

# Chương 5

Câu 1:

## Khái niệm luồng (Thread)

Luồng (thread) là đơn vị nhỏ nhất của sự thực thi trong một tiến trình. Mỗi luồng thực hiện một đoạn mã của chương trình và có thể hoạt động song song với các luồng khác trong cùng một tiến trình. Các luồng cùng chia sẻ không gian địa chỉ bộ nhớ, tài nguyên, và dữ liệu của tiến trình cha, nhưng mỗi luồng vẫn có ngữ cảnh thực thi riêng, gồm bộ đếm chương trình, thanh ghi và stack riêng.

### Sự khác biệt cơ bản giữa luồng và tiến trình

Tiêu chí	Tiến trình (Process)	Luồng (Thread)
Không gian bộ nhớ	Mỗi tiến trình có không gian địa chỉ riêng biệt	Các luồng trong cùng tiến trình chia sẻ cùng không gian địa chỉ
Tài nguyên	Không chia sẻ tài nguyên giữa các tiến trình (trừ khi dùng kỹ thuật IPC)	Các luồng chia sẻ tài nguyên như file, biến toàn cục, vùng nhớ...
Chi phí tạo/switch	Tạo và chuyển đổi tiến trình tốn nhiều tài nguyên và thời gian hơn	Tạo và chuyển đổi luồng nhanh và ít tốn kém hơn
Tính độc lập	Các tiến trình hoạt động độc lập, không ảnh hưởng nhau nếu không có liên kết	Một luồng lỗi có thể ảnh hưởng đến toàn bộ tiến trình
Quản lý bởi	Hệ điều hành quản lý hoàn toàn	Có thể được quản lý bởi hệ điều hành hoặc ứng dụng (user-level threads)
Mức độ cách ly	Cách ly tốt, phù hợp cho tính trong suốt và an toàn	Không cách ly hoàn toàn, dễ chia sẻ dữ liệu hơn nhưng cũng dễ xảy ra lỗi chia sẻ tài nguyên

### Vì sao chia tiến trình thành nhiều luồng giúp cải thiện hiệu năng trên máy tính đa bộ vi xử lý

Trong máy tính đa bộ vi xử lý (multi-core/multi-CPU), nhiều đơn vị xử lý trung tâm (CPU core) có thể thực thi đồng thời nhiều luồng. Khi một tiến trình được chia thành nhiều luồng, các luồng có thể phân tán và chạy song song trên các nhân khác nhau, giúp:

1. Tăng khả năng tận dụng tài nguyên phần cứng: Mỗi luồng chạy trên một nhân CPU riêng → tránh lãng phí năng lực xử lý.

2. Cải thiện hiệu năng tổng thể: Giảm thời gian hoàn thành vì các tác vụ độc lập được xử lý cùng lúc.
3. Giảm độ trễ phản hồi: Một luồng có thể xử lý giao diện người dùng, trong khi luồng khác xử lý tính toán nền.

### Ví dụ minh họa: Microsoft Excel

Trong Microsoft Excel, giả sử người dùng mở một bảng tính lớn và:

- Luồng A: đang tính toán công thức trong hàng ngàn ô.
- Luồng B: tiếp tục cho phép người dùng nhập dữ liệu mới, cuộn trang, hoặc chỉnh sửa định dạng.
- Luồng C: tự động lưu tạm thời (autosave) hoặc kiểm tra lỗi.

Trong hệ thống đa nhân:

- Luồng A có thể chạy trên CPU core 1, Luồng B trên CPU core 2, Luồng C trên CPU core 3 → song song hóa hoàn toàn.
- Như vậy, Excel không bị "đơ" giao diện khi tính toán nặng, người dùng vẫn có thể thao tác mượt mà.


Trong mô hình ứng dụng phân tán kiểu khách-chủ (client-server), mạng là yếu tố gây độ trễ đáng kể và ảnh hưởng đến hiệu năng hệ thống nếu không được xử lý hợp lý. Việc sử dụng luồng (threads) ở cả phía máy khách và máy chủ có thể giúp:

- Che giấu độ trễ mạng (network latency)
- Tăng khả năng xử lý đồng thời (concurrency)

### Phía máy khách (Client-side)

Mục tiêu:

- Không bị treo giao diện hoặc chặn người dùng khi chờ phản hồi từ máy chủ.
- Xử lý đồng thời nhiều yêu cầu mạng.

 Cách sử dụng luồng:

Luồng	Nhiệm vụ
Luồng chính	Giao tiếp với người dùng (UI), nhận yêu cầu nhập liệu.

Luồng mạng (worker thread)	Gửi yêu cầu đến máy chủ, chờ phản hồi mà không chặn UI.
Luồng xử lý phản hồi	Nhận dữ liệu từ server → cập nhật giao diện, hiển thị kết quả.

Ví dụ:

Trong ứng dụng chat:

- Khi người dùng gửi tin nhắn, một luồng phụ đảm nhận việc gửi tin.
- Một luồng khác luôn nghe (listen) phản hồi mới từ server và hiển thị lên giao diện.
- Giao diện luôn mượt mà, không bị đơ dù mạng chậm.

## Phía máy chủ (Server-side)

Mục tiêu:

- Xử lý đồng thời nhiều yêu cầu từ nhiều máy khách.
- Không bị nghẽn hoặc chặn do chờ xử lý mạng hoặc I/O.

Cách sử dụng luồng:

Mô hình	Cách hoạt động
Mỗi kết nối - một luồng (thread-per-connection)	Mỗi máy khách kết nối → server tạo một luồng riêng để xử lý.
Thread pool (bể luồng)	Có sẵn N luồng, khi yêu cầu đến → lấy luồng nhàn rồi xử lý.
Luồng riêng cho I/O	Một luồng chuyên đọc dữ liệu từ socket, một luồng xử lý nội dung.

Ví dụ:

Trong máy chủ HTTP:

- Khi 100 máy khách cùng truy cập:
  - 100 luồng riêng biệt xử lý mỗi yêu cầu GET.
  - Kết quả trả về cho từng client không chậm trễ.

## So sánh hai cách cài đặt luồng

(a) Luồng ở mức người dùng (User-Level Threads - ULT)

Đặc điểm:

- Quản lý luồng hoàn toàn bởi thư viện người dùng (user-space library), kernel không biết các luồng này.
- Mỗi tiến trình chỉ có một đơn vị thực thi ở mức nhân (kernel thread hoặc LWP).

Ưu điểm:

- Chuyển ngữ cảnh rất nhanh vì không cần chuyển vào kernel mode.
- Không phụ thuộc vào hệ điều hành → dễ tùy biến, linh hoạt.
- Có thể chạy ở những hệ thống không hỗ trợ luồng kernel.

Nhược điểm:

- Không thực sự song song trên hệ thống đa nhân (vì chỉ có 1 LWP).
- Bị phong tỏa toàn bộ nếu 1 luồng gọi system call chặn (blocking).
- Phụ thuộc hoàn toàn vào thư viện luồng người dùng → khó tương tác với I/O kernel.

(b) Luồng kết hợp ULT với LWP (Hybrid model)

Đặc điểm:

- Gồm nhiều user threads ánh xạ lên nhiều LWP (lightweight processes).
- Mỗi LWP là 1 kernel thread, do kernel quản lý và lập lịch.

Ưu điểm:

- Tận dụng đa nhân tốt hơn → user threads có thể thực thi song song thực sự.
- Nếu 1 luồng bị block → các luồng khác vẫn chạy được vì có nhiều LWP.
- Cân bằng giữa tính linh hoạt và hiệu năng.

Nhược điểm:

- Chi phí chuyển ngữ cảnh cao hơn do có sự xen kẽ giữa user-level và kernel-level.
- Phức tạp hơn trong triển khai và quản lý ánh xạ giữa ULT và LWP.
- Cần thiết kế hệ thống lịch riêng trong user-space để phối hợp với kernel.

Tiêu chí	User-Level Threads (ULT)	Hybrid (ULT + LWP)
Chi phí chuyển ngữ cảnh	Thấp (do ở user-space)	Trung bình (có kernel-level context switch)
Phong tỏa	Cao (block 1 = block tất cả)	Thấp (block 1 luồng, LWP khác vẫn chạy)
Tận dụng CPU đa nhân	Không	Có
Độ phức tạp triển khai	Thấp	Cao
Khả năng tương thích system call	Kém	Tốt hơn

**Chọn mô hình xử lý trong hệ thống phân tán "read-heavy, write-light"**

Ngữ cảnh:

Hệ thống phục vụ nhiều yêu cầu đọc, ít ghi. Ví dụ: dịch vụ chia sẻ tập tin, CDN, API đọc dữ liệu tĩnh.

So sánh 3 mô hình xử lý:

Tiêu chí	Đa luồng (Multithreaded)	Đơn luồng (Single-threaded)	FSM bất đồng bộ (Event-driven FSM)
Hiệu năng	Tốt nếu có ít context-switch, tận dụng đa nhân	Kém nếu nhiều client	Rất cao nếu dùng tốt non-blocking I/O
Mở rộng (scalability)	Trung bình (bị giới hạn bởi số luồng hoặc LWP)	Tệ (tắc nghẽn khi nhiều client)	Cao (1 luồng xử lý hàng ngàn kết nối với epoll/kqueue)
Độ phức tạp phát triển	Trung bình, dễ debug	Dễ nhất	Phức tạp (phải quản lý trạng thái và callback)
Khả năng chịu lỗi	Nếu 1 luồng lỗi có thể ảnh hưởng tiến trình	Ổn định (vì đơn giản)	Khó debug, dễ bị lỗi logic (race/state bug)

Khuyến nghị mô hình phù hợp: FSM bất đồng bộ (Event-driven FSM)

Vì:

- Đọc là chủ yếu → non-blocking I/O là tối ưu.
- Không cần tạo nhiều luồng hoặc tiến trình → tiết kiệm bộ nhớ.
- Tỷ lệ ghi thấp → tránh được các khó khăn trong xử lý race condition/phân mảnh trạng thái.

Ví dụ thành công: Nginx, Redis, Node.js → đều dùng event loop + FSM để đạt hiệu năng rất cao.

Câu2 :

### Định nghĩa ảo hóa (Virtualization)

Ảo hóa (virtualization) là kỹ thuật tạo ra phiên bản ảo của một tài nguyên thực, như: phần cứng, hệ điều hành, bộ nhớ, thiết bị lưu trữ hoặc mạng. Mục tiêu là cho phép nhiều môi trường hoặc hệ thống độc lập chạy đồng thời trên cùng một tài nguyên vật lý, mà không can thiệp lẫn nhau.

### Vai trò chính của ảo hóa trong hệ thống phân tán

Ảo hóa đóng vai trò quan trọng trong hệ thống phân tán nhờ các lợi ích:

Vai trò	Mô tả
Tối ưu tài nguyên	Cho phép nhiều máy ảo chia sẻ tài nguyên vật lý (CPU, RAM, ổ cứng), giảm lãng phí.
Tách biệt và cô lập	Mỗi máy ảo là một hệ thống độc lập → an toàn, dễ phục hồi khi lỗi.
Linh hoạt và dễ mở rộng	Có thể khởi tạo / di chuyển máy ảo nhanh chóng qua các nút mạng.
Khả năng phục hồi và sao lưu	Snapshots và live migration hỗ trợ high availability.
Triển khai nhanh	Dễ dàng tạo bản sao hệ thống để phát triển, kiểm thử và triển khai.

Phân loại ảo hóa dựa trên lớp giao diện

#### A. Máy ảo tiến trình (Process-level VM)

Ví dụ: JVM (Java Virtual Machine), .NET CLR

Đặc điểm	Mô tả
Giao diện ảo hóa	Instruction set (tập lệnh ảo)
Mục tiêu	Ảo hóa môi trường thực thi cho một ứng dụng duy nhất
Cơ chế hoạt động	Dịch bytecode (mã trung gian) sang mã máy tại runtime hoặc sử dụng JIT compiler.
Tính di động	Cao — chương trình viết một lần chạy trên nhiều hệ điều hành nếu cài đúng VM.

Ví dụ:

- Chạy chương trình Java .class trên JVM: bytecode được chạy trên một VM trừu tượng thay vì trực tiếp trên CPU.

## B. Giám sát máy ảo (Hypervisor-based VM – System-level VM)

Ví dụ: VMware, KVM, Xen, VirtualBox

Đặc điểm	Mô tả
Giao diện ảo hóa	System level (ảo hóa cả hệ điều hành và phần cứng)
Mục tiêu	Cho phép nhiều hệ điều hành độc lập chạy đồng thời trên một máy vật lý
Cơ chế hoạt động	Hypervisor (trình giám sát máy ảo) phân phối tài nguyên phần cứng cho từng máy ảo (VM).
Loại hypervisor	- Type 1: chạy trực tiếp trên phần cứng (bare metal)

- Type 2: chạy bên trong một hệ điều hành chủ (host OS) |

Ví dụ:

- Chạy 3 hệ điều hành khác nhau (Linux, Windows, BSD) đồng thời trên một máy nhờ VirtualBox.

---

### So sánh hai hình thức ảo hóa

Tiêu chí	Máy ảo tiến trình (Process VM)	Giám sát máy ảo (System VM)
Mức độ ảo hóa	Ảo hóa môi trường thực thi chương trình	Ảo hóa toàn bộ hệ điều hành và phần cứng
Phạm vi	Chỉ chạy một ứng dụng	Chạy nhiều hệ điều hành và ứng dụng độc lập
Tính di động	Rất cao (đa nền tảng)	Tùy thuộc vào phần mềm ảo hóa
Tài nguyên tiêu tốn	Nhẹ (ít sử dụng phần cứng)	Nặng hơn (dùng nhiều tài nguyên hơn)
Ví dụ	Java VM, .NET CLR	VMware, KVM, Xen, VirtualBox

Bối cảnh:

Triển khai một hệ thống microservices nhỏ trên một máy chủ vật lý duy nhất.

Các lựa chọn:

Lựa chọn	Mô tả ngắn
(a) Đa container (Docker)	Mỗi service là 1 container chạy độc lập, chia sẻ kernel của host.
(b) Nhiều máy ảo (KVM/VMware)	Mỗi service chạy trên 1 hệ điều hành ảo riêng, có nhân độc lập.
(c) Máy ảo tiến trình (GraalVM native image)	Dịch chương trình thành mã nhị phân chạy độc lập, không cần JVM thời gian chạy.

So sánh theo tiêu chí

Tiêu chí	(a) Docker	(b) Máy ảo (KVM/VMware)	(c) GraalVM native image
Thời gian khởi động	Rất nhanh (vài giây)	Chậm (vài chục giây → phút)	Cực nhanh (tương đương chương trình C/C++)
Mức độ cô lập	Trung bình (chia sẻ kernel)	Cao nhất (kernel, OS tách biệt)	Thấp (chạy như process bình thường)
Tài nguyên tiêu thụ	Nhẹ (chia sẻ layer, không OS riêng)	Rất nặng (mỗi VM cần RAM + OS riêng)	Rất nhẹ (native binary không cần runtime JVM)
Quản lý phức tạp	Trung bình (dùng Docker Compose, volumes...)	Cao (quản lý VM, disk, images...)	Thấp (biên dịch là xong, không cần daemon)

Kết luận & Lựa chọn phù hợp:

→ Lựa chọn khuyến nghị: (a) Cài đặt đa container với Docker

Lý do:

- Thời gian khởi động nhanh hơn máy ảo, phù hợp với microservice có khả năng scale.
- Cô lập hợp lý, đủ an toàn cho nhiều service cùng chia sẻ kernel (nếu bảo mật được cấu hình đúng).
- Tài nguyên tiêu thụ thấp hơn VM, dễ triển khai và bảo trì.
- Hỗ trợ tốt CI/CD, mạng giữa container, volume, log, scale.
- 

Tuy nhiên:



- Nếu bạn triển khai một vài service rất đơn giản, ít yêu cầu cô lập → (c) GraalVM native image có thể là lựa chọn siêu nhẹ và cực nhanh.
- Nếu bạn cần mức độ bảo mật, tách biệt cao cấp hệ điều hành → (b) máy ảo sẽ phù hợp, nhưng là giải pháp nặng và dư thừa cho hệ thống nhỏ.

### So sánh Hypervisor thuần (bare-metal) vs Hypervisor lưu trữ (hosted)

Tiêu chí	Hypervisor thuần (Type 1 - Bare Metal)	Hypervisor lưu trữ (Type 2 - Hosted)
Cách hoạt động	Chạy trực tiếp trên phần cứng, không có hệ điều hành chủ	Chạy trên hệ điều hành chủ, giống như một ứng dụng
Ví dụ	VMware ESXi, Microsoft Hyper-V, Xen, KVM	VMware Workstation, VirtualBox, Parallels

#### a. Hiệu năng I/O và CPU

- Type 1: Tốt hơn → vì không qua lớp hệ điều hành trung gian.
- Type 2: Chậm hơn → vì I/O, syscall phải đi qua hệ điều hành host.

Kết luận: Type 1 phù hợp với các môi trường sản xuất, đòi hỏi hiệu năng cao.

#### b. Chi phí phát triển, vận hành

- Type 1:
  - Hiệu quả cho môi trường quy mô lớn.
  - Cần cấu hình chuyên sâu, khó quản trị nếu không có công cụ quản lý trung tâm (vSphere...).
- Type 2:
  - Dễ cài đặt, phù hợp phát triển, demo, thử nghiệm.
  - Không phù hợp cho vận hành thực tế lâu dài, khó scale.

Kết luận: Type 2 dễ dùng nhưng không phù hợp sản xuất, trong khi Type 1 đòi hỏi đầu tư ban đầu cao hơn nhưng đáng tin cậy hơn.

#### c. Khả năng sử dụng lại driver thiết bị

- Type 1: Khó hơn — cần hỗ trợ thiết bị trực tiếp trong hypervisor.
- Type 2: Dễ hơn — sử dụng driver của hệ điều hành host, tương thích cao.

Kết luận: Type 2 có lợi khi cần hỗ trợ phần cứng đa dạng mà không có driver riêng trong hypervisor.

## Tóm tắt Ưu – Nhược điểm

Loại	Ưu điểm	Nhược điểm
Type 1 (bare-metal) - Hiệu năng cao		
	<ul style="list-style-type: none"><li>• Cách ly tốt</li><li>• Phù hợp sản xuất lớn   - Cấu hình phức tạp</li><li>• Thiếu tính linh hoạt khi thử nghiệm</li><li>• Cần hỗ trợ phần cứng riêng  </li></ul>	
	<ul style="list-style-type: none"><li>  Type 2 (hosted)   - Dễ triển khai</li><li>• Tương thích tốt với driver</li><li>• Thích hợp phát triển, học tập   - Hiệu năng kém</li><li>• Không phù hợp cho môi trường sản xuất</li><li>• Kém bảo mật hơn  </li></ul>	

## 2. Đánh giá phù hợp của Hypervisor truyền thống vs Container trong Public Cloud

Dịch vụ	Mục tiêu chính
(i) PaaS Tự động mở rộng nhanh, triển khai dễ dàng	
(ii) IaaS Cho phép người dùng kiểm soát hệ điều hành, nhân (kernel), module	
(i) PaaS – Nên dùng Container-based Virtualization	
Lý do	Giải thích
Hiệu năng cao	Container chia sẻ nhân OS, tránh overhead của VM.
Khởi động nhanh	Container có thể spin-up trong vài giây, hỗ trợ auto-scaling tốt.
Tính di động	Docker image chạy ổn định trên nhiều nền tảng cloud.

Kết luận: PaaS cần lightweight, dễ scale → container là lựa chọn tối ưu.

(ii) IaaS – Nên dùng Hypervisor truyền thống (VM)	
Lý do	Giải thích
Tùy chỉnh kernel VM cho phép root access, thay đổi nhân, cài module.	
Cách ly mạnh	VM cách ly cấp hệ điều hành → bảo mật hơn container.
Tính ổn định	VM ít bị ảnh hưởng bởi container-level exploit.

Kết luận: IaaS cần sự tách biệt tuyệt đối, linh hoạt kernel → dùng VM là phù hợp.

---

### Trade-off: Container vs Hypervisor

Tiêu chí	Container	Hypervisor/VM
Bảo mật	Kém hơn (dùng chung kernel)	Cách ly tốt hơn
Hiệu năng	Nhanh hơn, ít overhead	Có overhead (guest OS, hypervisor)
Tính di động	Rất cao (Docker image)	Trung bình (phụ thuộc hypervisor)
Tùy chỉnh kernel	Không (dùng kernel của host)	Có (chạy kernel riêng)

Câu 4 :

Định nghĩa máy chủ và hai đặc điểm cơ bản so với máy khách

Máy chủ (server) trong hệ thống phân tán là một thành phần cung cấp dịch vụ cho máy khách (client) thông qua mạng. Máy chủ lắng nghe, tiếp nhận và xử lý các yêu cầu từ máy khách, sau đó phản hồi kết quả phù hợp.

Hai đặc điểm cơ bản của máy chủ so với máy khách:

- Tính phục vụ: Máy chủ luôn trong trạng thái sẵn sàng tiếp nhận và phục vụ nhiều yêu cầu từ nhiều máy khách.
- Tính đồng thời (concurrency): Máy chủ cần khả năng xử lý đồng thời nhiều yêu cầu mà không bị nghẽn hay chặn toàn bộ hệ thống.

## 2. Phân biệt máy chủ tương tác và máy chủ tương tranh

Loại máy chủ	Cách hoạt động	Mô hình xử lý đa luồng	Ví dụ
Máy chủ tương tác (iterative server)	Xử lý tuần tự từng yêu cầu một. Trong khi xử lý một yêu cầu, các yêu cầu khác phải chờ.	Có thể dùng một luồng chính duy nhất.	Server echo đơn giản bằng TCP.
Máy chủ tương tranh (concurrent server)	Mỗi yêu cầu được xử lý trong một luồng hoặc tiến trình riêng → cho phép xử lý đồng thời.	Tạo luồng riêng cho mỗi kết nối hoặc sử dụng thread pool.	Web server phục vụ nhiều client cng lúc.

Ví dụ đa luồng:

- Với máy chủ tương tranh: mỗi khi nhận kết nối mới từ WebSocket, máy chủ tạo 1 luồng (hoặc coroutine) để phục vụ kết nối đó.
- Với máy chủ tương tác: không dùng đa luồng, chỉ xử lý 1 client tại một thời điểm.

---

## 3. Thiết kế máy chủ WebSocket thời gian thực

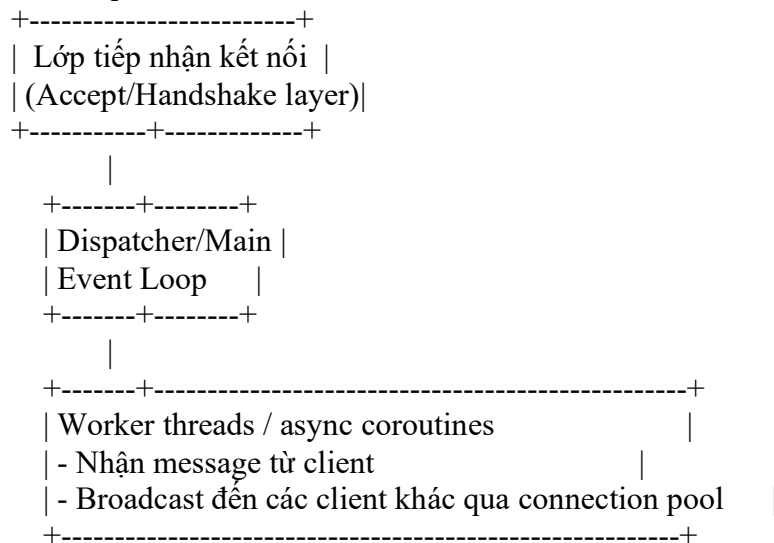
Yêu cầu:

- Nhận và phân phối thông điệp thời gian thực đến nhiều client.
- Luôn sẵn sàng nhận kết nối mới và xử lý dữ liệu song song.

Kiến trúc đề xuất:

pgsql

Sao chépChỉnh sửa



Các thành phần chính:

1. Lớp tiếp nhận kết nối:
  - Lắng nghe cổng (port), xử lý handshake WebSocket.
  - Dùng thư viện như websockets, aiohttp (Python), Socket.io (Node.js), SignalR (.NET).
2. Quản lý kết nối:
  - Lưu trữ tất cả các kết nối đang mở trong một connection pool (dạng map ID → socket).
  - Hỗ trợ phân luồng gửi nhận dữ liệu (I/O) không đồng bộ hoặc đa luồng.
3. Luồng xử lý:
  - Mỗi client gắn với một luồng hoặc coroutine để nhận dữ liệu.
  - Khi có message mới từ một client, máy chủ broadcast đến các client liên quan khác thông qua channel pub-sub hoặc cơ chế chia nhóm (room).

Công nghệ phù hợp:

- Node.js: dùng Socket.io, có event loop hiệu quả.
- Python (asyncio): dùng websockets, FastAPI, Quart.
- Golang: nhẹ, dùng goroutine để quản lý từng client.
- .NET: dùng SignalR.

So sánh hai mô hình điểm truy nhập dịch vụ

### (a) Gắn cổng cố định (well-known port)

- Ưu điểm:
  - Đơn giản, dễ triển khai.
  - Máy khách dễ kết nối vì địa chỉ dịch vụ đã biết (ví dụ: HTTP dùng port 80, SSH dùng port 22).
- Nhược điểm:
  - Thiếu linh hoạt khi muốn chạy nhiều phiên bản dịch vụ cùng loại trên một máy.
  - Dễ bị tấn công (scanning ports, DOS) do cổng cố định dễ đoán.
  - Giới hạn số lượng dịch vụ (chỉ 65535 cổng TCP/UDP).
- Tính linh hoạt: thấp.
- Khả năng mở rộng: kém, khó quản lý nhiều phiên bản hoặc microservices trên cùng máy chủ.
- Bảo mật: dễ bị tấn công do cổng lộ diện.

### (b) Daemon trung gian (multiplexing)

- Cơ chế: một tiến trình daemon lắng nghe trên cổng duy nhất (ví dụ: inetd, xinetd) và phân phối yêu cầu đến dịch vụ tương ứng.
- Ưu điểm:
  - Dễ mở rộng: có thể dùng 1 cổng phục vụ nhiều dịch vụ.
  - Cấu hình linh hoạt hơn, cho phép khởi động dịch vụ theo yêu cầu.
  - Giảm nguy cơ tấn công dịch vụ không cần thiết (do chưa khởi động).
- Nhược điểm:
  - Tăng độ trễ do qua lớp trung gian.
  - Daemon trung gian là điểm đơn lỗi (single point of failure).
- Tính linh hoạt: cao.
- Khả năng mở rộng: tốt hơn, nhất là khi dùng nhiều dịch vụ cùng lúc.
- Bảo mật: tốt hơn nếu daemon cấu hình đúng, nhưng vẫn có rủi ro nếu bị tấn công.

---

## 2. So sánh kiến trúc lưu trữ dịch vụ

### (i) Cụm máy chủ tập trung (cluster LAN)

- Hiệu năng truy cập: cao trong nội bộ LAN do độ trễ thấp, băng thông cao.
- Khả năng chịu lỗi: phụ thuộc vào độ dư thừa và clustering. Một sự cố điện/lỗi vùng có thể làm sập toàn bộ.
- Chi phí đầu tư: cao do cần máy cấu hình mạnh, hệ thống mạng nội bộ chất lượng cao.
- Độ phức tạp vận hành: đơn giản hơn do tất cả trong 1 trung tâm dữ liệu.
- Phù hợp: hệ thống nhỏ - trung bình, có thể kiểm soát được truy cập nội bộ.

(ii) Máy chủ phân tán địa lý (geo-distributed nodes)

- Hiệu năng truy cập: tốt cho người dùng toàn cầu vì dữ liệu gần người dùng (giảm latency).
- Khả năng chịu lỗi: cao, nếu một vùng bị lỗi, vùng khác vẫn hoạt động (resilience).
- Chi phí đầu tư: có thể thấp hơn nếu dùng máy cấu hình vừa phải, nhưng chi phí vận hành (kết nối, đồng bộ, giám sát) sẽ tăng.
- Độ phức tạp vận hành: cao do yêu cầu đồng bộ dữ liệu, xử lý phân tán, giám sát đa vùng.
- Phù hợp: hệ thống lớn, cung cấp dịch vụ toàn cầu, cần độ sẵn sàng cao.