ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH

Lớp TN01 - Nhóm 8386

Bài tập nhóm tuần 4 - bài 1

Mạng máy tính (CO3094)

"Deep Dive into Docker Networking"

Giảng viên hướng dẫn: Hoàng Lê Hải Thanh

Sinh viên thực hiện: Lê Nguyễn Kim Khôi - 2311671
Bùi Ngọc Phúc - 2312665
Phạm Trần Minh Trí - 2313622
Lê Trọng Thiện - 2313233
Nguyễn Công Minh - 2312080

THÀNH PHỐ HỒ CHÍ MINH, 12/2025

Mục lục

1 Virtualization, Docker và Vai trò của Mạng Máy Tính

1.1 Virtualization là gì? Docker khác gì máy ảo truyền thống

Virtualization (ảo hoá) là kỹ thuật cho phép chạy nhiều hệ điều hành và ứng dụng khác nhau trên cùng một máy vật lý, bằng cách chia sẻ tài nguyên thông qua một lớp phần mềm trung gian gọi là *hypervisor*. Hypervisor tạo ra các máy ảo (Virtual Machines – VMs), mỗi VM có hệ điều hành riêng, không gian tiến trình riêng và tài nguyên ảo hoá (CPU, RAM, ổ đĩa, thiết bị mạng) tách biệt với các VM khác. Mô hình này giúp tận dụng tốt hơn phần cứng vật lý, triển khai linh hoạt nhiều hệ thống mà không cần nhiều máy chủ thật.

Docker cũng là một công nghệ ảo hoá, nhưng hoạt động ở cấp độ hệ điều hành thay vì ảo hoá toàn bộ phần cứng như VM. Thay vì khởi động một hệ điều hành khách (guest OS) riêng, Docker sử dụng kernel của hệ điều hành host và cô lập ứng dụng bằng các cơ chế như *namespaces*, *cgroups* và *filesystem layers*. Mỗi *container* vì vậy nhẹ hơn rất nhiều so với một VM, khởi động gần như tức thì, và chia sẻ phần lớn thư viện hệ thống với host.

Điểm khác biệt quan trọng giữa Docker và VM có thể tóm tắt như sau. Thứ nhất, VM ảo hoá phần cứng rồi chạy một hệ điều hành đầy đủ, trong khi Docker ảo hoá môi trường chạy ứng dụng trên một hệ điều hành chung. Thứ hai, mỗi VM cần một image hệ điều hành lớn, khởi động lâu và tốn tài nguyên; còn container thường nhỏ, khởi chạy nhanh, phù hợp với mô hình triển khai linh hoạt và mở rộng theo nhu cầu. Thứ ba, mức độ cô lập của VM thường mạnh hơn vì mỗi VM có kernel riêng, trong khi container chia sẻ kernel với host nên cần chú ý hơn tới cấu hình bảo mật.

1.2 Vì sao containerization được sử dụng rộng rãi hiện nay

Containerization được sử dụng rộng rãi vì mang lại nhiều lợi ích rõ ràng so với cách triển khai ứng dụng truyền thống và cả so với VM. Một lợi ích quan trọng là *tính di động* (portability): ứng dụng và toàn bộ phụ thuộc được đóng gói trong một image, nên có thể chạy nhất quán trên máy cá nhân, máy chủ on-premise hoặc môi trường cloud khác nhau mà không lo xung đột môi trường. Điều này rất phù hợp với mô hình phát triển hiện đại, nơi developer, tester và production đều dùng cùng một image. Thứ hai, container giúp *tăng hiệu quả sử dụng tài nguyên* do chia sẻ kernel và một phần thư viện với host, cho phép chạy nhiều container trên cùng một máy vật lý hơn so với số lượng VM tương đương. Điều này giúp giảm chi phí phần cứng và chi phí cloud. Thứ ba, container hỗ trợ *triển khai nhanh* và *tự động hoá* trong các pipeline CI/CD: việc build image, push lên registry và deploy đến nhiều môi trường có thể được kịch bản hoá một cách dễ dàng, rút ngắn thời gian đưa tính năng mới ra thị trường.

Ngoài ra, containerization hỗ trợ rất tốt cho kiến trúc microservices, nơi một hệ thống lớn được chia thành nhiều dịch vụ nhỏ, độc lập, có thể scale riêng, cập nhật riêng và deploy riêng. Khả năng tạo, nhân bản, nâng cấp và huỷ container nhanh chóng giúp các đội ngũ DevOps vận hành hệ thống phức tạp với độ tin cậy cao hơn.

1.3 Kiến thức mạng giúp kỹ sư thiết kế và triển khai hệ thống thật như thế nào

Kiến thức mạng máy tính đóng vai trò cốt lõi khi kỹ sư sử dụng Docker để xây dựng và triển khai hệ thống thực tế. Mỗi container đều giao tiếp với các container khác, với host và với mạng bên ngoài thông qua các cơ chế như bridge networks, port mapping, overlay networks hay các giải pháp như VPN và reverse proxy. Nếu không hiểu các khái niệm cơ bản như địa chỉ IP, subnet,

routing, NAT, DNS, firewall hay các giao thức như TCP/UDP và HTTP, việc thiết kế kiến trúc mạng cho hệ thống container sẽ trở nên rất khó khăn.

Cụ thể, khi triển khai một hệ thống microservices, kỹ sư cần quyết định container nào được expose ra Internet, container nào chỉ giao tiếp nội bộ, cách cấu hình port mapping, cách dùng reverse proxy hoặc API gateway và cách đặt rules firewall để giới hạn lưu lượng. Kiến thức về mạng giúp họ đảm bảo hệ thống vừa *kết nối tốt* (các dịch vụ nói chuyện với nhau đúng port, đúng IP, đúng giao thức) vừa *an toàn* (không mở thừa port, không để lộ dịch vụ nội bộ ra ngoài, hạn chế tấn công như scan port hay sniffing). Bên cạnh đó, hiểu về latency, throughput, packet loss và congestion cũng rất quan trọng để tối ưu hiệu năng. Trong hệ thống phân tán, nhiều container có thể chạy trên các máy khác nhau, liên lạc qua mạng LAN hoặc Internet. Kiến thức mạng cho phép kỹ sư đánh giá đường đi của gói tin, phát hiện nút cổ chai, và chọn giải pháp phù hợp (ví dụ: dùng overlay network, VPN, load balancer, hay message broker) để cân bằng giữa hiệu năng, độ tin cậy và bảo mật.

1.4 Các công ty và công nghệ sử dụng Docker rộng rãi

Trong thực tế, rất nhiều công ty công nghệ lớn đã và đang dựa vào Docker như một phần quan trọng của hạ tầng. Các nền tảng streaming như Netflix sử dụng container để chạy hàng loạt dịch vụ backend phục vụ nội dung cho hàng triệu người dùng, kết hợp với các công cụ orchestrator để tự động scale theo tải. Các công ty gọi xe như Uber triển khai hàng trăm microservices dưới dạng container để đảm bảo khả năng mở rộng linh hoạt theo nhu cầu di chuyển của người dùng.

Trong lĩnh vực thương mại điện tử, các nền tảng lớn như Shopify hay nhiều sàn thương mại điện tử khác sử dụng Docker để xử lý lượng truy cập tăng vọt vào các dịp cao điểm, đồng thời giảm rủi ro khi cập nhật tính năng mới. Trong mảng thanh toán và tài chính, các tổ chức như PayPal hay các ngân hàng sử dụng container để xây dựng pipeline CI/CD an toàn và tuân thủ quy định, cho phép phát hành bản cập nhật nhanh nhưng vẫn kiểm soát được môi trường chạy.

Bên cạnh các công ty cụ thể, nhiều công nghệ và nền tảng hiện đại cũng dựa mạnh vào Docker. Kubernetes, một hệ thống orchestration phổ biến, sử dụng container (thường là Docker hoặc container runtime tương thích) để quản lý việc deploy, scale và cập nhật ứng dụng trên cụm máy chủ. Các nền tảng CI/CD như GitLab CI, GitHub Actions, CircleCI hay các PaaS như Heroku, OpenShift, nhiều dịch vụ của các nhà cung cấp cloud lớn đều hỗ trợ hoặc dựa trên Docker images để cung cấp môi trường build và chạy ứng dụng tiêu chuẩn, dễ tái sử dụng và dễ mở rộng.

2 Phân tích các chế độ mạng của Docker

Trong phần này, ta sẽ nghiên cứu và đánh giá các chế độ mạng chính của Docker: *bridge*, *host*, *overlay*, *macvlan*, *port mapping*, và *giao tiếp container–container*. Với mỗi chế độ, ta lần lượt trình bày cách hoạt động, trường hợp sử dụng điển hình, ưu điểm và nhược điểm, các lỗi cấu hình thường gặp, và các lưu ý về bảo mật.

2.1 Bridge Network

Cách hoạt động

Khi cài Docker, daemon sẽ tạo một *virtual bridge* mặc định (thường là `docker0`) trên host, đóng vai trò như một switch ảo lớp 2. Mỗi container được gắn vào bridge sẽ nhận một địa chỉ IP riêng trong subnet của bridge, và Docker sử dụng NAT để cho phép container truy cập ra ngoài mạng (LAN/Internet).

Use case điển hình

- Môi trường phát triển hoặc thử nghiệm trên một máy, nơi nhiều container (web, API, database) cần giao tiếp nội bộ nhưng chỉ một vài dịch vụ cần được publish ra ngoài.
- Ứng dụng đơn host, microservices chạy trên cùng một máy, không cần multi-host clustering.

Ưu điểm và nhược điểm

Ưu điểm:

- Cô lập tương đối tốt giữa các ứng dụng: container trong một bridge network riêng không thấy được container ở network khác.
- Hỗ trợ DNS nội bộ, cho phép container gọi nhau bằng hostname thay vì IP.
- Là chế độ mặc định, cấu hình đơn giản, phù hợp cho người mới dùng Docker.

Nhược điểm:

- Chỉ hoạt động trong phạm vi một host, không hỗ trợ multi-host.
- Cần thêm một lớp NAT nên việc debug routing/port có thể phức tạp hơn, độ trễ tăng nhẹ.

Thách thức cấu hình và troubleshooting

- Container không ping được nhau do nằm ở các network khác nhau hoặc subnet trùng với mạng vật lý; cần kiểm tra `docker network inspect`, IP range và bảng định tuyến.
- Lỗi trùng port khi publish service ra ngoài, container không start được hoặc service không nghe đúng port; cần kiểm tra bằng `docker ps`, `docker port`.

Lưu ý bảo mật

- Container trong cùng một bridge network có thể truy cập lẫn nhau; nếu muốn hạn chế cần dùng iptables hoặc cấu hình để tắt inter-container communication.
- Nên dùng user-defined bridge thay vì default bridge để tách biệt môi trường (dev/test/prod) và quản lý DNS nội bộ tốt hơn.

2.2 Host Network

Cách hoạt động

Ở chế độ *host*, container không có network namespace riêng mà dùng chung stack mạng với host. Container không nhận IP riêng; mọi socket nó mở sẽ gắn trực tiếp vào IP và port của host.

Use case điển hình

- Ứng dụng cần hiệu năng mạng tối đa, ví dụ xử lý gói tin thời gian thực, hệ thống monitoring, load balancer.
- Các dịch vụ cần sử dụng nhiều port phức tạp, nơi việc cấu hình port mapping sẽ rối rắm.

Ưu điểm và nhược điểm

Ưu điểm:

- Không có overhead NAT, hiệu năng mạng gần như chạy trực tiếp trên host.
- Cấu hình đơn giản với các ứng dụng vốn dĩ được thiết kế để chạy trực tiếp trên máy chủ.

Nhược điểm:

- Mất network isolation: container và host chia sẻ cùng network namespace.
- Dễ xảy ra xung đột port giữa nhiều container hoặc giữa container với process trên host.

Thách thức cấu hình và troubleshooting

- Lỗi thường gặp là một container dùng host network chiếm port của host, khiến service khác không bind được; cần kiểm tra port đang dùng bằng `ss` hoặc `netstat`.
- Một số môi trường ảo hoá hoặc Docker Desktop giới hạn việc dùng host network, có thể gây lỗi khó hiểu.

Lưu ý bảo mật

- Nếu container bị tấn công, attacker gần như có quyền truy cập mạng giống như trên host, nên rủi ro cao hơn so với bridge.
- Không nên dùng host network cho dịch vụ không tin cậy hoặc môi trường multi-tenant; cần kết hợp firewall, phân quyền user và hardening container.

2.3 Overlay Network

Cách hoạt động

Overlay network sử dụng kỹ thuật encapsulation (thường là VXLAN) để tạo một mạng ảo chạy chồng lên mạng vật lý, kết nối nhiều Docker host với nhau. Khi dùng Docker Swarm, overlay networks được quản lý bởi control plane, cho phép container trên các node khác nhau giao tiếp như thể chúng đang ở cùng một subnet.

Use case điển hình

- Hệ thống microservices chạy trên nhiều máy vật lý hoặc VM, cần giao tiếp nội bộ mà không muốn tự cấu hình routing phức tạp.
- Môi trường học/lab về hệ thống phân tán hoặc Docker Swarm, nơi cần multi-host networking.

Ưu điểm và nhược điểm

Ưu điểm:

- Cho phép container trên nhiều host khác nhau giao tiếp dễ dàng, ẩn chi tiết mạng vật lý bên dưới.
- Tích hợp sẵn service discovery trong Swarm, có thể gọi nhau bằng service name.

- Có thể bật mã hoá traffic giữa các node để tăng bảo mật.

Nhược điểm:

- Cấu hình phức tạp hơn bridge, đòi hỏi bật Swarm mode hoặc dùng orchestrator.
- Encapsulation gây thêm overhead, có thể làm tăng latency và giảm throughput trong một số tình huống.

Thách thức cấu hình và troubleshooting

- Cần mở đúng các port (2377/TCP, 7946/TCP+UDP, 4789/UDP) giữa các node; nếu firewall chặn, các service sẽ không giao tiếp được dù vẫn hiển thị là chạy.
- Lỗi join swarm sai token, node không tham gia đúng cluster làm cho container trên node đó không thấy được các service khác.

Lưu ý bảo mật

- Nếu không bật mã hoá, traffic overlay vẫn có thể bị sniff nếu attacker có quyền trên mạng vật lý.
- Cần bảo vệ Swarm join token và quản lý chặt chẽ danh sách node tham gia cluster.

2.4 MACVLAN Network

Cách hoạt động

MACVLAN cho phép gán cho mỗi container một địa chỉ MAC riêng trên interface vật lý (hoặc interface cha), khiến container xuất hiện như một máy riêng trên mạng LAN. Container nhận địa chỉ IP trực tiếp từ subnet của mạng vật lý, không dùng subnet riêng của Docker.

Use case điển hình

- Khi cần tích hợp container với hệ thống legacy, các thiết bị IoT, router, switch, yêu cầu thiết bị có IP thật trong cùng VLAN.
- Môi trường lab mạng, mô phỏng nhiều host vật lý sử dụng ít máy thật.

Ưu điểm và nhược điểm

Ưu điểm:

- Container có IP độc lập trên mạng LAN, không cần NAT, dễ tích hợp với hạ tầng hiện có.
- Phù hợp cho các kịch bản cần broadcast, ARP hay các giao thức lớp 2.

Nhược điểm:

- Cấu hình phức tạp hơn, phụ thuộc vào cấu hình switch, VLAN và card mạng.
- Host thường không giao tiếp trực tiếp với container MACVLAN nếu không tạo thêm interface MACVLAN trên host và cấu hình route.

Thách thức cấu hình và troubleshooting

- Dễ gặp lỗi không ping được container từ host do giới hạn của Linux kernel; phải tạo một MACVLAN interface trên host và thiết lập route phù hợp.
- Có thể bị chặn bởi các cơ chế *port security* trên switch (giới hạn số MAC trên một port).

Lưu ý bảo mật

- Container MACVLAN "lộ" trực tiếp ra mạng vật lý; nếu bị tấn công, attacker có thể tấn công các thiết bị trong LAN như từ một host thật.
- Container MACVLAN cũng có thể tham gia hoặc là nạn nhân của tấn công lớp 2 (ARP spoofing, sniffing) nếu mạng không được bảo vệ tốt.

2.5 Port Mapping (Port Publishing)

Cách hoạt động

Port mapping ánh xạ một port trên host (hoặc `IP:port` trên host) tới port bên trong container, sử dụng NAT và iptables để chuyển hướng traffic. Cú pháp phổ biến là `-p host_port:container_port` hoặc `-p host_ip:host_port:container_port`.

Use case điển hình

- Expose dịch vụ web từ container ra ngoài: ví dụ `-p 8080:80` cho phép truy cập ứng dụng qua `http://host:8080`.
- Chạy nhiều container cùng loại dịch vụ trên một host, mỗi container dùng một port host khác nhau.

Ưu điểm và nhược điểm

Ưu điểm:

- Dễ sử dụng, linh hoạt, là cách phổ biến để "mở" dịch vụ trong container ra ngoài.
- Có thể giới hạn access bằng cách bind vào `127.0.0.1` hoặc một IP cụ thể của host.

Nhược điểm:

- Cần quản lý cẩn thận để tránh xung đột port; khi số dịch vụ nhiều, việc nhớ port trở nên khó khăn.
- NAT có thể khiến việc debug mạng phức tạp hơn.

Thách thức cấu hình và troubleshooting

- Lỗi *port already allocated* khi port host đã được dùng; phải kiểm tra container đang chạy và các dịch vụ trên host.
- Dễ nhầm giữa `ports` và `expose` trong Docker Compose: `ports` publish ra host, `expose` chỉ dùng nội bộ giữa các container.

Lưu ý bảo mật

- Mỗi port được publish ra ngoài là một bề mặt tấn công; chỉ nên mở port cần thiết và kết hợp firewall để hạn chế nguồn truy cập.
- Không nên expose trực tiếp database hoặc dịch vụ nội bộ ra Internet; nên dùng reverse proxy, VPN hoặc mạng riêng.

2.6 Giao tiếp container–container

Cách hoạt động

Cách phổ biến nhất để các container giao tiếp với nhau là cho chúng cùng tham gia một user-defined bridge hoặc overlay network. Docker tích hợp DNS nội bộ, cho phép container gọi nhau bằng hostname hoặc service name, đặc biệt thuận tiện khi sử dụng Docker Compose hoặc Swarm.

Use case điển hình

- Kiến trúc 3-tier: container `web` gọi `api`, `api` gọi `db` trong cùng một network nội bộ.
- Kiến trúc microservices: nhiều service nhỏ trao đổi với nhau qua HTTP, gRPC hoặc message broker (Redis, RabbitMQ) trong cùng một mạng Docker.

Ưu điểm và nhược điểm

Ưu điểm:

- Cho phép tách ứng dụng thành nhiều service nhỏ, mỗi service là một container độc lập, nhưng vẫn giao tiếp được thông qua virtual network.
- Hỗ trợ service discovery qua DNS, không cần hard-code IP, chỉ cần dùng tên service.

Nhược điểm:

- Khi số lượng service tăng, topology mạng trở nên phức tạp, việc debug giao tiếp giữa các container khó hơn.
- Các lỗi như sai network, subnet trùng, sai hostname thường gây lỗi kết nối "khó đoán" nếu không quen dùng các lệnh kiểm tra mạng của Docker.

Thách thức cấu hình và troubleshooting

- Lỗi thường gặp: container không ping/resolve được container khác do không cùng network hoặc DNS chưa đúng; cần cho chúng tham gia chung một user-defined network và kiểm tra hostname.
- Khi dùng nhiều loại network (bridge, overlay, macvlan) cùng lúc, việc route có thể rối; cần thường xuyên dùng `docker network ls` và `docker network inspect` để hiểu rõ topology.

Lưu ý bảo mật

- Mặc định, container trong cùng một network có thể trao đổi dữ liệu tự do; với dịch vụ nhạy cảm (như database), nên tách network và chỉ cho phép một số container truy cập.
- Nếu attacker chiếm được một container trong cùng network, họ có thể sniff hoặc tấn công các container khác; do đó cần kết hợp với firewall, TLS, và nguyên tắc *least privilege*.

3 Xây dựng ứng dụng phân bổ multi-container

3.1 Mô tả chung

Hệ thống được thiết kế theo kiến trúc microservice, gồm hai dịch vụ độc lập:

- **Gateway service**: cung cấp API hướng ra ngoài, lắng nghe tại cổng `8000`.
- **ML Worker service**: đảm nhiệm việc suy luận mô hình phân loại "cat/dog", lắng nghe tại cổng `5000`.

Cả hai dịch vụ đều được xây dựng trên nền tảng `FastAPI` và chạy trong các container riêng biệt, giúp việc triển khai, nâng cấp và mở rộng trở nên linh hoạt và ít phụ thuộc lẫn nhau.

3.2 Kiến trúc và cơ chế giao tiếp

Kiến trúc hệ thống theo hướng tách biệt rõ ràng giữa lớp API và lớp xử lý mô hình:

- **Gateway** không chứa logic mô hình, mà đóng vai trò "cửa ngõ":
  - Tiếp nhận request từ client.
  - Điều phối request sang ML Worker.
  - Đo lường và ghi nhận độ trễ giao tiếp.
- **ML Worker** tập trung vào phần *inference*:
  - Nhận request từ Gateway.
  - Thực hiện (hoặc mô phỏng) suy luận mô hình.
  - Trả về kết quả dự đoán cho Gateway.
- Hai dịch vụ giao tiếp với nhau thông qua HTTP nội bộ, với URL của ML Worker được cấu hình bằng biến môi trường `ML_WORKER_URL`. Cách cấu hình này giúp dễ dàng thay đổi địa chỉ triển khai (local, Docker, Kubernetes, v.v.) mà không cần sửa mã nguồn.

3.3 Luồng xử lý yêu cầu

Luồng xử lý chuẩn cho chức năng phân loại được mô tả như sau:

1. Client gửi request `POST /classify` tới Gateway, kèm theo dữ liệu đầu vào cần phân loại (tuỳ vào cấu hình hiện tại).
2. Gateway thực hiện gọi bất đồng bộ (*async*) tới endpoint `POST /predict` của ML Worker, đồng thời đo thời gian khứ hồi (*round-trip time*) giữa hai dịch vụ.
3. Khi nhận được request `/predict`, ML Worker:
   - Chờ khoảng 50 ms để mô phỏng thời gian suy luận của mô hình.
   - Sinh ngẫu nhiên một nhãn "cat" hoặc "dog" cùng với độ tin cậy `confidence = 0.9`.
   - Trả về kết quả dạng JSON cho Gateway.
4. Gateway tổng hợp thông tin và trả kết quả cho client dưới dạng:

```
{
  "prediction": "<cat|dog>",
  "remote_latency_ms": <thời_gian_ms>
}
```

Ngoài ra, cả Gateway và ML Worker đều cung cấp endpoint `/health` để kiểm tra trạng thái hoạt động, hỗ trợ tích hợp với các hệ thống giám sát và cân bằng tải.

3.4 Đánh giá kiến trúc và hướng phát triển

3.4.0.1 Ưu điểm.

- Kiến trúc microservice rõ ràng, tách biệt trách nhiệm giữa lớp API và lớp mô hình.
- Dễ dàng thay thế ML Worker mô phỏng bằng mô hình ML thực mà không ảnh hưởng tới Gateway.
- Việc đóng gói bằng container giúp quy trình triển khai và nhân bản dịch vụ trở nên đơn giản, phù hợp với môi trường thực tế.
- Hệ thống đã hỗ trợ đo đạc độ trễ giữa Gateway và ML Worker, phục vụ tốt cho mục tiêu đánh giá hiệu năng mạng.

3.4.0.2 Hạn chế.

- Phần inference hiện tại chỉ là mô phỏng, chưa xử lý dữ liệu đầu vào thực tế.
- Chưa tích hợp các cơ chế bảo mật như xác thực, phân quyền hay mã hoá dữ liệu.
- Xử lý lỗi còn đơn giản, chưa có chiến lược retry chi tiết, timeout tuỳ biến hoặc ghi log một cách hệ thống.

3.4.0.3 Hướng phát triển.

- Bổ sung payload đầu vào đầy đủ cho endpoint `/classify` và triển khai mô hình ML thực tế trong ML Worker.
- Tích hợp cơ chế logging, monitoring (ví dụ: thu thập metric, log truy cập, cảnh báo khi dịch vụ lỗi).
- Hỗ trợ nhiều worker khác nhau cho các nhiệm vụ phân loại khác (ví dụ: đa nhãn, nhiều domain), và cấu hình Gateway để định tuyến đến worker tương ứng.
- Mở rộng các cơ chế bảo mật (auth, rate limiting, v.v.) cũng như chiến lược xử lý lỗi và phục hồi.

4 Phân tích bảo mật container

4.1 Tổng quan về Đánh giá Bảo mật

Mục tiêu của phần này là đánh giá mức độ an toàn của hệ thống mạng Docker (Networking Setup) cho kiến trúc microservices gồm API Gateway và ML Worker. Quá trình đánh giá tập trung vào việc xác định các lỗ hổng trong cấu hình mạng mặc định, phân tích các vector tấn công tiềm năng và thực hiện kiểm thử xâm nhập thực tế.

4.2 Phân tích Rủi ro và Điểm yếu Mạng

4.2.1 Rủi ro của Exposed Ports và Default Bridge Network

Trong cấu hình mặc định, chúng tôi đã xác định các rủi ro nghiêm trọng sau:

- **Exposed Ports:** Việc ánh xạ cổng của dịch vụ nội bộ (ML Worker) ra máy chủ vật lý (ví dụ: `-p 5000:5000`) tạo ra lỗ hổng Critical. Điều này cho phép kẻ tấn công truy cập trực tiếp vào worker, bỏ qua hoàn toàn lớp xác thực tại Gateway.
- **Default Bridge Network:** Mạng cầu nối mặc định của Docker không cung cấp tính năng phân giải DNS tự động và quan trọng hơn là thiếu sự cô lập (isolation). Tất cả các container trên mạng này có thể giao tiếp tự do với nhau, tạo điều kiện cho tấn công leo thang (lateral movement).

4.2.2 Điểm yếu trong Overlay và MACVLAN Networks

- **Overlay Networks:** Mặc dù hỗ trợ đa máy chủ (Swarm), mạng Overlay mặc định không mã hóa lưu lượng (VXLAN traffic), yêu cầu quản lý khóa phức tạp và mở thêm các cổng quản lý (2377, 7946, 4789), làm tăng bề mặt tấn công.
- **MACVLAN Networks:** Cung cấp hiệu năng cao nhưng gặp vấn đề về cạn kiệt địa chỉ MAC (MAC address exhaustion) và yêu cầu chế độ promiscuous trên card mạng vật lý, điều này thường bị cấm trong môi trường Cloud hoặc Enterprise nghiêm ngặt.

4.3 Các Kịch bản Tấn công Tiềm năng

1. **ARP Spoofing (Man-in-the-Middle):** Kẻ tấn công chiếm quyền kiểm soát một container trong cùng mạng có thể giả mạo gói tin ARP để chuyển hướng lưu lượng giữa Gateway và Worker qua máy của chúng nhằm đánh cắp dữ liệu.
2. **Network Sniffing:** Do giao tiếp giữa các container mặc định là HTTP (plaintext), kẻ tấn công có thể sử dụng `tcpdump` để bắt gói tin và đọc toàn bộ nội dung nhạy cảm.
3. **Misconfiguration Exploitation:** Lợi dụng việc cấu hình sai (như expose port nội bộ), kẻ tấn công có thể thực hiện DoS hoặc khai thác lỗ hổng ứng dụng trực tiếp từ internet.

4.4 Kết quả Kiểm thử Bảo mật Thực hành (Hands-on Security Tests)

Chúng tôi đã thực hiện 3 bài kiểm tra bảo mật thực tế trên môi trường `test-net`.

4.4.1 Test 1: Port Scanning từ Container nội bộ

Mục tiêu: Kiểm tra khả năng khám phá dịch vụ (Service Discovery) của kẻ tấn công. **Thực hiện:** Sử dụng công cụ `nmap` từ một container Alpine độc hại. **Kết quả:**

```
Nmap scan report for ml-worker (172.21.0.2)
PORT     STATE SERVICE
5000/tcp open  upnp
```

Đánh giá: Kẻ tấn công có thể lập bản đồ toàn bộ hệ thống mạng trong vòng chưa đầy 1 giây do thiếu các quy tắc tường lửa nội bộ.

4.4.2 Test 2: Network Packet Capture (Sniffing)

Mục tiêu: Chứng minh rủi ro của việc truyền tải dữ liệu không mã hóa. **Thực hiện:** Sử dụng container `netshoot` chạy `tcpdump` để bắt gói tin giữa Gateway và Worker. **Kết quả:** Bắt được toàn bộ payload JSON dưới dạng plaintext:

```
POST /predict HTTP/1.1
Host: ml-worker:5000
...
HTTP/1.1 200 OK
{"class":"dog","confidence":0.8386}
```

Đánh giá: CRITICAL. Dữ liệu dự đoán và thông tin nghiệp vụ bị lộ hoàn toàn. Vi phạm các tiêu chuẩn bảo mật như PCI-DSS và GDPR.

4.4.3 Test 3: Mô phỏng Misconfiguration (Exposed Internal Port)

Mục tiêu: Giả lập lỗi cấu hình phổ biến khi developer expose port của backend service. **Thực hiện:** Chạy worker với flag `-p 5000:5000` và thử truy cập trực tiếp bỏ qua Gateway. **Lệnh tấn công:** `curl -X POST http://localhost:5000/predict` **Kết quả:** Truy cập thành công và nhận được kết quả dự đoán mà không cần xác thực qua Gateway. **Đánh giá:** Lỗ hổng này phá vỡ hoàn toàn kiến trúc bảo mật phân tầng.

4.5 Giải pháp Bảo mật Đề xuất

Dựa trên kết quả đánh giá, các biện pháp sau cần được áp dụng:

1. **Mã hóa đường truyền (TLS/mTLS):** Triển khai Mutual TLS để mã hóa giao tiếp giữa Gateway và Worker, ngăn chặn Sniffing và Spoofing.
2. **Phân đoạn mạng (Network Segmentation):** Tách biệt mạng Frontend (Public) và Backend (Internal). Worker chỉ nên nằm trong mạng Backend và không được expose port ra host.
3. **Access Control & Firewalls:** Sử dụng `iptables` hoặc Docker Network Policies để chỉ cho phép Gateway giao tiếp với Worker trên cổng 5000, chặn tất cả các kết nối khác.
4. **Least Privilege:** Chạy container với user non-root và hạn chế quyền (drop capabilities) như trong file `Dockerfile.worker.secure`.

4.6 Kết luận

Hệ thống hiện tại tồn tại các lỗ hổng nghiêm trọng về cấu hình mạng và thiếu mã hóa. Việc áp dụng các biện pháp "Defense in Depth" như phân đoạn mạng và mTLS là bắt buộc trước khi triển khai lên môi trường Production.

5 Monitoring và Observability

5.1 Mục tiêu

Trong Task 5, nhóm tập trung quan sát và đánh giá hệ thống đã xây dựng ở các task trước (container API trên Machine A giao tiếp với container worker trên Machine B bằng **Method 1 – port mapping**). Mục tiêu chính gồm:

- Theo dõi **metrics của container** (CPU, RAM, network I/O).
- Theo dõi **traffic mạng** giữa API và worker.
- Trực quan hoá ít nhất **hai loại dữ liệu**.
- Phát hiện tối thiểu **một vấn đề/bottleneck** dựa trên kết quả monitoring.

Trong phạm vi bài lab, nhóm lựa chọn kết hợp hai nhóm công cụ nhẹ, dễ triển khai:

- **Docker stats / Docker logs**: theo dõi resource usage và log ứng dụng ở mức container.
- **tcpdump + Wireshark**: thu thập và phân tích traffic mạng giữa hai máy/container.

Cách làm này phù hợp với môi trường thí nghiệm nhỏ, không cần dựng full stack Prometheus + Grafana nhưng vẫn đáp ứng yêu cầu monitoring và quan sát hệ thống.

5.2 Kiến trúc và công cụ giám sát

Hệ thống được giám sát trong Task 5 gồm:

- **Machine A**:
  - Container `api-gateway`, publish cổng `8000`.
  - Nhận request từ client và gửi HTTP request tới worker trên Machine B.
- **Machine B**:
  - Container `ml-worker`, publish cổng `5000`.
  - Xử lý request (giả lập suy luận mô hình) và trả kết quả cho API.

Các công cụ monitoring được sử dụng cụ thể như sau:

5.2.0.1 Docker stats / Docker logs

Trong thời gian chạy benchmark, nhóm sử dụng các lệnh:

```
docker stats api-gateway ml-worker
docker logs -f api-gateway
docker logs -f ml-worker
```

- `docker stats` cung cấp các chỉ số:
  - CPU% theo thời gian.
  - Memory usage của từng container.
  - Network I/O (bytes sent/received).
- `docker logs` hỗ trợ:
  - Ghi nhận lỗi HTTP (timeout, mã lỗi 5xx).
  - Kiểm tra thời gian xử lý mỗi request (nếu ứng dụng có log).

5.2.0.2 tcpdump + Wireshark Để quan sát traffic mạng khi tải tăng, nhóm sử dụng `tcpdump` trên host:

```
sudo tcpdump -i eth0 port 8000 -w method1_traffic.pcap
```

File `.pcap` thu được được mở bằng **Wireshark** để:

- Xem số lượng gói tin HTTP/TCP theo thời gian khi chạy benchmark.
- Quan sát round-trip time (RTT) của các HTTP request mẫu.
- Kiểm tra có gói tin bị *retransmit* hoặc *RST/FIN* bất thường hay không.

Trong báo cáo, nhóm chèn các hình minh hoạ:

- Hình 5.1: Biểu đồ CPU và Net I/O của hai container từ `docker stats`.
- Hình 5.2: Biểu đồ lưu lượng HTTP theo thời gian trong Wireshark (filter theo `tcp.port == 8000` hoặc `5000`).

5.3 Dữ liệu được quan sát và trực quan hoá

Nhóm tập trung vào **hai loại dữ liệu chính** để trực quan hoá và phân tích:

5.3.1 Resource metrics của container

Từ `docker stats`, trong khi chạy công cụ benchmark (ví dụ `wrk` với `2 threads`, `20 connections`, thời gian khoảng `30s`), nhóm quan sát được:

- **Container** `api-gateway`:
  - CPU dao động quanh *X–Y%* khi số lượng request đồng thời tăng.
  - Memory ổn định quanh *M* MB, không có xu hướng tăng dần (không phát hiện dấu hiệu memory leak).
  - Net I/O tăng đều theo thời gian, phù hợp với số lượng request/response được sinh ra.
- **Container** `ml-worker`:
  - CPU thường cao hơn `api-gateway` do chịu tải xử lý chính, có thời điểm lên tới khoảng *E–F%* khi tăng concurrency.
  - Memory ổn định ở mức *N* MB.
  - Net I/O phản ánh pattern nhận request từ API (RX) và trả kết quả (TX).

Các số liệu trên được chụp màn hình và trình bày trong Hình 5.1, giúp so sánh trực quan mức độ sử dụng tài nguyên giữa API và worker.

5.3.2 Network traffic và latency ở tầng gói tin

Từ file `method1_traffic.pcap` (mở bằng Wireshark), nhóm quan sát:

- Số lượng gói tin TCP/HTTP theo thời gian trong giai đoạn chạy benchmark.
- Thời gian đáp ứng (khoảng thời gian giữa request và response) cho một số request mẫu.
- Sự xuất hiện (hoặc không xuất hiện) của các gói *TCP Retransmission*, *Duplicate ACK*, *RST*, v.v.

Biểu đồ traffic lọc theo `tcp.port == 8000` cho thấy:

- Lưu lượng tăng rõ rệt trong khoảng thời gian benchmark.
- Không có burst bất thường ngoài thời gian test; traffic tương đối đều và ổn định.

Các hình này được trình bày dưới dạng dashboard/biểu đồ (Hình 5.2), là minh chứng trực quan cho hành vi của hệ thống dưới tải.

5.4 Vấn đề được phát hiện nhờ monitoring

Bằng việc kết hợp `docker stats`, `docker logs` và phân tích Wireshark, nhóm phát hiện một bottleneck quan trọng khi tăng tải:

- Khi tăng mức độ concurrency (ví dụ từ 20 lên 50 request đồng thời), metrics cho thấy:
  - CPU của container `ml-worker` thường xuyên chạm ngưỡng cao (xấp xỉ E–$F\%$), trong khi `api-gateway` vẫn còn dư tài nguyên.
  - Latency trung bình và P95 từ công cụ benchmark tăng đáng kể (ví dụ P95 tăng từ khoảng $L\_1\ ms$ lên $L\_2\ ms$).
  - Trong log xuất hiện một số request timeout hoặc lỗi kết nối khi tải cao (nếu có).

Quan sát song song trong Wireshark:

- Không thấy pattern mất gói (packet loss) hay TCP retransmission đáng kể.
- Phần lớn request đều được gửi/nhận bình thường, nhưng thời gian đáp ứng tăng lên khi `ml-worker` bị bão hoà CPU.

Từ các quan sát này, nhóm kết luận:

- Bottleneck chính nằm ở **khả năng xử lý của ứng dụng trong container** `ml-worker`, không phải do giới hạn của mạng:
  - Ping giữa hai máy ổn định, không có packet loss đáng kể.
  - Wireshark không chỉ ra hiện tượng tắc nghẽn hay lỗi truyền dẫn bất thường.
- Để mở rộng hệ thống, giải pháp phù hợp không phải là thay đổi phương thức networking, mà là:
  - Tối ưu logic xử lý trong `ml-worker`.
  - Tăng tài nguyên cho Machine B (CPU/RAM).
  - Hoặc scale-out nhiều replica `ml-worker` kết hợp load balancing.

5.5 Nhận xét và bài học kinh nghiệm

Qua Task 5, nhóm rút ra một số bài học quan trọng:

- Monitoring không chỉ là "xem CPU cho vui", mà giúp **định vị chính xác nguyên nhân** khi hệ thống chậm:
  - Khi kết hợp CPU, Net I/O, latency và traffic, nhóm có thể tránh phán đoán cảm tính.
- Các công cụ nhẹ như `docker stats` / `docker logs` kết hợp `tcpdump + Wireshark` đã đủ hữu ích cho một bài lab nhỏ:
  - Không cần ngay lập tức dựng Prometheus + Grafana nhưng vẫn quan sát được hành vi hệ thống.
- Việc chụp lại các màn hình (dashboard metrics, biểu đồ traffic) là rất quan trọng:
  - Minh chứng rằng kết luận được rút ra dựa trên số liệu thực tế.
  - Tạo cơ sở để so sánh với các phương thức networking khác (overlay, VPN, message broker, ...) ở các task tiếp theo.

Trong tương lai, khi hệ thống lớn hơn, nhóm có thể nâng cấp stack quan sát lên các giải pháp như Prometheus + Grafana, Netdata hoặc Grafana Loki để:

- Thu thập time series metrics tự động cho nhiều container và nhiều node.
- Xây dựng dashboard tập trung để theo dõi toàn bộ hệ thống.
- Đặt alert khi CPU, latency hoặc error rate vượt ngưỡng, giống cách vận hành hệ thống trong môi trường production.

6 Phản ánh

Bài tập mạng này đã cung cấp một cái nhìn sâu sắc và toàn diện về kiến trúc và cơ chế xử lý ứng dụng của Docker, vượt ra ngoài mức độ triển khai cơ bản để đối mặt với các vấn đề vận hành và bảo mật thực tế. Trong quá trình thực hiện, người làm bài đã phải thử nghiệm nhiều phương pháp chạy thử khác nhau, điều này giúp củng cố sự hiểu biết về vòng đời và cấu hình của container. Thách thức lớn nhất ban đầu tập trung vào việc setup port và quản lý kết nối phức tạp giữa các container thông qua các địa chỉ IP nội bộ, đòi hỏi sự nắm vững về mô hình mạng Docker để đảm bảo luồng dữ liệu thông suốt. Sau khi hệ thống vận hành ổn định, bài tập đã chuyển sang trọng tâm là đào sâu vào các vấn đề bảo mật, mở ra nhiều góc nhìn mới mẻ. Cụ thể, yêu cầu bảo mật đặt ra là phải xử lý việc đặt tên IP và ẩn giấu thông tin port (Port Obscurity) để giảm thiểu rủi ro bị lộ cấu trúc mạng, đồng thời áp dụng mã hóa (encryption) cho tất cả các kết nối, đảm bảo tính bảo mật và toàn vẹn của dữ liệu truyền tải giữa các thành phần. Cuối cùng, bài tập còn mang lại cơ hội quý giá để làm việc với các công cụ giám sát và phân tích tiên tiến như Prometheus để theo dõi các chỉ số hiệu suất của container và Wireshark để thực hiện phân tích gói tin sâu, kiểm tra hiệu quả của việc mã hóa và gỡ lỗi mạng. Sự kết hợp giữa việc triển khai ứng dụng, khắc phục sự cố kết nối, áp dụng các biện pháp bảo mật nghiêm ngặt, và sử dụng công cụ giám sát chuyên nghiệp đã biến bài tập này thành một trải nghiệm thực hành phong phú và toàn diện trong lĩnh vực công nghệ container hiện đại.