

BAN CƠ YẾU CHÍNH PHỦ  
HỌC VIỆN KỸ THUẬT MẬT MÃ



ĐỀ CƯƠNG  
CHUYÊN ĐỀ AN TOÀN HỆ THỐNG THÔNG TIN  
Nghiên cứu giải pháp đảm bảo an toàn cho Microservices trên Kubernetes

Ngành: An toàn thông tin

*Sinh viên thực hiện:*

**Phương Văn Sơn**

Mã sinh viên: AT160258

**Lê Huy Dũng**

Mã sinh viên: AT160211

*Người hướng dẫn:*

**TS. Nguyễn Mạnh Thắng**

Khoa An toàn thông tin - Học viện Kỹ thuật mật mã

Hà Nội, 2022

# Lời mở đầu

Nhiều năm trước, hầu hết các ứng dụng phần mềm đều được xây dựng với kiến trúc monolith hay còn gọi là kiến trúc 1 khối là mẫu thiết kế được dùng nhiều nhất trong giới lập trình web hiện nay bởi tính đơn giản của nó khi phát triển và khi triển khai. Các ứng dụng này chạy dưới dạng một tiến trình đơn lẻ hoặc số lượng nhỏ các tiến trình trên một số ít máy chủ. Chúng có khả năng cập nhật và nâng cấp chậm và yêu cầu nâng cấp thường xuyên. Trong trường hợp có sự cố như lỗi phần cứng hệ thống phần mềm này sẽ phải được di chuyển một cách thủ công sang các máy chủ còn hoạt động tốt.

Ngày nay các ứng dụng được xây dựng với kiến trúc lớn và phức tạp đang dần được chia thành các thành phần nhỏ hơn, có khả năng hoạt động độc lập được gọi là microservices. Vì các Microservices tách biệt với nhau nên chúng có thể được phát triển, triển khai hay cập nhật và mở rộng quy mô một cách riêng lẻ. Nhờ khả năng này cho phép ta thay đổi các thành phần nhanh chóng và thường xuyên khi cần thiết để theo kịp với các yêu cầu thay đổi nhanh chóng thời nay.

Nhưng với số lượng lớn các thành phần cũng như cơ sở dữ liệu việc cấu hình, quản lý và giữ hệ thống hoạt động trơn tru ngày càng trở nên khó khăn đặc biệt trong việc tối ưu hiệu quả sử dụng tài nguyên. Kubernetes ra đời để đáp ứng nhu cầu tự động hoá như lập lịch tự động, cấu hình tự động hay giám sát và xử lý lỗi.

Kubernetes cung cấp cho các nhà phát triển khả năng triển khai các ứng dụng một cách thường xuyên mà không cần thông qua nhóm vận hành. Không chỉ dừng lại ở đó Kubernetes cũng giúp nhóm vận hành tự động theo dõi và khắc phục sự cố.

Đi cùng với sự phát triển lớn mạnh của kiến trúc Microservices cũng như Kubernetes đó là nhu cầu về việc đảm bảo tính an toàn cho các hệ thống này. Trong bài báo cáo này chúng em sẽ giới thiệu về giải pháp đảm bảo an toàn cho Microservices bằng Istio Service Mesh

# Lời cảm ơn

Nhóm chúng em xin chân thành cảm ơn các thầy cô trường Học viện Kỹ thuật Mật mã nói chung, quý thầy cô của khoa An toàn thông tin nói riêng đã tận tình dạy bảo, truyền đạt kiến thức cho chúng em trong suốt quá trình học.

Kính gửi đến Thầy Nguyễn Mạnh Thắng lời cảm ơn chân thành và sâu sắc nhất, cảm ơn thầy đã tận tình theo sát, chỉ bảo và hướng dẫn cho nhóm em trong quá trình thực hiện đề tài này. Thầy không chỉ hướng dẫn chúng em những kiến thức chuyên ngành, mà còn giúp chúng em học thêm những kỹ năng mềm, tinh thần học hỏi, thái độ khi làm việc nhóm.

Trong quá trình tìm hiểu nhóm chúng em xin cảm ơn các bạn sinh viên đã góp ý, giúp đỡ và hỗ trợ nhóm em rất nhiều trong quá trình tìm hiểu và làm đề tài.

Do kiến thức còn nhiều hạn chế nên không thể tránh khỏi những thiếu sót trong quá trình làm đề tài. Chúng em rất mong nhận được sự đóng góp ý kiến của quý thầy cô để đề tài của chúng em đạt được kết quả tốt hơn.

**Chúng em xin chân thành cảm ơn!**

# Mục lục

Lời mở đầu	ii
Lời cảm ơn	iii
Danh sách hình vẽ	v
<b>1 Giới thiệu về công nghệ Container và kiến trúc Microservices</b>	<b>1</b>
1.1 Giới thiệu về công nghệ Container . . . . .	1
1.1.1 Hiểu về công nghệ Container . . . . .	1
1.1.2 Container và Máy ảo . . . . .	2
1.1.3 Đặc điểm kỹ thuật của Container . . . . .	3
1.1.4 Ứng dụng của Container trong thực tế . . . . .	3
1.1.5 Về Containerization . . . . .	4
1.1.6 Những thách thức trong việc sử dụng Container . . . . .	5
1.2 Giới thiệu về kiến trúc Microservices . . . . .	6
1.2.1 Khái niệm Microservices . . . . .	6
1.2.2 Những đặc điểm của Microservices . . . . .	7
1.2.3 Ưu điểm của Microservices . . . . .	7
1.2.4 Các nhược điểm của kiến trúc Microservice . . . . .	8
1.2.5 Những yêu cầu bắt buộc khi phát triển phần mềm theo kiến trúc Microservice . . . . .	8
1.3 Giới thiệu về Kubernetes . . . . .	9
1.3.1 Kiến trúc của Kubernetes . . . . .	9
1.3.2 Tổng quan về Pod . . . . .	10
1.3.3 Vòng đời của Pod . . . . .	11
1.3.4 Quản lý pod bằng Workload trên Kubernetes . . . . .	11
1.4 Một số vấn đề bảo mật kiến trúc Microservices . . . . .	12
1.4.1 Càng nhiều microservices thì nguy cơ bị tấn công càng cao . . . . .	12
1.4.2 Kiểm tra bảo mật phân tán có thể dẫn đến hiệu suất giảm . . . . .	12
1.4.3 Sự phức tạp khi triển khai xác thực khởi động giữa các microservices . . . . .	13
1.4.4 Khó theo dõi các request của các microservice . . . . .	13
1.4.5 Tính bất biến của container làm việc xác thực và chính sách kiểm soát truy cập của các dịch vụ khó khăn . . . . .	14
1.4.6 Kiến trúc đa ngôn ngữ đòi hỏi các developer phải có thêm nhiều kiến thức bảo mật hơn . . . . .	14
<b>2 Tổng quan về Istio</b>	<b>15</b>
2.1 Tổng quan về Istio . . . . .	15
2.1.1 Tổng quan về Service Mesh . . . . .	15
2.1.2 Kiến trúc của Istio . . . . .	19

2.1.3	Tổng quan về Envoy Proxy . . . . .	21
2.2	Quản lý mạng giữa các Microservices với Istio . . . . .	28
2.2.1	Tổng quan về Istio Ingress Gateway . . . . .	28
2.2.2	Định tuyến trong Istio . . . . .	34
2.2.3	Giải quyết các vấn đề về mạng trong Microservices . . . . .	38
2.3	Giám sát các Microservices với Istio . . . . .	41
2.3.1	Một số Metrics quan trọng của Istio . . . . .	41
2.3.2	Giám sát các lưu lượng mạng qua Jaeger và Kiali . . . . .	41
2.4	Bảo mật các Microservices bằng Istio . . . . .	41
2.4.1	Xác thực giữa các Microservices với Istio . . . . .	41
2.4.2	Phân quyền cho các Microservices với Istio . . . . .	41
<b>3</b>	<b>Triển khai Istio trên Kubernetes</b>	<b>42</b>
3.1	Mô hình triển khai . . . . .	42
3.2	Kịch bản triển khai . . . . .	42
3.3	Thực nghiệm . . . . .	42
3.4	Kết luận . . . . .	42

# Danh sách hình vẽ

1.1	Kiến trúc của Kubernetes . . . . .	9
2.1	Ứng dụng kiến trúc Microservices trong Security . . . . .	16
2.2	Các vấn đề gặp phải . . . . .	16
2.3	Kiến trúc của Istio . . . . .	19
2.4	Luồng dữ liệu sau khi cài đặt Istio . . . . .	21
2.5	Ví dụ về lưu lượng trong Istio . . . . .	21
2.6	Proxy là một trung gian bổ sung các chức năng trong lưu lượng . . . . .	22
2.7	Một proxy có thể ẩn cấu trúc liên kết phụ trợ khỏi máy khách và thực hiện các thuật toán để phân phối lưu lượng truy cập một cách công bằng (cân bằng tải). . . . .	23
2.8	Một yêu cầu đến từ một hệ thống cấp dưới thông qua các listeners, đi qua quá trình định tuyến và kết thúc bằng một cụm gửi nó đến một dịch vụ cấp trên. . . . .	24
2.9	Một số số liệu thống kê mà Envoy proxy thu thập . . . . .	26
2.10	Istio Gateway đóng vai trò là điểm vào mạng và sử dụng proxy Envoy để thực hiện định tuyến và cân bằng tải. . . . .	29
2.11	Cấu trúc một control plane và các thành phần phụ trợ . . . . .	29
2.12	Cầu hình cài nguyên của gateway . . . . .	30
2.13	Ví dụ về VirtualService . . . . .	31
2.14	Luồng lưu lượng từ máy khách bên ngoài lưới/cụm dịch vụ đến các dịch vụ bên trong lưới dịch vụ thông qua gateway . . . . .	34
2.15	Gọi danh mục dịch vụ trực tiếp thông qua gateway . . . . .	36
2.16	Định tuyến cho các yêu cầu có nội dung nhất định . . . . .	38
2.17	Định tuyến cho các yêu cầu có nội dung cụ thể trong biểu đồ lời gọi . . . . .	38
2.18	Dịch vụ A, dịch vụ gọi B, có thể đang gặp sự cố. . . . .	39
2.19	Proxy web đơn giản dùng các điểm đầu cuối phụ trợ. . . . .	40
2.20	Ưu tiên các dịch vụ cục bộ . . . . .	41

# Chương 1

## Giới thiệu về công nghệ Container và kiến trúc Microservices

### 1.1 Giới thiệu về công nghệ Container

Xây dựng phần mềm theo xu hướng Cloud Native đang phát triển rất nhanh. Trong đó, công nghệ container đóng 1 vai trò rất quan trọng để theo đuổi cách triển khai này.

Công nghệ container, hay gọi đơn giản là container, là một phương pháp đóng gói ứng dụng để ứng dụng có thể chạy với các phụ thuộc của mình (gồm source code và library, runtime, framework...) một cách độc lập, tách biệt với các chương trình khác. Các nhà cung cấp dịch vụ đám mây lớn hiện nay đã cung cấp các dịch vụ dành cho việc quản lý container để hỗ trợ việc xây dựng ứng dụng sử dụng công nghệ container.

#### 1.1.1 Hiểu về công nghệ Container

Container được đặt tên theo thuật ngữ container của ngành vận tải biển vì có cùng chung ý tưởng với nhau. Thay vì chỉ vận chuyển từng sản phẩm, hàng hóa được đặt vào các thùng hàng bằng thép, được thiết kế theo các tiêu chuẩn phù hợp về kích thước và trọng tải để có thể cấu lên bên tàu và lắp vào con tàu. Như vậy, bằng cách tiêu chuẩn hóa quy trình và nhóm các thành phần liên quan lại với nhau, từng container sẽ được chuyển đi riêng lẻ và sẽ tốn ít chi phí hơn để làm theo cách này.

Trong công nghệ, tình huống cũng khá tương tự. Một chương trình chạy hoàn hảo trên một máy, nhưng khi chuyển sang máy khác thì lại không hoạt động được. Điều này có thể xảy ra khi di chuyển phần mềm từ PC của developer sang test server hoặc từ server vật lý sang cloud server. Các vấn đề phát sinh khi di chuyển phần mềm là do sự khác biệt giữa các môi trường máy tính, chẳng hạn như OS, thư viện SSL, storage, bảo mật và cấu trúc mạng trên các máy khác nhau sẽ khác nhau..

Container giải quyết vấn đề trên bằng cách tạo ra một môi trường bị cô lập (isolated) chứa mọi thứ mà phần mềm cần để có thể chạy được mà không bị các yếu tố liên quan đến môi trường hệ thống làm ảnh hưởng tới cũng như không làm ảnh hưởng tới các phần còn lại của hệ thống.

Giống như việc toàn bộ 1 container sẽ được nhấc lên tàu hoặc xe tải để vận chuyển, công nghệ container cũng như vậy. Container không chỉ chứa phần mềm mà còn chứa các phần phụ thuộc bao gồm các library, binary và file cấu hình cùng với nhau và chúng được

di chuyển như một bộ phận, để tránh sự không tương thích và sự cố. Các container giúp việc triển khai phần mềm lên máy chủ thuận lợi hơn.

Như vậy, những container có tác dụng giúp cho một ứng dụng có thể vận hành một cách nhất quán và đáng tin cậy. Bất kể là môi trường hệ điều hành hay cơ sở hạ tầng nào. Các container thực hiện điều này bằng cách đóng gói mọi thứ mà một dịch vụ cần để có thể chạy được (những thứ như code, runtime, các tool, thư viện và cài đặt), tạo ra một package linh động, độc lập, có khả năng thực thi được.

### 1.1.2 Container và Máy ảo

Trước khi các container dần được ưa chuộng, máy ảo VM là một phương pháp sử dụng phổ biến. Ở phương pháp này, một máy chủ vật lý có thể được sử dụng cho nhiều ứng dụng thông qua công nghệ ảo hóa, còn được gọi là virtual machine, trong đó mỗi máy ảo chứa toàn bộ hệ điều hành, cũng như các ứng dụng cần thiết để chạy.

#### Về cấu trúc

VM, hay Virtual Machine/máy ảo là một phiên bản tóm tắt của máy tính, từ hệ điều hành cho đến bộ nhớ và lưu trữ. Image dùng để tạo một VM có thể tương tự hệ điều hành để cài đặt ứng dụng lên hoặc có sẵn tất cả các ứng dụng bạn cần, như web server và database, thậm chí cả chính ứng dụng của bạn. Mỗi VM sẽ hoạt động độc lập hoàn toàn với máy chủ mà VM chạy trên đó, cũng như độc lập với bất kỳ VM nào khác trên máy chủ đó.

Trong khi đó, container sẽ chạy một phần của máy hiện có, chia sẻ kernel của máy chủ đó với bất kỳ container nào khác đang chạy trên hệ thống. Chỉ chứa vừa đủ hệ điều hành và bất kỳ thư viện hỗ trợ nào cần thiết để chạy code. Container được xây dựng từ những image bao gồm mọi thứ nó cần - và không có gì khác (trong TH lý tưởng nhất).

#### Nhu cầu tài nguyên

Do cấu trúc khác nhau nên nhu cầu để chạy VM và container có thể khác nhau đáng kể. Bởi vì về cơ bản VM tương đương với toàn bộ một máy tính, nên đương nhiên sẽ cần nhiều tài nguyên hơn là một container, trong khi container chỉ cần đến một phần nhỏ nhất của hệ điều hành. Tóm lại, việc mở rộng các container sẽ ít tốn tài nguyên, thời gian, công sức hơn và có thể “xếp” nhiều container hơn trên một máy chủ duy nhất.

Tuy nhiên, cần hết sức lưu ý là vì nhiều dịch vụ có thể “chia sẻ” tài nguyên của một máy ảo duy nhất, có thể có những trường hợp phức tạp trong đó cần thiết phải mở rộng nhiều container để thay thế một máy ảo duy nhất. Điều này dẫn đến việc kiểm soát tài nguyên không còn nhiều ý nghĩa. Ví dụ: nếu bạn tách một máy ảo đơn lẻ thành 50 dịch vụ khác nhau, thì đó là 50 bản sao một phần của hệ điều hành so với một bản sao đầy đủ. Vì vậy, điều quan trọng là cần hiểu chính xác để lựa chọn đúng.

Vậy Container và máy ảo thì đều là những “package”. Mỗi container là một package bao gồm ứng dụng của bạn và mọi thứ nó cần để có thể chạy, ngoại trừ hệ điều hành. Máy ảo là một package ứng dụng và mọi thứ nó cần để chạy, bao gồm cả hệ điều hành của nó.



Bạn có thể chạy nhiều container trên một hệ điều hành. Và bạn có thể chạy nhiều máy ảo trên cùng một máy chủ vật lý. Bạn thậm chí có thể chạy container trên máy ảo.

Một lợi thế quan trọng của container so với máy ảo đó là chúng không bao gồm hệ điều hành, container cần ít tài nguyên hệ thống và ít chi phí hơn. Chúng cũng có xu hướng khởi động / tắt nhanh hơn và tính di động cao trong nhiều môi trường khác nhau. Nhưng chúng vẫn sử dụng công suất cơ sở hạ tầng khi không sử dụng, điều này có thể làm tăng chi phí không cần thiết.

### 1.1.3 Đặc điểm kỹ thuật của Container

Mô hình kiến trúc của container bao gồm các thành phần chính là Server (máy chủ vật lý hoặc máy ảo), host OS (hệ điều hành cài đặt trên server) và các container.

Mỗi một ứng dụng (App A và App B) sẽ có những sự phụ thuộc riêng của nó bao gồm cả về phần mềm (các dịch vụ hay thư viện) lẫn cả về phần cứng (CPU, bộ nhớ, lưu trữ).

Các ứng dụng này sẽ được Container Engine, một công cụ ảo hóa tinh gọn, được cài đặt trên host OS, nó sẽ cô lập sự phụ thuộc của các ứng dụng khác nhau bằng cách đóng gói chúng thành các container. Các tiến trình (process) trong một container bị cô lập với các tiến trình của các container khác trong cùng hệ thống tuy nhiên tất cả các container này đều chia sẻ kernel của host OS (dùng chung host OS).

Với mô hình trên, sự phụ thuộc của ứng dụng vào tầng OS cũng như cơ sở hạ tầng được loại bỏ giúp việc triển khai phương pháp “deploy anywhere” (triển khai ở bất kỳ nơi đâu) của container được hiệu quả hơn. Thêm vào đó, do chia sẻ host OS nên container có thể được tạo gần như một cách tức thì, giúp việc scale-up và scale-down theo nhu cầu được thực hiện một cách nhanh chóng.

### 1.1.4 Ứng dụng của Container trong thực tế

Các container đại diện cho tương lai của máy tính cùng với các công nghệ như DevOps, cloud native, AI, machine learning. Các trường hợp sử dụng phổ biến bao gồm:

- Hiện đại hóa các ứng dụng hiện có trên đám mây
- Tạo các ứng dụng mới tối đa hóa lợi ích của container
- Cô lập, triển khai, mở rộng quy mô và hỗ trợ microservices và các ứng dụng phân tán
- Tăng cường hiệu quả DevOps, hiệu quả thông qua việc build/test/triển khai được sắp xếp một cách hợp lý
- Cung cấp cho nhà phát triển một môi trường sản xuất nhất quán, tách biệt khỏi các ứng dụng và quy trình khác
- Đơn giản hóa và tăng tốc các chức năng có tính lặp đi lặp lại

Tạo điều kiện thuận lợi cho các môi trường máy tính kết hợp với multi-cloud, vì các container có thể chạy nhất quán ở bất kỳ đâu.

### 1.1.5 Về Containerization

Containerization là hành động tạo một container, bao gồm việc chỉ lấy ra ứng dụng hay dịch vụ mà bạn cần chạy, cùng với các cấu hình và những phần phụ thuộc của nó, đồng thời rút nó ra khỏi hệ điều hành và cơ sở hạ tầng bên dưới. Sau đó, cho ra kết quả là container image có thể chạy trên bất kỳ nền tảng container nào.

Nhiều container có thể được chạy trên cùng một máy chủ và chia sẻ cùng một hệ điều hành với các container khác, mỗi container chạy các quy trình biệt lập trong không gian được bảo mật riêng của nó. Bởi vì các container chia sẻ base OS (hệ điều hành), do vậy kết quả là có thể chạy mỗi container bằng cách sử dụng một lượng tài nguyên rất ít, ít hơn đáng kể so với việc sử dụng số lượng máy ảo (VM) riêng biệt.

#### Những lợi ích của Container

##### Tốn rất ít dung lượng

Các container chia sẻ kernel của máy chủ lưu trữ, chúng chỉ chứa các thành phần thực sự cần thiết với hệ điều hành và thư viện. Đồng thời các container thường cũng chỉ giới hạn ở một chức năng duy nhất, nên có kích thước rất nhỏ. Nhờ vậy, việc xây dựng, triển khai cực kỳ nhanh chóng.

Bởi vì chúng được tách biệt khỏi lớp OS nên việc container chạy hiệu quả và nhẹ về tài nguyên hơn so với máy ảo cũng là điều dễ hiểu.

##### Các Container có tính linh hoạt

Vì container bao gồm có tất cả các cấu hình cần thiết và các thành phần phụ thuộc, do vậy bạn có thể viết một lần và di chuyển giữa các môi trường. Có một câu thần chú nổi tiếng đó là “Build once, run everywhere”.

- **Triển khai nhanh:**

Do kích thước nhỏ, các container có thể chỉ cần vài giây để khởi động, thậm chí là ít hơn, nên rất thích hợp cho các ứng dụng cần được đẩy lên và xuống liên tục, chẳng hạn như các ứng dụng “serverless”.

- **CI/CD:**

CI là tên viết tắt của Continuous Integration, theo nghĩa tiếng Việt là tích hợp liên tục. Quá trình hoạt động cho phép các thành viên trong một team liên tục lưu trữ những mã mới vào một kho nhất định. Nhờ vào số lượng dữ liệu này, CI sẽ tự động chạy test và kiểm tra độ chính xác. Cùng lúc đó cũng hỗ trợ phát triển phần mềm một cách nhanh chóng hơn bằng việc báo lỗi sai và đưa ra gợi ý giải quyết.

CD là tên viết tắt của Continuous Delivery, nghĩa là quá trình chuyển giao liên tục. Về cơ bản, CD cũng sở hữu những kỹ năng của CI, tuy nhiên sẽ phức tạp và nâng cao hơn một chút.

Trong khi CI chỉ chạy và kiểm tra những code đã có sẵn, CD thậm chí còn tự sửa code đã được build và test nếu phát hiện lỗi sai. Ngoài ra, nó cũng tự động thay đổi môi trường testing hoặc staging để nâng cao chất lượng kiểm tra.

Chính vì các container được thiết kế để có thể start và restart thường xuyên, nhờ vậy mà dễ dàng tiếp nhận các thay đổi tạo điều kiện vô cùng phù hợp để triển khai CI/CD.

- **Khả năng mở rộng:**

Do kích thước nhỏ của chúng, các container có thể dễ dàng lấy ra, mở rộng quy mô trong quá trình vận hành, tắt đi khi không sử dụng và nhanh chóng khởi động lại khi cần thiết.

- **Tiết kiệm chi phí:**

Thông qua việc giảm lượng nhu cầu về tài nguyên và mở rộng quy mô một cách thông minh, các container cung cấp một giải pháp linh hoạt, nhanh chóng và tiết kiệm chi phí.

### **Khả năng chịu lỗi cao**

Các nhóm phát triển phần mềm sử dụng bộ chứa để xây dựng các ứng dụng có khả năng chịu lỗi cao. Họ sử dụng nhiều bộ chứa để chạy vi dịch vụ trên đám mây. Bởi vì các vi dịch vụ trong bộ chứa hoạt động trong không gian người dùng riêng biệt, một bộ chứa bị lỗi riêng lẻ sẽ không ảnh hưởng đến các bộ chứa khác. Điều này làm tăng khả năng phục hồi và tính khả dụng của ứng dụng.

### **Quản lý ít cơ sở hạ tầng hơn**

Container buộc bạn phải nắm bắt được những gì mà bạn thực sự cần qua đó mang lại trải nghiệm tốt nhất cho khách hàng của bạn. Điều này giúp quản lý cơ sở hạ tầng tốt hơn vì đơn giản là có ít cơ sở hạ tầng hơn để quản lý.

### **Container tạo ra khả năng tập trung**

Các teams IT sẽ dành ít thời gian hơn cho các hệ điều hành và phần cứng, điều đó cho phép họ tập trung hơn vào các dự án quan trọng của doanh nghiệp.

### **Thúc đẩy sự phát triển**

Container cung cấp một môi trường ổn định, dễ dàng dự đoán được, nơi CPU/memory được tối ưu hóa và code thì được trừu tượng hóa từ cơ sở hạ tầng để có tính khả chuyển.

### **Tạo điều kiện cho những kiến trúc hiện đại**

Sử dụng container, các nhà phát triển có thể chia các ứng dụng thành các microservices, điều này có thể tăng tốc độ phát triển và khi được triển khai thì được mở rộng một cách riêng biệt.

## **1.1.6 Những thách thức trong việc sử dụng Container**

### **Container còn tương đối mới**

Kubernetes được phát hành lần đầu tiên vào năm 2014 và nhanh chóng được thị trường đón nhận. Việc trở thành một “hot tech” có thể gây khó khăn trong việc tìm kiếm những người có kinh nghiệm và biết cách làm việc với nhiều môi trường container.

### **Không phải dịch vụ nào cũng được hỗ trợ Container hóa**

Nếu ứng dụng của bạn đưa vào các dịch vụ không hỗ trợ container. Bạn có thể cần phải đầu tư nhiều để chuyển đổi nó thành một giải pháp container.

## Container yêu cầu nhiều thay đổi về quy trình và kỹ năng

Container có thể đẩy nhanh quá trình chuyển đổi của bạn sang kiểu phát triển agile hay efficient, nhưng điều này cũng đồng nghĩa với việc tạo ra một thay đổi lớn đối với các quy trình hiện có như quy trình phát triển, triển khai, review và giám sát. Dẫn đến việc các team mà tổ chức hiện có cần phải được điều chỉnh và đào tạo lại.

- **Có thể yêu cầu kết nối mạng phức tạp:**

Thường thường các chức năng sẽ được chia thành nhiều container và cần phải giao tiếp với nhau. Việc số lượng rất nhiều các container phải giao tiếp với nhau có thể phức tạp. Một số hệ thống điều phối như Kubernetes có các multi-container pods giúp việc trao đổi dễ dàng hơn một chút, nhưng được cho là vẫn phức tạp hơn so với sử dụng máy ảo. Thực tế thì mô hình mạng L3 trong Kubernetes đơn giản hơn nhiều so với mô hình L2 trong hạ tầng máy ảo OpenStack. Vì vậy, vấn đề nằm ở chỗ cần xác định được việc giao tiếp xảy ra giữa các chức năng hay giữa các máy ảo

- **Có thể cần thao tác nhiều hơn so với máy ảo:**

Nếu sử dụng container, bạn sẽ phân tách ứng dụng thành các dịch vụ thành phần khác nhau, mặc dù việc này có ích lợi nhưng lại không cần thiết khi sử dụng VM.

- **Công nghệ đang phát triển với tốc độ chóng mặt:**

Điều này không chỉ dành riêng cho container, nhưng về bản chất thì nhịp độ nhanh của công nghệ container có nghĩa là bạn cần mọi người (hoặc đối tác) có mặt để đưa ra quyết định đúng đắn, giảm rủi ro và đảm bảo việc triển khai không bị cản trở bởi tính trì trệ của công ty.

- **Container không phải là một viên đạn thần kỳ:**

Đọc lướt qua một loạt các lợi ích thì trông có vẻ container là một thứ lý tưởng, nhưng hãy cẩn thận vì bất kỳ quá trình chuyển đổi nào cũng cần phải suy nghĩ nghiêm túc. Bạn phải biết mình đang làm gì, những gì sẽ mang lại hiệu quả và ngược lại. Hoặc tìm một ai có thể giúp bạn vượt trong sự chuyển đổi đó.

## 1.2 Giới thiệu về kiến trúc Microservices

Một vài năm trở lại đây, khái niệm kiến trúc Microservices hiện là chủ đề rất hot trong cộng đồng lập trình viên. Thật không khó để có thể tìm thấy một bài viết, một bản báo cáo hay một bài thuyết trình về chủ đề này. Vậy Microservices là gì? Ưu điểm và nhược điểm của kiến trúc Microservices ra sao?

### 1.2.1 Khái niệm Microservices

Trong tiếng anh, micro có nghĩa là nhỏ, vi mô. Vậy Microservice, như tên của nó, đó chính là chia một khối phần mềm thành các service nhỏ hơn, có thể triển khai trên các server khác nhau. Mỗi service sẽ xử lý từng phần công việc và được kết nối với nhau thông qua các giao thức khác nhau, như http, SOA, socket, Message queue (Active MQ, Kafka)... để truyền tải dữ liệu.

Trước khi Microservices xuất hiện, các ứng dụng thường phát triển theo mô hình Monolithic architecture (Kiến trúc một khối). Có nghĩa là tất cả các module (view, business, database) đều được gộp trong một project, một ứng dụng được phát triển theo mô hình kiến trúc một khối thường được phân chia làm nhiều module. Nhưng khi được đóng gói và cài đặt sẽ thành một khối (monolithic). Lợi ích của mô hình kiến trúc một khối đó là dễ dàng phát triển và triển khai. Nhưng bên cạnh đó nó cũng có nhiều hạn chế ví dụ như khó khăn trong việc bảo trì, tính linh hoạt và khả năng mở rộng kém, đặc biệt với những ứng dụng doanh nghiệp có quy mô lớn. Đó chính là lí do ra đời của kiến trúc Microservices.

### 1.2.2 Những đặc điểm của Microservices

**Decoupling:** Các service trong một hệ thống phần lớn được tách rời. Vì vậy, toàn bộ ứng dụng có thể dễ dàng được xây dựng, thay đổi và thu nhỏ.

**Componentization:** Microservices được coi là các thành phần độc lập có thể dễ dàng thay thế và nâng cấp.

**Business Capabilities:** Mỗi một thành phần trong kiến trúc microservice rất đơn giản và tập trung vào một nhiệm vụ duy nhất.

**Autonomy:** Các lập trình viên hay các nhóm có thể làm việc độc lập với nhau trong quá trình phát triển.

**Continous Delivery:** Cho phép phát hành phần mềm thường xuyên, liên tục.

**Decentralized Governance:** Không có mẫu chuẩn hóa hoặc bất kỳ mẫu công nghệ nào. Được tự do lựa chọn các công cụ hữu ích tốt nhất để có thể giải quyết vấn đề.

**Agility:** Microservice hỗ trợ phát triển theo mô hình Agile.

### 1.2.3 Ưu điểm của Microservices

Kiến trúc Microservices được sinh ra để khắc phục những hạn chế của kiến trúc một khối.

- Hoạt động độc lập, linh hoạt, có tính chuyên biệt cao: Do không bị ràng buộc bởi những yêu cầu chung, nên mỗi service nhỏ có thể tự do lựa chọn công nghệ, nền tảng phù hợp. Tất cả các service có thể được phát triển dễ dàng dựa trên chức năng cá nhân của từng service. Có thể chia nhỏ để phát triển độc lập.
- Nâng cao khả năng xử lý lỗi: Với mô hình này, một service bất kỳ nào gặp lỗi sẽ không gây ra ảnh hưởng đối với những bộ phận còn lại. Việc khắc phục lỗi trên quy mô hẹp cũng sẽ được tiến hành một cách dễ dàng, khi một service của ứng dụng không hoạt động, hệ thống vẫn tiếp tục hoạt động.
- Independent Deployment: Có thể được triển khai riêng lẻ trong bất kỳ ứng dụng nào.
- Mixed Technology Stack: Các ngôn ngữ và công nghệ khác nhau có thể được sử dụng để xây dựng các service khác nhau của cùng một ứng dụng.

- Thuận tiện trong nâng cấp, mở rộng: Tương tự như trường hợp xử lý lỗi, việc nâng cấp, bảo trì service hoàn toàn độc lập sẽ không làm gián đoạn quá trình vận hành của cả phần mềm. Nhờ vậy, những phiên bản mới có thể được cập nhật thường xuyên.
- Đơn giản hóa trong quản lý và kiểm thử: Với từng service nhỏ, các bước quản lý, tính toán và kiểm soát, xử lý lỗi sẽ trở nên đơn giản và nhanh chóng hơn so với cả phần mềm.

Kiến trúc Microservices giúp đơn giản hóa hệ thống, chia nhỏ hệ thống ra làm nhiều service nhỏ lẻ dễ dàng quản lý và triển khai từng phần so với kiến trúc nguyên khối. Phân tách rõ ràng giữa các service nhỏ. Cho phép việc mỗi service được phát triển độc lập. Cũng như cho phép lập trình viên có thể tự do chọn lựa technology stack cho mỗi service mình phát triển. mỗi service có thể được triển khai một cách độc lập (VD: Mỗi service có thể được đóng gói vào một docker container độc lập, giúp giảm tối đa thời gian deploy). Nó cũng cho phép mỗi service có thể được scale một cách độc lập với nhau. Việc scale có thể được thực hiện dễ dàng bằng cách tăng số instance cho mỗi service rồi phân tải bằng load balancer.

#### 1.2.4 Các nhược điểm của kiến trúc Microservice

Kiến trúc Microservices đang là một xu hướng, nhưng nó cũng có nhược điểm của nó. Microservice khuyến khích làm nhỏ gọn các service, nhưng khi chia nhỏ sẽ dẫn đến những thứ vụn vặt, khó kiểm soát. Hơn nữa chính từ đặc tính phân tán khiến cho các lập trình viên phải lựa chọn cách thức giao tiếp phù hợp khi xử lý request giữa các service. Trong trường hợp dự án quá lớn, số lượng service nhiều, chia nhỏ rời rạc, thiếu tính liên kết. Cùng với cách thức liên kết thông tin giữa qua môi trường mạng, việc trao đổi giữa các service càng trở nên khó khăn. Đôi khi, các lỗi kết nối cũng có thể xảy ra khiến việc trao đổi này bị gián đoạn.

Việc liên tục di chuyển qua các database khác nhau sẽ khiến dữ liệu bị đảo lộn, không còn nguyên vẹn, thậm chí phải đối mặt với nguy cơ an ninh, bị đánh cắp.

Hơn nữa việc quản lý nhiều database, và transaction giữa các service trong một hệ thống phân tán cũng là một khó khăn không nhỏ. Hay khi thực hiện test một service, bạn cũng cần test các service có liên quan. Gây khó khăn trong quá trình mở rộng, phát triển. Khi phần mềm được phát triển với quy mô lớn hơn, số lượng service cũng trở nên nhiều hơn. Các lập trình viên không chỉ mất thời gian tính toán chính xác kích thước của từng service, mà còn gặp khó khăn khi sử dụng những công cụ hỗ trợ tự động mã nguồn mở bên ngoài khác.

Triển khai microservice cũng sẽ phức tạp hơn so với ứng dụng kiến trúc một khối, cần sự phối hợp giữa nhiều service, điều này không đơn giản như việc triển khai WAR trong một ứng dụng kiến trúc một khối.

#### 1.2.5 Những yêu cầu bắt buộc khi phát triển phần mềm theo kiến trúc Microservice

Để phát triển một phần mềm theo mô hình kiến trúc Microservice, lập trình viên cần đảm bảo một số yếu tố chính như sau:

- Xây dựng hệ cơ sở dữ liệu (database) độc lập.
- Xác định kích thước service phù hợp.
- Đề ra vai trò, chức năng cụ thể, riêng biệt cho từng service.

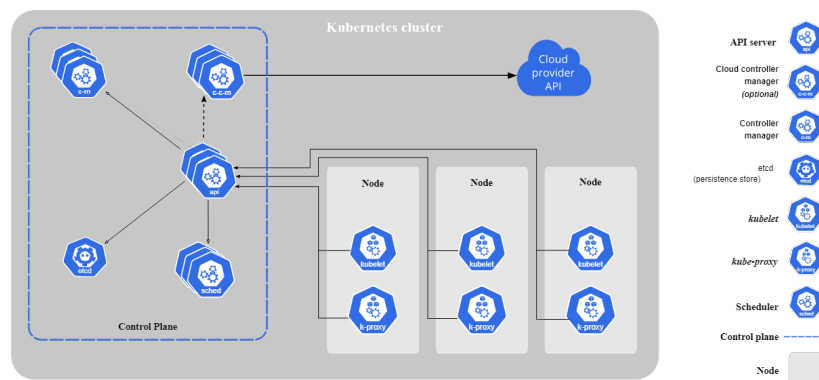
Việc phát triển một phần mềm theo mô hình kiến trúc Microservice chưa bao giờ là điều đơn giản.

## 1.3 Giới thiệu về Kubernetes

Kubernetes là một nền tảng nguồn mở quản lý các ứng dụng được đóng gói và các service, giúp thuận lợi trong việc cấu hình và tự động hoá việc triển khai ứng dụng.

Tên gọi Kubernetes có nguồn gốc từ tiếng Hy Lạp, có ý nghĩa là người lái tàu hoặc hoa tiêu. Google mở mã nguồn Kubernetes từ năm 2014. Kubernetes xây dựng dựa trên một thập kỷ rưỡi kinh nghiệm mà Google có được với việc vận hành một khối lượng lớn workload trong thực tế, kết hợp với các ý tưởng và thực tiễn tốt nhất từ cộng đồng.

### 1.3.1 Kiến trúc của Kubernetes



Hình 1.1: Kiến trúc của Kubernetes

Các thành phần của cụm Kubernetes:

- Control Plane: Là trung tâm điều khiển các cụm và làm các cụm kubernetes hoạt động. Đây là nơi quản lý, lên kế hoạch, lập lịch và theo dõi các pod, các node.
- Worker Node: Là một máy ảo hoặc máy vật lý chạy Kubernetes. Đây là nơi container thực sự được triển khai để chạy các ứng dụng.

Các thành phần của Control Plane:

- Kubernetes API server: Nơi mà các quản trị viên và các thành phần khác của Control Plane giao tiếp với nhau.
- Scheduler: lập lịch cho các ứng dụng của bạn (chỉ định các workload ví dụ như pods được triển khai ở worker node nào)

- Controller Manager: thực hiện các chức năng cấp cụm, chẳng hạn như nhân bản các thành phần, theo dõi các node, xử lý các node lỗi,...
- etcd: một kho dữ liệu phân tán đáng tin cậy lưu trữ cấu hình của các node dưới dạng key value.

Các thành phần của Worker Node:

- kubelet: dùng để giao tiếp với API server và quản lý các container nằm trong node.
- kube-proxy: là một proxy chạy trên các node trong Kubernetes. kube-proxy duy trì các quy tắc mạng trên các nút. Các quy tắc mạng này cho phép các Pods trong cùng một node hoặc khác node có thể giao tiếp với nhau.
- container-runtime: có thể là Docker, rkt, containerd hoặc một loại container-runtime khác.

### 1.3.2 Tổng quan về Pod

Pod là đơn vị thực thi cơ bản của 1 ứng dụng Kubernetes, là đơn vị nhỏ nhất và đơn giản nhất trong mô hình object của Kubernetes. Một Pod đại diện cho các process (tiến trình) chạy trên cluster.

Pod đóng gói một container của ứng dụng (hoặc trong một số trường hợp là nhiều container), tài nguyên lưu trữ, một định danh network duy nhất (địa chỉ IP) cũng như các tùy chọn chi phối cách thức các container sẽ chạy. Một Pod đại diện cho một đơn vị triển khai: một instance (phiên bản) của một ứng dụng trong Kubernetes, có thể chứa một container hoặc vài container được liên kết chặt chẽ và chia sẻ tài nguyên.

Docker là container runtime phổ biến nhất được sử dụng cho Pod trong Kubernetes, tuy nhiên Pods cũng hỗ trợ các container runtime khác.

Pod trong Kubernetes cluster có thể được sử dụng theo hai cách chính:

- Pods chạy một container duy nhất. Mô hình “một-container-một-Pod” là trường hợp sử dụng phổ biến nhất của Kubernetes; trong trường hợp này, ta có thể nghĩ về một Pod như một trình bao bọc (đóng gói) xung quanh một container và Kubernetes quản lý các Pod thay vì quản lý trực tiếp các container.
- Pods chạy nhiều container cần tương tác với nhau. Một Pod có thể đóng gói một ứng dụng chứa nhiều container cùng hoạt động và được liên kết chặt chẽ với nhau cũng như cần chia sẻ tài nguyên (được gọi là co-located container). Các co-located container này có thể tạo thành một đơn vị dịch vụ gắn kết duy nhất nghĩa là 1 container phục vụ các file từ một volume chia sẻ cho public, trong khi đó, một “sidecar” container khác sẽ làm mới (refresh) hoặc cập nhật các file đó. Pod đóng gói các container và tài nguyên lưu trữ này lại với nhau như một thực thể có thể quản lý được.



### 1.3.3 Vòng đời của Pod

Giống như các container, Pod được coi là các thực thể tạm bợ, không bền vững (thay vì lâu bền). Các Pod được tạo, gán một ID (UID) duy nhất và được lên lịch cho các node mà chúng vẫn duy trì cho đến khi bị hủy bỏ (theo chính sách khởi động lại) hoặc xóa. Nếu một node chết, các Pod được lập lịch cho node đó sẽ được lập lịch để xóa sau một khoảng thời gian chờ.

Pod không tự chữa lành. Nếu một Pod được lên lịch cho một node sau đó không thành công, Pod đó sẽ bị xóa; tương tự như vậy, một Pod sẽ không tồn tại sau khi bị trục xuất do thiếu tài nguyên hoặc bảo trì Node. Kubernetes sử dụng phần trừu tượng cấp cao hơn, được gọi là bộ điều khiển, xử lý công việc quản lý các cá thể Pod tương đối dùng một lần.

Một Pod nhất định (như được xác định bởi UID) không bao giờ được "lên lịch" đến một nút khác; thay vào đó, Pod đó có thể được thay thế bằng một Pod mới, gần giống, thậm chí có cùng tên nếu muốn, nhưng với một UID khác.

Giai đoạn của Pod là một bản tóm tắt đơn giản về vị trí của Pod trong vòng đời của nó. Các giai đoạn của Pod:

- Pending: Pod đã được chấp nhận bởi hệ thống kubernetes, nhưng 1 hoặc vài container image chưa được tạo ra. Nó bao gồm thời gian trước khi được lập lịch cũng như thời gian download image trên mạng về (có thể mất một lúc)
- Running: Pod đã được đưa vào 1 node và tất cả các container đã được tạo ra. Ít nhất 1 container vẫn đang chạy hoặc đang ở trong quá trình bắt đầu hoặc khởi động lại.
- Succeeded: Tất cả các container trong pod đã kết thúc thành công và sẽ không được khởi động lại nữa.
- Failed: Tất cả các container trong pod đã kết thúc và ít nhất 1 container đã thất bại trong quá trình kết thúc. Có nghĩa là container đã exited (thoát) với trạng thái non-zero hoặc đã bị kết thúc bởi hệ thống.
- Unknown: Vì một số lý do, trạng thái của pod không thể xác định được, thường là do bị lỗi trong việc giao tiếp với host của pod.

### 1.3.4 Quản lý pod bằng Workload trên Kubernetes

Workload là một ứng dụng chạy trên Kubernetes. Cho dù workload là một thành phần đơn lẻ hay nhiều thành phần hoạt động cùng nhau, trên Kubernetes, chúng chạy bên trong một tập hợp các pod. Trong Kubernetes, Pod đại diện cho một tập hợp các container đang chạy trên cụm của bạn.

Các Pod có một vòng đời xác định. Ví dụ: khi một Pod đang chạy trong cụm thì một lỗi nghiêm trọng trên node nơi pod đó đang chạy có nghĩa là tất cả các pod trên node đó đều bị lỗi. Kubernetes coi mức độ thất bại đó là cuối cùng: bạn sẽ cần tạo một Pod mới để khôi phục, ngay cả khi node sau đó trở lại bình thường.

Tuy nhiên, để làm cho công việc dễ dàng hơn, chúng ta không cần phải quản lý trực tiếp từng Pod. Thay vào đó, bạn có thể sử dụng các workload để quản lý một nhóm các Pod. Các tài nguyên này định cấu hình bộ điều khiển để đảm bảo số lượng phù hợp của loại Pod phù hợp đang chạy, phù hợp với trạng thái chúng ta đã cấu hình.

Kubernetes cung cấp một số workload tích hợp sẵn:

- Deployment và ReplicaSet (thay thế cho ReplicationController). Deployment rất phù hợp để quản lý workload không trạng thái trên cụm, nơi bất kỳ Pod nào trong Deployment đều có thể hoán đổi cho nhau và có thể được thay thế nếu cần.
- StatefulSet là workload API object được dùng để quản lý các ứng dụng stateful (có trạng thái). Nó quản lý việc triển khai và co giãn (scale) các pod và cung cấp sự đảm bảo về thứ tự và tính duy nhất của các pod này.
- Một DaemonSet đảm bảo rằng tất cả hoặc một vài node sẽ chạy 1 bản sao của pod. Khi các node được thêm vào cluster thì pod sẽ được lập lịch vào chúng.
- Job và CronJob xác định các nhiệm vụ chạy đến khi hoàn thành và sau đó dừng lại. Công việc đại diện cho các nhiệm vụ một lần, trong khi CronJobs lặp lại theo lịch trình.

## 1.4 Một số vấn đề bảo mật kiến trúc Microservices

Trong hầu hết các ứng dụng được xây dựng trên cấu trúc monolith, bảo mật được thực thi tập trung và các thành phần riêng lẻ không cần phải lo lắng về việc thực hiện các kiểm tra bảo mật trừ khi có yêu cầu. Kết quả là, mô hình bảo mật của một ứng dụng dựa trên cấu trúc monolith là đơn giản hơn nhiều so với ứng dụng được xây dựng dựa trên cấu trúc microservices. Sau đây là một số vấn đề về bảo mật của kiến trúc Microservices gặp phải.

### 1.4.1 Càng nhiều microservices thì nguy cơ bị tấn công càng cao

Trong một ứng dụng được xây dựng trên cấu trúc monolith, giao tiếp giữa các thành phần bên trong xảy ra trong một tiến trình duy nhất — ví dụ: trong một ứng dụng Java, trong cùng một Máy ảo Java (JVM). Theo kiến trúc microservices, các thành phần bên trong đó được thiết kế dưới dạng các microservices riêng biệt, độc lập và các tiến trình phải gọi lẫn nhau để trao đổi các thông tin. Ngoài ra, mỗi microservice hiện chấp nhận các request một cách độc lập hoặc có các điểm truy cập riêng. Thay vì một vài điểm truy cập, như trong một ứng dụng xây dựng trên cấu trúc monolith, bây giờ chúng ta có số lượng lớn các điểm truy cập. Khi số lượng truy cập vào hệ thống tăng lên, chúng ta sẽ có nhiều điểm bị tấn công hơn. Vấn đề này là một trong những thách thức cơ bản trong việc xây dựng thiết kế bảo mật cho microservices.

### 1.4.2 Kiểm tra bảo mật phân tán có thể dẫn đến hiệu suất giảm

Không giống như trong một ứng dụng được xây dựng trên cấu trúc monolith, mỗi microservice trong triển khai microservices phải thực hiện kiểm tra an ninh độc lập. Từ quan điểm của một ứng dụng dựa trên cấu trúc monolith, trong đó việc kiểm tra bảo

mật được thực hiện một lần và các request được gửi đến thành phần tương ứng, nhưng đối với các microservice chúng ta phải kiểm tra tất cả các điểm truy cập của chúng. Ngoài ra, trong khi xác thực các yêu cầu tại mỗi microservice, bạn có thể cần kết nối với dịch vụ mã thông báo bảo mật từ xa (STS). Các kiểm tra bảo mật phân tán, lặp đi lặp lại và kết nối từ xa này có thể làm tăng độ trễ và làm giảm đáng kể hiệu suất của hệ thống.

Một số giải quyết vấn đề này bằng cách đơn giản là tin tưởng vào mạng và tránh kiểm tra bảo mật các microservice. Theo thời gian, mạng tin cậy đã trở thành lỗi thời và đang tiến tới các nguyên tắc mạng không tin cậy. Với nguyên tắc mạng không tin cậy, chúng ta thực hiện bảo mật với từng tài nguyên trong mạng. Mọi thiết kế bảo mật microservices phải có hiệu suất tổng thể có thể chấp nhận được.

### 1.4.3 Sự phức tạp khi triển khai xác thực khởi động giữa các microservices

Giao tiếp dịch vụ với dịch vụ diễn ra giữa các microservice. Mỗi kênh liên lạc giữa các microservice phải được bảo vệ. Chúng có nhiều tùy chọn, nhưng giả sử rằng chúng ta sử dụng chứng chỉ (certificates).

Giờ đây, mỗi microservice phải được cấp chứng chỉ (và khóa bí mật tương ứng), chứng chỉ này sẽ sử dụng để xác thực chính nó với một microservice khác trong quá trình tương tác giữa dịch vụ với dịch vụ. Microservice của người nhận phải biết cách xác thực chứng chỉ được liên kết với microservice đang gọi. Do đó cần một cách để tin tưởng bootstrap giữa các microservices. Ngoài ra cần có khả năng thu hồi chứng chỉ (trong trường hợp khóa bí mật tương ứng bị xâm phạm) và cấp các chứng chỉ mới (thay đổi chứng chỉ định kỳ để giảm thiểu mọi rủi ro khi vô tình làm mất chìa khóa). Những tác vụ này rất cồng kềnh và trừ khi chúng ta tìm ra cách tự động hóa chúng.

### 1.4.4 Khó theo dõi các request của các microservice

Khả năng giám sát là thước đo những gì có thể suy ra về trạng thái bên trong của một hệ thống dựa trên kết quả đầu ra bên ngoài của nó. Nhật ký (logs), chỉ số (metrics) và dấu vết (traces) được coi là ba trụ cột của khả năng giám sát.

Nhật ký có thể là bất kỳ sự kiện nào bạn ghi lại tương ứng với một dịch vụ nhất định.

Tổng hợp một tập hợp các nhật ký có thể tạo ra các chỉ số. Theo một cách nào đó, các chỉ số phản ánh trạng thái hệ thống. Ví dụ về mặt bảo mật, các yêu cầu truy cập không hợp lệ trung bình mỗi giờ là một chỉ số.

Dấu vết cũng dựa trên nhật ký nhưng cung cấp một góc nhìn khác về hệ thống. Theo dõi giúp bạn theo dõi một yêu cầu từ điểm mà nó đi vào hệ thống đến chỉ nơi nó rời khỏi hệ thống. Quá trình này trở nên khó khăn trong mô hình microservices. Không giống như trong một ứng dụng theo cấu trúc monolith, một yêu cầu triển khai microservices có thể xâm nhập vào hệ thống thông qua một microservice và trải dài trên nhiều microservices trước khi nó rời khỏi hệ thống.

### 1.4.5 Tính bất biến của container làm việc xác thực và chính sách kiểm soát truy cập của các dịch vụ khó khăn

Máy chủ không thay đổi trạng thái sau khi quay được gọi là máy chủ bất biến. Các mô hình triển khai phổ biến nhất cho microservices là dựa trên container. Mỗi microservice chạy bằng container riêng của nó và tất nhiên, container phải là một máy chủ bất biến. Nói cách khác, sau khi container khởi động, nó sẽ không thay đổi bất kỳ tệp nào trong hệ thống tệp của nó hoặc duy trì bất kỳ trạng thái của nó không thay đổi lúc nó đang chạy.

Tính bất biến có tác động gì đến bảo mật và tại sao khái niệm máy chủ bất biến lại được coi là thách thức bảo mật microservices? Trong kiến trúc bảo mật microservices, mỗi microservice trở thành một điểm cần được bảo mật. Do đó, cần duy trì danh sách các máy khách được phép (có thể là các microservice khác) có thể truy cập vào microservice này và cần một bộ chính sách kiểm soát truy cập. Những danh sách này không tĩnh; chúng ta cần được phép cập nhật các danh sách này. Nhưng với tính chất bất biến của container, chúng ta không thể update hệ thống tệp trong các container của các microservice.

### 1.4.6 Kiến trúc đa ngôn ngữ đòi hỏi các developer phải có thêm nhiều kiến thức bảo mật hơn

Trong triển khai microservices, các dịch vụ nói chuyện với nhau qua mạng. Họ không phụ thuộc vào việc triển khai từng dịch vụ, mà phụ thuộc vào giao diện dịch vụ. Tình huống này cho phép mỗi microservice chọn ngôn ngữ lập trình của riêng mình và nền tảng công nghệ để triển khai. Trong một môi trường, trong đó mỗi đội development triển khai các microservice của riêng mình, mỗi nhóm có thể linh hoạt để chọn các stack công nghệ để phù hợp với sản phẩm. Kiến trúc này, cho phép các thành phần trong hệ thống để chọn công nghệ tốt nhất cho chúng, được gọi là một kiến trúc đa ngôn ngữ.

Một kiến trúc đa ngôn ngữ làm cho vấn đề bảo mật trở nên khó khăn. Bởi vì các đội khác nhau sử dụng các stack công nghệ khác nhau để phát triển, mỗi nhóm phải có các chuyên gia bảo mật riêng. Các chuyên gia này phải chịu trách nhiệm xác định các phương pháp bảo mật tốt nhất và hướng dẫn, nghiên cứu các công cụ bảo mật cho mỗi stack công nghệ để phân tích mã nguồn tĩnh và thử nghiệm động và tích hợp các công cụ đó vào quá trình xây dựng. Trách nhiệm của một nhóm bảo mật tập trung, nhưng bây giờ chia vào các đội development. Điều này làm cho các đội development không chú tâm vào việc phát triển sản phẩm mà phải lo một phần về bảo mật nữa làm cho chất lượng sản phẩm giảm do phải viết các module bảo mật làm cho mã nguồn bị rối và khó đọc.

## Kết luận Chương 1

Như vậy trong chương một nhóm đã nêu ra những thông tin cơ về những công nghệ cũng như kiến trúc của tương lai - Container và Microservices. Đưa ra những ưu điểm cũng như thách thức trong việc triển khai, sử dụng những công nghệ này trong thực tế. Đồng thời nhóm cũng nêu ra một hệ thống tự động hoá triển khai, mở rộng và quản lý của công nghệ Container là Kubernetes. Qua việc hiểu tổng quan của các công nghệ trên nhóm đã nhận ra một số vấn đề bảo mật của kiến trúc Microservices như nguy cơ bị tấn công cao khi mở rộng, việc ảnh hưởng đến hiệu suất khi triển khai bảo mật phân tán cũng như sự phức tạp, khó khăn khi triển khai các giải pháp an toàn cũng yêu cầu nhiều kiến thức đối với những nhà phát triển. Sau đây nhóm sẽ trình bày một giải pháp để có thể giải quyết những vấn đề nói trên.

# Chương 2

## Tổng quan về Istio

### 2.1 Tổng quan về Istio

#### 2.1.1 Tổng quan về Service Mesh

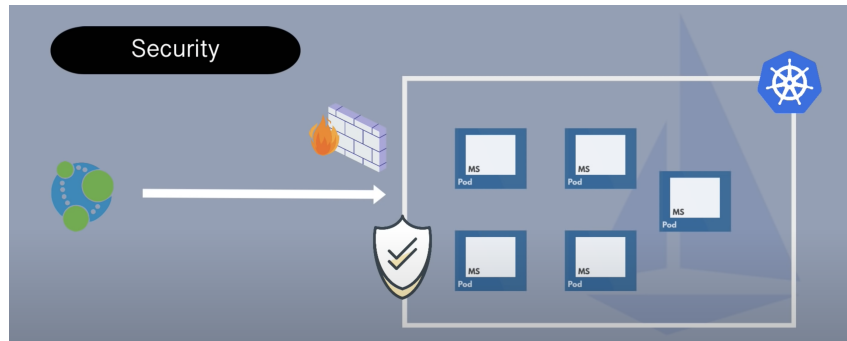
Kiến trúc Microservice ngày càng trở nên phổ biến và trở thành lựa chọn hàng đầu trong quá trình phát triển phần mềm. Trong kiến trúc microservice, ứng dụng monolithic truyền thống được chia nhỏ ra thành các thành phần có thể deploy độc lập. Một ứng dụng dần trở thành một nhóm các service, khi bạn có hàng trăm, hàng ngàn các service nhỏ cùng hoạt động, một bài toán mới về vấn đề giao tiếp giữa các service cần được giải quyết.

Giả sử chúng ta có website bán hàng online được triển khai theo kiến trúc microservice, chúng ta sẽ có:

- Web server service: xử lý các request về UI
- Payment service: xử lý các request về thanh toán
- Shopping cart service: xử lý các request về giỏ hàng
- Product inventory service: xử lý các request về kho hàng
- Database service
- ..., rất nhiều các microservice xử lý các vấn đề khác nữa

Khi chúng ta deploy website trong Kubernetes cluster, để website hoạt động ổn định, trước tiên các service cần hoạt động chính xác, web server cần xử lý chính xác business logic, database cần lưu đầy đủ thông tin,... hơn hết các service cần giao tiếp thông suốt với nhau. Khi user gửi 1 request thêm sản phẩm vào giỏ hàng lên web server, web server cần thông báo sang cho shopping cart service, rồi lưu thông tin đó vào database. Vì vậy, các service cần phải biết làm thế nào để giao tiếp với các service khác, đâu là đích đến tiếp theo của request, địa chỉ endpoint của từng service, để web server biết phải giao tiếp với endpoint nào chúng ta cần cấu hình cho nó. Khi ta scale ứng dụng lên và có nhiều thêm các microservice, chúng ta lại config endpoint cho tất cả các service đã hoạt động trước đó. Cấu hình địa chỉ endpoint trở thành 1 phần không thể tách rời của các service.

Nói về vấn đề security, ứng dụng dùng kiến trúc microservice trông sẽ như này:

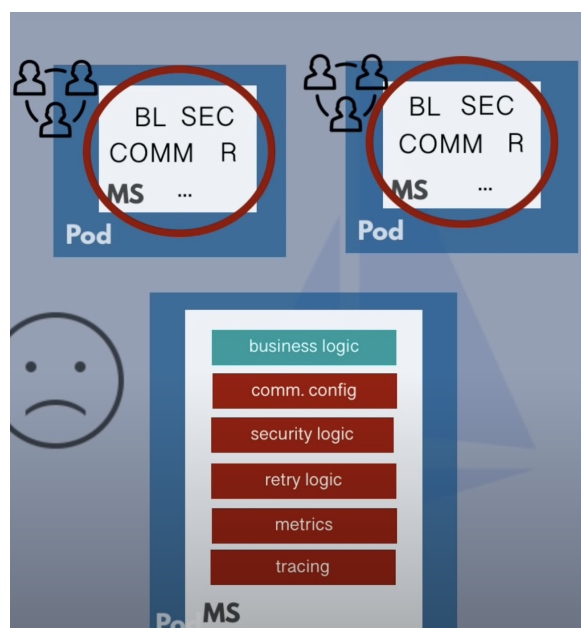


Hình 2.1: Ứng dụng kiến trúc Microservices trong Security

Chúng ta có firewall hoặc proxy để tránh Kubernetes cluster được truy cập trực tiếp, bảo vệ cho các dịch vụ bên trong cluster. Tuy nhiên ngay khi yêu cầu đi vào bên trong cluster, các dịch vụ giao tiếp với nhau 1 cách kém bảo mật, chúng giao tiếp với nhau thông qua http hoặc protocol kém bảo mật hơn, các dịch vụ giao tiếp tự do với nhau bên trong cluster - nơi không được bảo mật nữa. Khi ai đó tấn công vào ứng dụng, vượt qua firewall/proxy để vào bên trong cluster, họ thoải mái làm mọi thứ, có thể lấy các thông tin nhạy cảm của người dùng. Vì vậy chúng ta bổ sung thêm các security logic cho dịch vụ để đảm bảo việc giao tiếp giữa các dịch vụ bên trong cluster được an toàn.

Mỗi dịch vụ cũng cần được bổ sung khả năng khởi động lại để khi dịch vụ không thể truy cập, hoặc các dịch vụ khác mất kết nối, tự nó có thể khởi động lại, để đảm bảo giao tiếp thông suốt.

Ngoài ra, bạn cũng cần giám sát cách hoạt động của dịch vụ, cần nắm được các thông số về thời gian dịch vụ xử lý một yêu cầu, số lượng yêu cầu/phản hồi của mỗi dịch vụ. Cuối cùng mỗi dịch vụ sẽ được bổ sung thêm rất nhiều các logic bên cạnh business logic của dịch vụ rồi các nhà phát triển sẽ không còn tập trung vào phát triển logic cho dịch vụ mà còn phải xử lý cả các logic về mạng cho metrics, security, communication,... Điều này khiến các dịch vụ trở nên phức tạp và không còn đơn giản, gọn nhẹ như mục tiêu của microservice nữa.



Hình 2.2: Các vấn đề gặp phải

Để giải quyết vấn đề này, chúng ta cần tách biệt các non-business logic với business logic khỏi service bằng solution được gọi là Service Mesh.

Service Mesh là lớp cơ sở hạ tầng chuyên dụng kiểm soát giao tiếp qua mạng giữa dịch vụ với dịch vụ. Service Mesh cho phép các phần riêng biệt của ứng dụng giao tiếp với nhau. Service Mesh xuất hiện phổ biến cùng với các ứng dụng, bộ chứa và vi dịch vụ dựa trên đám mây.

Service Mesh kiểm soát việc cung cấp các yêu cầu dịch vụ trong một ứng dụng. Service Mesh cung cấp các tính năng phổ biến gồm khám phá dịch vụ, cân bằng tải, mã hóa và khôi phục lỗi. Thay vì thông qua phần cứng, tính sẵn sàng cao cũng phổ biến thông qua việc sử dụng phần mềm được kiểm soát bởi API. Service Mesh có thể giúp giao tiếp giữa dịch vụ với dịch vụ nhanh chóng, đáng tin cậy và an toàn.

Một tổ chức có thể chọn cổng API xử lý các giao dịch giao thức thay Service Mesh. Dù vậy, các nhà phát triển phải cập nhật cổng API mỗi khi thêm hoặc xóa một vi dịch vụ. Khả năng mở rộng quản lý mạng và tính linh hoạt của Service Mesh thường vượt trội so với khả năng của các cổng API truyền thống.

## Cách thức hoạt động của Service Mesh

Trong bất kỳ mô hình phát triển nào đang được sử dụng thì Service Mesh sử dụng một phiên bản proxy được gọi là sidecar. Trong một ứng dụng microservice, một sidecar gắn vào mỗi dịch vụ. Trong một vùng chứa, sidecar gắn vào từng vùng chứa ứng dụng, VM hoặc đơn vị điều phối vùng chứa, chẳng hạn như nhóm Kubernetes. Qua đây các Service Mesh trở thành ứng dụng bên thứ ba và giúp người vận hành cluster có thể cấu hình chúng dễ dàng hơn.

Sidecar có thể xử lý các tác vụ được trừu tượng hóa từ chính dịch vụ, chẳng hạn như giám sát và bảo mật.

Trong Service Mesh sidecar và các tương tác của chúng tạo nên được gọi là data plane. Một lớp khác quản lý các tác vụ như tạo phiên bản, giám sát và triển khai các chính sách để quản lý và bảo mật mạng được gọi là control plane. Các control plane có thể kết nối với CLI hoặc giao diện GUI để quản lý ứng dụng. Nhờ có control plane giúp cho các nhà phát triển không cần đưa Service Mesh vào trong deploy config của microservices.

## Tại sao áp dụng Service Mesh?

Một ứng dụng được cấu trúc theo kiến trúc microservices có thể chứa hàng chục hoặc hàng trăm dịch vụ, Mỗi dịch vụ đều có các phiên bản riêng hoạt động trong môi trường trực tiếp. Điều này gây khó khăn cho các nhà phát triển khi theo dõi những thành phần nào phải tương tác, theo dõi tình trạng và hiệu suất của chúng cũng như thực hiện các thay đổi đối với dịch vụ hoặc thành phần nếu có sự cố.

Service Mesh cho phép các nhà phát triển tách biệt và quản lý các giao tiếp giữa dịch vụ với dịch vụ trong một lớp cơ sở hạ tầng chuyên dụng. Lợi ích của việc sử dụng Service Mesh càng tăng khi số lượng vi dịch vụ liên quan đến một ứng dụng tăng lên.

## Các tính năng chính của Service Mesh

Service Mesh thường cung cấp nhiều khả năng giúp quá trình giao tiếp trong vùng chứa và vi dịch vụ trở nên đáng tin cậy, an toàn và dễ quan sát hơn.

- **Độ tin cậy:** Quản lý thông tin liên lạc thông qua proxy sidecar và control plane cải thiện hiệu quả và độ tin cậy của các yêu cầu dịch vụ, chính sách và cấu hình. Các khả năng cụ thể bao gồm cân bằng tải và tiêm lỗi.

- **Khả năng quan sát:** Service Mesh có thể cung cấp thông tin chi tiết về hành vi và tình trạng của dịch vụ. Control plane có thể thu thập và tổng hợp dữ liệu đo từ xa từ các tương tác thành phần nhằm xác định tình trạng dịch vụ, chẳng hạn như lưu lượng truy cập và độ trễ, theo dõi phân tán và nhật ký truy cập. Tích hợp của bên thứ ba với các công cụ, chẳng hạn như Prometheus, Elasticsearch và Grafana, cho phép theo dõi và trực quan hóa hơn nữa.
- **Bảo vệ:** Service Mesh có thể tự động mã hóa thông tin liên lạc và phân phối các chính sách bảo mật, bao gồm xác thực và ủy quyền, từ mạng đến ứng dụng và các dịch vụ nhỏ riêng lẻ. Quản lý tập trung các chính sách bảo mật thông qua control plane và proxy sidecar giúp theo kịp các kết nối ngày càng phức tạp bên trong và giữa các ứng dụng phân tán.

## Lợi ích và hạn chế của Service Mesh

Service Mesh có một số ưu điểm của như sau:

- Đơn giản hóa giao tiếp giữa các dịch vụ trong cả microservice và container.
- Dễ chẩn đoán lỗi giao tiếp hơn vì chúng sẽ xảy ra trên lớp cơ sở hạ tầng của chính chúng.
- Hỗ trợ các tính năng bảo mật như mã hóa, xác thực và ủy quyền.
- Cho phép phát triển, thử nghiệm và triển khai ứng dụng nhanh hơn.
- Sidecars được đặt bên cạnh một cụm container có hiệu quả trong việc quản lý các dịch vụ mạng.

Service Mesh có một số nhược điểm như sau:

- Thời gian chạy các dịch vụ tăng thông qua việc sử dụng Service Mesh.
- Trước tiên, mỗi cuộc gọi dịch vụ phải chạy qua proxy sidecar, điều này sẽ bổ sung thêm một bước.
- Các Service Mesh không đề cập đến việc tích hợp với các dịch vụ hoặc hệ thống khác cũng như loại định tuyến hoặc ánh xạ chuyển đổi.
- Độ phức tạp của quản lý mạng được trừu tượng hóa và tập trung, nhưng không bị loại bỏ. Do đó phải tích hợp Service Mesh vào quy trình công việc và quản lý cấu hình của nó.

## Thực trạng Thị trường Service Mesh

Theo một cuộc khảo sát được thực hiện vào giữa năm 2020, việc áp dụng Service Mesh vào doanh nghiệp vẫn còn non trẻ và thua xa so với container. Theo khảo sát thì Istio, Linkerd và HashiCorp Consul là những mạng lưới dịch vụ được sử dụng nhiều nhất.

Istio, một Service Mesh mã nguồn mở do Google, IBM và Lyft cung cấp, là một control plane chung ban đầu với mục tiêu cho các triển khai Kubernetes, nhưng các kiến trúc sư có thể sử dụng nó trên nhiều nền tảng. data plane của nó dựa trên các proxy được gọi là Envoy sidecar.

Linkerd, một Service Mesh đa nền tảng, mã nguồn mở khác, được phát triển bởi Buoyant và được xây dựng trên thư viện Finagle của Twitter. Service Mesh này hỗ trợ các nền tảng như Kubernetes, Docker và Amazon ECS.



HashiCorp's Consul cung cấp khả năng khám phá dịch vụ và Service Mesh để xử lý việc quản lý mạng trong môi trường phân tán. Nó hoạt động với AWS và Microsoft Azure và cũng có sẵn dưới dạng sản phẩm SaaS.

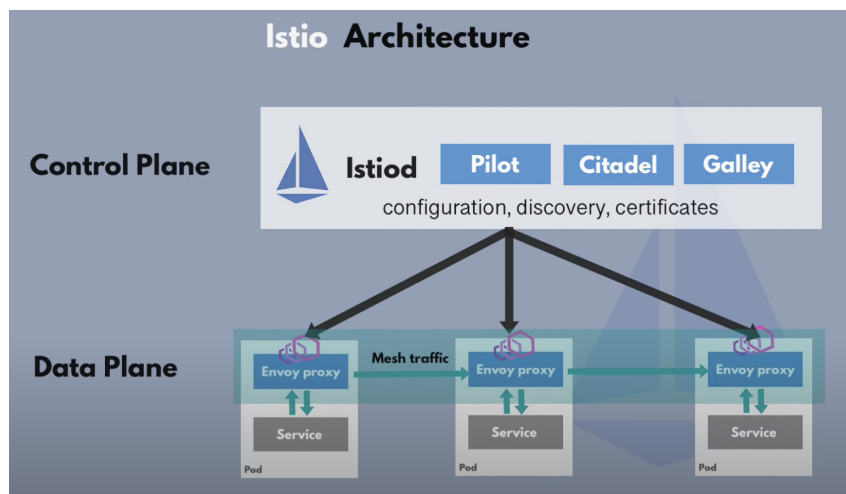
### 2.1.2 Kiến trúc của Istio

Service Mesh là 1 pattern và Istio là một trong những ứng dụng implement pattern này. Istio là service mesh độc lập, mã nguồn mở được viết bằng go-lang, một ngôn ngữ khá phổ biến để xây dựng những ứng dụng cloud-native (docker, kubernetes...), cung cấp các nguyên tắc cơ bản, giảm sự phức tạp trong việc quản lý microservice bằng cách cung cấp một cách thống nhất để bảo mật, kết nối và giám sát microservice với nhau. Istio có các API cho phép nó tích hợp vào bất kỳ nền tảng logging nào.

#### Kiến trúc của Istio

Kiến trúc của Istio gồm:

- **Data Plane:** Istio sử dụng Envoy proxy, chịu trách nhiệm:
  - Service Discovery
  - Load Balancing
  - Authentication and Authorization
  - Request Tracing
  - Traffic Management
  - Fault Injection
  - Rate Limiting
  - Observability
- **Control Plane** của Istio là istiod tiếp nhận các tương tác của DevOps/SysAdmin thông qua Command Line Interface — CLI, chịu trách nhiệm quản lý và truyền các envoy proxy vào service.



Hình 2.3: Kiến trúc của Istio

Ở phiên bản 1.5 trở về trước, control plane của Istio được gồm nhiều thành phần như Pilot, Citadel, Galley, Mixer,... mỗi thành phần thì lại được deploy trên Kubernetes như 1 pod riêng biệt. Tuy nhiên từ những version sau, chúng được gộp thành một thành phần duy nhất là Istiod, điều này làm cho việc cấu hình và vận hành Istio trở nên dễ dàng hơn.

## Ưu điểm khi sử Istio

Istio là 1 implementation của Service Mesh, nên tính độc lập của application là một trong những ưu tiên hàng đầu vì thế để đưa các chức năng của Istio vào trong ứng dụng microservice của thì chúng ta không cần thiết phải thay đổi các file cấu hình yaml của các Deployment hay Service, mọi thứ đã được đóng gói trong chính file cấu hình của Istio. Việc cài đặt Istio cũng tách rời với logic của ứng dụng, không làm thay đổi các business logic.

Các component của Istio được kế thừa từ Kubernetes API, dưới dạng CRD, giúp ta dễ dàng cấu hình chúng như cấu hình 1 deployment, service luôn. Istio hỗ trợ cấu hình:

- Traffic routing: cho phép cài đặt nhiều rules khác nhau, quy định các service được giao tiếp với nhau.
- Traffic split
- Retry rule, timeout

Hai CRD chính của Istio là VirtualService định tuyến traffic tới 1 service, khi lưu lượng đã được định tuyến tới dịch vụ, chúng ta có thể sử dụng DestinationRule để cấu hình các chính sách lên nó.

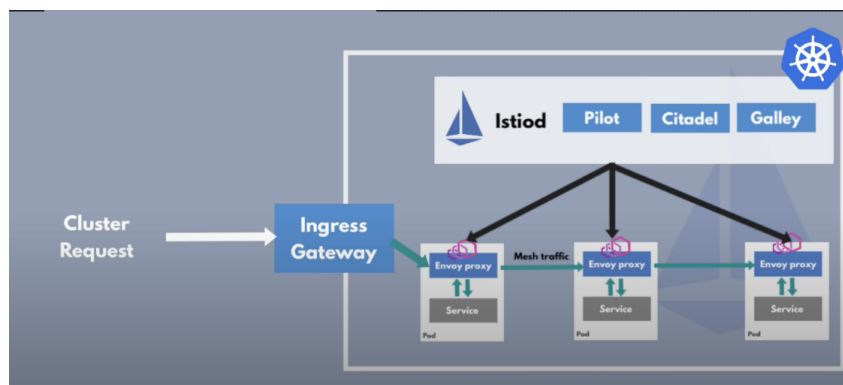
Khi chúng ta tạo các CRD trên Kubernetes, istiod - control plane của Istio, sẽ chuyển đổi chúng thành cấu hình của Envoy và gửi tất cả các cấu hình đó tới các Envoy Proxy sidecar. Khi các Envoy Proxy đã có được cấu hình, chúng sẽ giao tiếp với nhau (không thông qua istiod) dưới các rule mà chúng ta đã định nghĩa.

Ngoài việc rất dễ dàng cấu hình Istio, nó còn mang tới những ưu điểm như:

- **Service discovery:** Istio có 1 central registry cho tất cả các microservice và các endpoint, không cần phải thêm các cài đặt cho endpoint mỗi microservice, mỗi khi có thêm 1 microservice mới được deploy, nó sẽ được đăng ký 1 cách tự động với central registry của Istio
- **Istiod** cũng có thể quản lý các certificates, istiod đóng vai trò như 1 certificate authority, tạo ra certificate cho tất cả các dịch vụ trong cluster, cho phép các dịch vụ giao tiếp bảo mật hơn thông qua kết nối bảo mật TLS.
- **Istio** cũng thu thập dữ liệu từ xa thông qua các Envoy Proxy.

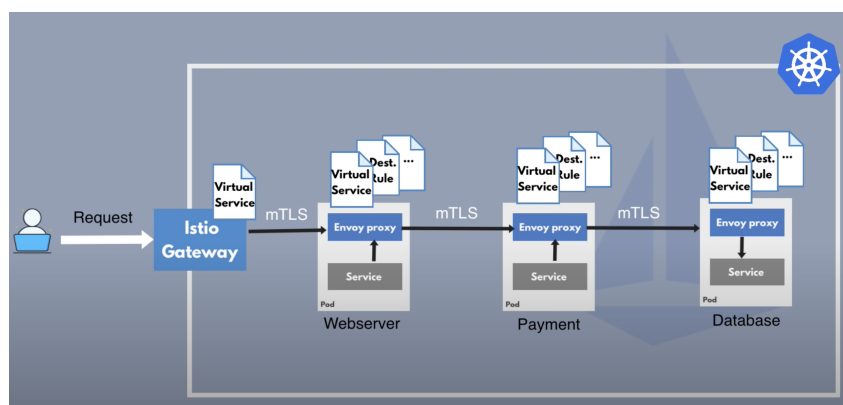
## Luồng traffic sau khi cài đặt Istio

Ngoài 2 CRD là VirtualService và DestinationRule, Gateway đóng vai trò là 1 endpoint cho các request từ cluster tới microservice application, nó hoạt động tương tự như Nginx Ingress Controller, đảm nhiệm việc cân bằng tải, điều hướng traffic tới các microservice bằng cấu hình của VirtualService.



Hình 2.4: Luồng dữ liệu sau khi cài đặt Istio

Với ví dụ một trang web bán hàng online, lưu lượng sau khi ứng dụng được cài đặt thêm Istio sẽ như sau: User gửi yêu cầu tới web server service trong Kubernetes, đầu tiên nó sẽ tới Istio Gateway, gateway sẽ tìm ra dịch vụ dựa vào VirtualService rule, và gửi yêu cầu tới dịch vụ web server, cuối cùng yêu cầu sẽ tới Envoy proxy bên trong dịch vụ và được chuyển tiếp tới web server container thông qua localhost. Từ dịch vụ web server, tiếp tục tạo 1 yêu cầu tới dịch vụ thanh toán, yêu cầu sẽ đi từ dịch vụ web server qua Envoy proxy của dịch vụ web server, chúng sẽ được so sánh với các rule của VirtualService và DestinationRule rồi mới tới Envoy proxy của dịch vụ thanh toán, giao tiếp giữa Envoy proxy của web server và payment là mutual TLS. Quá trình này được lặp lại trên các dịch vụ tiếp theo mà yêu cầu gửi đến cho tới khi phản hồi về UI.



Hình 2.5: Ví dụ về lưu lượng trong Istio

### 2.1.3 Tổng quan về Envoy Proxy

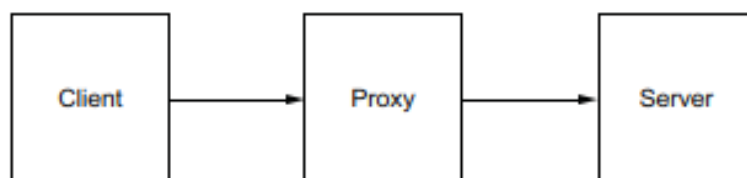
Một dịch vụ proxy khi chạy được sắp xếp nằm ngoài quy trình với ứng dụng, ứng dụng sẽ nói chuyện thông qua dịch vụ proxy cứ khi nào nó muốn giao tiếp với các dịch vụ khác. Với Istio, các Envoy proxy được triển khai cùng với tất cả các ứng dụng trong một mạng lưới services mesh, được quản lý bởi services data plane. Vì Envoy là một thành phần quan trọng trong data plane và trong kiến trúc services mesh tổng thể, nên chúng ta cần phải làm quen với nó. Điều này sẽ giúp bạn hiểu rõ hơn về Istio và cách gỡ lỗi hoặc khắc phục sự cố trong quá trình triển khai của bạn.

#### Định nghĩa về Envoy proxy

Envoy được phát triển bởi Lyft để giải quyết một số vấn đề khó khăn về mạng ứng

dụng nảy sinh khi xây dựng các hệ thống phân tán. Nó đã được xây dựng như một dự án nguồn mở vào tháng 9 năm 2016 và một năm sau (tháng 9 năm 2017) nó đã tham gia Nền tảng điện toán đám mây (CNCF). Envoy được viết bằng C++ với nỗ lực tăng hiệu suất và quan trọng hơn là làm cho nó ổn định hơn và mang tính quyết định hơn ở các mức tải cao hơn.

Envoy là một proxy, vì vậy trước khi đi xa hơn chúng ta nên làm rõ proxy là gì. Proxy là một thành phần mạng trung gian trong kiến trúc mạng được đặt trong giao tiếp giữa máy khách và máy chủ, cho phép nó cung cấp các tính năng bổ sung như các chính sách, bảo mật và quyền riêng tư.

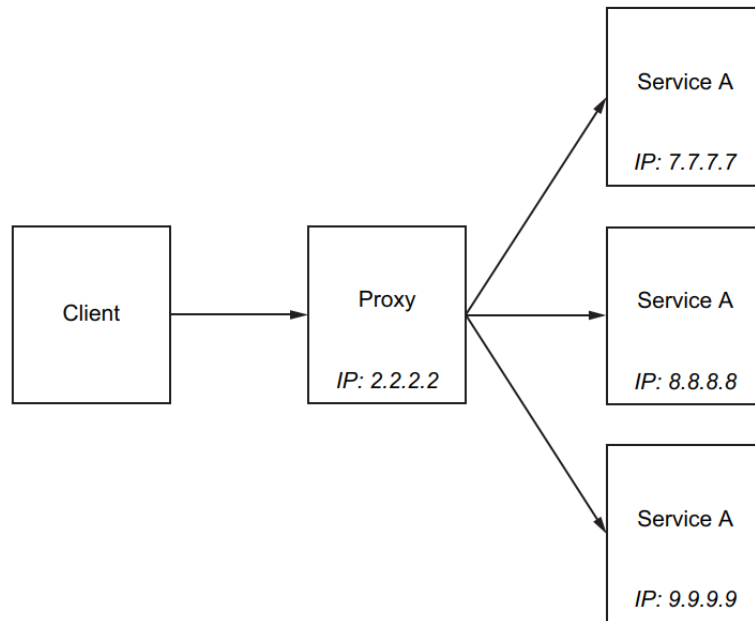


Hình 2.6: Proxy là một trung gian bổ sung các chức năng trong lưu lượng

Proxy có thể đơn giản hóa những gì máy khách cần biết khi nói chuyện với một dịch vụ. Ví dụ: một dịch vụ có thể được triển khai dưới dạng một tập hợp các phiên bản giống hệt nhau (một cụm), mỗi phiên bản có thể xử lý một lượng tải nhất định. Làm cách nào để máy khách biết phiên bản hoặc địa chỉ IP nào sẽ sử dụng khi đưa ra yêu cầu đối với dịch vụ đó? Một proxy có thể đứng ở giữa với một số nhận dạng hoặc địa chỉ IP duy nhất và máy khách có thể sử dụng nó để giao tiếp với các phiên bản của dịch vụ. Hình 2.7 cho thấy cách proxy xử lý cân bằng tải trên các phiên bản của dịch vụ mà không cần máy khách biết bất kỳ chi tiết nào về cách mọi thứ thực sự được triển khai. Một chức năng phổ biến khác của loại proxy ngược này là kiểm tra tình trạng của các phiên bản trong cụm và định tuyến lưu lượng xung quanh các phiên bản phụ trợ bị lỗi hoặc hoạt động sai. Bằng cách này, proxy có thể bảo vệ máy khách khỏi các thành phần phụ trợ đang bị quá tải hoặc lỗi.

Proxy Envoy cụ thể là một proxy tầng ứng dụng mà chúng ta có thể chèn vào đường dẫn của yêu cầu ứng dụng để cung cấp những thứ như tìm kiếm dịch vụ, cân bằng tải và kiểm tra tình trạng, nhưng Envoy có thể làm được nhiều hơn thế. Envoy có thể hiểu các giao thức lớp 7 mà một ứng dụng có thể nói khi giao tiếp với các dịch vụ khác. Ví dụ: Envoy hiểu HTTP 1.1, HTTP 2, gRPC và các giao thức khác và có thể thêm hành vi như request-level timeouts, retries, perretry timeouts, circuit breaking, và các tính năng phục hồi khác. Những thứ như thế này không thể thực hiện được với các proxy cấp kết nối cơ bản (L3/L4) chỉ hiểu các kết nối.

Envoy có thể được mở rộng để hiểu các giao thức ngoài các giao thức mặc định sẵn có. Các bộ lọc đã được viết cho các cơ sở dữ liệu như MongoDB, DynamoDB và thậm chí cả các giao thức không đồng bộ như Giao thức xếp hàng tin nhắn nâng cao (AMQP). Độ tin cậy và mục tiêu minh bạch mạng cho các ứng dụng là những nỗ lực đáng kể, nhưng điều quan trọng không kém là khả năng hiểu nhanh những gì đang xảy ra trong một kiến trúc phân tán, đặc biệt là khi mọi thứ không hoạt động như mong đợi. Vì Envoy hiểu các giao thức cấp ứng dụng và lưu lượng ứng dụng đi qua Envoy, nên proxy có thể thu thập nhiều dữ liệu từ xa về các yêu cầu truyền qua hệ thống, chẳng hạn như thời gian thực hiện các yêu cầu đó, số lượng yêu cầu mà một số dịch vụ nhất định đang xem (thông qua put) và tỷ lệ lỗi mà các dịch vụ đang gặp phải.



Hình 2.7: Một proxy có thể ẩn cấu trúc liên kết phụ trợ khỏi máy khách và thực hiện các thuật toán để phân phối lưu lượng truy cập một cách công bằng (cân bằng tải).

Là một proxy, Envoy được thiết kế để bảo vệ các nhà phát triển khỏi các mối lo ngại về mạng bằng cách chạy các ứng dụng không mang tính quy trình. Điều này có nghĩa là bất kỳ ứng dụng nào được viết bằng bất kỳ ngôn ngữ lập trình nào hoặc với bất kỳ khuôn khổ nào đều có thể tận dụng các tính năng này. Hơn nữa, mặc dù các kiến trúc dịch vụ (SOA, microservice, v.v.) là kiến trúc tất yếu, nhưng Envoy không quan tâm liệu bạn có đang thực hiện các dịch vụ siêu nhỏ hay bạn có các ứng dụng nguyên khối và kế thừa được viết bằng bất kỳ ngôn ngữ nào hay không. Miễn là họ trao đổi bằng các giao thức mà Envoy có thể hiểu (như HTTP), Envoy đều có thể được sử dụng.

Envoy là một proxy rất linh hoạt và có thể được sử dụng trong các vai trò khác nhau: như một proxy ở rìa cụm (như một điểm vào), như một proxy được chia sẻ cho một máy chủ hoặc một nhóm dịch vụ và thậm chí là một proxy cho mỗi dịch vụ như chúng ta thấy với Istio. Với Istio, một proxy Envoy duy nhất được triển khai cho mỗi phiên bản dịch vụ để đạt được sự linh hoạt, hiệu suất và khả năng kiểm soát cao nhất. Chỉ vì ta sử dụng một loại mẫu triển khai (proxy dịch vụ sidecar) không có nghĩa là ta cũng không thể có lợi thế được phân phối với Envoy. Trên thực tế, việc có proxy được triển khai giống nhau ở rìa cũng như được đặt trong lưu lượng ứng dụng có thể giúp cơ sở hạ tầng của ta dễ vận hành và suy luận hơn. Như chúng ta thấy, Envoy có thể được sử dụng ở biên để xâm nhập và liên kết với lưới dịch vụ để cung cấp toàn quyền kiểm soát và quan sát lưu lượng từ điểm nó đi vào cụm cho đến các dịch vụ riêng lẻ trong biểu đồ lời gọi cho một yêu cầu cụ thể.

## Các chức năng chính của Envoy

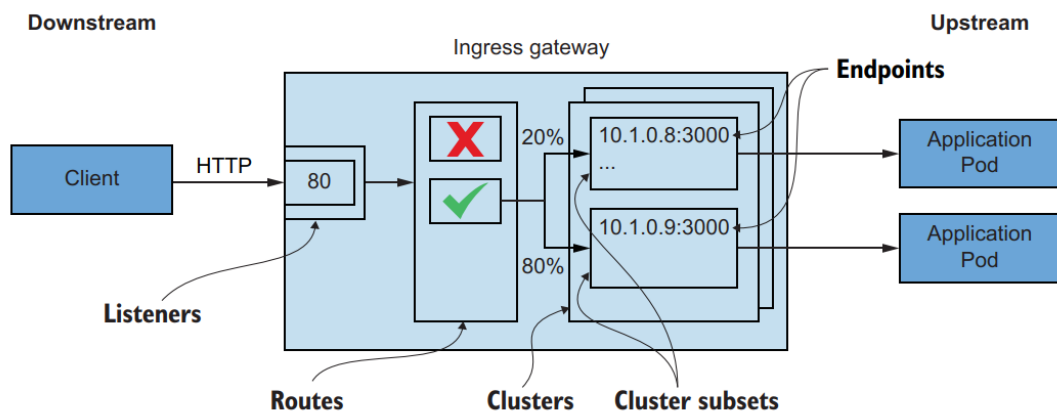
Envoy có nhiều tính năng hữu ích cho giao tiếp giữa các dịch vụ. Để giúp hiểu các tính năng và khả năng này, bạn nên nắm rõ các khái niệm Envoy sau đây:

- **Listeners**—Đưa một cổng ra bên ngoài, nơi mà các ứng dụng có thể kết nối. Ví dụ: một listener trên cổng 80 chấp nhận lưu lượng và áp dụng các hành vi được cấu hình chỉ định cho lưu lượng đó.

- **Route**—Quy tắc định tuyến để xử lý lưu lượng đến từ listener. Ví dụ: nếu một yêu cầu đến và khớp với `/catalog`, hãy hướng lưu lượng truy cập đó đến cụm danh mục `/catalog`.
- **Clusters**—Các dịch vụ cấp trên cụ thể mà Envoy có thể định tuyến lưu lượng. Ví dụ: `catalog-v1` và `catalog-v2` có thể là các cụm riêng biệt và các tuyến đường có thể chỉ định các quy tắc về cách hướng lưu lượng truy cập đến v1 hoặc v2 của dịch vụ `catalog`.

Đây là những khái niệm cơ bản về những gì mà Envoy có thể thực hiện đối với lưu lượng L7

Envoy sử dụng thuật ngữ tương tự như thuật ngữ của các proxy khác khi chuyển hướng lưu lượng truy cập. Ví dụ: lưu lượng truy cập đến một listener từ hệ thống cấp dưới. Lưu lượng này được định tuyến đến một trong các cụm của Envoy, cụm này chịu trách nhiệm gửi lưu lượng đó đến một hệ thống cấp trên (như trong hình 2.8). Dòng lưu lượng qua Envoy từ hạ lưu đến thượng nguồn. Bây giờ, hãy chuyển sang một số tính năng của Envoy.



Hình 2.8: Một yêu cầu đến từ một hệ thống cấp dưới thông qua các listeners, đi qua quá trình định tuyến và kết thúc bằng một cụm gửi nó đến một dịch vụ cấp trên.

### Service discovery

Thay vì sử dụng các thư viện runtime đặc thù để tìm kiếm dịch vụ phía máy khách, Envoy có thể tự động thực hiện việc này cho một ứng dụng. Bằng cách định cấu hình Envoy để tìm kiếm các điểm cuối dịch vụ từ một API tìm kiếm đơn giản, các ứng dụng có thể không biết cách tìm thấy các điểm cuối dịch vụ. API tìm kiếm là một API REST đơn giản có thể được sử dụng để bọc các API tìm kiếm dịch vụ phổ biến khác (như HashiCorp Consul, Apache ZooKeeper, Netflix Eureka, v.v.). Control plane của Istio triển khai API này ngay lập tức.

Envoy được xây dựng đặc biệt để dựa vào các bản cập nhật nhất quán cuối cùng cho danh mục tìm kiếm dịch vụ. Điều này có nghĩa là trong một hệ thống phân tán, ta không thể biết chính xác trạng thái của tất cả các dịch vụ mà chúng ta có thể liên lạc và liệu chúng có khả dụng hay không. Điều tốt nhất chúng ta có thể làm là sử dụng kiến thức có sẵn, sử dụng kiểm tra chủ động và thụ động.

Istio đơn giản hoá rất nhiều chi tiết này bằng cách cung cấp một bộ tài nguyên cấp cao hơn giúp điều khiển cấu hình cơ chế tìm kiếm dịch vụ của Envoy. Chúng ta sẽ xem xét kỹ hơn vấn đề này trong suốt cuốn sách.

### Cân bằng tải

Envoy thực hiện một vài thuật toán cân bằng tải nâng cao mà các ứng dụng có thể tận

dụng. Một trong những khả năng thú vị hơn của các thuật toán cân bằng tải của Envoy là cân bằng tải nhận biết cục bộ. Trong tình huống này, Envoy đủ thông minh để ngăn lưu lượng truy cập vượt qua bất kỳ ranh giới nào trừ khi nó đáp ứng các tiêu chí nhất định và sẽ mang lại sự cân bằng lưu lượng tốt hơn. Ví dụ: Envoy đảm bảo rằng lưu lượng dịch vụ đến dịch vụ được định tuyến đến các phiên bản trong cùng một địa điểm trừ khi làm như vậy sẽ tạo ra tình huống lỗi. Envoy cung cấp các thuật toán cân bằng tải vượt trội cho các chiến lược sau:

- Random
- Round robin
- Weighted, least request
- Consistent hashing (sticky)

### **Kiểm soát và định hướng đường lưu lượng**

Vì Envoy có thể hiểu các giao thức ứng dụng như HTTP 1.1 và HTTP 2 nên nó có thể sử dụng các quy tắc định tuyến tinh vi để hướng lưu lượng truy cập đến các cụm phụ trợ cụ thể. Envoy có thể thực hiện định tuyến proxy ngược cơ bản như ánh xạ máy chủ ảo và định tuyến theo ngữ cảnh; nó cũng có thể thực hiện định tuyến dựa trên tiêu đề và mức độ ưu tiên, retries and timeouts để định tuyến cũng như fault injection.

### **Dịch chuyển và tạo bóng lưu lượng**

Envoy hỗ trợ phân tách/chuyển dịch lưu lượng dựa trên tỷ lệ phần trăm (nghĩa là có trọng số). Điều này cho phép các nhóm nhanh chóng sử dụng các kỹ thuật phân phối liên tục để giảm thiểu rủi ro, chẳng hạn như phát hành canary. Mặc dù chúng giảm thiểu rủi ro cho nhóm người dùng nhỏ hơn, các bản phát hành canary vẫn xử lý lưu lượng người dùng trực tiếp.

Envoy cũng có thể tạo các bản sao của lưu lượng truy cập và tạo bóng cho lưu lượng truy cập đó trong chế độ chạy và quên thành một cụm Envoy. Bạn có thể coi khả năng tạo bóng này giống như phân tách lưu lượng, nhưng các yêu cầu mà cụm cấp trên nhìn thấy là một bản sao của lưu lượng trực tiếp; do đó, chúng ta có thể định tuyến lưu lượng truy cập ẩn đến phiên bản mới của dịch vụ mà không thực sự tác động đến lưu lượng truy cập sản xuất trực tiếp. Đây là một khả năng rất mạnh mẽ để thử nghiệm các thay đổi dịch vụ với lưu lượng khác mà không ảnh hưởng đến khách hàng.

### **Khả năng phục hồi mạng**

Envoy có thể được sử dụng để giảm tải một số loại vấn đề về khả năng phục hồi, nhưng lưu ý rằng trách nhiệm của ứng dụng là tinh chỉnh và định cấu hình các tham số này. Envoy có thể tự động thực hiện hết thời gian chờ yêu cầu cũng như thử lại ở cấp độ yêu cầu (với thời gian chờ cho mỗi lần thử lại). Loại hành vi thử lại này rất hữu ích khi một yêu cầu gặp phải tình trạng mạng không ổn định liên tục. Mặt khác, thử lại khuếch đại có thể dẫn đến lỗi xếp tầng; Envoy cho phép bạn giới hạn hành vi thử lại. Cũng lưu ý rằng vẫn có thể cần thử lại ở cấp độ ứng dụng và không thể giảm tải hoàn toàn cho Envoy. Ngoài ra, khi Envoy gọi các cụm ngược dòng, nó có thể được định cấu hình với các đặc điểm phân vùng như giới hạn số lượng kết nối hoặc yêu cầu chưa xử lý trong chuyến bay và nhanh chóng thất bại bất kỳ kết nối nào vượt quá các ngưỡng đó (với một số rung pha trên các ngưỡng đó). Cuối cùng, Envoy có thể thực hiện phát hiện ngoại lệ, hoạt động giống như một bộ ngắt mạch và loại bỏ các điểm cuối khỏi nhóm cân bằng tải khi chúng hoạt động sai.

### **HTTP/2 và gRPC**

HTTP/2 là một cải tiến đáng kể đối với giao thức HTTP, cho phép ghép các yêu cầu qua một kết nối duy nhất, tương tác đẩy máy chủ, tương tác truyền trực tuyến và áp

lực ngược yêu cầu. Envoy được xây dựng ngay từ đầu để trở thành proxy HTTP/1.1 và HTTP/2 với khả năng ủy quyền cho từng giao thức cả hạ lưu và ngược dòng. Ví dụ, điều này có nghĩa là Envoy có thể chấp nhận các kết nối HTTP/1.1 và ủy quyền cho HTTP/2 hoặc ngược lại hoặc ủy quyền HTTP/2 đến cho các cụm HTTP/2 cấp trên. gRPC là một giao thức RPC sử dụng Google Protocol Buffers (Protobuf) nằm trên HTTP/2 và cũng được hỗ trợ bởi Envoy. Đây là những tính năng mạnh mẽ (và khó sửa sai khi triển khai) và phân biệt Envoy với các proxy dịch vụ khác.

### Khả năng giám sát với Metric Collection

Như chúng ta đã thấy trong thông báo của Envoy từ Lyft vào tháng 9 năm 2016, một trong những mục tiêu của Envoy là giúp mạng trở nên dễ hiểu. Envoy thu thập một tập hợp lớn các số liệu để giúp đạt được mục tiêu này. Nó theo dõi nhiều chiều xung quanh các hệ thống cấp trên gọi nó, chính máy chủ và các cụm cấp dưới mà nó gửi yêu cầu. Số liệu thống kê của Envoy được theo dõi dưới dạng bộ đếm, đồng hồ đo hoặc biểu đồ. Bảng 3.1 liệt kê một số ví dụ về các loại thống kê được theo dõi cho một cụm cấp dưới.

Statistic	Description
<code>downstream_cx_total</code>	Total connections
<code>downstream_cx_http1_active</code>	Total active HTTP/1.1 connections
<code>downstream_rq_http2_total</code>	Total HTTP/2 requests
<code>cluster.&lt;name&gt;.upstream_cx_overflow</code>	Total number of times that the cluster's connection circuit breaker overflowed
<code>cluster.&lt;name&gt;.upstream_rq_retry</code>	Total number of request retries
<code>cluster.&lt;name&gt;.ejections_detected_consecutive_5xx</code>	Number of detected consecutive 5xx ejections (even if unenforced)

Hình 2.9: Một số số liệu thống kê mà Envoy proxy thu thập

Envoy có thể phát ra số liệu thống kê bằng cách sử dụng các định dạng và bộ điều hợp có thể định cấu hình. Ngoài ra, Envoy hỗ trợ như sau:

- StatsD
- Datadog; DogStatsD
- Hystrix formatting
- Generic metrics service

### Khả năng giám sát với Distributed Tracing

Envoy có thể báo cáo các khoảng thời gian theo dõi cho các công cụ OpenTracing (<http://opentracing.io>) để trực quan hóa luồng lưu lượng, bước nhảy và độ trễ trong biểu đồ lời gọi. Điều này có nghĩa là bạn không phải cài đặt các thư viện OpenTracing đặc biệt. Mặt khác, ứng dụng chịu trách nhiệm truyền tải các tiêu đề Zipkin cần thiết, điều này có thể được thực hiện với các thư viện trình bao bọc mỏng.

Envoy tạo tiêu đề `x-request-id` để tương quan các lệnh gọi giữa các dịch vụ và cũng có thể tạo tiêu đề `x-b3*` ban đầu khi kích hoạt theo dõi. Các tiêu đề mà ứng dụng chịu trách nhiệm tuyên truyền như sau:

- `x-b3-traceid`
- `x-b3-parentspanid`
- `x-b3-sampled`



- x-b3-flags

### Sử dụng tự động TLS

Envoy có thể dùng lưu lượng Transport Level Security (TLS) dành cho một dịch vụ cụ thể ở cả rìa của cụm và sâu bên trong mạng lưới proxy dịch vụ. Một khả năng thú vị hơn là Envoy có thể được sử dụng để tạo lưu lượng truy cập TLS đến cụm cấp trên thay mặt cho ứng dụng. Đối với các nhà phát triển và nhà điều hành doanh nghiệp, điều này có nghĩa là ta không phải loay hoay với các cài đặt dành riêng cho ngôn ngữ và keystores hay truststores. Bằng cách có Envoy trong đường dẫn yêu cầu, chúng ta có thể tự động nhận TLS và thậm chí cả TLS lẫn nhau.

### Rate Limit

Một khía cạnh quan trọng của khả năng phục hồi là khả năng hạn chế hoặc giới hạn quyền truy cập vào các tài nguyên được bảo vệ. Các tài nguyên như cơ sở dữ liệu hoặc bộ đệm hoặc dịch vụ dùng chung có thể được bảo vệ vì nhiều lý do:

- Expensive to call (per-invocation cost)
- Slow or unpredictable latency
- Fairness algorithms needed to protect against starvation

Đặc biệt là khi các dịch vụ được định cấu hình để thử lại, chúng ta không muốn phóng đại tác động của một số lỗi nhất định trong hệ thống. Để giúp điều tiết các yêu cầu trong các tình huống này, chúng ta có thể sử dụng dịch vụ giới hạn tốc độ toàn bộ. Envoy có thể tích hợp với dịch vụ giới hạn tốc độ ở cả cấp độ mạng (trên mỗi kết nối) và HTTP (theo yêu cầu).

### Envoy mở rộng

Về cốt lõi, Envoy là một công cụ xử lý byte trên đó có thể xây dựng các codec giao thức (lớp 7) (được gọi là bộ lọc). Envoy làm cho việc xây dựng các bộ lọc bổ sung trở thành trường hợp sử dụng cụ thể và là một cách thú vị để mở rộng Envoy cho các trường hợp sử dụng cụ thể. Bộ lọc Envoy được viết bằng C++ và được biên dịch thành mã nhị phân Envoy. Ngoài ra, Envoy hỗ trợ tập lệnh Lua([www.lua.org](http://www.lua.org)) và WebAssembly (Wasm) để có cách tiếp cận ít xâm lấn hơn nhằm mở rộng chức năng của Envoy.

### So sánh với các loại proxy khác

Điểm hấp dẫn của Envoy là nó đóng vai trò của proxy ứng dụng hoặc dịch vụ, trong đó Envoy tạo điều kiện cho các ứng dụng nói chuyện với nhau thông qua proxy và giải quyết các vấn đề về độ tin cậy và khả năng giám sát. Các proxy khác đã phát triển từ bộ cân bằng tải và máy chủ web thành các proxy có khả năng và hiệu suất cao hơn. Một số cộng đồng này không phát triển như vậy hoặc là mã nguồn đóng và đã mất một thời gian để phát triển đến mức chúng có thể được sử dụng trong các tình huống từ ứng dụng đến ứng dụng. Đặc biệt, Envoy tỏa sáng so với các proxy khác trong các lĩnh vực như sau:

- Extensibility with WebAssembly
- Open community
- Modular codebase built for maintenance and extension
- HTTP/2 support (upstream and downstream)
- Deep protocol metrics collection
- C++ / non-garbage-collected

- Dynamic configuration, no need for hot restarts

## Cấu hình Envoy

Envoy được điều khiển bởi một tệp cấu hình ở định dạng JSON hoặc YAML. Tệp cấu hình chỉ định listener, route và cluster cũng như các cài đặt dành riêng cho máy chủ như có bật API quản trị hay không, vị trí nhật ký truy cập, theo dõi cấu hình công cụ, v.v. Nếu bạn đã quen thuộc với Envoy hoặc cấu hình Envoy, bạn có thể biết rằng có nhiều phiên bản khác nhau của cấu hình Envoy. Các phiên bản ban đầu, v1 và v2, không được dùng nữa để chuyển sang phiên bản v3. Chúng tôi chỉ xem xét cấu hình v3 trong cuốn sách này, vì đó là phiên bản tiếp theo và là những gì Istio sử dụng.

API cấu hình v3 của Envoy được xây dựng trên gRPC. Envoy và những người triển khai API v3 có thể tận dụng các khả năng phát trực tuyến khi gọi API và giảm thời gian cần thiết để các proxy của Envoy hội tụ về cấu hình chính xác. Trên thực tế, điều này giúp loại bỏ nhu cầu thăm dò API và cho phép máy chủ đẩy các bản cập nhật lên Envoys thay vì thăm dò proxy theo các khoảng thời gian định kỳ.

## 2.2 Quản lý mạng giữa các Microservices với Istio

### 2.2.1 Tổng quan về Istio Ingress Gateway

Istio có một gateway vào đóng vai trò là điểm vào mạng và chịu trách nhiệm bảo vệ và kiểm soát quyền truy cập vào cụm bởi lưu lượng truy cập bắt nguồn từ bên ngoài cụm. Ngoài ra, gateway của Istio xử lý cân bằng tải và định tuyến máy chủ ảo.

Hình 2.10 cho thấy thành phần cổng vào của Istio cho phép lưu lượng truy cập vào cụm và thực hiện chức năng proxy ngược. Istio sử dụng một proxy Envoy duy nhất làm gateway. Chúng ta đã thấy ở trên rằng Envoy là một proxy có khả năng kết nối dịch vụ đến dịch vụ, nhưng nó cũng có thể được sử dụng để cân bằng tải và định tuyến lưu lượng từ bên ngoài lưới dịch vụ đến các dịch vụ chạy bên trong nó. Tất cả các tính năng của Envoy mà chúng ta đã thảo luận trong chương trước cũng có sẵn ở đây.

Chúng ta hãy xem xét kỹ hơn cách Istio sử dụng Envoy để triển khai thành phần gateway của nó. Khi cài đặt Istio, hình 2.11 hiển thị danh sách các thành phần tạo nên control plane và các thành phần bổ sung hỗ trợ cho nó.

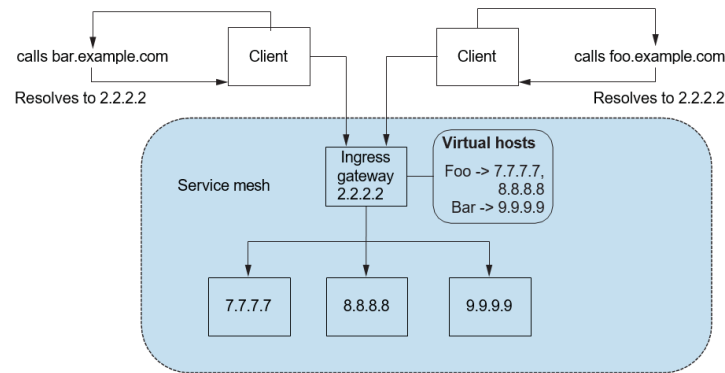
Trong hình 2.11, bên cạnh Pod istio-ingressgateway, hãy lưu ý thành phần istio-egressgateway. Thành phần này chịu trách nhiệm định tuyến lưu lượng ra khỏi cụm. Cổng ra được cấu hình với cùng tài nguyên như cổng vào mà chúng ta sẽ thấy trong chương này.

Nếu ta muốn xác minh rằng proxy dịch vụ Istio (proxy Envoy) thực sự đang chạy trong Istio gateway, ta có thể chạy thử một số câu lệnh như sau:

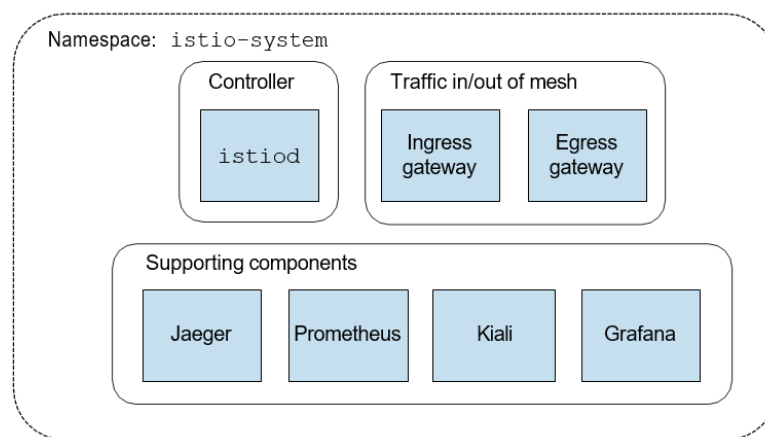
```
# kubectl -n istio-system exec
deploy/istio-ingressgateway - ps
```

```
PID TTY TIME CMD
1 ? 00:00:04 pilot-agent
14 ? 00:00:24 envoy
44 ? 00:00:00 ps
```

Ta sẽ thấy đầu ra dưới dạng đầu ra một danh sách, hiển thị dòng lệnh dịch vụ proxy Istio với các pilot-agent và envoy dưới dạng các tiến trình đang chạy. Tiến trình pilot-agent ban đầu định cấu hình và khởi động proxy Envoy và nó cũng triển khai proxy DNS.



Hình 2.10: Istio Gateway đóng vai trò là điểm vào mạng và sử dụng proxy Envoy để thực hiện định tuyến và cân bằng tải.



Hình 2.11: Cấu trúc một control plane và các thành phần phụ trợ

Để cấu hình gateway của Istio nhằm cho phép lưu lượng truy cập vào cụm và thông qua lưới dịch vụ, chúng ta sẽ bắt đầu bằng cách khám phá hai tài nguyên của Istio: Cổng và Dịch vụ ảo. Cả hai đều là nền tảng để có được lưu lượng truy cập trong Istio, nhưng chúng ta sẽ chỉ xem xét chúng trong bối cảnh cho phép lưu lượng truy cập vào cụm.

### Chỉ định tài nguyên của cổng

Để cấu hình gateway trong Istio, chúng ta sử dụng tài nguyên Cổng và chỉ định cổng nào chúng ta muốn mở và máy chủ ảo nào cho phép các cổng đó. Ví dụ về cấu hình tài nguyên Cổng mà chúng ta sẽ khám phá khá đơn giản và hiển thị một cổng HTTP trên cổng 80 chấp nhận lưu lượng dành cho máy chủ ảo webapp.istioinaction.io như dưới hình 2.12.

Gateway ở đây cấu hình Envoy để nghe trên cổng 80 và cho phép lưu lượng HTTP. Hãy bắt đầu với việc tạo các cấu hình tài nguyên và theo dõi quá trình hoạt động. Trong thư mục gốc của mã nguồn ở ví dụ này là tệp `ch4/coolstore-gw.yaml`. Để tạo tệp cấu hình, ta chạy câu lệnh như sau:

```
# kubectl -n istioinaction apply -f ch4/coolstore-gw.yaml
```

```

apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: coolstore-gateway
spec:
  selector:
    istio: ingressgateway
  servers:
    - port:
        number: 80
        name: http
        protocol: HTTP
      hosts:
        - "webapp.istioinaction.io"

```

Annotations in the image:

- Name of the gateway**: points to `name: coolstore-gateway`
- Which gateway implementation**: points to `istio: ingressgateway`
- Ports to expose**: points to `number: 80`
- Host(s) for this port**: points to `"webapp.istioinaction.io"`

Hình 2.12: Cầu hình cài nguyên của gateway

Ta kiểm tra xem liệu cài đặt của chúng ta có hiệu lực hay không bằng cách dùng câu lệnh:

```
# istioctl -n istio-system proxy-config
listener deploy/istio-ingressgateway
```

```

ADDRESS PORT MATCH DESTINATION
0.0.0.0 8080 ALL Route: http.80
0.0.0.0 15021 ALL Inline Route: /healthz/ready*
0.0.0.0 15090 ALL Inline Route: /stats/prometheus*

```

Nếu bạn thấy đầu ra này, thì bạn đã hiển thị đúng cổng HTTP (cổng 80). Nhìn vào các tuyến đường cho các dịch vụ ảo, chúng ta thấy rằng cổng hiện không có bất kỳ tuyến đường nào (ta có thể thấy một tuyến đường khác cho Prometheus, nhưng hiện tại ta có thể bỏ qua nó):

**Note** Nếu không sử dụng Docker Desktop, tên của listener (trong trường hợp này là “http.8080”) có thể khác. Hãy cập nhật lệnh dưới đây cho phù hợp.

```

1 $ istioctl proxy-config route deploy/istio-ingressgateway \
2 -o json --name http.8080 -n istio-system
3
4 [
5 {
6   "name": "http.8080", "virtualHosts": [
7   {
8     "name": "blackhole:80", "domains": [
9       "*"
10    ],
11    },
12  ],
13  "validateClusters": false
14 }
15 ]

```

Ở đây listener bị ràng buộc với một tuyến đường mặc định dẫn tới một lỗ đen đưa mọi thứ tới HTTP 404. Trong phần tiếp theo, ta thiết lập một máy chủ ảo để định tuyến lưu lượng truy cập từ cổng 80 đến một dịch vụ trong mạng lưới dịch vụ.

Trước khi tiếp tục, có một điểm quan trọng cuối cùng cần được thực hiện. Pod mà gateway chạy trên nó, cho dù đó là gateway vào hệ thống mặc định hay gateway tùy chỉnh

của riêng bạn, đều phải có khả năng lắng nghe trên một cổng hoặc IP được hiển thị bên ngoài cụm. Ví dụ: trên Docker Desktop cục bộ trong mô hình mà đang sử dụng cho các ví dụ này, gateway đang lắng nghe trên cổng 80. Nếu bạn đang triển khai trên một dịch vụ đám mây như Google Container Engine (GKE), hãy đảm bảo rằng bạn sử dụng một dịch vụ loại LoadBalancer, nhận địa chỉ IP có thể định tuyến bên ngoài. Bạn có thể tìm thêm thông tin tại <https://istio.io/v1.13/docs/tasks/traffic-management/ingress/>.

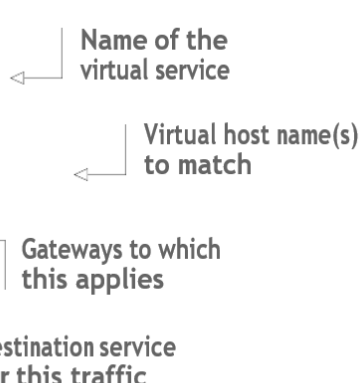
Ngoài ra, gateway mặc định của hệ thống không cần có đặc quyền để mở bất kỳ cổng nào vì nó không lắng nghe trên bất kỳ cổng hệ thống nào (80 cho HTTP). istio-ingressgateway theo mặc định lắng nghe trên cổng 8080; tuy nhiên, bất kỳ dịch vụ hoặc bộ cân bằng tải nào bạn sử dụng để hiển thị gateway đều là cổng thực tế. Trong các ví dụ ở đây với Docker Desktop, chúng hiển thị dịch vụ trên cổng 80.

## Định tuyến gateway với các dịch vụ ảo

Cho đến bây giờ, tất cả những gì ta đã làm là cấu hình Istio gateway để hiển thị một cổng cụ thể, cho phép một giao thức cụ thể trên cổng đó và xác định các máy chủ cụ thể sẽ phục vụ từ cặp cổng/giao thức. Khi lưu lượng truy cập vào cổng, chúng ta cần một cách để đưa nó đến một dịch vụ cụ thể trong lưới dịch vụ; và để làm điều đó, chúng ta sẽ sử dụng VirtualService. Trong Istio, VirtualService xác định cách máy khách trao đổi với một dịch vụ cụ thể thông qua tên miền đủ điều kiện của nó, phiên bản nào của dịch vụ khả dụng và các thuộc tính định tuyến khác (như retries và request timeouts). Chúng ta sẽ đề cập sâu hơn về Dịch vụ ảo trong chương tiếp theo khi khám phá định tuyến lưu lượng truy cập; trong chương này, chỉ cần biết rằng VirtualService cho phép chúng ta định tuyến lưu lượng truy cập từ cổng vào đến một dịch vụ cụ thể là đủ.

Một ví dụ về VirtualService định tuyến lưu lượng truy cập cho máy chủ ảo "webapp.istioinaction.io" tới các dịch vụ được triển khai trong lưới dịch vụ của chúng ta trông giống như hình 2.13.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: webapp-vs-from-gw
spec:
  hosts:
  - "webapp.istioinaction.io"
  gateways:
  - coolstore-gateway
  http:
  - route:
    - destination:
        host: webapp
        port:
          number: 8080
```



Hình 2.13: Ví dụ về VirtualService

Với tài nguyên VirtualService này, chúng tôi xác định phải làm gì với lưu lượng truy cập khi đi vào gateway. Trong trường hợp này, như ta có thể thấy từ trường spec.gateways, các quy tắc lưu lượng này chỉ áp dụng cho lưu lượng truy cập đến từ cổng được định sẵn coolstore-gateway mà chúng ta đã tạo trong phần trước. Ngoài ra, chúng ta chỉ định máy chủ ảo webapp.istioinaction.io mà lưu lượng truy cập phải tuân thủ các quy tắc đã được định sẵn. Một ví dụ về các quy tắc này là một ứng dụng khách đang truy vấn http://webapp.istio inaction.io, truy vấn này phân giải thành một IP mà cổng Istio đang

lắng nghe. Ngoài ra, một máy khách có thể đặt tiêu đề Máy chủ trong yêu cầu HTTP là `webapp.istioinaction.io`, như sẽ trình bày qua một ví dụ.

Một lần nữa, để có thể xác minh rằng ta đang ở trong thư mục gốc của mã nguồn:

```
1 $ kubectl apply -n istioinaction -f ch4/coolstore-vs.yaml
```

Sau một lúc (cấu hình cần phải đồng bộ hóa; chờ cấu hình trong lưới dịch vụ Istio nhất quán), chúng ta có thể chạy lại các lệnh của mình để liệt kê những listener và route:

```
1 $ istioctl proxy-config route deploy/istio-ingressgateway \
2 -o json --name http.8080 -n istio-system
```

```
1 [
2 {
3   "name": "http.8080", "virtualHosts": [
4     {
5       "name": "webapp-vs-from-gw:80",
6       "domains": [
7         "webapp.istioinaction.io"
8       ],
9       "routes": [
10        {
11          "match": {
12            "prefix": "/"
13          },
14          "route": {
15            "cluster":
16              "outbound|8080||webapp.istioinaction.svc.cluster.local", "
17            timeout": "0.000s"
18          }
19        }
20      ]
21    }
22  ]
23 ]
24
```

Đầu ra cho route ở đây sẽ trông giống như danh sách trước đó, mặc dù nó có thể chứa các thuộc tính và thông tin khác. Phần quan trọng là chúng ta có thể thấy cách xác định VirtualService đã tạo tuyến đường Envoy trong Istio gateway của chúng ta định tuyến lưu lượng truy cập phù hợp với miền `webapp.istioinaction.io` tới ứng dụng web trong lưới dịch vụ.

Chúng ta đã thiết lập định tuyến cho các dịch vụ của mình, nhưng chúng ta nên triển khai các dịch vụ để chúng hoạt động. Các lệnh sau đây được dùng để chạy từ thư mục gốc của mã nguồn:

```
1 $ kubectl config set-context $(kubectl config current-context) \
2 --namespace=istioinaction
3 $ kubectl apply -f services/catalog/kubernetes/catalog.yaml
4 $ kubectl apply -f services/webapp/kubernetes/webapp.yaml
```

Khi tất cả các Pod đã sẵn sàng, ta sẽ thấy một thông báo như thế này:

```
1 $ kubectl get pod
2 NAME          READY STATUS  RESTARTS  AGE
3 webapp-bd97b9bb9-q9g46  2/2 Running 18      19d
4 catalog-786894888c-8lbk4 2/2 Running 8        6d
```

Xác minh rằng các tài nguyên gateway và VirtualService của ta đã được cài đặt chính xác:

```

1 $ kubectl get gateway
2 NAME      CREATED AT
3 coolstore-gateway 2h
4
5 $ kubectl get virtualservice
6 NAME      GATEWAYS  HOSTS
7 webapp-vs-from-gw ["coolstore-gateway"] ["webapp.istioinaction.io"]
8

```

Bây giờ, hãy thử gọi gateway và xác minh rằng lưu lượng truy cập được phép vào cluster. Hãy nhớ rằng chúng ta đang sử dụng phương pháp Docker Desktop, trong đó Istio gateway có sẵn trên cổng 80 trên máy chủ cục bộ. Nếu bạn đang sử dụng dịch vụ đám mây hoặc dịch vụ Node-Cổng, bạn sẽ cần tìm hiểu địa chỉ IP công khai bên ngoài đó là gì. Ví dụ, chúng ta có một cách để lấy đúng máy chủ cho cổng vào được hiển thị trên bộ cân bằng tải công khai trông như sau:

```

1 $ URL=$(kubectl -n istio-system get svc istio-ingressgateway \
2 -o jsonpath='{.status.loadBalancer.ingress[0].ip}')

```

Khi bạn có một điểm đầu cuối chính xác, bạn có thể chạy thứ gì đó tương tự như thế này (hãy nhớ rằng localhost nằm trên Docker Desktop):

```

1 $ curl http://localhost/api/catalog

```

Ta sẽ không thấy phản hồi. Tại sao vậy? Nếu chúng ta xem xét kỹ hơn lời gọi bằng cách in ra các tiêu đề, chúng ta sẽ thấy rằng tiêu đề Máy chủ mà chúng ta đã gửi không phải là máy chủ mà cổng nhận ra:

```

1 $ curl -v http://localhost/api/catalog
2
3 * Trying ::1...
4 *
5 * TCP_NODELAY set
6 Connected to localhost (::1) port 80 <-- Host
7 > GET /api/catalog HTTP/1.1
8 > Host: localhost
9 > User-Agent: curl/7.54.0
10 > Accept: */*
11 >
12 <
13 HTTP/1.1 404 Not Found <-- Not found
14 < date: Tue, 21 Aug 2018 16:08:28 GMT
15 < server: envoy
16 < content-length: 0
17 <
18 * Connection #0 to host 192.168.64.27 left intact
19

```

Cả Istio gateway lẫn bất kỳ quy tắc định tuyến nào mà chúng tôi đã khai báo trong Virtual services đều không biết gì về Máy chủ: localhost:80, nhưng nó biết về máy chủ ảo webapp.istioinaction.io. Hãy ghi đè tiêu đề Máy chủ trên dòng lệnh của chúng ta và sau đó lệnh gọi sẽ hoạt động:

```

1 $ curl http://localhost/api/catalog -H "Host: webapp.istioinaction.io"

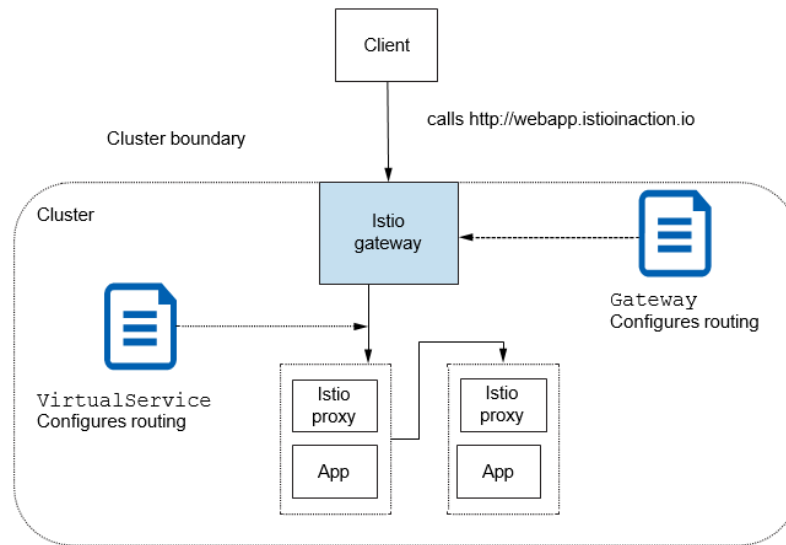
```

Lúc này ta sẽ nhận được một phản hồi thành công

## Tổng quan về luồng lưu lượng

Trong các phần trước, chúng ta đã thực hành với các tài nguyên Gateway và Virtual Services từ Istio. Tài nguyên gateway xác định các cổng, giao thức và máy chủ ảo mà

chúng tôi muốn lắng nghe ở rìa cụm lưới dịch vụ của chúng tôi. Tài nguyên Virtual Services xác định nơi lưu lượng truy cập sẽ đi sau khi được cho phép ở biên. Hình 2.14 cho thấy toàn bộ luồng từ đầu đến cuối.



Hình 2.14: Luồng lưu lượng từ máy khách bên ngoài lưới/cụm dịch vụ đến các dịch vụ bên trong lưới dịch vụ thông qua gateway

## 2.2.2 Định tuyến trong Istio

Trong phần trước, chúng ta đã sử dụng Istio để kiểm soát lưu lượng truy cập vào dịch vụ catalog. Chúng ta đã sử dụng tài nguyên Istio VirtualService để chỉ định cách định tuyến lưu lượng. Chúng ta hãy xem xét kỹ hơn cách thức hoạt động của nó. Chúng ta sẽ kiểm soát lộ trình của một yêu cầu dựa trên nội dung của nó (bằng cách theo dõi các tiêu đề của nó). Bằng cách này, chúng tôi có thể cung cấp mô hình triển khai cho một số người dùng nhất định bằng kỹ thuật gọi là dark launch. Trong lần dark launch, một tỷ lệ lớn người dùng được chuyển đến phiên bản dịch vụ đã biết, trong khi một số lớp người dùng nhất định được chuyển đến phiên bản mới hơn. Do đó, chúng tôi có thể hiển thị chức năng mới theo cách có kiểm soát cho một nhóm cụ thể mà không ảnh hưởng đến những người khác.

Trước khi bắt đầu, hãy dọn dẹp môi trường của chúng ta để chúng ta có thể bắt đầu từ một môi trường mới. Nếu bạn chưa ở trong không gian tên istioaction trong cụm Kubernetes của mình, hãy chuyển sang không gian tên istioaction như sau:

```
1 $ kubectl config set-context $(kubectl config current-context) \  
2 --namespace=istioaction
```

Sau đó xóa toàn bộ các tài nguyên:

```
1 $ kubectl delete deployment,svc,gateway,\  
2 virtualservice,destinationrule --all -n istioaction
```

### Triển khai phiên bản đầu tiên của dịch vụ

Hãy triển khai version 1 của dịch vụ catalog của chúng tôi. Từ thư mục gốc của mã nguồn, hãy chạy lệnh sau:

```
1 $ kubectl apply -f services/catalog/kubernetes/catalog.yaml
```



```
serviceaccount/catalog created
service/catalog created deployment.extensions/catalog created
```

Đợi một vài phút để các tiến trình khởi động. Bạn có thể xem quá trình tiến trình hoạt động bằng lệnh sau:

```
1 $ kubectl get pod -w
```

```
NAME READY STATUS RESTARTS AGE
catalog-98cfcf4cd-xnv79 2/2 Running 0 33s
```

Tại thời điểm này, chúng ta chỉ có thể truy cập dịch vụ catalog từ bên trong cụm. Chạy lệnh sau để xác minh rằng chúng ta có thể truy cập dịch vụ catalog và dịch vụ đó phản hồi chính xác:

```
1 $ kubectl run -i -n default --rm --restart=Never dummy \
2 --image=curlimages/curl --command -- \
3 sh -c 'curl -s http://./catalog.istioinaction/items'
```

Bây giờ, hãy hiển thị dịch vụ catalog cho các máy khách đặt bên ngoài cụm. Chúng ta sử dụng Istio Gateway để thực hiện việc này (lưu ý rằng miễn chúng ta đang sử dụng là catalog.istioinaction.io):

```
apiVersion: networking.istio.io/v1alpha3 kind: Gateway
metadata:
name: catalog-gateway spec:
selector:
istio: ingressgateway servers:
- port:
number: 80 name: http protocol: HTTP
hosts:
- "catalog.istioinaction.io"
```

Chạy câu lệnh sau:

```
1 $ kubectl apply -f ch5/catalog-gateway.yaml
```

```
gateway.networking.istio.io/catalog-gateway created
```

Tiếp theo, chúng ta cần tạo một tài nguyên VirtualService để định tuyến lưu lượng truy cập đến dịch vụ catalog của chúng ta. Tài nguyên VirtualService sẽ trông như thế này:

```
apiVersion: networking.istio.io/v1alpha3 kind: VirtualService
metadata:
name: catalog-vs-from-gw spec:
hosts:
- "catalog.istioinaction.io" gateways:
- catalog-gateway http:
- route:
- destination: host: catalog
```

Khởi tạo VirtualServices:

```
1 $ kubectl apply -f ch5/catalog-vs.yaml
```

```
virtualservice.networking.istio.io/catalog-vs-from-gw created
```

Bây giờ chúng ta có thể truy cập dịch vụ catalog từ bên ngoài cụm bằng cách gọi vào Istio gateway. Chúng ta đang sử dụng Docker Desktop với Istio gateway trên localhost:80, vì vậy chúng ta có thể chạy lệnh sau:

```
1 $ curl http://localhost/items -H "Host: catalog.istioinaction.io"
```

Ta sẽ thấy kết quả giống như khi chúng ta gọi dịch vụ từ bên trong cụm. Trong trường hợp này, chúng ta sẽ đi qua gateway và gọi dịch vụ catalog từ bên ngoài cụm (xem hình 2.15).



Hình 2.15: Gọi danh mục dịch vụ trực tiếp thông qua gateway

## Triển khai phiên bản cập nhật của dịch vụ

Để xem các tính năng kiểm soát lưu lượng của Istio, hãy triển khai phiên bản thứ 2 của dịch vụ catalog. Lệnh này giả định rằng bạn đang ở thư mục gốc của thư mục mã nguồn:

```
1 $ kubectl apply -f services/catalog/kubernetes/catalog-deployment-v2.yaml
```

```
deployment.extensions/catalog-v2 created
```

Liệt kê các Pod trong cụm của bạn:

```
1 $ kubectl get pod
```

```
NAME READY STATUS RESTARTS AGE
catalog-98cfcf4cd-xnv79 2/2 Running 0 14m
catalog-v2-598b8cfbb5-6vw84 2/2 Running 0 36s
```

Nếu bạn yêu cầu gọi dịch vụ catalog nhiều lần, một số phản hồi sẽ có trường bổ sung. phản hồi v2 có một trường được gọi là imageUrl, trong khi phản hồi v1 thì không:

```
1 $ for i in {1..10}; do curl http://localhost/items \
2 -H "Host: catalog.istioinaction.io"; printf "\n\n"; done
```

```
1 [
2 {
3   "id": 0,
4   "color": "teal",
5   "department": "Clothing",
6   "name": "Small Metal Shoes",
7   "price": "232.00",
8   "imageUrl": "http://lorempixel.com/640/480"
9 }
10 ]
```

```

11 [
12   {
13     "id": 0,
14     "color": "teal",
15     "department": "Clothing",
16     "name": "Small Metal Shoes",
17     "price": "232.00"
18   }
19 ]

```

## Định tuyến các lưu lượng cụ thể đến v2

Có thể chúng ta muốn định tuyến bất kỳ lưu lượng truy cập nào bao gồm tiêu đề HTTP `x-istio-cohort: internal` tới v2 của catalog. Chúng ta có thể cấu hình định tuyến các yêu cầu này trong Istio Virtual Service như sau:

```

1  apiVersion: networking.istio.io/v1alpha3 kind: VirtualService
2  metadata:
3  name: catalog-vs-from-gw spec:
4  hosts:
5  - "catalog.istioinaction.io" gateways:
6  - catalog-gateway http:
7  - match:
8  - headers:
9  x-istio-cohort: exact: "internal"
10 route:
11 - destination: host: catalog
12   subset: version-v2
13 - route:
14 - destination: host: catalog
15   subset: version-v1

```

Cập nhật Virtual Services:

```

1 $ kubectl apply -f ch5/catalog-vs-v2-request.yaml

```

`virtualservice.networking.istio.io/catalog-vs-from-gw` configured

Khi chúng ta gọi dịch vụ của mình, chúng ta vẫn thấy phản hồi v1. Tuy nhiên, nếu chúng ta gửi yêu cầu có tiêu đề `x-istio-cohort`, chúng ta sẽ được chuyển đến phiên bản 2 của dịch vụ catalog và chờ phản hồi thể hiện trong hình 2.16 như dự kiến:

```

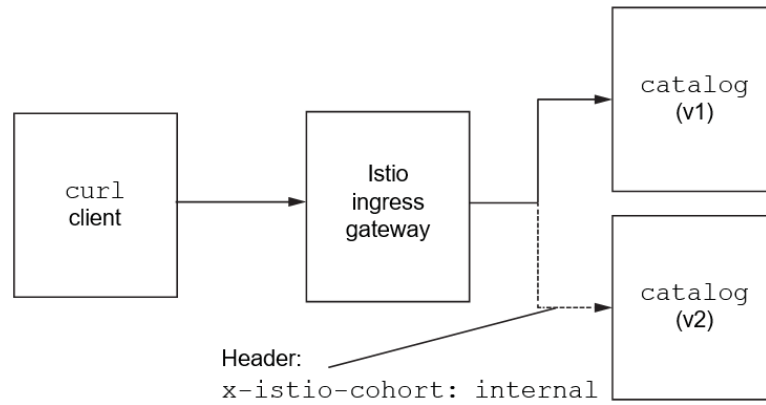
1 $ curl http://localhost/items \
2 -H "Host: catalog.istioinaction.io" -H "x-istio-cohort: internal"

```

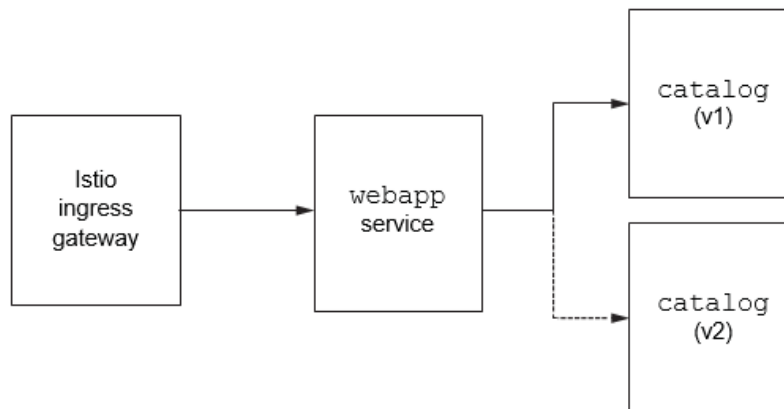
## Định tuyến sâu bên trong biểu đồ cuộc gọi

Cho đến thời điểm này, chúng ta đã thấy cách ta có thể sử dụng Istio để thực hiện định tuyến các yêu cầu, nhưng chúng ta đã thực hiện định tuyến từ rìa/gateway. Các quy tắc lưu lượng này cũng có thể được áp dụng sâu bên trong biểu đồ lời gọi (xem hình 2.17). Chúng ta đã làm điều này trước đó, vì vậy hãy tạo lại quy trình và xác minh rằng nó hoạt động như mong đợi:

**Note** Khả năng định tuyến của Istio bắt nguồn từ khả năng của Envoy. Đối với định tuyến theo yêu cầu cụ thể, các nhóm có thể chọn sử dụng các tiêu đề được đưa vào ứng dụng, như chúng ta thấy trong ví dụ này về việc sử dụng `x-istio-cohort` hoặc dựa vào các tiêu đề đã biết như `Agent` hoặc giá trị từ cookie. Trong thực tế, bạn cũng có thể sử dụng các công cụ quyết định để chọn những tiêu đề nào sẽ được đưa vào và sau đó cho ra các yêu cầu định tuyến.



Hình 2.16: Định tuyến cho các yêu cầu có nội dung nhất định



Hình 2.17: Định tuyến cho các yêu cầu có nội dung cụ thể trong biểu đồ lời gọi

### 2.2.3 Giải quyết các vấn đề về mạng trong Microservices

Khi chúng ta có lưu lượng truy cập vào cụm của mình thông qua Istio gateway, chúng ta có thể điều khiển lưu lượng truy cập ở cấp độ yêu cầu và kiểm soát chính xác nơi định tuyến các yêu cầu. Trong chương trước, chúng ta đã đề cập đến kiểm soát lưu lượng cho định tuyến có trọng số, định tuyến dựa trên kết hợp yêu cầu và một số kiểu mẫu phát hành nhất định mà sau đó có thể được kích hoạt. Chúng ta cũng có thể sử dụng khả năng điều khiển lưu lượng này để định tuyến né tránh các sự cố trong trường hợp xảy ra lỗi ứng dụng, phân vùng mạng và các sự cố lớn khác.

Vấn đề với các hệ thống phân tán là chúng thường bị lỗi theo những cách không thể lường trước và chúng ta không thể thực hiện các hành động thay đổi lưu lượng theo cách thủ công. Chúng ta cần một cách để xây dựng các hành động hợp lý tác động vào ứng dụng để chúng có thể tự phản hồi khi gặp sự cố. Chúng ta có thể làm điều đó với Istio, bao gồm thêm timeouts, retries, and circuit breaking mà không phải thay đổi mã nguồn ứng dụng. Trong chương này, chúng ta xem xét cách thực hiện điều này và những tác động đối với phần còn lại của hệ thống.

## Xây dựng khả năng phục hồi cho ứng dụng

Microservices phải được xây dựng với khả năng phục hồi là mối quan tâm hàng đầu. Thế giới nơi mà mọi thứ “just build it so it won’t fail” là không có thật; và khi xảy ra sự cố, chúng ta có nguy cơ bị mất truy cập tất cả các dịch vụ của mình. Khi chúng ta xây dựng các hệ thống phân tán với các dịch vụ giao tiếp qua mạng, chúng ta có nguy cơ tạo ra nhiều điểm lỗi hơn và đối mặt với khả năng xảy ra lỗi nghiêm trọng. Chủ sở hữu dịch vụ nên áp dụng một vài chức năng khôi phục nhất quán trên các ứng dụng và dịch vụ của họ.

Nếu dịch vụ A gọi dịch vụ B, như được mô tả trong hình 2.18 và gặp phải độ trễ trong các yêu cầu được gửi đến các điểm đầu cuối cụ thể của dịch vụ B, thì chúng ta muốn dịch vụ đó chủ động xác định điều này và định tuyến đến các điểm đầu cuối khác, các nơi khả dụng khác hoặc thậm chí các khu vực khác. Nếu dịch vụ B gặp lỗi không liên tục, chúng ta có thể gửi lại một yêu cầu không thành công. Tương tự, nếu chúng ta gặp sự cố khi gọi dịch vụ B, chúng ta có thể tạm dừng lại cho đến khi dịch vụ này có thể khôi phục sau bất kỳ sự cố nào mà nó có thể gặp phải. Nếu chúng tôi tiếp tục sử dụng dịch vụ B (và trong một số trường hợp, tăng tải khi chúng ta gửi lại các yêu cầu), thì chúng ta có nguy cơ làm quá tải dịch vụ. Tình trạng quá tải này có thể ảnh hưởng đến dịch vụ A và bất kỳ điều gì phụ thuộc vào các dịch vụ này và gây ra các lỗi chồng lỗi đáng kể.



Hình 2.18: Dịch vụ A, dịch vụ gọi B, có thể đang gặp sự cố.

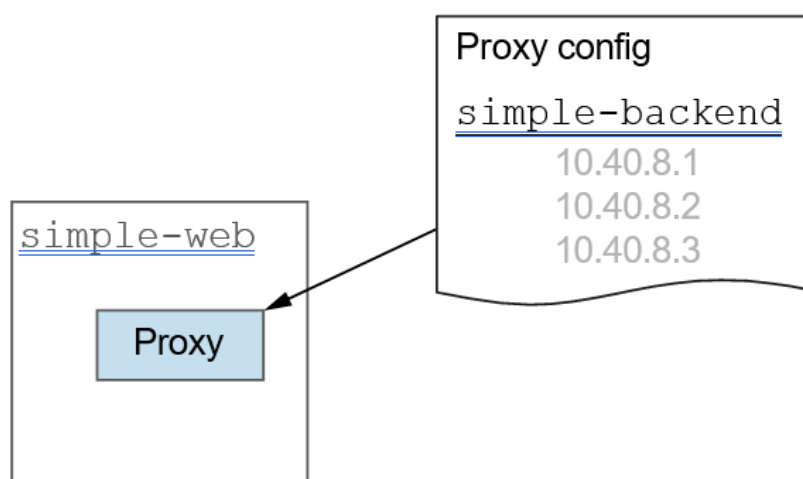
Giải pháp là xây dựng các ứng dụng của chúng tôi để dự kiến các lỗi và có cách để chúng tự động cố gắng khắc phục hoặc quay lại các đường dẫn thay thế khi phục vụ một yêu cầu. Ví dụ: khi dịch vụ A gọi dịch vụ B và bắt đầu gặp sự cố, chúng ta có thể thử gửi lại một yêu cầu, giới hạn thời gian của các yêu cầu hoặc hủy bất kỳ yêu cầu gửi đi nào khác bằng cách sử dụng circuit-breaking. Istio có thể được sử dụng để giải quyết các vấn đề này một cách dễ dàng để các ứng dụng có cách triển khai chính xác và nhất quán cho các vấn đề về khả năng phục hồi bất kể ứng dụng được viết bằng ngôn ngữ lập trình nào.

## Cân bằng tải phía máy khách

Cân bằng tải phía máy khách là hoạt động thông báo cho khách hàng về các điểm đầu cuối khác nhau có sẵn cho một dịch vụ và cho phép khách hàng chọn các thuật toán cân bằng tải cụ thể để phân phối yêu cầu tốt nhất trên các điểm đầu cuối. Điều này làm giảm nhu cầu dựa vào cân bằng tải tập trung, vốn có thể tạo ra các nút cổ chai và điểm gây lỗi, đồng thời cho phép khách hàng thực hiện các yêu cầu trực tiếp đến các điểm đầu cuối cụ thể mà không cần phải thực hiện thêm các bước không cần thiết. Do đó, khách hàng và dịch vụ của chúng ta có thể mở rộng quy mô tốt hơn và đối phó với cấu trúc liên kết đang thay đổi hàng ngày.

Istio sử dụng khả năng tìm kiếm dịch vụ và điểm đầu cuối để trang bị cho proxy phía máy khách trong giao tiếp giữa dịch vụ với dịch vụ thông tin chính xác và cập nhật nhất,

như minh họa trong hình 2.19. Sau đó, nhà phát triển và nhà điều hành dịch vụ có thể cấu hình hành vi cân bằng tải phía máy khách này thông qua cấu hình Istio.



Hình 2.19: Proxy web đơn giản dùng các điểm đầu cuối phụ trợ.

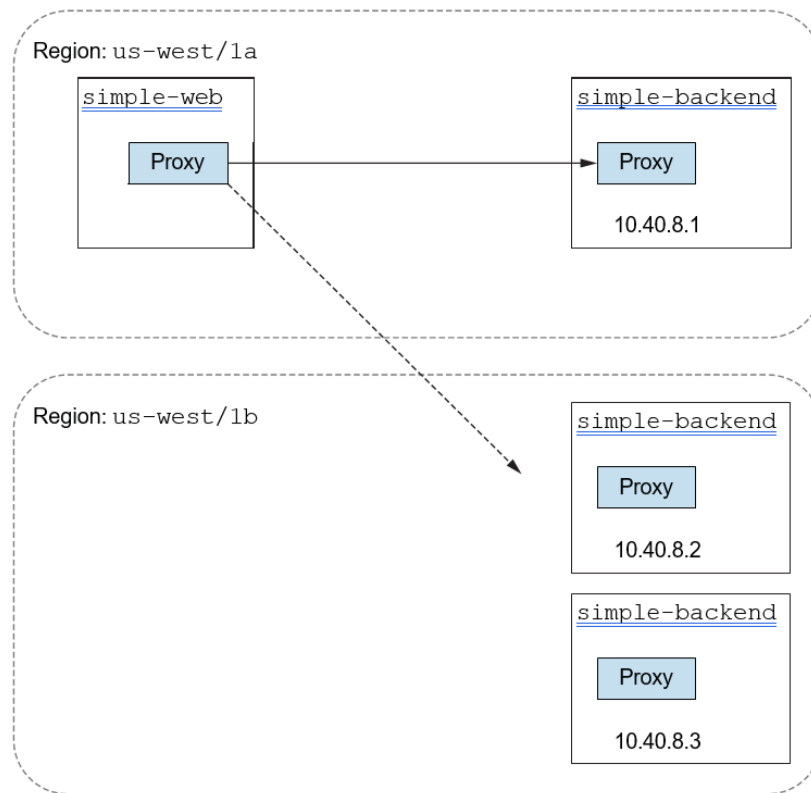
Nhà khai thác và nhà phát triển dịch vụ có thể cấu hình thuật toán cân bằng tải mà máy khách sử dụng bằng cách xác định tài nguyên DestinationRule. Dịch vụ Proxy của Istio dựa trên Envoy và hỗ trợ các thuật toán cân bằng tải của Envoy, một số thuật toán bao gồm:

- Round robin (Mặc định)
- Random
- Weighted least request

### Cân bằng tải nhận biết cục bộ

Một vai trò của control plane như với Istio là nắm được cấu trúc liên kết của dịch vụ và cách cấu trúc liên kết đó có thể phát triển. Một lợi thế của việc nắm được cấu trúc liên kết tổng thể của các dịch vụ trong lưới dịch vụ là tự động đưa ra các quyết định định tuyến và cân bằng tải dựa trên kinh nghiệm như vị trí của dịch vụ và dịch vụ ngang hàng.

Istio hỗ trợ một loại cân bằng tải cung cấp trọng số cho các tuyến và đưa ra các quyết định định tuyến dựa trên vị trí của một khối lượng công việc cụ thể. Ví dụ: Istio có thể xác định khu vực và vùng khả dụng trong đó một dịch vụ cụ thể được triển khai và ưu tiên cho các dịch vụ gần hơn. Nếu các dịch vụ phụ trợ được triển khai trên nhiều khu vực (tây Mỹ, đông Mỹ, tây Âu), thì có nhiều tùy chọn để gọi dịch vụ đó. Nếu một dịch vụ web được triển khai ở khu vực phía tây của chúng ta, chúng tôi muốn các cuộc gọi từ dịch vụ web đến dịch vụ phụ trợ để duy trì cục bộ ở phía tây của chúng ta (xem hình 2.20). Nếu chúng ta đối xử bình đẳng với tất cả các điểm đầu cuối, chúng ta có thể sẽ phải chịu độ trễ cao cũng như chi phí khi chúng ta thực hiện chuyển vùng hoặc khu vực.



Hình 2.20: Ưu tiên các dịch vụ cục bộ

## 2.3 Giám sát các Microservices với Istio

### 2.3.1 Một số Metrics quan trọng của Istio

### 2.3.2 Giám sát các lưu lượng mạng qua Jaeger và Kiali

## 2.4 Bảo mật các Microservices bằng Istio

### 2.4.1 Xác thực giữa các Microservices với Istio

### 2.4.2 Phân quyền cho các Microservices với Istio

## Kết luận chương 2

## Chương 3

# Triển khai Istio trên Kubernetes

3.1 Mô hình triển khai

3.2 Kịch bản triển khai

3.3 Thực nghiệm

3.4 Kết luận