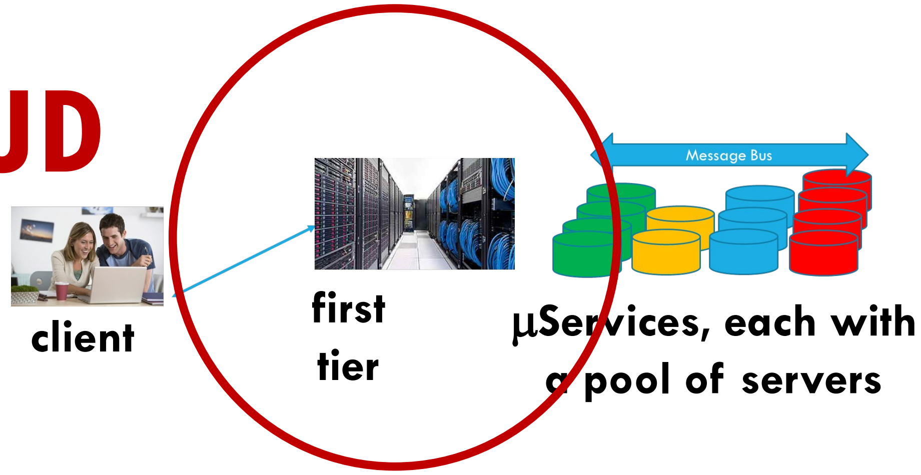
CS 5412/LECTURE 3: MORE CLOUD ARCHITECTURE DETAILS

Ken Birman
Spring, 2022

RECAP: LIFE-CYCLE OF A CLOUD INTERACTION



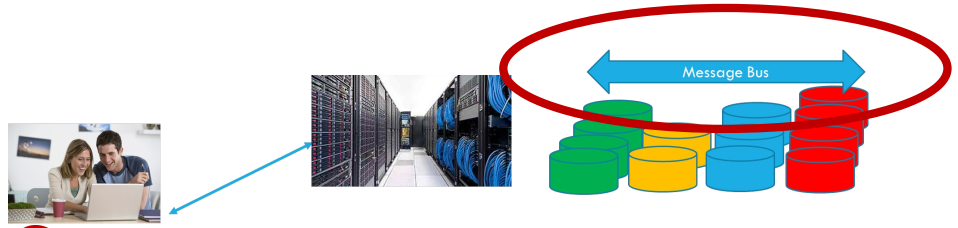
Client system connects to the cloud, makes a request

Outer tier relays the request to a logic layer where pools of compute servers (µServices) handle distinct aspects.

- Like a “modular” decomposition of a big program.
- They rely on a deeper tier with standard vendor-supplied µServices for tasks like key-value storage (DHT), image storage/compression/resizing, etc.
- They often use hardware accelerators

Outer tier reassembles the result and streams the reply back to the client.

TOOLS FOR RELAYING REQUESTS TO THE µSERVICES

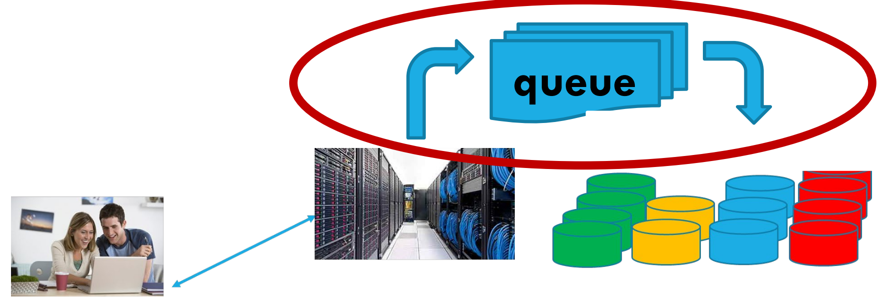


GRPC: **remote procedure call.** *Client must know the Server IP address.*

- This is a limitation: Often, a first-tier component won't know.

A **message bus** automatically tracks the members of the pool.

- You can ask for your request to go to any single member, or to all. Later the member that picked the request up can reply.
- If a timeout occurs (like because the member crashed), you reissue it.
- Sometimes called a publish/subscribe bus, or a data distribution service



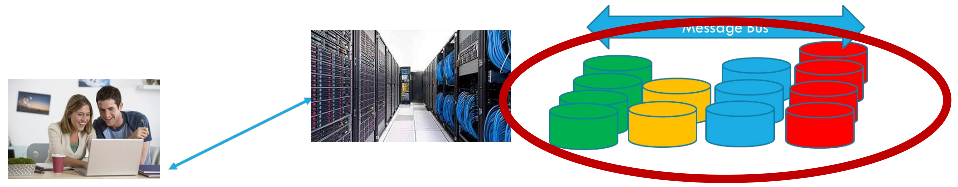
MESSAGE QUEUE

A **message queue** is more like an email system. Your request has a message group address (like an email address), and is stored under that folder. **Kafka** is the most famous message queue product.

A member of the μService pool can ask for the next "unread" message, delete a message, reply to the sender, etc.

- For better efficiency, many applications read a batch of messages, all at once, from the same group: "all pending messages", or "next 100"
- Batched processing reduces overheads of talking to the bus again and again

APP SERVICE



We saw that each μ-Service is managed by the "App Service", which controls how many instances are launched, when to launch/kill members, and monitors overall load.

Job is to (1) monitor the load on μ-Service nodes, and (2) detect overload cases, such as backlog developing. Then (3) grow the pool of replicas for that μ-Service by launching new copies – new container that hold instances of the program implementing the μ-Service.

BUILDING CODE TO RUN IN THE "POOL OF SERVERS" MODEL

Recall from lectures 1 and 2 that it can be much easier and more flexible to write a single program that isn't heavily multi-threaded, and have each request processed by a single program instance.

We can scale our service out by just running it many times

STATELESS VERSUS STATEFUL PROGRAMS

A newly launched program has some initial state:

- Variables you initialized
- Files on the local disk that it reads as input

In the cloud, we call this *local* data and the *local file system*.

The cloud also has a *global* cloud file system. You talk to it via messages.

- The cloud uses Google GRPC, or the Azure/AWS equivalent

THE STATELESS MODEL

Stateless guy

What does "state" mean to you? For a single

- Data in program variables, in memory, used
- Data that the program reads <u>but didn't cr</u>
  - Information in the Linux "environme ''
  - Data in parameter files or conf ation f
  - Data received in a reques om the client.
  - Data in files or a data ase that the program only reads.
- Data the program <u>does create, or modify</u>.

A stateless program actually <u>can</u> update local state (nothing stops it). But the updates will be lost.

STATELESS MODEL

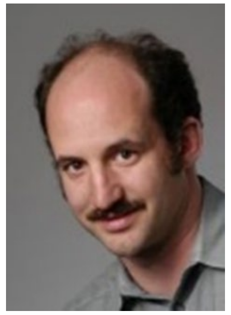
So… a stateless program is a *normal* program, written the way you write any computer program.

… but it follows one rule: no *updates* to "persistent" data (that would still be there if I shut it down, then restart it) on the computer where it runs.

- It still has "state" in its variables and so forth. Not an issue.
- But if it needs to save something, that update has to be sent to some other place, like into the cloud's global file system, or a database, etc.

THE FIRST TIER OF A CLOUD IS "STATELESS"

This design idea dates to Professor Eric Brewer, at Berkeley



He observed very scalable pools of μ-Services are easier to build and extend if the data used is all read-only.

- Sometimes he called this "soft state", meaning the "hard version is elsewhere"
- But most people just call this a stateless model.

A stateless server is easy to shut down (kill it, throw away any files it created).

WHICH CAN WE DO IN A STATELESS WAY?

Consider the Amazon shopping web pages. We can build these by looking up data held in other services.

- **A μ-service that tracks your purchase history.**
- **A μ-service that computes recommendations.**
- **A μ-service that resizes images of products to fit your screen**
- **A μ-service that manages the shopping cart**

WHICH CAN WE DO IN A STATELESS WAY?

Consider the Amazon shopping web pages. We can build these by looking up data held in other services.

- **A μ-service that hosts (keeps) your purchase history.**
- **A μ-service that computes recommendations.**
- **A μ-service that resizes images of products to fit your screen**
- **A μ-service that manages the shopping cart (and remembers the list)**

OFTEN WE PAIR A STATELESS FRONT END WITH STATEFUL μ-SERVICES

Stateful services are harder to build, so usually vendors like Oracle (the database company) or Amazon or Microsoft do that for you.

The cloud file system is the most obvious example. Others include the image storage service (in addition it usually can resize photos, segment them, perhaps even recognize who is in them), databases, DHTs, etc.

Then we might build a stateless service that sits in front and adds some of our own logic but relays anything that needs to be saved into the stateful tier.

SOME IMPORTANT STATEFUL SERVERS IN THE CLOUD

Azure: The global file system, the Cosmos Database, the BLOB store. The nessage queue service. Various DHT products, like Cassandra

AWS: Many of the same options, plus DynamoDB, Amazon's scalable DHT

Note: Even though these may use the term "database", and you access them with SQL, Cosmos and Dynamo are not true databases. The model is "NoSQL" (meaning "This is not transactional SQL")

HELPFUL IDEAS FOR STATELESS DESIGN

Use a scalable DHT as a cache for any data the program reads. With luck we will get a good hit rate (like in Facebook) and this will shield the big stateful systems at the back-end from most of the load.

A DHT can hold much more stuff than any single computer could ever hold.

Be tolerant of staleness. This a high cost to be sure our cached data. For example, if a site says “2 Xbox Series X units left at this price”, that could be a bit stale… I wouldn’t notice.

KEY IDEAS FOR STATELESS DESIGN

Soft state can easily be regenerated, so don’t worry much about fault tolerance.

In our DHT examples last week, we talked about state machine replication for fault-tolerance, but this involves using atomic multicast or Paxos for updates, to ensure that all replicas see the same update sequence.

If we don’t care about losing data during a crash, we can skip that step.

- The real data would live in a back-end database or file system.
- Since we are only keeping cached data in the DHT, if a shard gets amnesia, we can always fetch it from the back-end system a second time.

DEFINITIONS (SLIGHTLY INFORMAL!)

Consistency: The system responds using *the most current values* (updates). *Conflicting updates are performed in some system-selected order by all replicas.* Queries thus will see a "single system" and will be up to date.

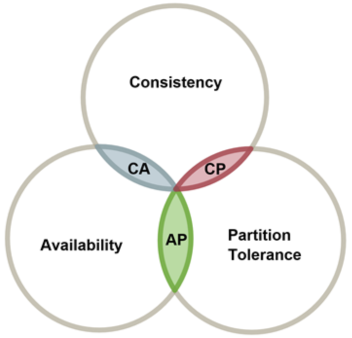
Availability: The system is *rapidly responsive*, and will *self-repair* if some single component fails, restoring normal functionality asap. Of course fault-tolerance isn't always possible: if too many components fail all at once, availability is lost.

Partition Tolerant: A data center can have network issues, or entire services can be down. Yet *as seen from outside, the cloud should continue to respond even if its first-tier services are temporarily unable to talk to some inner services* they would normally depend upon.

IN THE CLOUD, NOT EVERY SUBSYSTEM NEEDS THE STRONGEST GUARANTEES

At Berkeley, Eric Brewer captured this insight with a “theorem”

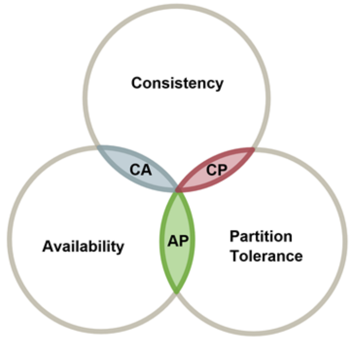
CAP is short for “Consistency, Availability and Partition Tolerance”



Basically, Eric argues that:

- The theoretically “best” solution often brings heavy costs.
- Consistency is one example: conflicting database updates can be forced into an agreed order, but this takes time and involves node-node dialog (hence ¬P).
- Remember that to earn the most money, you need the fastest possible responses. Eric concluded that this means you might have to relax consistency: **~~C~~AP**.

ERIC BREWER'S CAP THEOREM



Claim: A system can only have 2 out of 3 from CAP.

What to do?

- Relax consistency (**C**),
- Gain faster response (**A**).
- Generate responses even when unable to talk to back-end servers (**P**).

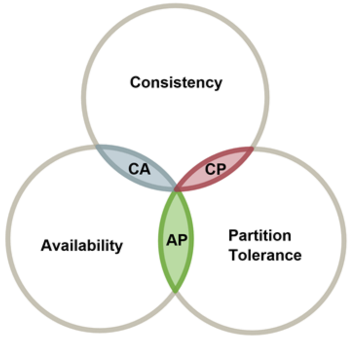
… ONE TINY NIT

The theorem isn't actually true. You actually can have all three at once.

How? As we will see, you need to have a stateful cloud (even the outer tiers), using consistent replication for fault-tolerance and availability. A Cornell tool called Derecho assists for this.

Call CAP more of a folk theorem: A useful rule of thumb.

BASE METHODOLOGY: GOES WITH CAP



BASE ≡ "CAP in practice"

Invented at eBay, adopted by Amazon and others

- **B**asic **A**vailability, **S**oft State and **E**ventual Consistency

"Use CAP. It may cause inconsistency.

Clean up later."

How BASE works: cache data but don't worry about cache entries getting stale (hey, they were valid a little while ago).

TODAY'S CLOUD IS A CAP+BASE "WORLD"

By and large, cloud systems manage with stale data / weak consistency.

Most applications are read-only and are fine with slightly stale data.

In IoT, though, this will change. When we look at IoT we will need more.

A STATELESS SERVICE IS JUST A POOL OF STATELESS PROGRAMS

- You upload a program, and the configuration data and parameter files.
- You tell the cloud App Service how many instances to launch and when to adjust the pool size. It knows how to monitor the request queue.
- The App Service will dynamically launch or kill your servers as needed.

It manages a collection of computers on which these servers can be launched

BUT INSTALLING PROGRAMS IS TRICKY!

Often, even if you don’t think of it, a program has a lot of *dependencies*.

- The runtime system of the programming language you picked.
- Other specialized libraries you may have downloaded and used.
- You may have trained a machine-learned model, and the program might need the model, the hyperparameters, and the trained parameter set to operate correctly.
- The program may only be able to run if other μ-Services on which it depends are already running. We call these *bindings*.

WHY DOES THIS POSE A PROBLEM?

It means that our App Service will be managing some pool of μServices. It needs to launch 5 more instances of "myService" (the one you wrote)

- It has to first pick 5 suitable servers (enough memory, maybe they need accelerators, maybe they have to be on machines that don't share the same power supply…)
- It has to copy your program and these dependencies to it.
- It has to verify that the bindings don't require launching additional μServices, and launch them if needed.
- Now it can start your program – your servers – by launching your code with any arguments you told it to pass in.

ISSUES THAT ARISE

How do we “tell” all of this to the App Service?

- You use a little configuration management application. It outputs a file in a format called JSON, and you can later tweak that file.
- All of the things the App Service needs to know go in the file.
- At the same time, it needs to bundle your program up in a convenient form for copying to a machine, and installing.

ISSUES THAT ARISE

There are dozens of popular versions of Linux, plus different revision levels of everything. You can't necessarily run a program that was built on Ubuntu Bionic Beaver (18.04) on a machine running CentOS 6.10

Package installation isn't always trivial and might require some kind of licensing.   Sometimes you need to rerun the install script.

Even installing the "most current" release isn't automatically a safe choice!

SO, HOW CAN WE PACKAGE A PROGRAM AND THE THINGS IT DEPENDS ON?

A common approach, in the early days, was to use a *virtual machine*.

This is a technology for making a snapshot of a computer system, as a single file, and then later running the snapshot on some other computer, perhaps one that doesn't even normally support that same operating system. A VM *Microkernel* or *Hypervisor* translates VM requests to whatever the actual computer supports.

In the early days, the main goal was back-compatibility for versions and for operating systems.

TODAY: EXAMPLES OF VM PLATFORMS

Many people use Oracle Virtual Box. It support "true" VMs. The VM mimics an entire computer. You pick the OS.

This is easily installed, and once you run it, you can create VMs, load ones your friend gave you or package one to send to a friend, etc.

Powerful, easily used and quite flexible.

MAIN ISSUE IS SLOWDOWN

VMs can usually run plain old code at the full speed of the computer. So individual cores don’t slow down.

But generally, system calls like file I/O pay a cost, and sometimes this cost is quite high.

VM approaches also use a lot of memory, making them expensive in $$ terms too.

CONTAINERS

This model emerged a few years ago in a project called Docker.

The idea was to run a normal Linux system, but to have individual processes see a virtualized world in which it would look as if they were alone on a VM.

Today, there are other solutions, notably Kubernetes, that are widely popular and used more commonly than Docker. Kubernetes is "cluster aware", while Docker and Virtual Box are "single machine oriented".

WHY DO CONTAINERS WIN?

Containers actually map everything to the same operating system and platform, so there is no need to emulate (for example) a Windows OS API on a Mac OS 10 computer.   The running program pays no overhead when it issues a file system call, or sends a message, or allocates memory.

The operating system can more easily notice that containers have some identical pages, and will avoid duplication, so less memory is wasted.

The delay to launch a new container is much smaller than for a new VM.

SO… IN OUR APP SERVER MANAGED POOLS

Each pool corresponds to:

- An application "defined" by a JSON file
- The file pointed to a container image, and the App Server copied that to machines where it might sometimes want to launch your server.
- At runtime, it needs to run 5 copies of myServer. It picks 1 to 5 physical machines and launches your container on them, 5 instances in total.
- Are we done?

MORE ISSUES THAT ARISE

What about the network, or files that get created?

Your program may need a way for servers to talk to one-another. How will that work?

And if your programs create files, where do those go if the App Server shuts down some instances?

And what if two containers clash in the way they operate?

… MANY DETAILS!

Kubernetes is normally told to create what looks like a private network just for you. Your applications sees a single network address space.

Kubernetes hides unrelated containers from one-another. They won't notice that they are sharing a computer.

But local files will typically be deleted when the container is shut down. A stateless design helps: those aren't allowed to hold hard state!

OTHER USES OF CONTAINERS

When we look at IoT programming, we will see that in IoT settings, we often want to launch a lightweight container to handle events that originate at sensors, like cameras.

This is done in something called the “Function Server”. It is a lot like the App Service, but focuses on event processing.

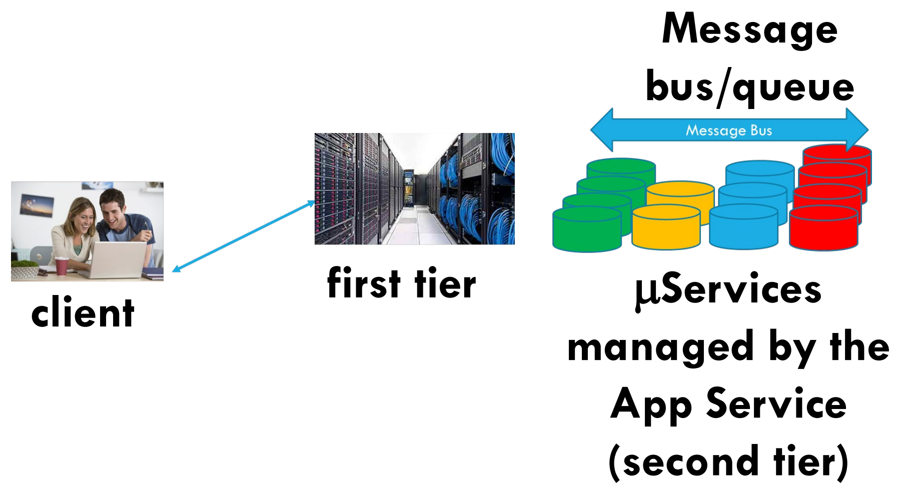
FLASK FOR PYTHON

Flask is an example of a container useful in PaaS cloud solutions and in IoT applications. It emerged from work on using Python in Web applications

It runs a small Python program in a container with a way to talk to web services. Flask is easy to "plug into" many Azure PaaS platform components, such as CosmosDB, Blob service, IoT Hub, etc.

Many students use Flask+Python as part of their CS5412 projects

SUMMARY



We revisited the basic cloud model

We didn't discuss the client platform itself, or the Internet, or load-balancer

But we saw how the first tier receives a request, relays it to one or more µServices via a message bus or queue, and how these pools are managed. We also learned that the programs are typically packaged in containers, like docker to reduce the risk of interference/conflicts, with low overheads.