ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

Nguyễn Hải Đan

ĐÁNH GIÁ CÁC CƠ CHẾ GIAO TIẾP TRONG KIẾN TRÚC MICROSERVICE

KHÓA LUẬN TỐT NGHIỆP ĐẠI HỌC HỆ CHÍNH QUY
Ngành: Công nghệ thông tin

HÀ NỘI – 2025

ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

Nguyễn Hải Đan

ĐÁNH GIÁ CÁC CƠ CHẾ GIAO TIẾP TRONG KIẾN TRÚC MICROSERVICE

KHÓA LUẬN TỐT NGHIỆP ĐẠI HỌC HỆ CHÍNH QUY
Ngành: Công nghệ thông tin

Cán bộ hướng dẫn: Your Supervisor's Name

HÀ NỘI – 2025

VIETNAM NATIONAL UNIVERSITY, HANOI
UNIVERSITY OF ENGINEERING AND TECHNOLOGY

Nguyễn Hải Đan

STUDY OF COMMUNICATION MECHANISMS IN MICROSERVICE ARCHITECTURE

Major: Information Technology

Supervisor: Your Supervisor's Name in English

HÀ NỘI – 2025

TÓM TẮT

Tóm tắt: Kiến trúc microservice đã trở thành một xu hướng quan trọng trong phát triển phần mềm hiện đại, cho phép xây dựng các hệ thống phức tạp từ các dịch vụ nhỏ, độc lập. Một trong những thách thức chính trong kiến trúc này là việc quản lý giao tiếp giữa các microservice. Khóa luận này tập trung nghiên cứu các cơ chế giao tiếp trong kiến trúc microservice, bao gồm các mô hình đồng bộ và bất đồng bộ, các giao thức và công nghệ được sử dụng, cũng như các thách thức và giải pháp trong việc triển khai. Nghiên cứu cũng đánh giá hiệu quả của các phương pháp giao tiếp khác nhau thông qua các trường hợp sử dụng thực tế và đề xuất các hướng tiếp cận tối ưu cho các tình huống cụ thể.
Từ khóa: *Microservice, Kiến trúc phần mềm, Giao tiếp dịch vụ, API, Message Queue.*

ABSTRACT

Abstract: Microservice architecture has become a significant trend in modern software development, enabling the construction of complex systems from small, independent services. One of the main challenges in this architecture is managing communication between microservices. This thesis focuses on studying communication mechanisms in microservice architecture, including synchronous and asynchronous models, protocols and technologies used, as well as challenges and solutions in implementation. The study also evaluates the effectiveness of different communication methods through real-world use cases and proposes optimal approaches for specific scenarios.

Keywords: *Microservice, Software Architecture, Service Communication, API, Message Queue.*

Mục lục

Danh sách hình vẽ

Danh sách bảng

Danh sách từ viết tắt

API: Application Programming Interface – Giao diện lập trình ứng dụng
RPC: Remote Procedure Call – Gọi thủ tục từ xa
MQ: Message Queue – Hàng đợi tin nhắn
REST: Representational State Transfer – Chuyển giao trạng thái biểu diễn
SOA: Service-Oriented Architecture – Kiến trúc hướng dịch vụ

LỜI CAM ĐOAN

Tôi xin cam đoan đây là công trình nghiên cứu của riêng tôi. Các số liệu, kết quả nêu trong khóa luận là trung thực và chưa từng được ai công bố trong bất kỳ công trình nào khác. Các tham khảo, trích dẫn trong khóa luận đều được chỉ rõ nguồn gốc. Nếu sai tôi xin hoàn toàn chịu trách nhiệm.

Hà Nội, ngày ……tháng ……năm 2025

Người cam đoan

Nguyễn Hải Đan

Chương 1.
Mở đầu

1.1. Bối cảnh và sự cần thiết của đề tài

Trong bối cảnh phát triển phần mềm hiện đại, các hệ thống ngày càng trở nên phức tạp và đòi hỏi khả năng mở rộng cao để đáp ứng nhu cầu kinh doanh không ngừng thay đổi. Kiến trúc microservice đã nổi lên như một giải pháp hiệu quả, cho phép các nhóm phát triển độc lập tạo ra, triển khai và mở rộng các dịch vụ nhỏ, tập trung vào một chức năng cụ thể của hệ thống. Sự phân tách này tạo điều kiện cho việc phát triển nhanh chóng và linh hoạt, giúp các tổ chức thích ứng tốt hơn với các yêu cầu thay đổi của thị trường.

Theo một báo cáo của IDC, đến năm 2025, hơn 80% các tổ chức doanh nghiệp sẽ chuyển đổi sang kiến trúc phân tán như microservice để tăng tốc độ phát triển và triển khai ứng dụng. Xu hướng này phản ánh nhu cầu ngày càng tăng về khả năng mở rộng, tính linh hoạt và tốc độ phát triển trong môi trường kinh doanh cạnh tranh. Tuy nhiên, việc phân tách một hệ thống thành nhiều dịch vụ nhỏ cũng đặt ra thách thức lớn về cách thức các dịch vụ này giao tiếp với nhau.

Trong kiến trúc microservice, giao tiếp giữa các dịch vụ là một yếu tố then chốt quyết định đến hiệu suất tổng thể của hệ thống. Mô hình giao tiếp không phù hợp có thể tạo ra các điểm nghẽn, làm tăng độ trễ và giảm khả năng đáp ứng của hệ thống. Khi nhiều microservice phải tương tác với nhau để hoàn thành một nhiệm vụ, hiệu suất của toàn bộ chuỗi dịch vụ có thể bị ảnh hưởng bởi thời gian phản hồi của dịch vụ chậm nhất hoặc khả năng xử lý thấp nhất trong chuỗi.

Về độ tin cậy, trong một hệ thống phân tán, các lỗi giao tiếp có thể xảy ra ở nhiều điểm, ảnh hưởng đến tính nhất quán và độ sẵn sàng của dịch vụ. Mạng không ổn định, dịch vụ quá tải, hoặc các sự cố không lường trước đều có thể dẫn đến lỗi giao tiếp. Các cơ chế phát hiện lỗi, xử lý lỗi và khôi phục là rất quan trọng để duy trì tính khả dụng của hệ thống.

Các mẫu giao tiếp phải hỗ trợ việc mở rộng số lượng dịch vụ và lưu lượng giao dịch mà không làm giảm hiệu suất, đồng thời đảm bảo hệ thống hoạt động ổn định ngay cả khi một số dịch vụ gặp sự cố. Khả năng mở rộng là đặc biệt quan trọng khi doanh nghiệp phát triển và nhu cầu về hệ thống tăng lên. Các mẫu giao tiếp không chỉ phải xử lý được lưu lượng hiện tại mà còn phải có khả năng thích ứng với sự tăng trưởng trong tương lai.

Thực tế cho thấy, việc lựa chọn cơ chế giao tiếp không phù hợp có thể dẫn đến các vấn đề nghiêm trọng. Theo nghiên cứu của Gartner, hơn 70% các dự án microservice gặp khó khăn trong giai đoạn đầu triển khai do thiếu hiểu biết về các mô hình giao tiếp và cách thức áp dụng chúng hiệu quả. Nhiều tổ chức đã phải thiết kế lại kiến trúc của họ sau khi gặp phải các vấn đề về hiệu suất, độ tin cậy và khả năng quản lý.

Các tổ chức phải đối mặt với sự đánh đổi giữa tính nhất quán và hiệu suất, giữa độ tin cậy và độ phức tạp. Những quyết định về cơ chế giao tiếp có ảnh hưởng sâu sắc đến kiến trúc tổng thể và thành công của hệ thống. Ví dụ, giao tiếp đồng bộ có thể đơn giản và dễ triển khai, nhưng có thể ảnh hưởng đến hiệu suất và khả năng chịu lỗi. Ngược lại, giao tiếp bất đồng bộ có thể cải thiện khả năng mở rộng và chịu lỗi, nhưng làm tăng độ phức tạp trong việc quản lý trạng thái và đảm bảo tính nhất quán của dữ liệu.

Do đó, việc phân tích và đánh giá các mẫu giao tiếp (communication patterns) trong kiến trúc microservice là vô cùng cần thiết, giúp các nhà phát triển và kiến trúc sư hệ thống hiểu rõ các lựa chọn có sẵn và những đánh đổi liên quan. Thông qua việc phân tích có hệ thống, các nhà phát triển có thể áp dụng các mẫu phù hợp với bối cảnh cụ thể, tối ưu hóa hiệu suất và độ tin cậy của hệ thống, đồng thời đảm bảo khả năng mở rộng trong tương lai.

Bài đánh giá này nhằm cung cấp một cái nhìn toàn diện về các mẫu giao tiếp trong kiến trúc microservice, phân tích ưu nhược điểm của từng mẫu, và đưa ra các hướng dẫn thực tiễn cho việc lựa chọn và triển khai các mẫu giao tiếp phù hợp.

1.2. Mục tiêu nghiên cứu

Khóa luận này hướng đến việc phân tích và đánh giá các mẫu giao tiếp trong kiến trúc microservice nhằm cung cấp cơ sở lý thuyết và thực tiễn cho việc lựa chọn, thiết kế và triển khai các cơ chế giao tiếp hiệu quả. Việc hiểu rõ và áp dụng đúng các mẫu giao tiếp không chỉ giúp tối ưu hóa hiệu suất hệ thống mà còn đảm bảo tính linh hoạt và khả năng mở rộng - những yếu tố then chốt trong thành công của các hệ thống microservice hiện đại.

Cụ thể, khóa luận hướng đến việc phân loại và hệ thống hóa các mẫu giao tiếp trong microservice theo tiêu chí giao tiếp đồng bộ/bất đồng bộ và mô hình one-to-one/one-to-many. Việc phân loại này giúp tạo ra một khung tham chiếu rõ ràng, từ đó người đọc có thể dễ dàng định vị và hiểu các mẫu giao tiếp phù hợp với nhu cầu cụ thể của họ. Khung phân loại này cũng phản ánh các đặc tính cơ bản của giao tiếp giữa các dịch vụ, bao gồm thời gian (đồng bộ hay bất đồng bộ) và phạm vi (một đối một hay một đối nhiều).

Đồng thời, khóa luận sẽ phân tích chi tiết và so sánh các mẫu giao tiếp trên nhiều khía cạnh. Về cơ chế hoạt động, khóa luận sẽ mô tả chi tiết cách thức các mẫu giao tiếp hoạt động, bao gồm các thành phần, luồng dữ liệu và tương tác giữa các dịch vụ. Về hiệu suất và độ trễ, khóa luận sẽ đánh giá thời gian phản hồi, thông lượng và khả năng xử lý đồng thời của các mẫu giao tiếp. Về khả năng mở rộng, khóa luận sẽ phân tích khả năng của các mẫu giao tiếp trong việc hỗ trợ việc mở rộng số lượng dịch vụ và lưu lượng giao dịch. Về độ tin cậy và khả năng chịu lỗi, khóa luận sẽ đánh giá khả năng của các mẫu giao tiếp trong việc duy trì hoạt động ổn định khi gặp lỗi hoặc sự cố. Về độ phức tạp trong triển khai và bảo trì, khóa luận sẽ xem xét mức độ phức tạp và nguồn lực cần thiết để triển khai và duy trì các mẫu giao tiếp. Cuối cùng, về tính phù hợp với các tình huống cụ thể, khóa luận sẽ xác định các ngữ cảnh và yêu cầu mà mỗi mẫu giao tiếp phù hợp nhất.

Để đánh giá các mẫu giao tiếp một cách khách quan, khóa luận sẽ xây dựng và triển khai môi trường thử nghiệm mô phỏng các kịch bản thực tế. Các kịch bản này bao gồm kiểm tra và cập nhật tồn kho, xử lý thanh toán, phân phối thông báo, và ghi nhận hoạt động người dùng. Những kịch bản này được chọn vì chúng đại diện cho các tình huống phổ biến trong các ứng dụng thực tế và đòi hỏi các đặc tính giao tiếp khác nhau. Thông qua việc triển khai và đánh giá các mẫu giao tiếp trong các kịch bản này, khóa luận có thể cung cấp một cái nhìn thực tế về hiệu quả của từng mẫu giao tiếp.

Khóa luận sẽ đo lường và phân tích hiệu năng của các mẫu giao tiếp trong điều kiện khác nhau về tải và độ trễ mạng. Việc đo lường này bao gồm các chỉ số như thời gian phản hồi, thông lượng, tỷ lệ lỗi, tính nhất quán dữ liệu, và khả năng phục hồi sau lỗi. Thông qua việc phân tích các chỉ số này, khóa luận có thể xác định các điểm mạnh và điểm yếu của từng mẫu giao tiếp trong các điều kiện khác nhau.

Kết quả cuối cùng của khóa luận là tổng hợp các nguyên tắc và hướng dẫn thực tiễn cho việc lựa chọn mẫu giao tiếp phù hợp dựa trên yêu cầu cụ thể của ứng dụng. Các hướng dẫn này sẽ

giúp các nhà phát triển và kiến trúc sư hệ thống đưa ra quyết định sáng suốt về cách thức các dịch vụ giao tiếp với nhau, từ đó tối ưu hóa hiệu suất, độ tin cậy và khả năng mở rộng của hệ thống microservice.

1.3. Phạm vi nghiên cứu

Để đảm bảo tính khả thi và giá trị thực tiễn của khóa luận, phạm vi đánh giá được giới hạn như sau:

Về nội dung, khóa luận tập trung vào các mẫu giao tiếp phổ biến trong kiến trúc microservice. Các mẫu này được lựa chọn không chỉ vì tính phổ biến của chúng mà còn vì chúng đại diện cho các phương pháp tiếp cận khác nhau trong việc giải quyết vấn đề giao tiếp giữa các dịch vụ. Cụ thể, khóa luận sẽ tập trung vào giao tiếp đồng bộ thông qua REST API và gRPC, giao tiếp bất đồng bộ one-to-one thông qua Message Queue (RabbitMQ), và giao tiếp bất đồng bộ one-to-many thông qua Pub/Sub (Kafka).

Các khía cạnh được khóa luận phân tích bao gồm mô hình giao tiếp và luồng dữ liệu, mô tả cách thức các dịch vụ tương tác và trao đổi thông tin với nhau. Giao thức và định dạng dữ liệu cũng được đề cập, bao gồm việc sử dụng HTTP, AMQP, và các định dạng như JSON, Protocol Buffers, và các định dạng tùy chỉnh khác. Khóa luận cũng xem xét cơ chế xử lý lỗi và retry, phân tích cách thức các mẫu giao tiếp xử lý các tình huống lỗi và đảm bảo độ tin cậy của hệ thống. Đo lường hiệu năng và tối ưu hóa là một khía cạnh quan trọng khác, bao gồm việc đánh giá hiệu suất của các mẫu giao tiếp và cách thức tối ưu hóa hiệu suất. Cuối cùng, khóa luận xem xét tính mở rộng và khả năng chịu tải của các mẫu giao tiếp, đánh giá khả năng của chúng trong việc hỗ trợ hệ thống mở rộng khi nhu cầu tăng lên.

Khóa luận không đi sâu vào chi tiết kỹ thuật triển khai của từng công nghệ cụ thể, vì điều này nằm ngoài phạm vi của một khóa luận tổng quan về các mẫu giao tiếp. Các vấn đề bảo mật và quản lý danh tính được đề cập nhưng không phải là trọng tâm, vì chúng đòi hỏi một phân tích chuyên sâu riêng biệt. Tương tự, các vấn đề liên quan đến cơ sở dữ liệu và quản lý trạng thái cũng nằm ngoài phạm vi chính của khóa luận này.

Về công nghệ, khóa luận sử dụng các nền tảng phát triển bao gồm Node.js và TypeScript. Các công nghệ này được chọn vì tính phổ biến, khả năng áp dụng rộng rãi và sự hỗ trợ tốt cho phát triển microservice. Các công nghệ giao tiếp mà khóa luận sử dụng bao gồm RESTful API, Message Queuing (RabbitMQ), và Event Streaming (Kafka). Các công cụ đo lường hiệu năng bao gồm K6, Prometheus, và Grafana, được sử dụng để thu thập và phân tích dữ liệu về hiệu suất của các mẫu giao tiếp.

Về thực nghiệm, khóa luận sẽ xây dựng ứng dụng mẫu với 4 kịch bản thực tế để đánh giá hiệu quả của các mẫu giao tiếp khác nhau. Kịch bản đầu tiên là Order-Inventory, liên quan đến việc kiểm tra và cập nhật tồn kho. Đây là một tình huống phổ biến trong các hệ thống thương mại điện tử, đòi hỏi tính nhất quán dữ liệu cao. Kịch bản thứ hai là Order-Payment, liên quan đến việc xử lý thanh toán. Quá trình thanh toán thường đòi hỏi độ tin cậy cao và cần xử lý các tình huống lỗi một cách cẩn thận. Kịch bản thứ ba là Order-Notification, liên quan đến việc phân phối thông báo. Việc gửi thông báo thường yêu cầu một cơ chế giao tiếp một-đến-nhiều, phản ánh nhu cầu thông báo cho nhiều đối tượng khác nhau. Kịch bản cuối cùng là User Activity Logging, liên quan đến việc ghi nhận hoạt động người dùng. Đây là một tình huống cần khả năng xử lý lượng lớn các sự kiện, đòi hỏi một cơ chế giao tiếp có khả năng chịu tải cao.

Các tiêu chí đánh giá trong thực nghiệm bao gồm độ trễ trung bình và percentile thứ 95, phản ánh thời gian phản hồi trung bình và trong trường hợp xấu. Thông lượng là một chỉ số khác, đo lường số lượng giao dịch có thể xử lý trong một đơn vị thời gian. Tỷ lệ lỗi được theo dõi để đánh

giá độ tin cậy của các mẫu giao tiếp. Tính nhất quán dữ liệu, khả năng phục hồi sau lỗi, và khả năng mở rộng theo chiều ngang cũng là những tiêu chí quan trọng được đánh giá.

1.4. Phương pháp nghiên cứu

Khóa luận này sử dụng kết hợp nhiều phương pháp để đảm bảo tính toàn diện và độ tin cậy của kết quả đánh giá. Việc kết hợp các phương pháp này giúp cung cấp một cái nhìn đa chiều về các mẫu giao tiếp, từ lý thuyết đến thực tiễn, từ định tính đến định lượng.

Về phương pháp nghiên cứu lý thuyết, khóa luận thực hiện việc thu thập, phân tích và tổng hợp các tài liệu học thuật, báo cáo kỹ thuật, sách chuyên ngành và tài liệu từ các hội nghị về kiến trúc microservice và các mẫu giao tiếp. Quá trình này bao gồm việc xem xét các tài liệu từ các nguồn uy tín như IEEE, ACM, OReilly và các blog kỹ thuật của các công ty công nghệ hàng đầu như Netflix, Uber, và Airbnb. Thông qua việc phân tích tài liệu, khóa luận có thể hiểu rõ các nguyên tắc, khái niệm và thực tiễn tốt nhất liên quan đến các mẫu giao tiếp trong kiến trúc microservice.

Khóa luận cũng thực hiện phân tích so sánh có hệ thống các mẫu giao tiếp dựa trên các tiêu chí định lượng và định tính. Các tiêu chí này bao gồm hiệu suất, độ tin cậy, khả năng mở rộng, độ phức tạp, và tính phù hợp với các tình huống cụ thể. Thông qua việc so sánh các mẫu giao tiếp trên cùng một bộ tiêu chí, khóa luận có thể xác định các đánh đổi giữa các lựa chọn khác nhau và cung cấp một cơ sở cho việc lựa chọn mẫu giao tiếp phù hợp.

Ngoài ra, khóa luận phân tích các trường hợp thực tế về việc triển khai các mẫu giao tiếp trong các tổ chức lớn và các bài học kinh nghiệm được rút ra. Việc nghiên cứu các trường hợp thực tế giúp hiểu rõ hơn về cách thức các mẫu giao tiếp được áp dụng trong thực tế, các thách thức gặp phải và các giải pháp đã được áp dụng. Thông qua việc phân tích các bài học kinh nghiệm, khóa luận có thể rút ra các nguyên tắc và hướng dẫn thực tiễn cho việc triển khai các mẫu giao tiếp.

Về phương pháp nghiên cứu thực nghiệm, khóa luận phát triển một ứng dụng microservice mẫu theo kiến trúc tham chiếu, đảm bảo tính đại diện và khả năng so sánh giữa các mẫu giao tiếp. Ứng dụng này được thiết kế để mô phỏng các tình huống thực tế trong môi trường doanh nghiệp, bao gồm quản lý đơn hàng, thanh toán, gửi thông báo và ghi nhận hoạt động người dùng. Việc phát triển một ứng dụng mẫu cho phép đánh giá các mẫu giao tiếp trong một bối cảnh thực tế, mang lại cái nhìn thực tiễn về hiệu quả của chúng.

Khóa luận thiết kế các kịch bản thử nghiệm mô phỏng các tình huống thực tế và các điều kiện tải khác nhau. Các kịch bản này được thiết kế để đánh giá hiệu suất, độ tin cậy và khả năng mở rộng của các mẫu giao tiếp trong các điều kiện khác nhau. Ví dụ, các kịch bản có thể bao gồm tải thấp, tải cao, tải đột biến, và các tình huống lỗi khác nhau. Việc thử nghiệm trong các điều kiện khác nhau giúp đánh giá toàn diện về hiệu quả của các mẫu giao tiếp.

Khóa luận thu thập dữ liệu về hiệu suất, độ tin cậy và khả năng mở rộng của các mẫu giao tiếp trong môi trường kiểm soát. Dữ liệu này bao gồm thời gian phản hồi, thông lượng, tỷ lệ lỗi, tính nhất quán dữ liệu, và khả năng phục hồi sau lỗi. Việc thu thập dữ liệu trong một môi trường kiểm soát đảm bảo tính nhất quán và so sánh công bằng giữa các mẫu giao tiếp.

Cuối cùng, khóa luận áp dụng các phương pháp thống kê để phân tích dữ liệu thu thập, đánh giá ý nghĩa thống kê của các kết quả và rút ra các kết luận. Việc phân tích thống kê giúp xác định xem liệu có sự khác biệt đáng kể giữa các mẫu giao tiếp hay không, và nếu có, mức độ khác biệt là bao nhiêu. Thông qua việc phân tích thống kê, khóa luận có thể đưa ra các kết luận dựa trên dữ liệu về hiệu quả của các mẫu giao tiếp.

Quy trình đánh giá được chia thành 5 giai đoạn. Giai đoạn 1 tập trung vào nghiên cứu lý thuyết và tổng hợp tài liệu, bao gồm tổng hợp và phân loại các mẫu giao tiếp, xác định các tiêu chí đánh giá, và xây dựng khung phân tích. Giai đoạn 2 là thiết kế và phát triển, bao gồm thiết kế kiến trúc tham chiếu, phát triển ứng dụng microservice mẫu, và triển khai các mẫu giao tiếp. Giai đoạn này đòi hỏi sự hiểu biết sâu sắc về các công nghệ và mẫu giao tiếp để đảm bảo rằng chúng được triển khai một cách chính xác và hiệu quả.

Giai đoạn 3 là thực hiện thử nghiệm, bao gồm thiết lập môi trường thử nghiệm, thực hiện các kịch bản thử nghiệm, và thu thập dữ liệu. Môi trường thử nghiệm được thiết lập để mô phỏng các điều kiện thực tế mà các mẫu giao tiếp sẽ hoạt động. Các kịch bản thử nghiệm được thực hiện để đánh giá hiệu suất, độ tin cậy và khả năng mở rộng của các mẫu giao tiếp trong các điều kiện khác nhau. Dữ liệu được thu thập một cách có hệ thống để đảm bảo tính chính xác và đầy đủ.

Giai đoạn 4 là phân tích và đánh giá, bao gồm phân tích dữ liệu thu thập, đánh giá hiệu quả của các mẫu giao tiếp, và xác định các yếu tố ảnh hưởng. Dữ liệu thu thập được phân tích để xác định các mẫu và xu hướng, và để đánh giá hiệu quả của các mẫu giao tiếp dựa trên các tiêu chí đã xác định. Các yếu tố ảnh hưởng đến hiệu quả của các mẫu giao tiếp cũng được xác định, giúp hiểu rõ hơn về cách thức các mẫu giao tiếp hoạt động trong các điều kiện khác nhau.

Giai đoạn 5 là tổng hợp và kết luận, bao gồm tổng hợp kết quả nghiên cứu, xây dựng hướng dẫn thực tiễn, và đề xuất hướng nghiên cứu tiếp theo. Kết quả nghiên cứu được tổng hợp để cung cấp một cái nhìn tổng thể về hiệu quả của các mẫu giao tiếp. Các hướng dẫn thực tiễn được xây dựng dựa trên kết quả nghiên cứu, cung cấp một tài liệu tham khảo cho các nhà phát triển và kiến trúc sư hệ thống trong việc lựa chọn và triển khai các mẫu giao tiếp. Các hướng nghiên cứu tiếp theo được đề xuất, chỉ ra các lĩnh vực cần nghiên cứu thêm hoặc các vấn đề chưa được giải quyết trong phạm vi của khóa luận hiện tại.

Thông qua việc kết hợp các phương pháp nghiên cứu lý thuyết và thực nghiệm, khóa luận này đảm bảo cung cấp một cái nhìn toàn diện và chính xác về các mẫu giao tiếp trong kiến trúc microservice. Các kết quả thu được từ khóa luận sẽ giúp các nhà phát triển và kiến trúc sư hệ thống lựa chọn và triển khai các mẫu giao tiếp phù hợp với nhu cầu cụ thể của ứng dụng của họ.

1.5. Ý nghĩa khoa học và thực tiễn

Khóa luận này có ý nghĩa quan trọng cả về mặt khoa học và thực tiễn, đóng góp vào sự phát triển của lĩnh vực kiến trúc phần mềm và hệ thống phân tán.

Về ý nghĩa khoa học, khóa luận đóng góp vào việc phân loại và hệ thống hóa các mẫu giao tiếp trong kiến trúc microservice, cung cấp một khung phân tích toàn diện cho việc đánh giá và lựa chọn các mẫu giao tiếp phù hợp. Thông qua việc phân tích và tổng hợp các tài liệu học thuật, khóa luận xác định các nguyên tắc cơ bản và các yếu tố ảnh hưởng đến hiệu quả của các mẫu giao tiếp. Điều này giúp xây dựng một nền tảng lý thuyết vững chắc cho việc nghiên cứu và phát triển các mẫu giao tiếp mới trong tương lai.

Khóa luận cũng đóng góp vào việc phát triển các phương pháp đánh giá hiệu quả của các mẫu giao tiếp, bao gồm các tiêu chí định lượng và định tính. Thông qua việc áp dụng các phương pháp thống kê và phân tích dữ liệu, khóa luận cung cấp một cách tiếp cận khoa học để đánh giá và so sánh các mẫu giao tiếp. Điều này giúp các nhà nghiên cứu và phát triển có thể đưa ra các quyết định dựa trên dữ liệu và bằng chứng thực nghiệm.

Ngoài ra, khóa luận cũng đóng góp vào việc xác định các hướng nghiên cứu tiếp theo trong lĩnh vực kiến trúc microservice và các mẫu giao tiếp. Thông qua việc phân tích các thách thức

và giới hạn hiện tại, khóa luận đề xuất các hướng nghiên cứu mới để giải quyết các vấn đề còn tồn tại. Điều này giúp thúc đẩy sự phát triển của lĩnh vực và mở ra các cơ hội nghiên cứu mới.

Về ý nghĩa thực tiễn, khóa luận cung cấp các hướng dẫn và khuyến nghị cụ thể cho việc lựa chọn và triển khai các mẫu giao tiếp trong các dự án thực tế. Thông qua việc phân tích các trường hợp thực tế và các bài học kinh nghiệm, khóa luận rút ra các nguyên tắc và thực tiễn tốt nhất cho việc triển khai các mẫu giao tiếp. Điều này giúp các nhà phát triển và kiến trúc sư hệ thống có thể áp dụng các mẫu giao tiếp một cách hiệu quả và tránh các lỗi phổ biến.

Khóa luận cũng cung cấp một ứng dụng microservice mẫu và các kịch bản thử nghiệm để đánh giá hiệu quả của các mẫu giao tiếp. Điều này giúp các nhà phát triển có thể thử nghiệm và đánh giá các mẫu giao tiếp trong một môi trường kiểm soát trước khi triển khai chúng trong các dự án thực tế. Việc có một ứng dụng mẫu và các kịch bản thử nghiệm cũng giúp giảm thiểu rủi ro và tăng cường sự tự tin trong việc triển khai các mẫu giao tiếp mới.

Ngoài ra, khóa luận cũng đóng góp vào việc nâng cao nhận thức và hiểu biết về các mẫu giao tiếp trong kiến trúc microservice. Thông qua việc trình bày rõ ràng và chi tiết về các mẫu giao tiếp, khóa luận giúp các nhà phát triển và kiến trúc sư hệ thống có thể hiểu rõ hơn về cách thức hoạt động và các đánh đổi của các mẫu giao tiếp. Điều này giúp họ có thể đưa ra các quyết định sáng suốt hơn trong việc thiết kế và triển khai các hệ thống microservice.

Cuối cùng, khóa luận cũng đóng góp vào việc thúc đẩy sự phát triển của cộng đồng phần mềm nguồn mở và các công cụ hỗ trợ cho việc triển khai các mẫu giao tiếp. Thông qua việc chia sẻ kiến thức và kinh nghiệm, khóa luận giúp xây dựng một cộng đồng mạnh mẽ hơn và thúc đẩy sự đổi mới trong lĩnh vực kiến trúc microservice. Điều này giúp tạo ra một hệ sinh thái phong phú và đa dạng cho việc phát triển và triển khai các hệ thống microservice.

Tóm lại, khóa luận này có ý nghĩa quan trọng cả về mặt khoa học và thực tiễn, đóng góp vào sự phát triển của lĩnh vực kiến trúc microservice và các mẫu giao tiếp. Thông qua việc kết hợp nghiên cứu lý thuyết và thực nghiệm, khóa luận cung cấp một cái nhìn toàn diện và chính xác về các mẫu giao tiếp, giúp các nhà phát triển và kiến trúc sư hệ thống có thể lựa chọn và triển khai các mẫu giao tiếp phù hợp với nhu cầu cụ thể của ứng dụng của họ.

1.6. Cấu trúc khóa luận

Khóa luận được tổ chức thành 5 chương như sau:

Chương 1, Mở đầu, giới thiệu bối cảnh và sự cần thiết của đề tài, xác định mục tiêu, phạm vi và phương pháp nghiên cứu, và trình bày ý nghĩa khoa học và thực tiễn của nghiên cứu.

Chương 2, Cơ sở lý thuyết, cung cấp tổng quan về kiến trúc microservice, khái niệm và phân loại các mẫu giao tiếp, và giới thiệu các công nghệ và giao thức giao tiếp phổ biến.

Chương 3, Phân tích các mẫu giao tiếp, phân loại các mẫu giao tiếp theo tiêu chí đồng bộ/bất đồng bộ và one-to-one/one-to-many, phân tích chi tiết các mẫu giao tiếp đồng bộ (one-to-one), phân tích chi tiết các mẫu giao tiếp bất đồng bộ (one-to-one), phân tích chi tiết các mẫu giao tiếp bất đồng bộ (one-to-many), và so sánh và đánh giá các mẫu giao tiếp.

Chương 4, Triển khai thử nghiệm, mô tả bài toán và yêu cầu, thiết kế và cài đặt ứng dụng mẫu, cài đặt và triển khai các mẫu giao tiếp, và kết quả triển khai và đánh giá hiệu năng.

Chương 5, Đánh giá và thảo luận, phân tích kết quả thực nghiệm, thảo luận về các phát hiện chính, đề xuất các nguyên tắc lựa chọn mẫu giao tiếp, và kết luận và hướng phát triển.

Chương 2.
Cơ sở lý thuyết

2.1. Tổng quan về Microservice Architecture

2.1.1. Định nghĩa và đặc điểm

Kiến trúc Microservice là một phương pháp phát triển phần mềm trong đó một ứng dụng được cấu thành từ nhiều dịch vụ nhỏ, độc lập và có khả năng triển khai riêng biệt. Mỗi dịch vụ này được thiết kế để thực hiện một chức năng cụ thể trong phạm vi nghiệp vụ được định nghĩa rõ ràng, và giao tiếp với các dịch vụ khác thông qua các cơ chế giao tiếp nhẹ, thường là API.

Theo Newman (2021), các đặc điểm chính của kiến trúc microservice bao gồm tính tự trị cao, trong đó mỗi dịch vụ có thể được phát triển, triển khai và mở rộng độc lập với các dịch vụ khác. Các dịch vụ được tổ chức xoay quanh các khả năng nghiệp vụ thay vì các lớp công nghệ, thể hiện sự phân tách theo chức năng nghiệp vụ. Quản lý dữ liệu trong microservice được thực hiện phi tập trung, với mỗi dịch vụ quản lý dữ liệu riêng và chỉ có thể truy cập dữ liệu thông qua API của dịch vụ sở hữu dữ liệu đó.

Thiết kế hướng lỗi là một đặc điểm quan trọng khác của microservice, trong đó các dịch vụ được thiết kế để xử lý lỗi và khả năng các dịch vụ khác không khả dụng. Cuối cùng, microservice cho phép tiến hóa độc lập, nghĩa là các dịch vụ có thể thay đổi và phát triển theo thời gian mà không ảnh hưởng đến toàn bộ hệ thống.

2.1.2. So sánh với kiến trúc nguyên khối (Monolithic)

Để hiểu rõ hơn về kiến trúc microservice, việc so sánh với kiến trúc nguyên khối là rất hữu ích. Trong kiến trúc nguyên khối, toàn bộ ứng dụng được xây dựng như một đơn vị duy nhất. Tất cả các chức năng nằm trong một codebase và được triển khai cùng nhau.

Về triển khai, kiến trúc nguyên khối đòi hỏi toàn bộ ứng dụng được triển khai cùng một lúc, trong khi kiến trúc microservice cho phép các dịch vụ được triển khai độc lập. Điều này có ý nghĩa quan trọng trong việc giảm thiểu rủi ro và tăng tốc độ phát hành.

Khả năng mở rộng cũng khác biệt đáng kể giữa hai kiến trúc. Trong kiến trúc nguyên khối, toàn bộ ứng dụng phải được mở rộng, ngay cả khi chỉ một phần cần thêm tài nguyên. Ngược lại, kiến trúc microservice cho phép mở rộng từng dịch vụ riêng biệt, tối ưu hóa việc sử dụng tài nguyên.

Về phát triển, kiến trúc nguyên khối thường có một nhóm phát triển làm việc trên một codebase, dẫn đến các xung đột trong quá trình phát triển và triển khai. Trong khi đó, kiến trúc microservice cho phép nhiều nhóm làm việc độc lập trên các dịch vụ khác nhau, tăng tốc độ phát triển và giảm thiểu xung đột.

Công nghệ là một khía cạnh khác có sự khác biệt. Kiến trúc nguyên khối thường bị giới hạn trong một stack công nghệ, trong khi mỗi microservice có thể sử dụng công nghệ phù hợp nhất với yêu cầu của nó. Điều này tạo ra sự linh hoạt và khả năng thích ứng với các công nghệ mới.

Khả năng chịu lỗi cũng là một điểm khác biệt quan trọng. Trong kiến trúc nguyên khối, lỗi ở một phần có thể ảnh hưởng đến toàn bộ ứng dụng, trong khi trong kiến trúc microservice, lỗi được cô lập trong một dịch vụ, giảm thiểu tác động đến toàn bộ hệ thống.

Cuối cùng, về độ phức tạp, kiến trúc nguyên khối đơn giản hơn trong các ứng dụng nhỏ, nhưng phức tạp hơn khi ứng dụng phát triển. Ngược lại, kiến trúc microservice phức tạp hơn ngay từ đầu do tính phân tán, nhưng độ phức tạp này được quản lý tốt hơn khi hệ thống phát triển.

2.1.3. Lợi ích và thách thức của kiến trúc microservice

Kiến trúc microservice mang lại nhiều lợi ích đáng kể cho việc phát triển và vận hành phần mềm. Một trong những lợi ích chính là khả năng mở rộng có mục tiêu. Các dịch vụ có thể được mở rộng độc lập dựa trên nhu cầu, tối ưu hóa việc sử dụng tài nguyên. Điều này đặc biệt quan trọng trong môi trường cloud, nơi chi phí tỷ lệ thuận với tài nguyên được sử dụng.

Phát triển nhanh hơn là một lợi ích khác của kiến trúc microservice. Các nhóm nhỏ có thể làm việc trên các dịch vụ độc lập, cho phép phát triển song song và chu kỳ phát hành nhanh hơn. Mỗi nhóm có thể tập trung vào một dịch vụ cụ thể, hiểu rõ nó và phát triển nó một cách hiệu quả.

Tính linh hoạt công nghệ cũng là một lợi thế đáng kể. Mỗi dịch vụ có thể sử dụng công nghệ phù hợp nhất với yêu cầu của nó. Ví dụ, một dịch vụ xử lý giao dịch có thể sử dụng một ngôn ngữ chú trọng vào tính nhất quán, trong khi một dịch vụ phân tích dữ liệu có thể sử dụng một ngôn ngữ tối ưu cho xử lý dữ liệu lớn.

Khả năng chịu lỗi tốt hơn là một lợi ích khác của kiến trúc microservice. Lỗi trong một dịch vụ không nhất thiết phải làm cho toàn bộ hệ thống không khả dụng. Ví dụ, nếu dịch vụ gợi ý sản phẩm không hoạt động, người dùng vẫn có thể duyệt và mua sản phẩm.

Khả năng bảo trì và hiểu biết tốt hơn cũng là một lợi thế của kiến trúc microservice. Các dịch vụ nhỏ hơn dễ hiểu và bảo trì hơn các ứng dụng lớn. Mã nguồn của mỗi dịch vụ nhỏ hơn và tập trung vào một chức năng cụ thể, giúp nhà phát triển dễ dàng hiểu và thay đổi nó.

Tuy nhiên, kiến trúc microservice cũng đặt ra một số thách thức đáng kể. Độ phức tạp phân tán là một thách thức lớn. Hệ thống phân tán vốn phức tạp hơn, đòi hỏi kiến thức và công cụ chuyên biệt. Các vấn đề như latency mạng, xử lý lỗi và đồng bộ hóa dữ liệu trở nên phức tạp hơn trong một hệ thống phân tán.

Giao tiếp giữa các dịch vụ là một thách thức khác. Thiết kế và quản lý giao tiếp giữa các dịch vụ đòi hỏi cân nhắc kỹ lưỡng về hiệu suất, độ tin cậy và khả năng mở rộng. Việc lựa chọn giao thức giao tiếp phù hợp và xử lý các trường hợp lỗi trong giao tiếp là các vấn đề phức tạp.

Quản lý dữ liệu cũng là một thách thức đáng kể trong kiến trúc microservice. Duy trì tính nhất quán dữ liệu giữa các dịch vụ có thể phức tạp, đặc biệt là khi mỗi dịch vụ có cơ sở dữ liệu riêng. Các mẫu như Saga và Event Sourcing được sử dụng để giải quyết vấn đề này, nhưng chúng cũng đưa ra sự phức tạp riêng.

Vận hành và giám sát là một thách thức khác của kiến trúc microservice. Triển khai và giám sát nhiều dịch vụ đòi hỏi công cụ và quy trình tinh vi hơn. Các công cụ như Kubernetes và Prometheus đã được phát triển để giải quyết vấn đề này, nhưng chúng cũng đòi hỏi kiến thức và nỗ lực đáng kể để sử dụng hiệu quả.

Cuối cùng, kiểm thử cũng trở nên phức tạp hơn trong kiến trúc microservice. Kiểm thử tích hợp đòi hỏi sự phối hợp giữa nhiều dịch vụ, có thể chạy trên các máy khác nhau và sử dụng các công nghệ khác nhau. Các kỹ thuật như kiểm thử hợp đồng và môi trường kiểm thử tích hợp được sử dụng để giải quyết vấn đề này.

2.1.4. Các nguyên tắc thiết kế

Để thiết kế một kiến trúc microservice hiệu quả, một số nguyên tắc thiết kế chính cần được tuân thủ. Nguyên tắc đầu tiên là Single Responsibility Principle (Nguyên tắc Trách nhiệm Đơn

lẻ), theo đó mỗi dịch vụ nên chịu trách nhiệm cho một chức năng nghiệp vụ duy nhất. Điều này giúp giữ các dịch vụ đơn giản và tập trung, dễ hiểu và bảo trì.

Domain-Driven Design (DDD) là một phương pháp thiết kế hữu ích cho kiến trúc microservice. DDD sử dụng các khái niệm như Bounded Context để định nghĩa ranh giới giữa các dịch vụ. Bounded Context giúp xác định phạm vi trách nhiệm của mỗi dịch vụ và cách chúng tương tác với nhau.

API First là một nguyên tắc khác, nhấn mạnh việc thiết kế API trước, xem nó như một hợp đồng giữa các dịch vụ. Điều này giúp đảm bảo rằng các dịch vụ có thể giao tiếp hiệu quả và rằng các thay đổi không phá vỡ tương thích ngược.

Tự động hóa là một phần quan trọng của kiến trúc microservice thành công. Tự động hóa quá trình xây dựng, kiểm thử và triển khai giúp quản lý sự phức tạp của việc phát triển và vận hành nhiều dịch vụ. Các công cụ CI/CD (Continuous Integration/Continuous Deployment) là rất quan trọng trong môi trường microservice.

Monitoring và Observability là các nguyên tắc quan trọng khác. Thiết kế hệ thống để dễ dàng giám sát và hiểu được hoạt động nội bộ giúp phát hiện và giải quyết vấn đề một cách nhanh chóng. Các công cụ như logging tập trung, theo dõi phân tán và thu thập số liệu là rất quan trọng.

Cuối cùng, Fault Tolerance (Khả năng chịu lỗi) là một nguyên tắc thiết kế quan trọng cho kiến trúc microservice. Các dịch vụ nên được thiết kế để xử lý lỗi một cách thanh nhã, sử dụng các kỹ thuật như Circuit Breaker. Circuit Breaker ngăn lỗi lan truyền bằng cách ngừng gửi yêu cầu đến các dịch vụ không phản hồi.

2.2. Communication trong Microservices

2.2.1. Vai trò của giao tiếp trong kiến trúc microservice

Giao tiếp đóng vai trò then chốt trong kiến trúc microservice. Khác với các ứng dụng nguyên khối, nơi các thành phần tương tác thông qua lời gọi hàm nội bộ, các microservice giao tiếp qua mạng, thường là thông qua HTTP, gRPC hoặc một middleware messaging.

Wolff (2018) nhấn mạnh rằng: ”Giao tiếp không chỉ là cách các dịch vụ trao đổi dữ liệu, mà còn định hình toàn bộ kiến trúc và ảnh hưởng đến tính khả dụng, hiệu suất và khả năng mở rộng của toàn bộ hệ thống.” Nhận xét này nhấn mạnh tầm quan trọng của việc lựa chọn mẫu giao tiếp phù hợp trong kiến trúc microservice.

Giao tiếp trong microservices tạo điều kiện cho sự hợp tác giữa các dịch vụ. Các dịch vụ cần phối hợp để hoàn thành các tác vụ nghiệp vụ phức tạp. Ví dụ, một quy trình đặt hàng có thể liên quan đến các dịch vụ quản lý đơn hàng, thanh toán, kho hàng và vận chuyển. Các dịch vụ này cần giao tiếp để đảm bảo đơn hàng được xử lý chính xác.

Giao tiếp cũng đóng vai trò quan trọng trong việc đảm bảo tính nhất quán dữ liệu. Trong một hệ thống dữ liệu phân tán, mỗi dịch vụ có thể quản lý một phần dữ liệu của hệ thống. Để duy trì tính nhất quán, các dịch vụ cần giao tiếp với nhau khi dữ liệu thay đổi. Ví dụ, khi một đơn hàng được tạo, dịch vụ đơn hàng cần thông báo cho dịch vụ kho hàng để đặt trước sản phẩm.

Giao tiếp hỗ trợ khả năng chịu lỗi của hệ thống microservice. Các cơ chế giao tiếp phù hợp có thể giúp hệ thống phục hồi từ lỗi và tiếp tục hoạt động. Ví dụ, mẫu Circuit Breaker có thể ngăn lỗi lan truyền bằng cách ngừng gửi yêu cầu đến các dịch vụ không phản hồi.

Cuối cùng, giao tiếp cho phép tính mở rộng của hệ thống microservice. Thiết kế giao tiếp tốt là chìa khóa để mở rộng hệ thống, cho phép thêm các dịch vụ mới hoặc phiên bản mới của các dịch vụ hiện có mà không ảnh hưởng đến các dịch vụ khác.

2.2.2. Các thuộc tính quan trọng của giao tiếp microservice

Khi thiết kế cơ chế giao tiếp cho microservice, một số thuộc tính quan trọng cần được xem xét. Độ tin cậy là một thuộc tính quan trọng, đề cập đến khả năng đảm bảo rằng thông điệp được gửi và nhận thành công. Trong một hệ thống phân tán, mạng có thể không đáng tin cậy và thông điệp có thể bị mất. Các cơ chế như xác nhận, thử lại và hàng đợi bền vững có thể được sử dụng để tăng độ tin cậy.

Độ trễ là một thuộc tính quan trọng khác, đề cập đến thời gian cần thiết để thông điệp đi từ nguồn đến đích. Độ trễ cao có thể ảnh hưởng đến hiệu suất và trải nghiệm người dùng. Các yếu tố ảnh hưởng đến độ trễ bao gồm khoảng cách vật lý giữa các dịch vụ, phương pháp tuần tự hóa và tải mạng.

Khả năng mở rộng là một thuộc tính quan trọng khác, đề cập đến khả năng xử lý khối lượng thông điệp tăng. Khi hệ thống phát triển, khối lượng giao tiếp giữa các dịch vụ cũng tăng. Cơ chế giao tiếp nên hỗ trợ khả năng mở rộng theo chiều ngang bằng cách thêm nhiều phiên bản của cùng một dịch vụ.

Cách ly lỗi là một thuộc tính quan trọng, đề cập đến khả năng ngăn lỗi lan truyền giữa các dịch vụ. Trong một hệ thống phân tán, lỗi là điều không thể tránh khỏi. Các mẫu như Circuit Breaker và Bulkhead có thể được sử dụng để ngăn lỗi từ một dịch vụ ảnh hưởng đến các dịch vụ khác.

Tính nhất quán là một thuộc tính quan trọng khác, đề cập đến mức độ và cách thức đảm bảo tính nhất quán dữ liệu. Trong một hệ thống phân tán, có một sự đánh đổi giữa tính nhất quán, khả năng sẵn sàng và khả năng chịu đựng phân vùng (định lý CAP). Các mẫu giao tiếp khác nhau cung cấp các mức độ nhất quán khác nhau, từ nhất quán mạnh đến nhất quán tối thiểu.

Định dạng dữ liệu là một thuộc tính quan trọng, đề cập đến cách dữ liệu được cấu trúc và tuần tự hóa. Các định dạng phổ biến bao gồm JSON, XML và Protocol Buffers. Mỗi định dạng có ưu và nhược điểm riêng về hiệu suất, khả năng đọc và khả năng tương tác.

Khả năng tương tác là một thuộc tính quan trọng, đề cập đến khả năng các dịch vụ sử dụng công nghệ khác nhau giao tiếp với nhau. Trong một môi trường microservice, các dịch vụ có thể được viết bằng các ngôn ngữ khác nhau và chạy trên các nền tảng khác nhau. Giao thức giao tiếp nên hỗ trợ khả năng tương tác giữa các dịch vụ không đồng nhất này.

Cuối cùng, bảo mật là một thuộc tính quan trọng, đề cập đến việc bảo vệ thông điệp khỏi truy cập trái phép. Các dịch vụ thường giao tiếp qua mạng, có thể không đáng tin cậy và không an toàn. Các cơ chế như mã hóa, xác thực và ủy quyền có thể được sử dụng để bảo vệ giao tiếp.

2.2.3. Các mô hình giao tiếp cơ bản

Có hai mô hình giao tiếp cơ bản trong microservices: đồng bộ và bất đồng bộ. Mỗi mô hình có ưu và nhược điểm riêng và phù hợp với các tình huống khác nhau.

Trong giao tiếp đồng bộ, người gửi đợi phản hồi từ người nhận trước khi tiếp tục xử lý. Ví dụ, khi một dịch vụ gửi yêu cầu HTTP đến một dịch vụ khác, nó đợi phản hồi HTTP trước khi tiếp tục. Giao tiếp đồng bộ đơn giản và dễ hiểu, làm cho nó trở thành một lựa chọn phổ biến cho nhiều tình huống. Nó cũng cung cấp phản hồi tức thì, cho phép người gửi biết ngay lập tức liệu yêu cầu của nó có thành công hay không.

Tuy nhiên, giao tiếp đồng bộ cũng có một số nhược điểm. Nó có thể dẫn đến hiệu suất kém hơn do người gửi bị chặn trong khi đợi phản hồi. Nó cũng có thể dẫn đến hiệu ứng xung hồi (ripple effect) khi một dịch vụ chậm hoặc không khả dụng ảnh hưởng đến tất cả các dịch vụ gọi nó. Hơn nữa, giao tiếp đồng bộ có thể gây ra các vấn đề về khả năng mở rộng khi số lượng yêu cầu tăng lên.

Trong giao tiếp bất đồng bộ, người gửi không đợi phản hồi từ người nhận và có thể tiếp tục xử lý. Ví dụ, khi một dịch vụ gửi một thông điệp đến một hàng đợi, nó có thể tiếp tục xử lý mà không cần đợi thông điệp được tiêu thụ. Giao tiếp bất đồng bộ cung cấp coupling lỏng lẻo hơn giữa các dịch vụ, vì người gửi không cần biết ai sẽ xử lý thông điệp của nó. Nó cũng cung cấp khả năng đệm, cho phép hệ thống xử lý các đợt tải cao bằng cách xếp hàng các thông điệp cho đến khi chúng có thể được xử lý.

Tuy nhiên, giao tiếp bất đồng bộ cũng có một số nhược điểm. Nó có thể phức tạp hơn để triển khai và gỡ lỗi do tính chất phi đồng bộ của nó. Nó cũng có thể dẫn đến độ trễ cao hơn, vì thông điệp có thể đợi trong hàng đợi trước khi được xử lý. Hơn nữa, giao tiếp bất đồng bộ có thể gây ra các vấn đề về tính nhất quán dữ liệu, vì các thay đổi không được phản ánh ngay lập tức trong tất cả các dịch vụ.

2.2.4. Kiểu tương tác

Ngoài các mô hình giao tiếp, các microservice cũng giao tiếp theo các kiểu tương tác khác nhau. Kiểu tương tác one-to-one (một-đối-một) liên quan đến một dịch vụ giao tiếp với một dịch vụ khác. Đây là kiểu tương tác đơn giản nhất và phổ biến nhất. Ví dụ, dịch vụ đơn hàng có thể gọi dịch vụ thanh toán để xử lý thanh toán cho một đơn hàng.

Kiểu tương tác one-to-many (một-đối-nhiều) liên quan đến một dịch vụ giao tiếp với nhiều dịch vụ khác. Kiểu này thường được sử dụng cho các tình huống broadcasting hoặc fanout. Ví dụ, dịch vụ đơn hàng có thể thông báo cho nhiều dịch vụ khác khi một đơn hàng được tạo, như dịch vụ kho hàng, dịch vụ vận chuyển và dịch vụ thông báo.

Kiểu tương tác many-to-one (nhiều-đối-một) liên quan đến nhiều dịch vụ giao tiếp với một dịch vụ. Kiểu này thường được sử dụng cho các tình huống tổng hợp hoặc orchestration. Ví dụ, nhiều dịch vụ có thể gửi dữ liệu đến một dịch vụ phân tích để xử lý.

Kiểu tương tác many-to-many (nhiều-đối-nhiều) liên quan đến nhiều dịch vụ giao tiếp với nhiều dịch vụ khác. Đây là kiểu tương tác phức tạp nhất và thường được triển khai bằng cách sử dụng một middleware như message broker hoặc service mesh. Ví dụ, nhiều dịch vụ có thể xuất bản các sự kiện đến một event bus, và nhiều dịch vụ khác có thể đăng ký để nhận các sự kiện này.

2.2.5. Các công nghệ và giao thức phổ biến

Có nhiều công nghệ và giao thức được sử dụng cho giao tiếp microservice. HTTP/REST là giao thức phổ biến nhất cho giao tiếp đồng bộ, sử dụng các phương thức HTTP và các tài nguyên đại diện. REST dựa trên các nguyên tắc của web và sử dụng các phương thức HTTP như GET, POST, PUT và DELETE để thực hiện các hoạt động trên tài nguyên. REST đơn giản, dễ hiểu và được hỗ trợ rộng rãi, làm cho nó trở thành một lựa chọn phổ biến cho giao tiếp microservice.

gRPC là một framework RPC hiệu suất cao, sử dụng HTTP/2 và Protocol Buffers. gRPC cung cấp hiệu suất tốt hơn REST nhờ sử dụng HTTP/2 và Protocol Buffers. HTTP/2 cung cấp multiplexing, cho phép nhiều yêu cầu và phản hồi được gửi qua một kết nối TCP duy nhất, trong khi Protocol Buffers cung cấp định dạng tuần tự hóa nhỏ gọn và hiệu quả hơn JSON hoặc XML. gRPC cũng hỗ trợ streaming hai chiều, cho phép các ứng dụng gửi và nhận luồng dữ liệu liên tục.

Message Queue (Hàng đợi thông điệp) là một mẫu giao tiếp bất đồng bộ, trong đó các dịch vụ gửi thông điệp đến một hàng đợi, và các dịch vụ khác nhận thông điệp từ hàng đợi. Các ví dụ phổ biến về middleware Message Queue bao gồm RabbitMQ, ActiveMQ và AWS SQS. Message Queue cung cấp một số lợi ích, bao gồm coupling lỏng lẻo, khả năng đệm và khả năng

xử lý lỗi được cải thiện. Tuy nhiên, chúng cũng đưa ra sự phức tạp bổ sung và có thể dẫn đến độ trễ cao hơn.

Pub/Sub (Publish/Subscribe) là một mẫu giao tiếp bất đồng bộ khác, trong đó các nhà xuất bản gửi thông điệp mà không biết ai sẽ nhận, và người đăng ký nhận thông điệp mà không biết ai đã gửi. Mẫu này thường được triển khai bằng Apache Kafka, AWS SNS/SQS, Google Pub/Sub hoặc NATS. Pub/Sub cung cấp coupling lỏng lẻo cao và khả năng mở rộng tốt, làm cho nó phù hợp cho các tình huống broadcasting và fanout. Tuy nhiên, giống như Message Queue, nó cũng đưa ra sự phức tạp bổ sung và có thể dẫn đến độ trễ cao hơn.

WebSockets là một giao thức cung cấp kênh liên lạc hai chiều toàn thời gian trên một kết nối TCP duy nhất. WebSockets được sử dụng cho giao tiếp thời gian thực, trong đó các dịch vụ cần trao đổi dữ liệu nhanh chóng trong cả hai hướng. Các ứng dụng phổ biến bao gồm ứng dụng trò chuyện, cập nhật thị trường tài chính và trò chơi trực tuyến. WebSockets cung cấp độ trễ thấp và overhead thấp, nhưng có thể khó triển khai và quản lý, đặc biệt là trong một hệ thống phân tán.

GraphQL là một ngôn ngữ truy vấn và thao tác dữ liệu, cung cấp một cách hiệu quả để truy xuất dữ liệu từ nhiều nguồn. GraphQL cho phép người dùng chỉ định chính xác dữ liệu họ cần, tránh over-fetching và under-fetching dữ liệu. Nó đặc biệt hữu ích cho các ứng dụng di động, nơi băng thông có thể hạn chế. GraphQL cung cấp một giao diện thống nhất cho nhiều dịch vụ, nhưng có thể phức tạp để triển khai và có thể dẫn đến các vấn đề về hiệu suất nếu không được sử dụng một cách thận trọng.

2.2.6. Thách thức trong giao tiếp microservice

Giao tiếp microservice đặt ra một số thách thức đáng kể. Network Reliability (Độ tin cậy của mạng) là một thách thức quan trọng. Mạng không đáng tin cậy và có thể gặp lỗi, dẫn đến mất thông điệp hoặc độ trễ cao. Các cơ chế để đối phó với độ tin cậy mạng thấp bao gồm retry, timeout và circuit breaker. Retry cho phép một dịch vụ thử lại một yêu cầu thất bại, timeout đặt giới hạn thời gian cho một yêu cầu, và circuit breaker ngăn một dịch vụ tiếp tục gửi yêu cầu đến một dịch vụ không phản hồi.

Service Discovery (Khám phá dịch vụ) là một thách thức khác. Trong một môi trường động, các dịch vụ cần phải tìm thấy nhau để giao tiếp. Các dịch vụ có thể thay đổi vị trí do triển khai mới, mở rộng theo chiều ngang hoặc khôi phục từ lỗi. Các giải pháp cho Service Discovery bao gồm Client-side Discovery, trong đó người dùng truy vấn một service registry để tìm vị trí của một dịch vụ, và Server-side Discovery, trong đó một thành phần trung gian như load balancer hoặc API gateway xử lý việc khám phá dịch vụ.

Load Balancing (Cân bằng tải) là một thách thức quan trọng khác. Phân phối tải công bằng giữa các phiên bản của cùng một dịch vụ là quan trọng để đảm bảo hiệu suất và độ tin cậy. Các cơ chế Load Balancing bao gồm Round Robin, Least Connections và Hash-based. Round Robin phân phối các yêu cầu đều nhau giữa các phiên bản, Least Connections phân phối các yêu cầu đến phiên bản có ít kết nối nhất, và Hash-based phân phối các yêu cầu dựa trên một giá trị hash, đảm bảo rằng các yêu cầu với cùng giá trị hash đi đến cùng một phiên bản.

Data Consistency (Tính nhất quán dữ liệu) là một thách thức đáng kể trong giao tiếp microservice. Duy trì tính nhất quán dữ liệu trên nhiều dịch vụ, đặc biệt là trong giao tiếp bất đồng bộ, có thể phức tạp. Các mẫu để đối phó với Data Consistency bao gồm Saga, Event Sourcing và CQRS. Saga quản lý giao dịch phân tán bằng cách sử dụng một chuỗi giao dịch địa phương với các hoạt động bù trừ, Event Sourcing lưu trữ trạng thái của hệ thống dưới dạng một chuỗi các sự kiện, và CQRS tách các hoạt động đọc và ghi thành các mô hình riêng biệt.

Versioning (Quản lý phiên bản) là một thách thức khác trong giao tiếp microservice. Quản lý các thay đổi API giữa các dịch vụ khi chúng phát triển có thể phức tạp. Các cơ chế để đối phó với

Versioning bao gồm Semantic Versioning, API Versioning và Backward Compatibility. Semantic Versioning sử dụng một hệ thống đánh số phiên bản rõ ràng (MAJOR.MINOR.PATCH) để truyền đạt tác động của các thay đổi, API Versioning duy trì nhiều phiên bản của cùng một API, và Backward Compatibility đảm bảo rằng các phiên bản mới của API có thể xử lý các yêu cầu được tạo ra cho các phiên bản cũ.

Error Handling (Xử lý lỗi) là một thách thức quan trọng trong giao tiếp microservice. Xử lý lỗi trong một hệ thống phân tán, nơi nhiều thứ có thể đi sai, có thể phức tạp. Các cơ chế để đối phó với Error Handling bao gồm Retry, Circuit Breaker và Fallback. Retry thử lại các yêu cầu thất bại, Circuit Breaker ngăn các yêu cầu đến các dịch vụ không phản hồi, và Fallback cung cấp một phản hồi thay thế khi một yêu cầu thất bại.

Cuối cùng, Monitoring and Debugging (Giám sát và gỡ lỗi) là một thách thức đáng kể trong giao tiếp microservice. Theo dõi và giải quyết vấn đề trong một hệ thống phân tán phức tạp có thể khó khăn. Các công cụ để đối phó với Monitoring and Debugging bao gồm Logging tập trung, Distributed Tracing và Metrics Collection. Logging tập trung thu thập log từ tất cả các dịch vụ vào một vị trí trung tâm, Distributed Tracing theo dõi yêu cầu khi chúng đi qua nhiều dịch vụ, và Metrics Collection thu thập các chỉ số về hiệu suất và sức khỏe của hệ thống.

2.2.7. Các mẫu giao tiếp (Communication Patterns)

Trong kiến trúc microservice, giao tiếp giữa các dịch vụ đóng vai trò quan trọng. Các dịch vụ cần trao đổi thông tin để phối hợp và hoàn thành các chức năng nghiệp vụ. Dưới đây là năm mẫu giao tiếp chính trong microservice:

Request-Response (Yêu cầu-Phản hồi) là mẫu giao tiếp đồng bộ phổ biến nhất, trong đó một dịch vụ gửi yêu cầu đến dịch vụ khác và đợi phản hồi trước khi tiếp tục xử lý. Dịch vụ gửi thiết lập kết nối và gửi yêu cầu HTTP/REST hoặc gRPC đến dịch vụ nhận, sau đó chờ đợi cho đến khi nhận được phản hồi. Dịch vụ nhận xử lý yêu cầu và trả về kết quả qua cùng một kết nối. Mẫu này có ưu điểm là đơn giản, dễ hiểu và triển khai, đồng thời đảm bảo tính nhất quán dữ liệu cao vì người gọi nhận được phản hồi ngay lập tức. Tuy nhiên, mẫu này cũng tạo ra coupling chặt chẽ giữa các dịch vụ, có hiệu suất kém trong trường hợp độ trễ mạng cao, và tiềm ẩn nguy cơ lỗi cascade nếu một dịch vụ trong chuỗi gọi không phản hồi.

Event-Driven (Hướng sự kiện) là mẫu trong đó các dịch vụ giao tiếp thông qua việc phát và lắng nghe các sự kiện, thường thông qua một message broker. Dịch vụ phát hành sự kiện không cần biết dịch vụ nào sẽ xử lý nó, và các dịch vụ khác đăng ký sự kiện quan tâm và phản ứng khi chúng xảy ra. Message broker đứng giữa đảm bảo việc phân phối sự kiện đến các dịch vụ đã đăng ký. Một ưu điểm lớn của mẫu này là tạo sự tách rời (decoupling) cao giữa các dịch vụ, do dịch vụ phát hành không cần biết ai đang lắng nghe. Khả năng mở rộng cũng rất tốt vì có thể thêm người lắng nghe mới mà không cần thay đổi người phát hành. Tuy nhiên, việc theo dõi luồng thực thi và gỡ lỗi trở nên phức tạp hơn, và có thể gây khó khăn trong việc duy trì tính nhất quán dữ liệu trong hệ thống phân tán.

Publish-Subscribe (Xuất bản-Đăng ký) là một dạng cụ thể của mẫu Event-Driven, cho phép phân phối thông tin từ một nguồn đến nhiều người nhận. Nhà xuất bản (publisher) gửi thông điệp đến một kênh hoặc chủ đề, và nhiều người đăng ký (subscribers) có thể nhận thông điệp từ cùng một kênh. Mẫu này thường được triển khai qua các nền tảng như Apache Kafka, RabbitMQ hoặc NATS. Mẫu Publish-Subscribe đặc biệt phù hợp cho các trường hợp cần truyền thông tin một-đến-nhiều, như thông báo về các sự kiện trong hệ thống. Việc mở rộng dễ dàng khi chỉ cần thêm người đăng ký mới mà không ảnh hưởng đến nhà xuất bản. Tuy nhiên, mẫu này làm tăng độ phức tạp trong quản lý tính nhất quán dữ liệu và có thể dẫn đến xử lý trùng lặp nếu không được cấu hình đúng.

Point-to-Point Messaging (Nhắn tin điểm-đến-điểm) là mẫu trong đó thông điệp được gửi từ một nguồn đến một đích cụ thể thông qua một hàng đợi. Một producer gửi thông điệp vào hàng đợi, và chỉ một consumer xử lý mỗi thông điệp. Thông điệp được lưu trữ trong hàng đợi cho đến khi được xử lý thành công, đảm bảo rằng không có thông điệp nào bị mất ngay cả khi consumer tạm thời không khả dụng. Mẫu Point-to-Point đảm bảo tin cậy cao vì thông điệp không bị mất và luôn được xử lý, ngay cả khi hệ thống gặp sự cố. Mẫu này phù hợp cho việc phân phối tác vụ và cân bằng tải giữa nhiều instance của cùng một dịch vụ. Tuy nhiên, nếu consumer xử lý chậm, có thể dẫn đến tình trạng nghẽn hàng đợi. Ngoài ra, mẫu này không phù hợp cho trường hợp nhiều dịch vụ khác nhau cần nhận cùng một thông tin.

Asynchronous Request-Response (Yêu cầu-Phản hồi bất đồng bộ) là biến thể bất đồng bộ của mẫu Request-Response, kết hợp ưu điểm của cả giao tiếp đồng bộ và bất đồng bộ. Dịch vụ gửi yêu cầu và tiếp tục xử lý mà không chờ đợi, trong khi dịch vụ nhận xử lý yêu cầu và gửi phản hồi (thường qua hàng đợi). Dịch vụ gửi được thông báo khi phản hồi có sẵn, thường thông qua callback, webhook hoặc long polling. Một lợi ích chính của mẫu này là tránh được việc chờ đợi và blocking, cải thiện hiệu suất và khả năng phản hồi của hệ thống. Dịch vụ gửi không bị chặn trong khi chờ đợi dịch vụ khác, nhưng vẫn duy trì được mối quan hệ yêu cầu-phản hồi. Tuy nhiên, việc triển khai và quản lý trở nên phức tạp hơn, đòi hỏi cơ chế tương quan giữa yêu cầu và phản hồi, cũng như cơ chế xử lý lỗi và timeout.

Mỗi mẫu giao tiếp có ưu và nhược điểm riêng, và lựa chọn mẫu phù hợp phụ thuộc vào các yêu cầu cụ thể như tính nhất quán, hiệu suất, khả năng mở rộng và độ tin cậy. Trong thực tế, một hệ thống microservice thường kết hợp nhiều mẫu giao tiếp khác nhau để giải quyết các tình huống khác nhau một cách hiệu quả.

2.3. Công nghệ và phương pháp đo lường hiệu năng

2.3.1. Các công nghệ triển khai microservices

Trong triển khai kiến trúc microservices, việc lựa chọn công nghệ phù hợp đóng vai trò quan trọng, ảnh hưởng trực tiếp đến hiệu suất, khả năng mở rộng và bảo trì của hệ thống. Đánh giá này sử dụng NestJS làm framework chính cho việc phát triển các microservices. NestJS là một framework Node.js tiến bộ, được phát triển dựa trên TypeScript, cung cấp kiến trúc ứng dụng được lấy cảm hứng từ Angular với các nguyên tắc SOLID và mô hình MVC. Framework này mang lại nhiều lợi ích trong phát triển microservices như hỗ trợ dependency injection, kiến trúc mô-đun hóa cao và tích hợp sẵn với nhiều công nghệ khác nhau.

NestJS sử dụng kiến trúc module mạnh mẽ cho phép tổ chức mã nguồn thành các thành phần có thể tái sử dụng và dễ bảo trì. Mỗi microservice trong bài đánh giá được triển khai như một ứng dụng NestJS độc lập, với cấu trúc bao gồm controllers (xử lý các yêu cầu HTTP), services (chứa logic nghiệp vụ), modules (đóng gói các thành phần liên quan) và entities (đại diện cho các đối tượng dữ liệu). NestJS cũng cung cấp một module microservices chuyên dụng hỗ trợ các giao thức như TCP, Redis, MQTT, gRPC, và Kafka, giúp đơn giản hóa việc triển khai các mẫu giao tiếp khác nhau.

TypeScript được chọn làm ngôn ngữ lập trình chính vì nó mang lại lợi thế của hệ thống kiểu dữ liệu tĩnh, giúp phát hiện lỗi sớm trong quá trình phát triển, tăng cường khả năng đọc hiểu và bảo trì mã nguồn. TypeScript cho phép xác định các interface và type rõ ràng cho các đối tượng dữ liệu và yêu cầu API, giảm thiểu các lỗi liên quan đến kiểu dữ liệu và cải thiện khả năng đọc hiểu mã nguồn. Đặc biệt, trong môi trường microservices nơi các dịch vụ giao tiếp qua mạng,

TypeScript giúp đảm bảo tính nhất quán của dữ liệu được truyền giữa các dịch vụ thông qua việc xác định các contract rõ ràng.

Để lưu trữ dữ liệu trong kiến trúc microservices, nguyên tắc ”mỗi dịch vụ có cơ sở dữ liệu riêng” được tuân thủ nhằm đảm bảo tính độc lập của các dịch vụ. TypeORM, một Object-Relational Mapping framework hiện đại cho TypeScript và JavaScript, được sử dụng để tương tác với cơ sở dữ liệu. TypeORM hỗ trợ nhiều hệ quản trị cơ sở dữ liệu và cung cấp các tính năng như quan hệ, kế thừa, migrations và nhiều kiểu lưu trữ dữ liệu khác nhau.

TypeORM sử dụng cách tiếp cận Active Record và Data Mapper, cho phép linh hoạt trong việc tương tác với cơ sở dữ liệu. Các entity trong TypeORM được định nghĩa bằng cách sử dụng decorators, giúp đơn giản hóa việc ánh xạ từ đối tượng trong code đến bảng trong cơ sở dữ liệu. Ngoài ra, TypeORM còn hỗ trợ các tính năng nâng cao như lazy loading, eager loading, transactions và query builder, giúp tối ưu hóa hiệu suất truy vấn cơ sở dữ liệu.

PostgreSQL được chọn làm hệ quản trị cơ sở dữ liệu chính do tính ổn định, hiệu suất cao và khả năng xử lý dữ liệu quan hệ phức tạp. PostgreSQL cung cấp hỗ trợ mạnh mẽ cho các kiểu dữ liệu phức tạp như JSON, JSONB và arrays, rất phù hợp cho các ứng dụng microservices hiện đại. Khả năng xử lý đồng thời và hỗ trợ transaction của PostgreSQL đảm bảo tính nhất quán dữ liệu trong môi trường phân tán.

Đối với giao tiếp giữa các microservices, bài đánh giá sử dụng nhiều công nghệ khác nhau để triển khai các mẫu giao tiếp. HTTP/REST API là nền tảng cho giao tiếp đồng bộ, với Axios được sử dụng làm HTTP client trong NestJS. Axios cung cấp API dựa trên Promise, hỗ trợ các tính năng như interceptors, timeout, và xử lý lỗi tiên tiến. NestJS cung cấp một lớp HttpService được xây dựng trên Axios, giúp đơn giản hóa việc gọi API từ một microservice đến microservice khác.

Trong mô hình RESTful, các microservices giao tiếp thông qua HTTP với các endpoint được định nghĩa rõ ràng. Controllers trong NestJS được trang bị các decorator như @Get(), @Post(), @Put(), và @Delete() để xử lý các phương thức HTTP tương ứng. NestJS cũng hỗ trợ validation thông qua các pipe và interceptor, đảm bảo dữ liệu được gửi và nhận đúng định dạng.

RabbitMQ, một message broker mạnh mẽ và đáng tin cậy, được triển khai cho các mẫu giao tiếp Point-to-Point và Asynchronous Request-Response. RabbitMQ cung cấp cơ chế bảo đảm tin cậy cao với các tính năng như xác nhận tin nhắn, hàng đợi bền vững và routing linh hoạt. RabbitMQ triển khai giao thức AMQP (Advanced Message Queuing Protocol), cho phép giao tiếp tin cậy và bảo mật giữa các dịch vụ.

Trong mô hình Point-to-Point với RabbitMQ, mỗi tin nhắn được gửi từ một producer đến một consumer thông qua một hàng đợi. RabbitMQ đảm bảo rằng mỗi tin nhắn chỉ được xử lý bởi một consumer, ngay cả khi có nhiều consumer cùng theo dõi một hàng đợi. Điều này đặc biệt hữu ích cho các tác vụ cần được xử lý chính xác một lần, như cập nhật đơn hàng hoặc xử lý thanh toán.

Đối với mô hình Asynchronous Request-Response, RabbitMQ được cấu hình với các hàng đợi phản hồi tạm thời và correlation IDs để theo dõi mối quan hệ giữa yêu cầu và phản hồi. Khi một microservice gửi yêu cầu, nó tạo một correlation ID duy nhất và một hàng đợi phản hồi tạm thời. Microservice nhận xử lý yêu cầu và gửi phản hồi đến hàng đợi tạm thời với cùng correlation ID, cho phép microservice gốc xác định đúng phản hồi cho yêu cầu của nó.

Apache Kafka, một nền tảng xử lý luồng sự kiện phân tán, được sử dụng cho các mẫu giao tiếp Publish/Subscribe và Event-Driven. Kafka nổi bật với khả năng xử lý hàng triệu sự kiện mỗi giây, độ trễ thấp và khả năng lưu trữ sự kiện lâu dài. Mô hình bảo lưu log của Kafka cho phép consumers đọc lại các sự kiện từ bất kỳ thời điểm nào trong quá khứ, điều này đặc biệt hữu ích cho việc phân tích dữ liệu và khôi phục sau sự cố.

Trong mô hình Publish/Subscribe với Kafka, các microservices phát hành sự kiện đến các topic, và nhiều consumer có thể đăng ký để nhận các sự kiện này. Kafka hỗ trợ phân vùng topic, cho phép xử lý song song và cân bằng tải giữa nhiều consumer trong cùng một consumer group. Điều này cải thiện đáng kể khả năng mở rộng của hệ thống, đặc biệt là trong các trường hợp khối lượng sự kiện lớn.

Mô hình Event-Driven với Kafka tận dụng khả năng lưu trữ và phát lại sự kiện của nền tảng này. Các microservices phát hành sự kiện khi trạng thái của chúng thay đổi, và các dịch vụ khác phản ứng với những thay đổi này. Mô hình này tạo ra sự tách rời cao giữa các dịch vụ, vì service phát hành không cần biết về các dịch vụ nào đang lắng nghe sự kiện của nó.

NestJS cung cấp tích hợp cho cả RabbitMQ và Kafka thông qua module microservices, giúp đơn giản hóa việc triển khai các mẫu giao tiếp khác nhau. Module này cung cấp các decorators như @MessagePattern() và @EventPattern() để xử lý các tin nhắn và sự kiện từ các transport khác nhau. NestJS cũng hỗ trợ serialization và deserialization tự động, giúp đơn giản hóa việc chuyển đổi giữa các định dạng tin nhắn khác nhau.

2.3.2. Các thông số đo lường chính

Để đánh giá hiệu năng của các mẫu giao tiếp trong microservices, cần xem xét một tập hợp các thông số đo lường toàn diện. Latency (Độ trễ) là một trong những thông số quan trọng nhất, đại diện cho thời gian cần thiết để hoàn thành một yêu cầu, từ khi gửi đến khi nhận phản hồi. Độ trễ thường được đo bằng mili giây (ms) và phản ánh trực tiếp trải nghiệm người dùng. Trong kiến trúc microservices, độ trễ có thể bị ảnh hưởng bởi nhiều yếu tố như khoảng cách vật lý giữa các dịch vụ, phương pháp tuần tự hóa dữ liệu, cơ chế giao tiếp được chọn và tải mạng. Độ trễ cao làm giảm khả năng phản hồi của hệ thống và có thể ảnh hưởng đáng kể đến sự hài lòng của người dùng.

Độ trễ trong microservices thường được phân tích theo nhiều khía cạnh khác nhau. Độ trễ đầu cuối (end-to-end latency) đo lường tổng thời gian từ khi client gửi yêu cầu đến khi nhận được phản hồi đầy đủ. Độ trễ dịch vụ (service latency) đo lường thời gian xử lý trong một microservice cụ thể, không bao gồm thời gian giao tiếp mạng. Độ trễ mạng (network latency) đo lường thời gian cần thiết để dữ liệu di chuyển giữa các dịch vụ qua mạng. Phân tách các loại độ trễ này giúp xác định chính xác các điểm nghẽn trong hệ thống và tối ưu hóa hiệu suất một cách có mục tiêu.

Trong các mẫu giao tiếp đồng bộ như Request-Response, độ trễ thường đơn giản để đo lường vì nó bằng với thời gian từ khi gửi yêu cầu đến khi nhận được phản hồi. Tuy nhiên, trong các mẫu giao tiếp bất đồng bộ, việc đo lường độ trễ phức tạp hơn. Đối với Asynchronous Request-Response, cần theo dõi thời gian từ khi gửi yêu cầu đến khi nhận được phản hồi bất đồng bộ, thường thông qua correlation IDs hoặc cơ chế tương tự. Đối với Event-Driven và Publish/Subscribe, độ trễ có thể được đo lường là thời gian từ khi một sự kiện được phát hành đến khi nó được xử lý bởi tất cả các consumer liên quan.

Throughput (Thông lượng) đo lường số lượng yêu cầu mà hệ thống có thể xử lý trong một đơn vị thời gian, thường được biểu thị bằng yêu cầu trên giây (RPS) hoặc giao dịch trên giây (TPS). Thông lượng cao là một chỉ số của hệ thống có khả năng xử lý đồng thời nhiều yêu cầu, điều này đặc biệt quan trọng cho các ứng dụng có tải cao. Trong bài đánh giá này, thông lượng được đánh giá dưới các mức tải khác nhau để hiểu rõ hơn về hiệu suất của mỗi mẫu giao tiếp trong các tình huống khác nhau.

Thông lượng trong microservices có thể được đo lường ở nhiều cấp độ khác nhau. Thông lượng hệ thống (system throughput) đo lường tổng số yêu cầu mà toàn bộ hệ thống có thể xử lý trong một giây. Thông lượng dịch vụ (service throughput) đo lường số lượng yêu cầu mà

một microservice cụ thể có thể xử lý. Thông lượng endpoint (endpoint throughput) đo lường số lượng yêu cầu mà một endpoint cụ thể trong một dịch vụ có thể xử lý. Việc phân tích thông lượng ở các cấp độ khác nhau giúp xác định các điểm nghẽn và cơ hội mở rộng trong hệ thống.

Các mẫu giao tiếp khác nhau có thể ảnh hưởng đáng kể đến thông lượng của hệ thống. Các mẫu đồng bộ như Request-Response thường có thông lượng thấp hơn do tính chất tuần tự của chúng, trong khi các mẫu bất đồng bộ như Publish/Subscribe và Event-Driven có thể đạt thông lượng cao hơn do khả năng xử lý song song. Tuy nhiên, điều này phụ thuộc nhiều vào trường hợp sử dụng cụ thể và cách triển khai.

Error Rate (Tỷ lệ lỗi) là tỷ lệ phần trăm của các yêu cầu thất bại so với tổng số yêu cầu được gửi đến hệ thống. Tỷ lệ lỗi có thể bị ảnh hưởng bởi nhiều yếu tố như lỗi mạng, lỗi dịch vụ, timeout hoặc lỗi logic nghiệp vụ. Một tỷ lệ lỗi cao không chỉ ảnh hưởng đến trải nghiệm người dùng mà còn làm giảm độ tin cậy tổng thể của hệ thống. Trong các mẫu giao tiếp bất đồng bộ, việc theo dõi và xử lý lỗi phức tạp hơn so với các mẫu đồng bộ, do đó tỷ lệ lỗi là một thông số đặc biệt quan trọng cần xem xét.

Trong kiến trúc microservices, các loại lỗi khác nhau có thể xảy ra, bao gồm lỗi mạng (network errors), lỗi thời gian chờ (timeout errors), lỗi dịch vụ (service errors) và lỗi logic nghiệp vụ (business logic errors). Mỗi loại lỗi cần được phân loại và xử lý riêng biệt. Ví dụ, lỗi mạng tạm thời có thể được giải quyết bằng cách thử lại, trong khi lỗi dịch vụ có thể yêu cầu failover sang một instance khác hoặc kích hoạt circuit breaker để ngăn chặn lỗi lan truyền.

Các mẫu giao tiếp khác nhau có cách tiếp cận khác nhau đối với xử lý lỗi. Trong Request-Response, lỗi thường được báo cáo ngay lập tức thông qua mã trạng thái HTTP hoặc thông báo lỗi. Trong Asynchronous Request-Response, lỗi có thể được báo cáo thông qua callback hoặc hàng đợi lỗi riêng biệt. Trong Event-Driven và Publish/Subscribe, việc xử lý lỗi phức tạp hơn và có thể yêu cầu cơ chế như dead-letter queues hoặc retry topics.

Resource Utilization (Sử dụng tài nguyên) đề cập đến lượng tài nguyên hệ thống như CPU, bộ nhớ và băng thông mạng được sử dụng bởi các microservices. Sử dụng tài nguyên cao có thể dẫn đến hiệu suất giảm và chỉ ra nhu cầu mở rộng theo chiều ngang hoặc tối ưu hóa hiệu suất. Trong bài đánh giá này, sử dụng tài nguyên được giám sát cho từng microservice riêng biệt cũng như cho toàn bộ hệ thống, cho phép phân tích chi tiết về hiệu quả sử dụng tài nguyên của mỗi mẫu giao tiếp.

Sử dụng CPU là một trong những thông số quan trọng nhất trong sử dụng tài nguyên. Nó đo lường tỷ lệ phần trăm thời gian CPU được sử dụng bởi các tiến trình microservice. Sử dụng CPU cao có thể chỉ ra nhu cầu tối ưu hóa mã nguồn hoặc mở rộng theo chiều ngang. Sử dụng bộ nhớ đo lường lượng RAM được sử dụng bởi các microservices. Sử dụng bộ nhớ cao có thể dẫn đến swapping và hiệu suất giảm. Băng thông mạng đo lường lượng dữ liệu được truyền qua mạng giữa các microservices. Băng thông mạng cao có thể chỉ ra nhu cầu tối ưu hóa định dạng dữ liệu hoặc giảm lượng dữ liệu được truyền.

Các mẫu giao tiếp khác nhau có yêu cầu tài nguyên khác nhau. Các mẫu đồng bộ như Request-Response thường có yêu cầu CPU và bộ nhớ thấp hơn nhưng có thể sử dụng nhiều kết nối mạng đồng thời. Các mẫu bất đồng bộ như Publish/Subscribe và Event-Driven có thể có yêu cầu CPU và bộ nhớ cao hơn do cần xử lý và lưu trữ tin nhắn, nhưng chúng thường sử dụng kết nối mạng hiệu quả hơn do khả năng batch processing.

Scalability (Khả năng mở rộng) đo lường khả năng của hệ thống trong việc xử lý tải tăng bằng cách thêm tài nguyên. Khả năng mở rộng tốt có nghĩa là hiệu suất tăng tỷ lệ thuận với tài nguyên được thêm vào. Các mẫu giao tiếp khác nhau có thể ảnh hưởng đáng kể đến khả năng mở rộng của hệ thống microservices. Ví dụ, các mẫu bất đồng bộ thường có khả năng mở rộng tốt hơn so với các mẫu đồng bộ do chúng tạo ra ít sự phụ thuộc trực tiếp hơn giữa các dịch vụ.

Khả năng mở rộng theo chiều ngang (horizontal scalability) đề cập đến khả năng tăng hiệu suất bằng cách thêm nhiều instance của các microservices. Khả năng mở rộng theo chiều dọc (vertical scalability) đề cập đến khả năng tăng hiệu suất bằng cách thêm tài nguyên cho các instance hiện có. Trong kiến trúc microservices, khả năng mở rộng theo chiều ngang thường được ưu tiên do tính linh hoạt và khả năng chịu lỗi cao hơn.

Các mẫu giao tiếp khác nhau hỗ trợ khả năng mở rộng theo những cách khác nhau. Các mẫu đồng bộ như Request-Response có thể gặp khó khăn khi mở rộng do yêu cầu kết nối trực tiếp giữa các dịch vụ. Điều này có thể được giải quyết bằng cách sử dụng load balancer hoặc API gateway. Các mẫu bất đồng bộ như Publish/Subscribe và Event-Driven thường có khả năng mở rộng tốt hơn do chúng tách rời các producer và consumer, cho phép chúng mở rộng độc lập.

Consistency (Tính nhất quán) trong context của microservices là khả năng duy trì trạng thái dữ liệu đồng bộ giữa các dịch vụ khác nhau. Đối với các mẫu giao tiếp đồng bộ, tính nhất quán thường dễ đạt được hơn do tính chất tuần tự của các hoạt động. Tuy nhiên, đối với các mẫu bất đồng bộ, đặc biệt là trong các kịch bản phân tán dữ liệu, tính nhất quán trở thành một thách thức và cần được đo lường cẩn thận. bài đánh giá này đánh giá mức độ nhất quán dữ liệu đạt được bởi các mẫu giao tiếp khác nhau trong các kịch bản như kiểm tra tồn kho và cập nhật đơn hàng.

Tính nhất quán mạnh (strong consistency) đảm bảo rằng tất cả các dịch vụ luôn nhìn thấy dữ liệu mới nhất. Điều này thường đạt được thông qua các giao thức phức tạp hoặc truy cập trực tiếp đến cùng một cơ sở dữ liệu. Tính nhất quán cuối cùng (eventual consistency) chấp nhận rằng các dịch vụ có thể tạm thời nhìn thấy dữ liệu không đồng bộ, nhưng cuối cùng tất cả sẽ đồng bộ. Mô hình này thường được sử dụng trong các hệ thống phân tán quy mô lớn do tính thực tế và hiệu suất cao hơn.

Các mẫu giao tiếp khác nhau hỗ trợ các mô hình nhất quán khác nhau. Các mẫu đồng bộ như Request-Response thường hỗ trợ tính nhất quán mạnh do tính chất tuần tự của chúng. Các mẫu bất đồng bộ như Publish/Subscribe và Event-Driven thường hỗ trợ tính nhất quán cuối cùng, đòi hỏi thiết kế cẩn thận để đảm bảo dữ liệu cuối cùng sẽ đồng bộ.

2.3.3. Phương pháp đo lường

Để thu thập dữ liệu hiệu năng toàn diện về các mẫu giao tiếp microservices, bài đánh giá này áp dụng nhiều phương pháp đo lường bổ sung cho nhau. Load Testing (Kiểm thử tải) là một phương pháp cơ bản được sử dụng để mô phỏng các điều kiện tải thực tế và đánh giá hiệu năng của hệ thống dưới áp lực. Thông qua kiểm thử tải, các điểm nghẽn, giới hạn và điểm lỗi trong hệ thống có thể được xác định. Trong bài đánh giá này, các kịch bản kiểm thử tải được thiết kế để mô phỏng các trường hợp sử dụng thực tế như kiểm tra tồn kho, xử lý thanh toán và thông báo đơn hàng với các mức tải khác nhau.

Kiểm thử tải có thể được thực hiện theo nhiều cách khác nhau, bao gồm kiểm thử tăng dần (ramp-up testing), kiểm thử chịu tải (stress testing), kiểm thử phá vỡ (spike testing) và kiểm thử độ bền (endurance testing). Kiểm thử tăng dần tăng tải từ từ, cho phép đánh giá hiệu suất của hệ thống khi tải tăng lên. Kiểm thử chịu tải đẩy hệ thống đến giới hạn để xác định điểm vỡ. Kiểm thử phá vỡ đánh giá phản ứng của hệ thống đối với sự gia tăng tải đột ngột. Kiểm thử độ bền đánh giá hiệu suất của hệ thống trong thời gian dài.

Trong bài đánh giá này, các kịch bản kiểm thử tải được thiết kế để mô phỏng các trường hợp sử dụng thực tế của hệ thống microservices. Các kịch bản này bao gồm các hoạt động như tạo đơn hàng, kiểm tra tồn kho, xử lý thanh toán và gửi thông báo. Mỗi kịch bản được thiết kế để tập trung vào một mẫu giao tiếp cụ thể, cho phép so sánh trực tiếp hiệu suất của các mẫu khác nhau trong cùng một ngữ cảnh.

Benchmarking (Đánh giá) là phương pháp so sánh hiệu năng của các cấu hình hệ thống khác nhau trong điều kiện tiêu chuẩn. Benchmarking cho phép xác định cấu hình hiệu quả nhất cho một trường hợp sử dụng cụ thể và theo dõi hiệu năng theo thời gian khi hệ thống phát triển. Trong bài đánh giá này, benchmarking được sử dụng để so sánh hiệu suất của các mẫu giao tiếp khác nhau dưới các điều kiện tải giống nhau, cung cấp cái nhìn trực quan về hiệu quả tương đối của mỗi mẫu.

Việc thiết lập benchmark chuẩn đòi hỏi sự cẩn thận trong thiết kế thử nghiệm để đảm bảo kết quả có ý nghĩa và có thể tái hiện. Điều này bao gồm việc kiểm soát môi trường thử nghiệm, chẳng hạn như cấu hình phần cứng và phần mềm, tải nền và độ trễ mạng. Cũng cần xác định các metric phù hợp để đo lường, chẳng hạn như thời gian phản hồi, thông lượng và sử dụng tài nguyên.

Trong bài đánh giá này, benchmark được tiến hành cho mỗi mẫu giao tiếp với các trường hợp thử nghiệm giống nhau, bao gồm Order-Inventory, Order-Payment và Order-Notification. Mỗi trường hợp thử nghiệm được chạy với các mức tải tăng dần, từ 10 đến 100 người dùng đồng thời, để đánh giá khả năng mở rộng của mỗi mẫu. Các metric được thu thập bao gồm thời gian phản hồi trung bình, thời gian phản hồi phân vị thứ 95, thông lượng, tỷ lệ lỗi và sử dụng tài nguyên.

Một khía cạnh quan trọng của benchmarking là khả năng so sánh hiệu suất tương đối của các mẫu giao tiếp khác nhau. Trong bài đánh giá này, các bảng và biểu đồ so sánh được tạo để trực quan hóa sự khác biệt về hiệu suất giữa các mẫu. Điều này giúp xác định mẫu nào phù hợp nhất cho các trường hợp sử dụng cụ thể, dựa trên các yêu cầu về hiệu suất và khả năng mở rộng.

Profiling (Lập hồ sơ) là một phương pháp phân tích chi tiết tài nguyên được sử dụng và thời gian thực thi của các thành phần hệ thống. Profiling cung cấp thông tin chi tiết về hiệu suất của các phần cụ thể trong mã nguồn và giúp xác định các đoạn mã không hiệu quả hoặc sử dụng tài nguyên nhiều. Bằng cách phân tích dữ liệu profiling, các cơ hội tối ưu hóa hiệu suất có thể được xác định và triển khai.

Trong môi trường Node.js, profiling có thể được thực hiện bằng cách sử dụng các công cụ như Node.js Profiler, Chrome DevTools Profiler hoặc các thư viện bên thứ ba như clinic.js. Các công cụ này cho phép thu thập dữ liệu về thời gian CPU, sử dụng bộ nhớ, hoạt động của garbage collector và các chỉ số hiệu suất khác. Dữ liệu này có thể được phân tích để xác định các điểm nghẽn hiệu suất và tối ưu hóa mã nguồn.

Trong bài đánh giá này, profiling được sử dụng để phân tích hiệu suất của các thành phần khác nhau trong mỗi mẫu giao tiếp. Cụ thể, profiling giúp xác định thời gian dành cho việc serialization/deserialization dữ liệu, xử lý mạng, xử lý logic nghiệp vụ và tương tác với cơ sở dữ liệu. Thông tin này rất có giá trị trong việc hiểu rõ hơn về hiệu quả của mỗi mẫu giao tiếp và xác định các cơ hội tối ưu hóa.

Một khía cạnh quan trọng của profiling trong kiến trúc microservices là khả năng phân tích hiệu suất xuyên suốt các dịch vụ. Điều này đòi hỏi sự phối hợp trong việc thu thập và phân tích dữ liệu profiling từ nhiều dịch vụ, thường thông qua các công cụ như distributed tracing. Bằng cách kết hợp dữ liệu profiling từ nhiều dịch vụ, có thể xác định các tương tác không hiệu quả giữa các dịch vụ và tối ưu hóa hiệu suất tổng thể của hệ thống.

Distributed Tracing (Theo dõi phân tán) là phương pháp theo dõi yêu cầu khi chúng đi qua nhiều dịch vụ trong một hệ thống phân tán. Phương pháp này đặc biệt quan trọng trong kiến trúc microservices, nơi một yêu cầu người dùng có thể đi qua nhiều dịch vụ khác nhau trước khi hoàn thành. Distributed tracing giúp xác định các điểm nghẽn, hiểu luồng yêu cầu và mối quan hệ phụ thuộc giữa các dịch vụ. Trong bài đánh giá này, distributed tracing được sử dụng để phân tích chi tiết luồng giao tiếp giữa các microservices và xác định các điểm tối ưu tiềm năng.

Trong môi trường Node.js, distributed tracing có thể được triển khai bằng cách sử dụng các thư viện như OpenTelemetry, một framework mã nguồn mở cung cấp API, thư viện và tác nhân để thu thập dữ liệu theo dõi phân tán. OpenTelemetry có thể được tích hợp với NestJS thông qua các interceptor và middleware, cho phép tự động thu thập dữ liệu tracing từ các yêu cầu HTTP và các tương tác microservices.

Mỗi trace trong distributed tracing đại diện cho một yêu cầu đi qua hệ thống và bao gồm một hoặc nhiều span. Mỗi span đại diện cho một hoạt động đơn lẻ trong trace, chẳng hạn như yêu cầu HTTP, truy vấn cơ sở dữ liệu hoặc xử lý logic nghiệp vụ. Spans được tổ chức thành một cấu trúc phân cấp, với các span con lồng bên trong các span cha, tạo thành một cây trace. Thông tin này cho phép hiểu rõ về luồng yêu cầu và các mối quan hệ phụ thuộc giữa các hoạt động.

Trong bài đánh giá này, distributed tracing được sử dụng để phân tích chi tiết luồng yêu cầu trong mỗi mẫu giao tiếp. Cụ thể, nó giúp xác định thời gian dành cho các hoạt động khác nhau trong quá trình xử lý yêu cầu, chẳng hạn như giao tiếp giữa các dịch vụ, xử lý logic nghiệp vụ và tương tác với cơ sở dữ liệu. Thông tin này rất có giá trị trong việc hiểu rõ hơn về hiệu quả của mỗi mẫu giao tiếp và xác định các cơ hội tối ưu hóa.

Metrics Collection (Thu thập số liệu) là quá trình thu thập và phân tích các chỉ số hiệu năng của hệ thống theo thời gian. Metrics collection cho phép theo dõi xu hướng hiệu năng, phát hiện bất thường và thiết lập cảnh báo cho các vấn đề tiềm ẩn. Trong bài đánh giá này, các số liệu như thời gian phản hồi, tỷ lệ lỗi, thông lượng và sử dụng tài nguyên được thu thập liên tục từ tất cả các microservices và các thành phần hỗ trợ như message broker và cơ sở dữ liệu.

Trong môi trường Node.js, metrics collection có thể được triển khai bằng cách sử dụng các thư viện như Prom-Client, một thư viện mã nguồn mở cung cấp các primitive để thu thập và hiển thị metrics trong định dạng mà Prometheus có thể hiểu được. Prom-Client có thể được tích hợp với NestJS thông qua một module tùy chỉnh, cho phép tự động thu thập các metrics tiêu chuẩn như thời gian phản hồi HTTP, số lượng yêu cầu đang xử lý và sử dụng bộ nhớ.

Các loại metrics khác nhau có thể được thu thập, bao gồm counters, gauges, histograms và summaries. Counters là các giá trị chỉ tăng lên, được sử dụng để đếm các sự kiện như số lượng yêu cầu hoặc lỗi. Gauges là các giá trị có thể tăng hoặc giảm, được sử dụng để đo lường các giá trị như số lượng kết nối đồng thời hoặc kích thước hàng đợi. Histograms và summaries đo lường phân phối các giá trị như thời gian phản hồi hoặc kích thước yêu cầu, cho phép tính toán các percentile như p50, p95 và p99.

Trong bài đánh giá này, metrics collection được sử dụng để thu thập dữ liệu hiệu năng từ tất cả các microservices và các thành phần hỗ trợ. Cụ thể, các metrics sau đây được thu thập:

- HTTP metrics: Thời gian phản hồi, tỷ lệ lỗi và số lượng yêu cầu cho mỗi endpoint HTTP.
- Microservice metrics: Thời gian xử lý, tỷ lệ lỗi và số lượng tin nhắn cho mỗi pattern microservice.
- Database metrics: Thời gian truy vấn, số lượng truy vấn và số lượng kết nối cho mỗi cơ sở dữ liệu.
- Message broker metrics: Kích thước hàng đợi, tốc độ tin nhắn và độ trễ tin nhắn cho mỗi hàng đợi hoặc topic.
- System metrics: Sử dụng CPU, sử dụng bộ nhớ và băng thông mạng cho mỗi service.

Dữ liệu metrics được thu thập và lưu trữ trong một hệ thống time-series database, cho phép truy vấn và phân tích dữ liệu theo thời gian. Điều này cho phép theo dõi xu hướng hiệu năng, phát hiện bất thường và thiết lập cảnh báo cho các vấn đề tiềm ẩn. Dữ liệu metrics cũng được

sử dụng để tạo bảng điều khiển hiển thị hiệu suất của hệ thống theo thời gian thực, giúp giám sát và khắc phục sự cố.

2.3.4. Công cụ đo lường hiệu năng

Để thực hiện các phương pháp đo lường đã nêu, bài đánh giá triển khai một bộ công cụ toàn diện cho việc thu thập và phân tích dữ liệu hiệu năng. K6, một công cụ kiểm thử tải mã nguồn mở dựa trên JavaScript, được sử dụng để tạo tải và đo lường hiệu năng của các mẫu giao tiếp microservices khác nhau. K6 cho phép viết các tập lệnh kiểm thử phức tạp mô phỏng hành vi người dùng thực tế, hỗ trợ thực hiện các yêu cầu HTTP, WebSocket và gRPC. Ngoài ra, K6 cung cấp tính năng phân tích dữ liệu tích hợp, cho phép tính toán các số liệu thống kê về độ trễ, thông lượng và tỷ lệ lỗi.

K6 nổi bật với khả năng mở rộng và tùy chỉnh cao. Nó cho phép viết các tập lệnh kiểm thử bằng JavaScript hiện đại, với hỗ trợ cho các tính năng như async/await, modules và promises. K6 cũng cung cấp một API phong phú để tạo và kiểm soát tải, bao gồm các chức năng như ramp-up, constant load, và step-load. Điều này cho phép mô phỏng các tình huống tải thực tế và đánh giá hiệu năng của hệ thống dưới các điều kiện khác nhau.

Trong đánh giá này, K6 được sử dụng để tạo tải và đo lường hiệu năng của các mẫu giao tiếp microservices khác nhau. Các tập lệnh K6 được viết để mô phỏng các kịch bản thực tế như kiểm tra tồn kho, xử lý thanh toán và thông báo đơn hàng. Mỗi tập lệnh được thiết kế để tập trung vào một mẫu giao tiếp cụ thể, cho phép so sánh trực tiếp hiệu suất của các mẫu khác nhau trong cùng một ngữ cảnh.

Ví dụ, để kiểm thử hiệu năng của mẫu giao tiếp Request-Response trong kịch bản kiểm tra tồn kho, K6 được cấu hình để gửi yêu cầu HTTP GET đến endpoint /inventory/check/productId của Order Service. Thời gian phản hồi, tỷ lệ lỗi và thông lượng được đo lường và so sánh với các mẫu giao tiếp khác như Asynchronous Request-Response và Publish/Subscribe.

Prometheus, một hệ thống giám sát mã nguồn mở, được triển khai để thu thập và lưu trữ các số liệu hiệu năng từ các microservices. Prometheus sử dụng mô hình pull để thu thập số liệu, trong đó nó truy vấn định kỳ các mục tiêu được cấu hình. Mô hình này đơn giản hóa việc triển khai và mở rộng, đồng thời cung cấp khả năng phát hiện tự động các instance mới. Prometheus cũng cung cấp ngôn ngữ truy vấn mạnh mẽ PromQL, cho phép truy vấn và phân tích phức tạp các số liệu thu thập được.

Prometheus nổi bật với khả năng mở rộng và tính tin cậy cao. Nó sử dụng một mô hình dữ liệu time-series hiệu quả, cho phép lưu trữ và truy vấn hàng triệu time-series với hiệu suất cao. Prometheus cũng cung cấp một hệ thống cảnh báo mạnh mẽ, cho phép định nghĩa các quy tắc cảnh báo dựa trên các biểu thức PromQL. Khi một quy tắc cảnh báo được kích hoạt, Prometheus có thể gửi thông báo đến các hệ thống như Alertmanager, Slack hoặc Email.

Trong đánh giá này, Prometheus được triển khai để thu thập các metric từ các microservices, bao gồm thời gian phản hồi, tỷ lệ lỗi và sử dụng tài nguyên. Các microservices được cấu hình để hiển thị các endpoint metrics mà Prometheus có thể truy vấn, thường là /metrics. Prometheus được cấu hình để truy vấn các endpoint này định kỳ, thường là mỗi 15 giây, và lưu trữ các giá trị thu thập được trong cơ sở dữ liệu time-series của nó.

Các metric được thu thập bởi Prometheus bao gồm các metric hệ thống như sử dụng CPU, sử dụng bộ nhớ và băng thông mạng, cũng như các metric ứng dụng như thời gian phản hồi HTTP, tỷ lệ lỗi và số lượng yêu cầu. Các metric này được sử dụng để phân tích hiệu suất của các mẫu giao tiếp khác nhau và xác định các điểm nghẽn và cơ hội tối ưu hóa.

Việc kết hợp các công nghệ triển khai và công cụ đo lường hiệu năng này tạo thành một môi trường toàn diện cho việc đánh giá và so sánh các mẫu giao tiếp khác nhau trong kiến trúc

microservices. Thông qua việc thu thập và phân tích dữ liệu hiệu năng từ nhiều góc độ, nghiên cứu có thể cung cấp cái nhìn sâu sắc về ưu và nhược điểm của mỗi mẫu giao tiếp và đưa ra khuyến nghị dựa trên bằng chứng cho việc lựa chọn mẫu giao tiếp phù hợp trong các tình huống khác nhau.

2.4. Tổng kết

Chương này đã cung cấp một cái nhìn tổng quan về kiến trúc microservice và vai trò quan trọng của giao tiếp trong kiến trúc này. Chúng ta đã thảo luận về các đặc điểm chính của microservices, so sánh với kiến trúc nguyên khối, và xem xét các lợi ích cũng như thách thức.

Về mặt đặc điểm, microservice là một kiến trúc phân tán, trong đó mỗi dịch vụ tự trị, tập trung vào một chức năng nghiệp vụ cụ thể, quản lý dữ liệu riêng của nó, được thiết kế để xử lý lỗi, và có thể phát triển độc lập. So với kiến trúc nguyên khối, microservice cung cấp khả năng mở rộng có mục tiêu, phát triển nhanh hơn, tính linh hoạt công nghệ, khả năng chịu lỗi tốt hơn, và khả năng bảo trì và hiểu biết tốt hơn.

Tuy nhiên, microservice cũng đặt ra một số thách thức, bao gồm độ phức tạp phân tán, giao tiếp giữa các dịch vụ, quản lý dữ liệu, vận hành và giám sát, và kiểm thử. Để giải quyết những thách thức này, một số nguyên tắc thiết kế nên được tuân thủ, bao gồm Single Responsibility Principle, Domain-Driven Design, API First, tự động hóa, Monitoring và Observability, và Fault Tolerance.

Trong phần về giao tiếp, chúng ta đã khám phá vai trò quan trọng của giao tiếp trong kiến trúc microservice, bao gồm tạo điều kiện cho sự hợp tác giữa các dịch vụ, đảm bảo tính nhất quán dữ liệu, hỗ trợ khả năng chịu lỗi, và cho phép tính mở rộng. Chúng ta cũng đã thảo luận về các thuộc tính quan trọng của giao tiếp microservice, bao gồm độ tin cậy, độ trễ, khả năng mở rộng, cách ly lỗi, tính nhất quán, định dạng dữ liệu, khả năng tương tác, và bảo mật.

Hai mô hình giao tiếp cơ bản trong microservices là đồng bộ và bất đồng bộ. Trong giao tiếp đồng bộ, người gửi đợi phản hồi từ người nhận, trong khi trong giao tiếp bất đồng bộ, người gửi không đợi phản hồi. Ngoài ra, các microservice cũng giao tiếp theo các kiểu tương tác khác nhau, bao gồm one-to-one, one-to-many, many-to-one, và many-to-many.

Có nhiều công nghệ và giao thức được sử dụng cho giao tiếp microservice, bao gồm HTTP/REST, gRPC, Message Queue, Pub/Sub, WebSockets và GraphQL. Mỗi công nghệ có ưu và nhược điểm riêng và phù hợp với các tình huống khác nhau.

Giao tiếp microservice đặt ra một số thách thức, bao gồm Network Reliability, Service Discovery, Load Balancing, Data Consistency, Versioning, Error Handling, và Monitoring and Debugging. Để giải quyết những thách thức này, một số mẫu giao tiếp đã được phát triển, bao gồm API Gateway, Circuit Breaker, Bulkhead, Retry, Timeout, Saga, Event Sourcing, và CQRS.

Trong phần về đo lường hiệu năng, chúng ta đã thảo luận về các thông số đo lường chính, bao gồm Latency, Throughput, Error Rate, Resource Utilization, và Scalability. Chúng ta cũng đã khám phá các phương pháp đo lường, bao gồm Load Testing, Benchmarking, Profiling, Distributed Tracing, và Metrics Collection. Cuối cùng, chúng ta đã giới thiệu một số công cụ phổ biến để đo lường hiệu năng của microservices, bao gồm K6, Prometheus, Grafana, Jaeger/Zipkin, và ELK Stack.

Các khái niệm và hiểu biết từ chương này sẽ làm nền tảng cho các chương tiếp theo, nơi chúng ta sẽ đi sâu vào việc phân tích chi tiết các mẫu giao tiếp cụ thể trong kiến trúc microservice. Chúng ta sẽ phân loại các mẫu này theo tiêu chí đồng bộ/bất đồng bộ và one-to-one/one-to-many, phân tích ưu và nhược điểm của từng mẫu, và cung cấp hướng dẫn cho việc lựa chọn mẫu phù hợp cho các tình huống cụ thể.

Việc hiểu rõ các khái niệm cơ bản và thách thức của giao tiếp microservice sẽ giúp chúng ta đánh giá tốt hơn hiệu quả của các mẫu giao tiếp trong các kịch bản thực tế. Đồng thời, các phương pháp và công cụ đo lường hiệu năng đã được giới thiệu sẽ được áp dụng trong các phần tiếp theo để đánh giá hiệu suất của các mẫu giao tiếp và đưa ra các khuyến nghị dựa trên dữ liệu.

Tóm lại, chương này đã cung cấp một cái nhìn toàn diện về kiến trúc microservice và vai trò quan trọng của giao tiếp trong kiến trúc này. Chúng ta đã hiểu được các đặc điểm, lợi ích và thách thức của microservice, cũng như các mô hình giao tiếp, công nghệ và mẫu thiết kế phổ biến. Những kiến thức này sẽ là nền tảng vững chắc cho các phân tích chi tiết hơn trong các chương tiếp theo.

Chương 3.
Phân tích các Communication Patterns

3.1. Cách phân loại các pattern

3.1.1. Tiêu chí phân loại theo communication mode

- Synchronous Communication
  - + REST API
  - + gRPC
  - + GraphQL
- Asynchronous Communication
  - + Message Queue
  - + Event Bus
  - + Pub/Sub

3.1.2. Tiêu chí phân loại theo communication scope

- One-to-One Communication
  - + Direct API calls
  - + Point-to-point messaging
- One-to-Many Communication
  - + Event broadcasting
  - + Pub/Sub messaging

3.1.3. Các yếu tố ảnh hưởng đến việc lựa chọn pattern

- Performance requirements
- Data consistency needs
- System scalability
- Error handling requirements
- Development complexity

3.2. Synchronous Communication Patterns

3.2.1. REST API Pattern

- Request-Response model
- HTTP methods (GET, POST, PUT, DELETE)
- Stateless communication
- Order-Inventory check
- Payment processing
- Simple CRUD operations
- Ưu điểm:
  + Simple implementation
  + Immediate feedback
  + Standard protocol
- Nhược điểm:
  + High latency
  + Resource blocking
  + Tight coupling

3.3. Asynchronous Communication (one-to-one)

3.3.1. Message Queue Pattern

- Producer-Consumer model
- Message persistence
- Guaranteed delivery
- Long-running payment processing
- Background tasks
- Batch processing
- Ưu điểm:
  + Better resource utilization
  + Loose coupling
  + Reliable delivery
- Nhược điểm:
  + Eventual consistency
  + Complex workflow
  + Message ordering

3.4. Asynchronous Communication (one-to-many)

3.4.1. Pub/Sub Pattern

- Publisher-Subscriber model
- Topic-based routing
- Event-driven architecture
- Order notifications
- User activity logging
- Real-time updates
- Ưu điểm:
  + High scalability
  + Decoupled services
  + Efficient broadcasting
- Nhược điểm:
  + Message ordering
  + Delivery guarantees
  + Complex setup

3.5. So sánh và đánh giá các patterns

3.5.1. Performance comparison

- Latency metrics
- Throughput capabilities
- Resource utilization

3.5.2. Error handling capabilities

- Retry mechanisms
- Error propagation
- Recovery strategies

3.5.3. Scalability considerations

- Horizontal scaling
- Load balancing
- Service discovery

Chương 4.
Triển khai thử nghiệm

4.1. Mô tả bài toán và yêu cầu

4.1.1. Hệ thống thử nghiệm

- E-commerce order processing system
- Microservices architecture
- Multiple communication patterns

4.1.2. Yêu cầu hệ thống

- Order-Inventory management
- Payment processing
- Order notifications
- User activity logging

4.2. Cài đặt và triển khai

4.2.1. Thiết kế kiến trúc

- Service boundaries
- Communication patterns
- Data flow
- Error handling

4.2.2. Lựa chọn công nghệ

- Spring Boot for services
- RabbitMQ for message queue
- Kafka for pub/sub
- Docker for containerization

4.2.3. Chi tiết triển khai

- REST API implementation
- Message Queue implementation
- Pub/Sub implementation
- Activity tracking system

4.3. Kết quả triển khai

4.3.1. Hiệu suất hệ thống

- Latency metrics
- Throughput results
- Resource utilization
- Error rates

4.3.2. Độ tin cậy

- Processing times
- Success rates
- Error handling
- Recovery times

4.3.3. Khả năng mở rộng

- Broadcast performance
- Service failure impact
- System stability
- Resource efficiency

4.3.4. Thiết lập hạ tầng

- Docker containers
- Service discovery
- Message brokers
- Monitoring tools

4.4. Đánh giá hiệu năng

4.4.1. Phương pháp đánh giá

- Test scenarios
- Performance metrics
- Testing tools
- Data collection

4.4.2. Phân tích so sánh

- Synchronous vs Asynchronous
- One-to-One vs One-to-Many
- Resource utilization
- Error handling

Chương 5.
Đánh giá và thảo luận

5.1. Phương pháp và tiêu chí đánh giá

5.1.1. Phương pháp đánh giá

- Phân tích định tính
- Phân tích định lượng
- So sánh với các nghiên cứu liên quan

5.1.2. Tiêu chí đánh giá

- Hiệu suất hệ thống
- Độ tin cậy
- Khả năng mở rộng
- Độ phức tạp triển khai
- Chi phí vận hành

5.2. Kết quả đánh giá

5.2.1. Đánh giá hiệu suất

- So sánh các phương thức giao tiếp
- Phân tích độ trễ
- Đánh giá thông lượng
- Hiệu quả sử dụng tài nguyên

5.2.2. Đánh giá độ tin cậy

- Khả năng chịu lỗi
- Cơ chế phục hồi
- Đảm bảo tính nhất quán
- Xử lý sự cố

5.2.3. Đánh giá khả năng mở rộng

- Khả năng mở rộng ngang
- Cân bằng tải
- Phát hiện dịch vụ
- Quản lý phiên bản

5.3. Thảo luận

5.3.1. Ưu điểm và hạn chế

- Điểm mạnh của các phương pháp
- Những hạn chế cần khắc phục
- Các vấn đề cần nghiên cứu thêm

5.3.2. Kiến nghị và hướng phát triển

- Cải tiến phương pháp
- Mở rộng phạm vi nghiên cứu
- Ứng dụng thực tế
- Hướng nghiên cứu tương lai

Tài liệu tham khảo

Tài liệu tham khảo

[1] Tài liệu tham khảo 1

[2] Tài liệu tham khảo 2