SE 464
Week 4

Basics of System Design: Message Queues, Load Balancing

Message Queues

The following slides were taken with permission from Professor Steve Tarzia of Northwestern University.
https://stevetarzia.com/teaching/310/slides/Scalability%20Lecture%2016%20-%20Asynchronous%20Processing.pdf

Recap – Choosing a data store

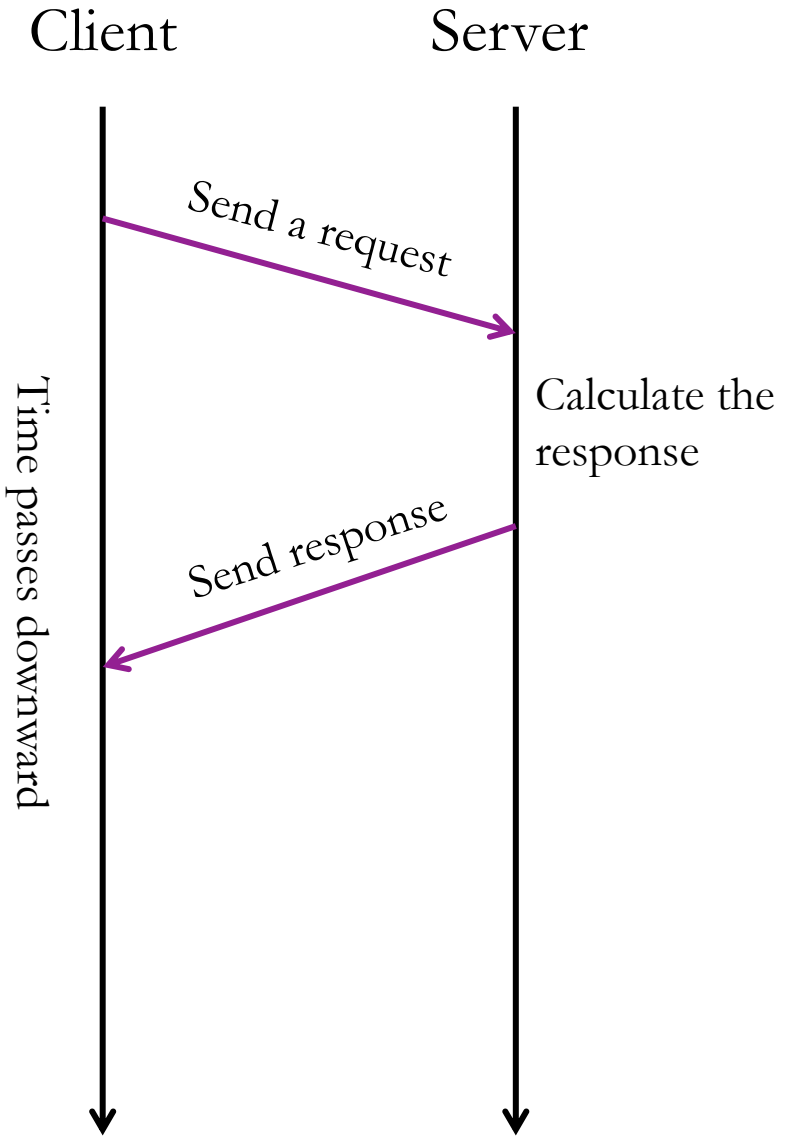
Data store	Examples	Data abstraction	
SQL Relational DB	MySQL, Oracle	Tables, rows, columns	Highly structured
Column-oriented DB	Snowflake, BigQuery	Tables, rows, columns	
Search engine	Elastic search	JSON, text	Semi-structured
Document store	MongoDB	Key → JSON	
Distributed cache	Redis	Key → value (lists, sets, etc.)	
NoSQL DB	Cassandra, Dynamo	2D Key-value (pseudo-cols)	
Cloud object store	S3, Azure Blobs	K-V / Filename-contents	Files with data "blobs"
Cluster filesystem	Hadoop dist. fs.	K-V / Filename-contents	
Networked filesystem	NFS, EFS, EBS	Filename-contents	

Your choice depends mainly on the **structure** of data and pattern of **access**.

- **Transactions** are easy on SQL DBs, available but slow on some NoSQL DBs

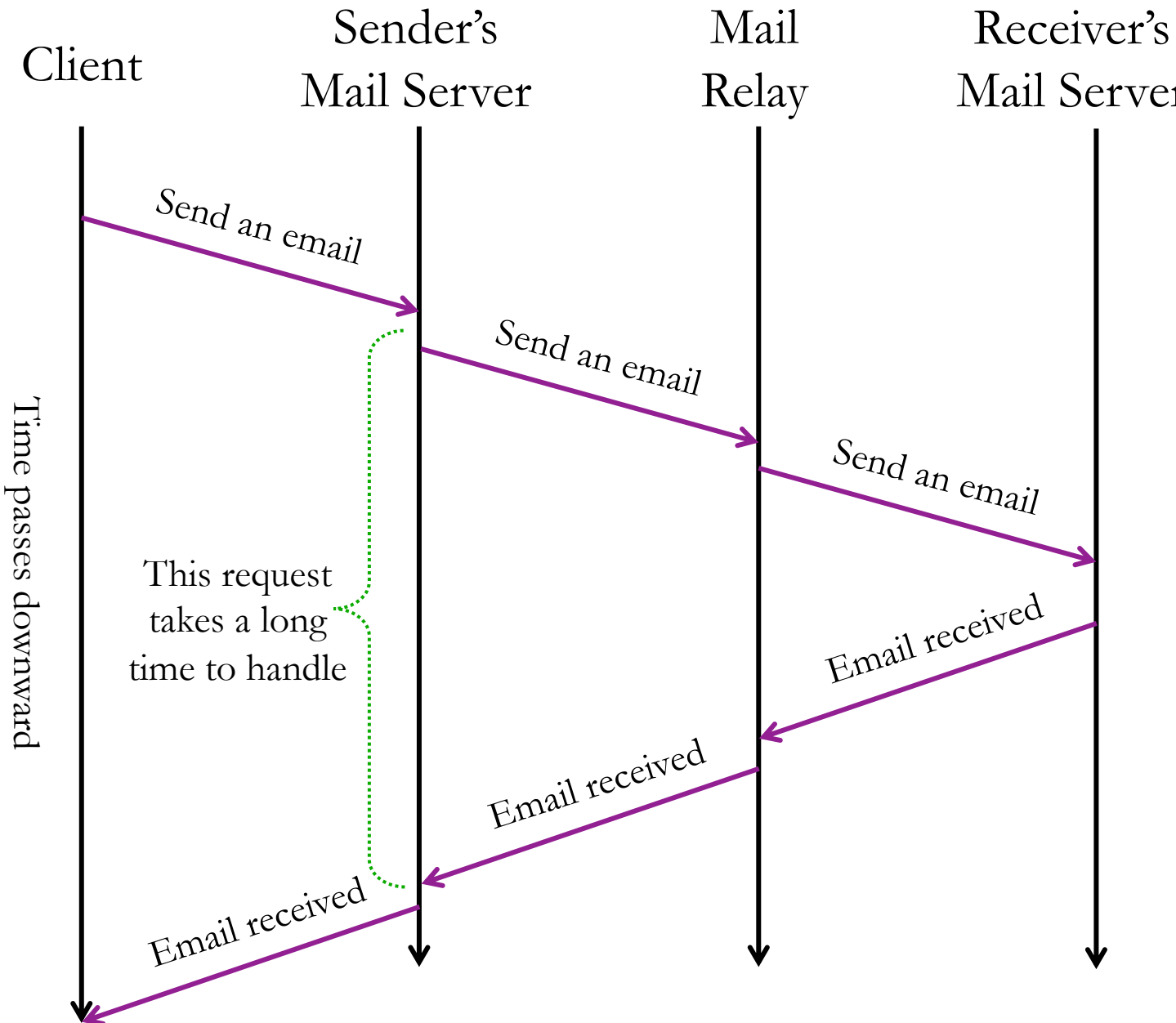
Ways to couple services

- **Responses** are important because:
  - Responses **acknowledge** that the request was received.
  - Responses may contain **data** that the client needs.
  - Responses may indicate a **failure** which the client must somehow react to (perhaps by retrying).
- So far, we have examined **synchronous** APIs.
  - The client waits for the server to finish processing, hence the two are *synchronized*.
  - Also called a **blocking** request.
  - This follows the pattern of HTTP and REST, which fetch documents/data.



But what if the request is really long?

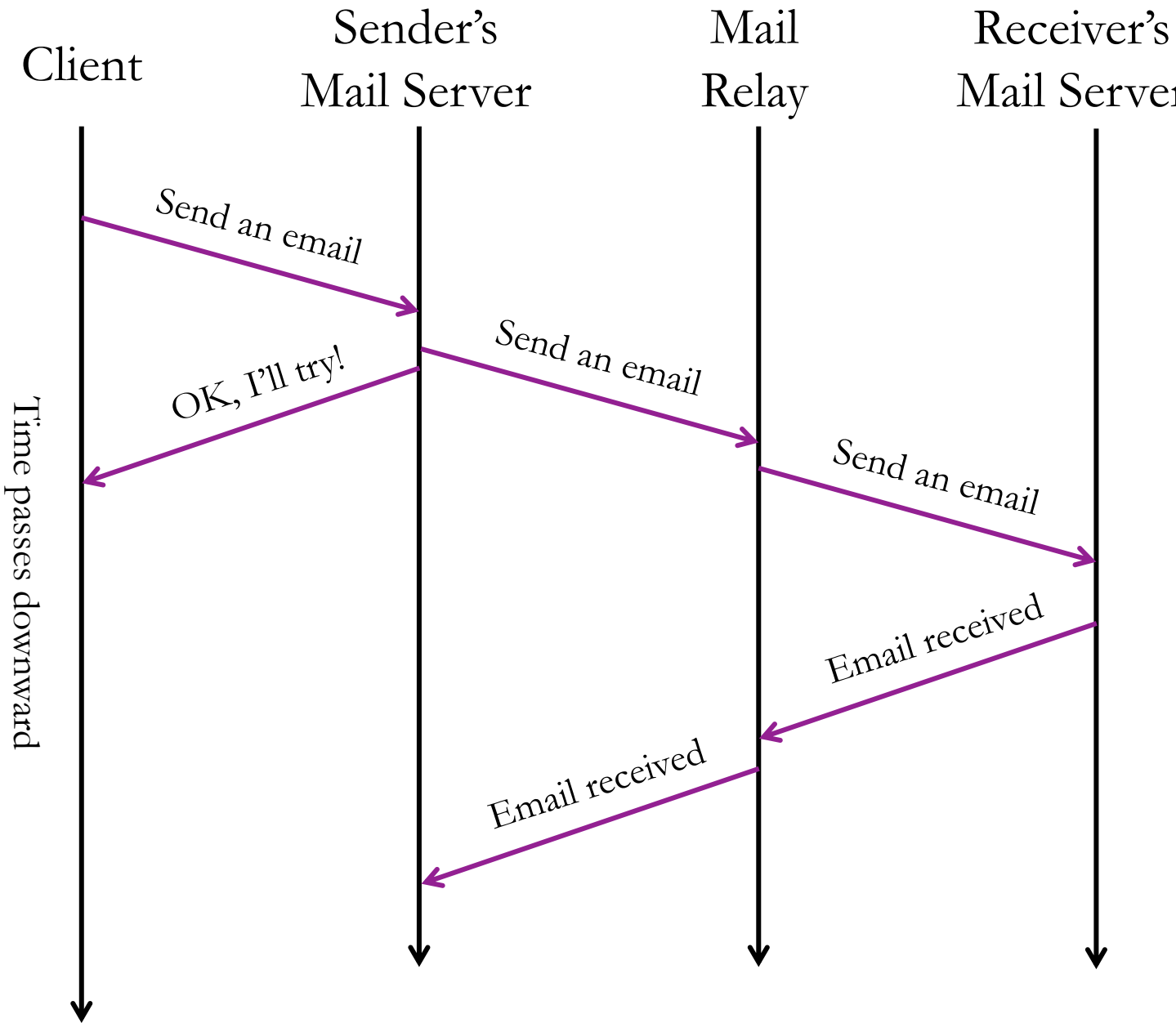
- For example, a request may involve lots of other services.
- The client may not care to wait until delivery of the message is verified.



Asynchronous Alternative

- Maybe it's OK to just acknowledge that the request was received, and finish the work later.

- **Pro:** Allows client to quickly move one.
- **Con:** Client does not learn whether the request succeeded.



Synchronous to Asynchronous – *what changed?*

- In both cases, a response was sent to the client.
- Both styles can be implemented with HTTP/REST.
- The difference is just the meaning of the requests and responses:

Synchronous style:

- *Request:* Deliver an email.
- *Response:* Delivery acknowledged.

Asynchronous style:

- *Request:* Send an email.
- *Response:* Attempt acknowledged.

What if client needs to know the results?

- The previous example was a *fire and forget* request, but sometimes the client wants asynchronous access to the results.
- Client wants to proceed immediately, but later will want to know whether request **succeeded** or to get response **data**.
- How can we support this?

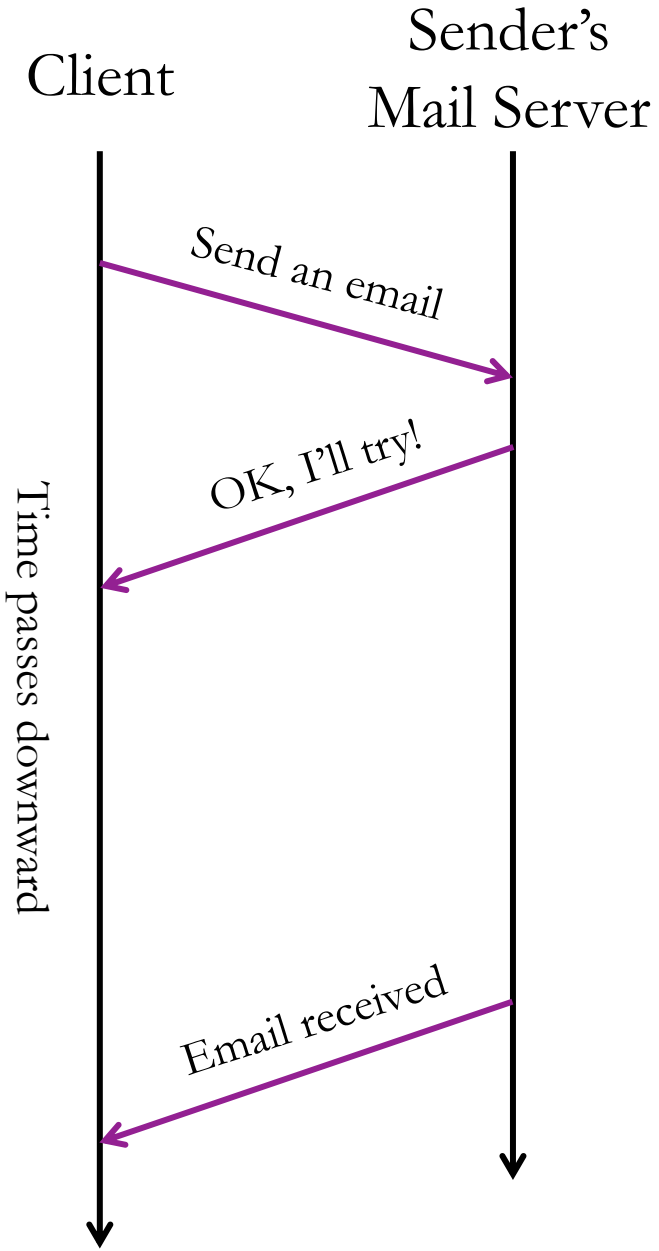
Option 1: Request Record

- Server can store a **request record** in a DB and return the unique id.
- When done, the server updates the request record in the DB.
- Client can later check on the results using the request id.
- Request → Response examples:
  - POST /messageAttempt → {“email_id”: 4390293}
  - GET /message/status/4390293 → {“status”: pending}
  - GET /message/status/4390293 → {“status”: failed,
    “error”: “invalid address”}

Option 2: Callback to Client

- Client provides a callback function (**webhook**) where it expects to receive a response.
- This only works if the client can listen for responses (always running, not NATed, etc.)

- Request → Response examples:
  - *Client sends:* POST /messageAttempt
    {"callback": "http://3.3.3.3:80/messageComplete"}
    … *time passes* …
  - Backend sends: POST /messageComplete

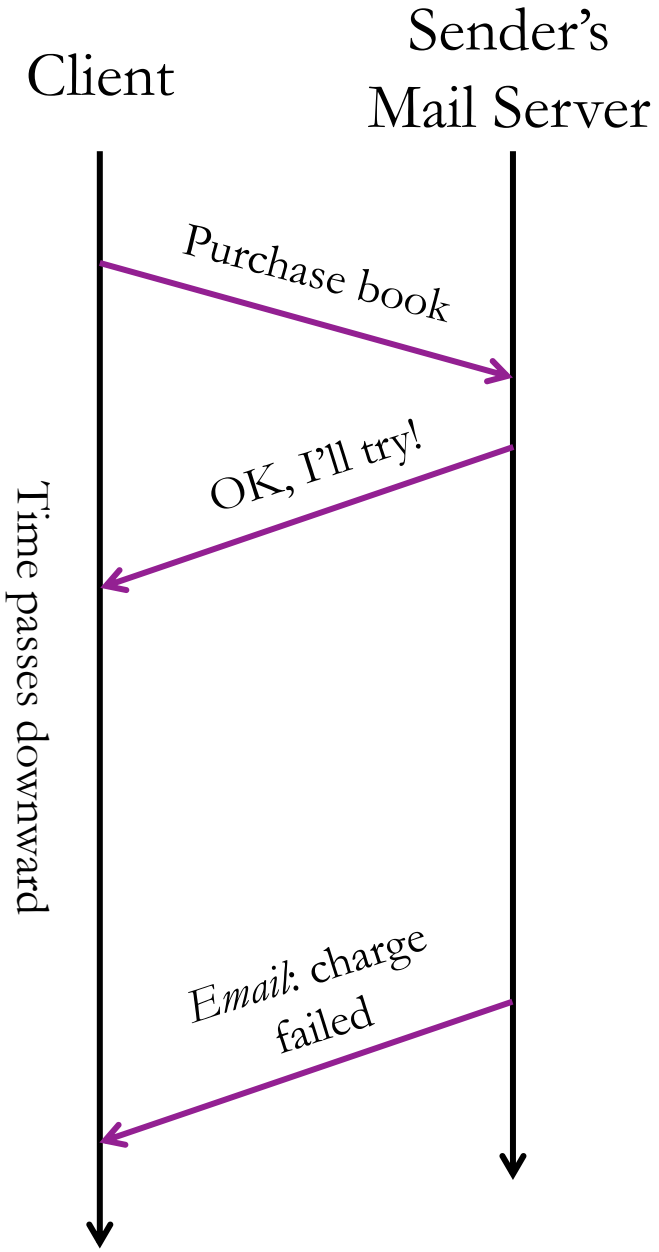


Option 3: Side channel for feedback

- Often, we let clients send "fire and forget" requests when failure is rare.
- When failure occurs, report the error to another system.

For example:

- Send customer an **email** if an order placed online fails (item was out of stock).
- **Log** an error for dev/ops or customer support staff to review.



Requests that would benefit from asynchronicity?

- Transfer a file over a network.
- Fetch a file from tape storage.
- Create a virtual machine (eg., EC2 on AWS)
- Purchase a shopping cart (ecommerce)
- Book a flight, concert, or sports ticket.
- LinkedIn connection request.

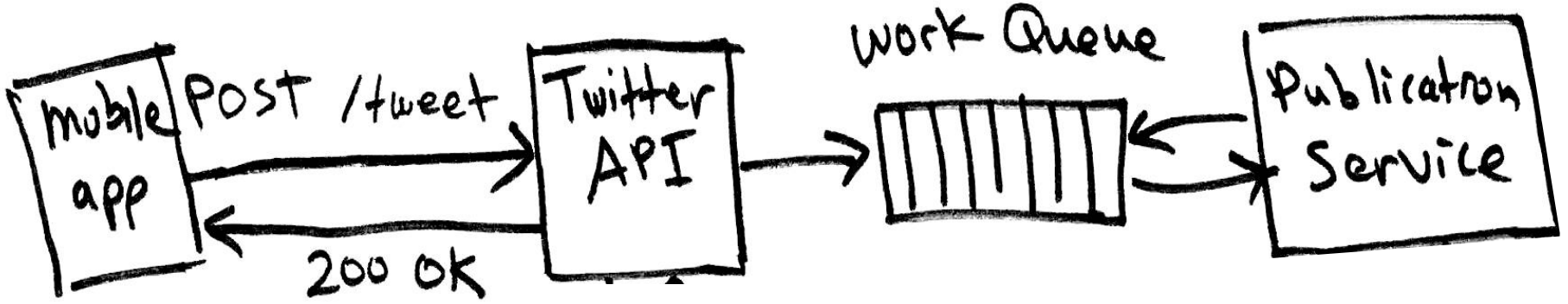
These use a different service (email) to communicate the response.

- Send a mobile push notification to another user.
- Send a text/SMS message.
- Copy tweet to 8 million follower's feeds.
- Google search: user clicked a particular search result
- Amazon.com: user searched for "dog toothbrush"

These will be used to refine the recommendation system. It's not urgent.

Message Queues provide asynchronicity & decoupling

- If client doesn't care about the response status, it can just put the request on a queue.



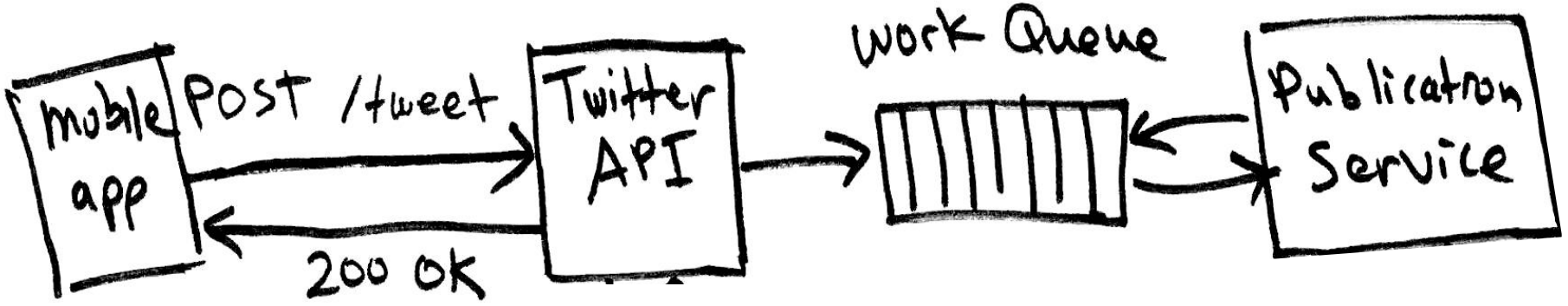
- Adding a message to a queue is very fast because it's just a data copy, without any parsing of the message or business logic.
- Prevents slowdown of upstream service due to downstream congestion. Can handle short bursts of traffic beyond system capacity.

Message Queues store requests ready to be handled

- Putting a **message** on a queue is like making an API request.
- The content of the message defines the request.

Message format == API

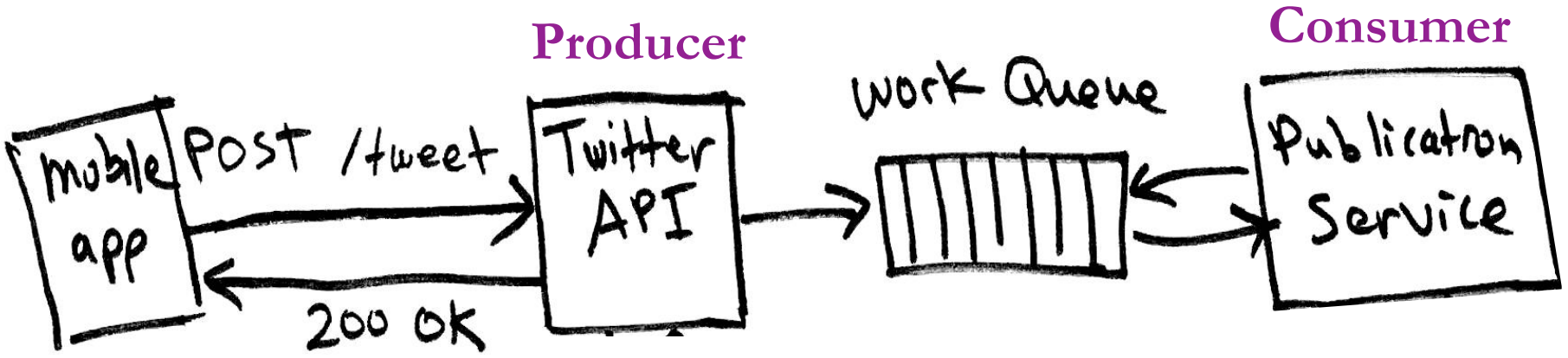
- The rules for formatting messages are a **contract** like an app's API.



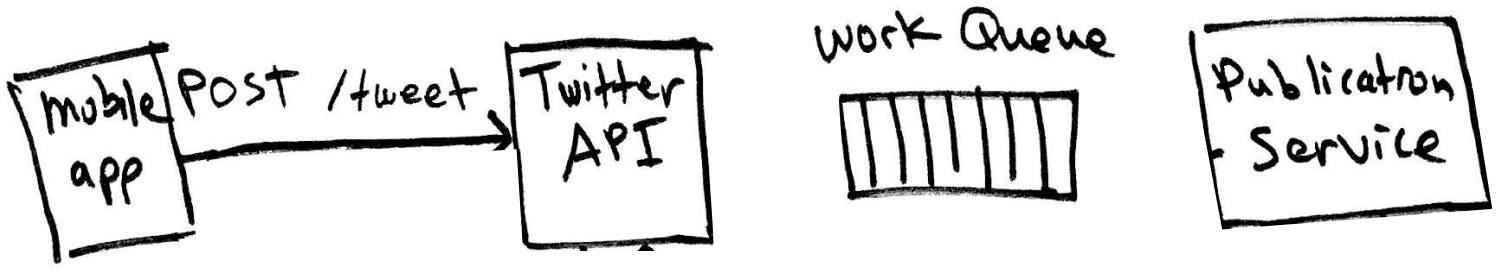
Queue terminology

- **Producer**: pushes/publishes/produces messages.
- **Consumer**: pulls/pops/consumes/subscribes-to messages.

- Optionally, a message queue can be partitioned into several virtual queues by assigning a **topic** to each message.
  - Consumers may subscribe to just one or a subset of the topics.

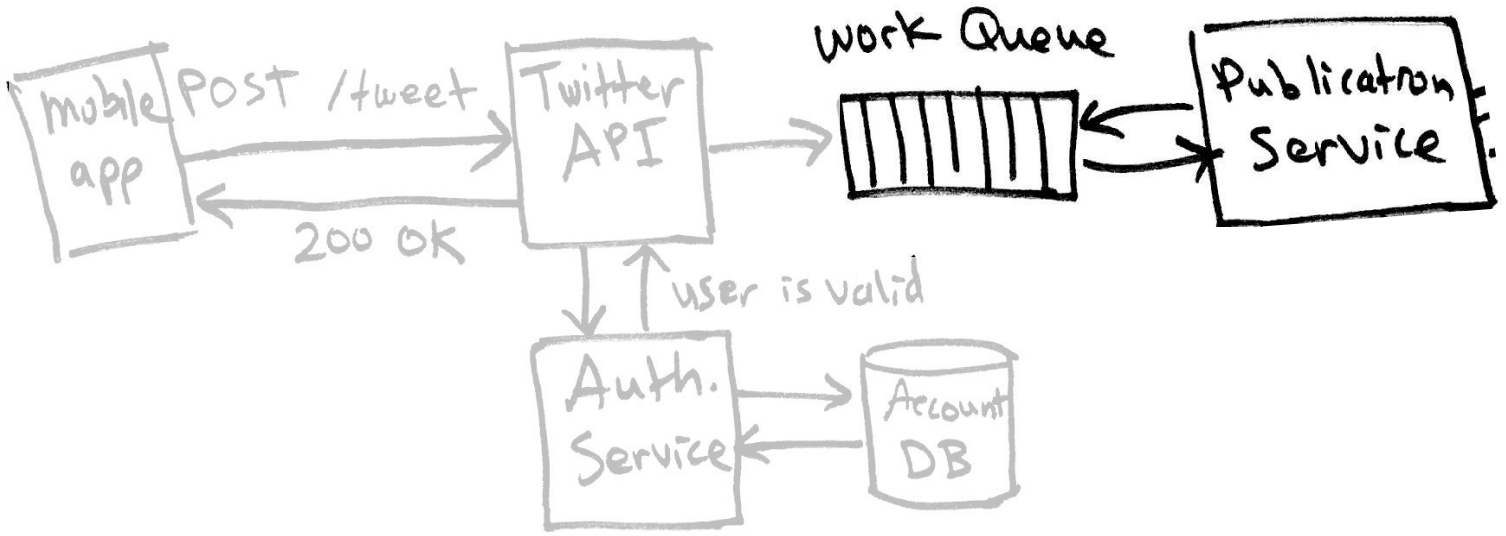


Client posts request



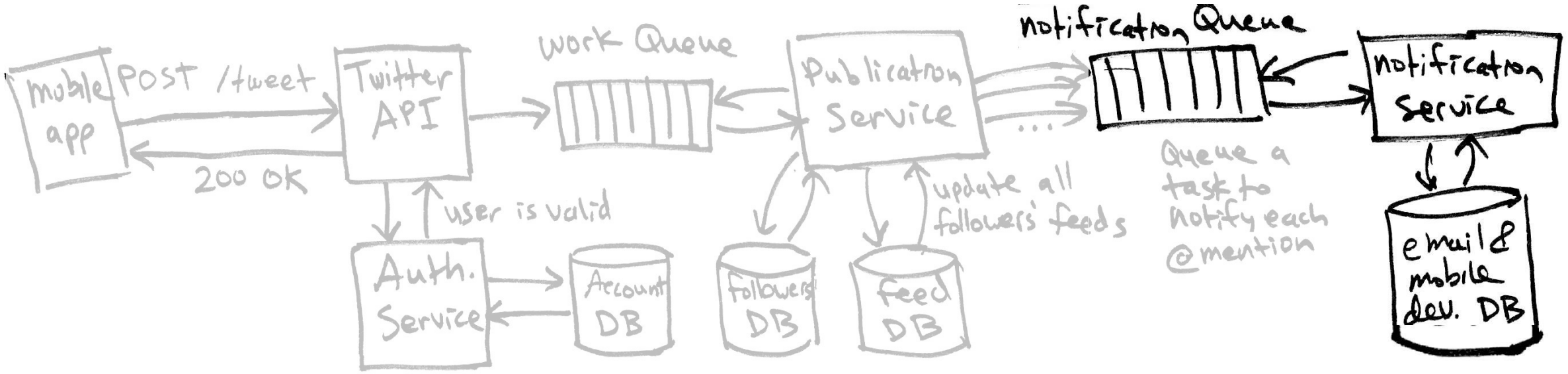
- API service receives request.
- Sends a request to auth. service to check that user token is valid.
- Puts a tweet-creation job on a queue, to be processed later.
- Sends "success" response back to user.
  - Is this premature?

Later, the Publication service handles the request



- Publication service fetches a job. It's a "publish tweet" job.
- Gets a list of followers to receive the tweet.
- Add the new tweet to all of the followers' feeds. *(Maybe millions!)*
- Queue another task to notify each of the @mentions and followers with alerts enabled. Notifications are not critical and may be slow.

Finally, the Notification service alerts users



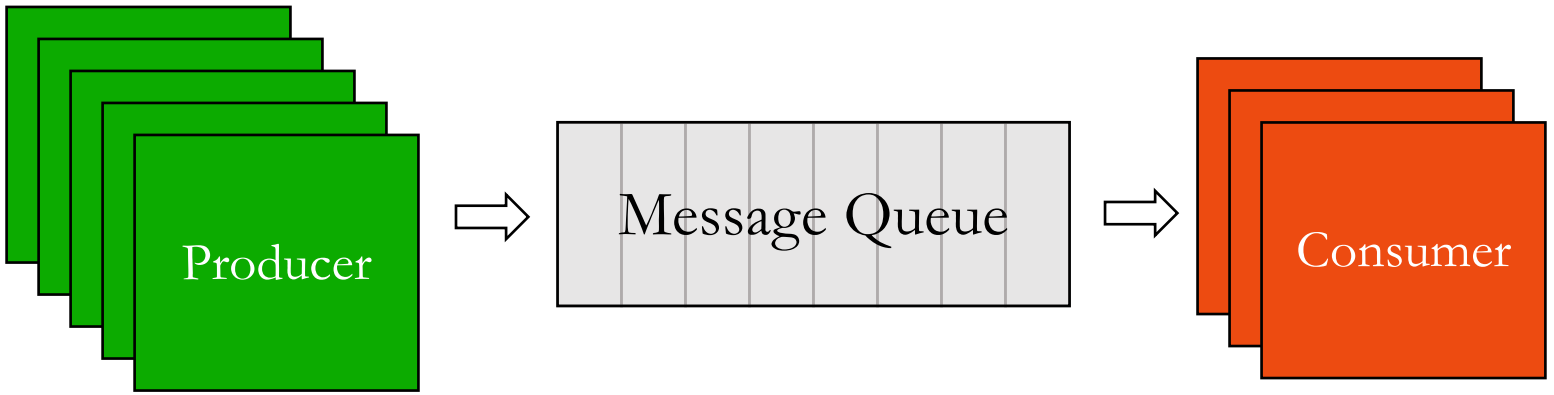
- There may be between zero and millions of notification tasks in the queue associated with the original tweet.
- Notification service dequeues each one and handles it.
- What happens if there is a failure?
  - Retry a few times, and then give up. The original tweeter does not care.

Tradeoffs

- Tightly coupled (synchronous) services are simpler to design & build.
- Loosely coupled (asynchronous) services can be faster, but either
  - Failures must be unimportant and ignored, or
  - Errors might be stored in a DB and somehow checked later.
    It can be very difficult to sensibly react to an error at a later time.
  - Errors might lead to some kind of an alert to user later (email?).

Decoupling helps scaling

- Msg Queues are a simple kind of database – store work requests.
- Many producers and many consumers can connect to the queue.
- Basic queues run on one machine & distributed queues run on clusters.
- Like a load balancer, a queue allows work to be distributed.
- Producer and consumers can be scaled separately, as needed.
- Queue *smooths* demand peaks by deferring work.



Active and passive queues

Passive Queue

- The queue accepts and stores messages until they are requested.
  - Queue is a specialized DB.
  - Maybe implemented as a DB table.
- Consumers must periodically request messages (poll).
- Producer pushes and consumer **pulls**.

Active Queue

- Queue knows where to send messages.
- Queue actively pushes messages out to subscribers.
- Subscribers must listen for messages.
- Producer pushes and queue **pushes** to consumer.

Some queue software supports both modes of operation.

Queues at different architectural levels

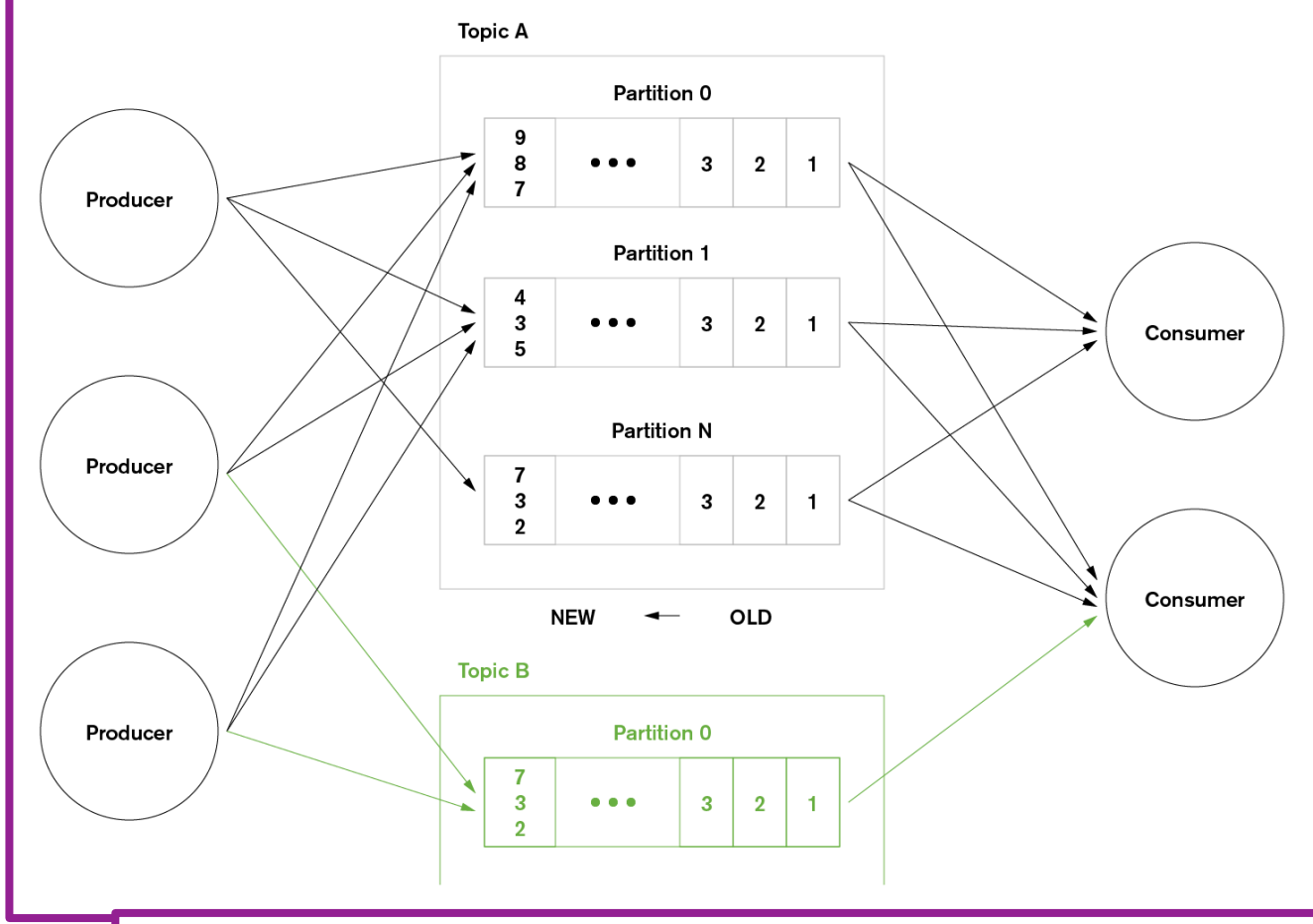
- **In-app queue**: an app can define its own queue to store work that it will do later, perhaps in a different thread.
  - For example, Java ExecutorService includes a work queue.
- **Separate queueing app**: a process that listens for pushes/fetches on a network connection.
  - Often it can run as a process on the same VM as the application pushing to it. In this case, the push's network communication is local.
  - For example, Netflix's Suro.
- **Distributed message queue**: a cluster of nodes that together implement a robust, scalable queue.
  - Allows all work to go to "one big queue."
  - For example, ActiveMQ, Kafka, RabbitMQ, AWS SQS

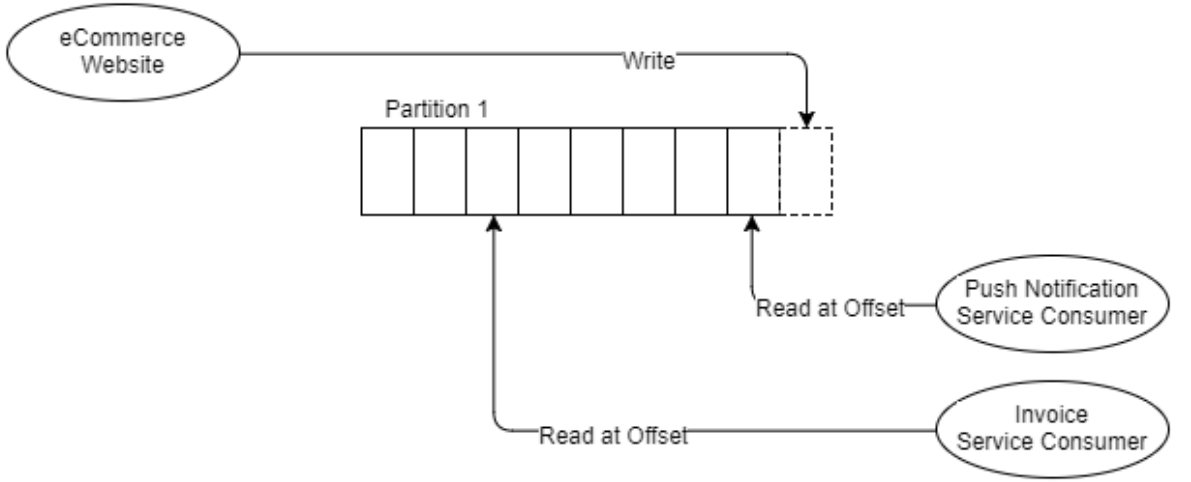
Pros and Cons

- **In-app queue**:
  - **Pros:** Simple. No separate app to deploy.
  - **Cons:** Usually not stored on disk. App crash/reboot may drop queued msgs.
- **Separate queueing app**:
  - **Pros:** Can reside on existing app VM. Can write queued msgs to a file.
  - **Cons:** Scalability is limited to one machine. Machine/disk failure drops msgs.
- **Distributed message queue**:
  - **Pros:** Massively scalable. Messages are replicated on many nodes.
    Provides a single point of coordination for many producers and consumers.
  - **Cons:** Complexity.
    Consistency side effects: eg., on RabbitMQ must chose delivery guarantee to be either "at least once" or "at most once" but cannot get "exactly once."

Eg., Kafka

- It's actually a distributed commit log, not a queue.
  - Messages are kept after reading.
- Like a DHT, data is partitioned onto multiple nodes.
- Multiple "Topics" are like separate queues.
- Uses Zookeeper for consumers to agree on point in the log to start reading.





Back pressure

- What happens if a queue "fills up?"
- It should be possible for the queue to give an error response to the producer trying to add to it.
- This is a bad thing because it will stall the service.
- DevOps/Operations staff should monitor size of queues to anticipate these problems.

Ordering

- Distributed queue cannot guarantee strict FIFO ordering of messages.
- *Tip*: If multiple messages must be ordered, send one big message.

Message Queues are backend creatures

- Like databases, messages queues are not designed to accept public requests or connections from thousands of clients.
- Your frontend should not connect directly to a Message Queue.

Recap – Message Queues.

- Services can be tightly or loosely coupled (synchronous or async.)
- Results from asynchronous calls are less apparent.
  - (fire-and-forget, request record, or callback)
- APIs can be asynchronous.
- Queues can be used to decouple systems.
  - Acts as a kind of deferred-work load balancer.
  - Allows producers and consumers to be scaled separately.
- Queues are useful at many levels:
  - In-app queues
  - Separate queueing apps
  - Distributed message queues.

Load Balancing

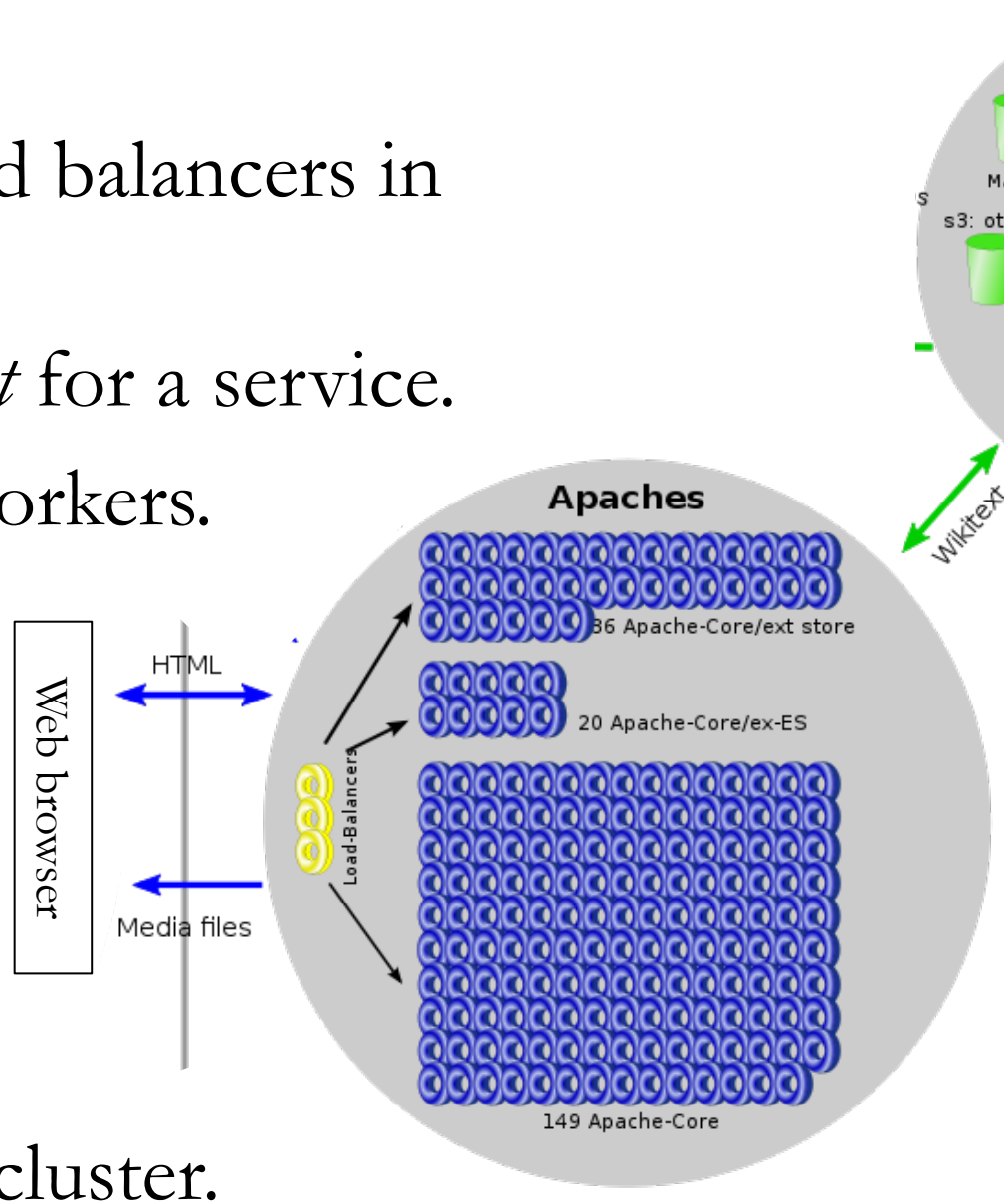
The following slides were taken with permission from Professor Steve Tarzia of Northwestern University.
https://stevetarzia.com/teaching/310/slides/Scalability%20Lecture%2007%20-%20Load%20Balancing.pdf

Last Lecture: Microservices, etc.

- Introduced **microservices** as an alternative to **monolithic** design.
- Services are **black boxes**, exposing **network APIs**.
  - Decouples development of different parts of the system.
  - Network APIs define the format and meaning of requests and responses.
- JS **Single-page Applications** (SPAs) interact directly with services.
  - Moves UI concerns away from backend code.
- In a **cross-platform system design**, the same backend service/API can serve mobile, web, and desktop apps.

Load balancers

- In the diagram below, there are 3 load balancers in front of 200 MediaWiki servers.
- Load balancer is a *single point of contact* for a service.
- Requests are *proxied* to a cluster of workers.
- Load balancer does very little work:
  - Just forward request/response and remember the request source.
  - Load balancer can relay requests for 10s-100s of application servers.
- Makes one IP address appear like one huge machine, but it's actually a cluster.

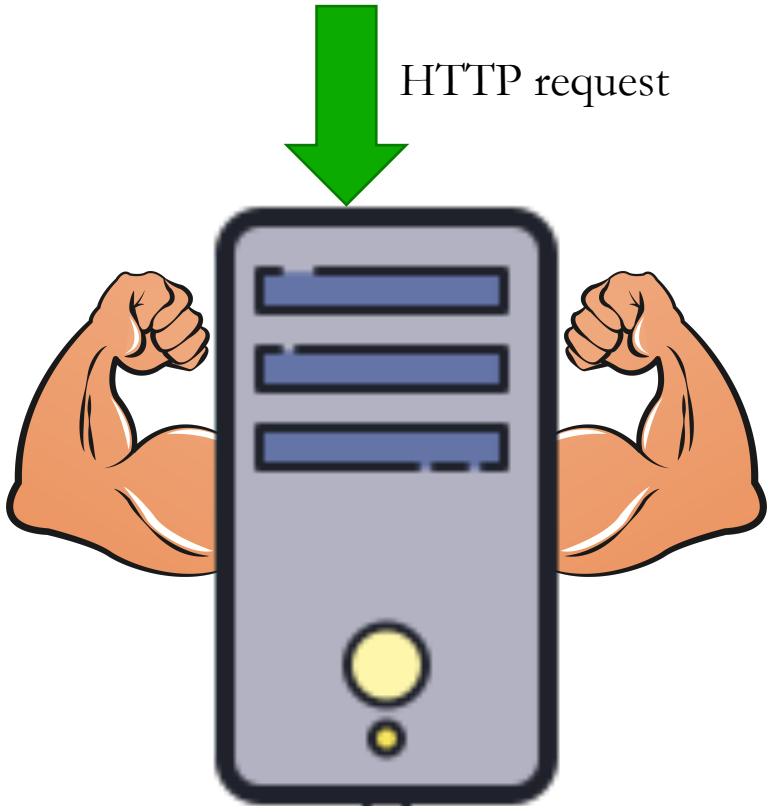


MediaWiki

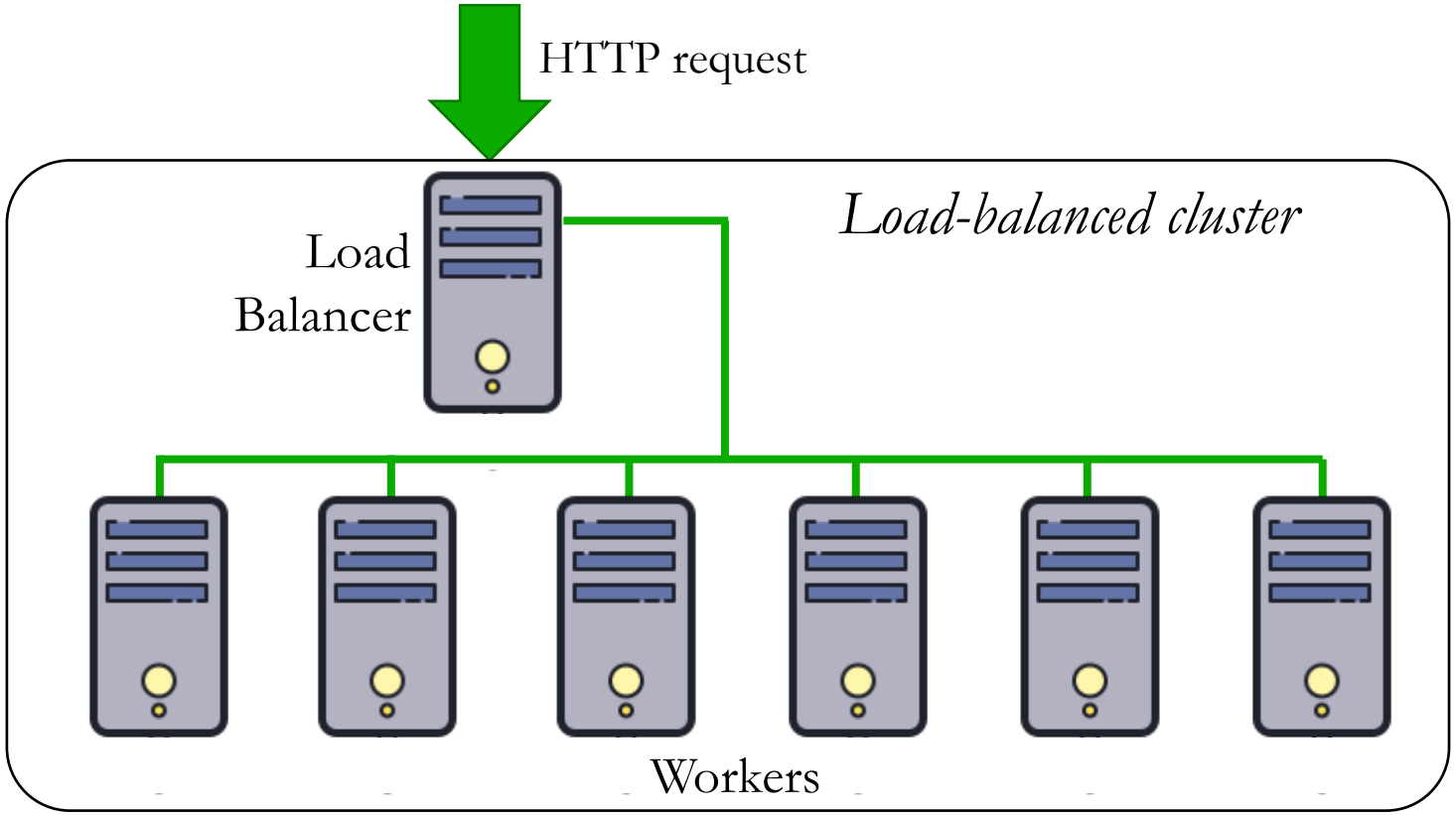
Basic idea of load balancer

- Make a cluster of servers look like one *superior* server.
- The load balancer provides the same interface as a single server. (An HTTP server operating on a single IP address)

Client thinks it's dealing with this:



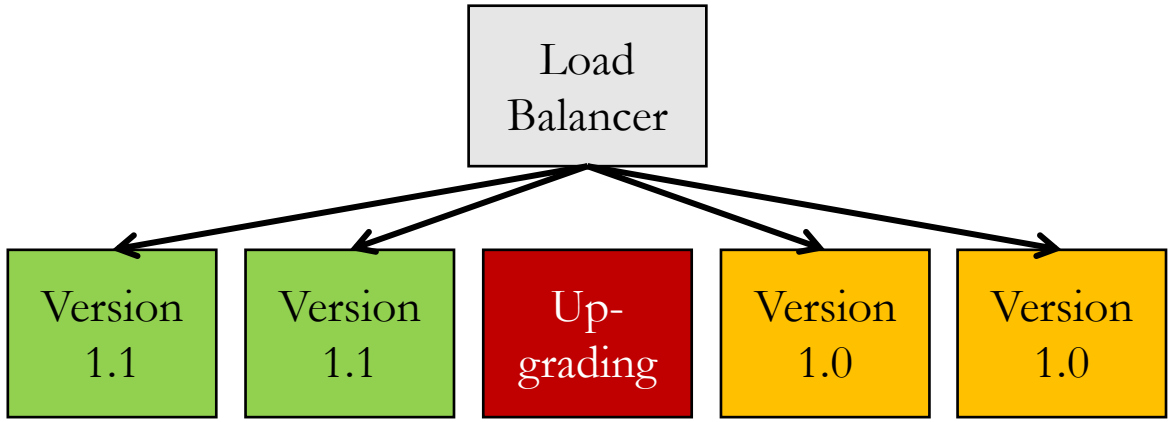
But it's just an illusion. The reality is:



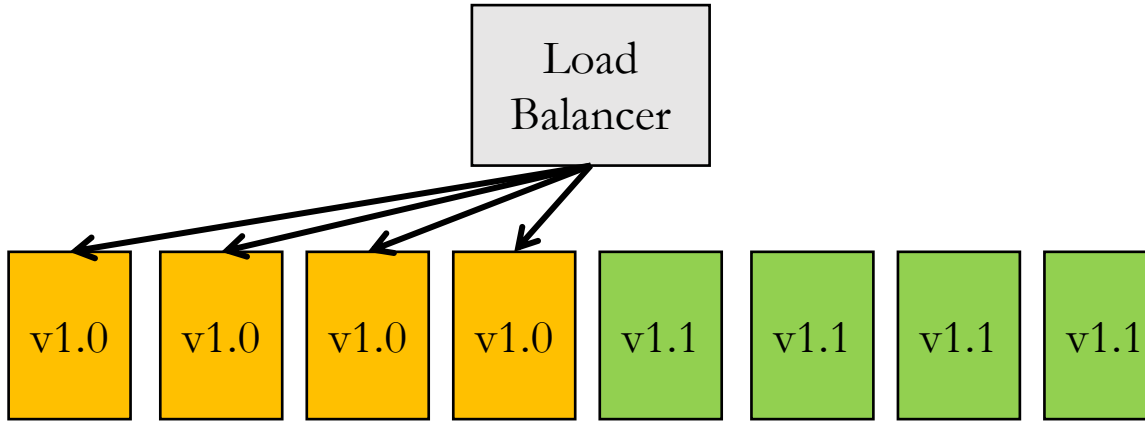
Additional benefits of a load balancer

Pros and cons of rolling vs. synchronized updates?

- Individual servers can be replaced without affecting overall service.
  - Deploy "rolling" app updates.
  - Or deploy a synchronized update



Update rolls through servers →



Created new VMs with updated SW. Will reconfigure LB all at once.

- Proxy can monitor **health** of servers
  - Periodically send a "health check" request. A simple GET API call.
  - If the request fails, then the server must be crashed.
  - Stop relaying new requests to that server.

Two types of *local* load balancers

Network Address Translation

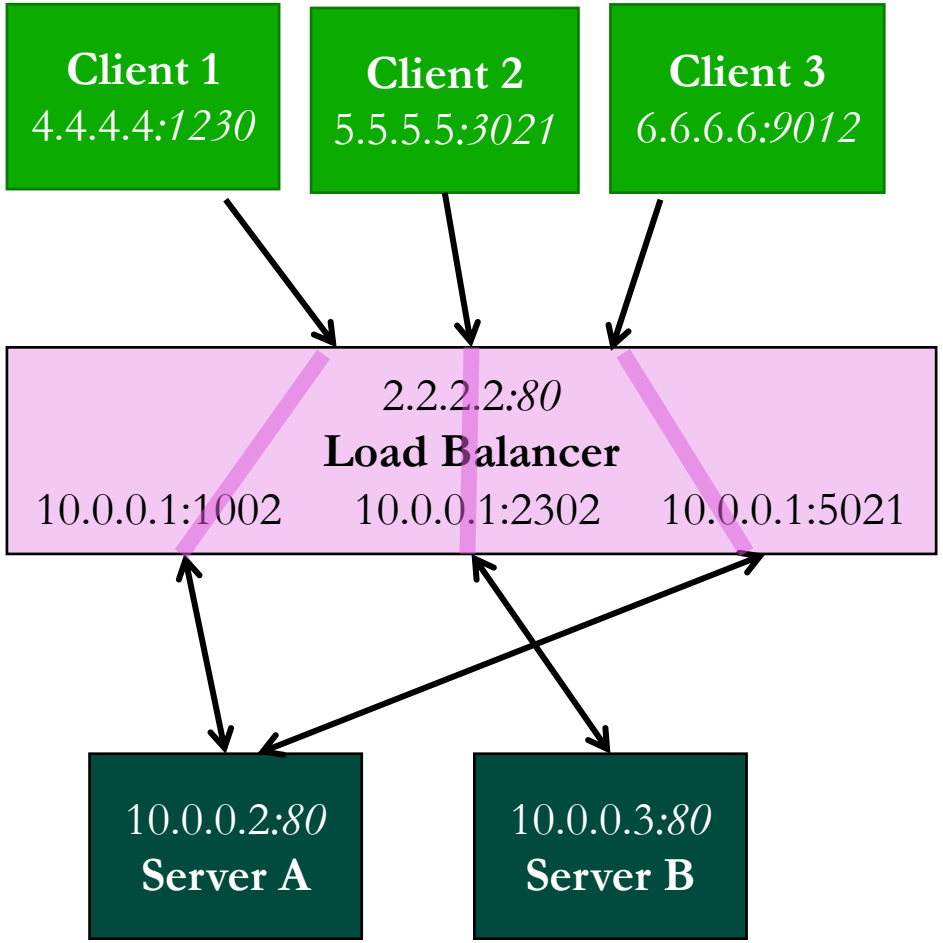
- Works at the **TCP/IP** layer.
  - Called a Layer-4 load balancer
- Forwards packets one-by-one, but remembers which server was assigned to each client.
- Is compatible with any type of service, not just HTTP.

Reverse Proxy

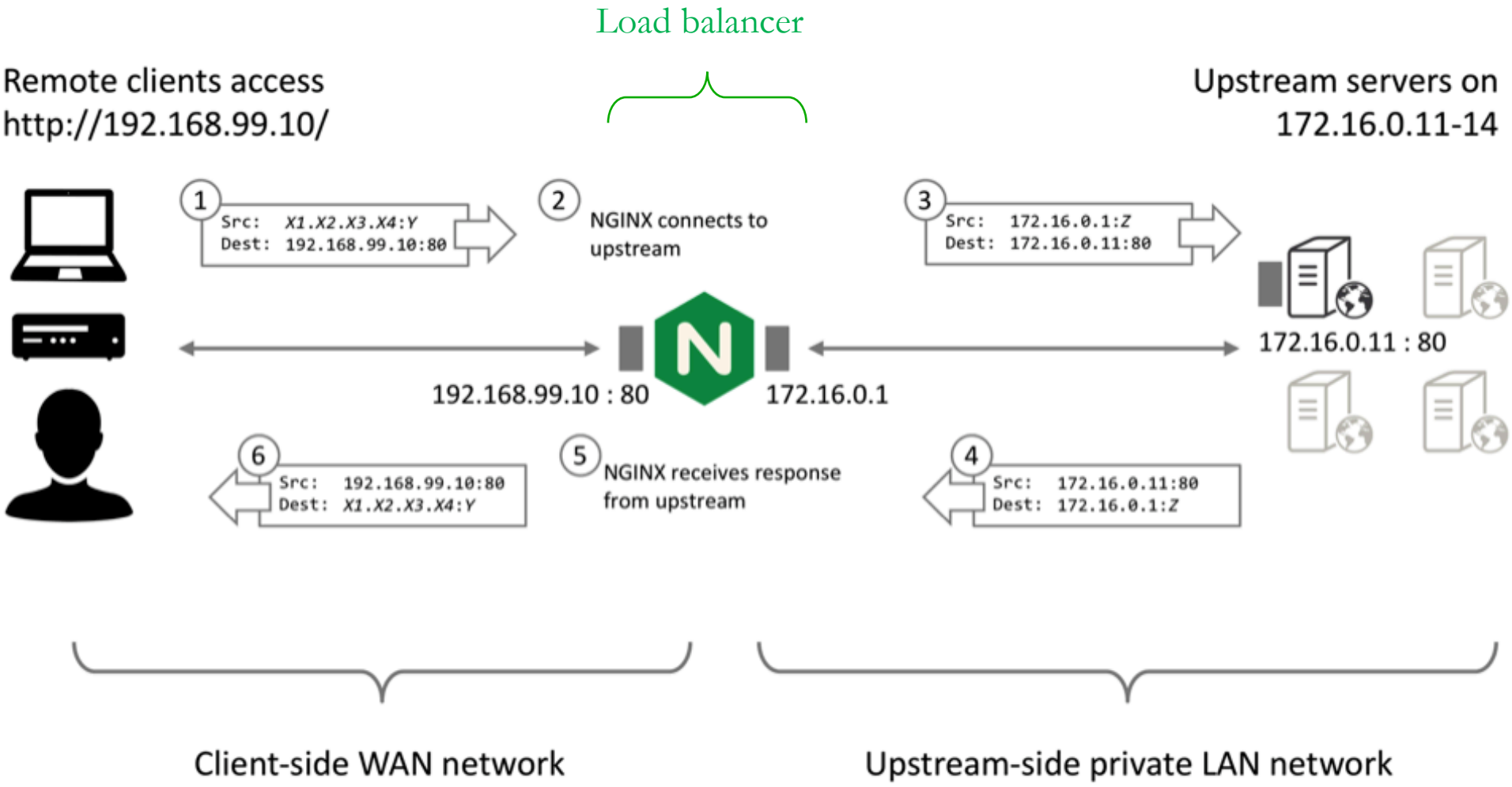
- Works at the **HTTP** layer.
  - Called an application-layer LB.
- Stores full requests/response before forwarding.
- Eg., Nginx (or maybe Squid)
- Reverse Proxy can also do:
  - SSL termination.
  - Caching.
  - Compression.

NAT Load Balancer

- For details, take CS-340.
- A type of NAT device that relays requests to multiple equivalent servers.
- NAT LB changes IP addresses and ports of packets in both directions.
- Load balancer maintains IP address and port mappings, like a traditional NAT.
- Simpler and more efficient than reverse proxy because it need not implement TCP, HTTP, or store full request/responses.

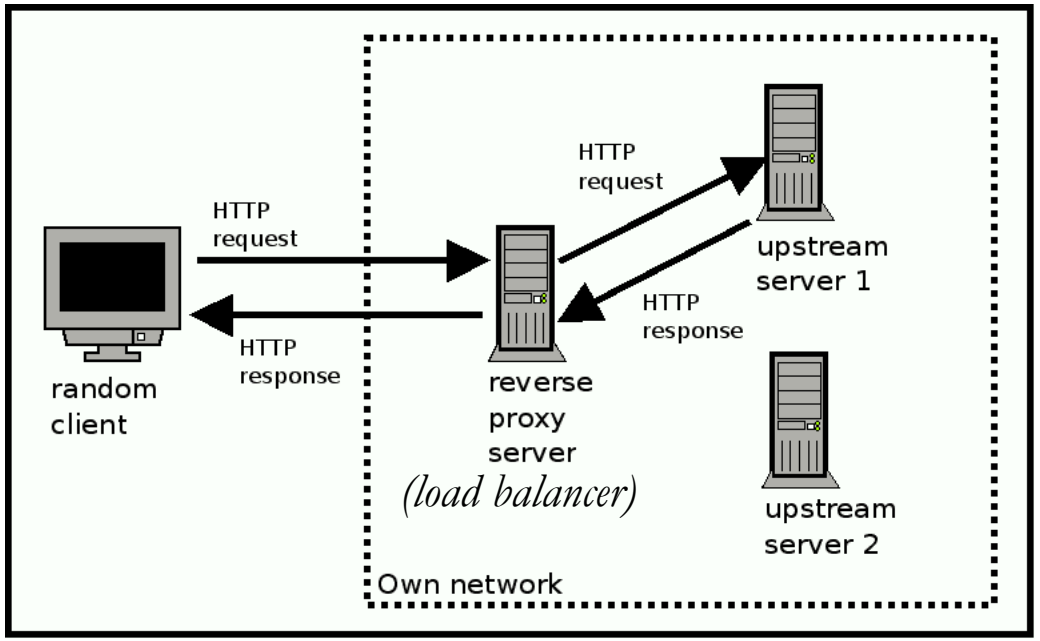


Nginx is can be used as a **Reverse Proxy** Load Balancer



1. Client sends request to the LB.
2. LB choses an "upstream" server.
3. LB relays request to upstream server.
4. Upstream server sends response to LB.
5. LB accepts response and looks up the client associated with this connection.
6. LB relays response to client.

Additional benefits of a Reverse Proxy LB



- TLS/SSL certificates can be stored just on the proxy.
  - Internal communication may be unencrypted.
- Proxy might also cache responses, but this limits its scalability.
  - Eg., **Squid** in the Wikipedia architecture.

Comparison of **Local** load balancing options

	NAT	Reverse Proxy
Routing done by:	IP address/port translation	HTTP proxy
Scale	~1–10M requests/s	~100k–1M requests/s
Services supported	Any	Only HTTP
Additional features	None	SSL termination. Caching.

- Expensive "hardware" load balancers implement NAT.
- Reverse proxies are the cheap, open-source option.
- Cloud-based LBs can do either.

Local load balancer limitations

- Load balancer machine is a **single point of failure.**
- Can only handle ~1M requests/sec.
- Resides in one data center, thus:
  - It's not near all your customers.
  - The data center is also a single point of failure.

- Huge services need more than just local load balancers.
- Can clients find a service replica without contacting a central bottleneck?
- We have a distributed **service discovery** problem.

Domain Name Service (DNS)

- DNS is a distributed directory that maps hostnames to IP addresses.
  - mail.stevetarzia.com → 54.245.121.172
- DNS uses a distributed, hierarchical, caching architecture for scalability.
- On campus, my machine sends queries to NU's DNS server.
- This local DNS resolver has cached copies of recent *(common)* answers.
  - gmail.com, northwestern.edu, facebook.com, etc…
- Local DNS resolver asks up the hierarchy if answer is not in its cache.
  - Need to ask the *nameserver* for "stevetarzia.com" for IP address of "mail".
  - To get IP address of that nameserver, would ask the "com" nameserver.
  - IP address of "com" TLD nameserver is almost certainly cached, but if not then query one of the few hard-coded root nameservers.
- For more details, take CS-340 Intro. to Computer Networking.
- AWS outage on October 22nd, 2019 was due to a DDoS attack on AWS DNS.

Round-robin DNS for load balancing

- DNS allows multiple answers to be given for a query (multiple IP addresses per domain name).
  - Client can then randomly choose one of the IP addresses.
  - Even better, DNS server can store multiple answers, but give different responses to different users (either randomly, or cyclically).
- Remember that DNS is a cached, distributed system.
  - Northwestern's DNS resolver may have been told google.com = 172.217.9.**78**.
  - UChicago's DNS resolver may have been told google.com = 172.217.9.**80.**
  - Comcast Chicago's DNS resolver was told google.com = 172.217.7.**83**.
  - Different answers are cached and relayed to all the users on the 3 networks.
- Each of those three IP addresses are different reverse-proxying load balancers sitting in front of hundreds of app servers.
- Is there a limit to scaling by DNS?

There's no limit, at least on the frontend!

Geographic load balancing with DNS

- More than just balancing load, DNS can also connect user to the **closest** replica of a service.
- Clever DNS server examines **IP address of requester** and resolves to the server that it thinks is closest to the client (IP address *geolocation*).
- In other words, the IP address answers are not just different, but **customized** for the particular client.

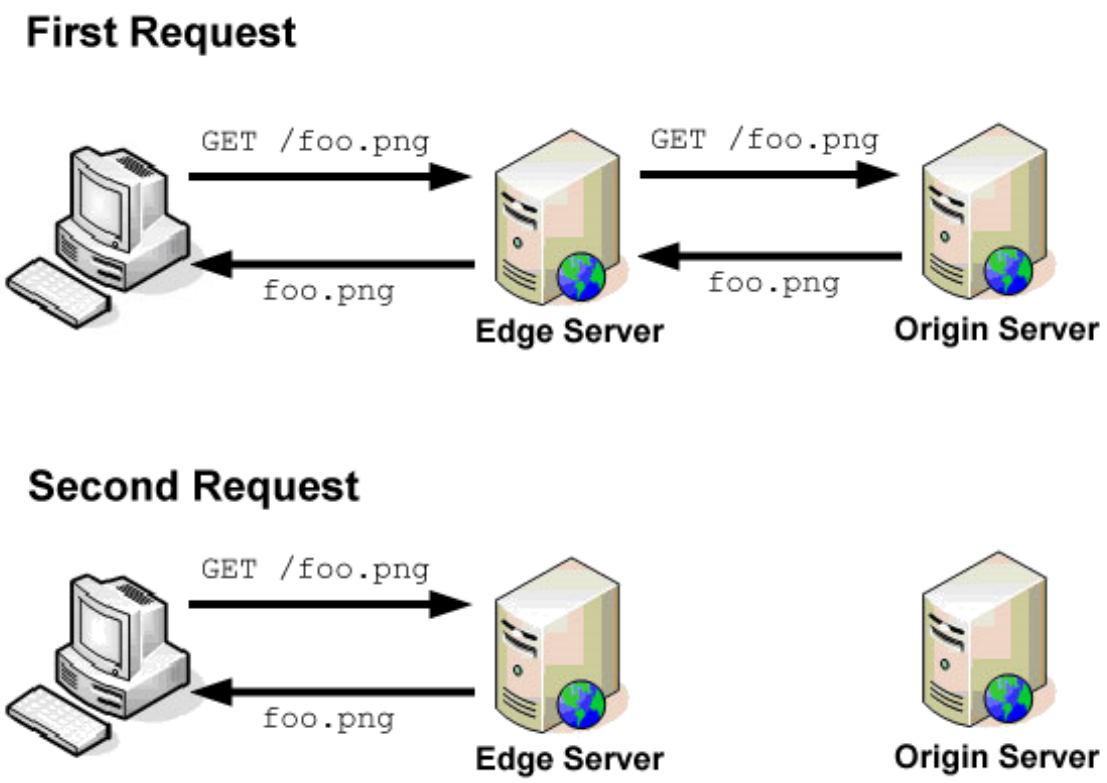
👀 *(command line demo with nslookup)*

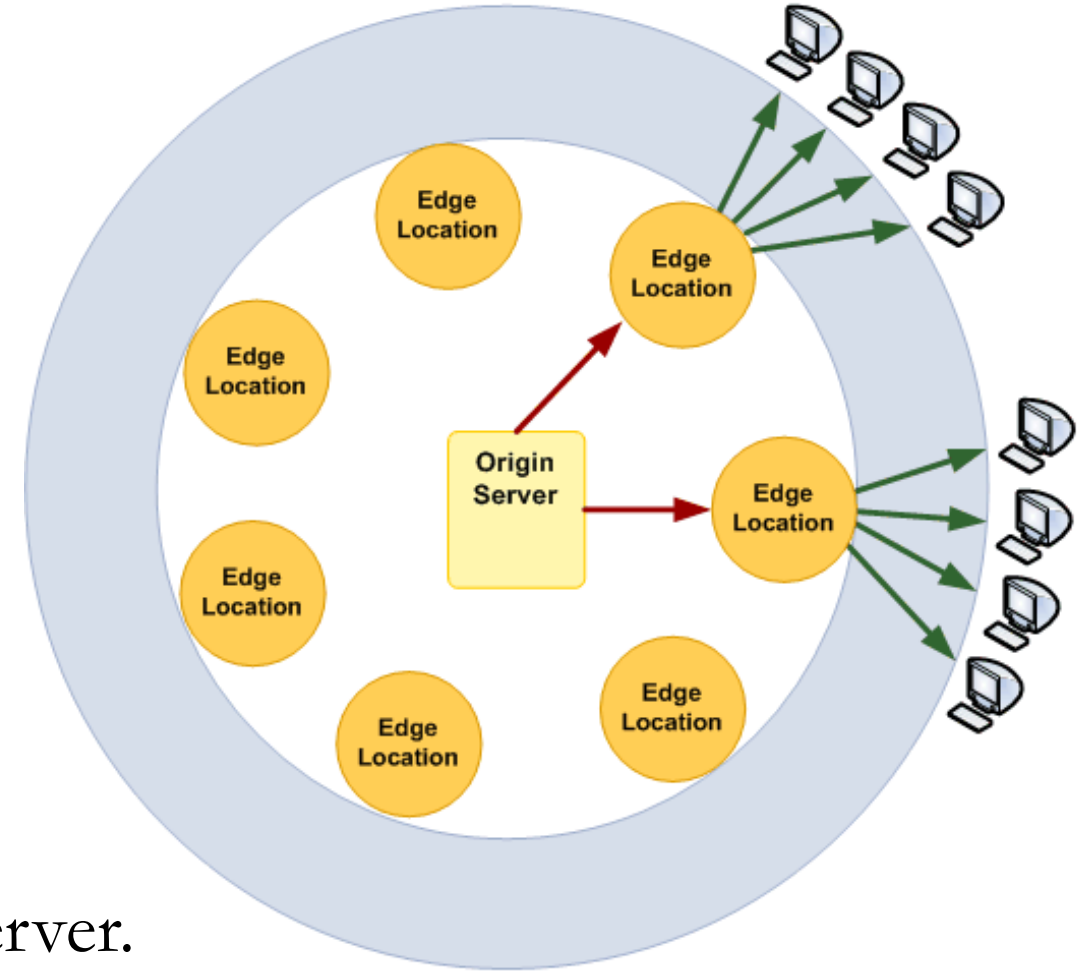
Content Delivery Network (CDN)

- Globally distributed web servers that *cache* responses for local clients.
- A CDN is just a distributed caching HTTP reverse proxy that uses DNS *(and other techniques)* to geographically load-balance.
- Eg., Akamai, Cloudflare, Cloudfront, Fastly

Distributed "edge" servers

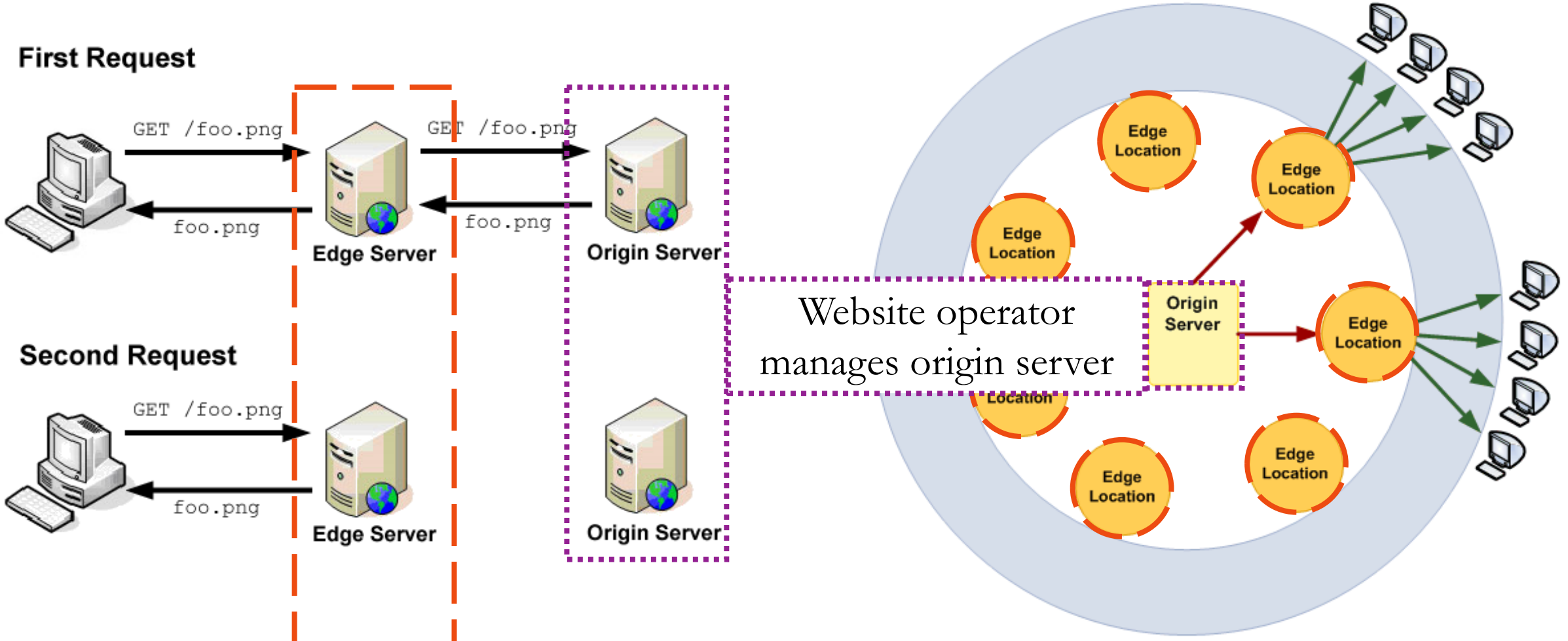
CDN uses HTTP caching proxies





- **Origin Server** is the original, central web server.
  (Sets *cache-control* HTTP headers in responses).
- Edge Servers are **caching proxies.**
  Ask origin server if don't have a cached response.

CDNs are *add-on* services



Cache-Control headers in HTTP responses

Command line demo:

- https://www.google.com does not allow caching:
  - cache-control: private, max-age=0
- https://www.google.com/images/branding/googlelogo/2x/googlelogo_color_272x92dp.png allow caching for **one year**
  - cache-control: private, max-age=31536000
- https://www.northwestern.edu also does not allow caching:
  - Cache-Control: max-age=0
- https://common.northwestern.edu/v8/css/images/northwestern.svg allows caching for one day:
  - Cache-Control: max-age=86400

Geographic load balancing with **IP Anycast**

- 8.8.8.8 is the **one IP address** for Google's huge public DNS service.
- Handles >400 billion DNS requests per day! *Anyone here use it?*
- Cannot rely on DNS for load balancing, because it **is** the DNS server!

5 million QPS!

- IP Anycast load balancing is implemented with BGP. *(Details in CS-340)*
- Basic idea is that many of Google's routers (around the world) all advertise to their neighbors that they can reach 8.8.8.8 in just one hop.
- Thus, traffic destined for 8.8.8.8 is sent to whichever of these Google routers are closest to the customer.
- Technically, this violates the principle that an IP address is a *particular* destination, but for DNS this doesn't matter because it's UDP and stateless.

Comparison of **Global** load balancing options

	DNS	IP anycast
Routing done by:	Domain Name Service	BGP
Maximum scale limited by:	Internet itself.	Internet itself.
How quickly can changes be made?	DNS TTL (minutes to hours)	BGP convergence (~minute)
Easy of deployment:	Requires advanced DNS software/config.	Must operate your own *Autonomous System* (ISP)

- For global load balancing, DNS is the most common choice.
- Tech giants (Google, Facebook, Amazon, MS, etc.) can use IP anycast.

Comparison of load balancing **Levels**

	Local	Global
Routing done by:	NAT or HTTP Proxy	DNS or BGP
Maximum scale limited by:	Speed of one machine.	Internet itself.
How quickly can changes be made?	Milliseconds	minutes to hours
Easy of deployment:	Simple, using off-the-shelf software/HW.	Requires advanced DNS software/config.

Most large services load balance at two levels:

- **Local** – provides continuous operation and scale within a data center
  - Includes *health checks & rolling updates*.
- **DNS** – for global scalability and low latency *(send users to nearby data center)*.

Recap: Load balancers

We have 2/3 of the *end-to-end* view of a basic scalable architecture!

(for *services*, at least)

- *Frontend:* Client connects to "the service" via a **load balancer**.
  - Really, the client is being directed to one of many copies of the service.
  - Global LBs (DNS and IP anycast) have no central bottlenecks.
  - Local LBs (Reverse Proxy or NAT) provide mid-level scaling and continuous operation *(health checks & rolling updates)*.

Today's topic

- *Services:* Implemented by thousands of clones.
  - If the code is **stateless** then any worker can equally handle any request.
- *Data Storage??*
  - The next big topic!