XÂY DỰNG HỆ THỐNG CÓ KHẢ NĂNG MỞ RỘNG VẬT LÝ

Mục đích tài liệu

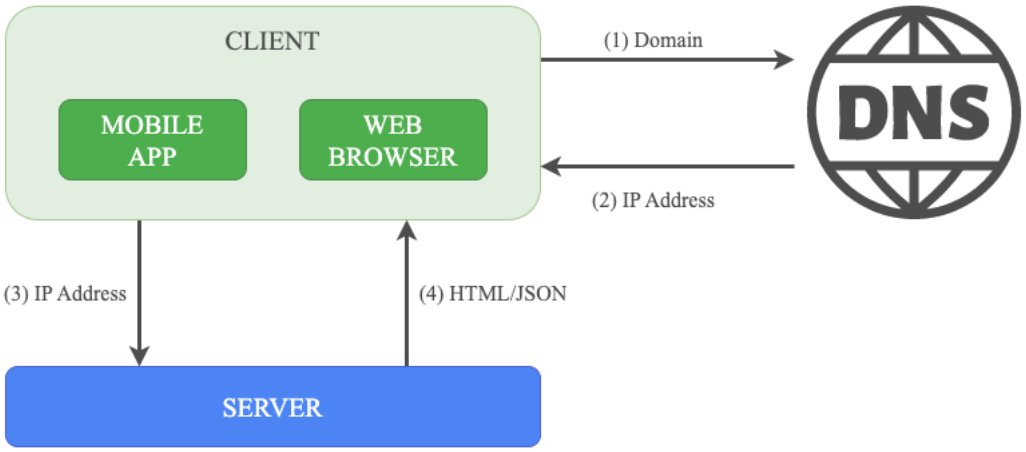
Thiết kế một hệ thống có khả năng mở rộng, thực chất là xây dựng một hệ thống cho phép chịu tải lớn. Để xây dựng nên hệ thống như vậy, không phải là logic thuật toán mà cái chính ở hàng loạt các vấn đề liên quan. Tài liệu này cung cấp cái nhìn tổng quát nhất về tất cả các phần cần quan tâm của hệ thống.

Lưu ý: khi nói về khả năng scale nghĩa là tính đến cả khả năng mở rộng về tính năng mới lẫn khả năng mở rộng về vật lý (số request/s, latency, …). Tài liệu này chỉ đề cập đến khả năng mở rộng vật lý.

Mục lục

Mô hình cổ điển

Mọi hệ thống lớn đều phát triển từ mô hình đơn giản với việc mọi thứ (execute, cache, database, …) đều được trên một máy chủ duy nhất.

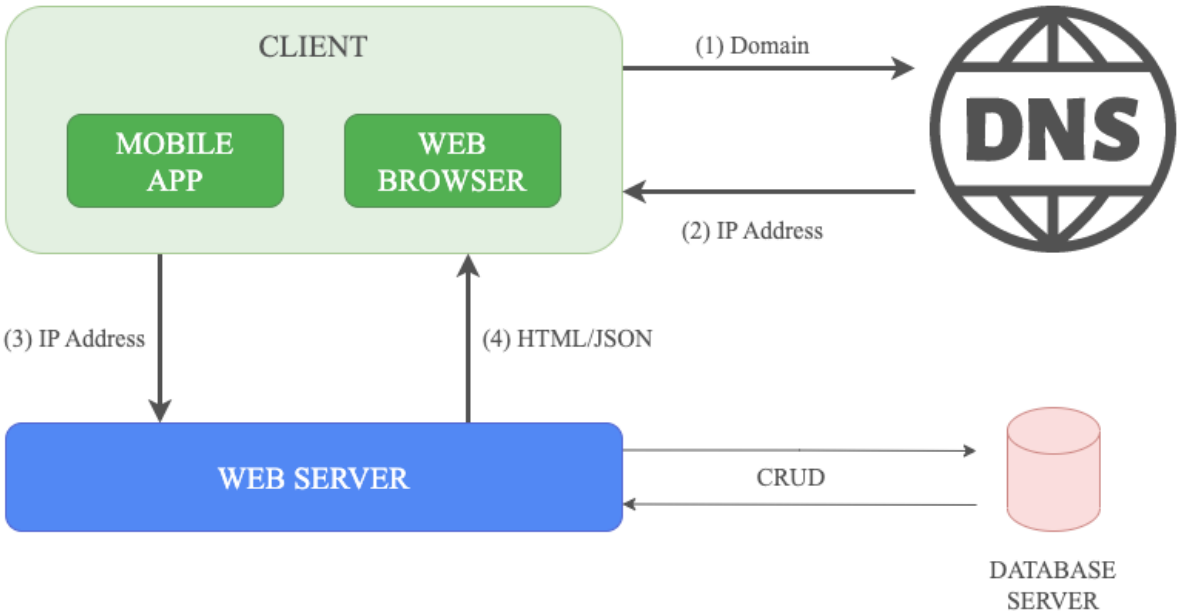


Flow traffic cơ bản như sau:

Bước	Diễn giải
1	Client gửi domain đến DNS
2	DNS phân giải domain thành địa chỉ IP và trả về cho Client
3	Client sử dụng địa chỉ IP kết nối trực tiếp đến server
4	Server trả về dữ liệu cho Client HTML(Web Browser) hoặc JSON(MobileApp)

Database

Tuy nhiên, khi số request tăng lên, một server không đủ đáp ứng. Lúc này, ta tách database ra riêng. Hệ thống cho hai server: Web Server (web tier) và Database Server (data tier). Việc chia server này giúp scale phần logic và database độc lập với nhau.



Có hai loại sở dữ liệu, cơ sở dữ liệu có có quan hệ (SQL) và phi quan hệ (noSQL).
Cơ sở dữ liệu có quan hệ (RDBMS) lưu trữ dữ liệu trong các bảng (table) và dòng (row). Theo đó, dữ liệu giữa các bảng khác nhau có thể có quan hệ phụ thuộc tồn tại lẫn nhau. Ta có thể truy vấn bằng cách kết bảng (join table). Ví dụ: mySQL, Postgres, Microsoft SQL Server, …
Cơ sở dữ liệu phi quan hệ lại rất đa dạng chủng loại: key-value (Amazon DynamoDB, Couchbase), graph (Neo4j), column (Cassandra), document (MongoDB).
Đối với phần lớn nhu cầu, ta sử dụng cơ sở dữ liệu có quan hệ. Những trường hợp cân nhắc dùng cơ sở dữ liệu phi quan hệ:

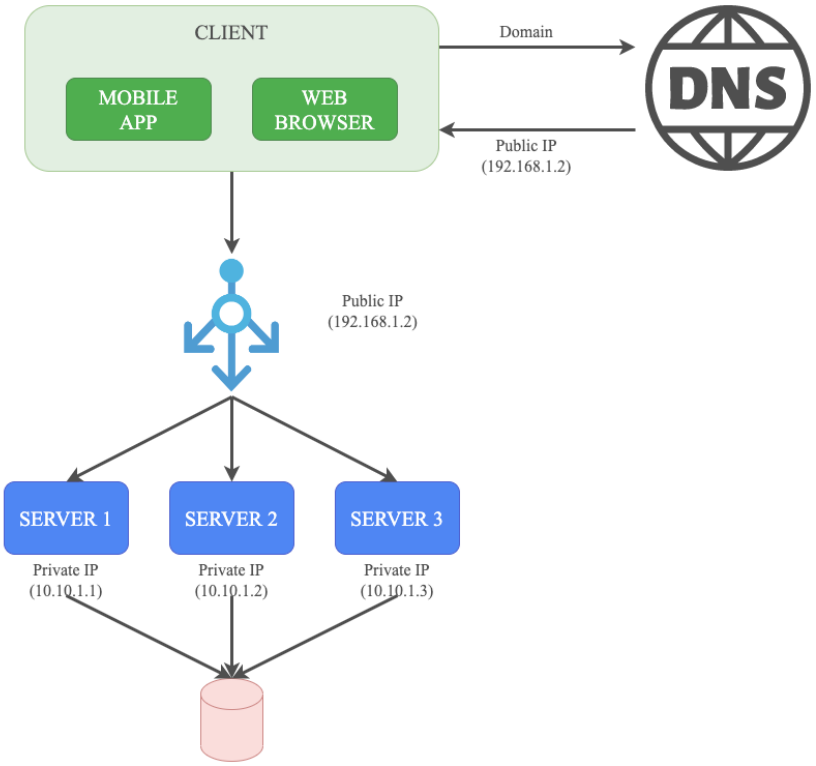
- Ứng dụng của bạn yêu cầu độ trễ cực thấp.
- Dữ liệu không có cấu trúc.
- Dữ liệu không có yêu cầu ràng buộc tính toàn vẹn.
- Dữ liệu chỉ có duy nhất nhu cầu serialize và deserialize với dữ liệu: JSON, XML, YAML, ..
- Dữ liệu có khối lượng lớn.

Vertical scaling và Horizontal scaling

Vertical scaling là mở rộng bằng cách tăng cường thiết bị phần cứng: thêm RAM, CPU, tăng băng thông. Đối với các trường hợp traffic bị nghẽn, đây là cách đơn giản và hiệu quả nhất. Nhưng bản thân cách này bị giới hạn do các giới hạn của phần cứng, vì RAM, CPU của server có giới hạn.
Horizontal scaling là cách tăng số instance, triển khai trên nhiều server khác nhau. Cụ thể, với mô hình cổ điển, client kết nối trực tiếp với server. Điều này cũng có nghĩa, chỉ duy nhất một instance được triển khai. Nếu instance đó có vấn đề, hệ thống sẽ dừng ngay lập tức. Hoặc nếu có nhiều request từ client, server sẽ bắt đầu quá tải và chậm dần. Lúc này, ta cần sử dụng đến một load balancer.

Load balancer

Khi hệ thống sử dụng load balancer, địa chỉ IP đăng ký với CDN không phải của server thật sự mà chỉ là địa chỉ công cộng của load balancer (LB). Khi client gửi request đến địa chỉ trên, thực chất là gửi đến LB.
Bản thân LB nắm giữa một tập các địa chỉ nội bộ (private IP) của các server thật sự. Request sẽ được phân phối đều giữa các server. Sau khi nhận được response từ server, LB sẽ trả về cho client.



Việc sử dụng LB có hai ba điểm:

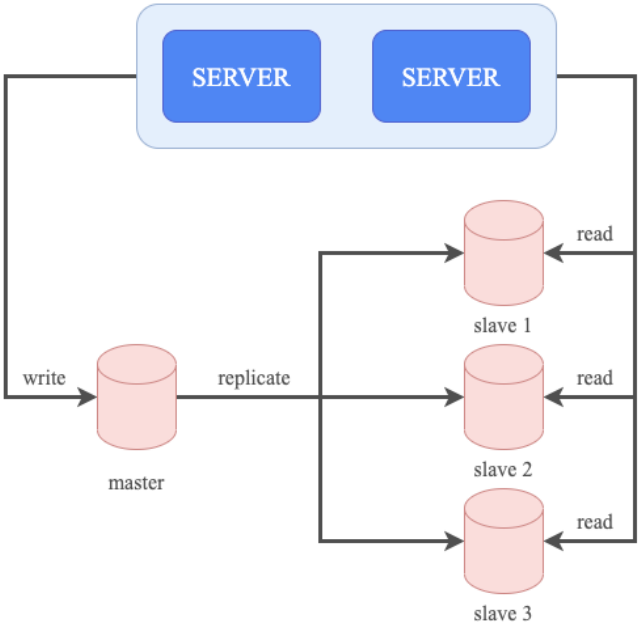
- Do LB phân phối đều giữa các server, nếu một trong số server gặp sự cố, request sẽ được chuyển cho một trong các server còn lại. Hệ thống vẫn hoạt động bình thường.

- Trường hợp số server hiện tại vẫn không đủ đáp ứng nhu cầu, ta chỉ việc thêm server và đăng kí private IP mới với LB.
- Dùng LB cũng là cách che giấu địa chỉ IP nội bộ của các server thật sự, tránh các rủi ro về bảo mật.

Tuy nhiên, lúc này các Web Server đều sử dụng chung một Database Server (data tier). Hệ quả, cho dù web tier đủ khả năng, nhưng traffic vẫn có khả năng nghẽn ở data tier.
Tương tự cách horizontal scaling của web tier, data tier cũng có phương pháp được gọi là database replication.

Database Replication

Database Replication (DR) thực chất là một hệ thống gồm một server master chuyên làm nhiệm vụ cho ghi (create, update, delete) và các server slave chuyên làm nhiệm vụ cho đọc (select). Thông tin từ server sẽ được đồng bộ về cho các slave theo định kỳ. Thông thường, số tác vụ read sẽ lớn hơn rất nhiều so với write nên chỉ cần duy nhất một master, khi cần scale, ta thêm mới slave.



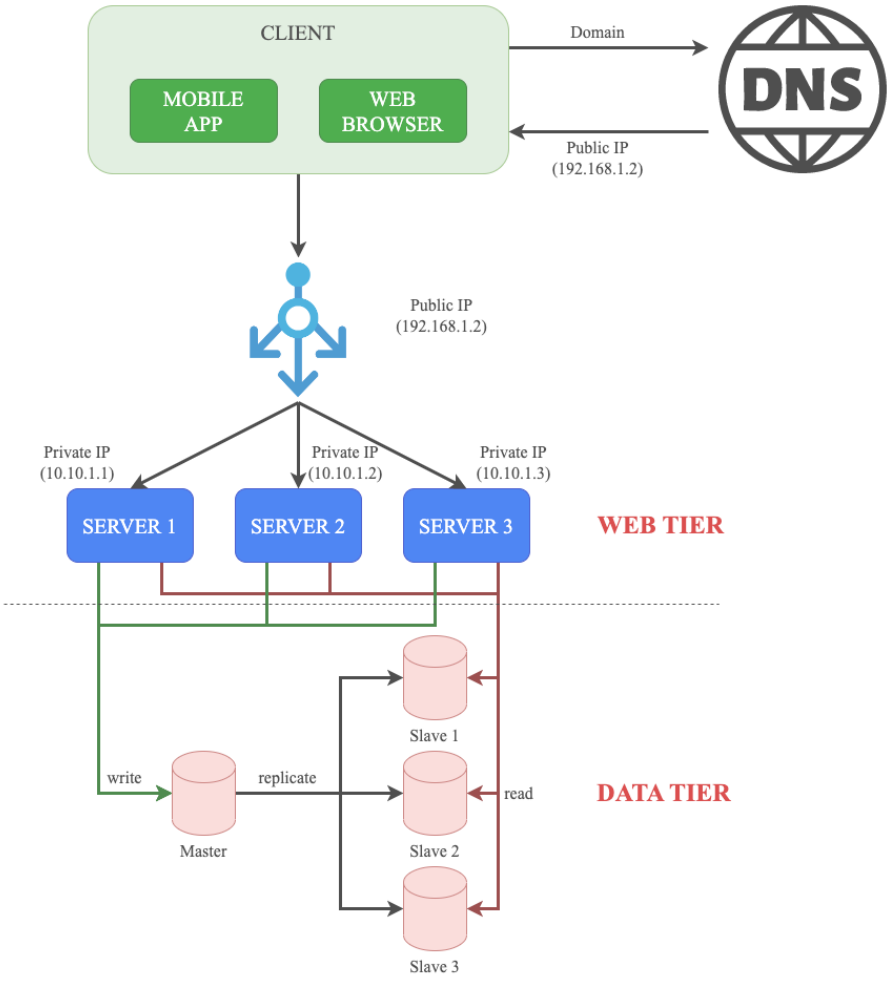
Ưu điểm của mô hình master-slave:

- Cải thiện chất lượng: Do có nhiều slave nên cho phép việc query diễn ra song song giữa các web server.
- Tăng tính tin cậy dữ liệu: khi một trong các server gặp sự cố, dữ liệu vẫn được bảo toàn nhờ đã được replicate giống nhau ở các slave.
- Tăng tính sẵn sàng của hệ thống: nếu một server gặp sự cố, hệ thống vẫn hoạt động bình thường với các server còn lại.

Cơ chế hoạt động của mô hình:

- Nếu một slave gặp sự cố, hệ thống vẫn hoạt động bình thường, ta chỉ việc thay slave vật lý mới.
- Nếu master gặp sự cố, một slave sẽ tự động được đưa lên thành master. Tuy nhiên, do master là nơi duy nhất để ghi nên có khả năng một phần dữ liệu sẽ bị mất. Để khắc phục vấn đề này, ta có thể dùng mô hình multi-masters (nhiều master trong một hệ thống) hoặc circular replication (mỗi server đều có khả năng write và read). Tuy nhiên, hệ thống như vậy rất phức tạp, kéo theo các vấn đề liên quan đến việc dữ liệu không bộ giữa các server.

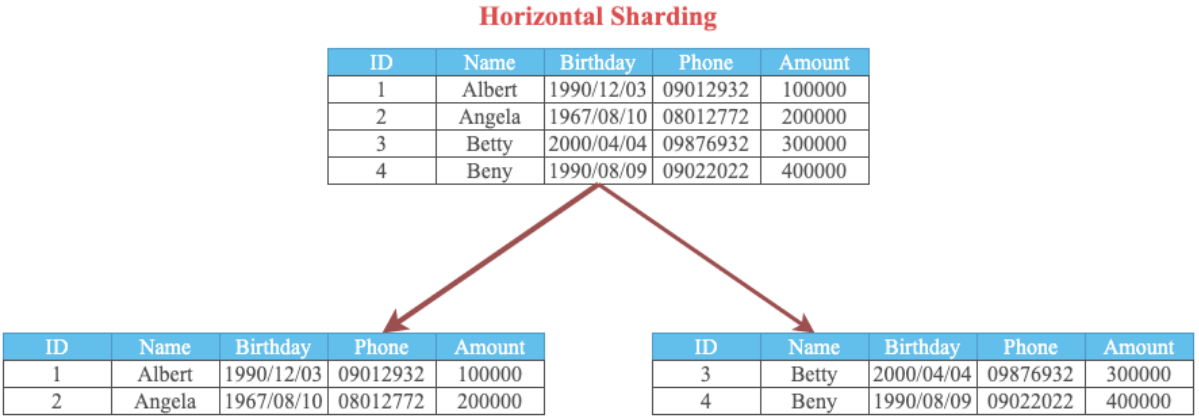
Kết hợp load balancer và database replication, ta có đã một hệ thống có khả năng scale tốt.

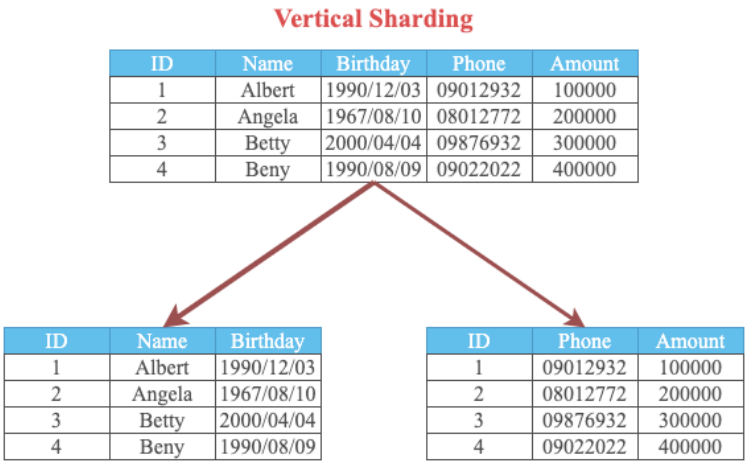


Tuy nhiên, hệ thống này vẫn còn truy cập trực tiếp vào database, nghĩa là vẫn đang thao tác với ổ cứng. Đây là hạn chế của hệ thống, lúc này ta đứng trước hai lựa chọn. Hoặc là đầu tư rất tốn kém cho ổ cứng loại tốt nhất để tăng tốc độ. Hoặc là thêm cache trước database.

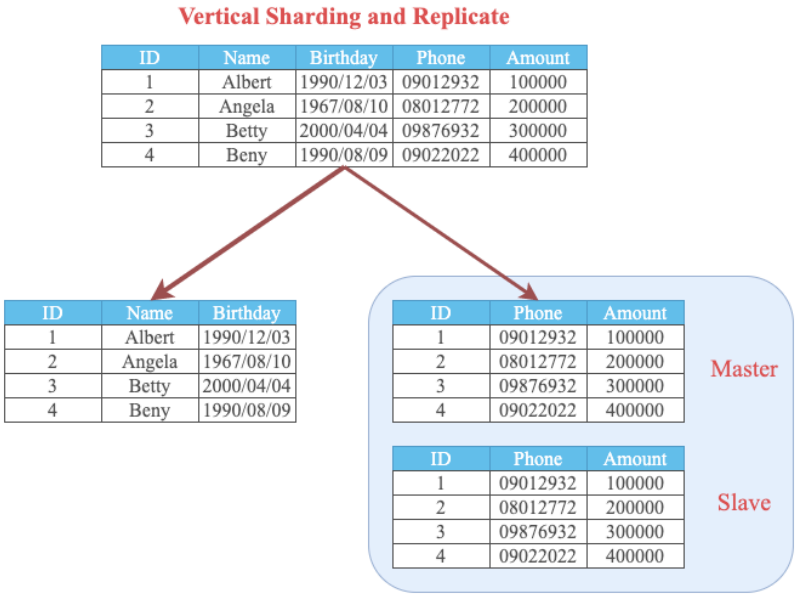
Database Sharding

Khi hệ thống chạy lâu ngày, dữ liệu sẽ càng lúc càng mở rộng, chi phí cho việc truy vấn sẽ càng lúc càng lớn dần. Các nghiệp vụ mới cũng phát sinh, khiến thông tin của một đối tượng cũng nhiều hơn. Nếu cứ để dữ liệu lớn dần và số cột tăng dần thì database sẽ thành thắt cổ chai của hệ thống. Lúc này ta cần chia nhỏ database.
Có hai phương pháp:
Theo chiều ngang (**Horizontal Sharding**) giữ nguyên cấu trúc bảng nhưng các tập dữ liệu riêng sẽ được chia vào các bảng con.
Theo chiều ngang (**Vertical Sharding**), một bảng lớn sẽ được chia làm nhiều bảng con, mỗi bảng con sẽ là một tập con các cột của bảng gốc.





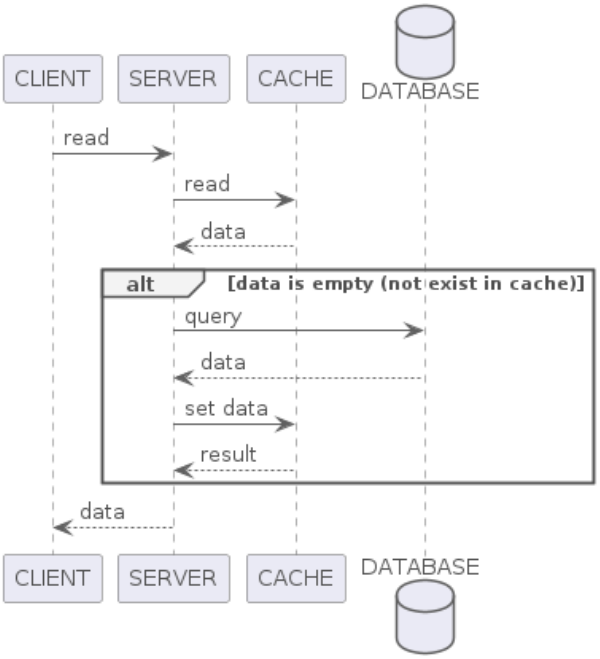
Sharding tốt cũng là cách tối ưu hoá chi phí cho replicate.



Tuy nhiên, với việc chia nhỏ, ta cũng đối mặt với bài toán khó khăn hơn khi cần liên kết dữ liệu lại với nhau.

Cache

Cache đứng trước storage, có tốc độ đọc lớn hơn storage rất nhiều lần (thường là RAM). Mỗi khi cần read dữ liệu, server sẽ ưu tiên read từ cache trước. Trường hợp không có dữ liệu trên cache, server sẽ query trên storage, write vào cache trước, sau đó mới trả dữ liệu cho client. Lần sau khi client read, thông tin đã có sẵn trên cache.



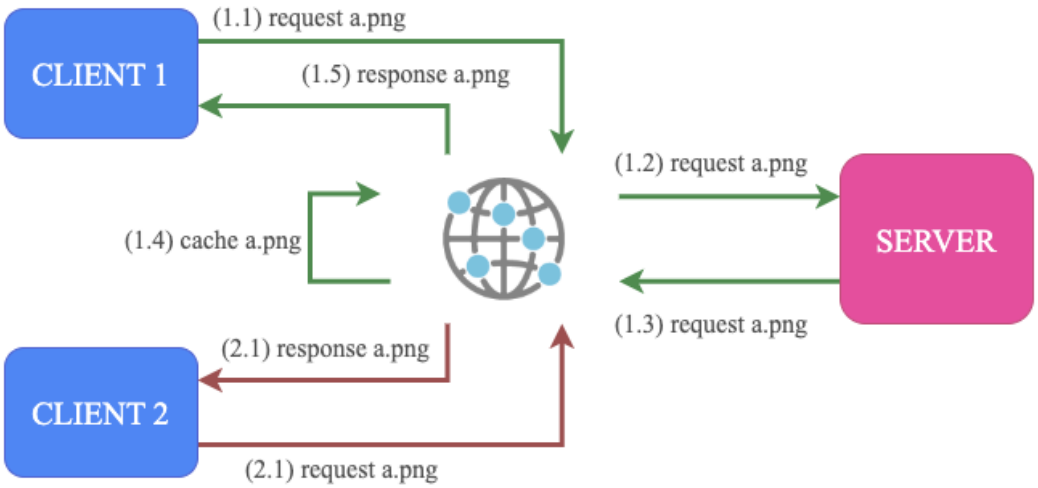
Cache tăng tốc rất lớn cho tác vụ read. Tuy nhiên, cần cân nhắc kĩ các vấn đề sau trước khi quyết định dùng cache.

- Hầu hết cache đều sử dụng RAM để lưu trữ, do đó nếu server khởi động lại thì thông tin sẽ không còn. Do đó, các thông tin quan trọng, cần lưu trữ lâu dài nhất định phải lưu trước trên storage. Cũng có một số cache cung cấp cơ chế cho phép lưu trữ lâu dài nhưng đó không phải ưu tiên của bất cứ hệ thống cache nào.
- Thận trọng khi thiết lập thời gian tồn tại của giá trị trong cache (expire time). Bản chất cache là bộ nhớ tạm, và được lưu trữ trên RAM. Do đó, bắt buộc phải thiết lập expire time cho dữ liệu, nếu không RAM sẽ bị đầy. Tuy nhiên, cần cân nhắc thời gian bao nhiêu là đủ, nếu quá ngắn thì số lần query data storage sẽ nhiều, cache không có ý nghĩa, nếu quá dài thì có thể hao phí RAM không cần thiết.
- Đảm bảo thông tin: nếu dữ liệu đã thay đổi ở storage nhưng chưa được cập nhật trên cache sẽ rất nguy hiểm. Do đó, nhất định phải xóa hoặc làm mới dữ liệu trên cache.
- Cache có khả năng trở thành single point of failure (SPOF - khi thực thế đó gặp sự cố, cả hệ thống dừng hoạt động), do mọi request sẽ đi qua đó. Có hai cách để phòng tránh, hoặc là sử dụng multiple cache server, và luôn dự trù phần trăm bộ đệm để đề phòng trường hợp hết resource (RAM).
- Luôn thiết lập cơ chế tự bảo vệ khi đầy bộ đệm. Cụ thể, các cache engine mới nhất luôn có cơ chế loại bỏ bớt dữ liệu cũ trong trường hợp bộ đệm không còn trống, nhằm có chỗ cho dữ liệu mới. LRU(Least recently used), MRU(Most Recently Used), LFU(Least Frequently Used), FIFO(First in First Out).

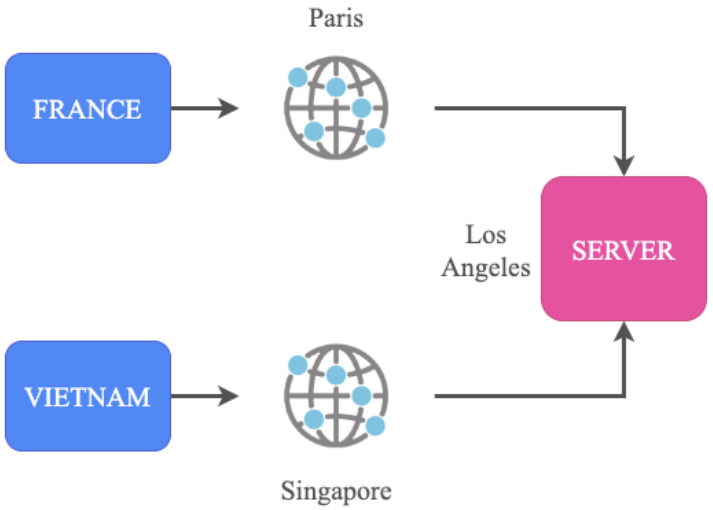
Sau khi có thêm cache, ta đã tối ưu được các request dạng API. Nhưng không có tác dụng với các tài nguyên tĩnh (css, image, file, …) Trong trường hợp này, ta cần quan tâm đến CDN.

Content delivery network (CDN)

Tương tự như việc sử dụng cache hỗ trợ cho data storage trong các API, các tài nguyên tĩnh của hệ thống (video, image, css, …) cũng được CDN hỗ trợ để hạn chế truy cập vào server lưu trữ tài nguyên gốc. CDN bản chất là một server riêng biệt, thuộc tầng network, hoàn toàn trong suốt với người dùng và server lưu trữ gốc.



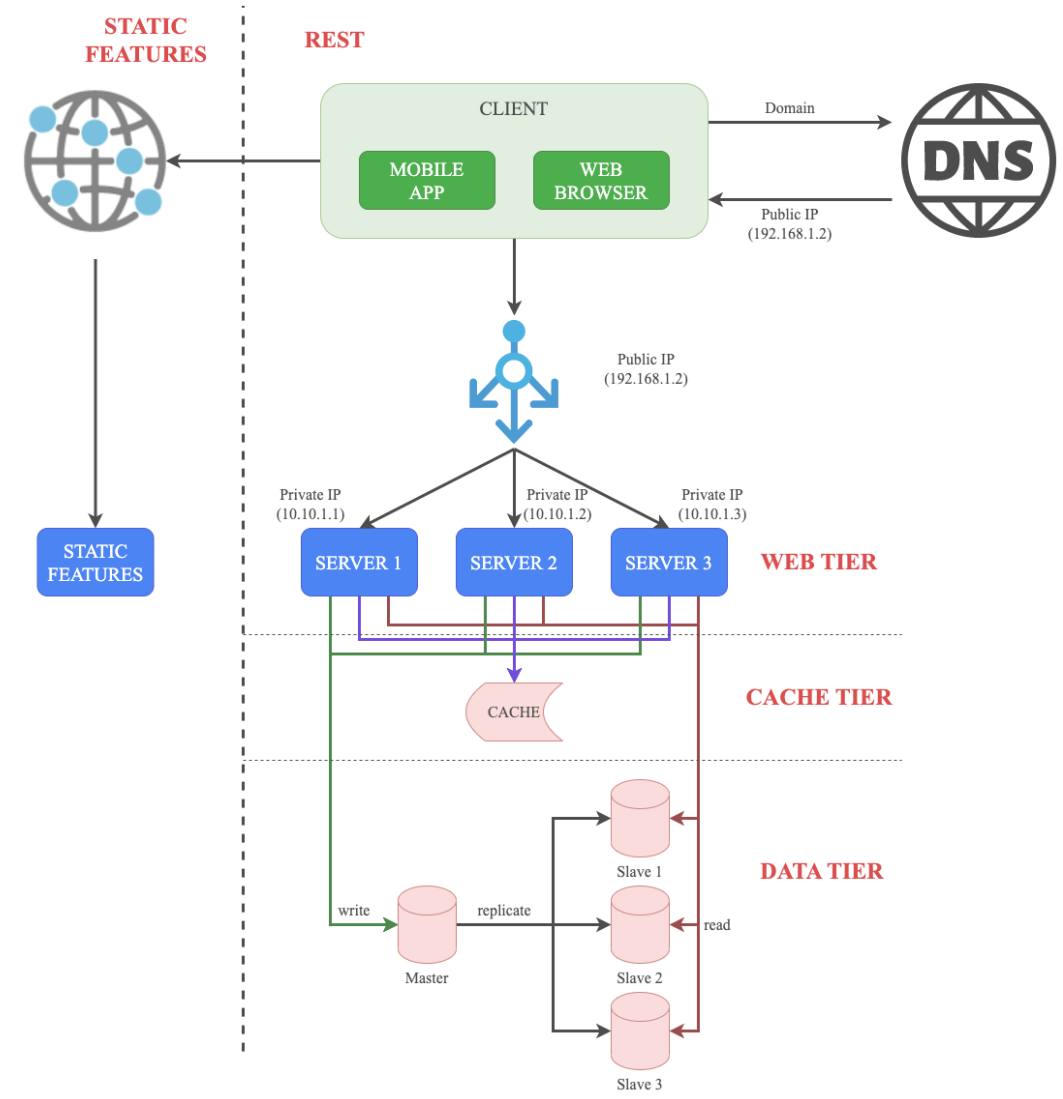
Trong ví dụ trên, client 1 và 2 không truy cập trực tiếp server lưu trữ mà thông qua server CDN. Trong lần truy cập đầu tiên của client 1 đến file a.png, server CDN không có dữ liệu về a.png nên truy cập nội dung file từ server. Sau khi cache lại, CDN sẽ trả nội dung a.png về cho client 1. Trong lần tiếp theo, client 2 cũng truy cập đến file trên, CDN đã có sẵn nên trả về ngay.
Việc sử dụng CDN không chỉ giúp giảm tải cho server gốc mà còn ý nghĩa lớn khi rút ngắn traffic truy cập đến file.



Theo ví dụ trên, trong trường hợp CDN đã cache được file, các user tại Pháp chỉ cần truy cập đến CDN tại Paris, còn tại Việt Nam thì chỉ cần truy cập đến Singapore là đã

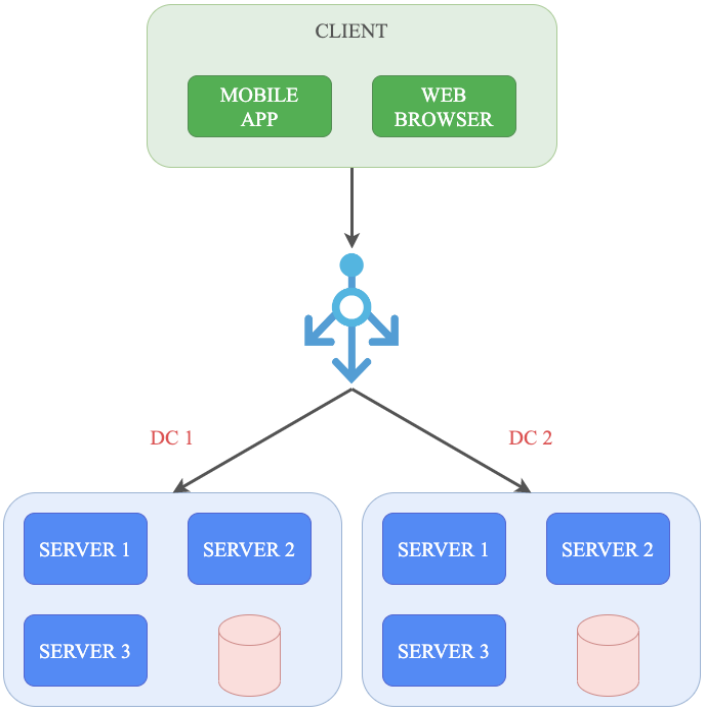
có được nội dung file trả về, không cần truy cập tận Los Angeles. Điều này đặc biệt có ý nghĩa đối với các ứng dụng độ phủ sóng lớn.
Khi dùng CDN cũng cần lưu ý:

- Bản thân CDN là dịch vụ của bên thứ ba, mọi lưu trữ đều tính phí. Vì vậy, cần tính thận những file thật sự cần đến CDN.
- Tương tự cache, CDN cũng có expire time cho từng tài nguyên, cần tính toán thời gian cho hợp lý, quá ngắn thì CDN không có nhiều tác dụng, quá dài thì tốn chi phí không cần thiết.
- CDN cũng không phải ổn định mãi mãi, do đó, client cần có phương án dự phòng cho trường hợp CDN bị lỗi.
- Trường hợp một file bất kỳ chưa hết thời hạn nhưng bị lỗi và cần xóa khỏi CDN, ta có thể dùng API của nhà cung cấp, hoặc đơn giản hơn là thay bằng version (a.png?v=2).



Data center

Đối với các ứng dụng có phạm vi phủ sóng lớn, việc đặt tất cả service ở một trung tâm duy nhất là rủi ro. Mặc dù bản thân các trung tâm dữ liệu luôn có quy trình chặt chẽ đảm bảo hệ thống ổn định nhất nhưng các sự cố không phải hiếm. Ví dụ: Zalo sập hệ thống do Data Center của Quang Trung hư nguồn điện (2018), Kakao sập hệ thống do Data Center bị chập điện (2022).
Do đó, việc triển khai sản phẩm trên nhiều Data Center khác nhau là yêu cầu bắt buộc với các ứng dụng lớn.
Điều này cũng giúp cho user có thể truy cập tốt hơn. Hệ thống sẽ tuỳ vào vùng địa lý mà hướng user đến Data Center gần nhất.

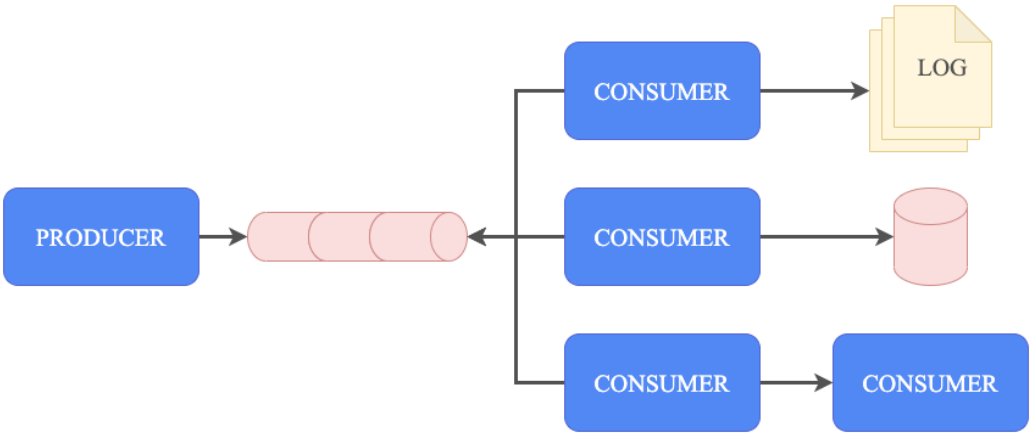


Tuy vậy, việc triển khai nhiều Data Center sẽ kéo theo các vấn đề khó khăn:

- Lựa chọn balancer và thuật toán chia vùng vật lý tốt là đều cần quan tâm đầu tiên. Thực ra cũng có nhiều ứng dụng cho phép làm việc này, phổ biến nhất là GeoDNS (DNS cũng là một dạng balancer).
- Vấn đề đồng bộ dữ liệu giữa các Data Center với nhau là rất khó. Do đó, thông thường, vấn đề xây dựng Data Cluster luôn được triển khai chung với Data Center.
- Fix bug, đồng bộ source code mỗi lần hệ thống có cập nhật rất tốn kém và rủi ro.

Message Queue

Khi các phần đã scale hết cỡ nhưng vẫn không thể đáp ứng được nhu cầu sử lý đồng bộ tại các thời điểm peak, chúng ta bắt buộc phải chuyển qua xử lý bất đồng bộ. Lúc này phải sử dụng đến các Message Queue (MQ).
Việc dùng MQ cũng là cách tách các hệ thống ra riêng biệt với nhau. Từ đó update/scale độc lập mà không ảnh hưởng lẫn nhau.



Nguyên tắc chung, một producer/publisher sẽ gửi message vào MQ, một hoặc nhiều consumer/publisher sẽ nhận message và thực hiện nghiệp vụ của riêng mình.
Mô hình cho phép scale hệ thống tốt, bởi lẽ, các service hoàn toàn không cần kết nối trực tiếp với nhau. Mỗi service phụ thuộc vào logic và khả năng chịu tải của mình mà có kiến trúc và chiến lược riêng.
Nhưng bản thân kiến trúc này cũng có nhược điểm:

- Request không được xử lý tức thời.
- Có khả năng consumer nhận một message hai lần và xử lý một tác vụ hai lần. Do đó, giải pháp chống trùng là cần thiết (thông thường mỗi message sẽ có một key để consumer phân biệt).