CS 5412/LECTURE 3: MORE CLOUD ARCHITECTURE DETAILS

Ken Birman

Fall, 2022

LET'S REVISIT OUR "WALK THROUGH"

How does a typical interaction with the cloud work?

First, the user does something on their browser. Of course, it might be an app. Under the surface an "app" is really a kind of browser

NEXT STEP: MESSAGES!

The app binds to the cloud where its servers live by making a TCP connection. It will reuse the connection over long periods, so this step only imposes costs the first time.

On the other hand, some apps might need to talk to two or more sets of servers, and they would need one binding per server.

COOKIES!

To identify the user, a cloud system often requires that the application include a “cookie” with each request

The cookie normally just holds a unique id, but could be a whole file with lots of data about you

For security, the rule used is that when an app talks to some domain, it can only access the cookies created by/for that same domain

NEXT STEP: MESSAGES!

So, the app can send a message holding the request, plus the cookie.

The request generally looks like a method call in the code in the app. But in fact a "stub" packages ("serializes") the arguments into a message (a vector of bytes). The message is sent over TCP and the first-tier server derserializes (unpacks) the request to perform it.

The reply message will come back on the same TCP connection.

μ-SERVICES GALORE!

The first-tier server builds a web page (with URLs for any photo or video or advertising content), but spreads any real work out over the pools of service instances we are calling μ-services.

This entails doing more requests, as messages. They might not be issued as RPC requests over TCP – some could be sent on a "message bus" or a "message queue". But these are all forms of communication tools, just specialized in different ways.

μ-SERVICES GALORE!

Some server instance in the μ-service becomes responsible for this request

The original request was “inline roller skates for a 12-year old boy”, but that was the version seen by the first tier.

This particular μ-service instance might be a specialist on “popular products matching the search” or “special deals to offer” or “customers like you also look at …”, etc. It has a specialized sub-role.

PARALLELISM

There was no real parallelism when the original message was sent from the app to the first tier.

But there could be *hundreds* of μ-services running concurrently, generating portions of the data used to construct the web page response!

This gives a big speedup compared to doing everything in one server

PERSONALIZATION

Recall that all of these instances are stateless.

But also remember that they do have the original cookie giving the account ID information. This allows the μ-service to create a personalized reply.

The real data resides in a key-value storage layer, or a file system, or a database system. The key-value model is popular.

TOOLS FOR RELAYING REQUESTS TO THE µSERVICES

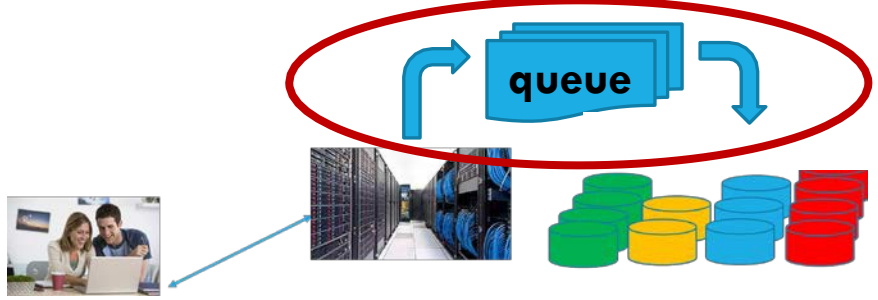
GRPC: **remote procedure call.** *Client must know the Server IP address.*

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

A **message <u>bus</u>** 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. In fact it can even be used as a message bus, by changing configuration parameters.

A member of the µService pool can ask for the next "unread" message, reply to the sender, etc. Kafka is designed for scaling: a message topic will automatically be sharded into a set of message pools that are accessed separately.

- 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

What does "state" mean for you? For a *program* it could mean:

- Values of variables inside the program as it runs
  - Constants as well as dynamically allocated variables/values
- Data that the program needs to read, but won't change
  - Such as data extracted from the event that triggered the program to run
  - Or data it loads from some sort of file or database
  - Or data read from the O/S like "today's date and time"
- Data that the program might actually have to modify
  - It could live in the "environment", or the "Windows active directory"
  - It could live in a server, such as a file server or a key-value store or a database

THE STATELESS MODEL

This model still allows any form of data to be saved in the storage tier

A **stateful program** would also hold data in memory, or on local files on the server where it runs, that isn't "backed up" elsewhere.

If a stateful program was shut down, we lose access to that information. And because a crash can delete files, we might permanently lose it.

Stateless programs don't hold this sort of "unique, non-backed up" data that would be needed when handling new web requests

STATELESS MODEL

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

… but it follows one rule: don't hold "persistent" data (that would still be needed 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.

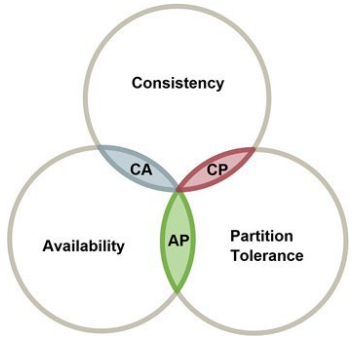
Availability: The system is *rapidly responsive*, and will *self-repair* if some single component fails, restoring normal functionality asap. Note: if too many components fail all at once, availability is lost.

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"

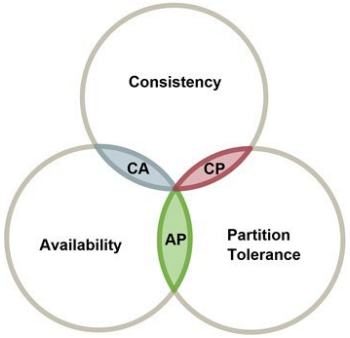
CAP is short for "Consistency, Availability and Partition Tolerance"



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: **¢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**).

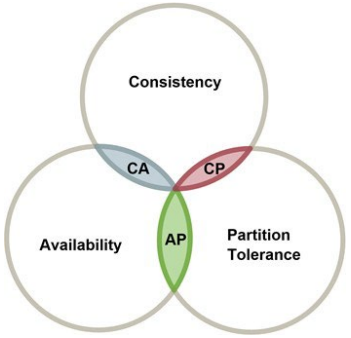
… ONE TINY NIT

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.

… AND A KEY-VALUE WORLD

The most common form of storage is a sharded key-value store

Benefit: Really, a KVS is just a file system, but one optimized for "whole objects" rather than an array of records represented by byte-vectors"

Databases are important too, but we will see that the cloud tries to shield them from load by putting big KVS caches in front of them.

KEY-VALUE MODEL

Basically, like a file system but where the data is read or written all at once, as if version 5 "replaces" version 4.

The "key" is just the word we use for "file pathname". As if the key-value store was really a file system. Exactly the same idea. They even look like file system pathnames, starting from the root ("/")

The "value" is just a vector of bytes. The key-value store doesn't care about the actual *meaning* of those bytes. API: **put, get, watch**

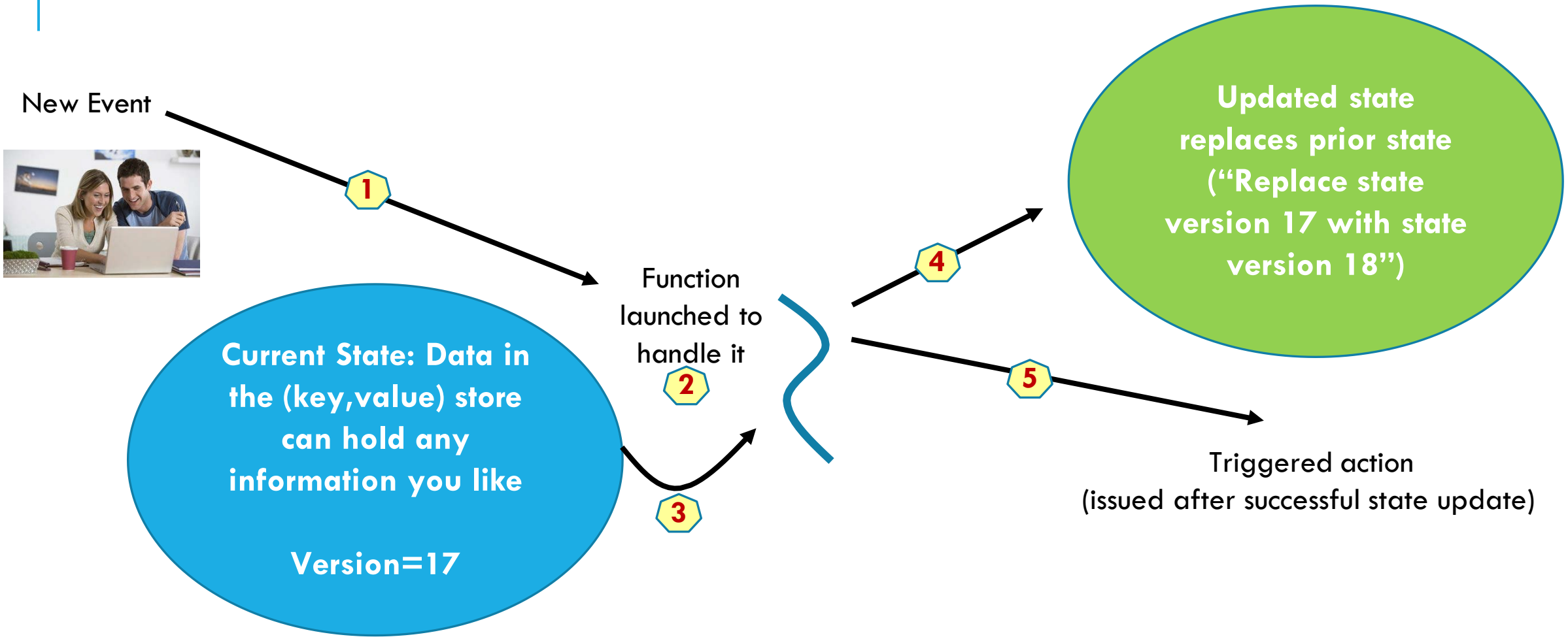
WHAT MAKES ALL OF THIS HARD?

Weak consistency. Many layers of the cloud cache data, and many actions are taken using potentially stale information.

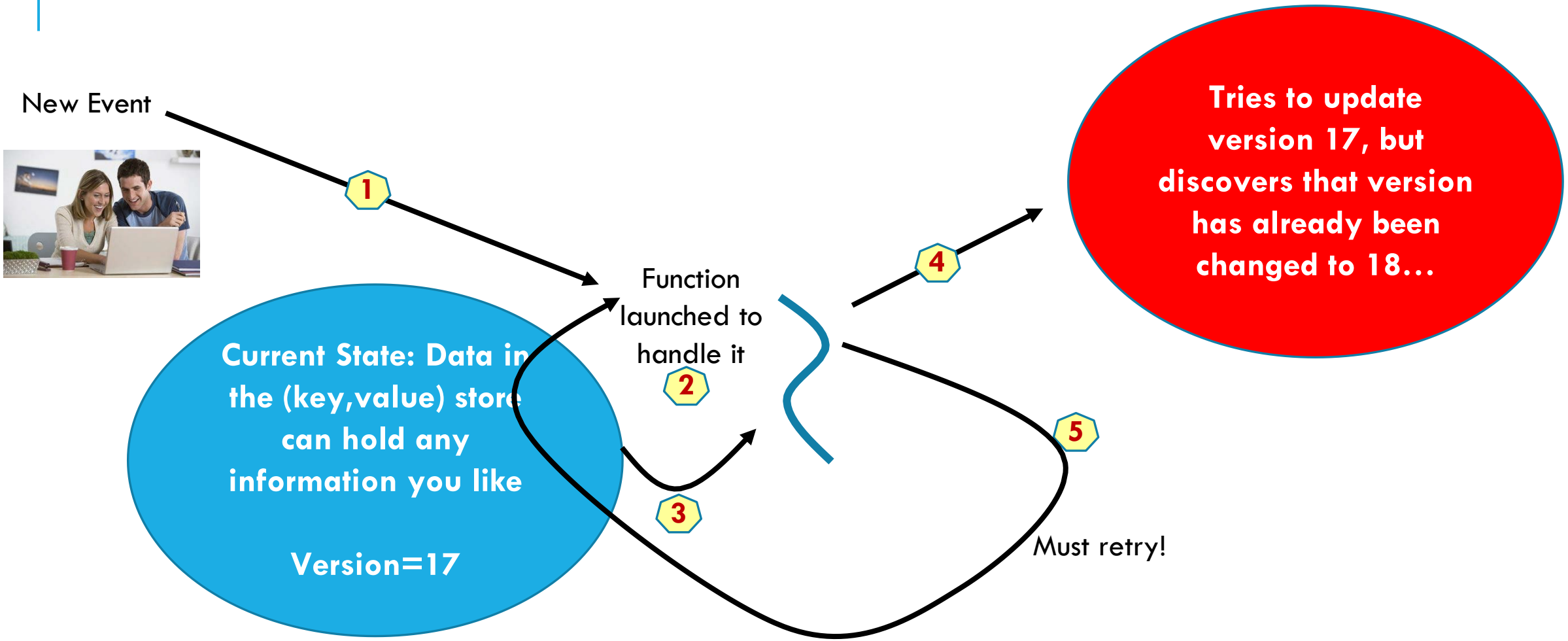
No locking. We don't have what database systems refer to as transactions. Each key-value operation is done all by itself, without locking.

But there is a kind of versioning that can be used as an alternative to locks

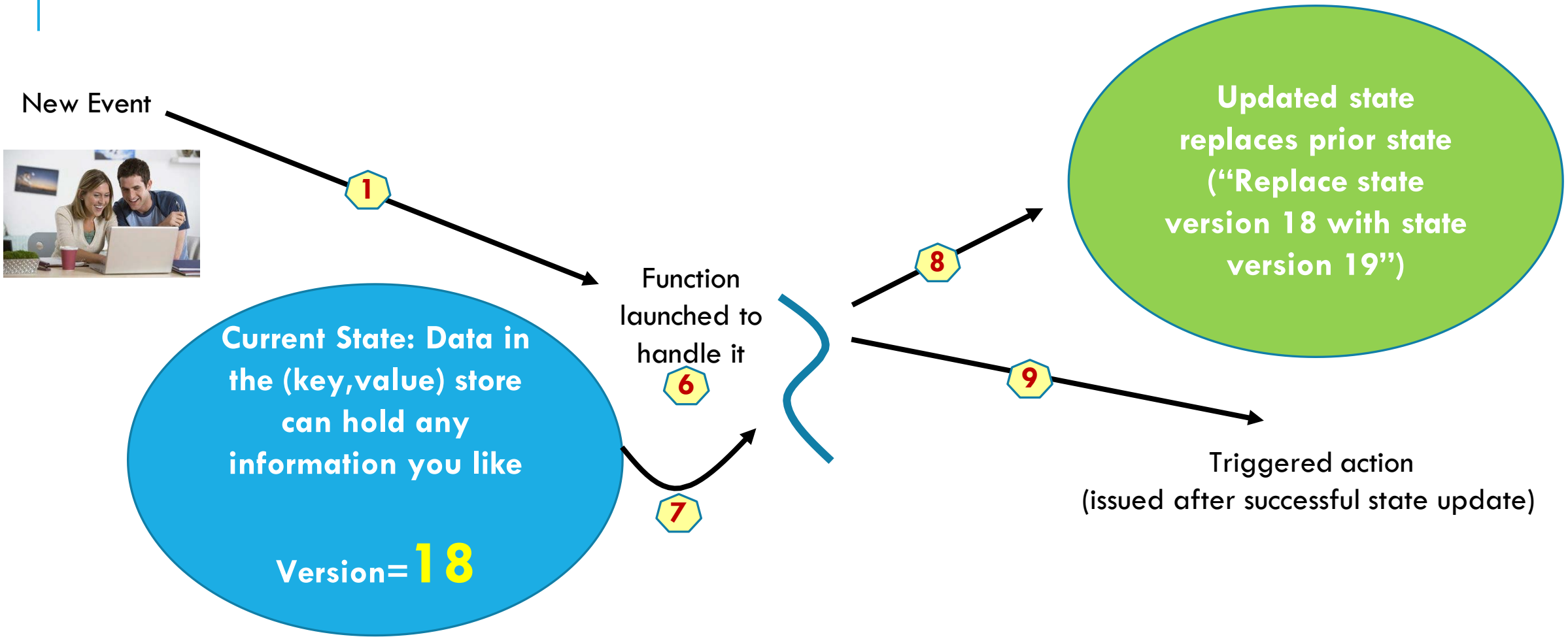
ATOMIC REPLACE WITH A STATELESS FUNCTION



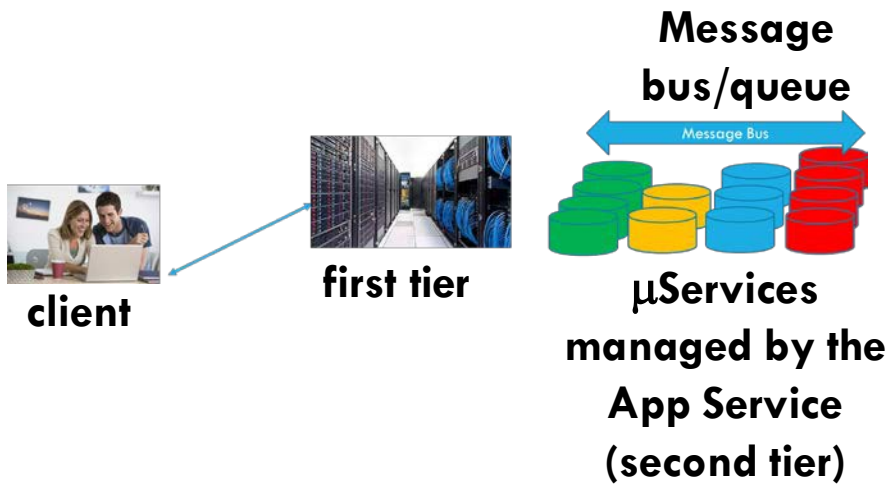
ATOMIC REPLACE WITH A STATELESS FUNCTION



ATOMIC REPLACE WITH A STATELESS FUNCTION



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.