1 octet, nên giá trị MSB = 0.



Hình 2.5 Mô tả LEB128 biểu diễn số “1”

Lấy ví dụ khác là giá trị “150”, trong LEB128 sẽ là “8134”, cụ thể như sau:



Hình 2.6 Mô tả cách LEB128 biểu diễn số “150”

Ở ví dụ trên, bằng cách dùng LEB128 thay vì dùng 4 octet cho số “150” như thông thường giờ chỉ còn 2 octet, giúp giảm 1 nửa không gian lưu trữ. Một số ưu điểm của LEB128:

- Tính tinh gọn: nhỏ gọn chính là một trong những điểm quan trọng của LEB128, vì nó sử dụng ít octet hơn để mã hoá một số nguyên dẫn đến kích thước của tệp nhỏ hơn, hữu dụng trong các yêu cầu tối ưu không gian lưu trữ
- Tính linh hoạt: số lượng octet dựa trên độ lớn của số nguyên, ta có thể dùng để biểu diễn số từ 8-bit tới 64-bit.

Bảng dưới đây cho thấy tính hiệu quả của VLQ trong việc biểu diễn số nguyên không âm, VLQ hoạt động hiệu quả ở các số nguyên dương nhỏ khi dùng ít hơn 4 octet, đối với giá trị lớn hơn “268435455” sẽ phải dùng 5 octet.

Bảng 2.4 So sánh biểu diễn thông thường và VLQ số nguyên không âm 32-bit

Integer (decimal)	Integer (binary)	VLQ (binary)	Integer (hexadecimal)	VLQ (hexadecimal)
0	00000000 00000000 00000000 00000000	00000000	00000000	00
127	00000000 00000000 00000000 01111111	01111111	0000007F	7F
128	00000000 00000000 00000000 10000000	10000001 00000000	00000080	81 00
8192	00000000 00000000 00100000 00000000	11000000 00000000	00002000	C0 00
16383	00000000 00000000 00111111 11111111	11111111 01111111	00003FFF	FF 7F
16384	00000000 00000000 01000000 00000000	10000001 10000000 00000000	00004000	81 80 00

2097151	00000000 00011111 11111111 11111111	11111111 11111111 01111111	001FFFFFF	FF FF 7F
2097152	00000000 00100000 00000000 00000000	10000001 10000000 10000000 00000000	00200000	81 80 80 00
134217728	00001000 00000000 00000000 00000000	11000000 10000000 10000000 00000000	08000000	C0 80 80 00
268435455	00001111 11111111 11111111 11111111	11111111 11111111 11111111 01111111	0FFFFFFF	FF FF FF 7F

### 2.3.3 Xử lý số nguyên âm

VLQ chỉ có thể áp dụng với với số nguyên không âm, việc dùng MSB để đánh dấu kết thúc hoặc tiếp tục đã làm mất khả năng biểu diễn số nguyên âm của octet đầu tiên (biểu diễn nhị phân thông thường), nhóm cần sử dụng kỹ thuật khác để xử lý số nguyên âm [20]:

- Mã hoá Zigzag: ý tưởng của thuật toán này sẽ là với mỗi số k:

$$f(x) = \begin{cases} 2x, & x \geq 0 \\ 2x + 1, & x < 0 \end{cases}$$

- Encode Zigzag:

$$V = (x \ll 1) \text{ XOR } (x \gg 32)$$

- Decode zigzag:

$$V = (x \gg 1) \text{ XOR } -(x \& 1)$$

Bảng 2.5 Giá trị mã hoá ZigZag đối số nguyên 32-bit

Signed Original	Encoded
0	0
-1	1

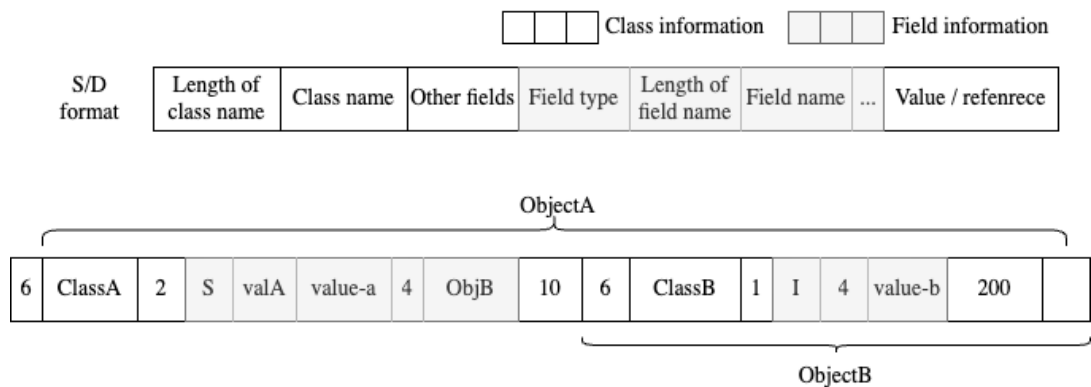
1	2
-2	3
...	...
0x7fffffff	0xffffffff
-0x80000000	0xffffffff

Như vậy ta có thể dùng mã hoá Zigzag trước khi encode/decode bằng LEB128 để giải quyết vấn đề số nguyên âm. Ngoài ra, còn cách khác như sử dụng phần bù 2, v.v..

## 2.4 Thư viện Kryo

Object Serialization là quá trình chuyển đổi các đối tượng trong Java sang các byte dữ liệu (byte stream), và ngược lại Deserialization chuyển đổi các byte dữ liệu sang đối tượng. Serialization/Deserialization (S/D) là nền tảng của các hoạt động phân tích dữ liệu, quản lý I/O trong hệ thống phân tán, và cấu trúc của gói tin trong các thủ tục gọi từ xa [21].

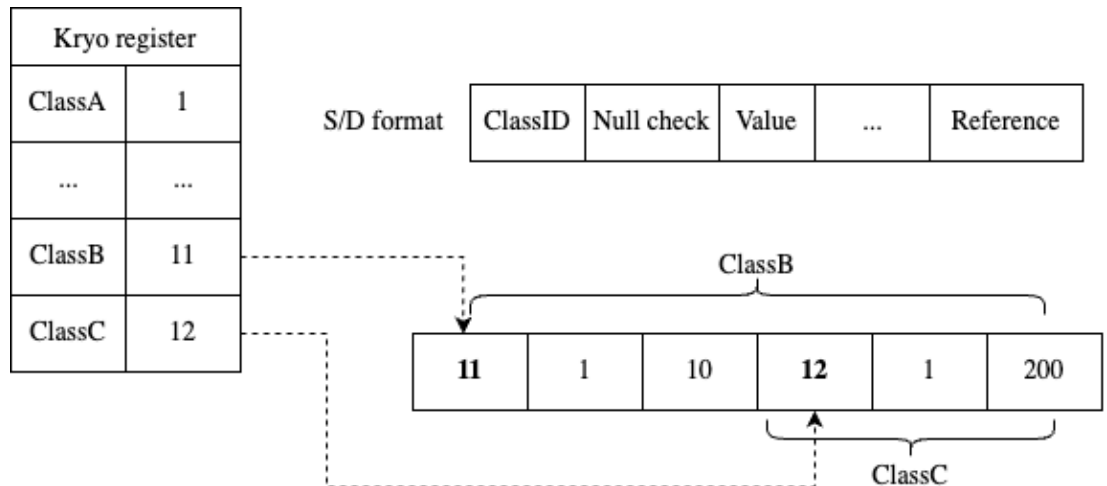
Java cung cấp sẵn một cơ chế Serialization để chuyển Object sang byte chứa các thông tin các trường của Class và các đối tượng tham chiếu.



Hình 2.7 Mô tả cấu trúc của Java S/D

Java Kryo là một trong những thư viện phổ biến nhất hỗ trợ S/D trong Java [8], Kryo được thiết kế để tối ưu hoá tốc độ và kích thước khi tuần tự hoá dữ liệu, rất phù hợp cho các ứng dụng có yêu cầu lưu trữ cao và xử lý dữ liệu nhanh. Bằng cách sử dụng các ID được đăng ký cho từng lớp, Kryo giảm chi phí cho việc lưu tên trường, metadata như trong Java S/D.





Hình 2.8 Mô tả cấu trúc Kryo S/D

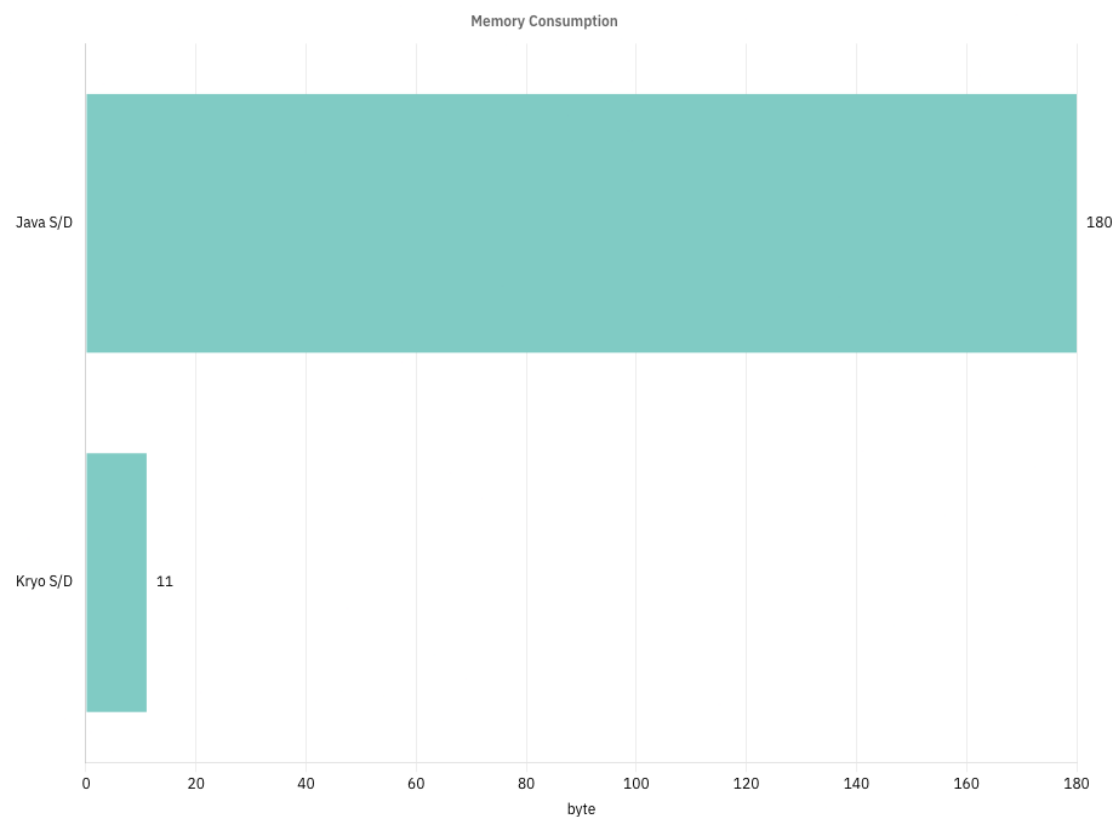
Các ưu điểm khi dùng Kryo:

- Hiệu suất cao: Kryo nhanh hơn so với cơ chế serializable mặc định của Java.
- Không gian lưu trữ: ít hơn so với mặc định của Java.
- Loại bỏ thông tin trùng lặp: Kryo sẽ phát hiện và xử lý các đối tượng trùng lặp bằng đồ thị, dữ liệu trùng sẽ được serialize bằng tham chiếu thay vì sao chép.
- Chỉ serialize các dữ liệu cần thiết (non-transient), các trường được đánh dấu transient sẽ bị bỏ qua.
- Áp dụng các kỹ thuật mã hoá VLQ kèm với zig-zag (mục 2.3.3) coding cho kiểu số nguyên.
- Nhờ những ưu điểm trên Kryo được sử dụng rộng rãi trong các phần mềm xử lý dữ liệu như Spark [22], Flink [23], v.v..

Tuy nhiên khi dùng Kryo cũng có 1 vài nhược điểm như:

- Phụ thuộc vào lớp được đăng ký, Kryo cần được đăng ký tên lớp để tối ưu việc nén, giải nén (vấn đề này sẽ được đề cập ở mục 3.3.2).
- Không an toàn khi trong môi trường đa luồng (thread), cần dùng kỹ thuật tạo pool để khắc phục vấn đề.

So sánh kết quả, nhóm em nhận thấy việc dùng Kryo S/D cho kết quả vượt trội với Java S/D, đặc biệt là đối với 1 ứng dụng cần lưu trữ nhiều và truyền tải dữ liệu qua mạng.



Hình 2.9 So sánh không gian lưu trữ Java và Kryo S/D

## 2.5 Thư viện Netty

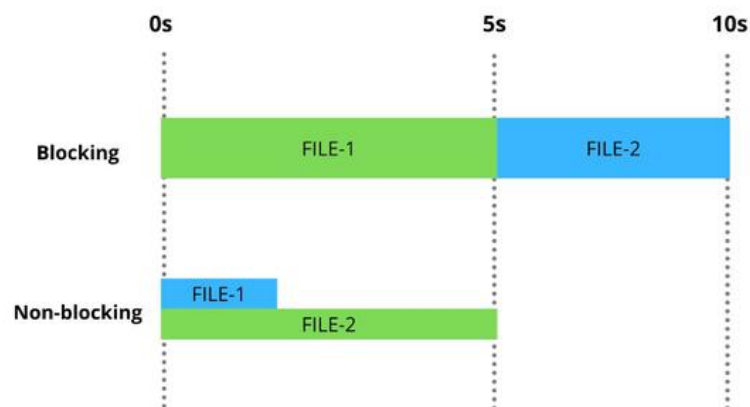
Một điểm quan trọng khác của Java NIO nhóm có thể tận dụng được là Non-Blocking IO [9]. Với Blocking IO, các tác vụ được thực hiện tuần tự, các tác vụ cần

nhiều thời gian xử lý I/O sẽ cản các tác vụ phía sau đó được thực thi, trong mô hình này mỗi luồng (thread) chỉ có 1 một tiến trình (process) được thực thi.



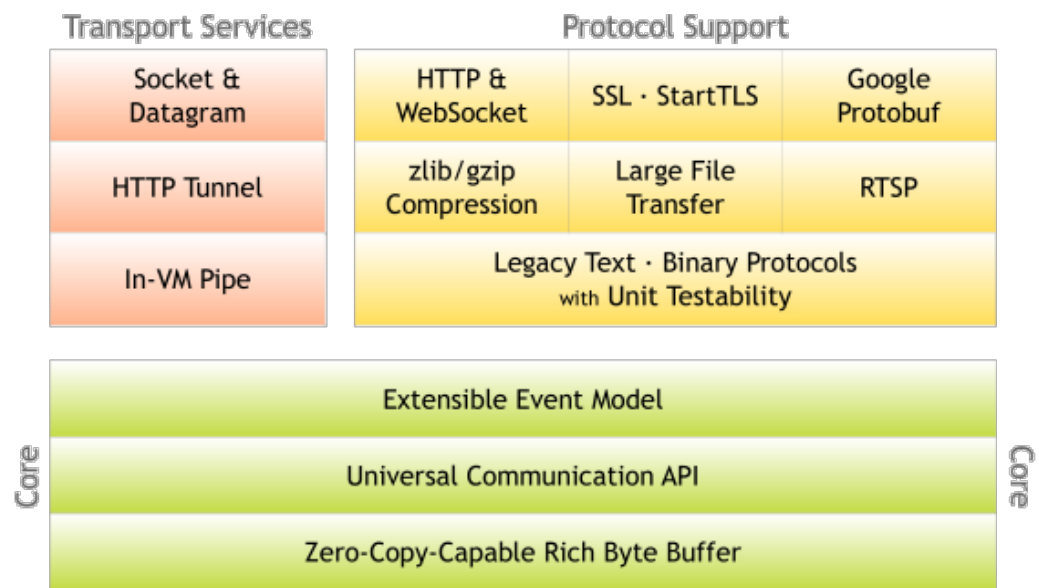
Hình 2.10 Minh hoạ Blocking I/O

Ngược lại, với Non-Blocking I/O các tác vụ không nhất thiết phải thực hiện tuần tự và đồng thời, về mặt hàm lý các tác vụ phía sau không phụ thuộc vào kết quả tác vụ phía trước thì nó có thể hoàn thành trước. Thậm chí nhiều tiến trình khác nhau có thể được xử lý trên 1 luồng [14]. Non-Blocking I/O nhờ vào khả năng xử lý nhiều tiến trình trên cùng 1 thread, cực kì phù hợp với các máy chủ phải phục vụ nhiều người dùng và thông lượng cao.



Hình 2.11 Minh hoạ so sánh Blocking và Non-Blocking I/O

Netty là một thư viện của Java hỗ trợ việc lập trình Non-Blocking IO [24], kiến trúc của Netty:



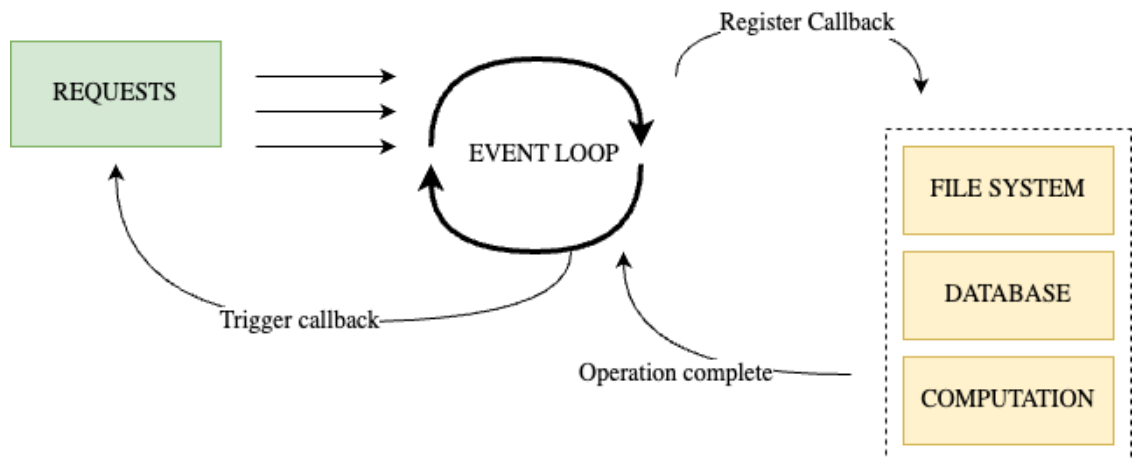
Ảnh 1 Kiến trúc của Netty

Netty được xây dựng dựa trên khung của tầng vận chuyển dữ liệu (Transport service) và các giao thức Internet (Protocol).

- Hỗ trợ Socket (TCP) & Datagram (UDP).
- Http Tunnel: một kỹ thuật đóng gói dữ liệu của TCP/IP trong gói tin HTTP.
- In-VM pipe: kênh giao tiếp của JVM.
- Hỗ trợ hầu hết các giao thức hiện nay (HTTP, SSL, TLS, v.v.)
- Hỗ trợ zero-copy tệp tin.

Thành phần quan trọng nhất của Non-Blocking I/O là event-loop, tương tự như trong V8-Engine của Chrome, event-loop tại mỗi thời điểm sẽ lấy ra một số tác vụ nhất định. Netty cung cấp sẵn cho 2 triển khai event-loop (epoll và kqueue [25]).

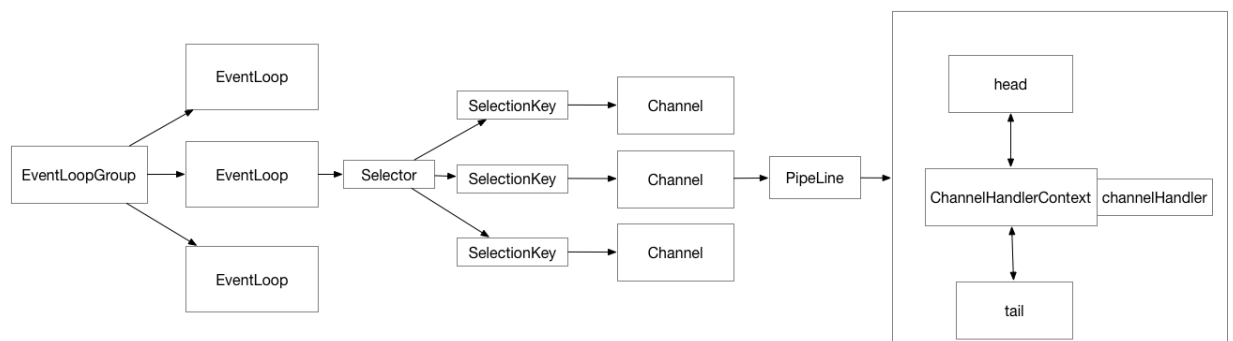
- Trong Linux, NioEventLoop sẽ tương thích với EpollEventLoop (sử dụng thuật toán epoll)
- Trong MacOS/BSD, NioEventLoop sẽ tương thích với KqueueEventLoop (sử dụng thuật toán kqueue)



Hình 2.12 Minh hoạ Event-Loop trong Netty

Event-loop sẽ được quản lý bởi EventLoopGroup, EventLoopGroup sẽ được đính vào một kênh (Channel) thông qua Selector [26] (H. 14).

- Kênh có thể là SocketChannel / ServerSocketChannel.
- Khi hoạt động EventLoopGroup sẽ liên tục lấy ra các Channel sẵn sàng cho việc đọc/ghi.

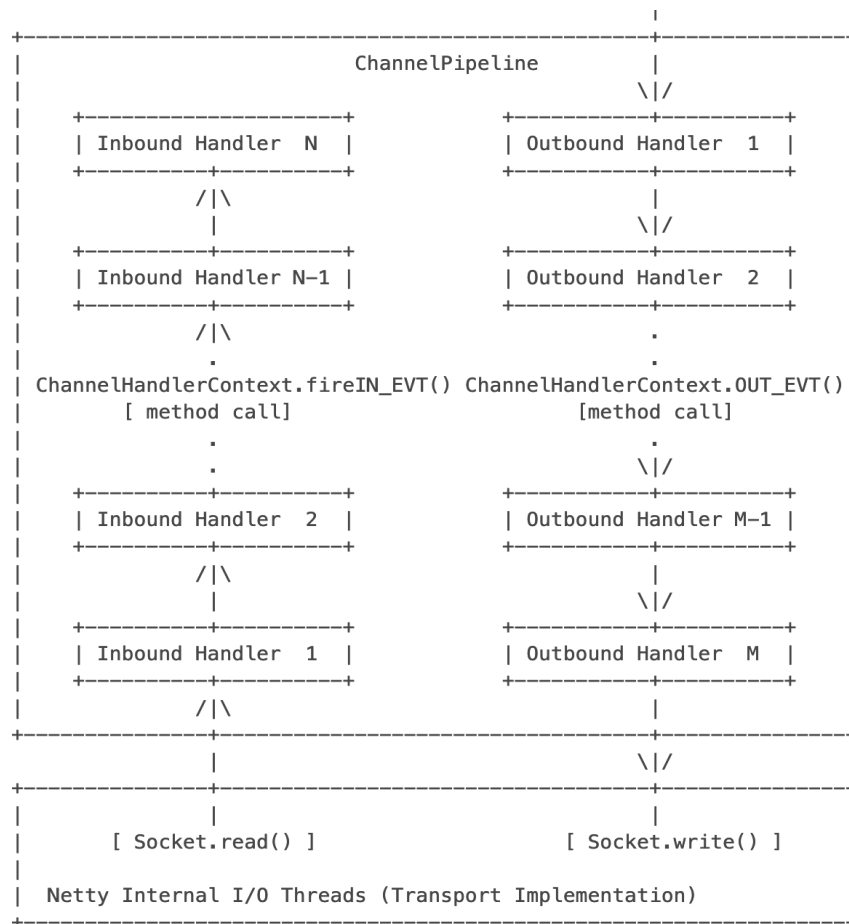


Hình 2.13 Netty event-loop, Selector và Channel

Về phần xử lý nghiệp vụ, Netty cung cấp 1 tính năng gọi là Pipeline, trong Pipeline sẽ có các triển khai của Handler do người dùng tự định nghĩa, và sẽ được gọi khi được đọc/ghi dữ liệu. Có 2 loại Handler là InboundHandler (khi đọc dữ liệu) và OutboundHandler (khi ghi dữ liệu).

- Các Handler sẽ được thực hiện lần lượt theo thứ tự được thêm vào Pipeline

- Một Handler có thể vừa là InboundHandler và là OutboundHandler, gọi là ChannelDuplexHandler.



Hình 2.14 Netty pipeline và handler

## 2.6 Hash table

Với một khoá vào, 1 hàm băm  $f(x)$  (hash function) sẽ chuyển khoá đó thành một số có giá trị nguyên dương nhỏ hơn kích thước bảng, và đưa vào mảng tại vị trí đó, bảng băm cung cấp một vài phương thức cơ bản như thêm, tìm kiếm, xoá [27].

### 2.6.1 Hàm băm

Hàm băm  $f(x)$  thực hiện tính toán một khoá bất kỳ sang một số nguyên không âm (32 hoặc 64-bit). Trong bảng băm, thường sẽ chọn các hàm băm non-cryptographic (NCHFs) do tốc độ tính toán nhanh cũng như không yêu cầu bảo mật. Một số NCHFs phổ biến như CRC32C, MD5, SHA1, Murmur3, Lookup3, SpookyHash, FNV, v.v..

Trong bảng băm, hai khoá khác nhau thông qua hàm  $f(x)$  có thể có cùng giá trị, đây chính là va chạm trong bảng băm, một số cách xử lý va chạm sẽ đề cập ở 2.6.2 và 2.6.3.

Các đặc tính quan trọng để lựa chọn một NCHF bao gồm:

- Tỷ lệ phân phối (percentage distribution): thể hiện sự phân bố của không gian bộ nhớ.
- Tỷ lệ va chạm (collision rate): thể hiện sự trùng lặp giá trị khi băm một khoá.
- Hiệu năng (performance): thời gian để tính toán 1 khoá.

Murmur3 và Lookup3 là 2 trong những NCHFs được khuyến dùng vì thể hiện sự vượt trội trong cả 3 tiêu chí [28], Lookup3 có thời gian tính toán nhanh hơn trong khi Murmur3 có tỷ lệ phân phối tốt hơn.

- Mã giả Murmur3:

```

c1 = 0xcc9e2d51, c2 = 0x1b873593
r1 = 15, r2 = 13
m = 5
n = 0xe6546b64
hash = seed
for each fourByteChunk of key do
    k = fourByteChunk
    k = k * c1
    k = k ROL r1
    k = k * c2

    hash = hash XOR k
    hash = hash ROL r2
    hash = (hash * m) + n

with any remainingBytesInKey do
    remainingBytes = remainingBytes * c1
    remainingBytes = remainingBytes ROL r1
    remainingBytes = remainingBytes * c2
    hash = hash XOR remainingBytes
hash = hash XOR key_len

hash = hash XOR (hash >> 16)
hash = hash * 0x85ebca6b
hash = hash XOR (hash >> 13)
hash = hash * 0xc2b2ae35
hash = hash XOR (hash >> 16)

return hash

```

- Mã giả Lookup3:

```

i = 0, hash = 0
while i != key_len do
    hash = hash + key[i++]
    hash = hash << 10
    hash = hash ^ (hash >> 6)

hash = hash << 3
hash = hash ^ (hash >> 11)
hash = hash << 15
return hash

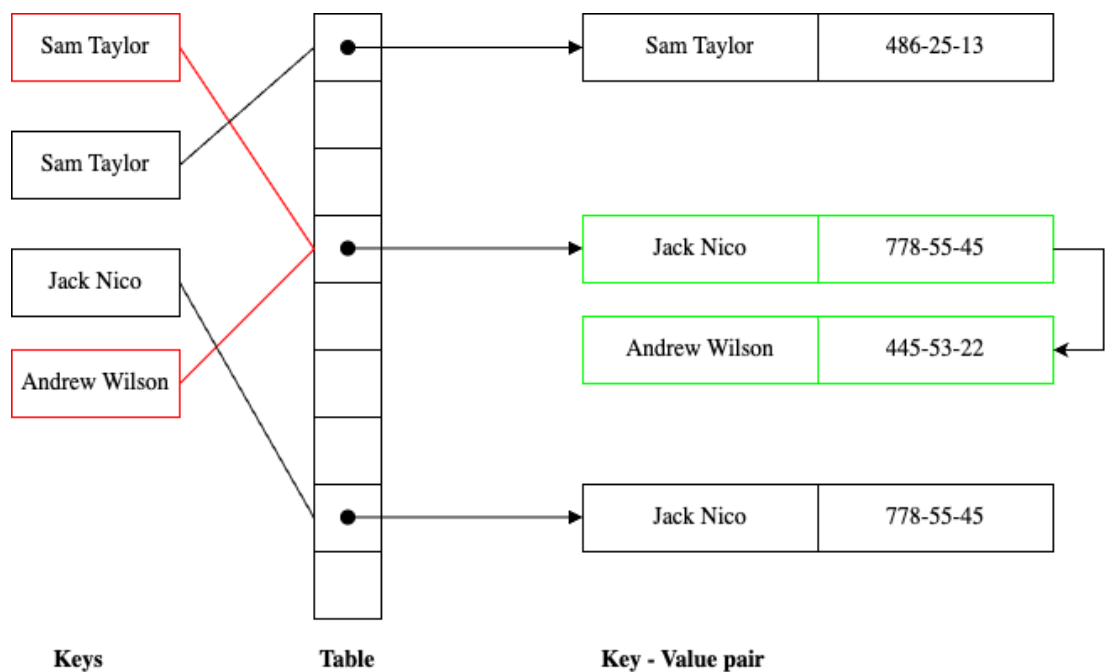
```



### 2.6.2 Chaining

Đây là cách phổ thông và dễ triển khai nhất để giải quyết vấn đề va chạm bằng cách sử dụng thêm một cấu trúc dữ liệu bên ngoài bảng. Mỗi ô dữ liệu của bảng sẽ là một danh sách liên kết đơn (single-linked list) chứa cặp khoá – giá trị.

- Việc tìm kiếm một khoá sẽ bắt đầu với việc dùng hàm băm  $f(x)$  để xác định vị trí của bảng.
- Duyệt qua danh sách liên kết của ô đó đến khi tìm node có khoá giống khoá cần tìm.
- Khi thêm mới một bản ghi, thêm vào cuối danh sách liên kết của mỗi vị trí trong bảng băm.



Hình 2.15 Mô tả Chaining Hash-Table

Một vấn đề của giải pháp Chaining là sẽ phụ thuộc vào con trỏ, sẽ phải tốn thêm bộ nhớ để duy trì danh sách liên kết cho từng ô nhớ của bảng, đặc biệt khi lưu trữ nhiều khoá [29].

### 2.6.3 Open-addressing (OA)

Open-Addressing Hash-Table (OA Hash-Table) là một loại bảng băm sử dụng kỹ thuật xử lý va chạm bằng cách tìm một vị trí khác chưa được sử dụng mà không cần dùng thêm một cấu trúc dữ liệu khác, kích thước của bảng luôn phải lớn hơn hoặc bằng tổng số lượng bản ghi.

Các cách khác nhau để xác định 1 vị trí trong bảng OA [27]:

- Thăm dò tuyến tính (Linear probing): Bảng băm bắt đầu tìm kiếm tuần tự từ vị trí ban đầu sau khi hàm  $f(x)$  xác định kết quả, nếu vị trí nhận được bị chiếm hoặc không phải của khoá, ta sẽ tìm vị trí kế tiếp.

```
h = hash(key)
probe = 1
x = h % table_size
while slot of x is used
    x = (x + probe) % table_size
    probe++
Return x
```

- Thăm dò bậc hai (Quadratic probing): Đối với Linear probing có khả năng xuất hiện hiện tượng phân cụm (clustering) rất cao, khi 1 vùng có quá nhiều ô trống.

```
h = hash(key)
probe = 0
x = h % table_size
while slot of x is used
    x = (h + probe*probe) % table_size
    probe++
return x
```

- Hàm băm đôi (Double hashing): Khoảng cách giữa các lần tìm kiếm ô nhớ được xác định bằng một hàm băm khác. Double hashing giảm phân cụm theo cách tối ưu nhất.

```

i = 1
h = hash(key)
probe = hash_2(key)
x = h % table_size
while slot of x is used
    x = (h + i*probe) % table_size
    i++
return x

```

Đối với OA Hash-Table, tỉ lệ tải (load factor) là một yếu tố quan trọng, một khi số lượng bản ghi đạt tỉ lệ tải, cần phải tăng kích thước bảng băm lên và tiến hành phân bổ lại toàn bộ bản ghi trong bảng băm.

Công thức tính load factor:

$$F = \frac{S}{C}$$

Trong đó:

- F: load factor của bảng.
- S: số lượng bản ghi hiện có.
- C: kích thước của bảng băm.

#### 2.6.4 So sánh Chaining và Open-addressing

So sánh giữa Chaining và OA Hash-Table [30]:

Bảng 2.6 So sánh Chaining và OA Hash-Table

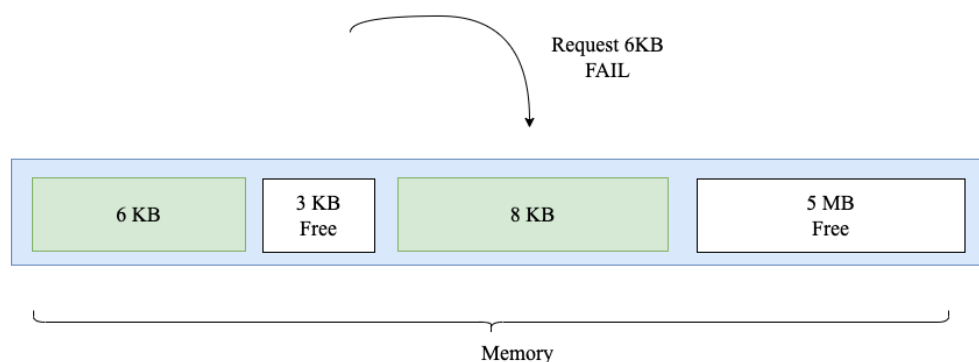
	Chaining	Open-addressing
Xử lý đụng độ	Sử dụng một cấu trúc khác như danh sách liên kết	Sử dụng chính bảng băm
Hiệu năng dựa trên hệ số tải	Tỉ lệ thuận với tải	Tỉ lệ theo $(load\_factor) / (1 - load\_factor)$
Cho phép nhiều bản ghi hơn kích thước bảng	Có	Không, load_factor nên dưới 0.75
Hàm băm	Phân phối đồng nhất	Phân phối đồng nhất, hạn chế phân cụm

Triển khai	Đơn giản	Khó hơn
------------	----------	---------

## 2.7 SLAB Allocation

Phân mảnh bộ nhớ (memory fragmentation) là hiện tượng xảy ra khi bộ nhớ bị chia nhỏ thành các vùng không liên tiếp, dẫn đến ảnh hưởng về hiệu năng, lỗi trang, và không đủ bộ nhớ. Việc cấp phát bộ nhớ là ngẫu nhiên, khi thêm/xoá đối tượng khỏi bộ nhớ tạo ra các vùng trống, dẫn đến phân mảnh bộ nhớ [31].

Phân mảnh bộ nhớ là vấn đề thường gặp đối với các ứng dụng chạy lâu cần lưu trữ nhiều dữ liệu cũng như cập nhật thêm/xoá thường xuyên.

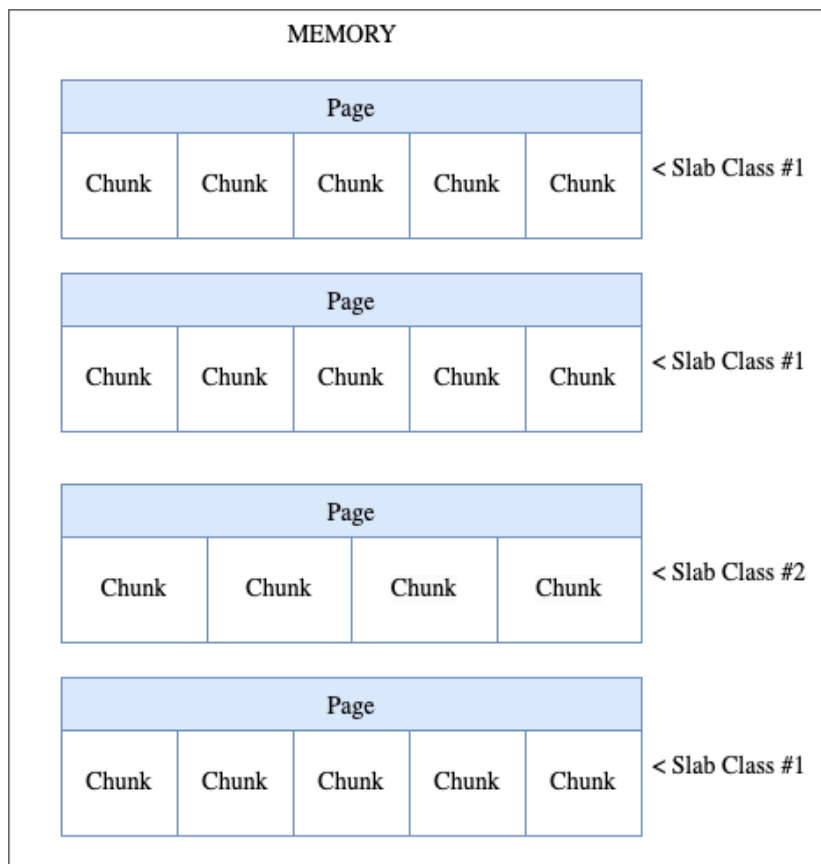


Hình 2.16 Minh hoạ phân mảnh bộ nhớ

Nhóm đã chọn sử dụng Slab Allocation (SLAB) làm cơ chế giúp hỗ trợ quản lý bộ nhớ, cấp phát động, và chống phân mảnh bộ nhớ. Slab Allocation thường được sử dụng trong các phần mềm sử dụng nhiều bộ nhớ (như Memcached [32]) giúp giảm đáng kể chi phí, tuần suất khởi tạo và huỷ các đối tượng là vùng nhớ trong hệ thống, các thành phần chính trong SLAB [33].

- **Chunk:** Chunk là đơn vị nhỏ nhất và chính là vùng nhớ SLAB cấp phát cho người dùng khi muốn ghi/đọc một đối tượng. Kích thước của dữ liệu ghi vào phải nhỏ hơn kích thước chunk. Các chunk này liên tiếp và tập hợp lại với nhau tạo thành 1 slab (page).

- Slab (Slab Page): 1 slab là một tập hợp của một số chunk có kích thước giống nhau (thường là bội của 2). Khi SLAB cần nhiều bộ nhớ hơn, 1 SlabPage mới sẽ được phân bổ trên RAM.
- SlabClass: là đơn vị lớn nhất trong SLAB Allocation, đây là nơi khởi tạo và quản lý các page, khi có một yêu cầu ghi, SLAB sẽ tìm kiếm SlabClass phù hợp (theo cơ chế first-fit) và trả về một chunk.



Hình 2.17 Mô tả cấu trúc SLAB

- Freelist: là một cấu trúc dữ liệu, thường là một danh sách liên kết, dùng để thêm/lấy những chunk được đánh dấu chưa dùng (free) trong 1 slab-class.
- Mã giả cho cấu trúc dữ liệu Freelist:

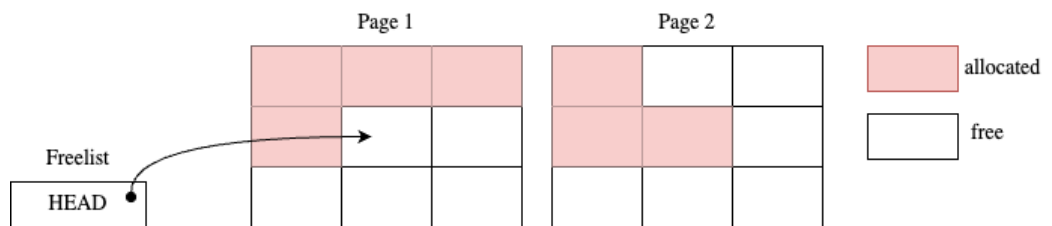
```

node_t {
    chunk_t next;
    chunk_t current;
}

freelist_t {
    node_t head;
    poll: {
        If head is null then emit null
        tmp = head->current
        head = head->next
        emit tmp
    }

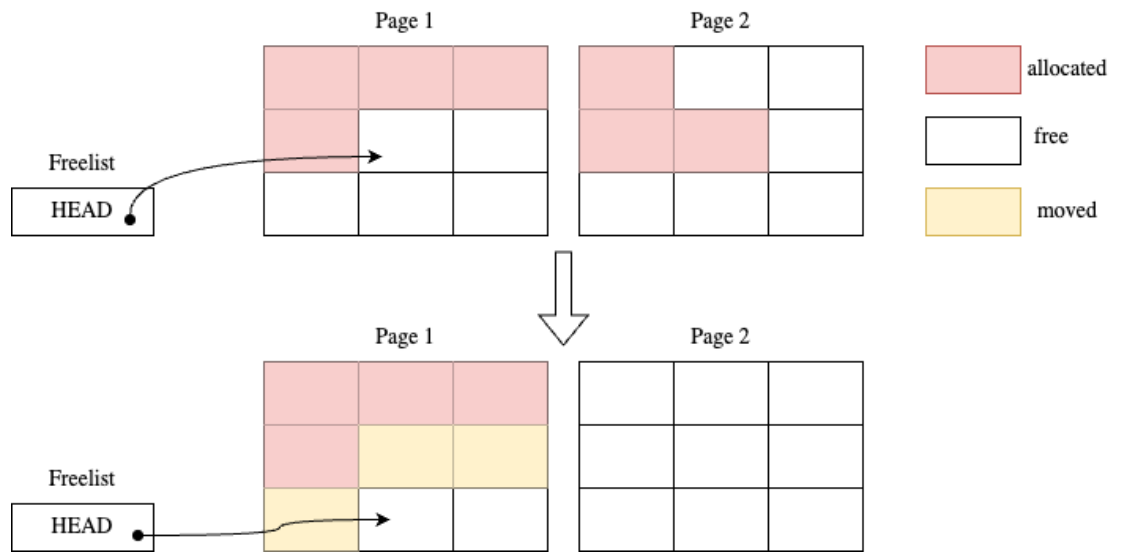
    offer(input): {
        newNode = new node
        newNode->next = head
        head = newNode
    }
}

```



Hình 2.18 Mô tả SLAB freelist

Với việc luôn cấp phát một kích thước cố định (fixed-size) của 1 chunk, ta chấp nhận việc phân mảnh nhỏ bên trong (internal fragmentation) [31, 30]. Để giảm vấn đề phân mảnh trong, SLAB sử dụng một cơ chế gọi là cân bằng (re-balancing) [33] khi một Page có nhiều chunk trống, lúc này SLAB sẽ đánh giá xem liệu có nên giải phóng toàn bộ chunk trong 1 page và chuyển vào các page khác.

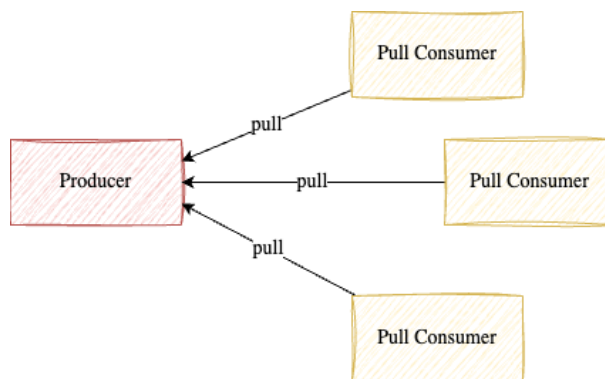


Hình 2.19 Minh hoạ SLAB re-balancing

## 2.8 Pull-based và push-based

Trong các hệ thống phân tán, có hai cơ chế chính để 2 thành phần là Producer và Consumer giao tiếp với nhau là pull-based và push-based [34].

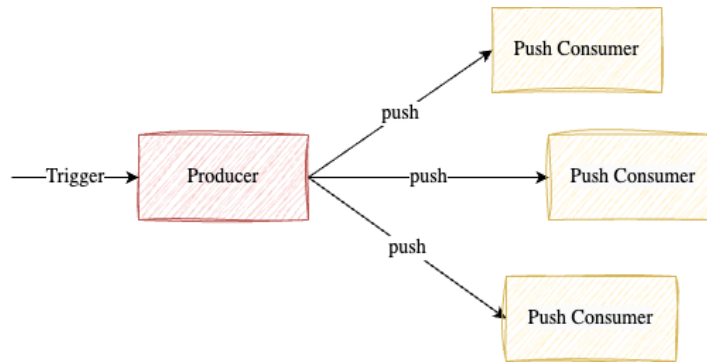
Đối với mô hình pull-based Consumer là đối tượng bắt đầu (actor), Consumer sẽ chủ động yêu cầu tài nguyên từ Producer sau một khoảng thời gian nhất định. Việc lựa chọn khoảng thời gian cho mỗi lần “pull” sẽ ảnh hưởng lên băng thông, cũng như khả năng làm mới dữ liệu. Mức thời gian “pull” thấp sẽ tiêu thụ ít băng thông hơn, tuy nhiên sẽ không thể bám sát thời gian thực với Producer, và ngược lại. Một số ứng dụng sử dụng pull-based: Kafka, Prometheus, v.v..



Hình 2.20 Minh hoạ mô hình pull-based

Đối với mô hình push-based Producer là đối tượng bắt đầu, Producer sẽ chủ động thông báo cho tất cả các Consumer về sự thay đổi và Consumer sẽ tải tài nguyên về, mô hình yêu cầu Producer và Consumer sẽ phải nhận biết và phát hiện ra nhau liên tục. Lợi điểm của mô hình push-based là thông tin gần như ngay lập tức (real-time), tuy nhiên sẽ gia tăng phí tổn của Producer khi phải duy trì quản lý tất cả Consumer. Một số ứng dụng sử dụng push-based: RabbitMQ, ActiveMQ, v.v..





Hình 2.21 Minh hoạ mô hình push-based

Theo các thiết kế truyền thống, các hệ thống trao đổi tin nhắn thường tuân theo mô hình pull-based (ngoại trừ các hệ thống thời gian thực), nhược điểm là phụ thuộc vào mức thời gian “pull”. Mục tiêu chung là cho phép Consumer tiêu thụ ở mức nhiều nhất có thể, đối với hệ thống push-based, Consumer có thể bị quá tải khi khả năng xử lý ở dưới mức hoạt động của Producer (ví dụ bị tấn công từ chối dịch vụ). Mặc dù phụ thuộc vào mức thời gian làm mới, tuy nhiên với một tỉ lệ phù hợp Consumer hoàn toàn có thể bắt kịp với trạng thái Producer, trong Kafka mức thời gian này là 500ms và gần như bám sát thời gian thực [35].

Bên cạnh đó, khả năng mở rộng của mô hình pull-based là tốt và đơn giản hơn khi Producer không cần phải quan tâm về Consumer như mô hình push-based, việc thêm một vài hay nhiều Consumer không gây bất cứ ảnh hưởng nào đáng kể lên Producer.

## Chương 3. PHÂN TÍCH YÊU CẦU

Dựa vào các đặc điểm cũng như nhu cầu được nêu ở Chương 1, phần này sẽ trình bày về các yêu cầu nghiệp vụ, use-case của phần mềm.

### 3.1 Tính năng

- Delta-load: Đối với mỗi lần thay đổi về dữ liệu, Producer sẽ phát hành một phiên bản bao gồm các thông tin và dữ liệu liên quan.
- Rollback: khi gặp các vấn đề về sự cố dữ liệu, có khả năng khôi phục lại trạng thái trước đó.
- API truy vấn: các máy khách (Consumer) phải có khả năng truy xuất dữ liệu từ các khoá cho trước.

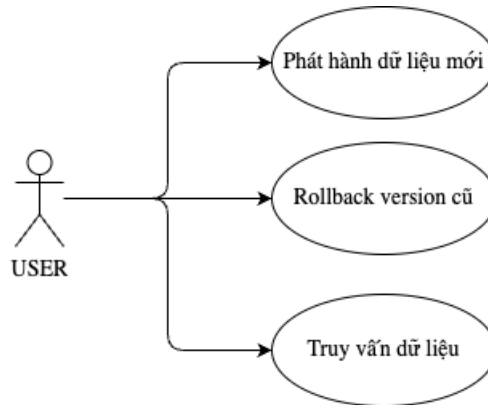
### 3.2 Danh sách các Actor

STT	Tên	Vai trò
1	Người dùng	Sử dụng các API của phần mềm
2	Consumer	Giao tiếp, cập nhật với Producer

### 3.3 Sơ đồ Use-case

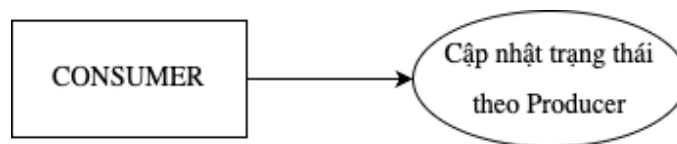
Có 2 actor chính trong kiến trúc của phần mềm: người dùng và Consumer.

Use-case của người dùng:



Hình 3.1 Sơ đồ Use-case của người dùng

Use case của Consumer:



Hình 3.2 Sơ đồ Use-case của Consumer

### 3.4 Danh sách các Use-case

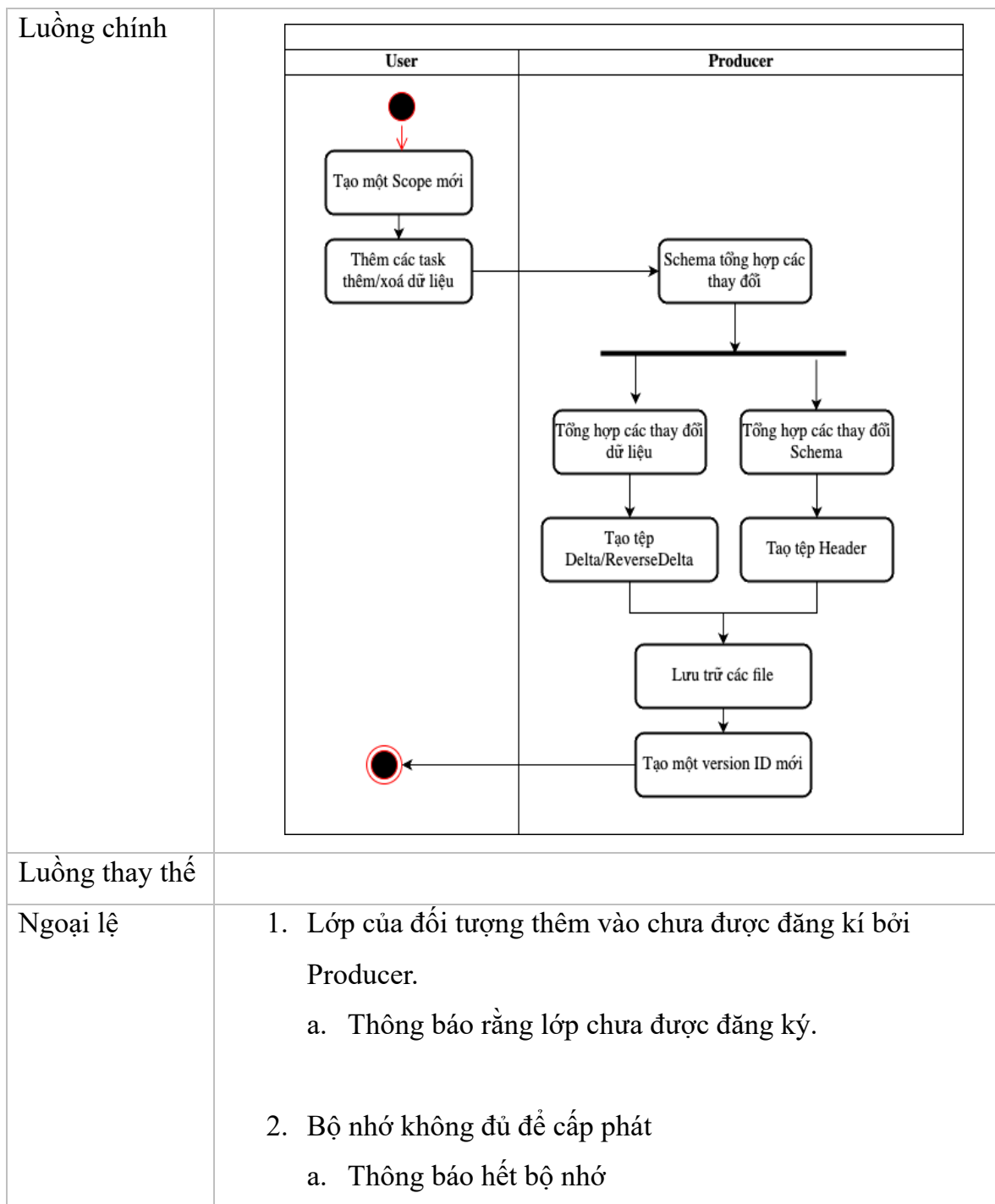
Bảng 3.1 Danh sách các Use-case

STT	Tên Use-case	Ý nghĩa
1	Phát hành dữ liệu mới.	Người dùng thêm/xoá dữ liệu.
2	Rollback version.	Người dùng điều chỉnh Producer về phiên bản cũ hơn.
3	Truy vấn dữ liệu.	Người dùng truy vấn dữ liệu bằng khoá ở Consumer.
4	Cập nhật theo trạng thái mới nhất của Producer.	Consumer chủ động cập nhật để bắt kịp với trạng thái version, dữ liệu của Producer

#### 3.4.1 Use-case phát hành dữ liệu mới

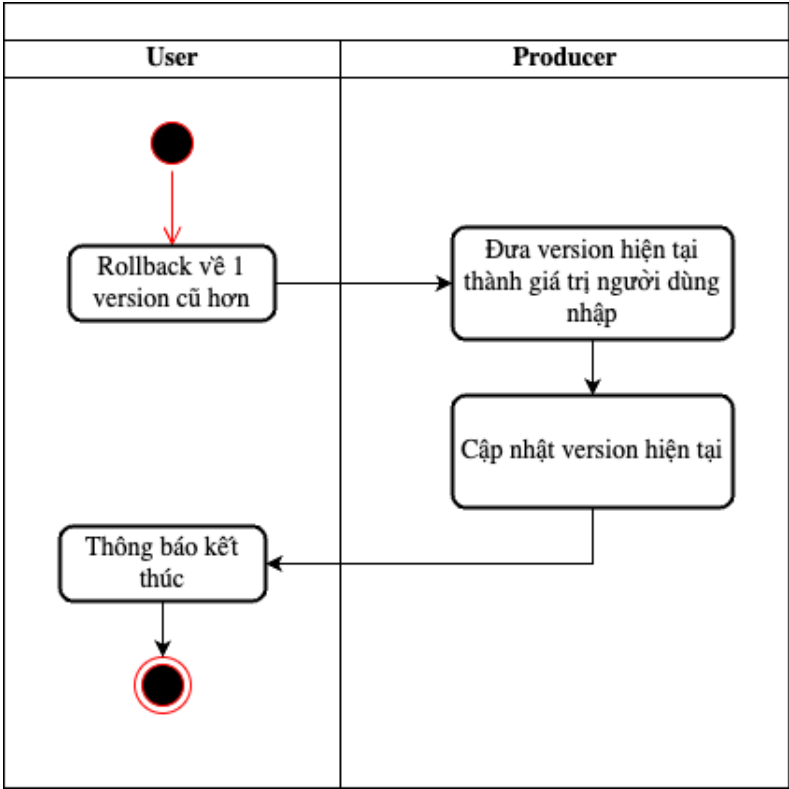
Bảng 3.2 Use-case phát hành phiên bản mới

ID	UC-1: Phát hành dữ liệu mới.
Actor	Người dùng
Mô tả	Người dùng sử dụng Producer để thêm/xoá dữ liệu
Tiền điều kiện	Producer được khởi tạo và chạy thành công
Hậu điều kiện	Sau khi phát hành thành công cập nhập phiên bản mới nhất



### 3.4.2 Use-case rollback về version cũ

Bảng 3.3 Use-case rollback version

ID	UC-2: Rollback version
Actor	Người dùng
Mô tả	Người dùng muốn trở lại một phiên bản trước đó
Tiền điều kiện	Producer được khởi tạo và chạy thành công
Hậu điều kiện	Producer đặt lại phiên bản và thông báo người dùng
Luồng chính	 <pre> sequenceDiagram     participant User     participant Producer     User-&gt;&gt;Rollback: Rollback về 1 version cũ hơn     Rollback-&gt;&gt;Producer:      Producer-&gt;&gt;Input: Đưa version hiện tại thành giá trị người dùng nhập     Input-&gt;&gt;Update: Cập nhật version hiện tại     Update-&gt;&gt;User: Thông báo kết thúc     User-&gt;&gt;End:  </pre> <p>The diagram illustrates the rollback process. It starts with a start node (red circle) leading to a use case 'Rollback về 1 version cũ hơn' (Rollback to an older version) within the 'User' actor. An arrow points from this use case to the 'Producer' actor. Inside the 'Producer' actor, the flow continues through two use cases: 'Đưa version hiện tại thành giá trị người dùng nhập' (Set current version to user input) and 'Cập nhật version hiện tại' (Update current version). An arrow then points from the 'Cập nhật version hiện tại' use case back to the 'User' actor, leading to the use case 'Thông báo kết thúc' (End notification). Finally, an arrow points from 'Thông báo kết thúc' to an end node (red circle).</p>
Luồng thay thế	
Ngoại lệ	

### 3.4.3 Use-case truy vấn dữ liệu

Bảng 3.4 Use-case truy vấn dữ liệu

ID	UC-3: Truy vấn dữ liệu.
Actor	Người dùng.
Mô tả	Người dùng sử dụng Consumer để truy vấn dữ liệu.
Tiền điều kiện	Consumer được khởi tạo và chạy thành công Khoá có tồn tại và trùng với loại dữ liệu muốn nhận.
Hậu điều kiện	Trả về kết quả được giải mã cho người dùng
Luồng chính	<pre> graph TD     Start(( )) --&gt; Check[Kiểm tra địa chỉ của key trong hash_table]     Check --&gt; Decision{Key tồn tại}     Decision -- YES --&gt; Read[Đọc giữ lại tại địa chỉ bộ nhớ]     Read --&gt; Deserialize[Deserialize dữ liệu nhị phân]     Deserialize --&gt; End(( ))     Decision -- NO --&gt; End </pre>
Luồng thay thế	
Ngoại lệ	<ol style="list-style-type: none"> <li>Lớp không hợp lệ với 1 khoá tồn tại. <ol style="list-style-type: none"> <li>Thông báo rằng không thể giải mã dữ liệu nhị phân.</li> </ol> </li> </ol>

### 3.4.5 Use-case cập nhật trạng thái theo Producer

Bảng 3.5 Use-case cập nhật trạng thái theo Producer

ID	UC-4: cập nhật trạng thái theo Producer.
Actor	Consumer.
Mô tả	Consumer giao tiếp với Producer và cập nhật trạng thái.
Tiền điều kiện	Consumer biết địa chỉ và kết nối thành công với Producer.
Hậu điều kiện	
Luồng chính	<pre> graph TD     subgraph Consumer_activity [Consumer activity]         direction TB         subgraph Consumer             Start(( )) --&gt; UpdateVersion[Cập nhập Version hiện tại]             UpdateVersion --&gt; Decision{ }             Decision -- "[latestVersion == currentVersion]" --&gt; End(( ))             Decision -- "[latestVersion != currentVersion]" --&gt; RequestArtifact[Yêu cầu Artifact cho version]             RequestArtifact --&gt; SaveArtifact[Lưu Artifact]             SaveArtifact --&gt; ForkBar[ ]             ForkBar --&gt; Caching[Caching Off-Heap]             ForkBar --&gt; UpdateVersionAgain[Cập nhật lại version hiện tại]             Caching --&gt; End             UpdateVersionAgain --&gt; End         end         subgraph Producer             RequestArtifact --&gt; ReturnVersion[Trả về version mới nhất]             ReturnVersion --&gt; TransferFiles[Chuyển các file về Consumer]             TransferFiles --&gt; SaveArtifact         end     end </pre>
Luồng thay thế	
Ngoại lệ	

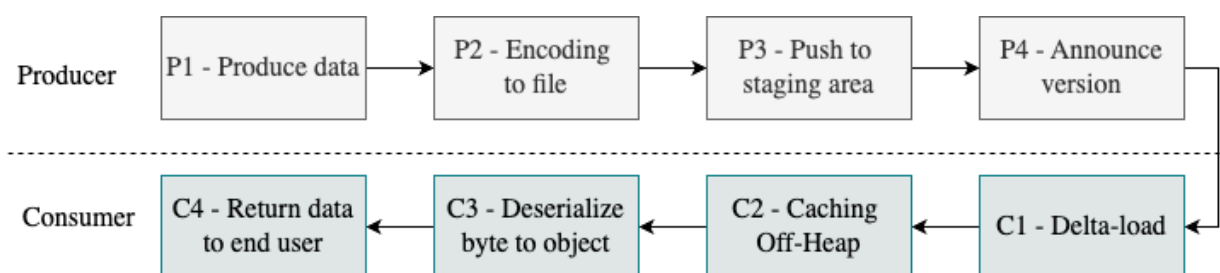


## Chương 4. THIẾT KẾ

Dựa vào những vấn đề và lý thuyết được nêu ở trên, thiết kế của phần mềm ngoài đáp ứng các tính năng (mục 3.1) còn cần có điều sau:

- Sử dụng mô hình pull-based trong giao tiếp Producer – Consumer.
- Tính chính xác và đồng nhất của dữ liệu.
- Khả năng lưu trữ trên RAM lớn và chống phân mảnh.
- An toàn trong môi trường đa luồng.
- Ngoài ra, cần có sự linh hoạt, cho phép người dùng tự triển khai các thành phần liên quan.

Hình dưới đây cho các quá trình chính trong phần mềm (H. 4.1), chi tiết sẽ hơn ở mục 3.1.1:

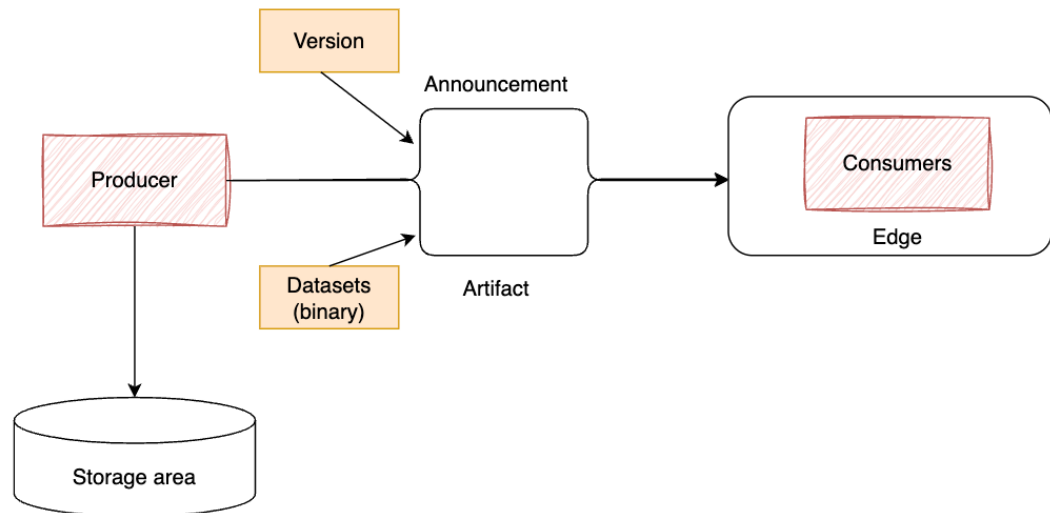


Hình 4.1 Các quá trình chính của phần mềm

## 4.1 Thiết kế kiến trúc

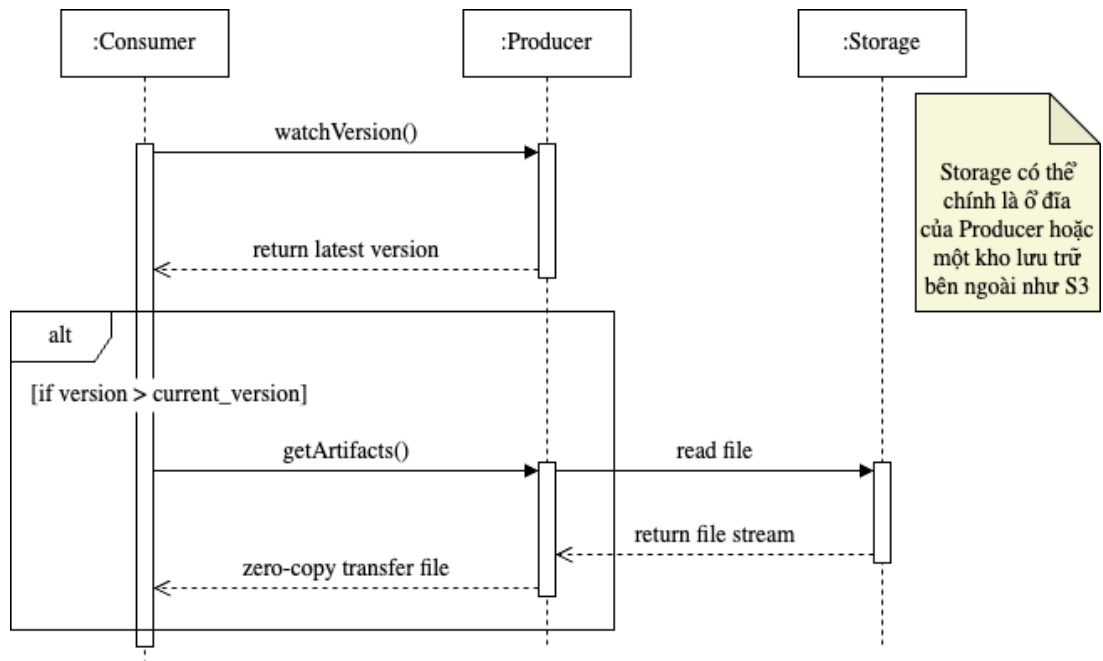
### 4.1.1 Producer - Consumer

Cho một mô hình như sau (H. 4.2), mô hình này lượt tả các thành phần hệ thống một cách đơn giản nhất.



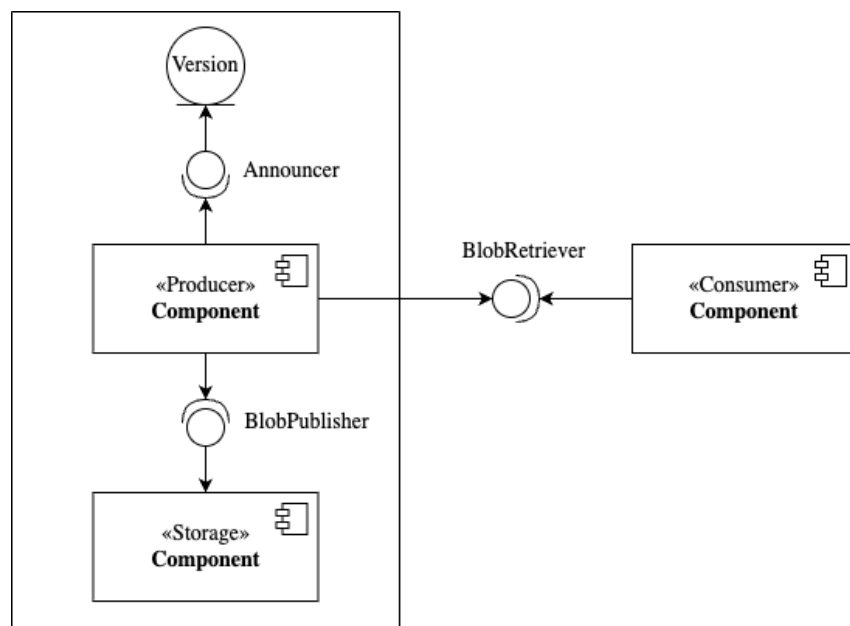
Hình 4.2 Mô hình đơn giản của hệ thống

- Producer và Consumer sẽ duy trì kết nối socket để ghi/nhận dữ liệu. Đối với mô hình pull-based, socket được sử dụng vì độ trễ thấp.
- Có 2 quá trình quan trọng trong giao tiếp giữa Producer – Consumer là thông báo trạng thái (Announcement) và truyền tải nội dung (Transferring Artifact).
  - Đối với Announcement, Consumer sẽ liên tục kiểm tra trạng thái phiên bản mới nhất với Producer.
  - Nếu có phiên bản mới nhất, Consumer sẽ tải Artifact về để cập nhật những thay đổi.
- Artifact: tệp nhị phân, chứa dữ liệu được mã hoá từ Producer, chi tiết về Artifact sẽ được nêu sau (mục 3.2.2).



Hình 4.3 Sơ đồ tuần tự Producer – Consumer

- Tham gia vào 2 quá trình trên (Annoucement và Transferring Artifacts) có 3 interface cần được triển khai:
  - Announcer: thông báo phiên bản mới nhất.
  - BlobPublisher: lưu và phát hành Artifacts.
  - BlobRetriever: tải về các artifact từ phiên bản hiện tại của Consumer tới phiên bản mới nhất được phát hành.



Hình 4.4 Mô tả interface của các component liên quan cần triển khai

a. Schema

Tương như @Entity trong JPA [36], Producer sẽ chỉ có thể thao tác với các đối tượng là lớp đã được đăng ký. Đối với mỗi lớp đăng ký, Producer sẽ tạo ra Schema tương ứng để quản lý.

```
producer.register(ClassA.class, ClassB.class);
```

b. Producer

Là một thư viện Java dùng để người dùng chỉnh sửa tập dữ liệu và phân tán đến các máy biên.

Đối với quá trình “P1 – Produce data”, Producer cung cấp cho người dùng 1 Scope (mỗi khi gọi hàm produce()), các thao tác dữ liệu sẽ được chuyển sang dạng byte và ghi nhớ trên bộ nhớ, Scope sẽ kết thúc khi hoàn thành hết các “task” và ghi dữ liệu vào tệp nhị phân.

```
producer.populate(task -> {  
    task.addObject("key-a", objA);  
    task.addObject("key-b", objB);  
})
```

Quá trình “P2 – Encoding to file”, mỗi Schema sẽ có 1 cấu trúc dữ liệu lưu trữ dữ liệu riêng, sau khi kết thúc một Scope, sẽ tổng hợp các dữ liệu và ghi vào tệp. Các Schema có thể được xử lý ở các luồng khác nhau giúp tăng tốc độ ghi.

Quá trình “P3 – Push to Staging area” đưa các tệp nhị phân được tạo ở P2 vào kho lưu trữ (có thể là S3 hoặc ổ đĩa máy).

Quá trình “P4 – Announce version”, Producer sẽ lưu phiên bản hiện tại và trả về cho Consumer khi được yêu cầu. Các phiên bản sẽ tăng tuần tự từ 0, 1, 2, ... N.

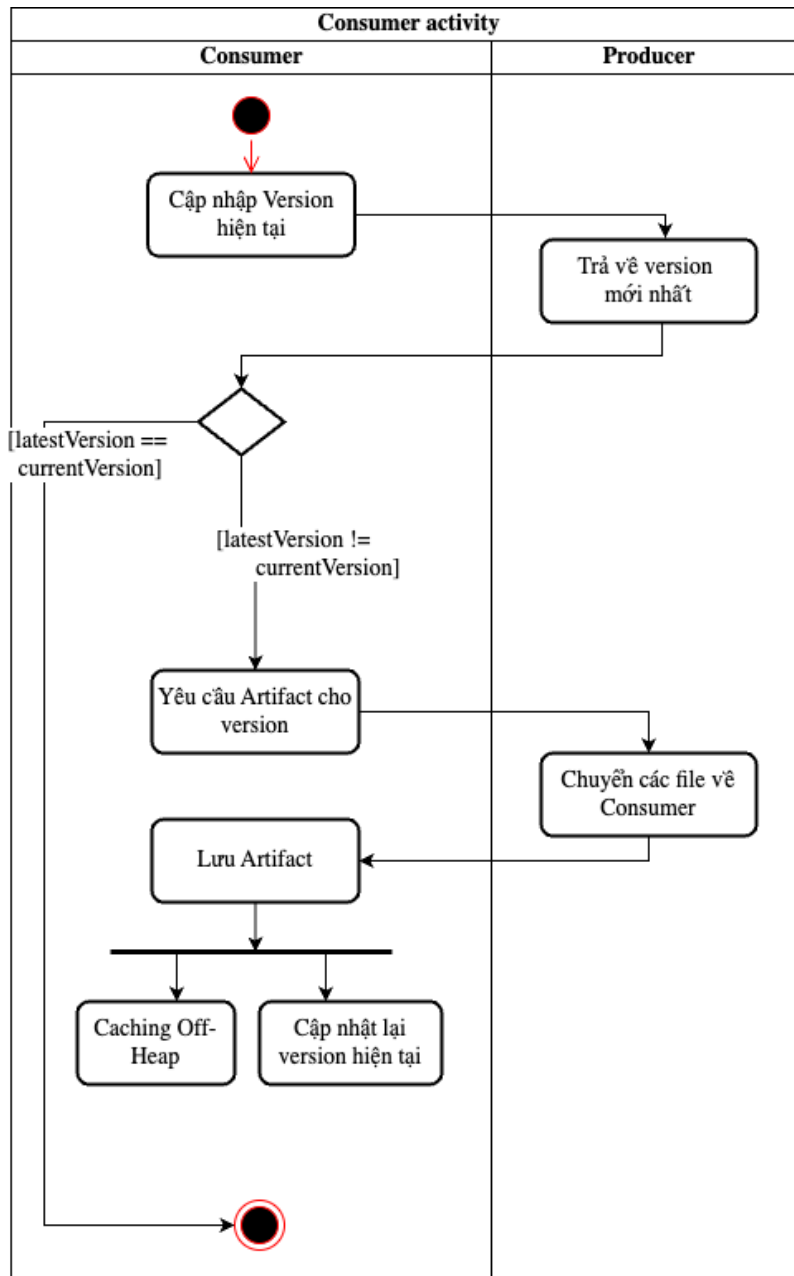
c. Consumer

Là một thư viện Java để kết nối với Producer thông qua socket. Chịu trách nhiệm ghi nhận thay đổi, cập nhật và giúp người dùng truy vấn dữ liệu thông qua khoá (key). Consumer chỉ có thể đọc, không thể chỉnh sửa dữ liệu được cache.

Quá trình “C1 – Delta-load”, Consumer sẽ tải các tệp Header, Delta hoặc ReverseDelta tương ứng với từng phiên bản cho đến khi trùng với phiên bản của Producer. Khác với Producer, Consumer sử dụng Blocking I/O. Ở Consumer, sau khi tải các Artifact vào Schema, ta có thể chia ra để xử lý dữ liệu mỗi Schema đồng thời trên nhiều luồng khác nhau.

- Nếu phiên bản hiện tại của Consumer bé hơn của Producer, tải Header và Delta.
- Nếu phiên bản hiện tại của Consumer lớn hơn của Producer, tải Header và ReverseDelta.

Quá trình “C2 – Caching Off-Heap” sẽ dùng các dữ liệu được tải vào Schema chuyển đổi thành các bản ghi và lưu vào RAM (mục 3.2.4). Khi người dùng yêu cầu một bản ghi, quá trình “C3 – Deserialize byte to Object” dùng thư viện Kryo để chuyển dữ liệu byte sang một lớp trong Java.



Hình 4.5 sơ đồ hoạt động Consumer

d. Staging area

Là vùng lưu trữ Artifacts (tương tự repository của Github). Có thể lưu trữ dữ liệu ngay tại máy Producer hoặc sử dụng một kho lưu trữ bên ngoài.

## 4.2 Artifact

Artifact (tài liệu) là 3 loại tệp nhị phân khác nhau lần lượt là: Header, Delta và ReverseDelta.

### 4.2.1 Tệp Header

Header là một tệp, mà trong đó Consumer sẽ dùng thông tin cần để cài đặt, chỉnh sửa trạng thái nhằm đảm bảo hoạt động tương thích với dữ liệu phiên bản hiện tại. Header giúp giải quyết vấn đề về đăng ký ID cho các lớp của Kryo, nhằm duy trì các cài đặt của Kryo là giống nhau ở cả 2 phía.

Giả sử ở Producer, ClassA đăng ký thêm ClassB, tệp header sẽ ghi nhận tất cả các thay đổi này và lan truyền tới Consumer, đảm bảo việc quá trình S/D không bị lỗi.

Cấu trúc của 1 tệp header, tệp có định dạng “header-`{version}`”, cấu trúc gồm:

- HeaderID: chiếm 4 byte đầu tiên, dùng để chỉ thị đây là tệp header cần dùng của thư viện.
- Hai số ngẫu nhiên (kiểu long): chiếm 16 byte tiếp theo, dùng để xác nhận các tệp delta, reversedelta có phải cùng artifact của header.
- Danh sách các lớp được đăng ký là một Schema của thư viện bởi Producer.
- Danh sách các lớp đăng ký vào Kryo, tên lớp theo sau đó là ID.
- Header chỉ ghi lại những Schema, Kryo Registration mới.

Bảng 4.1 Ví dụ cấu trúc tệp Header

ID (4 byte)	Random tag (8 byte)	Random tag (8 byte)	List of Schema	List of Kryo registration
1	36	112	ClassA, ClassB,	ClassA, 1, ClassB, 2, String, 3

#### 4.2.2 Tệp Delta, ReverseDelta

Các tệp Delta, ReverseDelta cho phép triển khai delta-load khi, 2 loại tệp này sẽ có cùng cấu trúc dữ liệu.

- Delta: dùng để cập nhật lên phiên bản khác
- ReverseDelta: dùng để trở về phiên bản trước đó
- Khi có phiên bản mới, người dùng thường thêm/xoá vài dữ liệu (không hỗ trợ chỉnh sửa vì điều đó là không khả thi khi xử lý dữ liệu ở dạng nhị phân). Producer sẽ tổng hợp các thay đổi này và ghi vào tệp Delta, tệp ReverseDelta là một tệp dịch ngược lại của delta (thêm thành xoá, xoá thành thêm).

Cấu trúc của 1 tệp delta:

- BlobID: chiếm 4 byte đầu, biểu diễn 1 số nguyên định nghĩa rằng đây là tệp delta của thư viện.
- Hai số nguyên (64-bits): chiếm 16 byte tiếp theo, là 2 ngẫu nhiên
- Theo sau đó lần lượt là thông tin của các Schema, cụ thể:
  - Tên Schema.
  - Số lượng đối tượng thêm vào.
  - Danh sách các khoá (key) và đối tượng được thêm vào.
  - Danh sách các khoá (key) và đối tượng được xoá.
- Tệp reversedelta thì ngược lại, đối tượng thêm vào trở thành đối tượng xoá.



Bảng 4.2 Cấu trúc tệp Delta

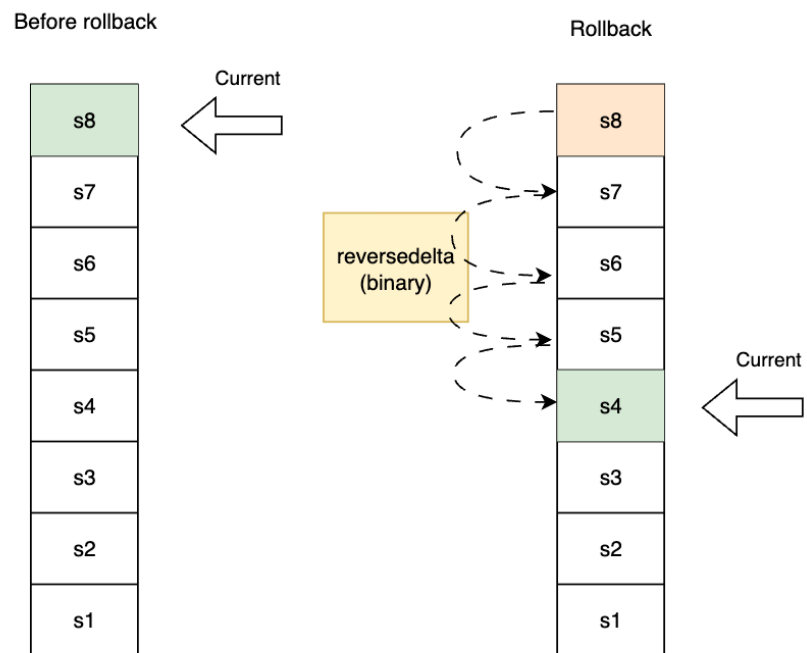
	ID	Random tag	Random tag	Schema name	Additional	Removals
Format	4 byte	8 byte	8 byte	String format	LEB128 fixed-length	LEB128 fixed-length
Sample	1	36	112	ClassA	(key1-val1), ... (keyN – valN)	keyDel1 ... keyDelN

Bảng 4.3 Cấu trúc tệp ReverseDelta

	ID	Random tag	Random tag	Schema name	Removals	Additional
Format	4 byte	8 byte	8 byte	String	LEB128 fixed-length	LEB128 fixed-length
Sample	1	36	112	ClassA	key1 ... keyN	(keyDel1 – valDel1) ... (keyDelN – valDelN)

### 4.2.3 Cách Rollback

Khi Producer đã đổi sang một trạng thái cũ hơn, Consumer phải dùng những tệp ReversedDelta để sửa dữ liệu hiện tại đến khi nào giống với phiên bản hiện tại trên Producer.



Hình 4.6 Minh hoạ rollback

## 4.3 Java Off-heap

Sử dụng Unsafe<sup>1</sup> API để uỷ thác OS giúp cấp phát và thu hồi bộ nhớ dùng để lưu trữ dữ liệu các bản ghi. Unsafe sẽ trả về địa chỉ của khối ô nhớ được cấp phát, sử dụng địa chỉ này để ghi/đọc dữ liệu. Lớp OSMemory cung cấp vài phương thức cơ bản như sau:

```
public long allocate(long size) {
    if (size <= 0)
        throw new IllegalArgumentException("Illegal required size: " + size);

    long address = UNSAFE.allocateMemory( bytes: size);
    if (address == 0)
        throw new OutOfMemoryError( s: "Not enough memory; required size: " + size / 1024 + "KiB");

    this.nativeMemUsed.addAndGet( delta: size);

    return address;
}

public byte readByte(long address) { 6 usages
    assert SKIP_ASSERT || address > 0;
    return UNSAFE.getBytes(address);
}

public void copyMemory(long toAddr, byte[] arr, int arrOffset, int len) { 3 usages
    assert SKIP_ASSERT || toAddr > 0;
    assert SKIP_ASSERT || arrOffset >= 0 && len >= 0;

    if (arrOffset + len > arr.length)
        throw new IllegalArgumentException("Invalid offset/len; array's length: " + arr.length);

    if (len == 0)
        return;

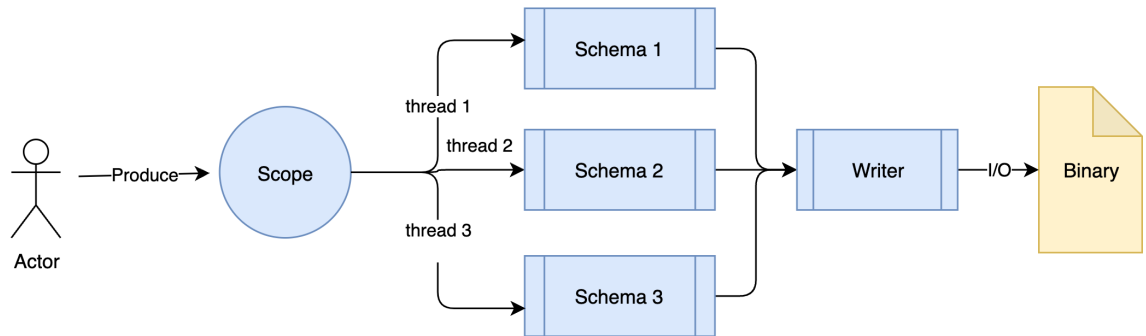
    UNSAFE.copyMemory( srcBase: arr, srcOffset: ARRAY_BYTE_BASE_OFFSET + arrOffset, destBase: null, destOffset: toAddr, bytes: len);
}
```

---

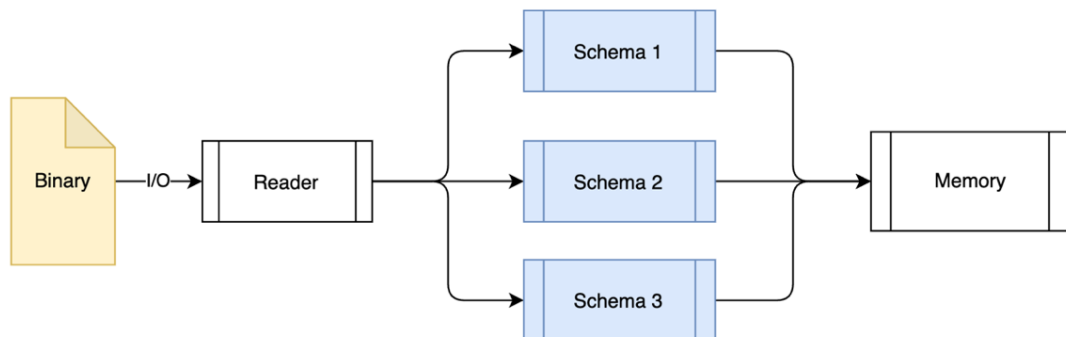
<sup>1</sup> Là lớp thuộc package sun.misc.Unsafe

## 4.4 Java NIO

Sử dụng `FileChannel` để ánh xạ Artifact lên RAM khi đọc/ghi Artifact, `FileChannel` cho phép xử lý một tệp bằng nhiều luồng khác nhau tương ứng với các Schema. Mỗi Schema được xử lý độc lập với nhau, và sẽ được ghi/đọc trên mỗi phân vùng riêng biệt của tệp Delta, ReverseDelta.



Hình 4.7 Mô tả xử lý ghi Schema và Artifact



Hình 4.8 Mô tả xử lý đọc Artifact và Schema

## 4.5 LEB128 và Kryo

LEB128 là kỹ thuật tối ưu không gian lưu trữ cho số nguyên được trình bày ở chương 2.3, triển khai lớp `VarInt` dùng giải thuật LEB128 và `ZigZag` để đọc/ghi số nguyên ở các cấu trúc dữ liệu như: `byte[]`, `ByteBuffer`, `InputStream`, `OutputStream`, v.v..

```

public void writeVarInt(byte[] data, int pos, int value) {
    int ui32 = Zigzag.encode(value);
    do {
        byte b0 = (byte) (ui32 & 0x7f);
        ui32 >>= 7;
        if (ui32 != 0) {
            b0 = setMsbContinuation(b0);
        }
        data[pos++] = b0;
    } while (ui32 != 0);
}

public int readVarInt(byte[] data, int pos) {
    int ui32 = 0, shift = 0;
    while (true) {
        byte b0 = data[pos++];
        ui32 |= (b0 & 0x7f) << shift;
        shift += 7;

        if ((b0 & 0x80) == 0) {
            break;
        }
    }

    return Zigzag.decode(ui32);
}

```

Cài đặt thư viện Kryo bằng cách khai báo trong build.gradle như sau:

```

dependencies {
    implementation 'com.esotericsoftware:kryo:5.6.2'
}

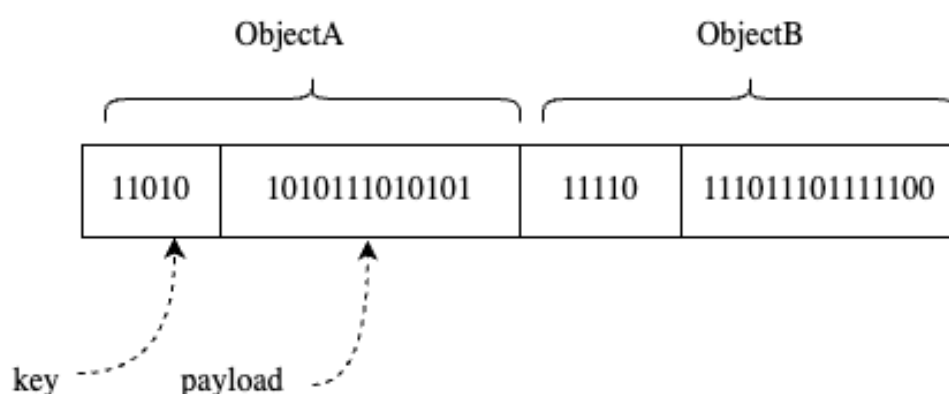
```

Mỗi bản ghi trong tệp Delta sẽ có một số lượng byte nhất định và có định dạng như sau:

N (độ dài)	N byte tiếp theo, dữ liệu của bản ghi
------------	---------------------------------------

Tạm gọi cấu trúc dữ liệu trên là một block<sup>2</sup>. Đối với dữ liệu của block (phần thứ 2), ta xác định kiểu dữ liệu của Java và dùng Kryo để chuyển Object sang dạng byte.

Đối với việc đánh dấu độ dài của block, thay vì dùng 4 byte cho một số nguyên dương để biểu diễn độ dài của dữ liệu, có thể thay thế dùng LEB128 để giảm không gian lưu trữ, thông thường sẽ giảm khoảng 2-3 byte vì dữ liệu sau khi qua thông qua Kryo thường không dài quá 16383 byte.



Hình 4.9 Ví dụ định dạng bản ghi trong tệp Delta

---

<sup>2</sup> Chỉ một khối cấu trúc dữ liệu bao gồm 1 số nguyên N, và N byte theo sau

Mã giả khi ghi/đọc một block từ Artifact:

```
write():
    len = varint.sizeOfInt(len)
    varint.writeVarInt(output, len)
    mem_copy(object, 0, stream, len)

read():
    len = varint.readVarInt(input)
    read_position = position + varint.sizeOfInt(len)
    mem_copy(input, read_position, destination, len)
    return destination
```

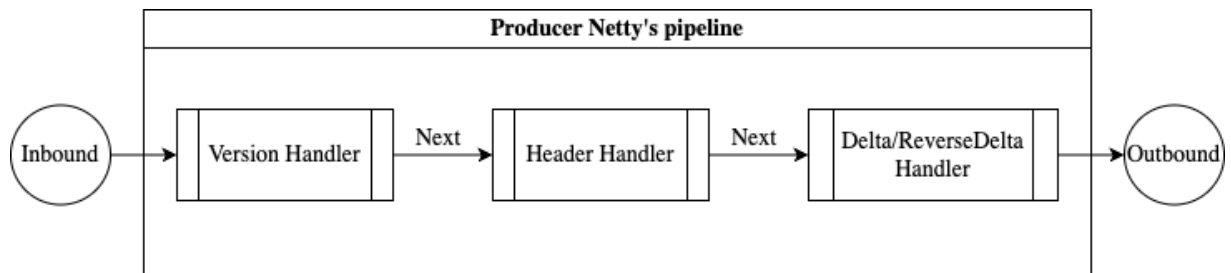
## 4.6 Netty

Trong mô hình pull-based, Producer luôn phải lắng nghe mọi kết nối của Consumer, giao thức TCP/IP được tin dùng vì:

- Chi phí duy trì kết nối thấp.
- Tốc độ ghi/đọc dữ liệu cao.

Sử dụng Netty triển khai một máy chủ Non-Blocking I/O ở Producer, bên cạnh đó triển khai 3 NettyHandler cho phép xử lý 3 tác vụ khác nhau lần lượt:

- VersionHandler: trả về kết quả phiên bản (version) hiện tại của Producer.
- HeaderBlobHandler: trả về tập Header tương ứng yêu cầu của Consumer.
- DeltaBlobHandler: trả về tập Delta/ReverseDelta tương ứng với version yêu cầu của Consumer.



Hình 4.10 Netty's pipeline của Producer

Netty cung cấp lớp DefaultFileRegion thực hiện zero-copy thông qua FileChannel, sử dụng khi truyền các tệp Artifact.

```
raf = new RandomAccessFile(filepathToFile(), "r");
context.writeAndFlush(new DefaultFileRegion(raf.getChannel(), 0, len));
```



## 4.7 Hash-Table và SLAB

### 4.7.1 Lưu trữ dữ liệu

#### a. SLAB

Sử dụng SLAB giúp cấp phát bộ nhớ động, đối với mỗi chunk ta sẽ dùng một vài byte để lưu metadata [27]. Theo cấu trúc SLAB, khi phân bổ bộ nhớ RAM, ta phải phân bổ toàn bộ 1 Page trong SlabClass, dùng Java Unsafe để uỷ thác OS phân bổ 1 Page.

```
alloc_address = UNSAFE.allocate(chunk_size * chunk_per_page)
slabclass[pageID]->address = alloc_address
```

Công thức tính địa chỉ của một chunk<sup>3</sup>:

$$A = S + (i * L)$$

Trong đó:

- A: địa chỉ bộ nhớ của chunk.
- S: địa chỉ bắt đầu của SlabPage.
- i: chỉ số của chunk trong một SlabPage.
- L: kích thước của chunk trong SlabClass.

---

<sup>3</sup> Đơn vị nhỏ nhất trong SLAB, dùng để lưu trữ dữ liệu một Object.

Mỗi Object trong Java sẽ dùng 1 chunk để lưu trữ, cấu trúc chi tiết của 1 chunk:

- 9 bytes đầu được dùng để lưu thông tin về vị trí hiện tại của chunk, dùng khi giải phóng hoặc di chuyển 1 chunk.
  - 1 byte chỉ giá trị thứ tự SlabClass.
  - 4 byte chỉ offset của chunk trong SlabClass.
  - 4 byte chỉ vị trí của dùng để lưu địa chỉ chunk trong Hash-Table (dùng trong quá trình cân bằng).
- Còn lại dùng để lưu các byte dữ liệu của bản ghi sau khi đọc từ tệp.

b. Open-Addressing Hash-Table

- Các khoá được truyền vào dưới dạng String, sau đó được chuyển thành kiểu dữ liệu byte và thông qua hàm mã hoá Murmur3 (mục 2.6.1) để đưa ra được 1 khoá là số nguyên.
  - `fmix_murmur3(str)` cho kết quả số nguyên dương 32-bit, giá trị số nguyên này chính là khoá của bảng băm.
- Sử dụng bảng băm Open-Addressing để lưu giữ/truy vấn khoá và địa chỉ bộ nhớ (địa chỉ 1 chunk). Sử dụng cơ chế Double-Hashing để giải quyết vấn đề va chạm, load factor ở mức 0.7
- Hàm  $f(x)$  và  $f'(x)$  tương ứng 2 công thức sau:

$$f(x) = x \bmod C$$

$$f'(x) = (p - x \bmod p)$$

- Trong đó:
- $x$ : giá trị số nguyên dương (key)
- $C$ : kích thước của bảng băm
- $p$ : một số nguyên tố bất kì, để xác định vị trí mới.
- Mã giả như sau:

```

find_available_bucket(key):
    x = h1(key), probe = h2(key)
    while table[x] != -1
        OR table[x] is not null
        OR table[x]->key == key:
            if table[x] == DELETE
                break
            x = (x + probe) % table_size
    return x

```

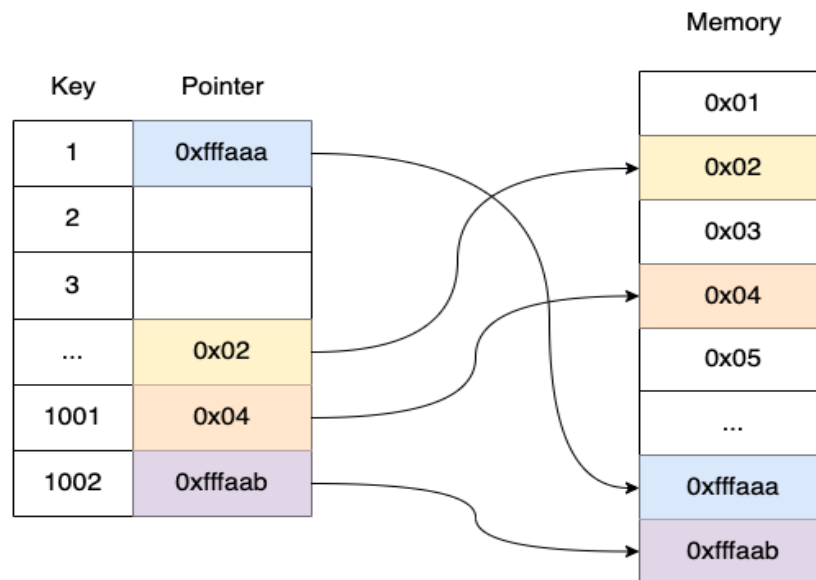
### c. Hash-Table và RAM

Khi lưu một dữ liệu mới:

- SLAB cấp phát một chunk với địa chỉ bộ nhớ cho Object.
- Ghi khóa – địa chỉ ô nhớ vào trong Hash-Table.
- Ghi Object vào ô nhớ.

Khi xoá một dữ liệu:

- Đưa chunk vào Freelist
- Xoá bản ghi trên Hash-Table dựa vào khoá.

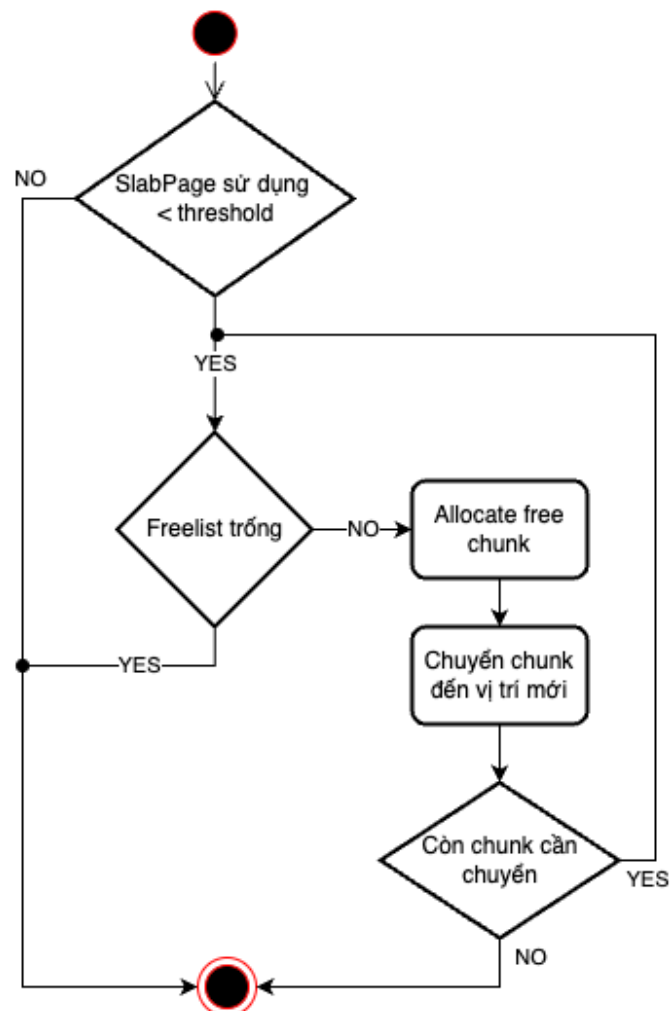


Hình 4.11 Quan hệ Hash-Table và Memory

### d. SLAB re-balancing

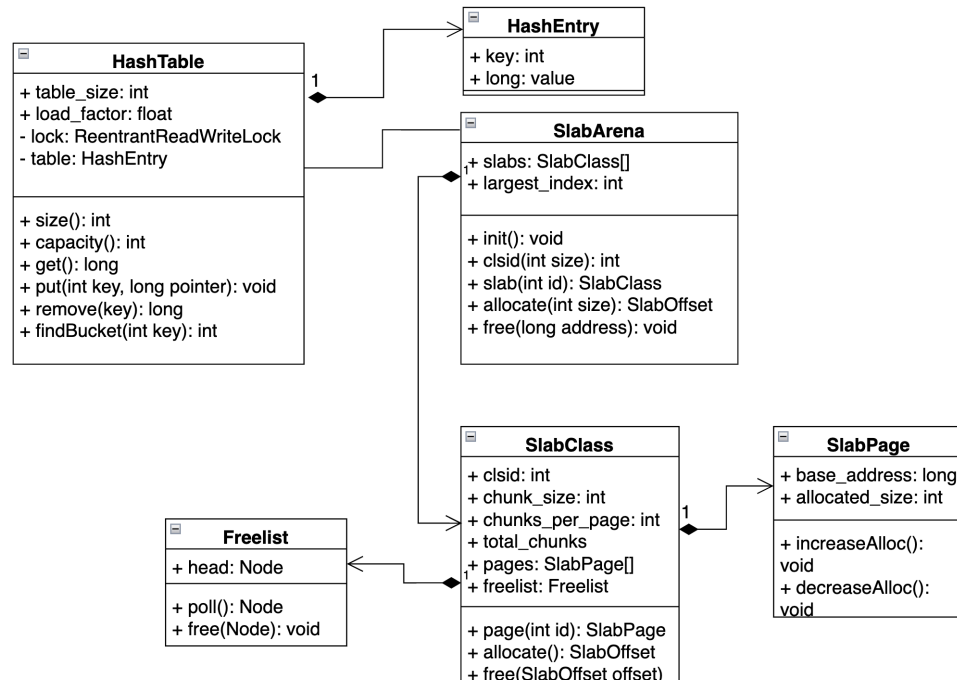
Chuyển các chunk từ một SlabPage dùng dưới mức cho phép (chỉ có 30% chunk được sử dụng) sang các SlabPage khác.

- S1: Kiểm tra tỉ lệ sử dụng của 1 SlabPage, nếu cao hơn mức cài đặt, tới S7.
- S2: Kiểm tra Freelist còn trống, nếu không tới bước S7.
- S3: Yêu cầu 1 chunk mới từ Freelist không phải của SlabPage này.
- S4: Sao chép dữ liệu sang chunk mới.
- S5: Nếu SlabPage trống, tới bước 7
- S6: Giải phóng vùng nhớ SlabPage
- S7: Kết thúc.



Hình 4.12 Sơ đồ SLAB re-balancing

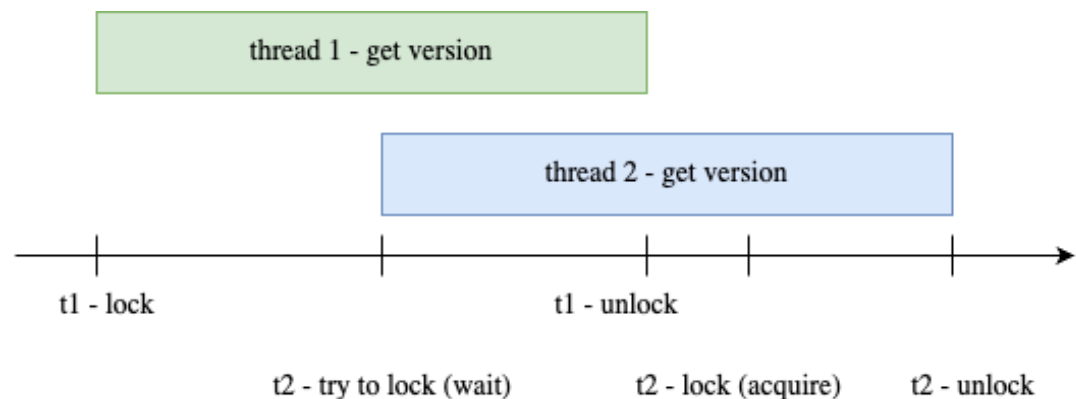
#### e. Sơ đồ lớp SLAB và Hash-Table



Hình 4.13 Sơ đồ lớp của Hash-Table và SLAB trong quản lý bộ nhớ

#### 4.7.2 Đồng bộ trong đa luồng

Với Consumer, khi yêu cầu kiểm tra phiên bản mới nhất và cập nhật Consumer sẽ yêu cầu 1 ReentrantLock [37], ngăn tất cả các thread khác cố thực hiện việc kiểm tra phiên bản.

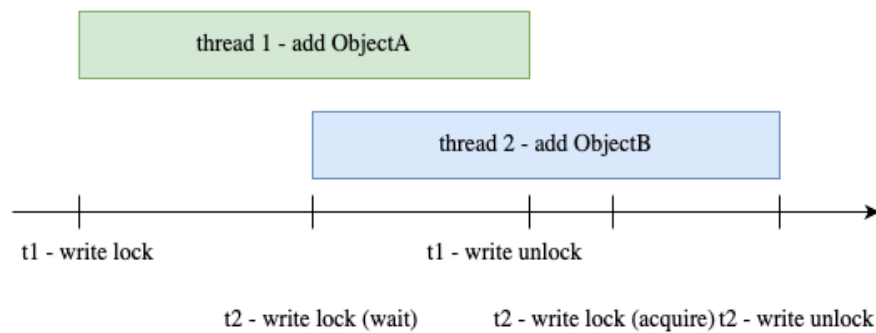


Hình 4.14 ReentrantLock khi kiểm tra phiên bản

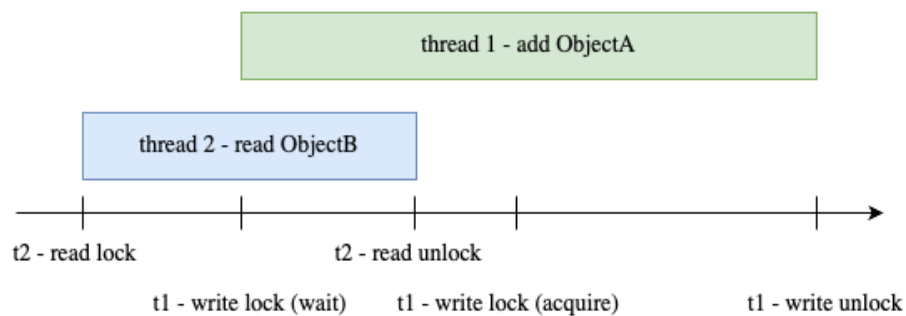
Đối với Hash-Table và SLAB có hai thao tác chính là đọc và ghi dữ liệu, dùng ReentrantLock sẽ làm giảm hiệu năng vì chỉ cho phép một thread duy nhất thao tác với dữ liệu, các tác vụ sẽ chặn lẫn nhau. Sử dụng ReadWriteLock để đảm bảo tính nhất quán khi cần đọc và ghi đối với một cấu trúc dữ liệu.

- ReadLock: nếu không có thread nào đang chiếm hoặc yêu cầu WriteLock, tất cả mọi thread đều có thể sử dụng ReadLock, nghĩa là có thể đọc dữ liệu.
- WriteLock: 1 thread có thể chiếm WriteLock khi và chỉ khi không còn ReadLock và WriteLock nào đang bị chiếm bởi thread khác.

WriteLock block ReadLock



ReadLock block WriteLock



Hình 4.15 Minh họa ReadWriteLock

## Chương 5. TRIỂN KHAI ỨNG DỤNG

Phần mềm sẽ được biên dịch ra thành một tệp JAR có tên là Cobra để làm thư viện trong Java, sẽ có 2 thành phần quan trọng được triển khai là Producer và Consumer.

Mã nguồn Github: <https://github.com/vishifu-default/cobra>.

### 5.1 Producer

Import thư viện JAR vào dự án, và khởi tạo Producer và khởi tạo Producer:

```
String dir = System.getenv("PUBLISH_DIR");
Path publishDir = Paths.get(dir);

CobraProducer.BlobPublisher publisher = new
    FilesystemPublisher(publishDir);
CobraProducer.Announcer announcer = new
    FilesystemAnnouncer(publishDir);
CobraProducer.BlobStagger stagger = new
    FilesystemBlobStagger();

CobraProducer producer = CobraProducer.fromBuilder()
    .withBlobPublisher(publisher)
    .withAnnouncer(announcer)
    .withLocalPort(7070)
    .withBlobStagger(stagger)
    .withBlobStorePath(publishDir)
    .withRestoreIfAvailable(true)
    .buildSimple();

producer.registerModel(Movie.class);
producer.bootstrap();
```

Có 2 option khi khởi chạy Java -jar cần quan tâm.

- -Xmx: cài đặt bộ kích thước tối đa của Heap cho JVM.
- -XX:MaxDirectMemorySize: cài đặt kích thước tối đa cho bộ nhớ JVM có thể sử dụng.

```

Error: A fatal exception has occurred. Program will exit.
[ec2-user@ip-172-31-20-152 ~]$ java -Xmx2g -XX:MaxDirectMemorySize=8g -jar SAMPLE_PRODUCER-1.0-SNAPSHOT.jar
2025-01-03T06:50:19.048 [INFO] [org.example.producer.Producer] - Publishing directory: ./misc/publish-dir
2025-01-03T06:50:19.096 [INFO] [org.cobra.core.serialization.SerdeClassResolver] - register self registration 9; clazz: java.lang.String
2025-01-03T06:50:19.176 [INFO] [org.cobra.core.serialization.SerdeClassResolver] - register self registration 10; clazz: org.example.datamodel.Publisher
2025-01-03T06:50:19.178 [INFO] [org.cobra.core.serialization.SerdeClassResolver] - register self registration 11; clazz: org.example.datamodel.Movie
2025-01-03T06:50:19.179 [INFO] [org.cobra.core.serialization.SerdeClassResolver] - register self registration 11; clazz: org.example.datamodel.Actor
2025-01-03T06:50:19.263 [WARN] [io.netty.bootstrap.ServerBootstrap] - Unknown channel option 'TCP_NODELAY' for channel '[id: 0xe82c53a1]'
2025-01-03T06:50:19.263 [WARN] [io.netty.bootstrap.ServerBootstrap] - Unknown channel option 'SO_KEEPALIVE' for channel '[id: 0xe82c53a1]'

```

Hình 5.1 Ví dụ khởi chạy Producer trên EC2

## 5.2 Consumer

Application SpringBoot dùng làm RestApi, khởi tạo Consumer

```

public CacheService() {
    Path cacheDir = Paths.get(consumePath);
    CobraConsumer.BlobRetriever blobRetriever =
        new FileSystemBlobRetriever(cacheDir);

    InetAddress addr = new InetSocketAddress(host,
port);
    consumer = CobraConsumer.fromBuilder()
        .withInetAddress(addr)
        .withBlobRetriever(blobRetriever)
        .build();

    consumer.poll();

    api = new CobraRecordApi(consumer);
}

```

Khởi tạo RecordApi, đây là interface cho phép truy suất dữ liệu thông qua khoá.

```

api = new CobraRecordApi(consumer);

```



## Chương 6. Đánh giá giải pháp

Thực hiện đánh giá giải pháp và so sánh với NetflixHollow (1 thư viện của Netflix phát triển) dựa trên các tiêu chí: hiệu năng của ứng dụng, bộ nhớ sử dụng và CPU được dùng.

### 6.1 Chuẩn bị môi trường

Các chương trình được chạy trên hạ tầng AWS EC2 với cấu hình như sau:

Bảng 6.1 Cấu hình AWS EC2

Instance type	t3.small
Core	2
CPU	2vCPU
RAM	2 GB
SSD	8 GB

Tập dữ liệu chuẩn bị gồm 18 tệp CSV khoảng 5Mb<sup>4</sup>, là danh sách bao gồm 90.000 bộ phim.

Sử dụng Grafana để giám sát hoạt động của chương trình (bộ nhớ, CPU), và K6 để thực hiện kiểm thử hiệu năng.

---

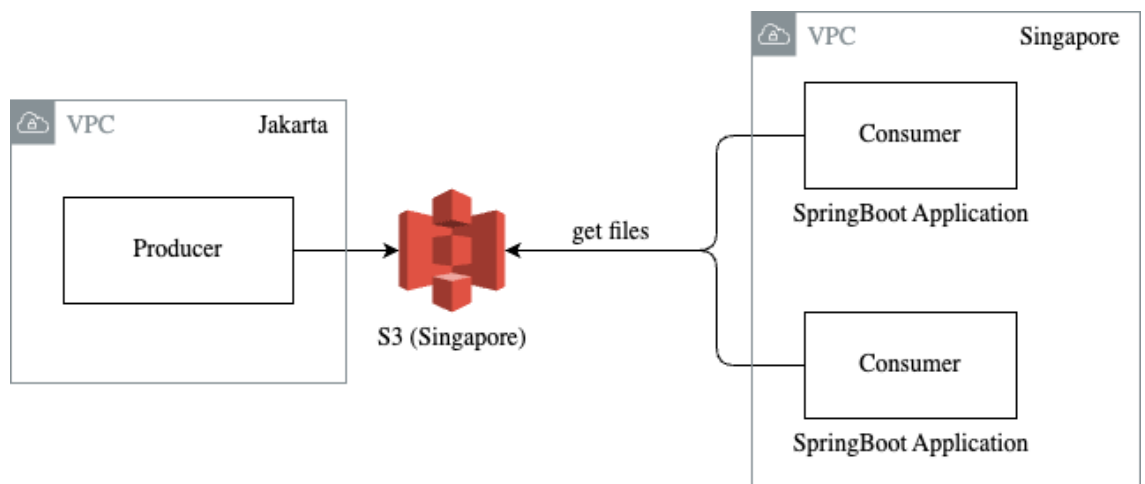
<sup>4</sup> <http://files.grouplens.org/datasets/movielens/ml-latest.zip>

## 6.2 Triển khai

### 6.2.1 Triển khai NetflixHollow

Cài đặt package `com.netflix.hollow:hollow` từ Maven.

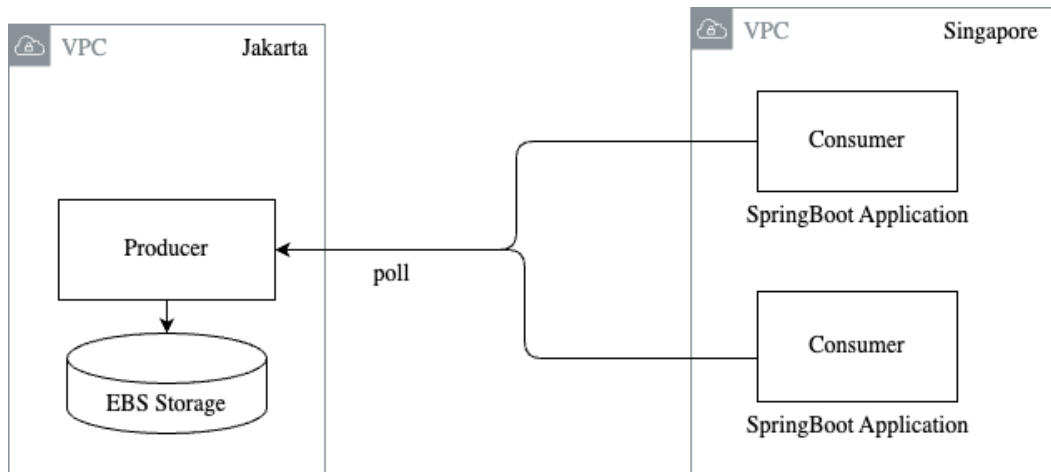
Dựa theo kiến trúc mà Hollow triển khai, Producer và Consumer không có cơ chế giao tiếp với nhau, nhóm sẽ dùng AWS S3 (region Singapore) để làm kho lưu trữ. Producer sẽ đưa tất cả các tệp chưa dữ liệu vào S3, Consumer sẽ tự tải về.



Hình 6.1 Mô hình triển khai Netflix Hollow

### 6.2.2 Triển khai Cobra

Triển khai 2 thành phần tương tự chương 5, Producer ở AWS Region Jakarta và Producer ở AWS Region Singapore.



Hình 6.2 Mô hình triển khai Cobra

## 6.3 Đánh giá hiệu năng

### 6.3.1 Sử dụng bộ nhớ

Có 2 tiêu chí quan trọng cần đánh giá cho bộ nhớ:

- Memory Usage: tổng bộ nhớ được cấp phát cho ứng dụng.
- Memory RSS: tổng bộ nhớ vật lý đang hoạt động của ứng dụng.



Hình 6.3 Đồ thị tiêu thụ bộ nhớ của Consumer

Đối với mức tiêu thụ RAM của Consumer

- Cobra-Consumer sử dụng bộ nhớ ít hơn trung bình khoảng 10% so với NetflixHollow.



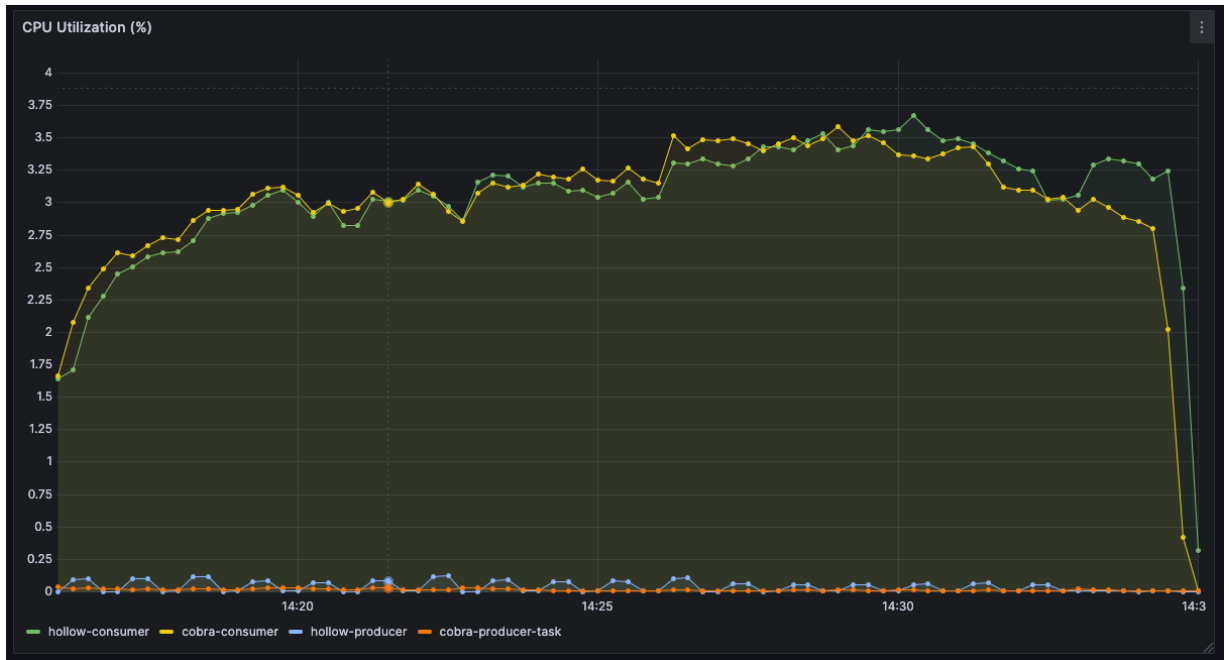
Hình 6.4 Đồ thị tiêu thụ bộ nhớ của Producer

Đối với mức tiêu thụ RAM của Producer:

- Trung bình Memory Usages của Cobra cao hơn NetflixHollow 10%, tuy nhiên chỉ số của đồ thị Memory RSS là thấp hơn.

### 6.3.2 Sử dụng CPU

Mức phần trăm sử dụng CPU của cả 2 gần như tương đương nhau, không có sự chênh lệch.



Hình 6.5 Đồ thị mức phần trăm sử dụng CPU

### 6.3.3 Hiệu năng

Kịch bản chạy K6, trong đó “duration” chỉ thời gian chạy, “target” chỉ số lượng VUS (Virtual User):

```
stages: [
  // ramp up to average load of 20 virtual users
  {duration: '30s', target: 10},
  // maintain load
  {duration: '1m', target: 20},
  {duration: '2m', target: 40},
  {duration: '5m', target: 100},
  {duration: '5m', target: 200},
  {duration: '2m', target: 150},
  {duration: '1m', target: 100},
  {duration: '1m', target: 120},
  {duration: '30s', target: 80},
  {duration: '30s', target: 40},
  {duration: '30s', target: 20},
],
```

Qua đánh giá kết quả K6, hiệu năng của Cobra nhỉnh hơn so với NetflixHollow.

- Cobra thực hiện được 2314 request/s so với 2265 request/s.

- p(95) của HTTP ở Cobra là 47.9 ms so với 50.45 ms của NetflixHollow.



Hình 6.6 Kết quả hiệu năng của K6

## 6.4 Đánh giá về tính linh hoạt

Kiến trúc của NetflixHollow yêu cầu Consumer làm việc trực tiếp với kho lưu trữ, điều này giảm đi tính linh hoạt trong quá trình phát triển phần mềm nếu muốn chuyển sang sử dụng một cách thức lưu trữ khác.

Mặt khác, Cobra đưa ra giải pháp Producer và Consumer giao tiếp qua interface, có thể dễ dàng chuyển đổi cũng như triển khai theo nhu cầu hạ tầng sử dụng.

## 6.5 Tổng kết

Đánh giá tổng kết:

- Cobra cho thấy tối ưu hơn về bộ nhớ cấp phát, bộ nhớ vật lý hoạt động so với NetflixHollow ở Consumer.
- Cobra Producer được cấp phát nhiều bộ nhớ hơn, tuy nhiên bộ nhớ vật lý được sử dụng ít hơn so với NetflixHollow.
- Tỷ lệ sử dụng CPU là tương đương nhau.
- Hiệu năng của Cobra nhỉnh hơn không đáng kể.
- Tuy nhiên tính linh hoạt của Cobra cao hơn, do cho phép các thành phần tách rời nhau.



# Chương 7. KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN

## 7.1 Kết luận

Thông qua đồ án, nhóm đề xuất một giải pháp giúp giải quyết các vấn đề liên quan đến việc chia sẻ, quản lý dữ liệu của bộ nhớ đệm, phù hợp với các bài toán có yêu cầu đặc thù, tính kỹ thuật như huấn luyện model Machine Learning, các website thương mại điện tử, hoặc các chương trình truyền hình.

## 7.2 Hướng phát triển tương lai

- Tối ưu hiệu năng, lưu trữ của phần mềm, cho phép xử lý lượng dữ liệu lớn hơn.
- Phát triển một công cụ có thể tích hợp để trực quan hoá những sự thay đổi, giúp việc nghiên cứu, giám sát, gỡ lỗi về dữ liệu dễ dàng hơn.
- Triển khai thêm tính năng sao lưu (backup) tại một thời điểm nhất định, thay vì các Consumer mới sẽ đi từ trạng thái đầu đến cuối, sẽ chỉ cần bắt đầu lấy dữ liệu từ điểm được sao lưu gần nhất.
- Hỗ trợ đánh chỉ mục thêm cho các trường trong lớp, giúp tăng sự linh hoạt khi truy vấn dữ liệu.

## Chương 8. TÀI LIỆU THAM KHẢO

- [1] "HotSpot Virtual Machine Garbage Collection Tuning Guide," [Online]. Available: <https://docs.oracle.com/en/java/javase/11/gctuning/garbage-first-g1-garbage-collector1.html#GUID-ED3AB6D3-FD9B-4447-9EDF-983ED2F7A573>. [Accessed 15 Jan 2025].
- [2] J. Jenkov, "The Producer Consumer Pattern," 19 Apr 2021. [Online]. Available: <https://jenkov.com/tutorials/java-concurrency/producer-consumer.html>. [Accessed 30 Jan 2025].
- [3] J. Bremner, "Bringing Metadata Faster to Edge Infrastructures with Hollow," Adobe, 28 Jan 2021. [Online]. Available: <https://blog.developer.adobe.com/bringing-metadata-faster-to-edge-infrastructures-with-hollow-2a9a3bd646ea>. [Accessed 10 Jan 2025].
- [4] "Write once, run anywhere?," 02 May 2002. [Online]. Available: <https://web.archive.org/web/20210813193857/https://www.computerweekly.com/feature/Write-once-run-anywhere>. [Accessed 20 Jan 2025].
- [5] "1.2 Design Goals of the Java TM Programming Language," 23 Jan 2013. [Online]. Available: <https://web.archive.org/web/20130123204103/http://www.oracle.com/technetwork/java/intro-141325.html>. [Accessed 10 Jan 2025].
- [6] L. Tim, Y. Frank, B. Gilad, B. Alex and S. Daniel, "Run-Time Data Areas," in *The Java® Virtual Machine Specification Java SE 11 Edition*, Oracle America, Inc, 2018.
- [7] "Difference between Direct, Non Direct and Mapped ByteBuffer in Java?," 23 May 2022. [Online]. Available: <https://javarevisited.blogspot.com/2015/08/difference-between-direct-non-direct-mapped-bytebuffer-nio-java.html>. [Accessed 13 Jan 2025].
- [8] H. Shiyu, G. Jianmei, L. Sanhong, L. Xiang, Q. Yumin and C. Kingsum, "SafeCheck: Safety Enhancement of Java Unsafe API," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, Montreal, 2019.
- [9] P. Lawrey, "Low GC in Java: Use primitives instead of wrappers," 5 July 2011. [Online]. Available: <https://blog.vanillajava.blog/2011/07/low-gc-in-java-use-primitives-instead.html>. [Accessed 10 Jan 2025].

- [10] "Java Magic. Part 4: sun.misc.Unsafe," 26 Feb 2013. [Online]. Available: <https://mishadoff.com/blog/java-magic-part-4-sun-dot-misc-dot-unsafe/>. [Accessed 13 Jan 2025].
- [11] "Guide to Java FileChannel," 8 Jan 2024. [Online]. Available: <https://www.baeldung.com/java-filechannel>. [Accessed 13 Jan 2025].
- [12] W. Teo, "Guide to FileOutputStream vs. FileChannel," 11 July 2024. [Online]. Available: <https://www.baeldung.com/java-fileoutputstream-filechannel-differences>. [Accessed 13 Jan 2025].
- [13] P. Sathish and N. Pramod, "Efficient data transfer through zero copy," 01 Sep 2008. [Online]. Available: <https://developer.ibm.com/articles/j-zero-copy/>. [Accessed 14 Jan 2025].
- [14] "Two's complement," 31 Jan 2025. [Online]. Available: [https://en.wikipedia.org/wiki/Two%27s\\_complement](https://en.wikipedia.org/wiki/Two%27s_complement). [Accessed 3 Feb 2025].
- [15] "Endianness," 4 Jan 2025. [Online]. Available: <https://en.wikipedia.org/wiki/Endianness>. [Accessed 13 Jan 2025].
- [16] DWARF Committee, "DWARF Debugging Information," in *DWARF Debugging Information Format, Version 4*, Free Standard Group, 2010, pp. 187-218.
- [17] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, San Jose, CA, USA, 2004.
- [18] M. Jürgen, L. Martin, G. Felix, Z. Alexander and P. Hasso, "Assessment of communication protocols in the EPC Network - replacing textual SOAP and XML with binary google protocol buffers encoding," in *2010 IEEE 17Th International Conference on Industrial Engineering and Engineering Management*, Xiamen, China, 2010.
- [19] A. Hass, A. Rossberg, D. L. Schuff and others, "Bringing the web up to speed with WebAssembly," in *PLDI 2017: ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, 2017.
- [20] "A better varint," 08 Mar 2021. [Online]. Available: <https://dcreager.net/2021/03/a-better-varint/>. [Accessed 15 Jan 2025].

- [21] J. Jaeyoung, J. J. Sung, J. Sunmin , H. Jun, S. Hoon and H. J. Tae, "A Specialized Architecture for Object Serialization with Applications to Big Data Analytics," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, Valencia, 2020.
- [22] "Data Serialization," [Online]. Available: <https://spark.apache.org/docs/latest/tuning.html>. [Accessed 17 Jan 2025].
- [23] K. Nico , "Flink Serialization Tuning Vol. 1: Choosing your Serializer — if you can," 15 April 2020. [Online]. Available: <https://flink.apache.org/2020/04/15/flink-serialization-tuning-vol.-1-choosing-your-serializer-if-you-can/>. [Accessed 15 Jan 2025].
- [24] "Research for Building High Performance Communication Service Based on Netty Protocol in Smart Health," in *ICSPS 2017: Proceedings of the 9th International Conference on Signal Processing Systems*, Auckland New Zealand, 2017.
- [25] "Scalable Event Multiplexing: epoll vs. kqueue," 21 Dec 2012. [Online]. Available: <https://long-zhou.github.io/2012/12/21/epoll-vs-kqueue.html>. [Accessed 20 Jan 2025].
- [26] M. Norman and W. A. Marvin, "Chapter 7. EventLoop and threading model," in *Netty in Action*, Manning, 2015, pp. 96-106.
- [27] K. E. Donald, "Chapter 6.4 Hashing," in *The Art of Computer Programming*, 2 ed., vol. Sorting and Searching, Addison-Wesley Professional, 1998, p. 513.
- [28] G. Cheng and Y. Yan, "Evaluation and Design of Non-cryptographic Hash Functions for Network Data Stream Algorithms," in *2017 3rd International Conference on Big Data Computing and Communications (BIGCOM)*, Chengdu, 2017.
- [29] "Hash table. Open addressing strategy," [Online]. Available: [https://www.algolist.net/Data\\_structures/Hash\\_table/Open\\_addressing](https://www.algolist.net/Data_structures/Hash_table/Open_addressing). [Accessed 10 Jan 2025].
- [30] A. A. Moe, T. T. Soe and M. M. Thein, "Review and Comparison for Collision resolution in a Hash a Table," *International Journal of Research Publications*, vol. 49, no. 1, p. 8, 2020.
- [31] K. Donald, "Chapter 2.5: Dynamic storage allocation," in *The Art of Computer Programming*, 3 ed., vol. Fundamental Algorithms, Addison-Wesley Professional, 1997, p. 435.

- [32] [Online]. Available: <https://github.com/memcached/memcached/blob/master/slabs.c>. [Accessed 31 Jan 2025].
- [33] B. Jeff, "The slab allocator: an object-caching kernel memory allocator," in *USTC 1994: Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1*, Boston Massachusetts, 1994.
- [34] H. Huang and L. Wang, "P&P: A Combined Push-Pull Model for Resource Monitoring in Cloud Computing Environment," in *2010 IEEE 3rd International Conference on Cloud Computing*, Miami, 2010.
- [35] "Apache Kafka Documentation, Push vs. pull," [Online]. Available: [https://kafka.apache.org/documentation/#design\\_pull](https://kafka.apache.org/documentation/#design_pull). [Accessed 30 Jan 2025].
- [36] V. Balasubramaniam, "Defining JPA Entities," 11 May 2024. [Online]. Available: <https://www.baeldung.com/jpa-entities>. [Accessed 30 Jan 2025].
- [37] "Guide to java.util.concurrent.Lock," 11 May 2024. [Online]. Available: <https://www.baeldung.com/java-concurrent-locks>. [Accessed 20 Jan 2025].