INGI2145: CLOUD COMPUTING (Fall 2014)

Cloud basics

9 October 2014

Certain lecture slides adapted from Upenn NETS212 by A. Haeberlen, Z. Ives and from Cornell CS5412 by K. Birman
Reproduced with permission

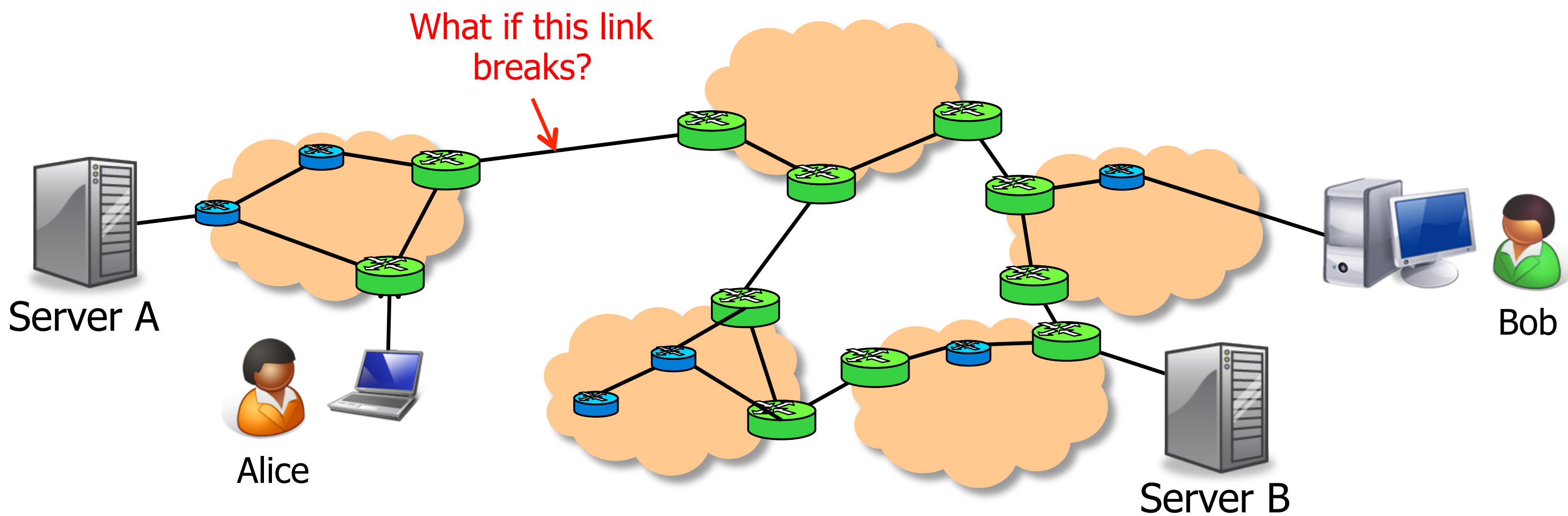
Announcements

- First homework assignment will be announced next week
  - You will need to be setup with AWS
  - Next week's lab (17 Oct) will cover important background for this assignment
- Tomorrow's lab session is about Amazon Storage Services
  - Bring your own laptop
  - We will try to use the Intel rooms again for better connectivity

Plan for today

- Parallel programming and its challenges
  - Parallelization and scalability, Amdahl's law
  - Network partitions, CAP theorem, relaxed consistency NEXT
- Cloud basics
  - Anatomy of Cloud applications
  - Scaling: stateless, caching, and sharding
- Example components
  - Application server: Node.js
  - In-memory cache: Memcached
- Scaling memcache at Facebook

Network partitions



- Network can partition
  - Hardware fault, router misconfigured, undersea cable cut, ...
  - Result: Gobal connectivity is lost
  - What does this mean for the properties of our system?

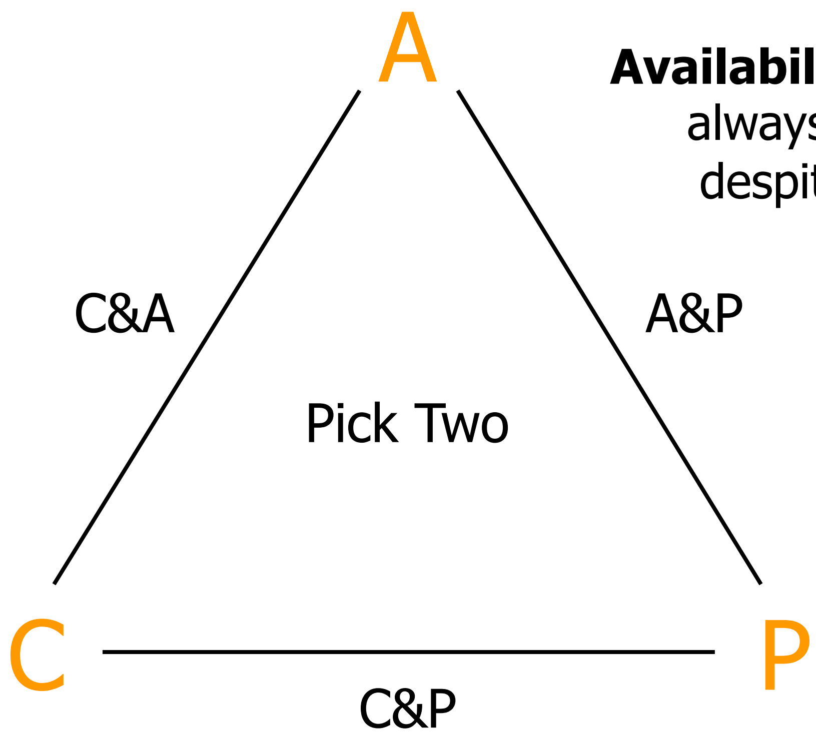
The CAP theorem

- What we want from a web system:
  - Consistency: All clients single up-to-data copy of the data, even in the presence of concurrent updates
  - Availability: Every request (including updates) received by a non-failing node in the system must result in a response, even when faults occur
  - Partition-tolerance: Consistency and availability hold even when the network partitions

- Can we get all three?
  - CAP theorem: We can get <u>at most two</u> out of the three
    - Which ones should we choose for a given system?
  - Conjecture by Brewer; proven by Gilbert and Lynch

Visual CAP

Availability: Each client can always read and write despite node failures



Consistency: All clients always have the same view of the data at the same time

Partition-tolerance: The system continues to operate despite arbitrary message loss

Common CAP choices

- Example #1: Consistency & Partition tolerance
  - Many replicas + consensus protocol
  - Do not accept new write requests during partitions
  - Certain functions may become unavailable

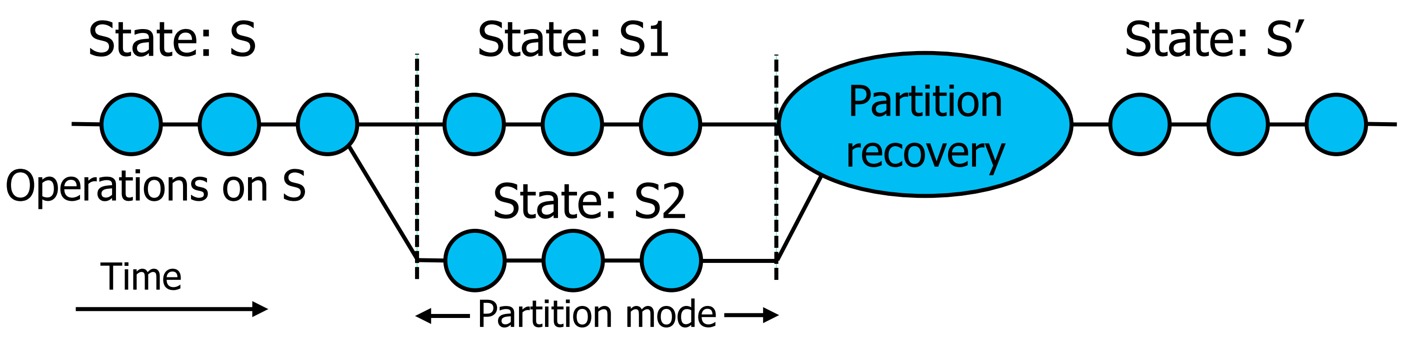
- Example #2: Availability & Partition tolerance
  - Many replicas + relaxed consistency
  - Continue accepting write requests
  - Clients may see inconsistent state during partitions

"2 of 3" view is misleading

- Meaning of C&A over P is unclear
  - If a partition occurs, the choice must be reverted to C or A
  - No reason to forfeit C or A when system is not partitioned
- Choice of C and A can occur many times within the same system at fine granularity
- Three properties are more of a continuous
  - Availability is 0 to 100
  - Many levels of consistency
  - Disagreement within the system whether a partition exists
- The modern CAP goal should be to maximize application-specific combinations of C and A

Dealing with partitions

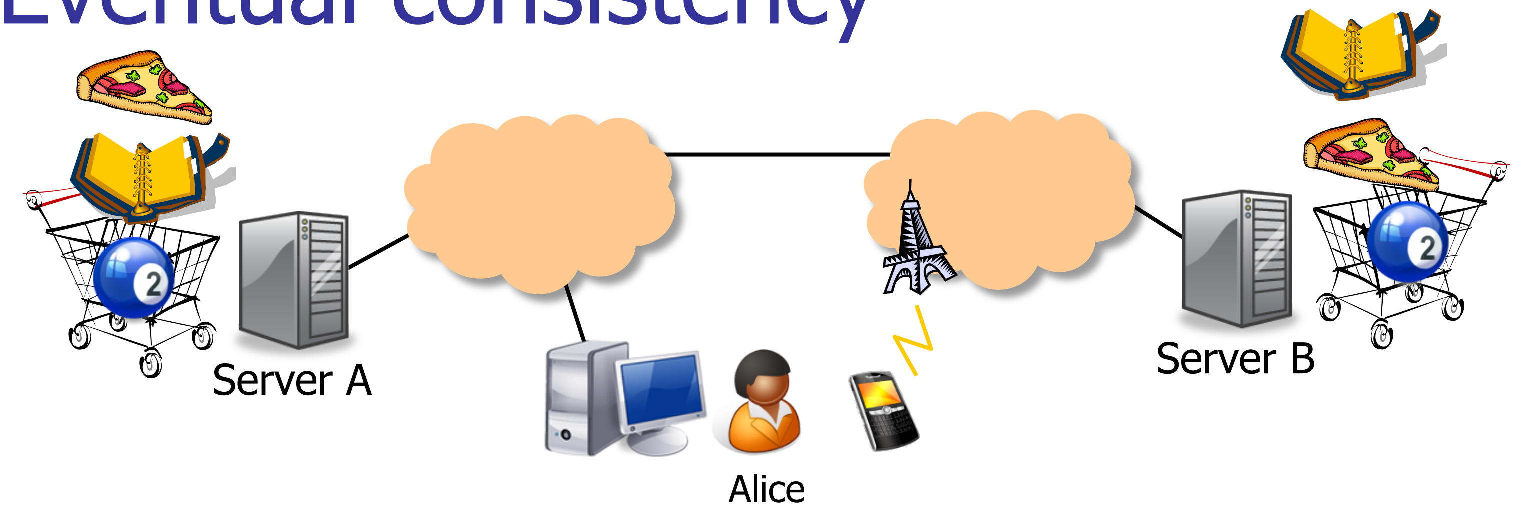
- Detect partition
- Enter an explicit partition mode that can limit some operations
- Initiate partition recovery when communication is restored
  - Restore consistency and compensate for mistakes made while the system was partitioned



Which operations should proceed?

- Depends primarily on the invariants that the system intends to maintain
- If an operation is allowed and turns out to violate an invariant, the system must restore the invariant during recovery
  - Example: 2 objects are added with the same (unique) key; to restore, we check for duplicate keys and merge objects
- If invariant cannot be violated, system must prohibit or modify the operation (e.g. record the intent and execute it after)
  - Example: delay charging the credit card; user does not see system is not available

Eventual consistency



- Idea: Optimistically allow updates
  - Don't coordinate with ALL replicas before returning response
  - But ensure that updates reach all replicas eventually
    - What do we do if conflicting updates were made to different replicas?
  - Good: Decouples replicas. Better performance, availability under partitions
  - (Potentially) bad: Clients can see inconsistent state

Partition recovery

- State on both sides must become consistent
- Compensation for mistakes during partition
- Start from state at the time of the partition and roll forward both sets of operations in some way, maintaining consistency
- The system must also merge conflicts
  - constraint certain operations during partition mode so that conflicts can always be merged automatically
  - detect conflicts and report them to a human
  - use commutative operations as a general framework for automatic state convergence
    - commutative replicated data types (CRDTs)

Compensate for mistakes

- Tracking and limitation of partition-mode operations ensures the knowledge of which invariants could have been violated
  - trivial ways such as "last writer wins", smarter approaches that merge operations, and human escalation
- For externalized mistakes typically requires some history about externalized outputs
- System could execute orders twice
  - If the system can distinguish two intentional orders from two duplicate orders, it can cancel one of the duplicates
  - If externalized, send an e-mail explaining the order was accidentally executed twice but that the mistake has been fixed and to attach a coupon for a discount

Relaxed consistency: ACID vs. BASE

- Classical database systems: ACID semantics
  - Atomicity
  - Consistency
  - Isolation
  - Durability

- Modern Internet systems: BASE semantics
  - Basically Available
  - Soft-state
  - Eventually consistent

Recap: Consistency and partitions

- Use replication to mask limited # of faults
  - Can achieve strong consistency by having replicas agree on a common request ordering
  - Even non-crash faults can be handled, as long as there are not too many of them (typical limit: 1/3)

- Partition tolerance, availability, consistency?
  - Can't have all three (CAP theorem)
  - Typically trade-off between C and A
  - If service works with weaker consistency guarantees, such as eventual consistency, can get a compromise (BASE)

Plan for today

- Parallel programming and its challenges ✓
  - Parallelization and scalability, Amdahl's law ✓
  - Network partitions, CAP theorem, relaxed consistency ✓
- Cloud basics
  - Anatomy of Cloud applications ← NEXT
  - Scaling: stateless, caching, and sharding
- Example components
  - Application server: Node.js
  - In-memory cache: Memcached
- Scaling memcache at Facebook

Recap: Cloud benefits

- Elastic, just-in-time infrastructure
- More efficient resource utilization
- Pay for what you use
- Potential to reduce processing time
  - Parallelization
- Leverage multiple data centers
  - High availability, lower response times

- How do applications exploit these benefits?

Today's Cloud applications

- Web applications
  - Client/server paradigm
  - Request/response messaging pattern
  - Interactive communication
- Processing pipelines
  - Examples: Indexing, data mining, image processing, video transcoding, document processing
- Batch processing systems
  - Example: report generation, fraud detection, analytics, backups, automated testing

Many styles of system

- Near the edge of the application focus is on vast numbers of clients and rapid response
- Inside we find data-intensive services that operate in a pipelined manner, asynchronously
- Deep inside the application we see a world of virtual computer clusters that are scheduled to share resources and on which applications like MapReduce (Hadoop) are very popular

Example: Obama for America AWS

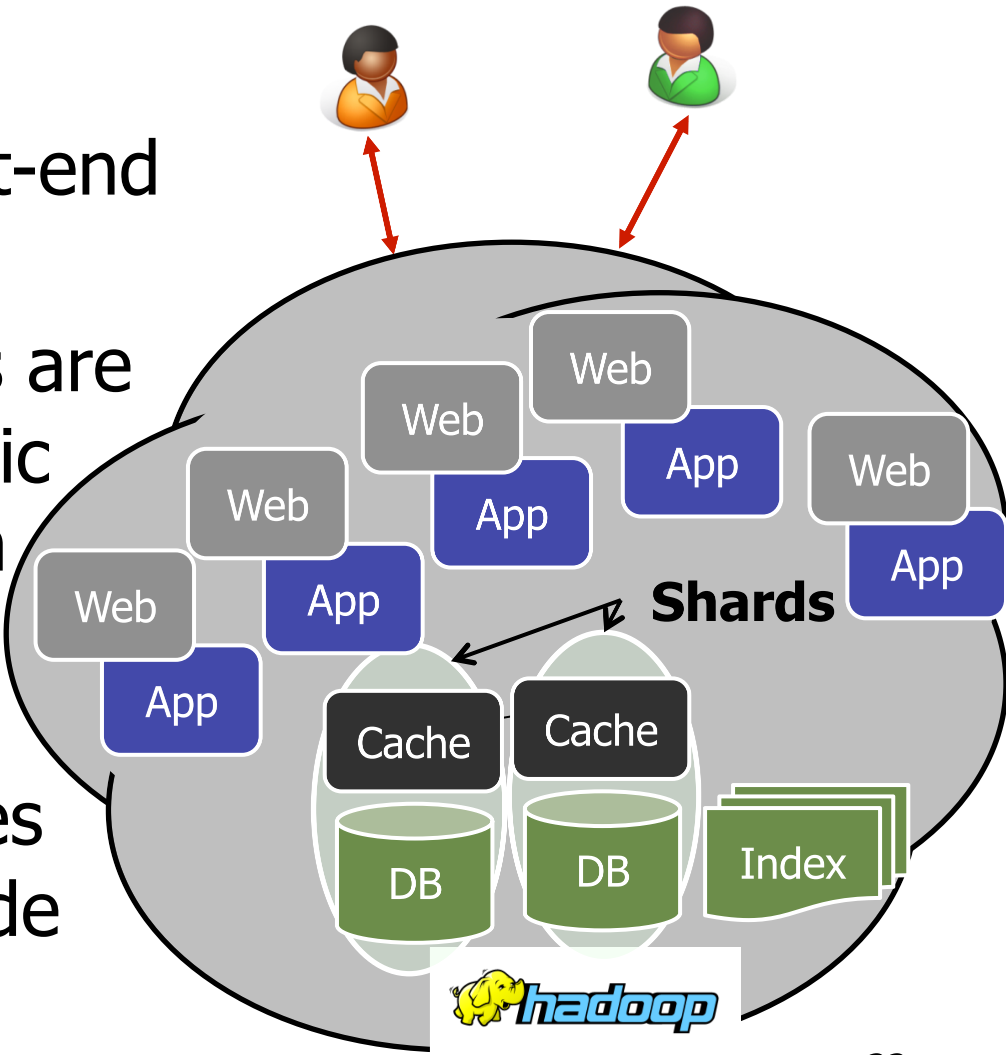
http://awsofa.info/

How are Cloud apps structured?

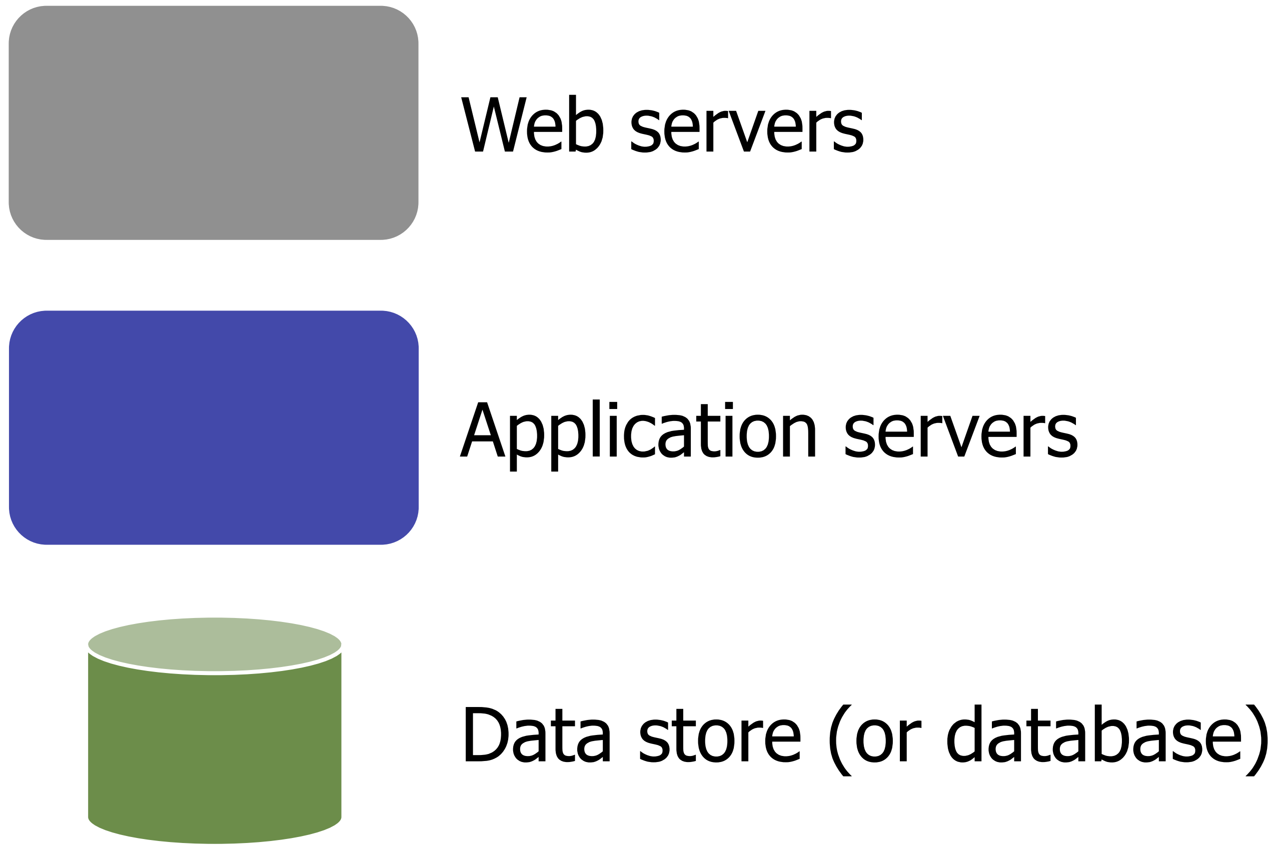
- Clients talk to application using Web browsers or the Web services standards
  - But this only gets us to the outer “skin” of the data center, not the interior
  - Consider Amazon: it can host entire company web sites (like Netflix.com), data (S3), servers (EC2), databases (RDS) and even virtual desktops!

Big picture overview

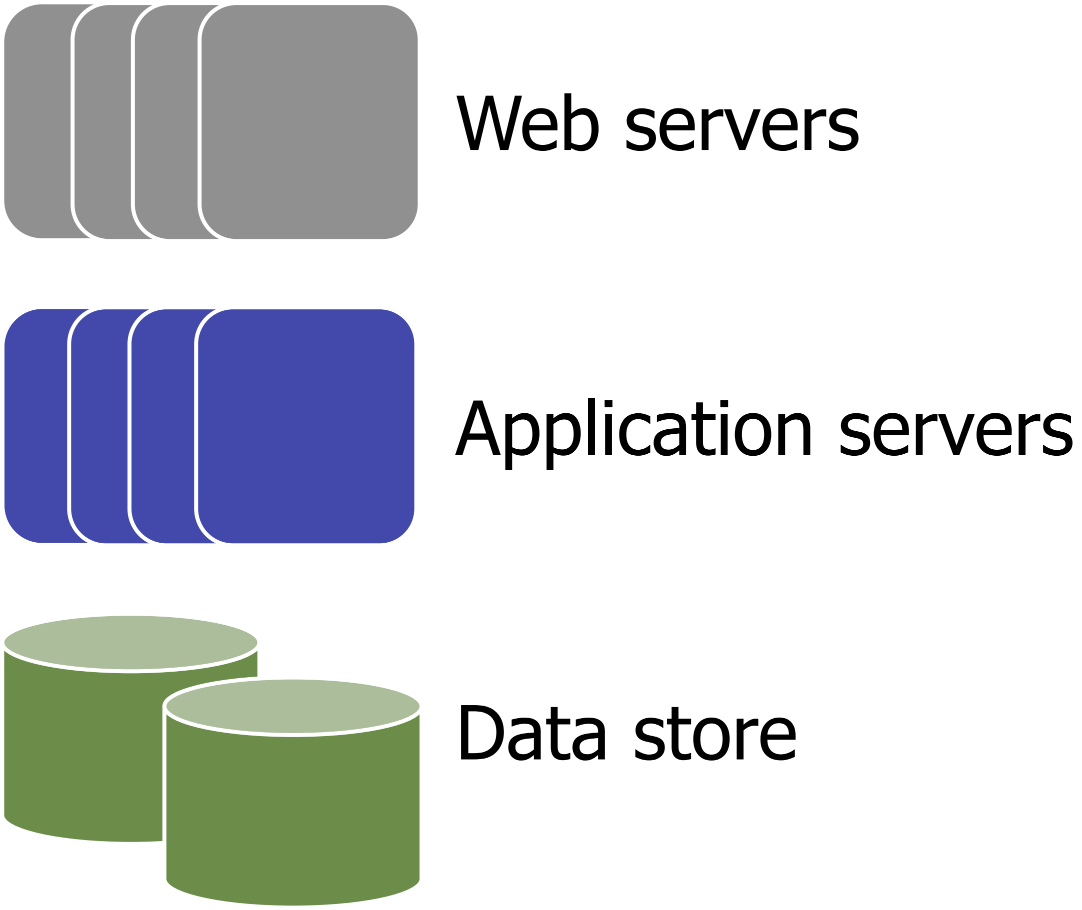
- Client requests are handled in by front-end Web servers
- Application servers are invoked for dynamic content generation and run app logic
  - PHP, Java, Python, …
- Back-end databases manage and provide access to data



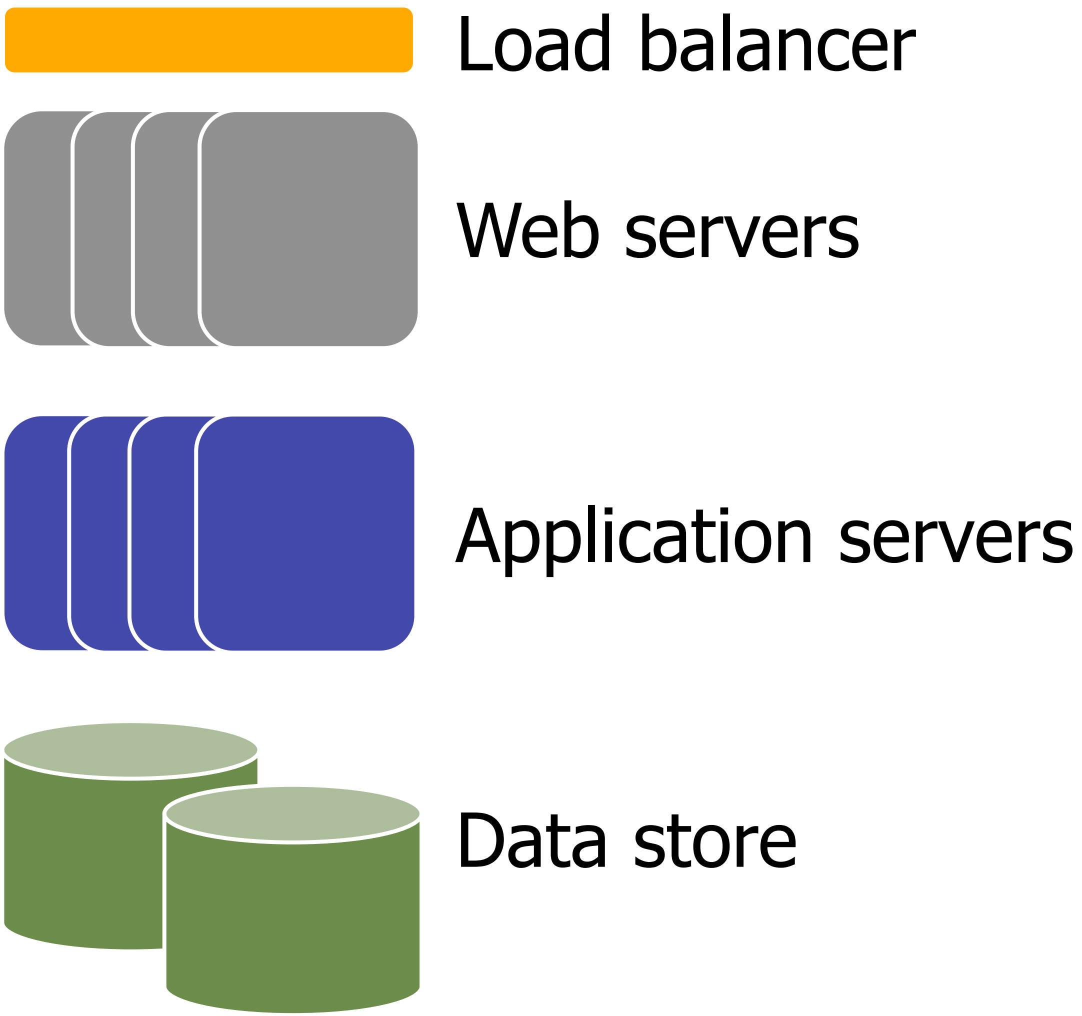
Applications with multiple tiers



Redundancy at each tier



Load balancer

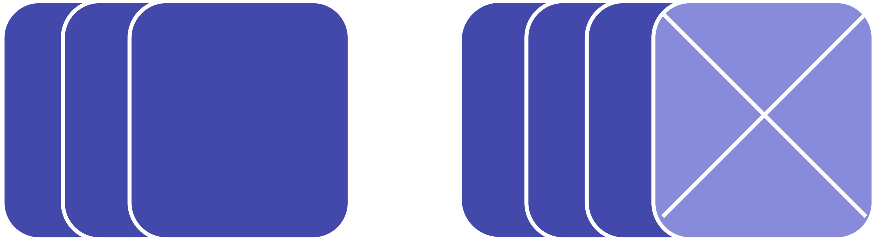


Plan for today

- Parallel programming and its challenges ✔
  - Parallelization and scalability, Amdahl's law ✔
  - Network partitions, CAP theorem, relaxed consistency ✔
- Cloud basics
  - Anatomy of Cloud applications ✔
  - Scaling: stateless, caching, and sharding ← NEXT
- Example components
  - Application server: Node.js
  - In-memory cache: Memcache
- Scaling memcached at Facebook

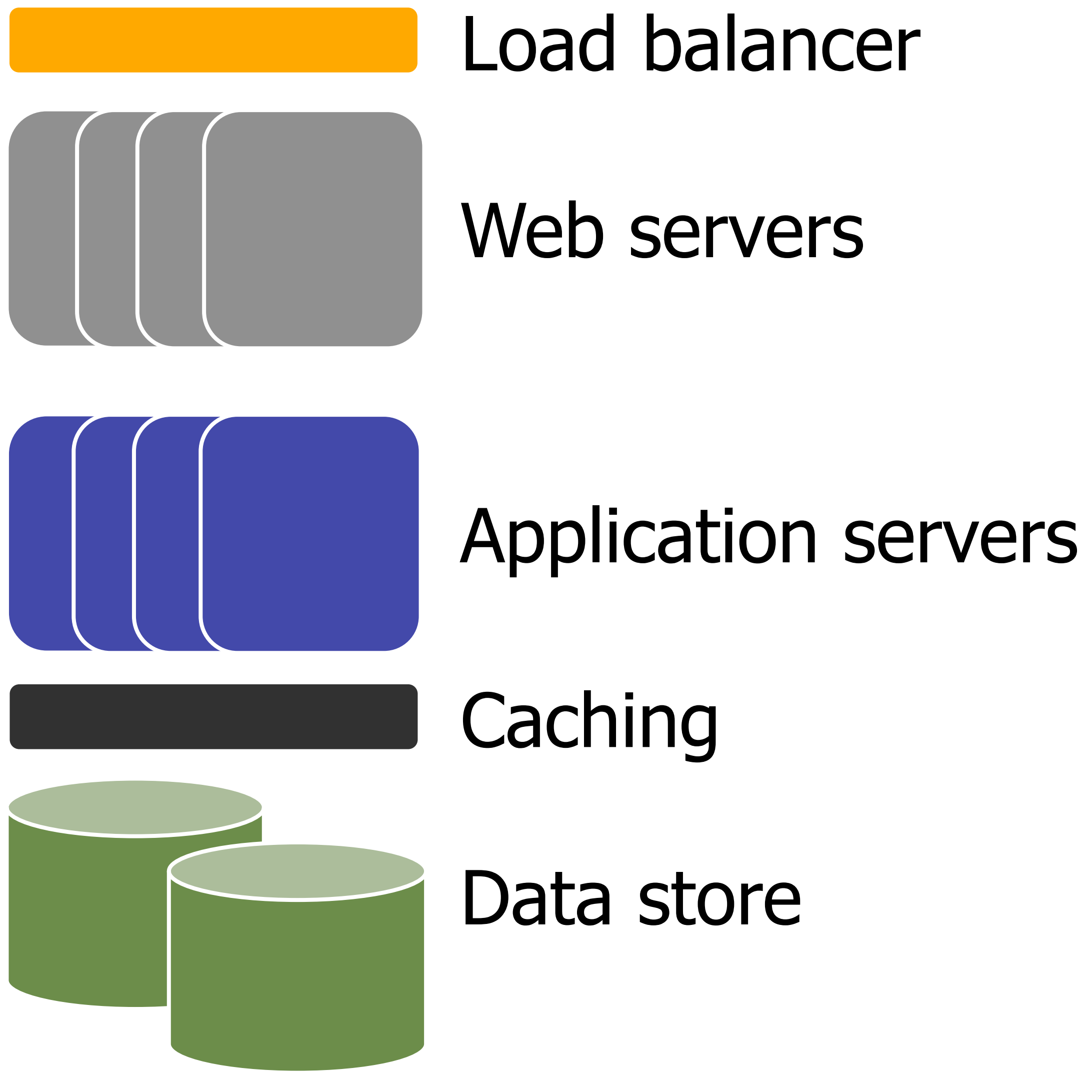
Stateless servers are easiest to scale

- Views a client request as an independent transaction and responds to it
- Advantages:
  - Simpler and easier to scale: does not maintain state
  - More robust: tolerating instance failures does not require overheads restoring state



Stateless servers

Caching

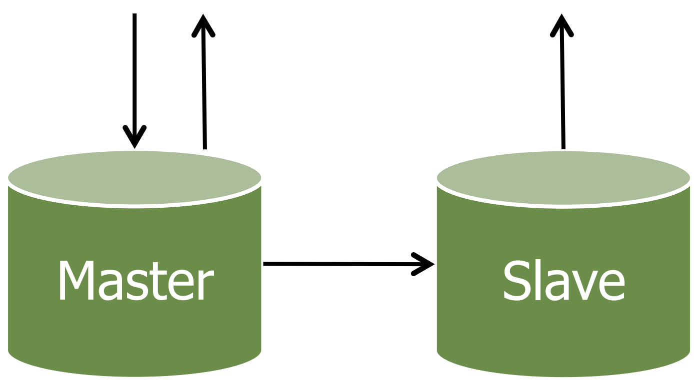

Load balancer
Web servers
Application servers
Caching
Data store

Caching

- Caching is central to responsiveness
  - Basic idea is to always used cached data if at all possible, so the inner services (data stores) are shielded from “online” load
  - Caching is only temporary storage, hence it is stateless
  - We can add multiple cache serves to spread loads
- Must think hard about patterns of data access
  - Some data needs to be heavily replicated to offer very fast access on vast numbers of nodes
  - In principle the level of replication should match level of load and the degree to which the data is needed

Stateful servers require attention

- Scaling a relational database is challenging
- Traditional approach is replication
  - Data is written to a master server and then replicated to one or more slave servers (synchronously or asynchronously)
  - Read operations can be handled by the slaves
  - All writes happen on the master



Data store

Stateful servers require attention

- Scaling a relational database is challenging
- Traditional approach is replication
  - Data is written to a master server and then replicated to one or more slave servers (synchronously or asynchronously)
  - Read operations can be handled by the slaves
  - All writes happen on the master
- Cons:
  - Master becomes the write bottleneck
  - Master is a single point of failure
  - As load increases, cost of replication increases
  - Slaves may fall behind and serve stale data

Sharding

- Data partitioning strategy
- Basic idea: split data between multiple machines and have a way to make sure you always access data from the right place
  - Typically define a sharding key and create a shard mapping (e.g., consistent hashing: shard_idx = hash(key) mod N)
  - Other partitioning schemes exist: e.g., allocate whole tables on the same machine

Benefits of sharding

- Increased read and write throughput
- High availability
- Possibility of doing more work in parallel within the application server

- Challenge: picking a good partitioning scheme
  - Otherwise risk of having hotspots in the system due to load imbalance

Sharding used in many ways

- Sharding is not only for partitioning data within a database
- Applies essentially to every application tier
  - Notion of sharding is cross-cutting

- Example: partition data across caching servers
- Two popular in-memory caching systems:
  - memcached: distributed object caching system
  - redis: distributed data structure server (also works as store)

And it isn't just about updates

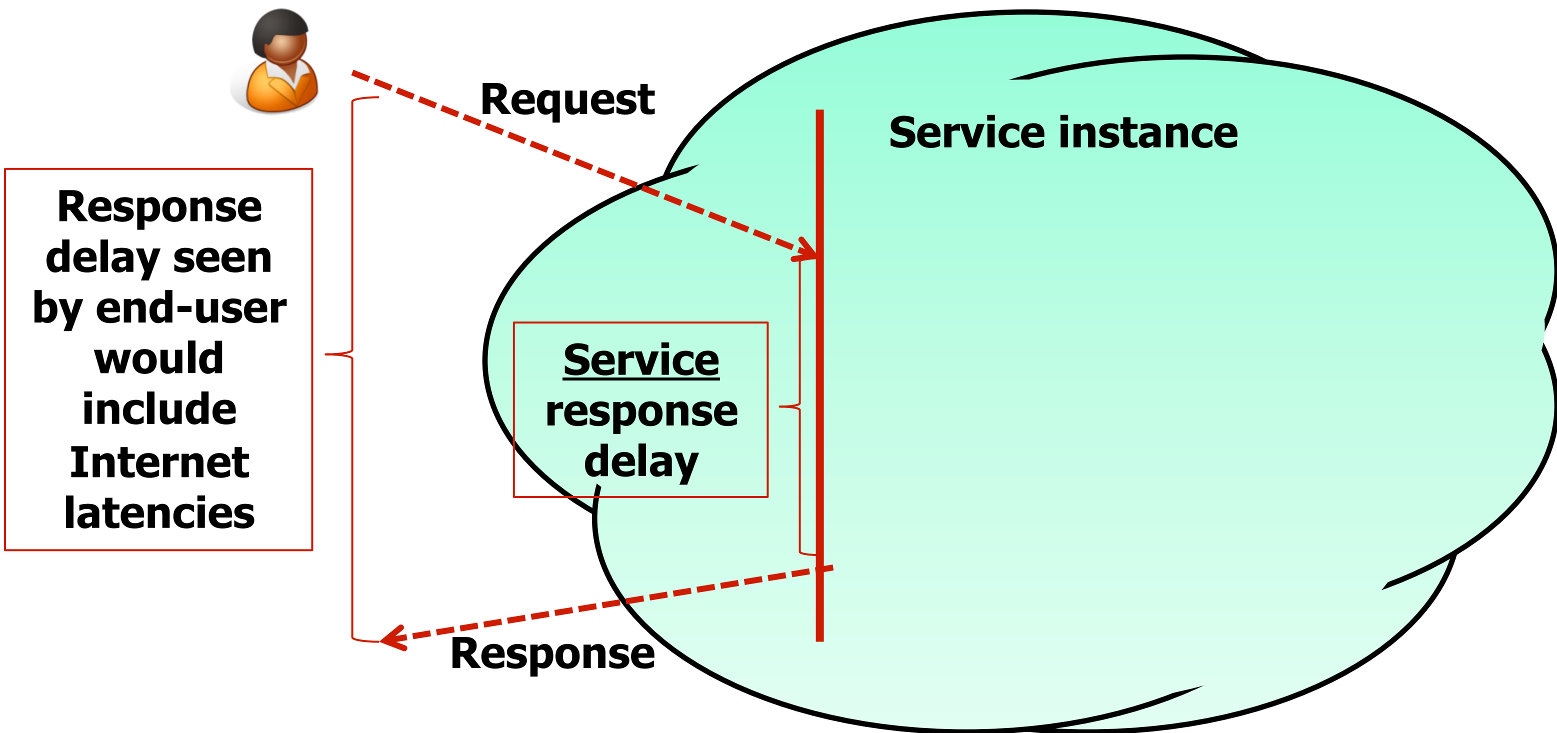
- Should also be thinking about patterns that arise when doing reads ("queries")
  - Some can just be performed by a single representative of a service
  - But others might need the parallelism of having several (or even a huge number) of machines do parts of the work concurrently
- The term sharding is used for data, but here we might talk about "parallel computation on a shard"

First-tier parallelism

- Parallelism is vital for fast interactive services
- Key question:
  - Request has reached some service instance X
  - Will it be faster…
    - … For X to just compute the response
    - … Or for X to subdivide the work by asking subservices to do parts of the job?
- Glimpse of an answer
  - When you make a search on Bing, the query is processed in parallel by even 1000s of servers that run in real-time on your request!
- Parallel actions must focus on the critical path

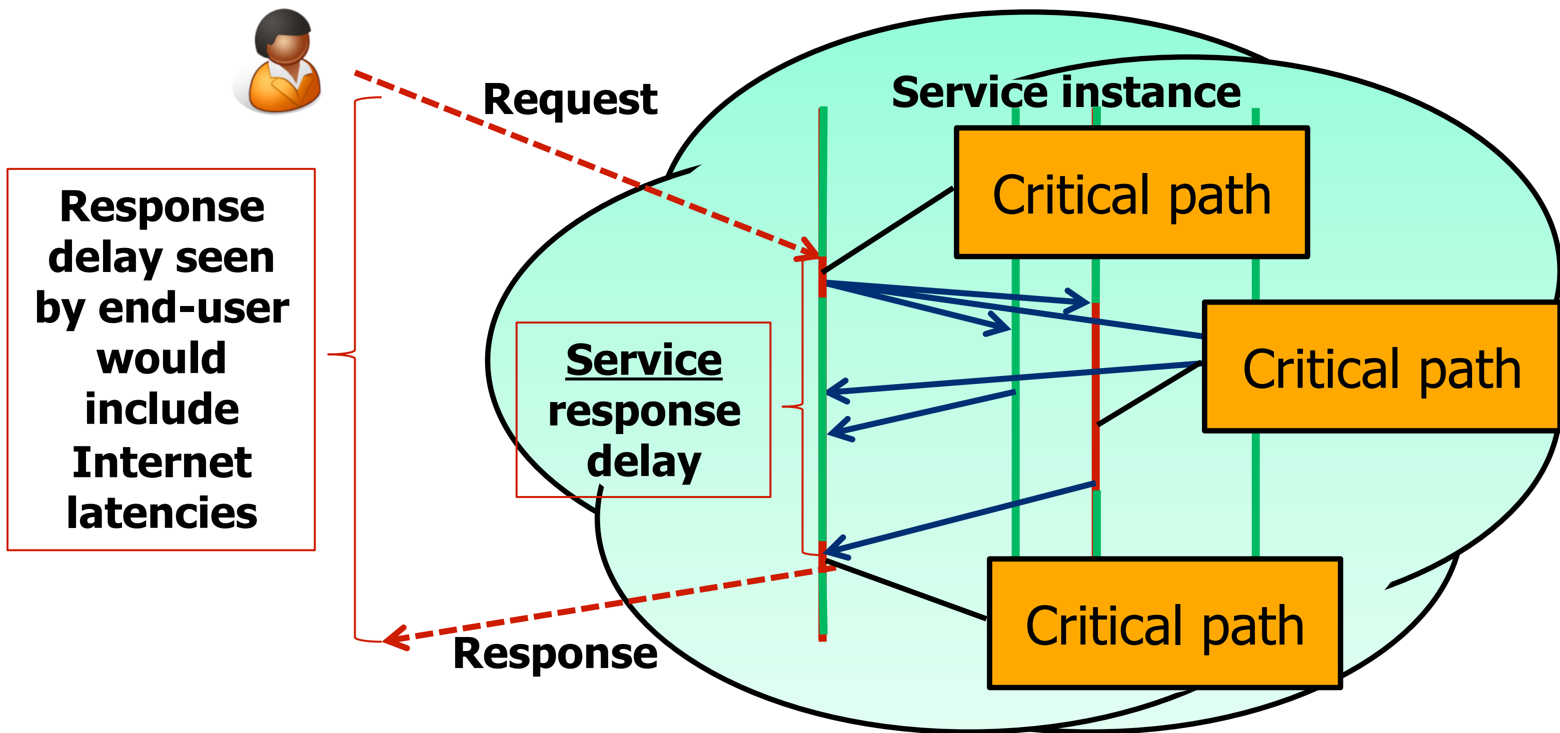
What does "critical path" mean?

- Focus on delay until a client receives a reply
- Critical path are actions that contribute to this delay

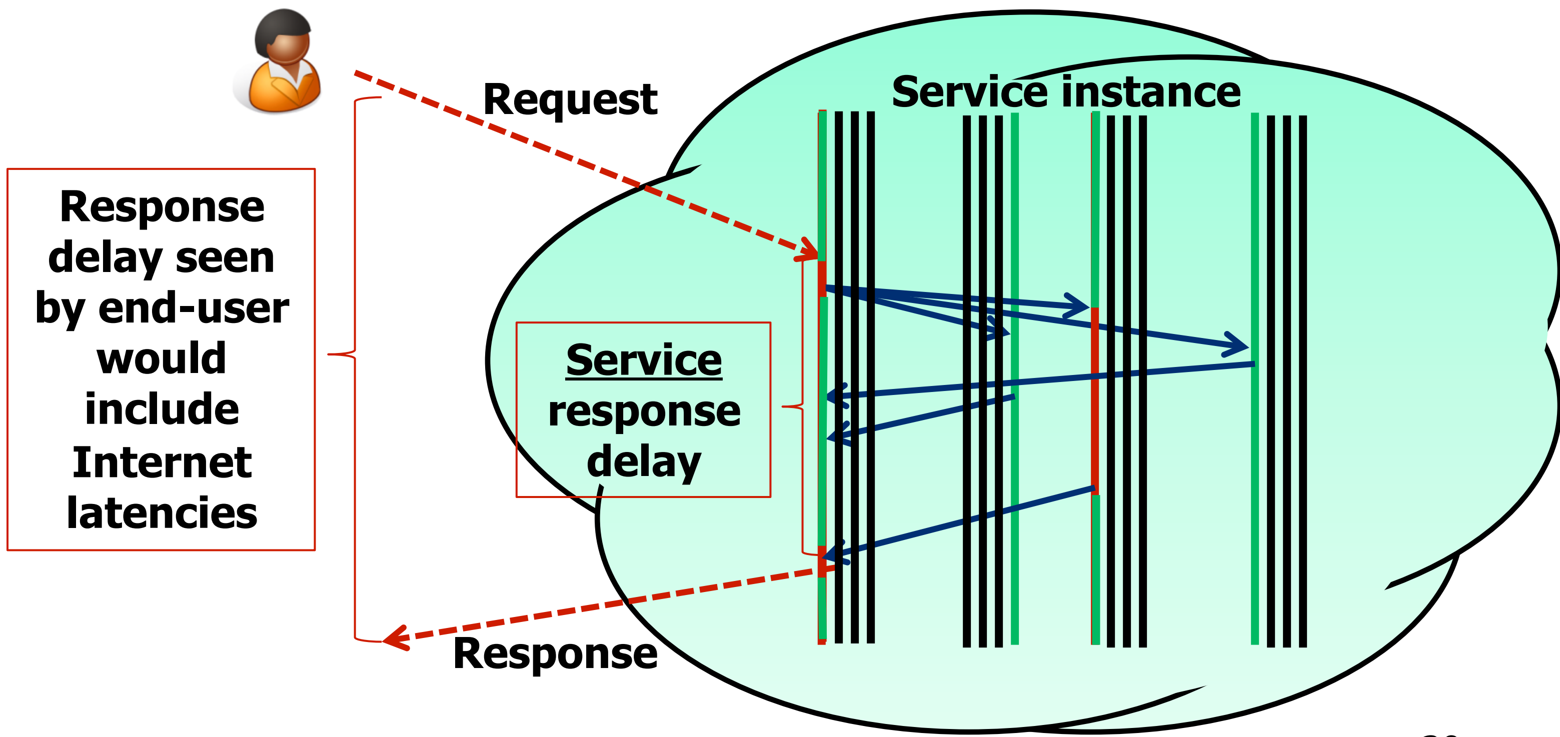


Parallel speedup

- In this example of a parallel read-only request, the critical path centers on the middle "subservice"



With replicas we just load balance



What if a request triggers updates?

- If updates are done "asynchronously" we might not experience much delay on the critical path
  - Cloud systems often work this way
  - Avoids waiting for slow services to process the updates but may force the tier-one service to "guess" the outcome
  - For example, store in the master database and replicate to the slave in the background
- Many cloud systems use these sorts of "tricks" to speed up response time

What if we send updates without waiting?

- Several issues now arise
  - Are all the replicas applying updates in the same order?
    - Might not matter unless the same data item is being changed
    - But then clearly we do need some "agreement" on order
  - What if the leader replies to the end user but then crashes and it turns out that the updates were lost in the network?
    - Data center networks can be surprisingly lossy at times
    - Also, bursts of updates can queue up
- Such issues result in *inconsistency*

Is inconsistency a bad thing?

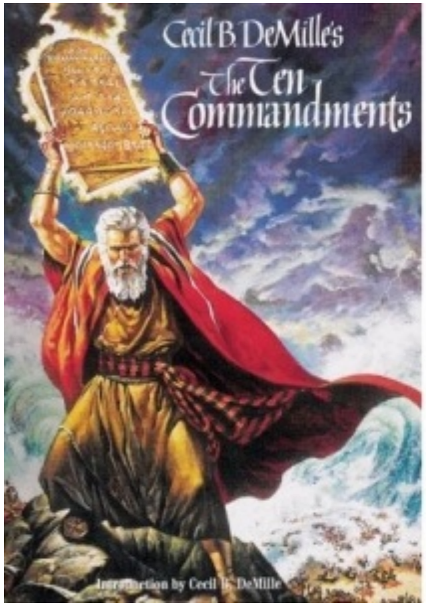
- How much consistency is really needed in the first tier of the cloud?
  - Think about YouTube videos. Would consistency be an issue here?
  - What about the Amazon "number of units available" counters. Will people notice if those are a bit off?
- Puzzle: can you come up with a general policy for knowing how much consistency a given thing needs?

eBay’s Five Commandments

- As described by Randy Shoup at LADIS 2008

Thou shalt…

1. **Partition Everything**
2. **Use Asynchrony Everywhere**
3. **Automate Everything**
4. **Remember: Everything Fails**
5. **Embrace Inconsistency**



Recap

- Cloud applications are multi-tiered systems
- Caching can enable significant speedups for read-heavy workloads
- Sharding provides opportunities for parallelization and improve read/write throughputs
- Asynchronous operations decouple systems and enable quicker responses at the expense strong consistency

Plan for today

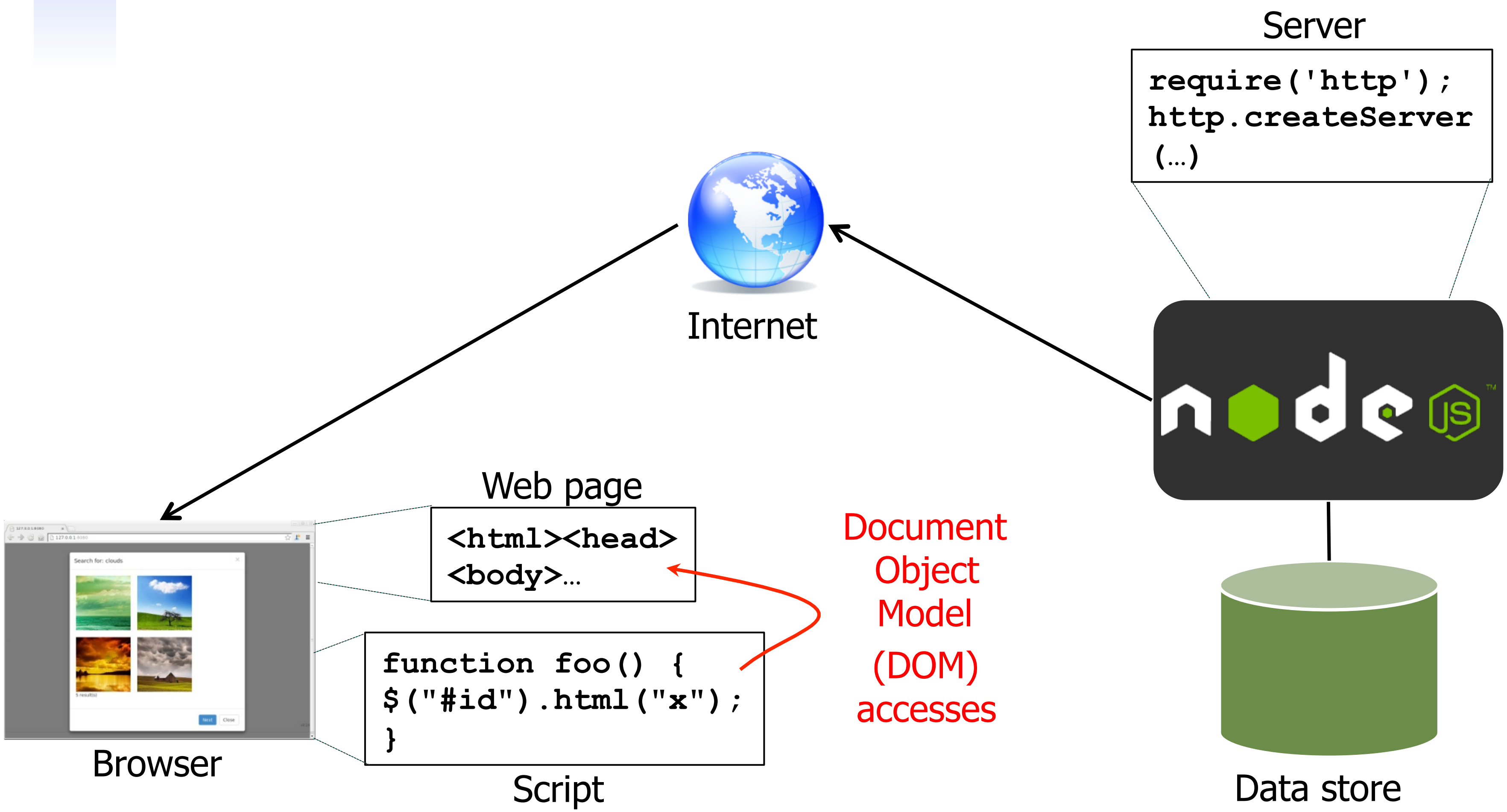
- Parallel programming and its challenges ✓
  - Parallelization and scalability, Amdahl's law ✓
  - Network partitions, CAP theorem, relaxed consistency ✓
- Cloud basics ✓
  - Anatomy of Cloud applications ✓
  - Scaling: stateless, caching, and sharding ✓
- Example components
  - Application server: Node.js ← NEXT
  - In-memory cache: Memcached
- Scaling memcache at Facebook

Application server

- Provides an environment where Web applications can run
- Many application server frameworks exists
  - Support different programming languages
  - At the core they support dynamic Web page generation
  - Differentiated by functionality and features (runtime libraries, database connectors, fail-over, load balancing, etc.)

- Let’s consider an example: Node.js

How does a Web application work?



What is JavaScript?

- A widely-used programming language
  - Started out at Netscape in 1995
  - Widely used on the web; supported by every major browser
  - Also used in many other places: PDFs, certain games, ...
  - ... and now even on the server side (Node.js)!

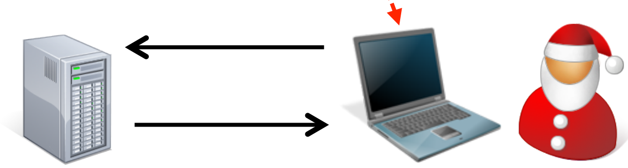
- What is it like?
  - Dynamic typing, duck typing
  - Object-based, but associative arrays instead of 'classes'
  - Prototypes instead of inheritance
  - Supports run-time evaluation via eval()
  - First-class functions

Running JavaScript in the browser

Uses DOM to change text on page

```
<html><head><script>
function update(){
document.getElementById("color").
    innerHTML = "Green";
}
</script></head><body>
<div id="color">Red</div><br>
<input onclick="update()" type="button"
value="Change color">
</body></html>
```

Event handler



- Web pages can contain JavaScript code
  - Example: Pop up a dialog box when user clicks a button
  - The code can receive user inputs (e.g., clicks) and produce outputs, e.g., by changing the web page in which it runs
    - This is done via the DOM (Document Object Model)
  - Not just a toy language: Entire applications are being written in it (think Google Apps!)

What is jQuery?

```html
<html><head>
<script src="jquery.min.js"></script>
<script src="app.js"></script>
</head><body>
<p id="test">This is some <b>bold</b>
text in a paragraph.</p>
<button id="btn1">Show Text</button>
<button id="btn2">Show HTML</button>
</body>
</html>
```

test.html

```javascript
$(document).ready(function(){
  $("#btn1").click(function(){
    alert("Text: " +
      $("#test").text());
  });
  $("#btn2").click(function(){
    alert("HTML: " +
      $("#test").html());
  });
});
```

app.js

- A lightweight JavaScript library
  - Makes many common functions, such as DOM manipulation or AJAX, much easier to implement (typically single line)
    - Examples: $("#id").html(), $("#id").click(), $.getJSON(), ...
  - Widely used (Google, Microsoft, IBM, Netflix, ...)

What is Node.js?

- A platform for JavaScript-based network apps
  - Based on Google's JavaScript engine from Chrome
  - Comes with a built-in HTTP server library
  - Lots of libraries and tools available; even has its own package manager (npm)

- Event-driven programming model
  - There is a single "thread", which must never block
  - If your program needs to wait for something (e.g., a response from some server you contacted), it must provide a callback function

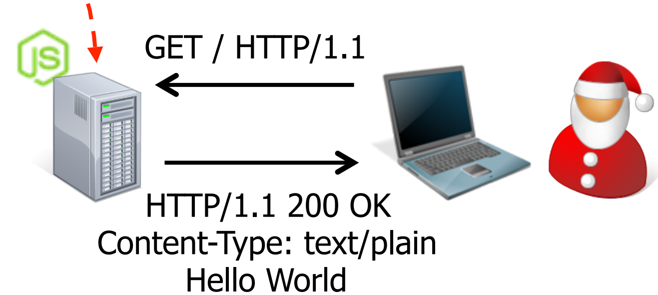
"Hello World" with Node.js

Callback function

```
var http = require('http');

http.createServer(
  function (request, response) {
    response.writeHead(200,
     {'Content-Type': 'text/plain'});
    response.end('Hello World\n');
  }
).listen(8080);

console.log('Server running' +
  ' at http://localhost:8080/');
```



- Uses built-in HTTP library to create a server
  - The server will listen on port 8080
  - createServer() is given a callback function that is called whenever someone requests a web page
    - Callback writes the required HTTP header followed by "Hello World"
  - To view the result, open http://localhost:8080/ in a browser

What is JSON?

"Object": Unordered collection of key-value pairs

```
{
  "firstName": "John",
  "lastName": "Smith",
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": 10021
  },
  "phoneNumber": [
    { "type": "home", "number": "212 555-1234" },
    { "type": "fax", "number": "646 555-4567" }
  ]
}
```

Array (ordered sequence of values; can be different types)

- A standard format for data interchange
  - "JavaScript Object Notation"; MIME type application/json
  - Basically legal JavaScript code; can be parsed with eval()
  - Often used in AJAX-style applications
  - Data types: Numbers, strings, booleans, arrays, "objects"

Calling the server

Status: Waiting...

Web page (in your browser)

```
$.getJSON('/search/' + $("#inputfield").val(),
  function(data) {
    $("#status").html(data.num_results+" result(s)");
  }
);
```

Client code (in your browser)

GET /search/clouds HTTP/1.1

{ num_results: 5, foo: 123 }

```
var express = require('express');
var app = express();
...
app.get('/search/:word', function(req, res) {
  var n = findResults(req.params.word);
  res.send(JSON.stringify({num_results: n, foo: 123}));
});
```

Server code (Node.js)

Plan for today

- Parallel programming and its challenges ✓
  - Parallelization and scalability, Amdahl's law ✓
  - Network partitions, CAP theorem, relaxed consistency ✓
- Cloud basics ✓
  - Anatomy of Cloud applications ✓
  - Scaling: stateless, caching, and sharding ✓
- Example components
  - Application server: Node.js ✓
  - In-memory cache: Memcached ← NEXT
- Scaling memcache at Facebook

Memcached

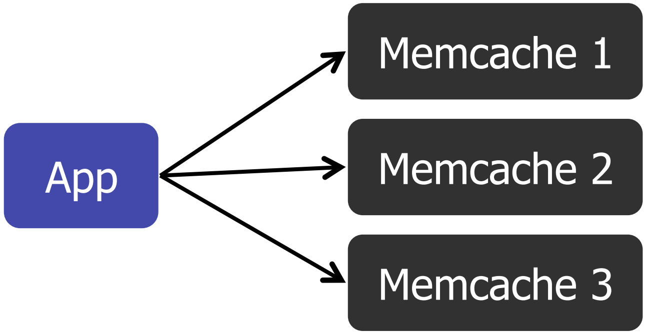
- Very simple concept:
  - High-performance distributed in-memory caching system that manages "objects"
  - Essentially a network-attached in-memory hash table
  - Implementations in many programming languages
  - Run as a distributed service implemented using a cluster of machines
- Developed by Brad Fitzpatrick for LiveJournal in 2003
- Now used by Facebook, Flickr, Twitter, Youtube, Wikipedia, Netlog, …

Memcached API

- Memcached defines a standard API
  - Defines the calls the application can issue to the library or the server (either way, it looks like library)
  - In theory, this means an application can be coded and tested using one version of memcached, then migrated to a different one

```
function get_foo(foo_id)
  foo = memcached_get("foo:" + foo_id)
  if foo is not None return foo
  foo = fetch_foo_from_database(foo_id)
  memcached_set("foo:" + foo_id, foo)
  return foo end
```

Memcached cluster



- Servers can run in pools
  - Example: 3 servers with 1 TB RAM each give you a single pool of 3 TB storage for caching (in principle)
- Servers are independent, clients manage the pool
  - Trivial approach just hashes the a certain key to a certain server
    - If a server goes down, all keys will now be queried on other serves
  - But this could lead to load imbalances, plus some objects are probably more popular than others
    - Would prefer to replicate the hot data to improve capacity
    - But this means we need to track popularity…

What to store in memcache?

- High demand
  - often used
- Expensive
  - hard to compute
- Common
  - shared across users
- Best? All three
- Example:
  - User sessions
  - Database results

Memcached principles

- Fast network access
  - memcached server close to application servers
- No persistency
  - if a server goes down, the data in memcached is gone
- No redundancy / fail-over
- No replication
  - single item in cache lives only on one server
- No enumeration of keys
  - thus no list of valid keys in cache at a certain time

Plan for today

- Parallel programming and its challenges ✔
  - Parallelization and scalability, Amdahl's law ✔
  - Network partitions, CAP theorem, relaxed consistency ✔
- Cloud basics ✔
  - Anatomy of Cloud applications ✔
  - Scaling: stateless, caching, and sharding ✔
- Example components ✔
  - Application server: Node.js ✔
  - In-memory cache: Memcached ✔
- Scaling memcache at Facebook ⇦ NEXT

Scaling Memcache at Facebook [NSDI '13]

https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala

Stay tuned

Next time you will learn about:

A programming model for the Cloud