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*.