NGUYỄN THẾ HUY

SYSTEM DESIGN

CHO NGƯỜI MỚI BẮT ĐẦU

NHỮNG KIẾN THỨC CƠ BẢN GIÚP
BẠN PHỎNG VẤN HIỆU QUẢ

https://huynt.dev

Lời mở đầu

Chào bạn, và cám ơn bạn đã đọc ebook này của mình.

Mình là Huy, hiện tại mình là một Technical Leader với trên 10 năm kinh nghiệm.

Ebook này của mình nhằm cung cấp cho bạn những khái niệm cốt lõi, cũng như kinh nghiệm tham gia phỏng vấn System Design (thiết kế hệ thống).

Dù bạn là người mới bắt đầu hay đã có kinh nghiệm, mình tin những kiến thức trong ebook này sẽ giúp bạn rất nhiều trong việc review lại kiến thức về thiết kế, cũng như chuẩn bị cho các cuộc phỏng vấn về System Design.

Chúc bạn có những giây phút thú vị, bổ ích khi đọc ebook.

Giới thiệu

System Design Interview là gì?

System Design Interview hay phỏng vấn thiết kế hệ thống là một vòng phỏng vấn kỹ thuật trong đó bạn được yêu cầu thiết kế kiến trúc cho một hệ thống phần mềm giả định nào đó, thay vì coding trực tiếp. Thông thường, người phỏng vấn sẽ đưa ra một bài toán mở, ví dụ: thiết kế một mạng xã hội chia sẻ ảnh như Instagram hoặc TikTok, một dịch vụ nhắn tin như WhatsApp, hoặc tối ưu hệ thống phân phối nội dung cho một nền tảng streaming như Netflix. Nhiệm vụ của bạn là thảo luận và đề xuất một giải pháp kiến trúc tổng thể: bao gồm các thành phần chính của hệ thống, cách các thành phần này tương tác với nhau, và làm thế nào hệ thống đáp ứng được các yêu cầu đề ra (về tính năng, số lượng người dùng, tốc độ, v.v.).

Trong vòng phỏng vấn này, bạn sẽ không viết code chi tiết mà tập trung vào việc phác thảo và giải thích ý tưởng thiết kế. Bạn có thể sẽ vẽ sơ đồ kiến trúc trên bảng hoặc giấy (hoặc công cụ online) để trình bày luận điểm của mình. Người phỏng vấn thường quan sát cách bạn tiếp cận một bài toán lớn: từ việc làm rõ yêu cầu, đề xuất các thành phần kiến trúc (như dịch vụ, cơ sở dữ liệu, cache, v.v.), cho đến việc phân tích những điểm mạnh, điểm yếu của từng lựa chọn thiết kế. Mục đích là để đánh giá khả năng của bạn trong việc xây dựng một hệ thống phần mềm có tính mở rộng (scalable), ổn định và hiệu quả dựa trên yêu cầu đưa ra. Nói cách khác, cuộc phỏng vấn đề system design nhằm kiểm tra kỹ năng giải quyết vấn đề kỹ thuật phức tạp ở tầm kiến trúc của bạn: hiểu bài toán, đề ra hướng giải quyết, thảo luận các phương án và đưa ra quyết định phù hợp.

Tại sao vòng phỏng vấn này quan trọng?

Vòng phỏng vấn system design ngày càng trở thành yếu tố then chốt trong quy trình tuyển dụng kỹ sư phần mềm, đặc biệt tại các công ty công nghệ lớn. Lý do là bởi việc xây dựng và vận hành các hệ thống phần mềm quy mô lớn tiêu tốn rất nhiều nguồn lực, và các công ty muốn đảm bảo rằng những kỹ sư họ tuyển vào có thể nhanh chóng bắt tay vào thiết kế những hệ thống như vậy. Nói cách khác, họ dùng vòng phỏng vấn này như phương pháp chính để đánh giá liệu ứng viên có kiến thức và tư duy thiết kế hệ thống đủ mạnh hay không.

Kết quả của vòng phỏng vấn system design thường ảnh hưởng trực tiếp đến quyết định tuyển dụng và đãi ngộ dành cho bạn. Tại nhiều công ty lớn, đặc biệt cho các vị trí cấp trung và cao (senior, staff engineer), ứng viên sẽ phải trải qua một hoặc vài vòng thiết kế

hệ thống chuyên sâu. Vòng phỏng vấn này đôi khi còn được đánh giá cao hơn vòng coding, đặc biệt trong thời kỳ AI thống trị, việc coding càng ngày càng được chuyển giao nhiều hơn cho trí tuệ nhân tạo.

Ngoài ra, vòng phỏng vấn này còn quan trọng vì nó đánh giá những kỹ năng ở tầm tổng quát mà một kỹ sư phần mềm cần có khi làm việc thực tế. Coding interview thường chỉ kiểm tra khả năng giải thuật và coding ở quy mô nhỏ, trong khi thiết kế hệ thống kiểm tra tư duy chiến lược hơn: bạn hiểu bức tranh toàn cảnh ra sao, bạn sẽ thiết kế hệ thống phục vụ hàng triệu người dùng như thế nào, và bạn ra quyết định kỹ thuật ra sao khi đối mặt với các ràng buộc thực tế (hiệu năng, chi phí, độ phức tạp, v.v.). Đó là lý do các công ty lớn rất coi trọng vòng này, họ muốn tuyển người không chỉ code giỏi, mà còn thiết kế hệ thống tinh gọn, linh hoạt và có thể mở rộng cho tương lai.

Những khó khăn mà người mới thường gặp

Khi mới làm quen với phỏng vấn thiết kế hệ thống, bạn có thể gặp phải nhiều trở ngại. Dưới đây là những khó khăn phổ biến đối với các lập trình viên junior khi tiếp cận dạng phỏng vấn này:

- **Câu hỏi mở, không có đáp án cố định:** Các câu hỏi thiết kế hệ thống thường rất mở và không có một đáp án duy nhất đúng hẳn. Không có tiêu chí "đúng" hoặc "sai" rõ ràng như trong bài lập trình, nên bạn dễ cảm thấy lúng túng, không biết nên bắt đầu từ đâu và định hướng giải quyết vấn đề ra sao. Mỗi người có thể đưa ra một cách tiếp cận khác nhau cho cùng một bài toán thiết kế, miễn là đáp ứng được các yêu cầu cơ bản: do đó sẽ không có phương án nào được coi là hoàn toàn đúng hoặc hoàn toàn sai. Sự mơ hồ này khiến người phỏng vấn thiếu kinh nghiệm dễ mất phương hướng ban đầu.

- **Thiếu kinh nghiệm thực tế về hệ thống lớn:** Nhiều lập trình viên trẻ chưa từng tham gia thiết kế hoặc vận hành những hệ thống lớn trong công việc thực tế. Thậm chí có người đã đi làm vài năm nhưng vẫn “gần như chưa có kinh nghiệm” về môi trường cloud hoặc thiết kế hệ thống chịu tải cao. Điều này dẫn đến việc bạn thiếu kiến thức về các thành phần kiến trúc quan trọng (ví dụ: máy chủ ứng dụng, cơ sở dữ liệu phân tán, caching, message queue, load balancing, v.v.) và không biết cách áp dụng chúng trong thiết kế. Khi nhận một bài toán thiết kế, bạn có thể bối rối không rõ nên dùng công nghệ nào, module nào cho phù hợp, đơn giản vì bạn chưa từng làm việc với hệ thống tương tự quy mô đó.

- **Áp lực thời gian và tâm lý:** Mỗi buổi phỏng vấn thiết kế hệ thống thường chỉ kéo dài khoảng 45 : 60 phút, đây không phải là nhiều thời gian để bạn trình bày một kiến trúc phức tạp. Bạn phải vừa tư duy giải pháp, vừa diễn đạt nó một cách mạch lạc dưới áp lực thời gian, nên rất dễ căng thẳng. Nhiều người mới có thể bị "khớp" khi đối diện với bảng trắng và một vấn đề lớn, hoặc phân bổ thời gian không hợp lý (ví dụ: mải mê vẽ chi tiết mà không kịp nói hết ý tưởng chính). Kết quả là bài thiết kế trình bày dang dở hoặc thiếu nhiều phần quan trọng. Rõ ràng, kỹ năng quản lý thời gian và bình tĩnh xử lý tình huống áp lực là một thách thức lớn ở vòng này.

- **Chưa có phương pháp tiếp cận bài bản:** Thiết kế một hệ thống quy mô lớn là vấn đề phức tạp, đòi hỏi một phương pháp luận rõ ràng. Tuy nhiên, nhiều ứng viên chưa rèn luyện được framework (khuôn khổ) để giải quyết dạng bài này một cách có hệ thống. Thực tế, đa số ứng viên thất bại ở vòng System Design là do thiếu một quy trình tư duy lặp lại được cho các câu hỏi mở. Nếu bạn không có sẵn cho mình một chiến lược (chẳng hạn: bước 1 hỏi rõ yêu cầu, bước 2 đề xuất thiết kế tổng quan, bước 3 phân tích chuyên sâu các thành phần...), bạn sẽ dễ bị lúng túng trước lượng thông tin và lựa chọn khổng lồ. Việc thiếu phương pháp cũng khiến bạn có thể bỏ sót những khía cạnh quan trọng của hệ thống.

- **Ám ảnh về “giải pháp hoàn hảo”:** Không ít bạn lo lắng rằng mình phải đưa ra một thiết kế hoàn mỹ không có bất kỳ sai sót nào thì mới được đánh giá cao. Thực ra, người phỏng vấn không kỳ vọng một giải pháp 100% hoàn hảo từ bạn. Không hề có “đáp án đúng tuyệt đối” trong phỏng vấn thiết kế hệ thống, vì vậy điều quan trọng hơn là cách bạn tiếp cận vấn đề. Họ muốn thấy cách bạn đưa ra quyết định khi thông tin chưa đầy đủ, cách bạn xử lý các tình huống không chắc chắn, và liệu bạn có đủ linh hoạt để thích ứng khi yêu cầu thay đổi hay không. Hiểu được điều này sẽ giúp bạn tự tin hơn và tập trung vào quy trình giải quyết vấn đề thay vì cố tìm một đáp án hoàn hảo duy nhất. Hãy nhớ rằng phỏng vấn thiết kế hệ thống chủ yếu đánh giá tư duy và kỹ năng giao tiếp kỹ thuật của bạn, chứ không phải tìm kiếm một thiết kế hoàn chỉnh để đem triển khai ngay.

Các kỹ năng mà nhà tuyển dụng mong muốn

Vậy nhà tuyển dụng thật sự tìm kiếm điều gì ở ứng viên trong vòng phỏng vấn thiết kế hệ thống? Dưới đây là một số kỹ năng và tố chất quan trọng mà bạn nên chú trọng rèn luyện:

- **Kiến thức nền tảng vững chắc:** Bạn cần nắm vững các kiến thức cơ bản về thiết kế và kiến trúc hệ thống. Điều này bao gồm hiểu biết về các thành phần như máy chủ, cơ sở dữ liệu (quan hệ vs phi quan hệ), hệ thống storage, networking, caching, load balancing, v.v. Bên cạnh đó, bạn cũng nên nắm các khái niệm như tính mở rộng (scalability), tính sẵn sàng (availability), độ trễ (latency), tính nhất quán (consistency)... Kiến thức nền tảng vững giúp bạn tự tin đề xuất các giải pháp phù hợp và giải thích được tại sao chọn thiết kế đó.

- **Tư duy phân tích và giải quyết vấn đề:** Kỹ năng phân tích một bài toán lớn thành các phần nhỏ hơn là rất quan trọng. Nhà tuyển dụng muốn thấy bạn biết cách làm rõ yêu cầu bài toán (functional & non-functional requirements), biết đặt câu hỏi để bổ sung thông tin, và từ đó định hình được vấn đề cần giải quyết. Bạn cũng cần có tư duy hệ thống để xâu chuỗi các phần lại thành một giải pháp hoàn chỉnh. Khả năng xác định trọng tâm của vấn đề và giải quyết lần lượt từng khía cạnh (dữ liệu, giao tiếp giữa các dịch vụ, giao diện API, v.v.) cho thấy bạn có phương pháp làm việc rõ ràng và hiệu quả.

- **Kỹ năng thiết kế hệ thống quy mô lớn:** Đây là khả năng hình dung và thiết kế một hệ thống có thể hoạt động tốt ở quy mô hàng triệu người dùng hoặc hơn. Bạn cần thể hiện được việc cân nhắc đến yếu tố tải cao (high load). Ví dụ: nếu hệ thống tăng từ 1 ngàn lên 1 triệu người dùng thì kiến trúc có chịu đựng nổi không, hay phải thay đổi gì. Đồng thời, kỹ năng này bao gồm việc thiết kế hệ thống có tính chịu lỗi (fault-tolerance): hệ thống vẫn tiếp tục hoạt động ngay cả khi một phần nào đó gặp sự cố. Nhà tuyển dụng muốn thấy rằng bạn biết nghĩ đến tương lai của hệ thống: cách hệ thống mở rộng khi lưu lượng tăng, cách sao lưu dữ liệu, cách xử lý khi một dịch vụ con bị hỏng, v.v. Đây là những yếu tố phân biệt một thiết kế tạm thời với một thiết kế tốt cho sản phẩm thực tế.

- **Khả năng ra quyết định và cân bằng đánh đổi:** Trong thiết kế hệ thống, không có giải pháp nào là hoàn hảo toàn diện: mỗi lựa chọn đều đi kèm đánh đổi (trade-off). Ví dụ: lựa chọn database SQL sẽ dễ truy vấn phức tạp nhưng khó mở rộng ngang bằng cách thêm máy chủ, trong khi NoSQL thì ngược lại. Người phỏng vấn sẽ đánh giá cao khả năng bạn đưa ra quyết định hợp lý và giải thích được lý do cho các lựa chọn thiết kế của mình. Họ muốn thấy bạn cân nhắc và cân bằng giữa các yếu tố đánh đổi: như hiệu năng vs. chi phí, tốc độ phát triển vs. khả năng mở rộng, độ phức tạp vs. tính linh hoạt, v.v.. Kỹ năng này thể hiện tư duy của một technical leader: biết lựa chọn hướng đi tối ưu dựa trên bối cảnh cụ thể của bài toán.

- **Kỹ năng giao tiếp và cộng tác:** Khả năng trình bày rõ ràng ý tưởng kỹ thuật của bạn là vô cùng quan trọng. Trong phỏng vấn thiết kế hệ thống, bạn cần giao tiếp mạch lạc để người phỏng vấn (và giả sử là đồng nghiệp tương lai) hiểu được kiến trúc bạn đề xuất. Điều này bao gồm việc dùng ngôn ngữ dễ hiểu, tổ chức ý tưởng có thứ tự, và minh họa bằng sơ đồ/trạng thái khi cần thiết. Một mẹo nhỏ là hãy cố gắng "nghĩ thành tiếng": chia sẻ suy nghĩ của bạn liên tục thay vì im lặng quá lâu. Bên cạnh đó, thái độ cởi mở và hợp tác cũng được đánh giá cao: người phỏng vấn có thể sẽ gợi ý hoặc đặt câu hỏi thử thách vào thiết kế của bạn, và họ muốn thấy bạn biết lắng nghe, trao đổi lại một cách tích cực, giống như khi làm việc nhóm thực sự. Việc truyền đạt ý tưởng rõ ràng và tương tác nhịp nhàng sẽ gây ấn tượng tốt rằng bạn có thể làm việc hiệu quả trong team.

Tóm lại, để vượt qua vòng phỏng vấn thiết kế hệ thống, bạn cần kết hợp cả kiến thức kỹ thuật nền tảng lẫn kỹ năng mềm về phân tích, ra quyết định và giao tiếp. Nhà tuyển dụng tìm kiếm một người kỹ sư có thể nhìn toàn cảnh, thiết kế giải pháp phù hợp và thuyết phục được người khác về giải pháp đó.

Các khái niệm nền tảng trong Thiết kế hệ thống

Trong chương 2, chúng ta sẽ cùng khám phá những khái niệm nền tảng nhất trong thiết kế hệ thống. Đây là những viên gạch đầu tiên giúp bạn xây dựng tư duy System Design vững chắc, đặc biệt hữu ích cho các bạn lập trình viên junior đến middle khi chuẩn bị cho phỏng vấn thiết kế hệ thống. Ta sẽ cùng nhau đi qua một vài khái niệm như **Latency** (độ trễ), **Throughput** (thông lượng), **Scalability** (khả năng mở rộng), **Availability** (tính khả dụng), **Reliability** (độ tin cậy), khái niệm **High Level Architecture** (kiến trúc cấp cao), và các thành phần phổ biến trong một hệ thống.

Latency (Độ trễ)

Độ trễ là khoảng thời gian chờ đợi để một yêu cầu hoặc hành động được thực thi và nhận kết quả. Nói một cách đơn giản, đó là độ trễ từ lúc bạn bắt đầu gửi yêu cầu đến lúc bạn nhận được phản hồi. Ví dụ trong web: khi bạn nhấp vào một liên kết, độ trễ là thời gian từ lúc bạn nhấn chuột đến khi trang web hiển thị nội dung. Độ trễ thường được đo bằng mili-giây (ms) : con số càng nhỏ thì phản hồi càng nhanh.

Hãy tưởng tượng một tình huống đời thường: bạn gửi một bưu phẩm từ Hà Nội vào TP. Hồ Chí Minh. Thời gian từ lúc bạn gửi đến khi người nhận nhận được chính là “độ trễ” của việc vận chuyển. Tương tự, trong hệ thống máy tính, nếu máy chủ ở xa hoặc đường truyền chậm, độ trễ mạng sẽ cao: nghĩa là bạn phải chờ lâu để nhận kết quả. Ngược lại, máy chủ gần và đường truyền tốt sẽ giúp độ trễ thấp, bạn nhận được phản hồi nhanh hơn rất nhiều.

Latency ảnh hưởng trực tiếp đến trải nghiệm người dùng. Ví dụ: một trang web có độ trễ cao (phản hồi chậm) sẽ làm người dùng khó chịu và có thể rời đi, trong khi trang web có độ trễ thấp (phản hồi gần như tức thì) sẽ tạo cảm giác mượt mà. Trong thiết kế hệ thống, chúng ta luôn cố gắng giảm độ trễ: bằng cách tối ưu đường truyền, đặt máy chủ gần người dùng, hoặc cải thiện tốc độ xử lý, để đem lại trải nghiệm tốt nhất.

Throughput (Thông lượng)

Nếu latency nói về tốc độ của một yêu cầu đơn lẻ, thì throughput lại nói về khối lượng công việc hệ thống có thể xử lý trong một khoảng thời gian. Throughput (hay băng

thông thực tế) đo lường xem có bao nhiêu dữ liệu hoặc bao nhiêu yêu cầu được xử lý thành công mỗi giây (hoặc mỗi phút, mỗi giờ). Nôm na, đây là "sức tải" của hệ thống.

Hãy lấy ví dụ đời thường để hình dung: tưởng tượng bạn đang quản lý một trạm thu phí trên cao tốc. Độ trễ tương tự như thời gian một chiếc xe cần để qua trạm thu phí (mỗi xe qua nhanh hay chậm). Còn thông lượng là số lượng xe có thể qua trạm trong một phút. Một trạm thu phí có thể cho 60 xe qua mỗi phút thì có thông lượng cao hơn trạm chỉ cho 30 xe/phút. Trong hệ thống cũng vậy, một server xử lý được 1000 yêu cầu/giây có thông lượng cao hơn server chỉ xử lý 100 yêu cầu/giây.

Throughput và latency có mối quan hệ nhưng không phải lúc nào cũng tỷ lệ thuận với nhau. Ví dụ: một hệ thống có thể có độ trễ thấp (phản hồi rất nhanh cho mỗi yêu cầu) nhưng nếu chỉ phục vụ được ít người dùng cùng lúc, thì throughput vẫn thấp. Ngược lại, hệ thống có thể phục vụ rất nhiều yêu cầu mỗi giây (throughput cao) nhưng mỗi yêu cầu lại mất nhiều thời gian (latency cao) nếu hệ thống bị quá tải. Do đó, khi thiết kế hệ thống, chúng ta cần cân bằng cả hai yếu tố: vừa đảm bảo latency thấp để người dùng có phản hồi nhanh, vừa có throughput cao để phục vụ được nhiều người dùng đồng thời.

Scalability (Khả năng mở rộng)

Khả năng mở rộng là khả năng của hệ thống thích nghi và xử lý hiệu quả khi khối lượng công việc tăng lên. Nói cách khác, một hệ thống có khả năng mở rộng tốt sẽ "lớn lên" cùng với số lượng người dùng hoặc dữ liệu ngày càng tăng mà không bị suy giảm hiệu năng. Điều này đặc biệt quan trọng đối với các ứng dụng có tham vọng phục vụ hàng triệu người dùng.

Hãy tưởng tượng bạn mở một quán phở nhỏ. Ban đầu quán chỉ có 5 bàn, phục vụ 20 khách mỗi sáng là tối đa. Nhưng nếu phở của bạn quá ngon và khách kéo đến ngày càng đông, bạn sẽ làm gì? Bạn có hai lựa chọn:

- **Mở rộng chiều dọc (Vertical Scaling):** Tăng công suất của quán hiện tại: thuê thêm đầu bếp giỏi hơn, nâng cấp nồi nấu to hơn, nguyên liệu nhiều hơn. Trong hệ thống, điều này tương tự với việc nâng cấp máy chủ hiện có (thêm CPU, thêm RAM, ổ cứng nhanh hơn). Quán phở phục vụ được nhiều khách hơn trên cùng một địa điểm, nhưng sẽ có giới hạn (một nồi phở to cỡ nào cũng có hạn chế, máy chủ cũng vậy).

- **Mở rộng chiều ngang (Horizontal Scaling):** Mở thêm nhiều quán phở ở các địa điểm khác hoặc thêm nhiều nồi phở và đầu bếp hoạt động song song. Trong hệ thống, đây là việc thêm nhiều máy chủ vào hệ thống. Mỗi máy chủ mới sẽ chia sẻ

gánh nặng xử lý, giống như có thêm chi nhánh phục vụ khách. Cách này thường linh hoạt và hiệu quả hơn: nếu một quán (hay một máy chủ) bận hoặc hỏng, quán khác vẫn phục vụ được.

Trong thiết kế hệ thống thực tế, mở rộng chiều ngang (thêm server) thường được ưu tiên vì nó tăng khả năng chịu tải và dự phòng lỗi tốt hơn. Tuy nhiên, triển khai mở rộng chiều ngang cần có cơ chế cân bằng tải (sẽ nói ở phần sau) để phân phối công việc giữa các máy chủ. Tóm lại, khả năng mở rộng đảm bảo rằng hệ thống của bạn có thể phình to khi cần thiết (và thậm chí thu nhỏ lại khi nhu cầu giảm) một cách trơn tru mà không làm gián đoạn dịch vụ hoặc giảm hiệu năng.

Availability (Tính khả dụng)

Tính khả dụng thể hiện mức độ mà hệ thống luôn sẵn sàng để phục vụ: hay nói nôm na, hệ thống chạy "ổn định" được bao nhiêu phần trăm thời gian. Một hệ thống có tính khả dụng cao nghĩa là hầu như bất kỳ lúc nào người dùng cần, hệ thống đều hoạt động và đáp ứng. Ngược lại, nếu hệ thống hay downtime (ngừng hoạt động) thường xuyên, tính khả dụng sẽ thấp.

Chúng ta thường nghe những con số "chín" huyền thoại như 99.9%, 99.99%... Ví dụ 99.99% thời gian hoạt động nghĩa là trong 10000 phút thì hệ thống chỉ được phép ngưng tối đa 1 phút. Những dịch vụ lớn đặt mục tiêu "năm số 9" (99.999%) tức downtime chỉ vài giây mỗi năm, tức là gần như không bao giờ "chết". Tất nhiên, đạt được điều này không dễ, nhưng nó cho thấy tầm quan trọng của tính khả dụng.

Để hình dung, hãy nghĩ đến điện sinh hoạt trong nhà bạn: bạn mong muốn đèn luôn sáng khi bật công tắc. Nếu một ngày bị cúp điện (dịch vụ không khả dụng), bạn sẽ rất bất tiện. Tương tự, một trang web hay ứng dụng nếu buổi tối bạn mở lên mà "không vào được" thì đó là vấn đề về tính khả dụng. Trong thiết kế hệ thống, để tăng tính khả dụng, người ta thường sử dụng các giải pháp như dự phòng: có nhiều máy chủ chạy song song, nếu một máy hỏng sẽ có máy khác thay thế ngay. Load balancing (cân bằng tải) cũng giúp tăng khả dụng (vì có thể rút một máy ra sửa mà người dùng không bị ảnh hưởng). Ngoài ra còn có các kỹ thuật như triển khai ở nhiều trung tâm dữ liệu khác nhau (multi-zone, multi-region) để tránh sự cố ở một nơi làm sập toàn bộ hệ thống. Mục tiêu cuối cùng là hệ thống luôn "sẵn sàng" phục vụ bất kể hoàn cảnh.

Reliability (Độ tin cậy)

Độ tin cậy thể hiện mức độ hệ thống vận hành ổn định và nhất quán trong thời gian dài, đảm bảo tính toàn vẹn của dữ liệu và dịch vụ. Một hệ thống tin cậy là hệ thống mà bạn có thể tin tưởng rằng nó sẽ không bị lỗi nghiêm trọng, không mất mát dữ liệu, và nếu có sự cố nhỏ thì hệ thống cũng tự phục hồi hoặc có cơ chế khôi phục mà không ảnh hưởng đến người dùng.

Hình dung trong đời sống: bạn có một chiếc xe máy rất "đáng tin cậy" nghĩa là suốt nhiều năm trời nó hiếm khi hỏng vặt giữa đường, bạn đi làm ngày nào cũng nổ máy là chạy, không lo dắt bộ. Hệ thống máy tính cũng vậy, độ tin cậy cao nghĩa là các dịch vụ chạy bền bỉ, ít lỗi, và đặc biệt dữ liệu của bạn an toàn qua thời gian. Ví dụ, trong một hệ thống thương mại điện tử hoặc ngân hàng, độ tin cậy là vô cùng quan trọng: mỗi giao dịch của khách hàng phải được lưu lại chắc chắn, không thể "mất giữa chừng" chỉ vì một server nào đó sập. Nếu một máy chủ xử lý giao dịch bị lỗi, một máy chủ khác phải tiếp quản ngay với đầy đủ dữ liệu (nhờ sao lưu thời gian thực), đảm bảo giao dịch không bị mất hoặc sai sót.

Bạn có thể nghe người ta so sánh tính khả dụng và độ tin cậy. Hai khái niệm này liên quan nhưng không giống hệt nhau: một hệ thống có độ tin cậy cao thì gần như chắc chắn cũng khả dụng cao, vì nó hiếm khi gặp sự cố. Nhưng một hệ thống khả dụng cao (ít downtime) chưa chắc đã tin cậy về dữ liệu. Nói cách khác, reliability bao hàm cả high availability (khả dụng cao) nhưng còn thêm yếu tố toàn vẹn và đúng đắn của hoạt động. Trong thực tế, đạt được độ tin cậy tuyệt đối là rất khó, nên các kiến trúc sư thường tập trung vào tính khả dụng cao và giảm thiểu rủi ro mất dữ liệu đến mức thấp nhất có thể. Các kỹ thuật nâng cao độ tin cậy thường gồm: sao lưu và phục hồi dữ liệu, replication (nhân bản dữ liệu sang nhiều máy), thiết kế loại bỏ điểm lỗi đơn (single point of failure), v.v. Tóm lại, độ tin cậy giúp người dùng yên tâm rằng hệ thống của bạn "sống dai sống khỏe" và giữ gìn dữ liệu của họ an toàn.

High-Level Architecture (Kiến trúc cấp cao)

Khi thiết kế hệ thống, trước khi đi vào chi tiết cụ thể, chúng ta thường vẽ ra kiến trúc cấp cao: một bức tranh tổng quát về hệ thống. Kiến trúc cấp cao là cái nhìn toàn cảnh về các thành phần chính của hệ thống và cách chúng tương tác với nhau, không đi sâu vào chi tiết implementation. Nó giống như bản thiết kế tổng thể của một ngôi nhà trước khi bạn lo chi tiết từng phòng ốc, hay như bản đồ thành phố trước khi zoom vào từng con hẻm.

Ví dụ, khi thiết kế một hệ thống mạng xã hội đơn giản, kiến trúc cấp cao có thể bao gồm: người dùng trên ứng dụng di động/website gửi yêu cầu lên máy chủ ứng dụng, máy chủ này đọc/ghi dữ liệu từ cơ sở dữ liệu, có thể gọi thêm một số dịch vụ phụ trợ khác (ví dụ dịch vụ lưu trữ hình ảnh, dịch vụ tìm kiếm), và tất cả được đặt sau một load balancer (cân bằng tải). Kiến trúc cấp cao có thể được vẽ dưới dạng sơ đồ các khối (boxes) và đường nối giữa chúng, mỗi khối là một thành phần (như web server, database, cache, v.v.).

Mục đích của kiến trúc cấp cao là giúp bạn và đội ngũ của bạn cùng hiểu rõ cấu trúc tổng thể của hệ thống trước. Nhờ đó, ta xác định được các thành phần chính, mối quan hệ giữa chúng, điểm mạnh, điểm yếu và các điểm có thể là nút thắt cổ chai. Kiến trúc cấp cao tốt sẽ đơn giản, rõ ràng, tránh ôm đồm quá nhiều chi tiết, nhưng cũng phải đủ thành phần để đáp ứng yêu cầu. Trong các buổi phỏng vấn thiết kế hệ thống, người ta rất hay yêu cầu bạn phác thảo kiến trúc cấp cao trước: đây là kỹ năng quan trọng, giúp bạn trình bày ý tưởng một cách mạch lạc và có tổ chức.

Các thành phần phổ biến trong hệ thống

Bây giờ, chúng ta cùng điểm qua những thành phần phổ biến thường xuất hiện trong kiến trúc của một hệ thống lớn. Hiểu rõ vai trò của từng thành phần này sẽ giúp bạn rất nhiều khi phân tích hoặc thiết kế hệ thống:

Web Server (Máy chủ Web)

Web server là thành phần chịu trách nhiệm tiếp nhận các yêu cầu HTTP từ client (trình duyệt hoặc ứng dụng) và phản hồi lại kết quả (thường là nội dung trang web hoặc dữ liệu API). Web server giống như lễ tân ở đầu hệ thống: mỗi khi có “khách” (request) đến gõ cửa, lễ tân này sẽ xem khách yêu cầu gì, sau đó quyết định sẽ đưa khách đó tới phòng ban nào (ví dụ đưa yêu cầu cho Application server xử lý) rồi nhận kết quả trả về cho khách.

Các web server phổ biến hiện nay như Apache, Nginx, hoặc IIS… không chỉ đơn thuần chuyển tiếp yêu cầu mà còn có thể phục vụ các nội dung tĩnh (static content) như hình ảnh, file CSS, JS trực tiếp cho client, hoặc thực hiện cân bằng tải đơn giản. Trong một số kiến trúc, web server và application server có thể tích hợp làm một (ví dụ nhiều framework tự nhúng server). Nhưng về khái niệm, web server là cửa ngõ giao tiếp giữa bên ngoài và hệ thống của bạn, quản lý phiên làm việc (session), bảo mật đầu vào ở mức cơ bản (như chặn một số request xấu) trước khi chuyển cho tầng xử lý bên trong.

Ví dụ: Khi bạn gõ URL https://facebook.com vào trình duyệt, yêu cầu đó sẽ được gửi tới web server của Facebook trước. Web server sẽ kiểm tra bạn có đăng nhập chưa, bạn yêu cầu trang nào, sau đó mới chuyển tiếp yêu cầu đến các dịch vụ bên trong để lấy dữ liệu về trang Facebook cá nhân của bạn rồi gửi lại trình duyệt.

Application Server (Máy chủ ứng dụng)

Nếu web server là lễ tân, thì application server chính là "phòng bếp" hoặc "đầu bếp" nơi thực sự xử lý yêu cầu và thực thi nghiệp vụ. Đây là nơi chứa logic chính của ứng dụng. Application server nhận các yêu cầu đã được web server "chuyển vào", sau đó thực hiện các công việc như: kiểm tra thông tin người dùng, xử lý các tính toán, truy vấn hoặc cập nhật dữ liệu trong database, gọi tới các dịch vụ khác nếu cần, rồi tạo ra kết quả (ví dụ nội dung trang HTML, hoặc dữ liệu JSON) để trả về cho web server gửi lại người dùng.

Application server có thể được xây dựng bằng nhiều ngôn ngữ và framework khác nhau: ví dụ Java với Spring Boot, Python với Django/Flask, JavaScript/Node.js với Express, C# với .NET, v.v. Dù công nghệ gì thì vai trò chung của nó vẫn là thực thi logic nghiệp vụ (business logic). Trong hệ thống lớn, ta thường có nhiều application server chạy song song (đặt sau load balancer) để đảm bảo phục vụ lượng lớn người dùng. Mỗi application server thường giống nhau (cùng một ứng dụng được deploy), nhờ đó bất kỳ cái nào cũng có thể xử lý yêu cầu như nhau : việc này giúp dễ mở rộng và dự phòng.

Ví dụ: Khi bạn đặt xe qua ứng dụng gọi xe, web server nhận yêu cầu và chuyển vào application server. Application server sẽ tính toán tìm tài xế gần bạn nhất, kiểm tra giá cước, xác nhận tài xế, sau đó tạo một đối tượng chuyến đi mới trong cơ sở dữ liệu, rồi trả về cho bạn thông tin tài xế và xe. Toàn bộ những việc "tính toán và ra quyết định" đó diễn ra ở application server.

Database (Cơ sở dữ liệu)

Database (cơ sở dữ liệu) là thành phần đóng vai trò lưu trữ dữ liệu lâu dài cho hệ thống. Nếu coi hệ thống như một ứng dụng thì database chính là "bộ nhớ" của ứng dụng đó. Mọi thông tin cần được ghi nhớ và truy xuất sau này: từ tài khoản người dùng, bài viết, đơn hàng, tin nhắn… đều sẽ lưu trong database.

Có nhiều loại database, nhưng phổ biến nhất trong hệ thống web là:

- Database quan hệ (SQL): như MySQL, PostgreSQL, Oracle, Microsoft SQL Server,... Lưu dữ liệu thành bảng, có cấu trúc rõ ràng, và dùng ngôn ngữ truy vấn SQL. Thích hợp cho dữ liệu có cấu trúc chặt chẽ và cần các truy vấn phức tạp

(như join nhiều bảng).

- Database phi quan hệ (NoSQL): như MongoDB, Cassandra, Redis, DynamoDB,... Lưu dữ liệu dạng linh hoạt hơn (document, key-value, graph,...). Thích hợp cho dữ liệu không có cấu trúc cố định, hoặc cần hiệu năng cao, mở rộng dễ dàng.

Trong một hệ thống lớn, database thường chạy trên một (hoặc nhiều) máy chủ cơ sở dữ liệu riêng, tách biệt với application server. Điều này cho phép ta mở rộng từng thành phần độc lập và đảm bảo hiệu năng. Đôi khi, để tăng khả năng đáp ứng, người ta còn triển khai replication (nhân bản cơ sở dữ liệu sang nhiều máy: một máy chính để ghi, nhiều máy phụ để đọc), giúp chia tải và tăng độ tin cậy.

Load Balancer (Bộ cân bằng tải)

Load balancer là thành phần đặc biệt có nhiệm vụ phân phối đều tải công việc đến nhiều máy chủ phía sau nó. Bạn có thể hình dung load balancer như một anh điều phối giao thông hoặc người gác cổng thông minh: mọi yêu cầu từ client thay vì đi thẳng đến một máy chủ cố định, sẽ đi qua load balancer trước. Load balancer sau đó quyết định gửi yêu cầu đó đến máy chủ nào đang rảnh rỗi hoặc phù hợp, đảm bảo không máy nào bị quá tải trong khi máy khác nhàn rỗi.

Vai trò chính của load balancer gồm:

- **Phân phối tải:** Ví dụ có 5 server ứng dụng, load balancer sẽ đảm bảo 5 server này mỗi cái nhận ~20% lượng request, không ai phải "gánh" 90% trong khi người khác chỉ 10%. Thuật toán phân phối có thể là round-robin (xoay vòng lần lượt), least connections (ưu tiên server nào đang ít kết nối), hoặc thông minh hơn tùy hệ thống.

- **Tính dự phòng và khả dụng cao:** Nếu một server phía sau bị sập, load balancer có thể loại server đó ra khỏi danh sách tạm thời và chuyển hướng request sang các server còn lại. Nhờ đó người dùng hầu như không nhận ra có sự cố. Khi server kia hoạt động lại, load balancer lại thêm vào. Điều này tăng tính khả dụng cho hệ thống (hệ thống vẫn phục vụ bình thường dù một máy hỏng).

- **Tính linh hoạt:** Bạn có thể thêm server mới hoặc gỡ server cũ xuống để bảo trì dễ dàng mà không phải dừng toàn bộ dịch vụ: chỉ cần báo cho load balancer biết để điều chỉnh phân phối.

Có các loại load balancer phần cứng (thiết bị chuyên dụng) hoặc phần mềm (ví dụ HAProxy, Nginx, hoặc các dịch vụ cloud load balancing). Đối với bạn phỏng vấn system design ở mức cơ bản, hiểu khái niệm là đủ: load balancer giúp mở rộng hệ thống theo chiều ngang và tránh điểm quá tải, điểm thất bại duy nhất.

Ví dụ: Trang web bán vé xem bóng đá online có những đợt mở bán vé rất đông người vào cùng lúc. Họ sẽ triển khai nhiều server web/app song song. Trước các server đó đặt một load balancer. Mỗi khi 100.000 người dùng cùng click “mua vé”, các request sẽ được load balancer chia đều cho nhiều server xử lý, tránh tình trạng một server bị dồn quá nhiều người gây sập trong khi server khác không ai truy cập.

Cache (Bộ nhớ đệm)

Cache là thành phần dùng để lưu trữ tạm thời những dữ liệu thường xuyên được truy cập nhằm tăng tốc độ truy xuất. Cache thường là một bộ nhớ đệm siêu nhanh (ví dụ dùng RAM, hoặc các hệ thống lưu trữ tối ưu) nằm ở vị trí “gần” với nơi xử lý để khi cần dữ liệu có thể lấy ra ngay, thay vì lúc nào cũng phải tìm đến nguồn dữ liệu chậm hơn (như database trên đĩa, hoặc dịch vụ bên ngoài).

Hãy tưởng tượng bạn có một tủ hồ sơ lưu hàng nghìn tài liệu. Mỗi lần cần tra cứu một tài liệu hay dùng, nếu lần nào bạn cũng đi tới tủ hồ sơ lục tìm thì rất mất thời gian. Thay vào đó, bạn photo vài bản và để ngay trên bàn làm việc: đó chính là cache. Khi cần, chỉ việc với tay lấy trên bàn, nhanh hơn nhiều so với việc đi đến tủ tìm bản gốc. Tuy nhiên, nếu tài liệu gốc cập nhật, bạn cũng nên thay đổi bản photo trên bàn cho đúng: đây chính là thách thức đồng bộ dữ liệu cache với dữ liệu gốc. (Có câu nói vui nổi tiếng: một trong những việc khó trong lập trình, đó là xử lý invalidate cache).

Trong hệ thống, cache có thể xuất hiện ở nhiều nơi:

- Cache phía client: ví dụ trình duyệt web của người dùng cache các file hình ảnh, CSS, JavaScript để lần sau mở trang web nhanh hơn vì không phải tải lại những nội dung đó.

- Cache phía server: ví dụ in-memory cache trên application server (như dùng thư viện cache của framework) để lưu tạm kết quả các truy vấn tốn thời gian, hoặc distributed cache (như Redis, Memcached) : một hệ thống cache riêng chạy trên RAM để các server khác có thể sử dụng chung.

Sử dụng cache phải cân đối: đúng dữ liệu, đúng chỗ thì hệ thống tăng tốc đáng kể; nhưng dùng không cẩn thận có thể gây sai lệch (nếu dữ liệu cache đã cũ mà không cập nhật). Khi thiết kế, bạn nên chọn cache cho những dữ liệu đọc nhiều hơn ghi, và không yêu cầu lúc nào cũng phải mới nhất tuyệt đối.

Ví dụ: Một trang tin tức có trang chủ hiển thị danh sách bài viết "hot" nhất. Danh sách này tính toán từ hàng chục nghìn bài viết và cập nhật mỗi giờ. Thay vì mỗi lần người dùng truy cập trang chủ đều truy vấn tính toán lại (rất chậm), hệ thống sẽ cache kết quả danh sách "hot" đó trong vòng 5 phút hoặc 1 giờ. Nhờ cache, khi 1000 người cùng mở trang chủ, hệ thống chỉ lấy danh sách từ bộ nhớ đệm ra phục vụ, rất nhanh và giảm tải cho database.

CDN (Mạng phân phối nội dung)

CDN (Content Delivery Network) là mạng lưới gồm nhiều máy chủ đặt tại nhiều vị trí địa lý khác nhau, giúp phân phối nội dung (thường là nội dung tĩnh như hình ảnh, video, file CSS/JS) đến người dùng từ máy chủ gần họ nhất. Mục tiêu của CDN là giảm độ trễ và tăng tốc độ tải nội dung cho người dùng trên toàn thế giới, đồng thời giảm tải cho máy chủ gốc của bạn.

Bạn có thể hình dung CDN như hệ thống kho hàng và cửa hàng vệ tinh: Giả sử bạn bán một sản phẩm online từ Hà Nội, nếu gửi hàng cho khách trong Hà Nội thì rất nhanh, nhưng gửi cho khách ở Mỹ thì sẽ chậm. Nếu bạn thông minh, bạn có thể đặt kho hàng tại Mỹ với sẵn sản phẩm; khi khách Mỹ đặt mua, hàng sẽ được giao từ kho Mỹ, nhanh hơn nhiều. Tương tự, hình ảnh/video trên website của bạn có thể được đẩy lên các máy chủ CDN ở khắp nơi. Khi một người dùng ở Việt Nam truy cập, ảnh sẽ được lấy từ máy chủ CDN tại Việt Nam; người dùng ở châu Âu sẽ nhận ảnh từ máy chủ CDN ở châu Âu, v.v. Kết quả là ai cũng tải nhanh hơn vì nội dung không phải đi nửa vòng trái đất.

Trong thiết kế hệ thống, tích hợp một CDN (như Cloudflare, Akamai, Amazon CloudFront, v.v.) là giải pháp phổ biến để tăng tốc website và giảm áp lực lên server chính. CDN thường được dùng cho hình ảnh, video, file tĩnh... nhưng cũng có thể cache cả trang HTML tùy trường hợp. Đối với người dùng cuối, CDN hoạt động "vô hình": họ chỉ cảm thấy website nhanh hơn, mượt hơn mà không cần biết chi tiết kỹ thuật phía sau.

Message Queue (Hàng đợi thông điệp)

Message Queue (hàng đợi thông điệp) là một thành phần cho phép các phần khác nhau của hệ thống giao tiếp gián tiếp với nhau thông qua việc gửi và nhận thông điệp không đồng bộ. Nghe có vẻ hàn lâm, nhưng thực chất nó giống như một dây chuyền giao việc:

một bên bỏ nhiệm vụ vào hàng đợi, bên kia rảnh thì lấy nhiệm vụ ra làm, thay vì bắt bên gửi phải chờ bên nhận làm xong ngay lập tức.

Ví dụ đời thường: bạn đến quán ăn gọi món phở đặc biệt. Quán đang đông nên anh phục vụ không thể đứng chờ bếp nấu xong tô phở của bạn rồi mới đi tiếp. Thay vào đó, anh ghi yêu cầu của bạn vào một phiếu gọi món và ghim vào hàng đợi trong bếp, sau đó anh tiếp tục đi phục vụ khách khác. Bên trong bếp, các đầu bếp sẽ lần lượt lấy các phiếu trong hàng đợi ra và nấu theo thứ tự. Khi nấu xong thì đưa ra quầy, anh phục vụ thấy phở đã xong sẽ mang đến cho bạn. Ở đây, phiếu gọi món chính là message và cái bảng ghim trong bếp là message queue. Nhờ có hàng đợi, quán có thể phục vụ nhiều khách hiệu quả: khách gọi món xong không phải đứng đợi bếp làm xong ngay trước mặt, bếp thì cứ làm tuần tự, ai làm xong việc gì lại lấy phiếu mới.

Trong hệ thống phần mềm, message queue (như RabbitMQ, Kafka, ActiveMQ, Amazon SQS, v.v.) cho phép tách các tiến trình thành phần thành các service độc lập hoạt động không đồng bộ. Một service có thể gửi thông điệp (nhiệm vụ) vào queue, service khác nhận và xử lý khi có thể:

- Điều này rất hữu ích cho xử lý tác vụ nền (background processing): ví dụ sau khi người dùng đăng ký tài khoản, hệ thống gửi một email xác nhận. Thay vì bắt người dùng chờ cho đến khi gửi email xong (làm tăng độ trễ cho hành động đăng ký), application server sẽ đẩy một thông điệp vào hàng đợi “Gửi email xác nhận”. Một service khác chuyên gửi email sẽ đọc thông điệp này và thực hiện gửi email. Người dùng thì có thể tiếp tục sử dụng ứng dụng ngay sau khi đăng ký, không phải chờ việc gửi email.

- Message queue cũng giúp điều hòa tải: nếu lúc cao điểm có quá nhiều nhiệm vụ, chúng xếp hàng trong queue và xử lý dần dần, hệ thống không bị sập do cố làm tất cả cùng lúc. Khi tải giảm, hàng đợi sẽ trống dần.

Tóm lại, hàng đợi thông điệp giúp hệ thống của bạn linh hoạt hơn, chịu tải tốt hơn và rời rạc hơn (các thành phần không phụ thuộc chặt thời gian thực vào nhau). Trong thiết kế hệ thống, khi thấy một tác vụ nào không cần kết quả tức thì cho người dùng, bạn có thể nghĩ đến việc đưa nó vào hàng đợi để xử lý bất đồng bộ.

Trên đây chúng ta đã đi qua những khái niệm cốt lõi của thiết kế hệ thống, từ các chỉ số hiệu năng như độ trễ, thông lượng; các tính chất hệ thống như khả năng mở rộng, tính

khả dụng, độ tin cậy; cho đến kiến trúc tổng quan và các thành phần cơ bản thường gặp. Ở các chương tiếp theo, chúng ta sẽ xây dựng dựa trên nền tảng này để đi sâu hơn vào cách thiết kế một hệ thống lớn hoàn chỉnh. Hãy nhớ, mọi hệ thống lớn đều được tạo nên từ những ý tưởng và thành phần cơ bản như trên: nắm vững chúng chính là chìa khóa để bạn chinh phục những buổi phỏng vấn thiết kế hệ thống khó nhằn.

Tính nhất quán và khả dụng trong System Design

Giới thiệu

Khi thiết kế hệ thống phân tán, hai khái niệm tính nhất quán (**consistency**) và tính khả dụng (**availability**) luôn xuất hiện như một cặp "bài trùng". Đặc biệt trong các buổi phỏng vấn thiết kế hệ thống, người phỏng vấn thường quan tâm bạn hiểu cách đánh đổi giữa hai yếu tố này ra sao. Chương này sẽ giúp bạn nắm vững khái niệm về nhất quán và khả dụng một cách đơn giản nhất. Chúng ta cũng sẽ tìm hiểu các mô hình nhất quán khác nhau (mạnh (**strong**), cuối cùng (**eventual**), yếu (**weak**)), các cơ chế kỹ thuật như nhân bản dữ liệu (**replication**), chuyển đổi dự phòng (**failover**), **quorum**, và định lý nổi tiếng **CAP**. Quan trọng không kém, chương này nhấn mạnh những đánh đổi phải chấp nhận trong thiết kế hệ thống: bạn không thể có mọi thứ hoàn hảo, cũng như gợi ý bạn vài cách trình bày khéo léo về chủ đề này khi phỏng vấn.

Hãy cùng bắt đầu bằng việc hiểu rõ hai khái niệm nền tảng: nhất quán và khả dụng.

Tính nhất quán vs Tính khả dụng

Tính nhất quán (**Consistency**) trong ngữ cảnh hệ thống phân tán được hiểu là mọi nút (node) hoặc mọi thành phần trong hệ thống đều nhìn thấy cùng một dữ liệu tại một thời điểm. Nói cách khác, sau khi một dữ liệu được cập nhật ở một nơi, tất cả các nơi khác cũng phải có dữ liệu mới đó ngay lập tức để đảm bảo hệ thống "nhất quán". Nếu đọc dữ liệu từ nhiều máy khác nhau mà kết quả luôn như nhau (đặc biệt là luôn là giá trị mới nhất vừa được cập nhật), thì hệ thống được coi là nhất quán mạnh. Ví dụ đơn giản: bạn và một người bạn cùng mở một tài liệu chung; nếu bạn chỉnh sửa nội dung và ngay lập tức người bạn thấy thay đổi đó, nghĩa là hệ thống chia sẻ tài liệu của bạn đã duy trì tính nhất quán rất cao.

Tính khả dụng (**Availability**) thể hiện khả năng hệ thống luôn sẵn sàng phục vụ mỗi khi có yêu cầu. Một hệ thống khả dụng cao là hệ thống luôn phản hồi mọi yêu cầu hợp lệ từ người dùng một cách nhanh chóng, không bị "đơ" hay từ chối dù có một số thành phần bị lỗi. Hiểu nôm na, khả dụng đồng nghĩa với thời gian hoạt động (uptime) gần như liên tục và người dùng luôn có thể truy cập dịch vụ. Ví dụ quen thuộc: một máy ATM ngân hàng luôn hoạt động 24/7 để rút tiền: đó là tính khả dụng (dù đôi khi máy ATM có thể hiển thị số dư cũ nếu hệ thống bên dưới chưa kịp đồng bộ, nhưng nó vẫn phục vụ rút tiền, tức là ưu tiên khả dụng).

Trong thiết kế hệ thống phân tán, nhất quán và khả dụng thường mâu thuẫn ở mức độ nào đó. Để duy trì dữ liệu nhất quán tuyệt đối, đôi khi hệ thống phải hy sinh tính khả dụng (ví dụ: tạm ngừng dịch vụ để đồng bộ dữ liệu). Ngược lại, để hệ thống luôn sẵn sàng, ta có thể phải chấp nhận dữ liệu chưa hoàn toàn cập nhật (nhất quán giảm đi). Nhiệm vụ của kỹ sư thiết kế hệ thống là tìm ra điểm cân bằng phù hợp giữa hai yếu tố này dựa trên yêu cầu cụ thể của ứng dụng.

Sau đây, chúng ta sẽ đi sâu hơn vào các mô hình nhất quán dữ liệu thông dụng, từ mạnh đến yếu, kèm ví dụ minh họa để hiểu rõ hơn sự đánh đổi.

Các mô hình nhất quán (Consistency Models)

Trong hệ thống phân tán, có nhiều mô hình (cấp độ) nhất quán khác nhau. Chúng quyết định cách dữ liệu được cập nhật và nhìn thấy giữa các thành phần của hệ thống. Chúng ta sẽ tìm hiểu ba mô hình phổ biến theo thứ tự từ chặt chẽ nhất đến linh hoạt hơn: nhất quán mạnh, nhất quán cuối cùng, và nhất quán yếu. Mỗi mô hình sẽ có ví dụ minh họa và tình huống ứng dụng phù hợp.

Nhất quán mạnh (Strong Consistency)

Nhất quán mạnh đảm bảo rằng ngay sau khi một thao tác ghi (cập nhật dữ liệu) hoàn tất, tất cả các lần đọc sau đó đều sẽ nhận được dữ liệu vừa cập nhật. Nói cách khác, hệ thống không cho phép đọc ra dữ liệu cũ sau khi đã có cập nhật mới. Điều này thường đòi hỏi cơ chế khóa hoặc đồng bộ chặt chẽ giữa các nút dữ liệu: mọi cập nhật phải được phân phối đến tất cả các nút gần như đồng thời trước khi cho phép bất kỳ ai đọc tiếp.

Ví dụ thực tế: Hãy hình dung một hệ thống đặt vé máy bay. Giả sử chỉ còn 1 ghế trống duy nhất cho chuyến bay và có hai đại lý đang bán vé. Với tính nhất quán mạnh, ngay khi đại lý A bán chiếc ghế đó cho khách hàng của mình, toàn bộ hệ thống sẽ lập tức cập nhật rằng không còn ghế trống. Nếu ngay sau đó đại lý B kiểm tra, họ sẽ thấy ghế đã hết

và không thể bán trùng. Hệ thống đã đảm bảo mọi nơi đều thấy cùng một sự thật (ghế đã bán) ngay lập tức. Điều này ngăn chặn trường hợp bán hai vé cho cùng một chỗ ngồi: một lỗi nghiêm trọng. Tương tự, trong ngân hàng, khi bạn rút tiền từ ATM, hệ thống ngân hàng sử dụng nhất quán mạnh để đảm bảo số dư cập nhật ngay tức thì trên toàn hệ thống: bạn rút tiền ở ATM thì ứng dụng mobile banking của bạn cũng phải hiển thị số dư giảm ngay, tránh bạn tiêu xài "khống" số tiền không còn.

Khi nào chọn nhất quán mạnh: Mô hình này phù hợp cho các hệ thống tài chính, ngân hàng, đặt chỗ hoặc các ứng dụng yêu cầu dữ liệu chính xác tuyệt đối tại mọi thời điểm. Bất kỳ sự sai lệch hoặc trễ nào trong dữ liệu đều không chấp nhận được trong những lĩnh vực này. Tuy nhiên, đánh đổi lớn nhất là độ trễ: hệ thống có thể phải chậm lại một chút để bảo đảm mọi bản sao dữ liệu đồng bộ, và nếu một nút không cập nhật kịp, hệ thống có thể phải chờ hoặc ngừng phục vụ nút đó (giảm khả dụng). Trong ví dụ đặt vé máy bay, hệ thống nhất quán mạnh có thể khiến đại lý B phải chờ khóa dữ liệu trong vài giây khi đại lý A đang xử lý giao dịch, nhưng đổi lại, đảm bảo không xảy ra lỗi double-booking.

Nhất quán cuối cùng (Eventual Consistency)

Nhất quán cuối cùng (**eventual consistency**) cho phép hệ thống tạm thời có dữ liệu không đồng bộ giữa các nút, miễn là về lâu dài (sau một khoảng thời gian) mọi nút cuối cùng sẽ đồng bộ và thấy dữ liệu giống nhau. Nói cách khác, sau khi một cập nhật diễn ra, có thể một số lần đọc ngay sau đó vẫn thấy dữ liệu cũ, nhưng nếu không có cập nhật mới nào nữa, toàn bộ hệ thống sẽ dần dần hợp nhất về trạng thái mới nhất. Đây là một dạng của nhất quán yếu, chấp nhận độ trễ trong đồng bộ nhằm đổi lấy tính khả dụng cao hơn và hiệu suất tốt hơn.

Ví dụ thực tế: Để dễ hiểu khái niệm này, hãy tưởng tượng bạn có thói quen lưu ảnh vào cả laptop cá nhân và một ổ cứng di động để dự phòng. Mỗi tối thứ Sáu, bạn đồng bộ (sao lưu) laptop sang ổ cứng. Giả sử tối thứ Bảy bạn chụp thêm nhiều ảnh mới lưu trên laptop, nhưng chưa kịp sao lưu sang ổ cứng. Chủ Nhật, bạn của bạn mượn ổ cứng di động để xem ảnh. Vì ổ cứng chưa được cập nhật từ tối thứ Sáu, nên ảnh mới chụp hôm Thứ Bảy sẽ không có trên ổ cứng: đó là dữ liệu cũ (không nhất quán vào thời điểm đó). Tuy nhiên, đến tối Chủ Nhật bạn cắm ổ cứng vào laptop và đồng bộ, tất cả ảnh mới sẽ được chép sang ổ cứng. Lúc này, ổ cứng "bắt kịp" laptop và dữ liệu lại nhất quán giữa hai nơi. Đây chính là tính nhất quán cuối cùng: dữ liệu sẽ nhất quán vào "cuối cùng", sau một độ trễ nhất định.

Một ví dụ khác trong thế giới phần mềm là mạng xã hội. Khi bạn cập nhật ảnh đại diện (avatar), có thể mất vài giây để ảnh mới hiển thị khắp mọi nơi. Ngay sau khi bạn đổi ảnh,

một số máy chủ hoặc bạn bè của bạn có thể tạm thời vẫn thấy ảnh cũ (do bộ nhớ đệm cache chưa hết hạn hoặc server chưa đồng bộ). Tuy nhiên, sau một thời gian ngắn, ảnh mới của bạn sẽ xuất hiện với tất cả mọi người. Hệ thống chấp nhận việc hiển thị thông tin cũ trong thời gian ngắn để đổi lại việc trang web luôn tải nhanh và không bị "khựng". Đối với một mạng xã hội, điều này chấp nhận được vì ảnh đại diện hiển thị chậm vài giây không gây hậu quả nghiêm trọng. Đó chính là eventual consistency: cuối cùng dữ liệu sẽ cập nhật, chỉ là không nhất thiết ngay lập tức.

Khi nào chọn nhất quán cuối cùng: Mô hình này được dùng rộng rãi trong các hệ thống ưu tiên hiệu năng cao, khả năng mở rộng và chịu tải, nơi một chút độ trễ trong đồng bộ dữ liệu là chấp nhận được. Ví dụ: các cơ sở dữ liệu NoSQL phân tán như Cassandra, DynamoDB thường tuân theo eventual consistency nhằm đạt tốc độ ghi/đọc nhanh hơn và khả dụng cao hơn. Các dịch vụ mạng xã hội, hệ thống nội dung (như cache web, CDN) cũng chấp nhận eventual consistency, miễn là sau một thời gian ngắn dữ liệu đúng sẽ đến nơi. Lựa chọn này hy sinh tính nhất quán tức thì để đổi lấy khả dụng và hiệu năng. Dĩ nhiên, ta chỉ nên chọn khi yêu cầu ứng dụng cho phép dữ liệu cũ tồn tại trong thời gian ngắn mà không gây hậu quả lớn. Nếu dữ liệu yêu cầu chính xác tuyệt đối tại mọi thời điểm (như số dư ngân hàng), ta không nên dùng mô hình này.

Nhất quán yếu (Weak Consistency)

Nhất quán yếu là thuật ngữ chung chỉ các hệ thống không đảm bảo tính nhất quán mạnh. Trên thực tế, eventual consistency chính là một dạng của nhất quán yếu. Trong mô hình nhất quán yếu, hệ thống có thể cho phép những giai đoạn dữ liệu không đồng bộ kéo dài hơn, hoặc thậm chí không đảm bảo tất cả các bản sao sẽ đồng bộ hoàn toàn nếu không có điều kiện nhất định. Nói một cách đơn giản, nhất quán yếu chấp nhận việc đọc có thể trả về dữ liệu cũ và không hứa hẹn một thời hạn cụ thể nào để dữ liệu trở nên nhất quán giữa các node.

Ví dụ thực tế: Một ví dụ gần gũi về nhất quán yếu là hệ thống cache dữ liệu không được cập nhật thường xuyên. Giả sử trang web của bạn hiển thị danh sách sản phẩm và để tăng tốc độ, bạn lưu cache danh sách này trong 60 giây. Khi thêm một sản phẩm mới vào cơ sở dữ liệu, hệ thống có thể mất tối đa 60 giây để cache hết hạn và người dùng thấy sản phẩm mới. Trong khoảng thời gian đó, một số người dùng thấy danh sách cũ (thiếu sản phẩm mới), dữ liệu họ thấy không nhất quán với thực tế hiện tại. Đây là chấp nhận một mức độ nhất quán yếu vì ưu tiên việc trang web tải nhanh. Khác với eventual consistency (đảm bảo cuối cùng sẽ thấy dữ liệu đúng khi đồng bộ xong), ví dụ cache 60s này kỹ thuật mà nói cũng là eventual (sau 60s sẽ nhất quán), nhưng ta có thể tưởng tượng trường hợp cache không có hạn (hoặc rất lâu mới cập nhật) thì hệ thống gần như

không đảm bảo khi nào dữ liệu mới được thấy. Đó là tình huống cực đoan của nhất quán yếu: dữ liệu có thể rất lâu mới đồng bộ, hoặc chỉ đồng bộ một phần.

Một trường hợp khác: trong một game trực tuyến, bảng xếp hạng điểm số của người chơi có thể không cập nhật ngay theo thời gian thực. Điểm số của bạn bè có thể chỉ làm mới mỗi vài phút. Nếu bạn đạt điểm cao mới, bạn bè có thể chưa thấy điểm đó ngay trong bảng xếp hạng lúc này: hệ thống game ưu tiên xử lý trò chơi mượt mà (khả dụng) hơn là cập nhật điểm liên tục. Mô hình này cũng chấp nhận nhất quán yếu ở mức độ ứng dụng cho phép.

Lưu ý: Thực tế, phần lớn hệ thống eventual consistency cũng được xếp vào loại nhất quán yếu, nên hai khái niệm này thường đi đôi. Hiếm khi một hệ thống chấp nhận "yếu" hơn eventual (vì eventual ít nhất có đảm bảo cuối cùng đồng bộ). Tuy nhiên, thuật ngữ "nhất quán yếu" nhắc chúng ta rằng còn nhiều mô hình nhất quán khác giữa hai thái cực mạnh và yếu. Ví dụ: nhất quán theo phiên (session consistency) đảm bảo trong phiên làm việc của một người dùng, họ thấy dữ liệu nhất quán với chính họ; hoặc nhất quán đọc sơ bộ (read-your-writes) đảm bảo sau khi bạn ghi dữ liệu, bạn sẽ đọc thấy ngay dữ liệu đó (dù người khác có thể chưa thấy). Những biến thể này đều nằm trong phổ "nhất quán yếu hơn mạnh". Tùy hệ thống mà kiến trúc sư sẽ chọn mức độ phù hợp.

Tổng kết lại về các mô hình nhất quán:

- Mạnh: Dữ liệu luôn cập nhật mới nhất mọi nơi, nhưng đòi hỏi chờ đồng bộ, có thể giảm tốc độ hoặc ngưng phục vụ tạm thời để đảm bảo nhất quán.

- Cuối cùng: Dữ liệu sẽ cập nhật đồng bộ đến tất cả, nhưng có độ trễ. Hệ thống chạy nhanh hơn, luôn sẵn sàng nhưng chấp nhận tạm thời không đồng bộ.

- Yếu: Dữ liệu có thể không đồng bộ trong khoảng thời gian dài, thậm chí không có đảm bảo chặt chẽ sẽ đồng bộ hoàn toàn nếu không có cơ chế bổ sung. Thường chỉ chọn khi yêu cầu nhất quán không cao.

Việc lựa chọn mô hình nào phụ thuộc vào bài toán cụ thể. Hiểu rõ các mức nhất quán giúp chúng ta thiết kế hệ thống phù hợp yêu cầu: chỗ nào cần chính xác cao, chỗ nào có thể "mềm dẻo" để đổi lấy hiệu suất.

Cơ chế đảm bảo tính nhất quán và khả dụng

Để xây dựng một hệ thống vừa đáp ứng mức nhất quán dữ liệu mong muốn, vừa đảm bảo khả dụng cao, các kỹ sư thường sử dụng nhiều cơ chế kỹ thuật. Dưới đây là một số cơ chế phổ biến và cách chúng ảnh hưởng đến nhất quán/khả dụng:

Nhân bản dữ liệu (Replication)

Nhân bản dữ liệu (replication) là quá trình sao chép dữ liệu từ nơi này sang nơi khác (ví dụ từ một máy chủ chính sang nhiều máy chủ phụ) nhằm cải thiện tính khả dụng và khả năng chịu lỗi của hệ thống. Khi dữ liệu được nhân bản ra nhiều bản, hệ thống có bản dự phòng khi một máy gặp sự cố, và cũng có thể phục vụ nhiều người dùng hơn (do các máy phụ chia tải đọc dữ liệu).

Ví dụ dễ hình dung: bạn có một cuốn sổ ghi chép quan trọng. Để phòng trường hợp mất sổ, bạn photo thêm vài bản và đặt ở nhiều nơi. Nếu lỡ một bản bị thất lạc, bạn vẫn còn bản khác: đây là khả dụng (bạn luôn có sổ để đọc). Tương tự, trong hệ thống, nếu một máy chủ dữ liệu bị hỏng, vẫn còn các máy chủ khác có dữ liệu để phục vụ người dùng.

Tuy nhiên, nhân bản đặt ra thách thức về tính nhất quán. Vì có nhiều bản sao dữ liệu, làm sao để đảm bảo tất cả bản sao đều giống nhau? Mỗi khi dữ liệu thay đổi, ta phải cập nhật các bản sao. Có hai cách tiếp cận chính:

- Đồng bộ chặt (**Strong replication**): Dữ liệu được cập nhật tới tất cả các bản sao ngay lập tức trước khi chấp nhận kết quả. Cách này đảm bảo nhất quán mạnh (mọi bản sao y hệt nhau mọi lúc) nhưng có thể làm chậm hệ thống, vì phải chờ cập nhật xong hết mọi nơi.

- Đồng bộ lỏng (**Asynchronous replication**): Dữ liệu được cập nhật trên một bản chính trước, sau đó lan tỏa dần sang các bản sao khác. Cách này nhanh hơn, hệ thống phản hồi ngay cho người dùng sau khi ghi vào bản chính, cải thiện khả dụng và hiệu năng. Nhưng đổi lại, trong thời gian ngắn, các bản sao chưa kịp cập nhật sẽ bị lệch (đây chính là eventual consistency mà ta nói ở trên).

Thông thường, các hệ thống hiện đại kết hợp replication với lựa chọn mô hình nhất quán phù hợp. Ví dụ: Hệ thống ngân hàng có thể dùng replication đồng bộ (các chi nhánh ngân hàng luôn cập nhật tức thì giao dịch mới), còn hệ thống mạng xã hội dùng replication bất đồng bộ (bài đăng mới có thể vài giây sau mới xuất hiện ở mọi nơi). Nhân

bản dữ liệu rõ ràng tăng khả dụng (vì không phụ thuộc một máy), nhưng nhất quán cao hay thấp tùy thuộc vào chiến lược đồng bộ mà bạn thiết kế.

Chuyển đổi dự phòng (Failover)

Chuyển đổi dự phòng (failover) là cơ chế đảm bảo hệ thống tự động chuyển sang thành phần dự phòng khi thành phần chính gặp sự cố. Thông thường, ta có một máy chủ chính (primary) đang phục vụ và một hoặc nhiều máy chủ dự phòng (secondary/backup) sẵn sàng. Khi máy chính bị hỏng hoặc ngừng hoạt động, hệ thống sẽ failover : chuyển tiếp các yêu cầu sang máy dự phòng để tiếp tục cung cấp dịch vụ. Quá trình này thường diễn ra nhanh và tự động để người dùng hầu như không nhận ra gián đoạn.

Ví dụ: Bạn quản lý một website. Bạn có 2 server đặt ở hai nơi khác nhau, một server đang chạy, server kia ở trạng thái chờ. Đột nhiên server chính gặp sự cố mất điện, server dự phòng ngay lập tức được kích hoạt để thay thế vai trò server chính. Người dùng truy cập website có thể chỉ thấy chậm một chút rồi lại tiếp tục sử dụng bình thường : đó là nhờ cơ chế failover.

Failover chủ yếu liên quan đến khả dụng: nó giúp hệ thống luôn sẵn sàng phục vụ ngay cả khi có lỗi. Về nhất quán, failover đòi hỏi dữ liệu của máy dự phòng phải được cập nhật kịp thời so với máy chính (thường kết hợp với cơ chế replication ở trên). Nếu máy dự phòng tụt hậu dữ liệu quá xa máy chính, khi failover xảy ra người dùng có thể thấy dữ liệu cũ: đây là điều cần tránh. Vì vậy, trong thiết kế failover, người ta cố gắng đảm bảo đồng bộ dữ liệu thường xuyên giữa primary và secondary (ví dụ như mô hình master-slave trong cơ sở dữ liệu). Tóm lại, failover giúp tăng tính khả dụng, và để giữ tính nhất quán khi failover, cần kết hợp với chiến lược đồng bộ dữ liệu phù hợp.

Quorum (Đa số phiếu)

Quorum trong hệ thống phân tán nghĩa là một số lượng tối thiểu các nút phải đồng ý thì một thao tác mới được chấp nhận. Ý tưởng “đa số phiếu” này thường được áp dụng trong việc đọc/ghi dữ liệu trên các cụm máy chủ để cân bằng giữa nhất quán và khả dụng.

Hãy hình dung một nhóm 5 người đang quyết định thời gian họp. Nếu áp dụng nguyên tắc quorum với yêu cầu ít nhất 3 người đồng ý, thì chỉ khi nào 3/5 người thống nhất thời gian, quyết định mới được chấp nhận. Tương tự, trong một hệ thống có nhiều bản sao dữ liệu (ví dụ 5 bản sao), ta có thể quy định một ngưỡng quorum cho phép:

- Một thao tác ghi dữ liệu phải được xác nhận bởi ít nhất W bản sao (ví dụ W = 3 trong 5 bản sao).

- Một thao tác đọc dữ liệu phải đọc từ ít nhất R bản sao (ví dụ R = 3).

Miễn sao W + R > N (trong đó N là tổng số bản sao, ở ví dụ trên N=5 thì cần W+R > 5), ta đảm bảo rằng tập các bản sao tham gia đọc và viết luôn chồng chéo lên nhau. Nhờ đó, bất kỳ dữ liệu mới nào được ghi cũng sẽ có ít nhất một bản sao nằm trong tập được đọc, đảm bảo người đọc thấy dữ liệu mới nhất. Đây chính là nguyên tắc giúp đạt nhất quán mạnh hơn mà vẫn cho phép hệ thống hoạt động phân tán.

Ví dụ kỹ thuật: Hệ thống cơ sở dữ liệu NoSQL như Cassandra cho phép ta cấu hình thông số W và R này (gọi là mức độ nhất quán có thể điều chỉnh, tunable consistency). Nếu ta chọn W = R = (N/2 + 1) (tức hơn một nửa số node), thì hệ thống sẽ yêu cầu đa số node cập nhật/lấy dữ liệu, nhờ đó đạt độ nhất quán cao (gần với strong consistency). Ngược lại, ta có thể chọn W nhỏ hơn để tốc độ ghi nhanh hơn (ít node cần xác nhận) hoặc R nhỏ hơn để đọc nhanh hơn: đánh đổi là nhất quán sẽ yếu hơn (có thể đọc phải node chưa kịp cập nhật).

Tóm lại, quorum là cách để "điều chỉnh" giữa hai thái cực nhất quán và khả dụng. Bằng cách chọn ngưỡng bao nhiêu node phải đồng thuận, ta có thể tăng nhất quán (yêu cầu nhiều phiếu hơn) hoặc tăng tốc và khả dụng (yêu cầu ít phiếu hơn). Quorum thường đi kèm với các thuật toán đồng thuận (consensus algorithms) và được dùng trong các hệ thống phân tán cần độ tin cậy cao.

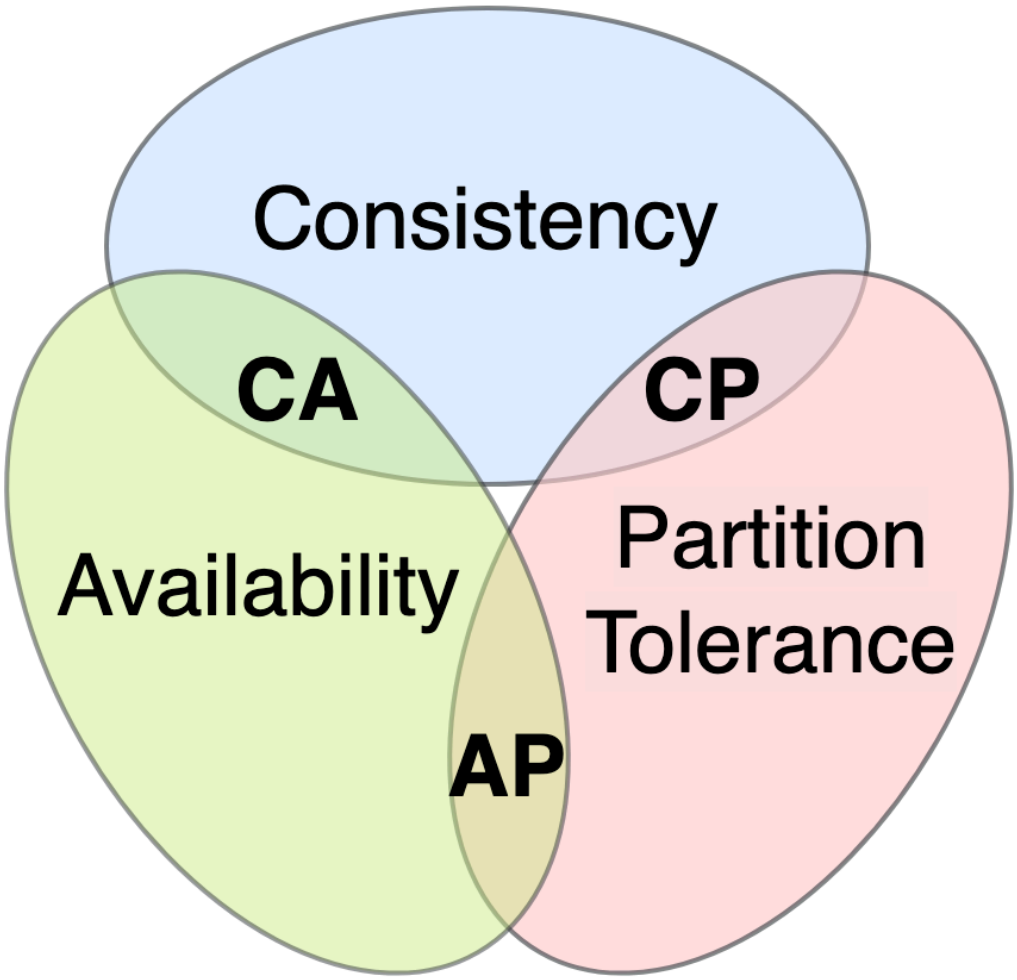
Định lý CAP và các đánh đổi kinh điển

Khi bàn về nhất quán và khả dụng, không thể không nhắc đến định lý CAP - một nguyên lý kinh điển trong thiết kế hệ thống phân tán. Định lý CAP phát biểu rằng một hệ thống phân tán không thể đảm bảo đồng thời cả ba yếu tố: tính nhất quán (Consistency), tính khả dụng (Availability) và khả năng chịu phân vùng (Partition tolerance). Nói cách khác, bạn chỉ có thể chọn tối đa hai trong ba thuộc tính đó trong một hệ thống phân tán thực tế.

- Consistency (C) - Tính nhất quán: Mọi lần đọc đều nhận được dữ liệu mới nhất hoặc thông báo lỗi (nếu dữ liệu chưa kịp cập nhật).

- Availability (A) - Tính khả dụng: Mọi yêu cầu nhận được một phản hồi (không lỗi), dù dữ liệu có thể không phải mới nhất.

- Partition tolerance (P) - Khả năng chịu phân vùng: Hệ thống tiếp tục hoạt động bình thường kể cả khi các kết nối mạng giữa các node bị gián đoạn (một dạng sự cố phân vùng mạng).



Trong môi trường phân tán, lỗi phân vùng mạng (ví dụ: đứt kết nối giữa các datacenter, hoặc một node bị mất liên lạc tạm thời) sớm muộn gì cũng xảy ra. Khi phân vùng mạng xảy ra, hệ thống buộc phải đánh đổi giữa C và A:

- Lựa chọn C (nhất quán): Hệ thống sẽ từ chối một số thao tác hoặc trả về lỗi thay vì cho phép dữ liệu sai lệch. Tức là ngừng phục vụ tạm thời (giảm khả dụng) đối với các yêu cầu không thể đảm bảo nhất quán. (Ví dụ: trong trường hợp hai trung tâm dữ liệu mất kết nối, tạm ngừng cho phép cập nhật ở một bên cho đến khi kết nối phục hồi để giữ dữ liệu thống nhất).

- Lựa chọn A (khả dụng): Hệ thống vẫn tiếp tục phục vụ tất cả yêu cầu dù các node không thể giao tiếp với nhau trọn vẹn. Điều này giữ cho dịch vụ không gián đoạn, nhưng có nguy cơ dữ liệu không đồng bộ tạm thời giữa các phần của hệ thống. (Ví dụ: hai trung tâm dữ liệu vẫn cho phép cập nhật độc lập khi mất kết nối; người dùng mỗi bên vẫn thao tác được, nhưng dữ liệu có thể xung đột hoặc lệch nhau cho đến khi đồng bộ lại).

Định lý CAP về bản chất nhấn mạnh: trong trường hợp có sự cố phân vùng, bạn không thể có cả tính nhất quán và tính khả dụng 100% cùng lúc. Bạn phải hy sinh một thứ nào đó. Nếu bạn không chấp nhận hy sinh tính khả dụng (muốn hệ thống luôn phục vụ), thì bạn phải chấp nhận dữ liệu có thể không nhất quán (khi mất kết nối mạng giữa các node). Ngược lại, nếu bạn muốn dữ liệu luôn nhất quán tuyệt đối, thì khi mất kết nối, hệ thống sẽ phải từ chối hoặc trì hoãn một số yêu cầu, tức là người dùng sẽ thấy hệ thống tạm thời không sẵn sàng.

Chú ý: Khả năng chịu lỗi phân vùng (P) thường được xem như một điều kiện bắt buộc trong hệ thống phân tán hiện đại : bởi lẽ không có hệ thống nào tránh được hoàn toàn nguy cơ phân vùng mạng. Do đó, thực tế triển khai, các kỹ sư thường coi P luôn phải có, và vì thế lựa chọn thường là giữa CP (Consistency + Partition tolerance) hoặc AP (Availability + Partition tolerance), tùy theo mục tiêu hệ thống:

- Hệ thống CP: Ưu tiên nhất quán, chấp nhận giảm khả dụng khi lỗi mạng. Ví dụ: các dịch vụ ngân hàng, hoặc các cơ sở dữ liệu SQL truyền thống thường nghiêng về CP: nếu không đảm bảo cập nhật đồng bộ giữa các node thì tạm dừng giao dịch và trả lỗi còn hơn là để sai lệch dữ liệu.

- Hệ thống AP: Ưu tiên khả dụng, chấp nhận dữ liệu không đồng nhất tạm thời giữa các node. Nhiều hệ thống NoSQL (BASE) hoặc mạng xã hội lựa chọn AP: họ chấp

nhận eventual consistency để đổi lấy việc hệ thống luôn online phục vụ.

Hãy xem một tình huống đời thường để hiểu CAP: Bạn có hai cửa hàng (hai chi nhánh) bán chung một kho hàng. Đột nhiên đường truyền mạng giữa hai cửa hàng bị mất (phân vùng). Lúc này:

- Nếu bạn muốn dữ liệu kho nhất quán giữa hai bên, bạn buộc phải ngừng bán ở một chi nhánh (giảm khả dụng) cho đến khi mạng nối lại và kho hàng được đồng bộ. Khách đến cửa hàng đó phải chờ hoặc quay lại sau: cửa hàng tạm thời mất doanh thu nhưng đảm bảo không bán quá số hàng có thực.

- Nếu bạn muốn tiếp tục bán bình thường ở cả hai nơi (khả dụng cao), mỗi bên sẽ tạm thời hoạt động với dữ liệu kho riêng (vì không liên lạc được với nhau). Khả năng xảy ra là có thể bán trùng một số sản phẩm: ví dụ chi nhánh A và B cùng bán chiếc laptop cuối cùng, dẫn đến dữ liệu không nhất quán (tổng số bán ra nhiều hơn số hàng thật). Sau khi mạng phục hồi, bạn sẽ phải giải quyết sự chênh lệch này (ví dụ xin lỗi khách hoặc nhập thêm hàng).

Không giải pháp nào hoàn hảo: hoặc mất doanh thu tạm thời nhưng dữ liệu an toàn, hoặc phục vụ trơn tru nhưng đối mặt rủi ro sai lệch. Đây chính là sự đánh đổi CAP trong thực tế.

Cách trình bày về nhất quán và khả dụng khi phỏng vấn

Khi vào phòng phỏng vấn thiết kế hệ thống, rất có thể bạn sẽ được hỏi về việc đảm bảo dữ liệu nhất quán hoặc xử lý tình huống mất kết nối, downtime. Dưới đây là một số gợi ý giúp bạn trình bày khéo léo và đầy đủ về chủ đề nhất quán vs khả dụng:

- **Bắt đầu bằng việc xác định yêu cầu:** Hãy làm rõ với người phỏng vấn rằng bài toán đang hướng tới điều gì. Nếu họ hỏi “Làm sao để đảm bảo tính nhất quán dữ liệu trong hệ thống X?”, trước hết bạn nên xác nhận xem trong hệ thống X, mức nhất quán nào thực sự cần thiết. Bạn có thể nói: “Trước tiên, cho phép tôi xác nhận: đối với hệ thống X, ta ưu tiên dữ liệu chính xác tuyệt đối (nhất quán mạnh) hay chấp nhận một độ trễ cập nhật (nhất quán cuối cùng) để có hiệu năng tốt hơn?”. Việc đặt câu hỏi này thể hiện bạn hiểu về trade-off và muốn làm rõ yêu cầu.

- **Nhấn mạnh việc không thể có "mọi thứ":** Thể hiện sự hiểu biết của bạn về định lý CAP bằng cách giải thích ngắn gọn rằng trong một hệ thống phân tán, không thể đồng thời đạt 100% nhất quán, 100% khả dụng khi xảy ra sự cố mạng. Ví dụ, bạn có thể nói: "Em biết trong hệ thống phân tán luôn phải đánh đổi giữa tính nhất quán và tính sẵn sàng. Với hệ thống này, nếu ta chọn dữ liệu luôn đúng mọi lúc, có thể phải hy sinh chút uptime khi có sự cố, và ngược lại." Cách trình bày này cho thấy bạn nắm kiến thức nền và biết áp dụng vào thực tế.

- **Đưa ra giải pháp cụ thể kèm lý do:** Tùy vào yêu cầu đã xác định, bạn đề xuất giải pháp thiết kế phù hợp. Nếu phỏng vấn viên muốn nhất quán cao, bạn có thể đề xuất: "Em sẽ sử dụng cơ chế quorum cho các thao tác ghi/đọc, đảm bảo đa số node xác nhận một cập nhật trước khi coi là thành công. Điều này giúp dữ liệu các node đồng bộ hơn, đạt gần như strong consistency. Ngoài ra, em sẽ thiết kế một leader để serial hóa các cập nhật, tránh xung đột." Ngược lại, nếu chấp nhận eventual consistency, bạn có thể nói: "Ta có thể dùng mô hình master-slave replication bất đồng bộ. Mọi ghi vào master được phản hồi ngay (tăng tốc độ), sau đó dữ liệu sẽ lan sang các slave. Với kiểu thiết kế này, người dùng luôn ghi được dữ liệu (hệ thống rất sẵn sàng), chỉ có điều có thể mất vài giây để tất cả các máy thấy dữ liệu mới." Quan trọng là bạn giải thích tại sao chọn giải pháp đó dựa trên ưu tiên của hệ thống (nhanh vs. chính xác).

- **Đề cập đến các cơ chế hỗ trợ:** Khi nói về giải pháp, nhớ lồng ghép các từ khóa kỹ thuật như đã học trong chương này: replication, failover, quorum, cache... Ví dụ: "Để cải thiện khả dụng, em sẽ triển khai failover giữa hai server đặt ở hai trung tâm dữ liệu khác nhau. Nếu một bên gặp sự cố, bên kia sẽ tiếp quản ngay : người dùng không bị gián đoạn." Hoặc: "Em sẽ cân nhắc dùng cache với TTL ngắn để giảm tải database, nhưng vẫn đảm bảo eventual consistency: dữ liệu cache sẽ làm mới sau mỗi 5 phút để không lệch quá lâu." Việc nêu ra các thuật ngữ này một cách đúng chỗ cho thấy bạn hiểu bức tranh tổng thể và biết áp dụng nhiều công cụ để đạt mục tiêu thiết kế.

- **Sử dụng ví dụ đơn giản nếu cần thiết:** Đôi khi, để đảm bảo người phỏng vấn hiểu rõ ý bạn (và cũng để bạn ghi điểm về kỹ năng truyền đạt), bạn có thể đưa một ví dụ ngắn. Chẳng hạn: "Ta có thể hình dung hệ thống như một nhóm người: nếu tất cả đều phải đồng ý mới quyết định (quorum toàn bộ) thì rất nhất quán nhưng sẽ chậm ra kết quả; nếu chỉ cần một vài người đồng ý (quorum nhỏ) thì quyết định nhanh hơn nhưng có khi không phản ánh ý kiến của tất cả : hệ thống phân tán cũng vậy." Một ví dụ đời thường, ngắn gọn sẽ giúp lời giải thích của bạn thêm ấn tượng và dễ hiểu. Tuy nhiên, hãy chắc chắn ví dụ đó phục vụ cho việc làm rõ ý

chứ không lan man.

- **Thể hiện sự linh hoạt và tập trung vào người dùng:** Nhấn mạnh rằng lựa chọn nhất quán vs khả dụng phải phù hợp với nhu cầu người dùng. Bạn có thể kết luận phần trả lời bằng cách nói: "Tóm lại, em sẽ ưu tiên giải pháp phù hợp với trải nghiệm người dùng: với tính năng quan trọng như giao dịch tiền, em chọn nhất quán mạnh; còn với tính năng bảng thông báo hay tin tức, em chọn eventual consistency để người dùng tải nhanh hơn. Luôn có đánh đổi, và em sẽ giải thích rõ cho team biết để ra quyết định đúng." Lời kết này vừa cho thấy bạn nắm vững kiến thức, vừa chứng tỏ bạn biết đặt người dùng và yêu cầu thực tế lên hàng đầu.

Nhìn chung, khi thảo luận về Consistency vs Availability trong phỏng vấn, hãy nhớ ba điều: hiểu nhu cầu, biết đánh đổi, và giải thích mạch lạc.

Lựa chọn Cơ sở dữ liệu, Sharding và Tối ưu hóa lưu trữ

Trong một hệ thống lớn, dữ liệu đóng vai trò trung tâm. Việc lựa chọn cách lưu trữ và tổ chức dữ liệu phù hợp có thể quyết định thành bại của thiết kế hệ thống. Chương này sẽ tập trung vào các khía cạnh chính trong thiết kế data cho phỏng vấn System Design, bao gồm: chọn SQL hay NoSQL, mô hình phân mảnh dữ liệu (sharding), chiến lược indexing cơ bản, các yếu tố cân nhắc khi chọn kiểu lưu trữ, mối quan hệ giữa dữ liệu phân tán và tính nhất quán, và cuối cùng là một số gợi ý cách trình bày về phần database design khi đi phỏng vấn. Văn phong sẽ gần gũi, kèm ví dụ minh họa để bạn : những kỹ sư junior đến middle : dễ dàng tiếp cận.

SQL vs NoSQL: lựa chọn cơ sở dữ liệu phù hợp

Cơ sở dữ liệu quan hệ (SQL) và NoSQL là hai lựa chọn phổ biến nhất để lưu trữ dữ liệu ứng dụng. Mỗi loại có đặc điểm riêng, phù hợp với những trường hợp khác nhau:

- **SQL (Relational Database):** Dữ liệu được lưu trong bảng (table) có dòng và cột cố định, các bảng có thể liên kết với nhau bằng khóa. CSDL SQL yêu cầu schema chặt chẽ (cấu trúc bảng phải được định nghĩa trước) và thường tuân thủ các nguyên tắc ACID để đảm bảo tính toàn vẹn dữ liệu. SQL hỗ trợ các JOIN phức tạp giữa bảng, phù hợp để lưu trữ dữ liệu có mối quan hệ chặt chẽ và cần tính nhất

quán cao. Ví dụ, hệ thống ngân hàng hoặc cửa hàng trực tuyến thường dùng SQL vì dữ liệu có cấu trúc rõ ràng và cần độ chính xác tuyệt đối. Nhược điểm: SQL truyền thống khó mở rộng ngang (scale-out): thường phải nâng cấp máy chủ (scale-up) khi dữ liệu lớn dần, và việc thay đổi schema có thể phức tạp.

- **NoSQL (Non-relational Database):** Dữ liệu được lưu trữ dưới nhiều hình thức linh hoạt như document JSON, key-value, column-family hoặc graph. NoSQL không đòi hỏi schema cố định, cho phép lưu dữ liệu bán cấu trúc hoặc không có cấu trúc một cách linh hoạt. Hệ thống NoSQL thường hy sinh một phần thuộc tính ACID (đặc biệt là tính nhất quán ngay lập tức) để đạt hiệu năng và khả năng mở rộng cao hơn. Ưu điểm: dễ dàng mở rộng horizontally bằng cách thêm máy (scale-out), phù hợp với dữ liệu lớn và yêu cầu tốc độ cao. Ví dụ, mạng xã hội, hệ thống quản lý nội dung hoặc log sự kiện thường dùng NoSQL để lưu những dữ liệu đa dạng, schema linh hoạt và có thể chịu được việc dữ liệu nhất quán cuối cùng (eventual consistency). Nhược điểm: hạn chế trong việc thực hiện các truy vấn phức tạp và JOIN giữa các tập dữ liệu, đồng thời lập trình viên phải tự đảm bảo một số tính toàn vẹn nếu cần.

Khi nào chọn SQL, khi nào chọn NoSQL? Không có câu trả lời tuyệt đối, nhưng vài gợi ý chung: nếu ứng dụng của bạn có dữ liệu quan hệ chặt chẽ, schema rõ ràng, cần giao dịch (transaction) đảm bảo tính nhất quán cao (ví dụ ứng dụng tài chính, ngân hàng): SQL là lựa chọn phù hợp. Ngược lại, nếu dữ liệu linh hoạt, không có cấu trúc cố định, có khả năng mở rộng nhanh chóng, hoặc không đòi hỏi tất cả thao tác phải nhất quán tức thì (ví dụ mạng xã hội, hệ thống phân tích dữ liệu, logging), NoSQL có thể là lựa chọn tốt. Trong thực tế, nhiều hệ thống kết hợp cả hai: dùng SQL cho phần cốt lõi cần tính toán chính xác, và NoSQL cho phần mở rộng, lưu trữ dữ liệu lớn hoặc caching.

Ví dụ gần gũi: Hãy hình dung bạn xây dựng một ứng dụng quản lý chi tiêu cá nhân. Dữ liệu người dùng, số dư, giao dịch tiền tệ: những thứ cần tính đúng đắn và ACID có thể lưu trong SQL (như PostgreSQL hoặc MySQL). Nhưng ứng dụng của bạn cũng cần lưu nhật ký hoạt động, hoặc thông tin để gợi ý khuyến mãi (vốn không quan trọng nếu mất vài bản ghi hoặc không cần cấu trúc cố định): bạn có thể sử dụng một NoSQL (như MongoDB) để lưu những thông tin này, tận dụng khả năng ghi nhanh và mở rộng linh hoạt của NoSQL. Việc kết hợp đúng công cụ với đặc thù dữ liệu sẽ giúp hệ thống của bạn vừa chặt chẽ ở lõi quan trọng, vừa linh hoạt và scalable ở các phần khác.

Phân mảnh dữ liệu (Sharding/Partitioning)

Khi dữ liệu và lượng người dùng tăng lên, một máy chủ cơ sở dữ liệu đơn lẻ có thể trở thành điểm nghẽn (bottleneck). Phân mảnh dữ liệu (data sharding) là kỹ thuật chia nhỏ cơ sở dữ liệu thành nhiều phần độc lập (gọi là shard hoặc partition) và phân bổ chúng trên nhiều máy chủ khác nhau. Mỗi máy chứa một phần dữ liệu, nhờ đó hệ thống có thể xử lý nhiều yêu cầu song song, cải thiện thông lượng và giảm tải cho từng máy. Phân mảnh cũng giúp tránh sự cố toàn hệ thống: nếu một shard gặp trục trặc, các shard khác vẫn hoạt động, hệ thống không bị sập hoàn toàn.

Có nhiều chiến lược sharding khác nhau, phổ biến nhất gồm:

- **Sharding theo phạm vi (Range-based):** Chia dữ liệu dựa trên khoảng giá trị của một khóa. Ví dụ: nếu ta shard theo ID khách hàng, có thể quy định shard 1 chứa các ID từ 1-1,000,000, shard 2 chứa ID 1,000,001-2,000,000, v.v. Hoặc shard theo alphabet: tên từ A-I vào shard A, J-S vào shard B, T-Z vào shard C. Ưu điểm của cách này là dễ hiểu, dễ truy vấn theo khoảng (ví dụ lấy tất cả users tên từ A đến I rất thuận tiện nếu chúng cùng shard). Tuy nhiên, nhược điểm là dữ liệu có thể phân bố không đều: một shard có thể quá nhiều dữ liệu hoặc tải nặng hơn shard khác nếu phạm vi không đồng đều (ví dụ shard chứa tên A-I có thể đông khách hàng hơn shard T-Z).

- **Sharding theo băm (Hash-based):** Sử dụng một hàm băm (hash function) trên khóa (ví dụ userID, tên người dùng) để quyết định dữ liệu thuộc shard nào. Hàm băm sẽ chuyển khóa thành một số (giá trị băm) và dùng số đó phân bổ vào các shard (ví dụ hash(id) mod 4 để chọn 1 trong 4 shard). Cách này thường giúp phân bố dữ liệu đồng đều hơn giữa các shard, tránh trường hợp một shard quá nóng. Hạn chế: do dữ liệu được chia không theo ý nghĩa liên tục, việc thêm một node mới (tăng số shard) đòi hỏi tính toán lại hàm băm hoặc sử dụng kỹ thuật băm nhất quán (consistent hashing) để hạn chế ảnh hưởng. Ngoài ra, sharding băm làm cho việc truy vấn theo khoảng (range query) khó khăn hơn, vì dữ liệu liên tiếp theo giá trị có thể nằm trên nhiều shard khác nhau.

- **Sharding theo danh mục khóa (Directory/Key-based):** Còn gọi là phân mảnh thư mục, phương pháp này dùng một bảng tra cứu (lookup table) để quyết định dữ liệu thuộc shard nào. Ví dụ: một dịch vụ có thể quy định tất cả khách hàng loại Doanh nghiệp ở shard A, Cá nhân ở shard B; hoặc một ứng dụng e-commerce có thể lưu sản phẩm điện tử ở shard X, thời trang ở shard Y. Bảng ánh xạ này đóng vai trò như directory: nhập giá trị thuộc tính, trả về chỉ mục shard tương ứng. Ưu điểm: rất linh hoạt, ta có thể quyết định phân chia theo bất kỳ tiêu chí nào phù hợp với truy vấn (theo loại sản phẩm, theo nhóm khách hàng, v.v.). Nhược điểm:

nếu bảng tra cứu bị lỗi hoặc phân loại không hợp lý, có thể gây nhầm lẫn và mất cân bằng dữ liệu giữa các shard. Việc duy trì bảng tra cứu cũng phức tạp khi hệ thống thay đổi tiêu chí sharding.

- **Sharding theo địa lý (Geo-based):** Chia dữ liệu theo vùng địa lý nhằm phục vụ người dùng tốt hơn ở các khu vực khác nhau. Thường dùng cho các hệ thống toàn cầu: ví dụ dữ liệu người dùng Châu Âu nằm trên cụm máy chủ EU, người dùng Châu Á nằm trên cụm máy chủ Asia. Điều này giúp giảm độ trễ (latency) vì người dùng truy cập máy chủ gần họ hơn. Geo-sharding cũng hỗ trợ tuân thủ quy định địa phương (như dữ liệu khách hàng EU lưu trong EU để tuân thủ GDPR). Tuy nhiên, tương tự range partition, phương pháp này có thể dẫn đến phân bố không đồng đều: nếu một khu vực có quá nhiều người dùng so với khu vực khác, cụm đó sẽ chịu tải lớn hơn. Mặt khác, thiết kế theo địa lý đòi hỏi cân nhắc về đồng bộ dữ liệu giữa các vùng.

Lưu ý: Dù chọn chiến lược sharding nào, điều quan trọng là xác định khóa phân mảnh (shard key) phù hợp. Khóa này nên là một thuộc tính dùng thường xuyên trong truy cập dữ liệu, có khả năng phân tán tương đối đều. Chọn sai shard key có thể tạo điểm nóng (hot spot): ví dụ shard chứa quá nhiều dữ liệu hoặc quá nhiều truy cập so với shard khác. Khi thiết kế hệ thống lớn, bạn nên thảo luận xem liệu cần sharding hay chưa. Ban đầu, giải pháp đơn giản (một database) có thể đủ cho MVP. Nhưng hãy thể hiện rằng bạn biết cách sharding khi quy mô dữ liệu vượt ngưỡng: đây là điểm cộng trong phỏng vấn System Design.

Ví dụ minh họa: Giả sử bạn thiết kế một hệ thống quản lý người dùng toàn cầu. Ban đầu, một database SQL duy nhất chứa tất cả user có thể chạy tốt. Nhưng khi đạt 10 triệu người dùng khắp thế giới, server bắt đầu quá tải. Bạn đề xuất: phân mảnh theo địa lý: người dùng Châu Mỹ trên cụm US, Châu Âu trên cụm EU, Châu Á trên cụm APAC. Mỗi cụm là một shard độc lập. Nhờ đó, truy vấn đăng nhập của user châu Âu sẽ đến thẳng cụm EU gần họ, nhanh hơn so với việc tất cả truy vấn dồn về một nơi. Đồng thời, bạn kết hợp phân mảnh băm trong từng cụm nếu một cụm vẫn quá lớn (ví dụ, trong cụm US, hash theo userID để chia nhỏ tiếp). Cách trình bày này cho thấy bạn biết áp dụng nhiều chiến lược sharding phối hợp để đạt hiệu quả tối đa.

Chỉ mục (Index): Tối ưu hóa truy vấn

Trong cơ sở dữ liệu, index (chỉ mục) giống như mục lục của cuốn sách: thay vì đọc tuần tự từng trang để tìm thông tin, ta tra mục lục để nhảy thẳng đến trang cần tìm. Tương tự,

index là một cấu trúc dữ liệu bổ sung giúp truy vấn dữ liệu nhanh hơn bằng cách cho phép hệ quản trị CSDL định vị nhanh các hàng cần thiết, thay vì quét toàn bộ bảng.

Cách hoạt động cơ bản: Khi tạo index trên một hoặc nhiều cột, CSDL sẽ xây dựng một cấu trúc (thường là B-Tree) sắp xếp các giá trị của cột đó cùng với con trỏ tới vị trí lưu trữ tương ứng. Nhờ được sắp xếp, việc tìm kiếm một giá trị cụ thể, hoặc tìm theo khoảng (>, <, BETWEEN) sẽ nhanh hơn rất nhiều so với duyệt tuần tự từng dòng. Một số hệ quản trị cũng hỗ trợ Hash index (chỉ hữu ích cho so sánh bằng = hoặc !=, vì hash không sắp xếp thứ tự), nhưng phổ biến nhất vẫn là B-Tree do tính đa năng (hỗ trợ cả so sánh lớn hơn/nhỏ hơn, LIKE, ORDER BY, v.v. trên cột được index).

Chiến lược sử dụng index hiệu quả: Dù index giúp tăng tốc truy vấn đọc, nhưng không nên lạm dụng : vì index tiêu tốn thêm không gian và làm chậm ghi dữ liệu (mỗi lần INSERT/UPDATE/DELETE phải cập nhật index liên quan). Dưới đây là một số hướng dẫn cơ bản:

- **Chỉ tạo index khi cần thiết:** Thường với các bảng dữ liệu trung bình đến lớn (ví dụ trên ~100k bản ghi) và cột được truy vấn thường xuyên. Nếu bảng rất nhỏ, scanning toàn bảng có khi còn nhanh hơn dùng index (do overhead của index).

- **Index các cột hay dùng trong truy vấn:** Các cột xuất hiện trong mệnh đề WHERE, các cột thường JOIN giữa các bảng, hoặc cột hay dùng để ORDER BY/GROUP BY nên được đánh index. Điều này giúp những truy vấn lọc/sắp xếp dựa trên các cột đó chạy nhanh hơn nhiều. Ví dụ: bảng users(name, email, age, city) nếu ứng dụng thường tìm user theo email hoặc tên, hãy tạo index trên các cột email và name.

- **Hạn chế index các cột có độ phân biệt thấp:** Nếu một cột chỉ có vài giá trị lặp đi lặp lại (ví dụ cột gender chỉ có “Male”/“Female”), việc index ít hiệu quả vì DB vẫn phải quét nhiều kết quả giống nhau. Tương tự, cột thường xuyên NULL hoặc dạng Boolean ít giá trị unique cũng không nên index.

- **Sử dụng index nhiều cột (composite index) cho các truy vấn kết hợp.** Ví dụ, nếu hay tìm kiếm user theo (country, name) cùng nhau, có thể tạo index kết hợp (country, name). Lưu ý: thứ tự cột trong composite index rất quan trọng : index (country, name) có thể phục vụ truy vấn theo country hoặc country+name, nhưng sẽ không hiệu quả cho truy vấn chỉ theo name (bỏ qua country). Hãy đặt cột nào thường được dùng để lọc trước trong index.

- Nhớ rằng Primary key thường tự động có index. Khi tạo khóa chính cho bảng SQL, hệ quản trị sẽ tạo index B-Tree cho khóa đó. Tương tự, các cột khai báo UNIQUE

hay khóa ngoại thường cũng được index tự động để đảm bảo nhanh chóng truy cập và kiểm tra tính duy nhất hoặc liên kết. Do đó, bạn không cần (và không nên) tạo trùng lặp index trên các cột đã là khóa chính/khóa ngoại.

- **Xóa bỏ các index không còn cần:** Thỉnh thoảng, ứng dụng thay đổi logic, có những index được tạo ra trước đây nhưng giờ ít dùng: index thừa sẽ làm chậm ghi và tốn dung lượng. Hãy soát lại và loại bỏ index không cần thiết để tối ưu hiệu năng tổng thể.

Ví dụ: Hãy tưởng tượng một bảng products (id, name, category, price, description) với hàng triệu sản phẩm. Ứng dụng thường có trang tìm kiếm sản phẩm theo tên và lọc theo danh mục. Rõ ràng, bạn nên tạo index trên cột name (để tìm kiếm text nhanh hơn) và có thể trên cột category (để lọc theo danh mục). Tuy nhiên, cột description (mô tả sản phẩm) dạng text dài, bạn không nên index toàn bộ vì sẽ rất nặng và ít tác dụng: nếu cần tìm trong mô tả thì có thể dùng giải pháp tìm kiếm khác (như ElasticSearch) thay vì index trong SQL. Thêm nữa, nếu bạn nhận thấy truy vấn hay kết hợp cả category và price (ví dụ lấy các sản phẩm giá thấp trong danh mục Điện tử), một composite index (category, price) có thể hữu ích để vừa lọc danh mục vừa sắp xếp theo giá nhanh chóng. Việc trình bày hiểu biết về index như vậy trong phỏng vấn cho thấy bạn biết cách tối ưu truy vấn cho hệ thống lớn, không chỉ dừng ở việc lưu dữ liệu.

Cân nhắc khi chọn kiểu lưu trữ cho hệ thống lớn

Mỗi hệ thống lớn thường có những yêu cầu và ràng buộc khác nhau, vì vậy việc chọn kiểu lưu trữ dữ liệu (loại database hoặc data store) phải cân nhắc nhiều yếu tố. Dưới đây là một số yếu tố chính bạn nên xem xét và đề cập khi thiết kế:

- **Tính chất workload:** OLTP vs OLAP: Hệ thống của bạn chủ yếu là xử lý giao dịch trực tuyến (nhiều thao tác đọc/ghi nhỏ, liên tục, yêu cầu đáp ứng nhanh - OLTP) hay phục vụ phân tích dữ liệu lớn (các truy vấn tính toán phức tạp trên dữ liệu tổng hợp - OLAP)? Nếu là OLTP (Online Transaction Processing): ví dụ ứng dụng web cho người dùng thao tác thường xuyên: hãy ưu tiên database tối ưu cho ghi/đọc ngẫu nhiên nhanh và ACID (thường là RDBMS hoặc Key-Value store trong một số trường hợp). Nếu là OLAP (Online Analytical Processing): ví dụ hệ thống báo cáo, kho dữ liệu, có thể cần một columnar storage hoặc data warehouse, tối ưu cho đọc theo lô lớn và tính toán tổng hợp. Đôi khi hệ thống lớn phải hỗ trợ cả hai dạng workload (HTAP : Hybrid Transaction/Analytical), khi đó có thể tách ra các thành phần khác nhau hoặc dùng giải pháp chuyên biệt (như dùng database

in-memory cho phân tích thời gian thực).

- **Mô hình dữ liệu và kiểu truy vấn:** Dữ liệu của bạn có cấu trúc như bảng biểu rõ ràng, hay là dữ liệu dạng document, hoặc có mối quan hệ phức tạp kiểu đồ thị? Hãy chọn data store phù hợp với bản chất dữ liệu:

    - Dữ liệu có nhiều bảng liên kết, quan hệ phức tạp: RDBMS (SQL) vẫn là lựa chọn tốt, vì nó sinh ra cho việc này (hỗ trợ join và đảm bảo toàn vẹn quan hệ).

    - Dữ liệu dạng cặp khóa-giá trị, truy cập chỉ qua key và không cần query phức tạp: dùng Key-Value store (VD: Redis, DynamoDB) sẽ nhanh và đơn giản hơn.

    - Dữ liệu dưới dạng JSON, XML với cấu trúc lồng nhau, linh hoạt: một Document DB (VD: MongoDB, Couchbase) sẽ phù hợp, cho phép lưu nguyên cấu trúc tài liệu và truy vấn theo thuộc tính bên trong dễ dàng.

    - Dữ liệu có mối quan hệ đồ thị (graph) như mạng xã hội (quan hệ bạn bè, follower): xem xét Graph DB (VD: Neo4j) để tận dụng tối đa mô hình đồ thị.

    - Dữ liệu là file, media lớn: database quan hệ không phù hợp để lưu ảnh, video dung lượng lớn : khi đó nên dùng Object storage (VD: AWS S3, Google Cloud Storage) cho phần dữ liệu này, kết hợp với DB lưu metadata.

- **Quy mô dữ liệu và khả năng mở rộng:** Lượng dữ liệu hiện tại và dự kiến trong 1-2 năm tới là bao nhiêu? Hệ thống có cần mở rộng ngang dễ dàng không? Nếu dự kiến rất lớn (hàng tỷ bản ghi, hàng TB dữ liệu), hãy ưu tiên giải pháp có khả năng partitioning tốt hoặc thiết kế phân tán sẵn. Ví dụ: Cassandra (NoSQL kiểu wide-column) được thiết kế để phân tán dữ liệu trên nhiều node dễ dàng; trong khi với MySQL ta có thể phải chủ động sharding thủ công khi đến ngưỡng. Ngược lại, nếu dữ liệu vừa phải và đơn máy đủ gánh, dùng một RDBMS đơn giản có khi lại đáng tin cậy và tiết kiệm hơn. Luôn cân nhắc chi phí vs lợi ích: không phải lúc nào cũng cần công nghệ “big data” phức tạp nếu quy mô chưa đến.

- **Yêu cầu về transaction và nhất quán:** Ứng dụng có cần tính nhất quán mạnh (strong consistency) không, hay chấp nhận nhất quán cuối cùng (eventual)? Nếu cần nhất quán mạnh cho các thao tác (ví dụ hệ thống ngân hàng cần số dư cập nhật tức thì trên toàn hệ thống), nên chọn những giải pháp đảm bảo ACID hoặc

chấp nhận hy sinh phần nào tính sẵn sàng để duy trì nhất quán (thường là các RDBMS hoặc NoSQL chế độ CP theo CAP). Nếu ứng dụng cho phép dữ liệu hơi trễ một chút giữa các node (ví dụ mạng xã hội, dữ liệu vài giây sau mới đồng bộ cũng không sao), có thể chọn các hệ thống AP ưu tiên availability (như Cassandra, DynamoDB) để đạt hiệu suất tốt hơn. Yêu cầu CAP theorem có thể được nhắc lại: trong môi trường phân tán, không thể đảm bảo đồng thời tính nhất quán tuyệt đối và sẵn sàng khi xảy ra phân hoạch mạng : bạn phải xác định ưu tiên cái nào cho hệ thống của mình.

- **Chi phí và kinh nghiệm đội ngũ:** Đây là yếu tố thực tế nhưng rất quan trọng. Sử dụng giải pháp mã nguồn mở miễn phí hay dịch vụ trả phí? Đội ngũ của bạn đã quen thuộc với SQL hay NoSQL nào chưa? Ví dụ, nếu ngân sách hạn chế, bạn có thể chọn PostgreSQL (miễn phí) thay vì Oracle (đắt đỏ) cho RDBMS. Nếu team đã có kinh nghiệm với MongoDB, thì dùng nó có thể nhanh chóng hơn là học một database mới. Trong phỏng vấn, khi đề cập đến công nghệ, nếu có thể, hãy nêu lý do lựa chọn cả về kỹ thuật lẫn bối cảnh team/chi phí: điều này cho thấy bạn biết cân nhắc thực tế chứ không chỉ lý thuyết.

- **Các yếu tố khác:** bảo mật dữ liệu, yêu cầu tuân thủ (compliance) có thể ảnh hưởng chọn lựa (ví dụ có DB cung cấp mã hóa tích hợp, audit log tốt hơn); độ trễ mạng và vị trí triển khai (on-premise vs cloud) cũng có thể làm bạn nghiêng về giải pháp này hay khác. Với hệ thống rất lớn hiện đại, kiến trúc microservices thường được áp dụng: mỗi service có thể dùng một loại database phù hợp riêng. Vì vậy trong một hệ thống, việc dùng nhiều loại data store kết hợp (polyglot persistence) là bình thường. Bạn nên thể hiện sự linh hoạt: không cố ép mọi thứ vào một database duy nhất nếu nhu cầu khác nhau.

Tóm lại, khi thảo luận về lưu trữ dữ liệu trong thiết kế, đừng chỉ nêu tên một công nghệ. Hãy chứng minh bạn hiểu vấn đề cốt lõi cần giải quyết và giải pháp đó đáp ứng ra sao. Một câu trả lời cân nhắc cả đặc tính dữ liệu, khối lượng, mô hình truy cập, và trade-off chi phí/độ phức tạp chắc chắn sẽ gây ấn tượng với người phỏng vấn.

Dữ liệu phân tán và tính nhất quán (Consistency)

Khi hệ thống của bạn phân tán dữ liệu trên nhiều server (ví dụ có replica hoặc shard ở nhiều nơi), một thách thức nảy sinh: đảm bảo tính nhất quán giữa các bản sao dữ liệu. Chương trước (Chương 3) có thể bạn đã tìm hiểu về các mô hình nhất quán, ở đây chúng ta liên hệ vào bối cảnh lưu trữ dữ liệu.

Có hai mô hình thường được nhắc đến: Strong Consistency (nhất quán mạnh) và Eventual Consistency (nhất quán cuối cùng). Hiểu đơn giản:

- Nhất quán mạnh: Sau khi dữ liệu được cập nhật, mọi truy cập đọc tiếp theo đều thấy dữ liệu mới. Hệ thống như vậy đảm bảo không có đọc nào thấy dữ liệu cũ sau khi ghi thành công. Để làm được điều này trong môi trường phân tán, thường phải đánh đổi: ví dụ mọi node (hoặc ít nhất đa số node) phải xác nhận cập nhật xong mới báo thành công. Điều này dẫn đến độ trễ cao hơn: người dùng có thể phải chờ thêm để dữ liệu đồng bộ. Đổi lại, hệ thống luôn trả kết quả chính xác mới nhất.

- Nhất quán cuối cùng (eventual): Sau khi cập nhật, các lần đọc sau không đảm bảo thấy ngay dữ liệu mới. Có thể một số node chưa kịp đồng bộ nên vẫn trả dữ liệu cũ trong một khoảng thời gian ngắn. Tuy nhiên, cuối cùng (sau một thời gian đồng bộ) thì tất cả các node sẽ có dữ liệu mới. Mô hình này chấp nhận trạng thái không đồng bộ tạm thời, bù lại hệ thống thường phản hồi nhanh hơn vì không cần chờ mọi node nhất quán trước khi trả lời yêu cầu. Nhiều hệ thống NoSQL phân tán (như Cassandra, Dynamo) mặc định eventual consistency: ví dụ khi bạn ghi dữ liệu, nó ghi vào 1-2 node và trả về ngay, các node khác sẽ đồng bộ sau. Nếu ngay lúc đó bạn đọc từ node chưa cập nhật, sẽ thấy dữ liệu cũ; nhưng chỉ một thời gian ngắn sau đọc lại sẽ thấy dữ liệu mới.

Một cách nôm na, strong consistency giống như tất cả các chi nhánh cửa hàng cập nhật giá ngay lập tức khi trụ sở thay đổi giá sản phẩm : khách hàng ở đâu kiểm tra cũng ra cùng một giá. Còn eventual consistency giống như trụ sở gửi email thông báo, các chi nhánh sẽ cập nhật giá dần trong ngày : tạm thời có chỗ giá cũ chỗ giá mới, nhưng cuối ngày thì mọi nơi đã đồng bộ giá mới. Sự đánh đổi này liên quan trực tiếp đến CAP Theorem: khi hệ thống phân tán gặp sự cố mạng (Partition), ta chỉ có thể chọn duy trì Consistency hoặc Availability cao, chứ không có cả hai tức thì.

Vậy nên chọn mô hình nào? Tùy thuộc vào yêu cầu ứng dụng. Nếu dữ liệu nhạy cảm tài chính (số dư tài khoản, đơn hàng thanh toán) hoặc dữ liệu cần tính chính xác tuyệt đối tại mọi thời điểm, bạn nên thiết kế để đạt gần strong consistency. Điều này có thể nghĩa là chấp nhận giảm một chút availability: ví dụ nếu một node chính bị lỗi, hệ thống tạm khóa ghi để đảm bảo không mất nhất quán. Ngược lại, nếu là dữ liệu mạng xã hội, tin tức, nội dung hiển thị cho người dùng: thường có thể chấp nhận eventual consistency để đạt hiệu năng và độ sẵn sàng cao hơn. Người dùng có thể không phiền nếu bài đăng mới của họ mất vài giây mới xuất hiện trên news feed, miễn là hệ thống luôn hoạt động mượt mà.

Trong phỏng vấn, khi nói về dữ liệu phân tán, hãy thể hiện rằng bạn hiểu những khái niệm trên. Một điểm cộng là đề cập đến việc hệ thống của bạn sẽ đảm bảo nhất quán dữ liệu ra sao: ví dụ, "Tôi dùng master-slave replication cho database, mọi ghi đi vào master để đảm bảo thứ tự và nhất quán, còn slave sẽ sync sau để phục vụ đọc. Điều này cho phép đọc mở rộng nhưng vẫn giữ được tính nhất quán khá cao (gần như strong cho giao dịch qua master)". Hoặc bạn có thể nói: "Đối với phần dữ liệu phiên người dùng, tôi chọn giải pháp eventual consistency: dùng cache phân tán, chấp nhận thông tin có thể chưa kịp đồng bộ giữa các server ngay lập tức, bù lại hệ thống login không bị chậm." Cách bạn cân nhắc và giải thích sẽ cho thấy bạn nắm chắc trade-offs trong thiết kế hệ thống phân tán.

Gợi ý trình bày phần thiết kế Database khi phỏng vấn

Cuối cùng, khi bước vào phỏng vấn System Design, làm thế nào để bạn trình bày mạch lạc về phần thiết kế cơ sở dữ liệu của hệ thống? Dưới đây là một số gợi ý giúp phần này của bạn gây ấn tượng tốt:

- **Bắt đầu từ yêu cầu dữ liệu:** Trước khi chọn ngay SQL hay NoSQL, hãy nhắc lại (tóm tắt) những yêu cầu liên quan đến dữ liệu của bài toán. Ví dụ: "Hệ thống quản lý ngân hàng này cần lưu giao dịch tài khoản, đòi hỏi tính chính xác và ACID cao, đồng thời lượng giao dịch rất lớn mỗi ngày." Việc này dẫn dắt tự nhiên đến lựa chọn công nghệ của bạn và cho thấy bạn thiết kế dựa trên yêu cầu chứ không thuộc lòng công thức.

- **Giải thích lựa chọn loại CSDL:** Sau khi xác định nhu cầu, hãy nêu rõ bạn chọn loại database nào và vì sao. So sánh nhanh với giải pháp khác để thể hiện bạn có cân nhắc. Ví dụ: "Tôi chọn SQL (quan hệ) cho phần lưu trữ user và giao dịch, vì dữ liệu có cấu trúc rõ, cần JOIN và transaction. NoSQL ở đây không đảm bảo tính nhất quán cao bằng, nên không phù hợp bằng SQL. Tuy nhiên, tôi có thể dùng thêm một Redis (Key-Value) để lưu cache phiên người dùng cho nhanh." Một câu trả lời như vậy vừa chỉ ra lựa chọn chính, vừa thể hiện bạn biết kết hợp nhiều thành phần.

- **Nhắc đến mô hình dữ liệu cụ thể:** Không cần thiết kế chi tiết mọi bảng, nhưng bạn nên nói qua các thực thể chính và quan hệ của chúng. Ví dụ: "Tôi sẽ có bảng User, bảng Post, mỗi Post có user_id làm foreign key trỏ sang User: thiết kế này cho phép truy vấn thông tin bài viết kèm thông tin người đăng dễ dàng." Hoặc nếu NoSQL: "Tôi sẽ lưu mỗi user như một document bao gồm cả profile và danh sách bạn bè, vì dữ liệu này thường được truy xuất cùng nhau." Cách trình bày này cho

thấy tư duy mô hình hóa dữ liệu của bạn.

- **Kế hoạch mở rộng và phân mảnh:** Nhà tuyển dụng muốn thấy bạn nghĩ xa hơn bản thiết kế ban đầu. Hãy nói về scaling: "Ban đầu một server DB có thể đủ, nhưng nếu số người dùng tăng lên 10 lần, tôi sẽ triển khai replication tạo nhiều read-replicas để chia tải đọc. Nếu tiếp tục tăng nữa, tôi sẵn sàng shard dữ liệu theo userID để phân tán sang nhiều cụm máy chủ. Sharding sẽ được triển khai khi số lượng user đạt mốc X, giúp hệ thống vẫn chịu tải tốt." Việc đưa ra các con số ước lượng (dù sơ bộ) về khi nào cần scale cũng gây ấn tượng rằng bạn có kinh nghiệm thực tế.

- **Nhắc đến chỉ mục và tối ưu truy vấn:** Một điểm mà nhiều ứng viên bỏ qua là nói về tối ưu hiệu năng truy vấn. Bạn có thể thêm: "Với bảng thông tin sản phẩm ~10 triệu dòng, tôi sẽ đánh index trên cột product_id (để tìm kiếm nhanh) và category (để lọc). Điều này đảm bảo truy vấn trang sản phẩm và danh mục nhanh chóng, không bị quét toàn bộ bảng. Ngoài ra, tôi sẽ sử dụng cache tại ứng dụng cho những truy vấn phổ biến." Chi tiết về indexing, cache cho thấy bạn hiểu sâu về hệ thống, không chỉ vẽ kiến trúc tổng quát mà quên mất hiệu năng.

- **Thảo luận về consistency và trade-off nếu có:** Nếu hệ thống của bạn phân tán, đừng ngại đề cập tới nhất quán dữ liệu. Ví dụ: "Tôi chọn mô hình eventual consistency cho module chat: tin nhắn có thể mất vài giây mới sync hết các server, nhưng bù lại hệ thống luôn sẵn sàng ngay cả khi một node gặp sự cố. Người dùng có thể chấp nhận độ trễ rất nhỏ này. Còn đối với module thanh toán, tôi sẽ dùng một central database để đảm bảo strong consistency: không bao giờ có trường hợp hiển thị số dư sai." Cách phân tích này chứng tỏ bạn biết đánh đổi và ưu tiên phù hợp cho từng phần của hệ thống.

- **Tự tin nhưng linh hoạt:** Cuối cùng, hãy tự tin với thiết kế của mình, nhưng cũng sẵn sàng trả lời câu "Nếu...thì sao?". Ví dụ nếu người phỏng vấn hỏi "Nếu lượng dữ liệu tăng đột biến gấp 100, làm sao DB chịu nổi?", bạn có thể nói về sharding nhiều hơn, hoặc dùng dịch vụ phân tán như Amazon DynamoDB, hoặc tách một số dữ liệu ít dùng sang kho lưu trữ khác. Thể hiện rằng bạn đã nghĩ đến phương án dự phòng cho kịch bản xấu sẽ gây thiện cảm lớn.

Mở rộng hệ thống và tối ưu hiệu suất

Trong chương này, chúng ta sẽ tìm hiểu về ba kỹ thuật quan trọng giúp hệ thống mở rộng (scale) và tối ưu hiệu suất. Đó là cân bằng tải (**Load Balancer**), bộ nhớ đệm (**Caching**) và hàng đợi thông điệp (**Message Queue**). Đây đều là những thành phần quen thuộc trong thiết kế hệ thống hiện đại, giúp ứng dụng phục vụ được nhiều người dùng hơn, nhanh hơn và ổn định hơn. Văn phong của chương hướng dẫn này sẽ gần gũi như một cuộc trò chuyện, kèm theo những ví dụ minh họa đời thực để bạn dễ hình dung. Chúng ta hãy cùng bắt đầu nhé!

Cân bằng tải (Load Balancer)

Load Balancer là gì và vai trò của nó?

Load Balancer (cân bằng tải) là một thành phần đứng trước một cụm máy chủ nhằm phân phối lưu lượng (các request từ người dùng) đến các máy chủ phía sau sao cho không máy chủ nào bị quá tải. Bạn có thể hình dung Load Balancer giống như nhân viên điều phối ở quầy giao dịch ngân hàng: khi có khách hàng đến (tương ứng với request), nhân viên này sẽ hướng dẫn họ tới một quầy giao dịch còn trống (tương ứng với một máy chủ) để được phục vụ. Nhờ có sự điều phối này, hệ thống đảm bảo rằng không có một quầy nào phải phục vụ quá nhiều khách cùng lúc, và trải nghiệm của khách hàng (người dùng) sẽ mượt mà, nhanh chóng hơn.

Vai trò chính của Load Balancer gồm:

- **Phân phối đều tải:** Đảm bảo mỗi máy chủ xử lý lượng yêu cầu vừa phải, tránh tình trạng một máy gánh hết còn các máy khác rảnh rỗi.

- **Tăng khả năng mở rộng (scalability):** Khi lượng người dùng tăng cao, ta có thể bổ sung thêm máy chủ phía sau Load Balancer. Load Balancer sẽ tiếp tục phân phối request, giúp hệ thống mở rộng dễ dàng.

- **Tăng độ sẵn sàng (availability):** Nếu một máy chủ gặp sự cố, Load Balancer có thể ngưng gửi request đến máy đó và chuyển sang các máy khác. Người dùng sẽ ít bị ảnh hưởng khi một phần hệ thống bị lỗi.

- **Đơn giản hóa kiến trúc:** Người dùng chỉ cần kết nối đến Load Balancer (thường qua một URL hoặc địa chỉ IP duy nhất), không cần biết đằng sau có bao nhiêu

máy chủ. Toàn bộ cụm máy chủ phía sau hoạt động như một máy chủ ảo duy nhất đối với người dùng.

Tóm lại, Load Balancer giúp hệ thống chịu tải tốt hơn và ổn định hơn. Đây là thành phần gần như không thể thiếu khi thiết kế hệ thống lớn hoặc các ứng dụng web phục vụ lượng người dùng đông đảo.

Cách hoạt động và các thuật toán phân phối

Một Load Balancer hoạt động như điểm trung gian giữa khách hàng và các máy chủ thực sự. Khi có request gửi tới, Load Balancer sẽ quyết định chuyển request đó tới máy chủ nào ở tầng sau. Quyết định này được thực hiện dựa trên các thuật toán phân phối mà ta có thể cấu hình cho Load Balancer. Dưới đây là một số thuật toán phổ biến:

- Round Robin (Vòng tròn luân phiên): Mỗi yêu cầu được phân lượt đến từng máy chủ theo thứ tự luân phiên. Ví dụ có 3 server A, B, C thì request 1 -> A, request 2 -> B, request 3 -> C, rồi request 4 lại quay vòng về A. Cách này đảm bảo phân phối tương đối đều, đơn giản như chia bài theo vòng.

- Least Connections (Ít kết nối nhất): Load Balancer sẽ theo dõi số lượng kết nối hoặc request đang xử lý trên mỗi máy chủ, và gửi request mới đến máy chủ nào đang bận ít nhất. Thuật toán này hiệu quả khi các request có thời gian xử lý không đồng đều. Ví dụ nếu server A đang xử lý 100 kết nối còn B chỉ 50, thì yêu cầu mới sẽ ưu tiên vào B.

- IP Hash: Thuật toán này dùng địa chỉ IP của client (người dùng) để tính toán và gán cố định mỗi IP client vào một máy chủ cụ thể. Nghĩa là cùng một người dùng (cùng IP) thì các request của họ luôn đi vào cùng một server. Cách này hữu ích để duy trì tính phiên (session stickiness) khi ta muốn người dùng tương tác với cùng một máy chủ (ví dụ phiên đăng nhập lưu ở memory của server thay vì database chung).

- Weighted Round Robin (Luân phiên theo trọng số): Biến thể của Round Robin, cho phép gán trọng số cho mỗi máy chủ. Máy chủ mạnh hơn (nhiều CPU/RAM hơn) có thể được nhận nhiều request hơn. Ví dụ server A (mạnh) trọng số 5, server B (yếu hơn) trọng số 1, thì Load Balancer sẽ gửi khoảng 5 request đến A rồi 1 request đến B luân phiên.

- Thuật toán khác: Tùy hệ thống, còn nhiều thuật toán nâng cao như Least Response Time (máy nào phản hồi nhanh nhất sẽ được ưu tiên), Geo-location (phân phối theo địa lý, gần máy chủ nào thì vào máy đó), v.v. Nhưng đối với phỏng vấn thiết kế hệ thống ở mức cơ bản, bạn chỉ cần hiểu các thuật toán chính như trên (Round Robin, Least Connections, IP Hash) là đủ.

Ví dụ minh họa: Hãy tưởng tượng bạn quản lý một website bán hàng trực tuyến. Ban đầu, bạn chỉ có một máy chủ xử lý toàn bộ request. Khi số lượng người truy cập tăng lên, máy chủ này bắt đầu chậm lại. Bạn quyết định thêm hai máy chủ nữa (tổng cộng 3 máy chủ web có cùng chức năng). Lúc này, làm sao để người dùng truy cập mà sử dụng được cả 3 máy chủ? Giải pháp là đặt một Load Balancer ở phía trước. Mỗi khi người dùng gửi request (ví dụ mở trang sản phẩm), Load Balancer sẽ chia các request này đều ra 3 máy theo thuật toán Round Robin. Nhờ đó, không có máy nào bị quá tải, và người dùng được phục vụ nhanh hơn. Nếu một máy gặp sự cố, Load Balancer sẽ tự động chuyển hướng request sang 2 máy còn lại, nhờ đó website vẫn hoạt động liên tục.

Load Balancer tầng 4 vs tầng 7 (Layer 4 vs Layer 7)

Khi tìm hiểu sâu hơn, bạn sẽ nghe đến khái niệm Load Balancer tầng 4 và Load Balancer tầng 7. Đây là cách phân loại dựa trên mô hình OSI: một mô hình mạng chia các tầng giao tiếp. Hiểu đơn giản:

- Load Balancer tầng 4 (Layer 4): hoạt động ở tầng Giao vận (Transport Layer), dựa trên thông tin địa chỉ mạng như IP và port của gói tin TCP/UDP. Load Balancer tầng 4 không hiểu nội dung bên trong của request; nó chỉ biết có một gói tin đến IP này, port này và chuyển tiếp đến một máy chủ dựa theo thuật toán đã chọn. Ưu điểm của LB tầng 4 là nhanh hơn vì nó xử lý ít thông tin (chỉ ở mức packet IP/port), phù hợp cho phân tải đơn giản. Tuy nhiên, nó không thể quyết định dựa trên nội dung; ví dụ không biết URL hay HTTP header là gì.

- Load Balancer tầng 7 (Layer 7): hoạt động ở tầng Ứng dụng (Application Layer), tức là hiểu được các giao thức ứng dụng như HTTP/HTTPS. LB tầng 7 đọc được nội dung request (URL, đường dẫn, cookie, header, thậm chí nội dung body). Do đó, nó có thể đưa ra quyết định thông minh hơn. Ví dụ: LB thấy request vào đường dẫn /video/720p thì chuyển tới cụm máy chủ chuyên phục vụ video độ phân giải thấp, còn /video/1080p sang máy chủ phục vụ độ phân giải cao hơn. Hoặc LB tầng 7 có thể terminate SSL (giải mã HTTPS) rồi mới chuyển request HTTP vào bên trong. Nhờ hiểu tầng ứng dụng, LB tầng 7 cho phép các tính năng

như content-based routing (phân tải dựa trên nội dung) và hỗ trợ sticky session bằng cookie. Nhược điểm là do phải đọc/ghi và xử lý nhiều thông tin hơn nên LB tầng 7 thường chậm hơn LB tầng 4 một chút và tiêu tốn tài nguyên hơn.

Nên dùng LB tầng 4 hay tầng 7? Điều này tùy thuộc vào nhu cầu của hệ thống:

- Nếu bạn chỉ cần phân phối lưu lượng đơn thuần và muốn tốc độ tối đa, ít tốn tài nguyên, LB tầng 4 là đủ. Ví dụ cho các ứng dụng không phụ thuộc nội dung gói tin, hoặc các giao thức không phải HTTP (như game server dùng UDP, kết nối VPN, ...).

- Nếu ứng dụng web cần quyết định phức tạp dựa trên URL, cookie, hoặc cần chia tách dịch vụ (như tách traffic tĩnh/động, mobile/PC), bạn sẽ cần LB tầng 7. Đa số các website, dịch vụ web lớn (thương mại điện tử, streaming video, API) đều dùng LB tầng 7 vì chúng cho phép linh hoạt cao trong điều phối request.

Khi nào cần sử dụng Load Balancer?

Nếu hệ thống của bạn chỉ có một máy chủ và lượng người dùng nhỏ, có thể chưa cần đến Load Balancer. Nhưng khi bạn muốn hệ thống có thể mở rộng hoặc đạt dự phòng cao, Load Balancer trở thành cần thiết. Một số tình huống điển hình nên dùng Load Balancer:

- Tăng lượng người dùng: Ứng dụng web bắt đầu có hàng ngàn, hàng triệu lượt truy cập mỗi ngày. Một máy chủ đơn lẻ không đáp ứng nổi, bạn cần nhiều máy chủ chạy song song -> cần Load Balancer để chia tải.

- High Availability (độ sẵn sàng cao): Bạn muốn hệ thống luôn hoạt động ngay cả khi một máy chủ hỏng. Với nhiều máy chủ và Load Balancer, nếu một máy "chết" thì LB chuyển lưu lượng sang máy khác, người dùng ít bị gián đoạn.

- Tách biệt chức năng: Đôi khi bạn có nhiều loại máy chủ cho các mục đích khác nhau (ví dụ: server phục vụ ảnh, server phục vụ API). Một Load Balancer tầng 7 có thể dựa theo đường dẫn để gửi đúng loại request đến đúng nhóm server tương ứng.

- Hiệu năng và bảo mật: Một số Load Balancer tích hợp sẵn chức năng như nén nội dung, lưu cache tạm thời, hoặc tường lửa ứng dụng (WAF). Đặt LB phía trước có

thể giảm tải công việc cho các máy chủ ứng dụng phía sau.

Tóm lại, bất cứ hệ thống nào cần mở rộng quy mô hay nâng cao tính ổn định đều sẽ sớm cần tới Load Balancer. Đây là lý do tại sao trong phỏng vấn thiết kế hệ thống, gần như chắc chắn người phỏng vấn mong đợi bạn đề cập đến Load Balancer khi nói về kiến trúc nhiều máy chủ.

Gợi ý khi phỏng vấn: Nếu được hỏi về Load Balancer, bạn có thể trả lời như sau: "Load Balancer giống như một bộ phận phân làn giao thông cho các request. Nó giúp phân phối đều request đến nhiều server phía sau, tránh cho một server bị quá tải. Em có thể dùng thuật toán Round Robin để chia đều hoặc Least Connections để gửi request tới server nào đang rảnh nhất. Với web app, em sẽ dùng Load Balancer tầng 7 để nó đọc được URL và quyết định thông minh hơn, ví dụ tách nội dung tĩnh/động. Nhờ có Load Balancer, hệ thống của em có thể mở rộng dễ dàng bằng cách thêm server, và cũng đảm bảo nếu một server hỏng thì lưu lượng sẽ được chuyển sang server khác, không ảnh hưởng người dùng." Câu trả lời như vậy vừa có ví dụ, vừa cho thấy bạn hiểu vai trò và cách hoạt động của Load Balancer.

Bộ nhớ đệm (Caching)

Cache là gì và có lợi ích gì?

Cache (bộ nhớ đệm) là một cơ chế lưu trữ tạm thời dữ liệu phục vụ các yêu cầu truy xuất nhanh hơn. Thay vì mỗi lần cần dữ liệu đều phải lấy từ nguồn gốc (ví dụ: đọc từ cơ sở dữ liệu hoặc gọi một dịch vụ bên ngoài chậm chạp), ta có thể lưu lại bản sao của dữ liệu ở một nơi truy xuất nhanh (như trong RAM, trên ổ đĩa gần người dùng, v.v.). Lần sau nếu cần lại dữ liệu đó, hệ thống sẽ lấy ngay từ cache, sẽ nhanh hơn rất nhiều so với lấy từ nguồn gốc.

Hãy liên hệ với đời sống: Bạn có nhớ lần đầu tiên đi đến một khu phố mới, bạn phải mở bản đồ hoặc hỏi đường mất thời gian. Nhưng từ lần thứ hai trở đi, bạn đã nhớ đường (bạn đã "cache" đường đi trong trí nhớ), nên đi nhanh hơn hẳn. Tương tự, máy tính khi đã có dữ liệu trong cache (nhớ rồi) thì phục vụ người dùng nhanh hơn là mỗi lần đều đi tra cứu từ đầu.

Lợi ích của caching:

- Tăng tốc độ phản hồi: Dữ liệu từ cache (thường ở RAM hoặc ngay gần người dùng) nhanh hơn dữ liệu từ database gốc hay từ server ở xa. Ví dụ, cache trong

RAM có thể nhanh gấp hàng trăm lần đọc từ ổ cứng, và nhanh hơn nhiều lần so với việc gọi qua mạng.

- Giảm tải cho hệ thống gốc: Khi nhiều yêu cầu được phục vụ từ cache, số lượng truy vấn đến database hoặc dịch vụ gốc sẽ giảm. Database sẽ đỡ bị quá tải hơn, cho phép phục vụ nhiều người dùng hơn.

- Tiết kiệm tài nguyên và chi phí: Việc giảm tải cho máy chủ gốc có thể giúp bạn tiết kiệm chi phí mở rộng hạ tầng. Thay vì phải mua thêm một database server mạnh, bạn có thể chỉ cần thêm một lớp cache (ví dụ dùng Redis, Memcached) để gánh bớt.

- Trải nghiệm người dùng tốt hơn: Thời gian phản hồi nhanh giúp người dùng hài lòng, web/app mượt mà. Điều này đặc biệt quan trọng cho các ứng dụng thời gian thực hoặc có lượng người dùng lớn.

Tuy nhiên, cache không phải "miễn phí": bạn sẽ tốn thêm bộ nhớ để lưu cache, và phức tạp hơn trong việc đảm bảo dữ liệu hợp lệ, không bị cũ (vì dữ liệu gốc thay đổi nhưng cache có thể vẫn lưu bản cũ, dẫn tới "cache stale"). Phần sau chúng ta sẽ nói về chiến lược để cập nhật cache.

Các vị trí có thể đặt cache

Cache có thể xuất hiện ở nhiều tầng khác nhau trong một hệ thống. Mục tiêu chung là đưa dữ liệu tới gần người dùng nhất có thể hoặc trong môi trường truy cập nhanh nhất. Dưới đây là các vị trí phổ biến bạn có thể triển khai cache, kèm ví dụ:

- Cache phía Client (người dùng): Đây là cache ngay trên thiết bị hoặc trình duyệt của người dùng. Ví dụ: trình duyệt web có cơ chế cache các file tĩnh (hình ảnh, CSS, JavaScript). Nhờ đó khi bạn mở một trang web lần thứ hai, nhiều thành phần không cần tải lại từ server vì đã có sẵn trong cache của trình duyệt. Một ví dụ khác là các ứng dụng di động thường lưu dữ liệu (như danh sách bài viết mới nhất) trên bộ nhớ điện thoại, giúp người dùng có thể xem lại nhanh chóng và giảm băng thông mạng.

- CDN Cache (máy chủ biên (Edge server)/đám mây (Cloud)): CDN (Content Delivery Network) là mạng lưới các máy chủ đặt ở nhiều nơi trên thế giới. CDN giữ bản sao cache của nội dung tĩnh (hình ảnh, video, file CSS/JS, nội dung không đổi

thường xuyên) và khi người dùng gần khu vực nào truy cập, CDN sẽ cung cấp nội dung từ máy chủ gần họ nhất. Ví dụ: một người dùng ở Việt Nam truy cập trang web đặt máy chủ gốc ở Mỹ, nhưng hình ảnh và video có thể được phân phát từ server CDN tại Việt Nam : nhanh hơn rất nhiều so với việc tải từ Mỹ. CDN là một dạng cache tầng mạng, rất hữu ích để tăng tốc độ truy cập toàn cầu.

- Cache tại Reverse Proxy (máy chủ proxy ngược): Một reverse proxy như Nginx, Varnish có thể được đặt phía trước máy chủ ứng dụng. Proxy này có thể lưu cache kết quả của các request phổ biến. Ví dụ: trang chủ của một tờ báo điện tử có thể được cache trong 60 giây tại lớp proxy. Khi hàng nghìn người dùng cùng truy cập trang chủ trong khoảng thời gian đó, proxy sẽ phục vụ từ cache thay vì bắt ứng dụng xử lý lại từng lần. Kết quả: giảm tải đáng kể cho máy chủ ứng dụng và tăng tốc độ phản hồi.

- Cache trong ứng dụng (Application Cache): Ở tầng ứng dụng, lập trình viên có thể chủ động lưu lại những dữ liệu thường dùng trong bộ nhớ. Ví dụ: một ứng dụng web có thể cache kết quả của một truy vấn database tốn kém (như tính toán thống kê, danh sách sản phẩm bán chạy) trong biến toàn cục hoặc trong một kho cache nhanh (Redis, Memcached). Lần sau nếu có request tương tự, ứng dụng kiểm tra cache và trả về ngay thay vì query database. Loại cache này thường triển khai bằng cách tích hợp thư viện hoặc dịch vụ cache vào code của ứng dụng.

- Cache tại cơ sở dữ liệu: Nhiều hệ quản trị cơ sở dữ liệu có sẵn cơ chế cache bên trong. Ví dụ MySQL có Query Cache (cache kết quả truy vấn), hoặc một số cơ chế như buffer pool trong database lưu dữ liệu hay dùng trong RAM. Ngoài ra, một cách hiểu khác về cache ở tầng này là sử dụng các hệ thống như Redis, Memcached (thường gọi chung là database cache hoặc caching layer) như một tầng dữ liệu nhanh. Ứng dụng có thể xem nó như một database đọc nhanh, lưu các cặp key-value mà truy xuất rất nhanh. Tuy nhiên, vì Redis/Memcached không phải database chính, ta vẫn coi nó là cache hỗ trợ cho database chính phía sau (thường là SQL hoặc NoSQL database chậm hơn).

Tất cả các tầng cache trên có thể được dùng kết hợp trong một hệ thống lớn. Ví dụ: một trang web có thể sử dụng CDN cho nội dung tĩnh, dùng cache ứng dụng hoặc Redis cho dữ liệu động, đồng thời trình duyệt người dùng cũng cache để giảm truy cập mạng. Hiểu được mỗi tầng cache giúp bạn thiết kế hệ thống tối ưu hơn và cũng có thể giải thích trong phỏng vấn khi nói về tối ưu hiệu suất.

Các chiến lược cập nhật cache (Cache update strategies)

Một thách thức quan trọng khi dùng cache là làm sao để dữ liệu trong cache luôn phù hợp (hợp lệ) so với dữ liệu gốc. Nếu cache quá cũ (stale data), người dùng có thể thấy thông tin sai lệch. Có một câu nói vui trong khoa học máy tính: "Có hai việc khó nhất: đặt tên biến, xử lý cache invalidation, và off-by-one error." Ý nói việc invalidation cache (làm mới/ghi đè dữ liệu cache) là rất khó. Dưới đây, chúng ta sẽ thảo luận một số chiến lược phổ biến để cập nhật dữ liệu cache khi dữ liệu gốc thay đổi:

- **Cache-Aside (Lazy Loading):** Đây là chiến lược phổ biến và dễ triển khai nhất. Ứng dụng sẽ đọc dữ liệu "bên cạnh" cache. Cụ thể:
    - Khi cần dữ liệu X, ứng dụng sẽ kiểm tra trong cache xem có X chưa.
    - Nếu cache có X (cache hit), lấy ra và dùng ngay.
    - Nếu cache không có (cache miss), ứng dụng sẽ truy vấn dữ liệu gốc (ví dụ database) để lấy X. Sau khi lấy được X, lưu bổ sung X vào cache để lần sau dùng cho nhanh, rồi trả kết quả về.
    - Khi dữ liệu X được cập nhật (write), ứng dụng ghi trực tiếp vào database và xóa (invalidate) cache của X (hoặc cập nhật cache nếu muốn).

  Chiến lược cache-aside này còn gọi là lazy loading vì chỉ khi nào cần mới nạp dữ liệu vào cache. Ưu điểm: đơn giản, ứng dụng kiểm soát được khi nào đọc/ghi cache. Nhược: lần đầu tiên cache miss sẽ vẫn chậm do phải xuống database, và lập trình viên phải cẩn thận việc xóa cache khi ghi để tránh dữ liệu cũ.

- **Write-Through:** Chiến lược này xử lý ở lúc ghi dữ liệu. Mỗi khi có thay đổi (ghi mới hoặc cập nhật dữ liệu), thay vì chỉ ghi vào database, ứng dụng hoặc hệ thống sẽ đồng thời ghi vào cache. Cụ thể:
    - Ứng dụng ghi dữ liệu X -> hệ thống ghi ngay X vào database và cập nhật cả cache X.
    - Lần sau đọc X, chắc chắn cache có dữ liệu mới nhất (vì đã được cập nhật tức thời khi ghi).

Write-through đảm bảo tính nhất quán cao giữa cache và nguồn dữ liệu chính, vì bất cứ thay đổi nào cũng lập tức có trên cache. Nhược điểm là hiệu năng ghi chậm hơn: mỗi lần ghi phải thực hiện hai thao tác (DB và cache). Nếu bất kỳ thao tác nào thất bại, thường phải rollback để giữ dữ liệu đồng bộ, điều này làm phức tạp hệ thống. Write-through thường phù hợp nếu ứng dụng đọc nhiều, ghi ít, và yêu cầu dữ liệu đọc ra phải thật mới. Ví dụ: caching cấu hình hệ thống hoặc danh sách tham số mà thỉnh thoảng mới đổi, còn khi đọc thì muốn chắc luôn đúng.

- **Write-Back (Write-Behind):** Đây là chiến lược ngược lại một phần với write-through. Ở write-back, ứng dụng khi ghi sẽ chỉ ghi vào cache trước, coi như cache là vùng đệm tạm cho viết. Dữ liệu trong cache sẽ được đánh dấu là "bẩn" (dirty). Hệ thống sẽ ghi ngược trở lại database sau theo lịch trình hoặc khi rảnh:
    - Ứng dụng ghi X -> ghi vào cache (nhanh) và coi như đã xong đối với người dùng.
    - Hệ thống nền (background process) định kỳ lấy các mục "dirty" trong cache để ghi xuống database (có thể gộp nhiều ghi thành một batch lớn).

  Lợi ích của write-back là ghi rất nhanh (vì ghi vào cache thường là RAM, và không chờ DB tại thời điểm tương tác người dùng). Thích hợp cho các hệ thống đòi hỏi tốc độ ghi cao hoặc ghi rất thường xuyên. Ngoài ra, nếu trong thời gian ngắn có nhiều lần cập nhật cùng một dữ liệu, hệ thống có thể chỉ cần ghi xuống database một lần cuối (vì các lần trước đã bị ghi đè trong cache), giảm tải đáng kể cho database. Tuy nhiên, nhược điểm lớn là nguy cơ mất dữ liệu: nếu cache (RAM) bị hỏng hoặc server cache chết trước khi kịp ghi ra database, các thay đổi chưa kịp lưu xuống sẽ mất. Vì vậy, write-back cần các cơ chế bảo đảm tin cậy như log giao dịch, hoặc thường chỉ dùng khi chấp nhận mất mát nhỏ hoặc có giải pháp khác đi kèm. Thực tế, write-back hay dùng trong các hệ thống như bộ nhớ CPU cache, hoặc các hệ thống cần hiệu năng ghi cao và có cơ chế bổ sung đảm bảo dữ liệu (ví dụ ghi ra đĩa định kỳ, hoặc cluster cache có sao lưu).

- **Refresh-Ahead (Prefetching):** Chiến lược này liên quan đến việc làm mới cache trước khi nó hết hạn. Bình thường, cache hay được thiết lập một khoảng thời gian sống (TTL). Ví dụ cache một kết quả trong 10 phút. Với refresh-ahead, hệ thống sẽ để ý khi sắp hết 10 phút (ví dụ còn 1 phút nữa hết hạn), nó sẽ chủ động đi lấy dữ liệu mới và cập nhật cache trước. Như vậy, lý tưởng là người dùng sẽ luôn gặp cache "nóng" và ít khi bị gián đoạn chờ load lại. Cách này phù hợp cho các dữ liệu

> mà ta dự đoán được nhu cầu. Ví dụ: một trang báo luôn có lượng truy cập cao vào 7h sáng, ta có thể refresh cache trang chủ vài phút trước 7h để đảm bảo đến giờ cao điểm, cache đã sẵn sàng dữ liệu mới nhất. Tuy nhiên, refresh-ahead có thể dẫn tới lãng phí nếu ta dự đoán sai (cập nhật cache cho dữ liệu mà không có ai hỏi đến).

Ngoài ra, còn một số thuật ngữ khác như write-around (một biến thể: khi ghi thì bỏ qua cache, chỉ ghi DB, để tránh làm "bẩn" cache với dữ liệu có thể không được đọc ngay; dữ liệu sẽ vào cache theo cache-aside khi có đọc) v.v. Nhưng bốn chiến lược chính đã nêu trên (cache-aside, write-through, write-back, refresh-ahead) là đủ để bạn hiểu cách quản lý cache trong hệ thống. Thường trong thực tế, người ta kết hợp các chiến lược này tùy trường hợp. Ví dụ: cache-aside cho đọc, kết hợp write-through cho ghi đối với một số dữ liệu quan trọng.

Ví dụ minh họa việc dùng cache: Giả sử bạn xây dựng một ứng dụng xem tỷ giá ngoại tệ. Tỷ giá lấy từ ngân hàng chỉ thay đổi 2 lần mỗi ngày. Nếu mỗi lần người dùng mở app mà bạn đều gọi API tới ngân hàng để lấy tỷ giá thì quá chậm và tốn tài nguyên. Thay vào đó, bạn có thể dùng cache-aside theo từng ngày: lần đầu tiên trong ngày khi có người dùng hỏi tỷ giá USD, app sẽ gọi ngân hàng lấy số liệu (ví dụ 23,500 VND/USD), sau đó lưu vào cache. Mọi người dùng khác trong cùng ngày hỏi tỷ giá USD sẽ được phục vụ ngay lập tức từ cache với con số 23,500, nhanh hơn hẳn so với gọi sang ngân hàng. Đến ngày hôm sau, tỷ giá thay đổi, lần truy cập đầu tiên sẽ fetch số mới (ví dụ 23,600) và cập nhật cache. Bằng cách này, hệ thống của bạn chịu được rất nhiều người dùng truy vấn mà ngân hàng cũng không bị quá tải bởi quá nhiều request.

Gợi ý khi phỏng vấn về Caching

Khi nói về tối ưu hiệu suất hệ thống trong phỏng vấn, gần như chắc chắn bạn nên nhắc đến caching. Nhà tuyển dụng muốn nghe rằng bạn biết sử dụng cache để giảm tải và tăng tốc. Bạn có thể trình bày như sau: "Để cải thiện hiệu suất, em sẽ áp dụng cache để lưu những dữ liệu được truy cập thường xuyên. Ví dụ, thay vì mỗi lần đều đọc database, em có thể lưu kết quả truy vấn vào Redis (một cache lưu trong RAM) trong vài phút. Nhờ đó các request sau sẽ lấy dữ liệu từ cache nhanh hơn rất nhiều. Em cũng chú ý cập nhật hoặc vô hiệu hóa cache khi dữ liệu nguồn thay đổi để đảm bảo tính nhất quán. Ví dụ dùng chiến lược cache-aside: khi ghi dữ liệu mới vào DB thì xóa cache cũ đi, còn khi đọc thì kiểm tra cache trước, cache miss mới truy vấn DB." Câu trả lời này cho thấy bạn hiểu cả lợi ích lẫn cách quản lý dữ liệu cache. Ngoài ra, nếu có thể, hãy đưa ví dụ cụ thể như cache kết quả truy vấn sản phẩm bán chạy, cache HTML của trang, hoặc nhắc đến các

tầng cache (client, CDN) nếu phù hợp với câu hỏi. Văn phong nên tự tin nhưng tránh quá sa đà thuật ngữ, hãy tập trung vào ý chính: cache = truy xuất nhanh hơn và giảm tải backend.

Hàng đợi thông điệp (Message Queue)

Khái niệm hàng đợi thông điệp & xử lý bất đồng bộ

Trong hệ thống nhỏ, khi người dùng gửi một yêu cầu, máy chủ xử lý và trả lời ngay (xử lý đồng bộ). Nhưng ở các hệ thống lớn, đôi khi ta không muốn hoặc không thể xử lý mọi việc ngay lập tức trong cùng một luồng. Hàng đợi thông điệp (Message Queue) xuất hiện như một giải pháp cho xử lý bất đồng bộ.

Hãy tưởng tượng một hàng đợi lấy số thứ tự ở quán ăn đông khách. Bạn đến quầy gọi món, thay vì người bán làm món của bạn ngay lập tức (và những người sau phải chờ bạn xong), họ đưa bạn một số thứ tự và bạn có thể ngồi đợi. Yêu cầu của bạn đã được đưa vào hàng đợi. Đầu bếp sẽ lần lượt đọc các số thứ tự (thông điệp) từ hàng đợi và chế biến món ăn. Khi món của bạn làm xong, số của bạn được gọi và bạn nhận đồ ăn. Ở đây, người gọi món đóng vai trò producer (sinh ra thông điệp yêu cầu món ăn), đầu bếp là consumer (tiêu thụ và xử lý yêu cầu), còn message queue chính là xấp phiếu gọi món theo thứ tự.

Tương tự trong kiến trúc phần mềm, Message Queue là một thành phần trung gian giữ các thông điệp (thường là nhiệm vụ cần làm hoặc dữ liệu cần xử lý). Thành phần gửi thông điệp (producer) sẽ đưa thông điệp vào queue rồi tiếp tục công việc của mình, không cần đợi kết quả ngay. Một (hoặc nhiều) thành phần khác (consumer) sẽ lấy thông điệp ra khỏi queue và xử lý nó theo tốc độ của riêng mình. Đây chính là mô hình xử lý bất đồng bộ: công việc được làm ở hậu trường, không chặn luồng chính đang tương tác với người dùng.

Khi nào nên dùng hàng đợi

Message Queue hữu ích trong rất nhiều trường hợp, đặc biệt là khi ta cần tách riêng công việc để xử lý sau hoặc điều hòa nhịp độ giữa các thành phần hệ thống. Một số tình huống thường gặp nên dùng hàng đợi:

- **Xử lý tác vụ nền (background jobs):** Khi người dùng thực hiện một hành động mà công việc để hoàn thành quá lâu hoặc không cần thiết phải xong ngay lập tức, ta có thể đưa công việc đó vào queue. Ví dụ: Người dùng đăng ký tài khoản -> ta gửi

email xác nhận. Việc gửi email có thể mất vài giây, thay vì bắt người dùng chờ trang đăng ký xoay vòng, ta trả lời ngay "Đăng ký thành công" và đẩy nhiệm vụ gửi email vào hàng đợi để hệ thống gửi sau. Người dùng không phải chờ, còn email sẽ được gửi trong nền.

- **Giảm tải cho hệ thống đỉnh điểm:** Khi có lượng lớn yêu cầu đến đột ngột (flash sale, sự kiện livestream, v.v.), hàng đợi có thể đóng vai trò như một bộ đệm. Các yêu cầu được xếp hàng trong queue thay vì tất cả dồn thẳng đến server xử lý cùng lúc. Server sẽ lấy yêu cầu ra dần dần để xử lý ở tốc độ nó chịu được. Như vậy hệ thống không bị "sập" vì quá tải, chỉ là người dùng có thể nhận kết quả chậm hơn một chút. Nhưng còn hơn là hệ thống chết hẳn, đúng không?

- **Giao tiếp giữa các dịch vụ (microservices):** Trong kiến trúc vi dịch vụ (microservices), các thành phần thường tách biệt và giao tiếp qua message queue để giảm phụ thuộc. Ví dụ: service A tạo một đơn hàng xong, nó không gọi thẳng service B (kho hàng) và service C (thanh toán) một cách đồng bộ. Thay vào đó, A sẽ gửi message "Order Created" vào queue. Các service B, C lắng nghe queue, nhận được message thì tự xử lý công việc của mình (B giảm số lượng tồn kho, C thu tiền...). Cách làm này giúp các service không bị khóa cứng chờ nhau, và nếu một service tạm thời chậm thì message vẫn nằm trong queue đợi, không làm sập dây chuyền.

- **Tính năng xếp hạng, thống kê, logging...:** Các hệ thống phân tích số liệu hoặc ghi log hoạt động thường dùng queue để thu thập sự kiện. Ví dụ: mỗi hành động của người dùng (click, view) được đẩy vào một queue "tracking". Hệ thống phân tích sẽ tiêu thụ dần các sự kiện này và tổng hợp báo cáo. Làm như vậy giảm tải cho server xử lý chính, và cũng tránh mất dữ liệu khi có quá nhiều sự kiện cùng lúc (vì queue có thể lưu lại, xử lý sau).

Tóm lại, hãy nghĩ đến Message Queue khi bạn muốn thực hiện điều gì đó "không cần kết quả tức thì" hoặc cần một cơ chế đệm giữa hai hệ thống có tốc độ xử lý khác nhau.

Một số hệ thống Message Queue phổ biến

Có nhiều công cụ và dịch vụ hiện thực hóa cơ chế hàng đợi thông điệp. Khi đi phỏng vấn hoặc thảo luận thiết kế, việc nêu tên vài hệ thống phổ biến sẽ ghi điểm vì cho thấy bạn có tìm hiểu thực tế:

- RabbitMQ: RabbitMQ là message broker mã nguồn mở rất thông dụng. Nó triển khai giao thức AMQP và thường được dùng cho các hàng đợi tác vụ (như xử lý background jobs). RabbitMQ hỗ trợ nhiều tính năng như routing linh hoạt (qua exchange, routing key), xác nhận message, cơ chế retry, v.v. Ưu điểm: dễ dùng, đáng tin cậy cho các trường hợp yêu cầu đảm bảo message được giao (reliability).

- Apache Kafka: Kafka thực chất là một nền tảng streaming phân tán, nhưng cũng thường được xếp vào nhóm hàng đợi thông điệp. Kafka được thiết kế cho throughput rất cao, lưu trữ message rất bền vững và cho phép subscribe theo nhóm (publish-subscribe). Kafka phù hợp khi bạn cần xử lý lượng lớn sự kiện (như log, sự kiện người dùng, dữ liệu thời gian thực) với tốc độ cao và muốn lưu lại lịch sử các message.

- AWS SQS (Amazon Simple Queue Service): Đây là dịch vụ hàng đợi hoàn toàn quản lý bởi Amazon Web Services. SQS rất dễ sử dụng: bạn tạo một queue trên AWS và các ứng dụng có thể gửi/nhận message mà không phải lo vận hành máy chủ. SQS đảm bảo độ tin cậy cao, tự động mở rộng và bạn trả phí theo lượng sử dụng. Tương tự SQS, các nền tảng cloud khác cũng có dịch vụ queue (Azure Service Bus, Google Cloud Pub/Sub...).

Ngoài ra, còn nhiều cái tên khác như ActiveMQ, IBM MQ, NSQ, Google Pub/Sub... Mỗi cái có ưu điểm riêng, nhưng nguyên lý chung của chúng đều là gửi-nhận message bất đồng bộ qua hàng đợi.

Mô hình Producer - Queue - Consumer

Như đã giải thích qua ví dụ quán ăn, mô hình tổng quát của hệ thống sử dụng message queue gồm ba thành phần chính:

- Producer (nhà sản xuất thông điệp): Đây là thành phần gửi message vào queue. Producer có thể là một phần của ứng dụng web, một service, hay thậm chí một script cron. Nhiệm vụ của Producer là tạo ra thông điệp mô tả công việc hoặc dữ liệu cần xử lý và đẩy nó vào hàng đợi. Ví dụ: web server nhận yêu cầu upload ảnh từ người dùng -> tạo message "resize ảnh X.png" và đưa vào queue "ImageTasks".

- Message Queue (hàng đợi thông điệp): Thành phần này giữ các message đã gửi cho đến khi có người nhận xử lý. Queue thường hoạt động theo nguyên tắc FIFO (First In First Out - vào trước ra trước) trừ khi có cấu hình ưu tiên khác. Mỗi message khi nằm trong queue sẽ đợi cho đến khi có consumer lấy nó ra. Queue có thể nằm trong bộ nhớ (nhanh nhưng mất dữ liệu nếu sập) hoặc ghi ra đĩa (chậm hơn chút nhưng bền vững). Nhiều hệ thống MQ (như RabbitMQ, Kafka) cho phép cluster hóa để đảm bảo queue không phải điểm thất bại duy nhất.

- Consumer (người tiêu thụ thông điệp): Đây là thành phần nhận message từ queue và thực hiện công việc tương ứng. Consumer thường chạy dưới dạng một hoặc nhiều tiến trình độc lập (có thể trên máy khác, container khác). Khi có sẵn năng lực xử lý, consumer sẽ lấy message kế tiếp từ queue (quá trình này thường gọi là poll hoặc subscribe tùy hệ thống) và thực thi. Ví dụ: một tiến trình worker nhận message "resize ảnh X.png" từ queue -> nó tiến hành resize ảnh đó và lưu kết quả. Sau khi xử lý xong, consumer có thể gửi một thông điệp phản hồi hoặc cập nhật trạng thái (không bắt buộc, tùy thiết kế hệ thống).

Quan trọng: trong mô hình này, producer và consumer không giao tiếp trực tiếp với nhau, chúng tách biệt thông qua queue. Producer không cần biết có bao nhiêu consumer, xử lý nhanh chậm ra sao; cứ đẩy message xong là xong việc của nó. Consumer cũng không biết chính xác ai gửi message, chỉ cần lấy message rồi làm. Điều này tạo ra tính decoupling (rời rạc) giữa các thành phần, giúp hệ thống linh hoạt và dễ mở rộng: muốn nhanh hơn, ta có thể tăng số lượng consumer chạy song song để xử lý nhiều message cùng lúc.

Cơ chế backpressure (phản áp lực) trong hàng đợi

Khi sử dụng hàng đợi, một vấn đề có thể nảy sinh: producer gửi message quá nhanh, consumer xử lý không kịp. Khi đó, các message sẽ ùn ứ lại trong queue, có thể dẫn đến đầy bộ nhớ hoặc đĩa, và cuối cùng làm sập hệ thống queue. Khái niệm backpressure đề cập đến việc kiểm soát dòng chảy này, nhằm tránh quá tải cho hệ thống tiêu thụ.

Bạn có thể hình dung backpressure như việc ở quán ăn, nếu quá nhiều khách hàng lấy số mà nhà bếp làm không kịp, cửa hàng tạm thời ngừng nhận thêm khách (hoặc yêu cầu khách xếp hàng chờ bên ngoài cửa nếu quá tải). Trong hệ thống phần mềm:

- Một số message queue có cơ chế chặn producer khi queue đầy. Ví dụ: RabbitMQ có thể đặt ngưỡng bộ nhớ, nếu vượt quá thì sẽ không nhận thêm message mới (hoặc trả về lỗi cho producer) cho đến khi consumer xử lý bớt và giải phóng

queue.

- Cơ chế khác là điều chỉnh tốc độ producer: ứng dụng gửi có thể được thiết kế để lắng nghe tín hiệu (như độ dài queue, hoặc phản hồi từ queue) để giảm tốc lại. Ví dụ: một service phát hiện queue đang chứa 10000 message chưa xử lý thì quyết định tạm ngừng gửi thêm việc mới hoặc giảm tần suất gửi.

- Nếu không kiểm soát được luồng gửi, một cách cuối là mở rộng hàng đợi hoặc consumer: ví dụ tự động scale-out thêm consumer để xử lý nhanh hơn, hoặc chấp nhận ghi queue ra đĩa (chậm nhưng lưu được nhiều) phòng trường hợp dồn ứ. Tuy nhiên, những biện pháp này chỉ giải quyết phần ngọn, quan trọng vẫn là thiết kế hệ thống để không dài hạn rơi vào trạng thái producer >> consumer quá lâu.

Từ khóa "backpressure" thường xuất hiện khi nói về các hệ thống xử lý stream hay message hiện đại (như Reactive Streams, Kafka). Trong phỏng vấn thiết kế hệ thống, bạn không nhất thiết phải đi quá sâu vào backpressure trừ khi được hỏi, nhưng hiểu đơn giản: backpressure = cách hệ thống phản ứng khi bị “ngộp” do tải. Bạn có thể đề cập rằng hệ thống hàng đợi của bạn sẽ cần cơ chế giới hạn hoặc thông báo khi queue quá đầy, ví dụ "Nếu số lượng message tồn đọng vượt X, tôi sẽ tạm dừng nhận thêm hoặc đưa ra cảnh báo để scaling".

Ví dụ minh họa sử dụng Message Queue

Hãy cùng xét một ví dụ để thấy rõ hơn lợi ích của message queue: Bạn xây dựng một trang web cho phép người dùng upload video và trang web sẽ xử lý chuyển đổi video đó (đổi định dạng, nén lại) sau khi tải lên. Quá trình chuyển đổi video có thể tốn cả phút, không thể bắt người dùng chờ trang web load lâu như vậy. Giải pháp: khi người dùng upload xong, server web sẽ lưu video, sau đó đưa một message vào hàng đợi (ví dụ queue "VideoProcessing") với nội dung "video ID 123 đã upload, hãy chuyển đổi nó". Sau đó, server phản hồi ngay cho người dùng (ví dụ: "Video của bạn đang được xử lý, bạn sẽ nhận được thông báo khi xong"). Bên phía hậu trường, bạn có một cụm các consumer (worker servers) chuyên xử lý video, chúng liên tục lắng nghe queue "VideoProcessing". Khi bắt được message "video 123", một worker sẽ lấy file video 123 ra, tiến hành chuyển đổi định dạng. Xong xuôi, nó có thể cập nhật trạng thái video trong database là "đã xử lý", và gửi email hoặc thông báo realtime cho người dùng biết đã xong.

Trong ví dụ trên, người dùng không phải chờ lâu, trang web không bị treo; phần việc nặng đã được đẩy qua hàng đợi cho các worker xử lý song song. Nếu nhiều người cùng upload, các video sẽ xếp hàng và lần lượt được xử lý, hoặc nếu bạn có nhiều worker thì có thể xử lý nhiều video cùng lúc. Đây chính là sức mạnh của kiến trúc bất đồng bộ với message queue.

Gợi ý khi phỏng vấn về Message Queue

Khi trả lời phỏng vấn về thiết kế hệ thống, nếu nhắc đến việc xử lý bất đồng bộ hoặc tích hợp các thành phần rời rạc, bạn nên đưa ra ý tưởng sử dụng message queue. Một cách diễn đạt có thể như sau: "Để hệ thống chịu tải tốt hơn, em sẽ tách các công việc nặng ra xử lý bất đồng bộ bằng message queue. Ví dụ, khi người dùng thực hiện hành động A, thay vì xử lý tất cả ngay lập tức, em sẽ đẩy một message vào queue để xử lý sau. Thành phần xử lý sẽ chạy nền, lấy message ra và hoàn thành công việc. Cách làm này giúp giảm thời gian phản hồi cho người dùng và làm hệ thống linh hoạt hơn. Em có thể dùng RabbitMQ hoặc Kafka cho hàng đợi này. Nếu lượng message tăng đột biến, em sẽ tăng số lượng consumer xử lý song song, đồng thời theo dõi độ dài queue để áp dụng backpressure nếu cần thiết."

Câu trả lời trên thể hiện bạn hiểu vì sao dùng queue (giảm thời gian chờ, decouple), biết ví dụ cụ thể (đưa tác vụ gửi email, xử lý video vào queue), và còn ghi điểm khi nhắc đến việc mở rộng consumer hay cơ chế backpressure khi tải cao. Tất nhiên, hãy tùy tình huống mà trả lời cho phù hợp, nhưng luôn nhớ nhấn mạnh lợi ích cốt lõi: Message Queue giúp hệ thống linh hoạt, chịu tải tốt hơn bằng cách phi đồng bộ hóa các tác vụ.

Idempotency và cơ chế khóa trong hệ thống phân tán

Idempotency (tính bất biến) là thuộc tính đảm bảo rằng thực hiện cùng một yêu cầu nhiều lần sẽ cho kết quả như chỉ thực hiện một lần. Điều này rất quan trọng trong hệ thống API và hệ phân tán vì giúp xử lý an toàn các trường hợp trùng lặp yêu cầu do lỗi mạng hoặc retry. Ví dụ, nếu gửi lệnh trừ 100k từ tài khoản ngân hàng nhiều lần, hệ thống idempotent sẽ chỉ trừ 100k một lần duy nhất. Nhờ đó, hệ thống tránh bị tác động phụ không mong muốn (không trừ tiền nhiều lần, không tạo đơn hàng lặp lại) và gia tăng độ tin cậy khi có lỗi kết nối.

- Ví dụ về idempotency: Các API GET, PUT, DELETE trong REST thường là idempotent: gọi nhiều lần không thay đổi kết quả cuối cùng. Ngược lại, POST hay

PATCH mặc định không idempotent, vì gọi nhiều lần sẽ tạo ra nhiều bản ghi mới hoặc thay đổi tích lũy dần. Để khắc phục, chúng ta thường gắn kèm "idempotency key" (token duy nhất) để xác định yêu cầu, đảm bảo khi retry thì chỉ thực hiện một lần duy nhất (cập nhật đúng một đơn hàng hoặc giao dịch).

Pessimistic Locking (Khóa bi quan)

Pessimistic Locking là cơ chế giả định có xung đột dữ liệu và chủ động đặt khóa để bảo vệ tài nguyên. Biện pháp này khóa tài nguyên từ trước và giữ khóa trong suốt giao dịch, ngăn người khác đọc/ghi cho đến khi giao dịch hoàn thành.

- Ưu điểm: Đảm bảo dữ liệu luôn nhất quán và an toàn, vì không ai có thể thay đổi trong lúc đang thao tác. Cách tiếp cận rõ ràng, đơn giản: ví dụ như trong giao dịch tài chính hay đặt chỗ quan trọng, ta chắc chắn tài nguyên chỉ một luồng truy cập.

- Nhược điểm: Dễ dẫn đến tình trạng chờ đợi và tắc nghẽn (các transaction khác phải chờ đến khi khóa được giải phóng). Nếu có nhiều giao dịch đồng thời cao, hiệu năng có thể giảm do lock bị giữ lâu, thậm chí gây deadlock.

Ví dụ: Khi một giao dịch chuyển tiền được tiến hành, có thể dùng pessimistic lock trên tài khoản để bảo đảm hai giao dịch không trừ tiền cùng lúc (tránh âm tài khoản). Tương tự, nếu hai người cùng sửa hồ sơ người dùng (profile), pessimistic locking có thể khóa bản ghi profile để người khác phải đợi. Trong SQL, ví dụ dùng `SELECT ... FOR UPDATE` để đặt khóa trên hàng dữ liệu. Khi giải phóng khóa (commit hoặc rollback), các transaction khác mới truy cập được.

Optimistic Locking (Khóa lạc quan)

Optimistic Locking là cơ chế giả định ít có xung đột, cho phép nhiều giao dịch đọc/ghi song song và chỉ kiểm tra xung đột khi commit. Cụ thể, mỗi bản ghi có thêm trường version hoặc timestamp. Khi cập nhật, hệ thống sẽ so sánh version hiện tại với version khi đọc ra. Nếu trùng khớp, ghi thay đổi và tăng version; nếu không (người khác đã thay đổi trước đó), giao dịch sẽ rollback hoặc retry.

- Ưu điểm: Không gây khóa cứng trên tài nguyên, tăng tính đồng thời và hiệu năng hệ thống khi xung đột hiếm. Thích hợp cho trường hợp nhiều client cùng đọc hoặc thực thi nhiều giao dịch nhỏ mà ít khi chạm trán (ví dụ hệ thống đọc nhiều,

ghi ít).

- Nhược điểm: Khi xung đột xảy ra (ví dụ hai giao dịch sửa cùng lúc), phải rollback hoặc retry, tăng độ phức tạp trong code. Không đảm bảo chính xác 100% dữ liệu ngay tức thì, chỉ phát hiện khi commit. Không phù hợp khi dữ liệu bị truy xuất/ghi sửa rất cao.

Ví dụ: Giả sử hai client cùng cập nhật thông tin người dùng. Với optimistic locking, mỗi client sẽ có version riêng khi đọc dữ liệu. Nếu client A commit thành công (version tăng lên), client B commit sẽ thất bại vì version cũ không khớp, buộc client B phải đọc lại và cập nhật lần nữa. Một ví dụ khác là tăng lượt xem trang: ta có thể không khóa truy vấn, chỉ cộng dồn và cập nhật cuối cùng, vì trường hợp xung đột không quan trọng. Trong code, ví dụ dùng câu lệnh `UPDATE ... WHERE id = ? AND version = ?`.

Khi sử dụng: Optimistic lock phù hợp với hệ thống ít xung đột (ví dụ workload đọc nhiều, giao dịch ngắn). Trong khi đó, Pessimistic lock được ưu tiên trong tình huống xung đột cao hoặc giao dịch dài (chẳng hạn hệ thống ngân hàng, hoặc khi đảm bảo tính chính xác tối đa là cần thiết).

Khi nào dùng & lựa chọn

- Idempotency: Nên áp dụng cho mọi API trong hệ phân tán có khả năng retry hoặc có thể gửi yêu cầu trùng lặp. Các API tạo giao dịch (chuyển tiền, đặt đơn hàng) đặc biệt cần idempotency để tránh double-charge hay tạo đơn kép. Nói chung, nếu luồng xử lý có thể thất bại và client retry, ta cần thiết kế idempotency key cho yêu cầu.

- Locking: Hầu hết ứng dụng web thông thường chấp nhận được dirty read nên có thể dùng optimistic locking để giảm thiểu deadlock và tăng tốc độ. Ngược lại, với các giao dịch tài chính hoặc kịch bản đòi hỏi chính xác tuyệt đối, pessimistic locking được ưu tiên vì đảm bảo tính chính xác mặc dù tốn hiệu năng hơn. Trong các buổi phỏng vấn System Design, bạn nên nhắc đến việc chọn optimistic locking nếu mức độ đồng thời cao và khả năng xung đột thấp, và chọn pessimistic locking khi nhất quán dữ liệu là ưu tiên hàng đầu.

Các phương thức giao tiếp và thiết kế API

Khi thiết kế hệ thống (đặc biệt trong bối cảnh phỏng vấn), việc lựa chọn phương thức giao tiếp và thiết kế API phù hợp là rất quan trọng. Mỗi dịch vụ trong hệ thống có thể cần trao đổi dữ liệu với dịch vụ khác hoặc với client, và có nhiều giao thức cũng như style API để thực hiện việc này. Chương 6 sẽ giới thiệu các phương thức giao tiếp phổ biến: gồm HTTP, TCP, UDP, RPC, và cách thiết kế API kiểu RESTful. Chúng ta sẽ tìm hiểu đặc điểm chính của từng phương thức, gợi ý khi nào nên dùng giao thức nào, và cách giải thích lựa chọn đó.

HTTP, Giao thức request/response phổ biến nhất

HTTP (HyperText Transfer Protocol) là giao thức tầng ứng dụng được sử dụng rộng rãi nhất cho web. HTTP hoạt động theo mô hình client-server dạng yêu cầu/đáp ứng (request-response): phía client (ví dụ: trình duyệt web của bạn) mở kết nối đến server, gửi một yêu cầu, rồi chờ server xử lý và trả về phản hồi tương ứng. HTTP có đặc tính stateless (phi trạng thái), nghĩa là bản thân server không ghi nhớ trạng thái phiên làm việc giữa các lần request: mỗi yêu cầu được xử lý độc lập với nhau. (Điều này lý giải vì sao khi xây dựng ứng dụng web chúng ta thường cần cơ chế như cookie/session hoặc token để duy trì trạng thái đăng nhập: đó là giải pháp bổ sung chứ HTTP không tự lưu trạng thái người dùng giữa các request).

Ví dụ minh họa: Khi bạn mở trình duyệt và truy cập https://www.example.com, trình duyệt sẽ gửi một HTTP request (cụ thể là một request GET) đến server của example.com. Server xử lý và trả về HTTP response (thí dụ: nội dung HTML của trang web). Mỗi lần bạn nhấp một đường link hay gửi form, một request mới lại được tạo ra. HTTP không mặc định nhớ bạn là ai qua các lần nhấp đó, trừ khi có dùng cơ chế bổ sung như cookie (nhằm "giả lập" trạng thái trên một giao thức stateless).

Các HTTP method (verb) cơ bản: HTTP định nghĩa nhiều phương thức để client yêu cầu hành động trên tài nguyên phía server. Phổ biến nhất là:

- GET: Lấy dữ liệu/tài nguyên từ server (ví dụ: tải một trang HTML, lấy thông tin user). GET thường chỉ đọc dữ liệu và không làm thay đổi dữ liệu trên server.

- POST: Gửi dữ liệu lên server (ví dụ: gửi form đăng ký, upload file) để tạo mới hoặc xử lý thông tin. POST có thể làm thay đổi trạng thái dữ liệu trên server (ví dụ tạo một bản ghi mới trong CSDL).

Ngoài ra HTTP còn có các phương thức khác như PUT (cập nhật tài nguyên), PATCH (cập nhật từng phần), DELETE (xóa tài nguyên) hay OPTIONS (hỏi server xem hỗ trợ những phương thức nào),... Mỗi phương thức có mục đích riêng, nhưng nhìn chung RESTful API thường tập trung vào 4 verb chính: GET (đọc), POST (tạo), PUT/PATCH (cập nhật), DELETE (xóa).

Khi nào nên dùng HTTP: Trong thiết kế hệ thống, HTTP gần như là lựa chọn mặc định cho các tương tác client-server trên web. Nếu bạn xây dựng một dịch vụ web, ứng dụng web hoặc API công khai cho bên thứ ba, HTTP/HTTPS là sự lựa chọn hàng đầu vì tính phổ biến và tính tương thích cao. HTTP (đặc biệt kết hợp với JSON/XML) giúp client trên mọi nền tảng (trình duyệt, mobile, v.v.) dễ dàng tương tác với server mà không cần thư viện đặc biệt. Ngoài ra, HTTP stateless nên dễ mở rộng theo chiều ngang: server có thể xử lý mỗi request độc lập, giúp phân tải sang nhiều server khác nhau.

TCP: Giao thức kết nối tin cậy

TCP (Transmission Control Protocol) là giao thức truyền tải hướng kết nối và đáng tin cậy. “Hướng kết nối” nghĩa là trước khi truyền dữ liệu, hai bên phải thiết lập kết nối với nhau; quá trình này thường được gọi là bắt tay ba bước (3-way handshake) giữa client và server. Sau khi kết nối thiết lập, TCP đảm bảo dữ liệu được truyền đi một cách đáng tin cậy, không bị thất lạc, và theo đúng thứ tự tới đích. Cụ thể, TCP đánh số thứ tự các gói tin, yêu cầu bên nhận gửi xác nhận (ACK) cho mỗi gói nhận được, và sẽ tự động gửi lại nếu phát hiện gói tin bị mất hoặc lỗi. Nhờ các cơ chế này, dữ liệu qua TCP gần như đến nơi nguyên vẹn hoặc sẽ được truyền lại đến khi thành công.

Ví dụ minh họa: Khi bạn tải xuống một tệp tin từ internet hoặc truyền dữ liệu giữa ứng dụng và cơ sở dữ liệu, thường quá trình này sử dụng TCP. Chẳng hạn, kết nối từ ứng dụng của bạn đến database (MySQL, PostgreSQL, v.v.) hoặc giao thức FTP truyền file đều chạy trên TCP: nhờ đó đảm bảo toàn bộ nội dung file hoặc truy vấn database tới nơi không sai sót. Nếu một gói tin chứa phần dữ liệu file bị thất lạc trên đường truyền, TCP sẽ tự động phát hiện và gửi lại gói đó, nhờ vậy file tải về không bị hỏng.

Đặc điểm chính: TCP cung cấp nhiều tính năng mạnh mẽ ở tầng giao vận:

- Kết nối tin cậy: Thiết lập phiên kết nối riêng giữa hai bên, đảm bảo hai máy “bắt tay” trước khi trao đổi dữ liệu.

- Kiểm soát luồng và lỗi: TCP kiểm soát tốc độ gửi để tránh nghẽn mạng, và có cơ chế phát hiện lỗi/các gói tin thất lạc để gửi lại cho đến khi nhận thành công.

- Tuần tự hóa: Gói tin được đánh số thứ tự, đảm bảo bên nhận lắp ráp đúng thứ tự như ban đầu.

Nhờ những cơ chế này, TCP phù hợp với các ứng dụng đòi hỏi độ chính xác của dữ liệu hơn là tốc độ.

Khi nào nên dùng TCP: Bạn nên chọn TCP cho các tình huống cần đảm bảo dữ liệu chính xác tuyệt đối. Ví dụ điển hình: truyền tệp tin, gửi email SMTP, kết nối CSDL, giao dịch tài chính : nơi việc mất mát hay sai lệch dữ liệu là không chấp nhận được. Trong thiết kế hệ thống, ở mức ứng dụng cao hơn, nhiều giao thức khác thực chất cũng xây dựng trên nền TCP để hưởng lợi từ tính tin cậy của nó. HTTP như đề cập ở trên thường chạy trên TCP (cổng 80 hoặc 443), nghĩa là mọi request HTTP đều được TCP đảm bảo chuyển đủ và đúng thứ tự. Do đó, khi thiết kế một API hay dịch vụ, nếu không nói gì khác thì ngầm định TCP đang được sử dụng bên dưới để truyền dữ liệu.

Tuy nhiên, TCP đòi hỏi thiết lập và duy trì kết nối, có bắt tay 3 bước, nên sẽ có độ trễ khởi tạo và chút overhead quản lý phiên. Vì vậy trong những trường hợp ưu tiên tốc độ hơn độ tin cậy (xem phần UDP bên dưới), ta có thể cân nhắc giao thức khác.

Mẹo phỏng vấn: Nếu được hỏi về tầng giao vận hoặc truyền thông tin giữa các thành phần trong hệ thống, đừng quên nhắc tới TCP như là xương sống của internet. Bạn có thể nói: “Em chọn TCP vì cần sự tin cậy: mọi gói tin đều đến nơi hoặc được truyền lại. Với những dữ liệu quan trọng như kết quả giao dịch ngân hàng hay file backup, TCP là phù hợp.” Chứng tỏ bạn hiểu ưu/nhược: TCP an toàn nhưng chậm hơn UDP do phải thiết lập kết nối và xác nhận gói tin.

UDP: Giao thức “gửi là quên”

UDP (User Datagram Protocol) là giao thức truyền tải hướng không kết nối (connectionless), trái ngược với TCP. Với UDP, bên gửi không cần bắt tay thiết lập kết nối trước; dữ liệu cứ có là gửi đi ngay, không chờ đợi. Giao thức UDP cũng không đảm bảo độ tin cậy: nghĩa là không có cơ chế xác nhận gói tin đã đến hay chưa, không tự động gửi lại nếu gói bị thất lạc, và không đảm bảo thứ tự nhận giống thứ tự gửi. Nói ngắn gọn, UDP hoạt động kiểu “fire-and-forget” (gửi đi và không chờ hồi đáp).

Ví dụ minh họa: Hãy hình dung bạn đang gọi video call hoặc chơi game online. Những ứng dụng real-time này thường sử dụng UDP để truyền dữ liệu âm thanh, hình ảnh hoặc trạng thái trò chơi. Lý do: UDP rất nhanh, độ trễ thấp: nó không mất thời gian bắt tay hay chờ ACK như TCP. Nếu một vài gói tin chứa dữ liệu âm thanh bị mất, cuộc gọi có thể hơi

nhiễu một chút nhưng phần lớn vẫn hiểu được, và quan trọng hơn là âm thanh hình ảnh vẫn đến liên tục, đúng thời gian thực. Tương tự, trong game online, nếu một vài gói tin vị trí nhân vật bị rớt, trò chơi vẫn tiếp tục (có thể đối thủ sẽ "dịch chuyển nhẹ" một cách hơi giật), bù lại game phản hồi nhanh. Những trường hợp này chấp nhận mất mát dữ liệu nhỏ để đổi lấy tốc độ. Ngoài ra, dịch vụ DNS (hệ thống phân giải tên miền) cũng dùng UDP cho các truy vấn nhỏ: mỗi truy vấn DNS là một gói tin độc lập hỏi IP của một domain, dùng UDP giúp xử lý hàng triệu truy vấn nhanh chóng và nếu một gói DNS thất lạc, client có thể gửi lại truy vấn mà không tốn nhiều chi phí.

Đặc điểm chính: Vì không đảm bảo và không có phiên kết nối, UDP có một số ưu điểm và nhược điểm rõ rệt:

- Ưu điểm: Tốc độ rất cao và độ trễ thấp. Không tốn thời gian bắt tay, header của gói UDP cũng nhỏ gọn, giúp truyền tải nhanh. Phù hợp cho truyền dữ liệu liên tục, real-time, multicast/broadcast đến nhiều thiết bị.

- Nhược điểm: Không tin cậy. Gói tin có thể bị mất hoặc đến lộn xộn, ứng dụng phải tự chịu trách nhiệm nếu cần độ chính xác. Không thích hợp cho dữ liệu quan trọng cần độ chính xác hoàn hảo (vì UDP "không quan tâm" gói tin có tới đích không).

Khi nào nên dùng UDP: Hãy chọn UDP cho các ứng dụng đòi hỏi tốc độ và thời gian thực, nơi mà việc mất một ít dữ liệu là chấp nhận được. Các ví dụ điển hình: VoIP (thoại/video qua IP), streaming video/truyền hình trực tuyến, trò chơi trực tuyến, hoặc truyền các gói tin cảm biến IoT liên tục. UDP cũng hữu ích cho các truy vấn nhỏ, số lượng lớn như DNS hoặc ping mạng, nơi thiết lập kết nối TCP cho từng truy vấn sẽ tốn kém không cần thiết.

Trong thiết kế hệ thống, nếu bạn xây dựng một dịch vụ streaming hay chat voice, đừng quên cân nhắc UDP ở tầng transport cho phần truyền tải media. Thực tế nhiều giao thức streaming (như RTP trong VoIP, hoặc QUIC mới của HTTP/3) đều tận dụng UDP để đạt hiệu suất cao, sau đó tự bổ sung một số cơ chế kiểm soát ở tầng ứng dụng.

Mẹo phỏng vấn: Nếu được hỏi về cách tối ưu tốc độ cho truyền dữ liệu thời gian thực, bạn có thể trả lời: "Em dùng UDP vì nhanh và ít độ trễ. Ví dụ stream video, mất một vài frame còn hơn bị trễ hình." Cho thấy bạn hiểu trade-off: UDP nhanh do không overhead bắt tay, nhưng đổi lại không đảm bảo: và bạn chấp nhận điều đó cho use case phù hợp. Bạn cũng có thể đề cập rằng đôi khi lập trình viên phải tự

xây dựng cơ chế kiểm lỗi nếu dùng UDP, nhưng với streaming real-time thì thường "mất thì thôi" để giữ mạch dữ liệu liên tục.

RPC: Gọi thủ tục từ xa như gọi hàm nội bộ

RPC (Remote Procedure Call) là một cơ chế giao tiếp cho phép gọi hàm trên một máy khác giống như đang gọi hàm cục bộ trong chương trình. Nói cách khác, RPC là phương pháp để một chương trình yêu cầu dịch vụ từ chương trình khác trên máy chủ khác và nhận kết quả trả về qua mạng. Lập trình viên có thể gọi một hàm hoặc phương thức mà thực chất hàm đó chạy ở server từ xa, mọi phức tạp về gửi yêu cầu qua mạng, chờ phản hồi, chuyển đổi dữ liệu… đều được RPC abstraction xử lý, giúp cho việc gọi hàm từ xa trở nên trực quan như gọi hàm local.

Ví dụ minh họa: Giả sử bạn có một ứng dụng với kiến trúc microservices, trong đó có Service A và Service B. Thay vì Service A gửi một request HTTP đến B rồi parse JSON, bạn có thể sử dụng RPC (ví dụ: gRPC) để Service A gọi thẳng hàm getUserInfo(userId) trên Service B và nhận về kết quả như một đối tượng, y như đang gọi hàm trong cùng một project. Lập trình viên chỉ cần gọi hàm, RPC framework lo hết việc truyền qua mạng. Các framework RPC phổ biến hiện nay gồm gRPC (dựa trên HTTP/2 và protocol buffers), Thrift, Apache Dubbo, v.v. chúng thường tạo ra code "stub" phía client và "service skeleton" phía server để thực hiện lời gọi từ xa một cách trong suốt.

Ưu nhược điểm của RPC:

- Ưu điểm: RPC thường có hiệu suất cao (sử dụng giao thức nhị phân, ít overhead hơn so với HTTP text). Giao tiếp RPC có độ trễ thấp, rất hữu dụng trong nội bộ hệ thống microservices cần trao đổi nhiều. Ngoài ra, RPC cung cấp trải nghiệm lập trình tiện lợi: gọi hàm từ xa như gọi hàm thường, giúp lập trình viên tập trung vào logic thay vì xử lý giao tiếp mạng.

- Nhược điểm: RPC có thể dẫn đến khớp nối chặt (tight coupling) giữa client và server: cả hai bên phải hiểu cùng một interface hàm (thường qua file định nghĩa .proto hoặc IDL). Việc thay đổi giao thức RPC (thêm tham số hàm, đổi kiểu dữ liệu…) có thể đòi hỏi cập nhật cả client lẫn server. Ngoài ra, RPC thường không được hỗ trợ trực tiếp trên trình duyệt web (ví dụ gRPC thuần dùng HTTP/2 không tương thích với call từ Javascript thuần, phải dùng gRPC-Web proxy). Vì vậy RPC hay được dùng cho giao tiếp backend-backend hơn là cho client frontend gọi thẳng.

Khi nào nên dùng RPC: RPC phù hợp cho hệ thống phân tán nội bộ, kiến trúc microservice: nơi các service thường gọi lẫn nhau. Nếu bạn cần hiệu năng cao và các dịch vụ có thể được update đồng bộ (cùng kiểm soát), RPC là lựa chọn tốt. Ví dụ, trong hệ thống doanh nghiệp nội bộ, các dịch vụ viết bằng Go hoặc Java trong cùng công ty có thể dùng gRPC để giao tiếp: vừa nhanh, gọn (nhờ Protocol Buffers), vừa có kiểu dữ liệu chặt chẽ. RPC cũng lý tưởng cho các tình huống yêu cầu tương tác kiểu request-response nhanh, chẳng hạn hệ thống quảng cáo real-time đấu giá: độ trễ mỗi request cần tính bằng milliseconds. Tuy nhiên, nếu hệ thống của bạn mở API ra ngoài cho nhiều client khác nhau hoặc bạn muốn sự linh hoạt, bạn có thể ưu tiên RESTful HTTP (vì client không cần thư viện đặc biệt và dễ debug hơn).

Mẹo phỏng vấn: Khi thảo luận về kiến trúc microservices, bạn có thể đề cập: "Với giao tiếp giữa các service nội bộ, em đề xuất dùng RPC (ví dụ gRPC) thay cho REST, do hiệu suất cao và giao tiếp trực tiếp kiểu hàm." Điều này cho thấy bạn nắm bắt xu hướng sử dụng RPC trong các hệ thống lớn (Google, Facebook đều dùng RPC nội bộ). Đừng quên nói rõ lý do: RPC nhanh hơn REST do binary format, và code sinh ra dùng được ngay như hàm local. Nhưng cũng thể hiện rằng bạn nhận thức hạn chế: RPC phức tạp hơn để triển khai, giám sát, và không phù hợp public API. Nếu interviewer hỏi sâu, bạn có thể nhắc đến việc gRPC cần HTTP/2 và không gọi trực tiếp từ browser (phải có gateway), cho thấy bạn hiểu bối cảnh sử dụng RPC.

REST: Phong cách thiết kế API rõ ràng, thống nhất

REST (Representational State Transfer) không phải một giao thức cụ thể, mà là một phong cách kiến trúc thiết kế API dựa trên giao thức HTTP. Một API được gọi là "RESTful" nếu nó tuân thủ các nguyên tắc REST: trong đó tài nguyên (resources) là trung tâm. Mỗi tài nguyên được định danh bằng một URL (điểm cuối), và việc thao tác dữ liệu sẽ thông qua các phương thức HTTP chuẩn như GET, POST, PUT, DELETE trên các URL đó. REST tận dụng tối đa đặc tính stateless của HTTP : mỗi request tự chứa đủ thông tin, server không cần nhớ trạng thái trước đó: giúp hệ thống dễ mở rộng và phân tán tốt (mỗi thành phần có thể độc lập xử lý).

Ví dụ minh họa: Giả sử bạn thiết kế một hệ thống quản lý công việc với RESTful API. Bạn có thể định nghĩa tài nguyên "công việc" (task) và "người dùng" (user) như sau:

- GET /tasks : lấy danh sách tất cả công việc.

- POST /tasks : tạo mới một công việc. (Dữ liệu công việc mới sẽ gửi kèm trong body dạng JSON).

- GET /tasks/123 : lấy chi tiết công việc với ID 123.

- PUT /tasks/123 : cập nhật thông tin công việc 123 (thay thế toàn bộ) hoặc PATCH /tasks/123 để cập nhật một phần.

- DELETE /tasks/123 : xóa công việc 123.

Tương tự, tài nguyên người dùng (/users) cũng có các endpoint GET/POST... REST hướng đến việc sử dụng nhất quán các verb HTTP cho các loại thao tác giống nhau (GET luôn để lấy dữ liệu, POST để tạo mới, v.v.), giúp API dễ hiểu và dự đoán được. Phản hồi từ REST API thường là JSON (trước đây đôi khi XML), đại diện cho trạng thái tài nguyên. Ví dụ, GET /tasks/123 trả về JSON của task 123, còn DELETE /tasks/123 có thể trả về mã trạng thái 204 No Content nếu xóa thành công.

Ưu nhược điểm của RESTful API:

- **Ưu điểm:** REST đơn giản và rõ ràng. API thiết kế theo REST dễ dàng cho người khác sử dụng vì tuân theo chuẩn chung (nhìn endpoint và method là hiểu được phần nào chức năng). Tính không trạng thái và phân lớp của REST giúp nó mở rộng tốt: ta có thể nhân bản nhiều server phục vụ API mà không lo đồng bộ session. Ngoài ra, REST tận dụng hạ tầng HTTP sẵn có (caching, authentication bằng token header, mã trạng thái response...) nên rất tiện lợi và tương thích rộng (hầu như mọi ngôn ngữ, nền tảng đều gọi được HTTP).

- **Nhược điểm:** Do dùng HTTP và thường trao đổi dữ liệu dạng text (JSON/XML), RESTful API có thể chậm hơn và ngốn băng thông hơn so với các giải pháp nhị phân tối ưu (như RPC/gRPC). REST cũng không ép chặt chẽ cấu trúc response, nên nếu thiết kế không cẩn thận có thể dẫn đến API thiếu nhất quán giữa các endpoint. Bên cạnh đó, tính stateless mặc dù tốt cho mở rộng nhưng đôi khi buộc client gửi lặp lại nhiều thông tin (ví dụ mỗi request đều phải kèm token xác thực, không tận dụng được thông tin đã biết từ request trước).

Khi nào nên dùng REST: Hầu hết các dịch vụ web công khai và API cho đối tác/khách hàng ngày nay đều sử dụng RESTful API. Nếu bạn xây dựng một Web Service hoặc Microservice mà client tiêu thụ rất đa dạng (trình duyệt web, ứng dụng mobile, hệ thống

bên thứ ba), REST là lựa chọn an toàn. Nó mang lại khả năng tương tác (interoperability) cao: bất kỳ client nào hiểu HTTP đều dùng được. REST cũng thích hợp cho các hệ thống lớn cần loose coupling: mỗi service có thể phát triển độc lập, miễn là tuân thủ hợp đồng API (các URL endpoints và định dạng request/response). Ngay cả khi nội bộ hệ thống bạn có thể dùng gRPC, bạn vẫn có thể xây dựng một lớp API Gateway RESTful bên ngoài để tương thích với mọi client công cộng.

Mẹo phỏng vấn: Nếu được hỏi “Vì sao chọn REST cho API này?”, bạn có thể trả lời: “Vì em muốn một API dễ sử dụng, tiêu chuẩn. RESTful API dùng HTTP nên bất cứ client nào cũng kết nối được, và dễ dàng thử nghiệm (có thể gọi bằng cURL, Postman). Kiến trúc REST lại stateless, scale out rất dễ. Hơn nữa, REST đã quá phổ biến trong giới lập trình, nên team nào cũng có thể nhanh chóng hiểu và dùng API của mình.” Điều này cho thấy bạn không chỉ biết REST là gì, mà còn hiểu giá trị của nó trong thiết kế hệ thống thực tiễn.

Đảm bảo khả năng chịu lỗi và phục hồi hệ thống

Khả năng chịu lỗi là gì và tại sao quan trọng?

Khả năng chịu lỗi (**fault tolerance**) của một hệ thống là khả năng tiếp tục hoạt động bình thường ngay cả khi một hoặc nhiều thành phần gặp sự cố. Nói cách khác, khi xảy ra lỗi thì hệ thống vẫn có thể vận hành (dù có thể với hiệu năng giảm) thay vì sập hoàn toàn. Điều này đặc biệt quan trọng trong các hệ thống phân tán và những dịch vụ yêu cầu độ sẵn sàng cao, bởi vài phút hoặc thậm chí vài giây ngừng phục vụ cũng có thể dẫn tới thiệt hại lớn về tài chính và uy tín. Thực tế cho thấy "mọi thứ có thể hỏng sẽ hỏng": không có hệ thống nào đảm bảo 100% uptime mãi mãi, nên việc lường trước sự cố và chuẩn bị phương án phục hồi là yêu cầu bắt buộc. Xây dựng hệ thống có khả năng chịu lỗi giúp đảm bảo tính liên tục cho dịch vụ, mang lại trải nghiệm tin cậy hơn cho người dùng và tránh được những hậu quả nghiêm trọng khi sự cố xảy ra.

Các sự cố phổ biến trong hệ thống phân tán

Trong một hệ thống lớn, lỗi có thể xảy ra ở bất kỳ đâu : từ máy chủ, kết nối mạng cho tới chính ứng dụng. Dưới đây là một số loại sự cố phổ biến trong hệ thống phân tán và phân tích kiến trúc:

- Lỗi phần cứng (Hardware failure): Máy chủ vật lý có thể bị hỏng đột ngột (cháy nguồn, lỗi CPU/RAM), ổ đĩa cứng có thể hỏng mất dữ liệu, hoặc toàn bộ máy chủ (node) có thể sập do sự cố điện. Đây là những lỗi khó tránh khỏi khi vận hành nhiều máy móc trong thời gian dài. Ví dụ, một cụm máy chủ có thể đột ngột mất một node do crash mà không báo trước.

- Lỗi phần mềm (Software bug): Lỗi trong code ứng dụng hoặc dịch vụ có thể dẫn đến memory leak khiến server chậm dần và treo, hoặc một bug logic khiến kết quả trả về sai. Những lỗi logic nghiêm trọng có thể làm một dịch vụ con không hoạt động đúng, ảnh hưởng dây chuyền đến các phần khác.

- Sự cố mạng (Network issue): Kết nối mạng có thể bị gián đoạn hoặc chập chờn. Trong hệ thống phân tán, đôi khi xảy ra hiện tượng network partition: một nhóm máy bị tách khỏi nhóm còn lại do lỗi mạng. Kết quả là các thành phần không

"nhìn thấy" nhau, gây ra độ trễ cao hoặc timeout. Các lỗi mạng như mất gói tin, mất kết nối hoặc đường truyền chậm cũng khá thường gặp.

- Lỗi do con người (Human error): Sai sót khi cấu hình hệ thống, thao tác nhầm (ví dụ xóa nhầm dữ liệu hoặc tắt nhầm server) cũng là nguyên nhân phổ biến gây sự cố. Dù không phải lỗi kỹ thuật của hệ thống, nhưng đây là yếu tố phải tính đến khi thiết kế cơ chế phục hồi.

- Quá tải hệ thống: Khi lưu lượng vượt quá dự kiến, một thành phần có thể quá tải (CPU 100%, đầy bộ nhớ hoặc hết kết nối). Lúc này dịch vụ có thể trở nên không phản hồi (giống như lỗi) và thậm chí sập. Quá tải thường không được coi là "lỗi" đơn lẻ, nhưng hiệu ứng gây ra tương tự một sự cố cần được xử lý.

Nhìn chung, "lỗi hệ thống" có thể đến từ nhiều nguyên nhân đa dạng: phần cứng, phần mềm, mạng, con người, môi trường vận hành,... Điều quan trọng là hệ thống của chúng ta được thiết kế để chịu được những lỗi này ở một mức độ chấp nhận được, thay vì dừng hoạt động hoàn toàn khi có sự cố.

Chiến lược tăng độ bền vững (resilience) cho hệ thống

Để xây dựng hệ thống chịu lỗi tốt, kỹ sư phải áp dụng nhiều chiến lược thiết kế giúp hệ thống vẫn hoạt động (hoặc nhanh chóng phục hồi) khi gặp sự cố. Dưới đây là các chiến lược phổ biến nhằm tăng tính bền vững và khả năng phục hồi cho hệ thống, kèm ví dụ minh họa:

- **Dự phòng và nhân bản (Redundancy & Replication):** Nguyên tắc "không để trứng vào một giỏ": luôn có sẵn thành phần dự phòng để thay thế khi thành phần chính gặp lỗi. Ví dụ: triển khai nhiều máy chủ ứng dụng giống nhau thay vì một cái (nếu một máy hỏng thì còn máy khác phục vụ), hoặc sao lưu/nhân bản dữ liệu quan trọng sang nhiều máy/ổ đĩa khác nhau. Khi một database có bản sao ở máy chủ thứ hai, nếu máy chủ chính hỏng thì hệ thống có thể chuyển sang dùng bản sao này. Việc replicate dữ liệu và dịch vụ trên nhiều node bảo đảm không có điểm lỗi duy nhất (single point of failure) nào: một thành phần ngừng thì đã có thành phần khác gánh thay.

- **Chuyển đổi dự phòng (Failover):** Đây là cơ chế tự động chuyển sang hệ thống backup khi hệ thống chính gặp trục trặc. Hình dung một cụm dịch vụ có một server chính (active) và một server chế độ chờ (passive). Bình thường server

chính xử lý toàn bộ lưu lượng, server dự phòng chỉ "stand by". Nếu server chính sập, cơ chế failover sẽ ngay lập tức chuyển người dùng sang server dự phòng để tiếp tục phục vụ, giảm thiểu gián đoạn. Ví dụ thực tế: một trang web chạy ở datacenter A có một bản dự phòng chạy ở datacenter B; khi A gặp sự cố (đứt mạng hoặc mất điện), hệ thống sẽ chuyển traffic sang B. Failover thường kết hợp chặt chẽ với redundancy và health-check: hệ thống phải phát hiện nhanh server chính bị down (nhờ cơ chế kiểm tra sức khỏe) rồi kích hoạt việc chuyển đổi.

- **Cân bằng tải (Load Balancing):** Đây là chiến lược vận hành nhiều máy chủ song song và phân phối yêu cầu giữa chúng một cách hợp lý. Cân bằng tải giúp tránh tình trạng một máy chủ bị quá tải dẫn đến sập, đồng thời nếu một máy hỏng thì load balancer sẽ loại nó ra khỏi vòng và chuyển người dùng sang các máy còn lại. Nhờ đó hệ thống vẫn phục vụ dù mất đi một phần công suất: hiệu năng có thể giảm chút ít nhưng không ngừng hẳn. Ví dụ: một hệ thống web có 5 server sau một load balancer; nếu 1 server chết, 4 server còn lại vẫn tiếp tục xử lý các request (người dùng có thể không hề hay biết, chỉ có thể thấy trang đáp ứng chậm hơn chút).

- **Thử lại khi lỗi (Retry logic):** Nhiều lỗi trong hệ phân tán mang tính tạm thời, ví dụ gói tin mạng bị rớt, hoặc một dịch vụ đang restart. Vì vậy, chiến lược thường dùng là tự động thử lại yêu cầu sau khi thất bại, thường kèm cơ chế chờ backoff tăng dần. Thay vì bỏ cuộc ngay, hệ thống sẽ thử gửi lại yêu cầu sau một khoảng thời gian (vài giây chẳng hạn). Quan trọng là phải có giới hạn số lần retry và khoảng cách giữa các lần thử, để tránh dồn dập quá mức vào một dịch vụ đang lỗi. Ví dụ: nếu call API tới service B thất bại, service A có thể chờ 2 giây rồi thử lại, lần tiếp theo chờ 4 giây rồi thử tiếp... Việc retry với khoảng nghỉ tăng dần (exponential backoff) giúp tăng cơ hội thành công khi lỗi chỉ là tạm thời, đồng thời tránh gây quá tải thêm cho dịch vụ đang gặp sự cố. Lưu ý rằng không phải lúc nào retry cũng hiệu quả: cần kết hợp với cơ chế phát hiện lỗi dài hạn để ngừng thử nếu lỗi có khả năng kéo dài (xem circuit breaker bên dưới).

- **Giảm thiểu dịch vụ (Graceful Degradation):** Thay vì sập hoàn toàn khi có lỗi, hệ thống được thiết kế để giảm dần tính năng một cách có kiểm soát, ưu tiên duy trì những chức năng cốt lõi. Ý tưởng là "thà chạy hạn chế còn hơn chết hẳn". Ví dụ: khi một tính năng phụ trợ bị lỗi, ta có thể tạm thời tắt nó đi và chỉ cung cấp những phần chính cho người dùng. Một mạng xã hội nếu module cập nhật bình luận trực tiếp bị hỏng thì có thể tắt tính năng này tạm thời, nhưng news feed chính vẫn hoạt động bình thường. Tương tự, một dịch vụ xem phim trực tuyến nếu hệ thống phân phối HD gặp sự cố thì có thể giảm chất lượng video xuống độ phân giải thấp thay

vì để người dùng bị gián đoạn hoàn toàn. Kỹ thuật graceful degradation thường đi kèm với circuit breaker: tức là ngắt dòng yêu cầu đến thành phần đang lỗi để tránh lỗi lan rộng, sau đó cố gắng phục hồi dần. Nói ngắn gọn: khi mọi thứ bắt đầu "toang", hãy ưu tiên "chạy chậm còn hơn đứng hẳn": tắt bớt phần kém quan trọng, giữ phần quan trọng tiếp tục chạy.

- **Kiểm tra sức khỏe và giám sát (Health Check & Monitoring):** Bạn không thể khắc phục sự cố nếu không sớm biết là nó đang xảy ra. Do đó, hệ thống cần liên tục theo dõi trạng thái các thành phần và cảnh báo khi có dấu hiệu bất thường. Health check là các cơ chế kiểm tra tự động: ví dụ: mỗi 5 giây server A ping server B một lần, nếu 3 lần không thấy phản hồi thì kết luận B đã chết và kích hoạt quy trình failover. Bên cạnh đó, cần thu thập các chỉ số runtime (CPU, bộ nhớ, độ trễ, tỉ lệ lỗi) và thiết lập cảnh báo. Các công cụ như Prometheus/Grafana, Datadog… cho phép đo lường và vẽ dashboard cho hệ thống; dịch vụ như PagerDuty có thể gửi alert cho kỹ sư trực nếu có sự cố lớn. Nhờ giám sát tốt, chúng ta có thể phát hiện sớm vấn đề (thậm chí trước khi người dùng nhận ra) và phản ứng kịp thời. Health check kết hợp với load balancing: ví dụ load balancer sẽ định kỳ health-check các server, server nào không thông sẽ bị loại khỏi vòng phân tải. Giám sát cũng cung cấp dữ liệu để phân tích sau sự cố và cải thiện hệ thống.

Ngoài các kỹ thuật trên, còn nhiều chiến lược nâng cao khác khi thiết kế hệ thống chịu lỗi (như cách ly lỗi bằng bulkhead pattern, cơ chế tự phục hồi - self-healing, v.v.). Tuy nhiên, những ý chính kể trên là nền tảng cơ bản cho một hệ thống đáng tin cậy. Tất nhiên, đánh đổi của việc thêm khả năng chịu lỗi là chi phí cao hơn và hệ thống phức tạp hơn. Nhiệm vụ của kỹ sư là cân bằng giữa mức độ chịu lỗi cần thiết và chi phí chấp nhận được. Nhưng nói chung, "cái giá của sự cố còn đắt hơn cái giá của phòng ngừa": thà tốn chi phí xây dựng dự phòng còn hơn bị gọi dậy lúc 3 giờ sáng vì hệ thống sập hoàn toàn.

Phân biệt High Availability và Fault Tolerance

Hai khái niệm High Availability (HA) - độ sẵn sàng cao; và Fault Tolerance (FT) - chịu lỗi - thường được nhắc cùng nhau, nhưng chúng không hoàn toàn giống nhau. Cả hai đều nhằm mục tiêu giữ cho dịch vụ liên tục phục vụ, nhưng cách tiếp cận và chi phí khác nhau:

- **High Availability (HA):** Hệ thống sẵn sàng cao nghĩa là thiết kế để giảm thiểu tối đa thời gian downtime, nhưng chấp nhận thỉnh thoảng vẫn có gián đoạn nhỏ. Một hệ thống HA đảm bảo hầu hết thời gian hệ thống ở trạng thái phục vụ bình

thường. Khi có sự cố, hệ thống HA sẽ khôi phục nhanh hoặc chuyển đổi nhanh sang thành phần dự phòng, nhưng thường vẫn có một chút thời gian ngắn ngừng phục vụ (ví dụ vài chục giây hoặc một phút). Ưu điểm của HA là dễ triển khai hơn và chi phí thấp hơn so với FT. Nhiều hệ thống sản xuất chọn HA vì sẵn sàng chấp nhận downtime rất ngắn để đổi lấy kiến trúc đơn giản hơn.

- **Fault Tolerance (FT):** Hệ thống chịu lỗi hướng tới mục tiêu không gián đoạn dịch vụ dù xảy ra bất kỳ lỗi nào. Điều này thường đạt được bằng cách nhân đôi toàn bộ hệ thống theo kiểu chạy song song: nếu một thành phần hỏng thì đã có thành phần khác đang chạy song song gánh ngay tức thì, người dùng hầu như không nhận thấy sự cố. Ví dụ, một cơ sở dữ liệu quan trọng chạy đồng thời hai máy chủ theo kiểu active-active (cả hai cùng xử lý, luôn đồng bộ dữ liệu); nếu một máy gặp trục trặc, máy còn lại đã có dữ liệu và phiên làm việc sẵn để tiếp tục phục vụ mà không cần thời gian chuyển đổi. Fault tolerance mang lại khả năng phục vụ liên tục nhất, nhưng đánh đổi là chi phí rất cao (vì phải nhân bản mọi thứ, tài nguyên dự phòng nhiều) và kiến trúc hệ thống phức tạp hơn đáng kể. Ngoài ra, không phải lúc nào FT cũng xử lý được mọi loại lỗi: ví dụ lỗi phần mềm chung (bug) thì nhân bản bao nhiêu cũng vẫn bị lỗi như nhau nếu không có cơ chế loại trừ.

Một cách đơn giản để hình dung sự khác biệt:

- HA giống như nhà hàng có nhiều đầu bếp và nguyên liệu dự trữ. Nếu một đầu bếp nghỉ, nhà hàng vẫn mở cửa nhưng có thể phục vụ chậm hơn một chút : vẫn có downtime nhưng rất ít.

- FT giống như nhà hàng có hai hệ thống điện độc lập và máy phát dự phòng: nếu lưới điện mất thì máy phát ngay lập tức cung cấp điện, nhà hàng không hề tắt đèn một giây nào. Đổi lại, chi phí lắp hai hệ thống điện và bảo trì chúng rất đắt đỏ.

Một ví dụ thực tế khác: máy bay thương mại hai động cơ được xem là có khả năng chịu lỗi, vì nếu một động cơ hỏng, máy bay vẫn có động cơ thứ hai để tiếp tục bay và hạ cánh an toàn. Ngược lại, trực thăng chỉ có một động cơ thì không thể chịu lỗi: hỏng động cơ là rớt, toàn hệ thống ngừng hoạt động. Tương tự, trong thiết kế hệ thống, ta phải quyết định đâu cần fault-tolerant tuyệt đối (như máy bay hai động cơ) và đâu chỉ cần highly available (chấp nhận ngừng ít phút). Thông thường, với các hệ thống thời gian thực đòi hỏi cao (như điều khiển không lưu, hệ thống tài chính...), yêu cầu chịu lỗi gần như tuyệt đối, còn nhiều ứng dụng web thông thường có thể chấp nhận HA với downtime vài chục giây đến vài phút khi failover.

Tóm lại: HA cho phép một tỷ lệ nhỏ downtime để đổi lấy sự đơn giản và tiết kiệm, còn FT nhằm loại bỏ hoàn toàn downtime nhưng tốn kém và phức tạp hơn nhiều. Kỹ sư hệ thống cần hiểu yêu cầu cụ thể để chọn giải pháp phù hợp, tránh "over-engineering" không cần thiết hoặc ngược lại thiết kế thiếu khi hệ thống đòi hỏi cao hơn.

Đánh giá độ tin cậy: Uptime, SLA, MTTR, MTBF

Làm thế nào để định lượng mức độ tin cậy và liên tục của một hệ thống? Dưới đây là các chỉ số quan trọng thường được dùng:

- **Uptime (độ sẵn sàng):** Thường được biểu thị bằng phần trăm thời gian hệ thống hoạt động tốt. Ví dụ, uptime 99% nghĩa là trong 100 đơn vị thời gian thì hệ thống trung bình hoạt động 99 đơn vị, còn 1 đơn vị bị downtime. Các dịch vụ đòi hỏi cao thường nói đến "9s": ví dụ 99,99% (bốn số 9) tương đương downtime chỉ ~53 phút mỗi năm. Còn 99,9% (ba số 9) cho phép downtime ~8,8 giờ mỗi năm. Nghe có vẻ nhỏ, nhưng với dịch vụ như ngân hàng hay hệ thống điều khiển máy bay thì 8 giờ ngừng trong một năm vẫn là quá nhiều! Mức uptime mong muốn phụ thuộc vào yêu cầu hệ thống: không phải lúc nào cũng phải nhắm 99,999% vì đạt càng cao chi phí xây dựng càng lớn. Thông thường, ta dựa vào cam kết của nhà cung cấp hạ tầng: các cloud provider lớn (AWS, Azure, GCP) đều có các SLA (thỏa thuận dịch vụ) đảm bảo uptime cho dịch vụ của họ, ví dụ dịch vụ lưu trữ S3 của AWS cam kết 99,99%.

- **SLA (Service Level Agreement):** Đây là thỏa thuận về mức độ dịch vụ giữa nhà cung cấp và khách hàng, thường quy định các chỉ tiêu như availability (độ sẵn sàng), hiệu năng, thời gian phản hồi sự cố... Trong bối cảnh độ tin cậy hệ thống, SLA thường nhắc tới cam kết uptime tối thiểu (ví dụ 99,9% uptime mỗi tháng). Nếu nhà cung cấp không đạt SLA (downtime vượt quá mức cam kết), khách hàng có thể được bồi thường hoặc hưởng dịch vụ miễn phí theo điều khoản. Khi thiết kế hệ thống sử dụng dịch vụ bên ngoài (như dùng database của cloud), ta cần lưu ý SLA của họ để biết giới hạn độ sẵn sàng tối đa có thể đạt. Như đã nói, các cloud provider có SLA cho dịch vụ của họ: ví dụ SLA của Azure Blob Storage hay AWS S3 Standard đều khoảng 99,99%. SLA giúp đặt kỳ vọng và là cơ sở pháp lý để đảm bảo hệ thống duy trì độ tin cậy đã hứa.

- **MTBF (Mean Time Between Failures) :** Thời gian trung bình giữa các lần hỏng: Chỉ số này đo độ ổn định của hệ thống, cho biết trung bình bao lâu thì hệ thống gặp một sự cố. MTBF cao nghĩa là các lần hỏng hóc cách xa nhau : hệ thống chạy lâu mới bị sự cố, tức là đáng tin cậy hơn. Ngược lại MTBF thấp (ví dụ vài giờ

lại sập một lần) thì rõ ràng hệ thống rất chập chờn. MTBF được tính bằng tổng thời gian hệ thống hoạt động chia cho số lần xảy ra sự cố trong giai đoạn đó. Chẳng hạn, trong một tuần (168 giờ) hệ thống bị sập 3 lần, tổng thời gian downtime là 3 giờ => uptime 165 giờ, vậy MTBF ≈ 55 giờ (trung bình ~55 giờ lại sập một lần). Lưu ý MTBF thường áp dụng cho những sự cố có thể sửa chữa/khôi phục (như lỗi phần mềm, mạng...). Còn với hỏng hóc không thể khắc phục mà phải thay thế (ví dụ cháy bo mạch, hỏng ổ cứng) thì người ta dùng chỉ số khác là MTTF (Mean Time To Failure) : thời gian trung bình tới khi hỏng hoàn toàn một thiết bị.

- **MTTR (Mean Time To Recovery/Repair) :** Thời gian khôi phục trung bình: MTTR đo thời gian trung bình để khắc phục một sự cố và đưa hệ thống trở lại hoạt động bình thường. Hiểu nôm na, mỗi lần hệ thống "sập" thì mất bao lâu để "sống lại". MTTR bao gồm từ lúc sự cố xảy ra (hoặc được phát hiện) đến khi hệ thống phục hồi hoàn toàn. Công thức tính MTTR là tổng thời gian downtime chia cho số sự cố. Ví dụ: trong tháng này có 4 sự cố, tổng thời gian downtime là 2 giờ, thì MTTR = 30 phút (0,5 giờ) : tức trung bình mỗi sự cố phải 30 phút để xử lý. MTTR càng thấp càng tốt, vì nghĩa là khi có sự cố, hệ thống phục hồi càng nhanh. Các kỹ thuật như tự động phát hiện và failover nhanh, đội ứng cứu giỏi, dự phòng phụ tùng sẵn... đều nhằm giảm MTTR. Lưu ý đôi khi tài liệu phân biệt MTTR là Mean Time to Repair (thời gian sửa lỗi thuần túy) và Mean Time to Recover (thời gian phục hồi dịch vụ). Dù định nghĩa khác nhau, mục tiêu chung vẫn là tối thiểu hóa thời gian hệ thống bị gián đoạn.

Những chỉ số trên thường đi cùng nhau: một hệ thống lý tưởng là MTBF cao (ít khi hỏng) và MTTR thấp (hỏng là phục hồi rất nhanh). Thực tế để đạt được điều đó cần chi tiết kỹ thuật tốt và quy trình phản ứng nhanh. Khi thảo luận thiết kế hệ thống, hiểu rõ các khái niệm này giúp bạn đưa ra mục tiêu cụ thể (ví dụ: "chúng tôi muốn MTTR dưới 15 phút, MTBF ít nhất 1 tháng"). Các chỉ số này cũng thường xuất hiện trong SLA : ví dụ một SLA có thể nói "MTTR tối đa 1 giờ".

Gợi ý trình bày chiến lược phục hồi sự cố trong phỏng vấn

Cuối cùng, khi bước vào phỏng vấn thiết kế hệ thống, bạn có thể sẽ được hỏi: "Anh/chị sẽ đảm bảo hệ thống của mình chịu lỗi và phục hồi ra sao?". Dưới đây là một số gợi ý để trình bày tự tin và mạch lạc trước người phỏng vấn:

- **Xác định yêu cầu độ tin cậy trước:** Hãy bắt đầu bằng việc làm rõ với người phỏng vấn về mức độ availability cần thiết của hệ thống giả định. Ví dụ: số người dùng, mức downtime chấp nhận được (vài giây, vài phút?), dữ liệu có được phép mất một chút hay không (yêu cầu consistency vs availability). Việc này cho thấy bạn hiểu rằng không phải hệ thống nào cũng cần độ chịu lỗi như nhau : yêu cầu kinh doanh quyết định kiến trúc. Nếu không được cung cấp sẵn, đừng ngại hỏi lại người phỏng vấn về các giả định này trước khi đưa ra giải pháp.

- **Nhận diện và loại bỏ single point of failure:** Giải thích rằng bạn sẽ tìm các điểm lỗi đơn lẻ trong kiến trúc và tìm cách dự phòng cho mỗi điểm đó. Ví dụ: nếu chỉ có một database chính, đó là single point of failure: bạn sẽ đề xuất dùng cơ chế replication (có một database phụ). Nếu một service quan trọng chạy trên một VM duy nhất, bạn sẽ đề xuất chạy nhiều instance service trên nhiều VM. Cho người phỏng vấn thấy bạn luôn có tư duy “nếu X hỏng thì sao?” cho mọi thành phần quan trọng.

- **Áp dụng chiến lược chịu lỗi phù hợp:** Trình bày các biện pháp cụ thể mà bạn sẽ dùng trong thiết kế. Bạn có thể cấu trúc câu trả lời theo các chiến lược đã nói ở trên, chẳng hạn: “Để đảm bảo chịu lỗi, tôi sẽ sử dụng load balancing phân tán tải giữa nhiều server web, kèm theo health check để phát hiện server hỏng. Tầng ứng dụng sẽ triển khai ít nhất 2 instance ở hai AZ khác nhau để redundancy. Dữ liệu sẽ được replicate qua cơ chế master-slave (primary-replica) cho database, hỗ trợ failover nếu node database chính gặp sự cố. Ngoài ra, tôi sẽ implement retry logic cho các yêu cầu giữa các service phòng khi network glitch, và sử dụng circuit breaker để cách ly service nào bị lỗi để không ảnh hưởng dây chuyền. Cuối cùng, hệ thống có cơ chế giám sát và cảnh báo qua CloudWatch/PagerDuty để đảm bảo đội vận hành biết ngay khi có sự cố lớn.” Cách trả lời này cho thấy bạn nắm được một loạt khái niệm và biết vận dụng chúng hài hòa.

- **Nhấn mạnh khả năng phục hồi nhanh:** Bên cạnh việc chịu lỗi, hãy nói về phục hồi: ví dụ thời gian chuyển đổi khi failover (dự kiến vài giây hay một phút?), cách tự động thay thế node hỏng (như sử dụng auto-scaling group để tạo máy mới thay máy cũ). Đề cập đến MTTR ngắn nhờ tự động hóa, hoặc nếu phù hợp, nói về kịch bản disaster recovery (phục hồi sau thảm họa lớn) nếu câu hỏi liên quan đến mất nguyên cả data center. Điều này thể hiện bạn không chỉ ngăn ngừa sự cố mà còn nghĩ đến quy trình khôi phục sau sự cố.

- **Cân bằng với chi phí và độ phức tạp:** Một điểm hay khi trả lời là thể hiện bạn nhận thức được trade-off. Ví dụ: bạn có thể nói “Để đạt mức downtime gần như

> zero thì cần chạy hệ thống active-active ở hai nơi, nhưng giải pháp này phức tạp và tốn kém. Với yêu cầu đề bài, em chọn kiến trúc highly available (đa khu vực, có dự phòng nóng) là đủ tốt, downtime vài chục giây khi failover là chấp nhận được để đổi lại đơn giản hơn." Người phỏng vấn sẽ đánh giá cao nếu bạn biết cân nhắc giữa fault tolerance và high availability tùy bối cảnh.

Cuối cùng, hãy nhớ mục đích của phỏng vấn thiết kế hệ thống không phải tìm một thiết kế hoàn hảo tuyệt đối, mà là để bạn thể hiện tư duy. Cho thấy bạn luôn nghĩ đến tính reliability và fault tolerance sẽ gây ấn tượng rằng bạn thiết kế có tâm và hiểu thực tế vận hành. Hãy sử dụng những thuật ngữ chính xác (như redundancy, failover, graceful degradation, SLA, MTTR...) kèm giải thích ngắn gọn, và nếu có thể, đưa vài ví dụ cụ thể cho dễ hình dung.

Bảo mật và phân quyền

Vai trò của bảo mật trong hệ thống phân tán hiện đại

Trong các hệ thống phân tán (microservices, cloud) hiện đại, bảo mật đóng vai trò sống còn để bảo vệ dữ liệu và tài nguyên quan trọng. Xác thực đảm bảo chỉ người dùng hoặc dịch vụ hợp lệ mới được phép truy cập tài nguyên hệ thống, giúp ngăn chặn truy cập trái phép và rò rỉ dữ liệu. Phân quyền giúp kiểm soát chính xác ai có thể làm gì trong hệ thống, từ đó tăng tính tin cậy, khả năng theo dõi và tuân thủ quy định. Nếu không thiết kế bảo mật tốt ngay từ đầu, cả hệ thống có thể dễ dàng bị tấn công hay lộ thông tin nhạy cảm. Do vậy, ta luôn phải đề cao bảo vệ nhiều lớp (multi-layer security), từ cơ sở hạ tầng (firewall, mạng) đến ứng dụng và dữ liệu trên từng thành phần.

2. Các khái niệm cơ bản

Xác thực (Authentication)

- **Đăng nhập và mật khẩu:** Phương pháp truyền thống nhất, người dùng cung cấp tên đăng nhập và mật khẩu để xác minh danh tính. Mật khẩu nên lưu dưới dạng hash an toàn (kèm salt) trên server.

- **Xác thực đa yếu tố (MFA/2FA):** Yêu cầu ít nhất hai yếu tố (chẳng hạn mật khẩu + mã OTP qua SMS hoặc ứng dụng) để tăng cường bảo mật. Ví dụ, dịch vụ ngân hàng thường yêu cầu nhập thêm mã OTP khi giao dịch quan trọng để chắc chắn chính chủ dùng tài khoản.

- **Session (phiên làm việc):** Sau khi đăng nhập thành công, server tạo một session duy nhất cho người dùng và lưu ID phiên trên server. Mỗi lần client gửi request, nó kèm session ID (thường trong cookie) để server nhận biết ai đang tương tác. Session-based Authentication cần lưu trữ thông tin phiên trên server (stateful) và có thể dễ dàng hủy (revoke) khi logout hoặc timeout.

- **JWT (JSON Web Token):** Là token tự chứa (self-contained) gồm ba phần (header, payload, signature). Sau khi đăng nhập, server trả cho client một JWT có ký số đảm bảo tính toàn vẹn. Client dùng token này để xác thực ở các request sau mà không cần server phải lưu trạng thái. JWT rất gọn, dễ truyền (trong header HTTP) và phù hợp cho hệ thống lớn, microservices, tuy nhiên cần cẩn thận: payload của

JWT có thể đọc được (chỉ được ký, không mã hóa) nên không lưu dữ liệu nhạy cảm trong đó. Ngoài ra, JWT khó hủy giữa chừng nếu bị đánh cắp (phải chờ hết hạn) nên với nhu cầu quản lý phiên linh hoạt, session-traditional có thể an toàn hơn.

Phân quyền (Authorization)

- **RBAC (Role-Based Access Control):** Hệ thống phân quyền theo vai trò. Mỗi người dùng sẽ được gán một hoặc nhiều vai trò (admin, editor, user…) và mỗi vai trò có tập quyền riêng. Ví dụ, trong hệ thống quản lý nội dung, vai trò Editor được phép chỉnh sửa bài viết, còn vai trò Viewer chỉ được xem. RBAC đơn giản, rõ ràng nhưng nếu cần chi tiết và linh hoạt hơn, có thể dễ dẫn đến "vai trò nổ tung" (role explosion).

- **ABAC (Attribute-Based Access Control):** Phân quyền dựa trên thuộc tính của người dùng, tài nguyên và môi trường. Ví dụ, có thể cho phép truy cập nếu người dùng thuộc phòng kế toán, file có độ nhạy cảm thấp, và thời gian hiện tại trong giờ hành chính. ABAC linh hoạt cao nhờ tính điều kiện (if-then rules) nhưng cũng phức tạp khi thiết lập.

- **OAuth2:** Là một chuẩn Ủy quyền (authorization protocol) phổ biến. Thay vì ứng dụng thứ ba phải lưu mật khẩu của người dùng, OAuth2 cho phép ứng dụng được ủy quyền (client) lấy một access token từ máy chủ uỷ quyền (authorization server). Access token này cho phép app truy cập tài nguyên (API) thay cho người dùng mà không cần biết password. Ví dụ, khi bạn dùng ứng dụng bên thứ ba truy cập dữ liệu Google Drive, Google cấp một token OAuth2 cho phép app đọc file mà không tiết lộ tài khoản của bạn. (Lưu ý OAuth2 chỉ cấp quyền, nếu cần xác thực nhận dạng người dùng thì thêm lớp OpenID Connect.) Các thành phần chính gồm: Resource Owner (người dùng), Client (ứng dụng), Authorization Server (phát token), Resource Server (API bảo vệ dữ liệu), và scope để giới hạn phạm vi quyền.

Bảo mật API

- **Bảo vệ API Key:** API Key là mật khẩu gọi API. Không được nhúng thẳng trong mã nguồn phía client (dễ lộ). Nên lưu trên server hoặc trong biến môi trường và đính kèm vào header khi gửi request. Cần giới hạn quyền của key (chỉ những API thực sự cần), luân chuyển key định kỳ, và xoá các key không còn dùng để giảm thiểu

tác động khi bị lộ.

- **CORS (Cross-Origin Resource Sharing):** Cơ chế cho phép hoặc chặn các request từ domain khác. Theo chính sách cùng nguồn (same-origin policy), trình duyệt chỉ cho phép trang web gọi API cùng domain. Để mở rộng (ví dụ front-end React ở app.example.com gọi API ở api.example.com), server API phải trả về header CORS cho phép domain đó. Chỉ nên cho phép những origin đáng tin cậy; nếu không cấu hình cẩn thận, kẻ tấn công có thể lợi dụng các trang độc hại để gọi API của bạn. CORS là công cụ bảo mật trình duyệt, cho phép máy chủ chỉ định các nguồn được phép truy cập tài nguyên của nó.

- **Giới hạn lưu lượng (Rate Limiting):** Xây dựng quy tắc giới hạn số request trong một khoảng thời gian từ một IP hoặc token nhất định. Rate limiting ngăn chặn các tấn công từ chối dịch vụ (DoS) hoặc brute-force, đảm bảo không có một user đơn lẻ nào làm quá tải hệ thống. Ví dụ, để chống tấn công đoán mật khẩu, ta có thể cho mỗi tài khoản tối đa 5 lần đăng nhập sai trong 10 phút, quá giới hạn sẽ khoá tạm thời.

- **Kiểm tra đầu vào (Input Validation):** Tất cả dữ liệu nhận từ client (body, params, query, file upload, v.v.) phải được kiểm tra và lọc kỹ. Mục tiêu là chỉ cho phép định dạng hợp lệ (whitelist) và loại bỏ các ký tự nguy hiểm. Điều này giúp phòng chống các lỗ hổng phổ biến như SQL Injection, XSS. Theo OWASP, input validation nên được áp dụng sớm nhất có thể trong luồng xử lý để loại bỏ dữ liệu độc trước khi hệ thống xử lý. Ví dụ, nếu một trường chỉ nhận chữ số, hãy kiểm tra chỉ chấp nhận 0-9; với input chuỗi, loại bỏ hoặc escape các ký tự < > nếu hiển thị ra HTML để tránh XSS.

Bảo vệ dữ liệu

- **Mã hóa khi truyền và lưu trữ:** Dữ liệu nhạy cảm luôn phải được mã hóa. Khi truyền qua mạng, dùng HTTPS (chạy trên TLS) để mã hóa kênh giao tiếp, chống nghe lén và chỉnh sửa thông tin. Theo khuyến cáo, mọi kết nối giữa client và server phải qua HTTPS, có thể bổ sung HTTP Strict Transport Security (HSTS) để ép buộc dùng HTTPS. Với dữ liệu tại nơi lưu trữ (database, file, backups), cần mã hóa ổ đĩa hoặc file, hoặc ít nhất mã hóa các trường quan trọng.
- **Lưu mật khẩu (hash + salt):** Không bao giờ lưu mật khẩu gốc (plaintext). Thay vào đó, dùng các thuật toán băm mật khẩu hiện đại (như Argon2, bcrypt, PBKDF2) với *salt* ngẫu nhiên mỗi user. Salt là một chuỗi ngẫu nhiên thêm vào

mật khẩu trước khi băm, giúp ngay cả khi hai người dùng có mật khẩu giống nhau thì cũng được hash khác nhau. Điều này ngăn chặn bảng tra (rainbow table) và làm tăng công sức khi hacker cố gắng bẻ mật khẩu.

- **Ngăn XSS:** XSS (Cross-Site Scripting) cho phép kẻ tấn công chèn script độc hại vào trang web và khi người dùng mở trang đó, script sẽ chạy với quyền của trang tin cậy, từ đó có thể ăn cắp cookie hoặc session token của nạn nhân. Để ngăn XSS, luôn escape hoặc encode dữ liệu người dùng khi hiển thị (trong HTML, JavaScript, CSS, SQL). Có thể sử dụng thư viện template an toàn, hoặc header CSP (Content Security Policy) để giới hạn nguồn script. Ngoài ra, đánh giá dữ liệu đầu vào (như đã nêu) cũng góp phần giảm khả năng XSS.

- **Ngăn CSRF:** CSRF (Cross-Site Request Forgery) là lỗ hổng khi một trang độc hại ép người dùng đã đăng nhập vào trang khác thực hiện các yêu cầu trái ý (chẳng hạn gửi form chuyển tiền). Do trình duyệt tự động gửi cookie phiên khi request, server sẽ không phân biệt được request giả này với request hợp lệ. Phòng CSRF thường dùng token bí mật gắn trong form và kiểm tra trên server, hoặc sử dụng cookie với thuộc tính SameSite (ngăn trình duyệt gửi cookie khi request từ site khác). Ví dụ, đặt cookie SameSite=Strict hoặc Lax để giảm khả năng cookie bị gửi theo request đến từ trang ngoài. Đồng thời, mọi action quan trọng (chuyển tiền, thay đổi thông tin cá nhân) nên yêu cầu nhập token/captcha hoặc đăng nhập lại để chắc chắn là user chủ động thực hiện.

HTTPS, TLS và bảo mật cookie

- **HTTPS/TLS:** Cần thiết cho mọi website hiện đại. HTTPS (HTTP over TLS) mã hóa toàn bộ giao tiếp, ngăn kẻ thứ ba nghe lén (MITM). Thiết lập TLS mạnh (certificate hợp lệ, ciphers mới như AES-GCM hoặc ECC) giúp bảo vệ dữ liệu cũng như đảm bảo tính toàn vẹn.

- **Cookie:** Cookie thường dùng lưu phiên hoặc token. Cần đặt các flag bảo mật:

    - Secure: Chỉ gửi cookie qua kết nối HTTPS. Nếu không có flag này, cookie có thể bị lộ qua HTTP.

    - HttpOnly: Chặn truy cập cookie từ JavaScript (vì kẻ tấn công có thể dùng XSS để lấy cookie nếu không có flag này). Nên set flag này cho cookie chứa session ID.

  - SameSite: Ngăn gửi cookie theo request từ site khác, giảm nguy cơ CSRF. Ví dụ dùng SameSite=Strict hoặc Lax tùy mức an toàn cần.

  - Tên cookie nên có prefix như __Host- hoặc __Secure- (theo MDN) để tăng tính an toàn và ngăn ghi đè bởi nguồn không an toàn. Ngoài ra, cũng cần expiration (thời gian hết hạn) hợp lý và hạn chế domain/path để giới hạn phạm vi cookie.

3. Ví dụ minh họa

- **Upload file:** Gọi ý một hệ thống cho phép người dùng upload tài liệu. Ta bắt buộc user phải **đăng nhập** (authentication) để upload. Sau khi nhận file, server kiểm tra **định dạng file** (chỉ cho phép PNG, JPG,…), giới hạn kích thước và sử dụng **thư viện quét virus** để đảm bảo không có mã độc. Tên file gốc nên được đổi thành tên sinh ngẫu nhiên để tránh lộ thông tin. File được lưu ở thư mục riêng biệt (hoặc bucket đám mây) với quyền chỉ cho phép server đọc. Cuối cùng, phân quyền (authorization) cho chỉ user đó hoặc nhóm tương ứng mới được xem/xóa file này. Ví dụ, vai trò *uploader* mới được quyền xóa file mình tạo.

- **Hệ thống quản lý người dùng (User Management):** Xây hệ thống nơi admin có thể tạo/sửa/xóa user, còn user bình thường chỉ xem thông tin cá nhân. Ta dùng RBAC: gán quyền *admin* cho nhân viên IT, quyền *user* cho khách hàng. Khi đăng nhập, nếu user thuộc nhóm admin thì mới hiển thị giao diện quản lý. Mỗi khi thay đổi mật khẩu, lưu mật khẩu mới dưới dạng hash+salt. Nếu user đăng nhập nhiều lần thất bại, tạm khoá tài khoản để tránh brute-force. Tích hợp MFA (ví dụ qua email hoặc ứng dụng OTP) cho các tài khoản nhạy cảm hoặc admin để tăng lớp bảo vệ. Mọi hành động của người dùng nên ghi log để sau này có thể **audit** (kiểm tra truy xuất nguồn gốc).

- **Dịch vụ tài chính:** Ví dụ một app ngân hàng: mọi giao dịch chuyển tiền đòi hỏi user phải đăng nhập và xác thực mạnh (mật khẩu + OTP SMS). Việc truyền dữ liệu số tiền và thông tin tài khoản đều qua HTTPS mã hóa. Dữ liệu quan trọng (thẻ, chứng minh thư, số tài khoản) được mã hóa trong cơ sở dữ liệu hoặc dùng HSM lưu trữ an toàn. Hệ thống phân quyền có thể dùng ABAC: ví dụ không cho chuyển tiền ngoài giờ giao dịch, hoặc chỉ cho giao dịch khi IP của user nằm trong vùng được phép. Giao dịch lớn có thể yêu cầu kiểm duyệt thêm bởi admin. Mọi

giao dịch được lưu lịch sử và ghi log đầy đủ để phòng ngừa gian lận.

4. Lời khuyên khi phỏng vấn về bảo mật trong System Design

- **Defense in depth (Phòng thủ nhiều lớp):** Khi trình bày, bạn nên đề cập đến việc bảo mật toàn diện ở nhiều tầng khác nhau. Ví dụ, bảo mật mạng (firewall, VPN), bảo mật API (token, TLS), bảo mật ứng dụng (Kiểm tra input, hạn chế quyền), bảo mật dữ liệu (mã hóa), và cả đào tạo người dùng (tránh phishing). Trình bày theo hướng này cho thấy bạn hiểu sâu rằng an toàn không nằm ở một biện pháp đơn lẻ mà là nhiều biện pháp phối hợp.

- **Cân nhắc hiệu năng - bảo mật:** Mã hóa và kiểm tra đầu vào đều gây thêm overhead. Ví dụ, thiết lập TLS mạnh sẽ làm tăng chi phí xử lý (TLS handshake tiêu tốn CPU đáng kể). Việc kiểm tra và băm mật khẩu nhiều lần cũng tốn thời gian. Trình bày ví dụ trade-off rõ ràng cho thấy bạn biết ưu tiên theo tình huống.

Monitoring và Observability

Trong thiết kế hệ thống, bên cạnh việc xây dựng tính năng, chúng ta cần đảm bảo hệ thống hoạt động ổn định và dễ dàng khắc phục sự cố khi xảy ra. Đây là lúc Monitoring (giám sát) và Observability (khả năng quan sát hệ thống) phát huy vai trò quan trọng. Monitoring giúp ta theo dõi các chỉ số sức khỏe của hệ thống (như CPU, bộ nhớ, độ trễ, tỷ lệ lỗi, v.v.) và cảnh báo khi có điều bất thường. Observability đi xa hơn: nó cho phép ta hiểu rõ tại sao hệ thống gặp sự cố bằng cách quan sát các dữ liệu đầu ra của hệ thống (logs, metrics, traces). Chương này sẽ giải thích chi tiết Monitoring và Observability, sự khác biệt giữa chúng, các công cụ phổ biến, "ba trụ cột" của Observability (logs, metrics, traces), kèm ví dụ thực tế và cách trình bày chủ đề này trong buổi phỏng vấn thiết kế hệ thống.

Monitoring là gì?

Monitoring (giám sát hệ thống) là quá trình thu thập, phân tích và sử dụng thông tin để theo dõi trạng thái hoạt động của hệ thống. Nói một cách đơn giản, monitoring là việc đo lường những gì ta đã biết và thiết lập cảnh báo khi có vấn đề xảy ra. Các hệ thống phần mềm thường có những chỉ số quan trọng cần được giám sát liên tục.

Ví dụ về chỉ số cần giám sát: CPU usage, dung lượng bộ nhớ (RAM), dung lượng ổ đĩa, lưu lượng network, độ trễ (latency) của request, tỷ lệ yêu cầu xử lý (throughput), tỷ lệ lỗi (error rate), v.v. Đây đều là những thông số phản ánh "sức khỏe" của hệ thống tại mỗi thời điểm. Với monitoring, ta theo dõi các chỉ số này thông qua biểu đồ trực quan và đặt ngưỡng cảnh báo.

Đặc điểm của Monitoring:

- Định sẵn chỉ số cố định: Monitoring tập trung vào những chỉ số cố định đã xác định trước, ví dụ CPU, RAM, ổ cứng, số lượng request, tỷ lệ lỗi,... Ta thường biết trước mình muốn đo lường điều gì.

- Dựa trên ngưỡng cảnh báo: Ta đặt ra ngưỡng cho các chỉ số. Khi một chỉ số vượt quá giới hạn định sẵn, hệ thống monitoring sẽ kích hoạt cảnh báo (alert). Ví dụ: nếu CPU sử dụng vượt 90% hoặc nếu tỷ lệ lỗi 5xx vượt 1%, hệ thống sẽ gửi cảnh báo.

- Trả lời câu hỏi "Điều gì đang xảy ra?": Monitoring giúp trả lời những câu hỏi cơ bản như "Hệ thống có đang hoạt động bình thường không?" và "Có vấn đề gì

đang xảy ra không?". Nó cho ta biết có hay không có vấn đề tại thời điểm hiện tại.

- Tính thụ động: Monitoring thường mang tính phản ứng: tức là nó thông báo khi sự cố đã xảy ra hoặc chỉ ra dấu hiệu bất thường. Điều này rất hữu ích để phát hiện nhanh vấn đề, nhưng hạn chế ở chỗ monitoring truyền thống không luôn cho biết nguyên nhân gốc rễ.

Ví dụ đơn giản về Monitoring: Bạn thiết lập hệ thống giám sát để theo dõi tỷ lệ lỗi HTTP 500 của dịch vụ web. Bạn quy định rằng nếu hơn 1% tổng số request trả về mã lỗi 500 trong một khoảng thời gian nhất định, thì đó là bất thường. Khi triển khai monitoring, bạn sẽ có biểu đồ hiển thị % lỗi 500 theo thời gian. Nếu đường biểu đồ vượt mức 1%, hệ thống sẽ gửi cảnh báo qua email hoặc Slack cho đội kỹ thuật. Nhờ đó, bạn sớm biết "Điều gì đang xảy ra": cụ thể là tỷ lệ lỗi đang tăng cao, và bắt đầu tiến hành xử lý.

Các công cụ Monitoring phổ biến: Ngày nay có nhiều công cụ giúp triển khai monitoring một cách hiệu quả, ví dụ:

- Prometheus: Hệ thống mã nguồn mở thu thập metrics (số liệu đo lường) và hỗ trợ cảnh báo. Prometheus rất phổ biến để lưu trữ chuỗi thời gian (time-series) các chỉ số như CPU, bộ nhớ, độ trễ, v.v., và trigger cảnh báo dựa trên ngưỡng đặt ra.

- Grafana: Nền tảng mã nguồn mở để trực quan hóa dữ liệu. Grafana thường được sử dụng cùng Prometheus (và các nguồn dữ liệu khác) để vẽ biểu đồ, dashboard hiển thị tình trạng hệ thống. Bạn có thể tạo bảng điều khiển realtime về lưu lượng người dùng, lỗi, v.v.

- Nagios/Zabbix: Các công cụ monitoring truyền thống, mạnh trong việc giám sát hạ tầng (server, network) với cảnh báo cơ bản. Chúng đã tồn tại lâu đời và thường dùng cho theo dõi tài nguyên hệ thống.

- Datadog: Dịch vụ SaaS giám sát toàn diện, tích hợp nhiều tính năng (theo dõi metrics, logs, traces trong một nền tảng). Datadog có thể thu thập dữ liệu từ nhiều nguồn và đưa ra cảnh báo thông minh, phù hợp cho doanh nghiệp muốn một giải pháp tất-cả-trong-một.

- New Relic: Nền tảng APM (Application Performance Monitoring) và quan sát hệ thống. New Relic giúp theo dõi hiệu năng ứng dụng (thời gian phản hồi, truy vấn database, v.v.) và cũng cung cấp khả năng thu thập metric, lỗi, thậm chí trace của

ứng dụng.

(Lưu ý: Prometheus và Grafana là giải pháp phổ biến trong cộng đồng mã nguồn mở; Datadog và New Relic là giải pháp thương mại được cung cấp dưới dạng dịch vụ.)

Tóm lại, Monitoring là việc giám sát các thông số đã biết của hệ thống và cảnh báo khi những thông số đó vượt giới hạn bình thường. Nó giống như việc bạn gắn các cảm biến và đồng hồ đo cho hệ thống: bất cứ khi nào nhiệt độ quá nóng hoặc áp suất quá cao, chuông báo động sẽ vang lên.

Observability là gì?

Observability (khả năng quan sát hệ thống) là khả năng hiểu được trạng thái bên trong của một hệ thống thông qua các dữ liệu mà hệ thống đó tạo ra. Nói cách khác, một hệ thống được coi là "observable" (có thể quan sát) khi chúng ta có đủ thông tin từ hệ thống (như log, metric, trace) để suy ra điều gì đang diễn ra bên trong nó. Thuật ngữ Observability bắt nguồn từ lý thuyết điều khiển (control theory), với ý nghĩa là từ các đầu ra có thể quan sát, ta suy luận ra trạng thái nội tại của hệ thống.

Mục tiêu của Observability: giúp trả lời những câu hỏi "Tại sao điều này xảy ra?" thay vì chỉ "Chuyện gì đang xảy ra?". Observability hướng đến việc cho phép bạn khám phá linh hoạt các vấn đề trong hệ thống, kể cả những vấn đề chưa từng được biết đến trước đó. Khi hệ thống có observability tốt, bạn có thể đặt ra những câu hỏi mới về hành vi của nó mà không cần phải dự đoán trước mọi loại sự cố.

Ba trụ cột chính của Observability: Để quan sát một hệ thống phức tạp, người ta thu thập ba loại dữ liệu chính: thường được gọi là "3 trụ cột của observability":

- Logs (Nhật ký): Logs là các bản ghi sự kiện trong hệ thống. Mỗi khi hệ thống thực hiện một hành động (ví dụ: người dùng đăng nhập, gọi API, lỗi xảy ra…), hệ thống có thể ghi lại một dòng log mô tả sự kiện đó. Logs thường có thông tin chi tiết như thời gian, thành phần nào log, mức độ (info, warning, error), và thông điệp lỗi hoặc ngữ cảnh sự kiện. Ví dụ: Khi xảy ra lỗi NullPointerException trong ứng dụng, log có thể ghi lại stack trace và thông tin request gây ra lỗi, giúp kỹ sư hiểu được lỗi xảy ra ở đâu trong mã. Logs cho phép ta "lần theo dấu vết" các sự kiện đã xảy ra trong quá khứ. Trong Observability, logs là dữ liệu cực kỳ quan trọng để điều tra sự cố, vì chúng cho biết chuyện gì đã diễn ra trước và trong khi lỗi xảy ra.

- Metrics (Chỉ số định lượng): Metrics là các số liệu định lượng đo đạc được theo thời gian. Thông thường metrics được lưu dưới dạng chuỗi thời gian (time-series) : tức là các cặp giá trị và thời điểm. Ví dụ về metric: số lượng request mỗi giây, thời gian đáp ứng trung bình của API, CPU usage %, memory sử dụng (MB), v.v. Metrics cho ta cái nhìn tổng quan về xu hướng và mức độ hoạt động của hệ thống. Chúng thường được biểu diễn bằng biểu đồ để dễ quan sát biến động. Ví dụ: Bạn có metric về độ trễ trung bình của dịch vụ thanh toán. Bình thường độ trễ ~100ms, nhưng bạn thấy metric này tăng đột biến lên 500ms vào lúc 10:00 sáng. Điều này báo hiệu rằng có thể có vấn đề về hiệu năng : và bạn sẽ cần tìm hiểu sâu hơn (qua log, trace) để biết nguyên nhân. Metrics thường được dùng để phát hiện bất thường nhanh chóng (như spike hoặc drop trong biểu đồ) và thiết lập cảnh báo. Tuy nhiên, metric thường chỉ cho biết rằng có vấn đề chứ không giải thích chi tiết vì sao có vấn đề.

- Traces (Dấu vết): Trace là dữ liệu theo dõi đường đi của một yêu cầu (request) xuyên qua các thành phần của hệ thống. Trong hệ thống phân tán (ví dụ kiến trúc microservices), một yêu cầu người dùng có thể đi qua nhiều service khác nhau. Distributed tracing (theo dõi phân tán) sẽ gắn một ID duy nhất cho yêu cầu đó và ghi lại các span (đoạn) tương ứng với từng dịch vụ hoặc thành phần xử lý yêu cầu. Trace cho phép ta tái hiện lại toàn bộ hành trình của một yêu cầu trong hệ thống, giống như ta xâu chuỗi các sự kiện liên quan với nhau. Ví dụ: Người dùng thực hiện giao dịch thanh toán trên ứng dụng. Yêu cầu này đi qua các service: Service A (cổng API) gọi Service B (xử lý đơn hàng), rồi Service B gọi Service C (xử lý thanh toán thẻ tín dụng). Nếu giao dịch bị lỗi hoặc chậm, trace sẽ cho ta biết bước nào gặp trục trặc. Bạn có thể phát hiện rằng thời gian xử lý ở Service C chiếm phần lớn tổng thời gian, và nguyên nhân gốc rễ là Service C bị timeout khi gọi API của bên thứ ba. Nhờ traces, ta trả lời được câu hỏi “Tại sao yêu cầu chậm/lỗi?” một cách cụ thể, đặc biệt hữu ích trong hệ thống phức tạp.

Khi kết hợp đầy đủ logs, metrics và traces, chúng ta đạt được observability tốt. Ba loại dữ liệu này bổ sung cho nhau: metrics giúp phát hiện vấn đề và đo lường mức độ ảnh hưởng, logs cung cấp ngữ cảnh chi tiết và lịch sử sự kiện, còn traces cho thấy bức tranh luồng sự kiện end-to-end để tìm nguyên nhân trong kiến trúc phân tán. Nhờ vậy, Observability cung cấp một cái nhìn toàn diện về hệ thống.

Khác biệt giữa Monitoring và Observability: Monitoring và Observability thường được nhắc chung, nhưng chúng không phải hai khái niệm đối lập mà bổ trợ cho nhau. Hiểu đơn giản:

- Monitoring cho biết điều gì đang diễn ra và có điều gì bất thường không. Còn Observability giúp hiểu tại sao điều đó diễn ra.

- Monitoring giống như hệ thống báo động khi có sự cố, còn Observability giống như bộ dụng cụ điều tra giúp tìm ra nguyên nhân sự cố đó.

- Monitoring thường dựa trên các chỉ số và kịch bản đã biết trước (ví dụ đặt ngưỡng CPU, RAM, lỗi,…), còn Observability cho phép đặt câu hỏi về những tình huống chưa từng biết (nhờ dữ liệu chi tiết như log, trace).

- Có thể nói Monitoring là một phần của Observability : bạn cần có monitoring (đo lường, cảnh báo) để hệ thống vận hành ổn, nhưng để hiểu sâu và giải quyết triệt để các vấn đề phức tạp, bạn cần observability tốt.

Một cách hình tượng, hãy hình dung Monitoring như những đèn cảnh báo trên xe hơi: nếu động cơ quá nhiệt hoặc dầu sắp cạn, đèn sẽ bật sáng để cảnh báo bạn có vấn đề. Còn Observability giống như việc bạn có thể mở nắp capo xe, cắm máy chẩn đoán vào để đọc mã lỗi và kiểm tra từng bộ phận bên trong: nhờ đó bạn hiểu rõ vì sao đèn cảnh báo bật sáng. Trong hệ thống phần mềm, monitoring cảnh báo chúng ta khi hiệu năng hoặc độ ổn định vượt khỏi giới hạn, còn observability giúp chúng ta đào sâu vào dữ liệu log, metric, trace để tìm ra gốc rễ vấn đề và hiểu rõ hành vi hệ thống trong từng hoàn cảnh.

Thực tế, trong các hệ thống hiện đại (đặc biệt là microservices và cloud), Observability ngày càng được đề cao. Observability cung cấp cái nhìn toàn diện về hành vi và hiệu suất hệ thống, giúp đội ngũ kỹ thuật nhanh chóng phát hiện, chẩn đoán và giải quyết vấn đề, đồng thời cải thiện hiệu năng vận hành. Nói cách khác, observability không chỉ phát hiện sự cố nhanh chóng mà còn hỗ trợ phân tích nguyên nhân sâu xa, điều này đặc biệt quan trọng khi hệ thống ngày càng phức tạp và có nhiều "điểm mù" nếu chỉ dùng monitoring truyền thống.

Công cụ Observability phổ biến: Để xây dựng observability, chúng ta thường kết hợp nhiều công cụ cho logs, metrics, và traces:

- ELK Stack (Elasticsearch, Logstash, Kibana): Bộ ba công cụ mã nguồn mở hỗ trợ thu thập và phân tích log. Logstash thu thập và xử lý log từ các nguồn, Elasticsearch lưu trữ và cho phép tìm kiếm mạnh mẽ, Kibana dùng để trực quan hóa (dashboard, biểu đồ) và truy vấn log. ELK stack thường được dùng để xây dựng hệ thống centralized logging (nhật ký tập trung), rất hữu ích cho

observability vì giúp bạn tìm kiếm lỗi và sự kiện xuyên suốt các dịch vụ.

- Jaeger, Zipkin: Các công cụ mã nguồn mở để triển khai distributed tracing. Chúng thu thập các trace từ hệ thống microservices và cung cấp giao diện để xem xét đường đi của từng request qua các dịch vụ. Nhờ Jaeger hoặc Zipkin, kỹ sư có thể dễ dàng thấy được service nào gây chậm trễ hoặc lỗi trong một giao dịch phức tạp.

- OpenTelemetry: Đây không hẳn là một sản phẩm riêng lẻ mà là một framework chuẩn mở để thu thập dữ liệu telemetry (bao gồm metric, log, trace) từ ứng dụng. OpenTelemetry cung cấp các thư viện cho nhiều ngôn ngữ để instrument (gắn mã theo dõi) vào ứng dụng, sau đó có thể xuất dữ liệu này sang các hệ thống như Prometheus, Jaeger, v.v. Việc nắm được OpenTelemetry giúp bạn thiết kế một hệ thống dễ quan sát ngay từ khâu viết mã.

- Honeycomb, Lightstep: Nền tảng thương mại tập trung vào observability. Các dịch vụ này cho phép lưu trữ và phân tích lượng lớn dữ liệu log/trace/metric, cung cấp truy vấn linh hoạt (thậm chí realtime), giúp tìm kiếm nguyên nhân sự cố nhanh chóng. Chúng được thiết kế cho hệ thống phân tán phức tạp, nơi việc debug thủ công trở nên khó khăn.

- Datadog, New Relic: Như đã nhắc ở phần monitoring, các nền tảng này không chỉ làm monitoring mà còn hỗ trợ observability toàn diện. Ví dụ, Datadog có tính năng APM cho phép theo dõi traces và hiển thị trực quan các dependency; New Relic cũng thu thập log và metric, cung cấp khả năng phân tích tất cả trong một giao diện.

(Ghi chú: Danh sách trên chỉ vài ví dụ tiêu biểu. Ngoài ra còn nhiều công cụ khác trong hệ sinh thái observability.)

Trình bày Monitoring và Observability trong buổi phỏng vấn

Khi phỏng vấn thiết kế hệ thống (System Design), ứng viên thường không chỉ nói về kiến trúc và tính năng, mà còn cần đề cập đến việc vận hành và giám sát hệ thống sau khi xây dựng. Nhà tuyển dụng muốn biết liệu bạn có suy nghĩ đến độ tin cậy (reliability) và

khả năng khắc phục sự cố (troubleshooting) cho hệ thống của mình hay không. Đây là nơi bạn nên khéo léo trình bày về Monitoring và Observability. Dưới đây là một số gợi ý:

- Nhấn mạnh tầm quan trọng: Hãy chủ động đề cập rằng bất kỳ hệ thống lớn nào cũng cần được giám sát và có khả năng quan sát tốt. Ví dụ: “Song song với thiết kế tính năng, em sẽ thiết kế hệ thống giám sát để đảm bảo service luôn khỏe mạnh và nhanh chóng phát hiện sự cố.” Câu nói này cho thấy bạn hiểu vận hành thực tế quan trọng như thế nào.

- Đề cập các metrics quan trọng: Trong thiết kế của bạn, hãy nêu ra một vài chỉ số chính (KPIs) mà bạn sẽ theo dõi. Ví dụ: “Đối với hệ thống này, em sẽ theo dõi các metric như số lượng yêu cầu mỗi giây, thời gian phản hồi trung bình, tỷ lệ lỗi của mỗi service, cũng như mức sử dụng CPU và bộ nhớ của các máy chủ.” Việc cụ thể hóa các metric cho thấy bạn hiểu rõ hệ thống cần đo lường gì để đảm bảo hiệu năng.

- Nói về hệ thống cảnh báo: Bạn nên trình bày rằng mình sẽ đặt ngưỡng và cấu hình cảnh báo (alerts) cho các tình huống bất thường. Chẳng hạn: “Nếu lưu lượng người dùng tăng đột biến dẫn đến CPU > 80% kéo dài hoặc tỷ lệ lỗi vượt 2%, hệ thống monitoring sẽ gửi cảnh báo qua email/SMS để đội kỹ thuật kịp thời xử lý.” Điều này chứng tỏ bạn biết cách phản ứng nhanh với sự cố.

- Giới thiệu về Observability và debug: Hãy thể hiện rằng thiết kế của bạn có tính observable : tức là dễ debug khi có vấn đề. Bạn có thể nói: “Em sẽ thêm cơ chế logging chi tiết cho các dịch vụ (như log mỗi giao dịch quan trọng, log error với stack trace đầy đủ). Đồng thời, tích hợp distributed tracing để theo dõi một request đi qua nhiều service, giúp việc xác định nút thắt (bottleneck) hoặc điểm lỗi dễ dàng hơn.” Điều này cho thấy bạn chuẩn bị cho tình huống xấu và có phương án điều tra nguyên nhân.

- Nhắc đến công cụ cụ thể (một cách tự nhiên): Nếu phù hợp, hãy đề cập một vài công cụ phổ biến mà bạn quen thuộc để gây ấn tượng. Ví dụ: “Phần metrics và alert em có thể dùng Prometheus kết hợp Grafana để lưu trữ và vẽ biểu đồ. Logs của các service em sẽ đưa về hệ thống tập trung (chẳng hạn ElasticSearch/Kibana) để dễ tìm kiếm. Còn trace thì dùng Jaeger để theo dõi request giữa các service.” Việc liệt kê công cụ cho thấy bạn có kinh nghiệm thực tế. Tuy nhiên, đừng lạm dụng tên tool nếu bạn không chắc : quan trọng hơn là mô tả được chức năng và mục đích của chúng.

- Thể hiện tư duy cân bằng: Bạn có thể nhấn mạnh rằng Monitoring và Observability đi đôi với nhau. Ví dụ: “Em hiểu rằng monitoring giúp mình biết sớm khi có sự cố, còn observability (nhờ log, metric, trace chi tiết) giúp mình điều tra nhanh nguyên nhân. Em sẽ thiết kế hệ thống đảm bảo cả hai khía cạnh này để giảm thiểu thời gian downtime.” Câu này vừa tóm tắt đúng tinh thần, vừa thể hiện bạn có cái nhìn toàn diện.

Cuối cùng, hãy nhớ trình bày một cách tự tin nhưng khiêm tốn. Nhà tuyển dụng không nhất thiết đòi hỏi bạn phải thuộc tên mọi công cụ, nhưng họ muốn thấy bạn ý thức được tầm quan trọng của việc giám sát và khả năng quan sát hệ thống. Bằng cách đưa Monitoring và Observability vào bài thiết kế của mình một cách hợp lý, bạn sẽ ghi điểm rằng mình nghĩ xa hơn vòng đời phát triển (development) : tức là quan tâm đến vận hành (operations) và độ tin cậy (reliability) của hệ thống về lâu dài.

Lời kết

Bạn đã đọc hết cuốn “Hướng dẫn tiếp cận phỏng vấn system design” của mình.
Từ đáy lòng, mình cám ơn bạn rất nhiều vì đã đọc đến cuối quyển Ebook này. Có thể cách viết mình còn lộn xộn, mình mong bạn hiểu những tâm huyết của mình, và những mong mỏi được chia sẻ kiến thức đến cộng đồng. Thiết kế ứng dụng là điều rất thú vị, và mình mong muốn mọi người đều có những giải pháp tốt nhất cho ứng dụng của mình. Mình hy vọng ebook này sẽ giúp đỡ bạn phần nào trên con đường nâng cấp và cải thiện kỹ năng, trở thành một kiến trúc sư ứng dụng đúng nghĩa.
Cám ơn bạn rất nhiều !
Nếu bạn thích những gì trong cuốn sách này và muốn trao đổi thêm, đừng ngần ngại liên lạc với mình qua:
Email: huynt57@gmail.com
Facebook: Tai đây

Cuốn Ebook này sẽ không tồn tại nếu không có vợ mình Dương Thị Thu Huyền và con trai mình, cố vấn tí hon Nguyễn Dương Hoàng Khôi. Cám ơn gia đình đã luôn bên cạnh và ủng hộ mình.

Happy Coding !