CS 5412/LECTURE 3 PROGRAMMING AN IoT SYSTEM

Ken Birman

Spring, 2019

HOW DO WE "PROGRAM" THE IoT CLOUD?

This is a very rapidly evolving and exciting question!

To give some context, let's start with the same question as of 2005, to understand the answer today, but also the underlying reasons.

But the quick summary is: a few things work well, but many don't. We need to stick to techniques that industry is prioritizing and supporting.

PROGRAMMING PRIOR TO 2005

In the early period of cloud computing, there was a (mistaken) tendency to view the cloud like a very big distributed computing system.

So people learned to program on smaller clusters, then joined companies like Amazon, Yahoo!, eBay, and so forth, and took this knowledge along.

But in fact at cloud scale, those smaller techniques don't work well!

BIG WEB COMPANIES AROUND 2005

To maximize concurrency, they started by spreading the work of building web responses over a set of side-by-side tier-one services.

The next challenge was to optimize the μ-services. Remember Jim Gray's paper!

- Some μ-services can adopt CAP, because for them, "weak consistency is safe".
- Some can use Paxos/SMR, like "one-shot transactions on a single shard."
- Some use transactional database solutions, but on sharded data.

WHAT HAPPENS WHEN THEY GET IT WRONG?

Convoy effect: If some service runs slow, often a large amount of concurrent work “queues up” waiting for it and we lose all concurrency.

Reboot storms: When such a big overload arises that everything times out, crashes, and restarts.

Inconsistency storms: When a system that “normally” runs with good cached data suddenly finds that all cached data is extremely stale.

DOES THIS MATTER?

Yes, it can be *extremely* disruptive! One client might cause the issue, yet the whole cloud becomes nearly useless.

Instabilities of these kinds are among the worst nightmares imaginable for cloud operators.

So stability and scalability are paramount requirements in the cloud!

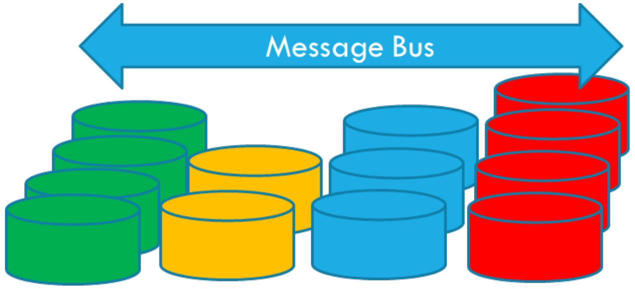
THE NEED TO AVOID INSTABILITIES RESHAPED THE CLOUD!

In lecture 1 we saw how companies like Amazon invented the μ-service *model*. Now we can see that they had to go much further. They needed new families of μ-services, ones that work really well at scale, and ways to teach people to build them.

Even with this, those μ-services can become hot-spots. They used special hardware accelerators and design tools to program the new hardware.

This work wasn't easy and required unusual expertise and luck.

THE ISSUE?



Racks of highly parallel workers do much of the data fetching and processing, ideally ahead of need
The old databases might still be present in this layer, or may have been split into lots of services.

Today's cloud-scale solutions work well!

But you need to use them in the way the vendor anticipated. You generally download a "story book" that describes some end-user need and how the company helped solve it, and comes with sample code.

The code mostly glues together existing services. Then you download that code and customize it to transform it into your own solution, for your case.

CAN YOU FULLY CUSTOMIZE THE FIRST TIER?

In the 2006-2015 period or so, we saw increasingly sophisticated products emerge to help automate the creation of web pages.

So often, the first tier is just one of these vendor-supplied solutions and you use some form of special design tool to tell it what you are hoping to do.

HYBRID CLOUD (WEB SERVICES MODEL)

An early response to the limitations of working with pre-built μ-services.

Issue: Company B can use company A's cloud offering, mostly, but that offering completely lacks some key subsystem on which B depends.

B has a choice: migrate the subsystem into the cloud, or continue to run the legacy subsystem "in house" and integrate it to the cloud remotely.

HYBRID CLOUD (WEB SERVICES MODEL)

Basically, the cloud vendor offers cloud-hosted services via APIs you can call via RPC from a networked application. This runs on HTTPS.

With this, the application running inside your company can make requests to services up in the cloud, with good security. *But there can be obstacles on the cloud-to-company path, so this isn't necessarily a bidirectional connection.*

The hope is that all the applications in your company can now be cloud-enabled. The reality is that hybrid solutions suffer from complexity and scalability issues.

INTERNET OF THINGS DISRUPTS EVERYTHING!

With IoT, we encounter a slew of new puzzles!

- Customers lack a high quality way to securely manage huge numbers of IoT devices, which are often "dumb" sensors.
- These devices need to be actively managed, like to update their firmware each time a patch is issued, to protect them against hackers, …
- There are thousands of devices and each has its own special, vendor-defined options for remote control.
- They often need real-time responses, large machine-learned knowledge bases that are frequently updated, fault-tolerance and consistency.

CANON EOS REBEL CAMERA…

This is Ken's SLR camera. What "events" can it generate?

- You don't normally think of a camera as a device…. Now we want to imagine that this camera is being used as a cloud IoT peripheral with a stable source of power. Some possible events:

Event	Meaning	Issues
On/Off/Idle	Power mode	When turned on, must authenticate
NewIMG	Took a photo	Should we download it? How fast is the link?
StorageWarning	Low on space	Which images to delete
FocusWarning	Dirty sensor	Something is preventing auto-focus from working.

AN IOT EVENT WOULD…

Come from some specific device, securely attached to the cloud and with a clearly defined owner who bought it and services it.

Have a device type and an event type and "tags" defining event *meta-data*, which is a term that just means that the event might talk about data but not include the raw data.

Why not include the data in every event?

- Typical photo might be 3MB in size. A video could be 1GB or more.
- But sensors rarely have ultra-fast connections.

IN AN IOT WORLD…

A typical end-user application (think of a hospital, or a small city, or a smart highway, or a smart farm) might have thousands or tens of thousands of smart sensors, of many kinds

- The devices would generally not be very smart, but each has its own superpowers, like autofocus or on-camera compression and storage.
- Different vendors/models: many "user manuals"
- We have limited battery lifetimes and bandwidth to contend with, and perhaps can't download every image or video.
- Some data may be more valuable than others.

PROGRAMABILITY

You can't run code right on the sensor itself: they aren't very smart.

You could build a specialized solution for each class of sensor, but this wouldn't be cost-effective

So companies wanting to be big players in IoT have begun to invent a new IoT-oriented first tier.

IDEA: A "DEVICE DRIVER" FOR EACH DEVICE

Suppose we could write code to manage the Canon camera?

This would let us create a set of categories of devices (cameras, videos, microphones, drones…).

- Vendors who create new devices could also create device management logic to integrate them with the Azure IoT Cloud.
- Probably they would need a different device manager for AWS, etc.

HOW WOULD SUCH A DEVICE DRIVER WORK?

In many ways this idea is similar to what operating systems do to manage devices we can plug in, like USB memory sticks or wireless mouse units.

But instead of plugging the device into your laptop, we are attaching it to a cloud system.

And instead of the driver running on some single computer, we would want to run it inside a cloud-based service.

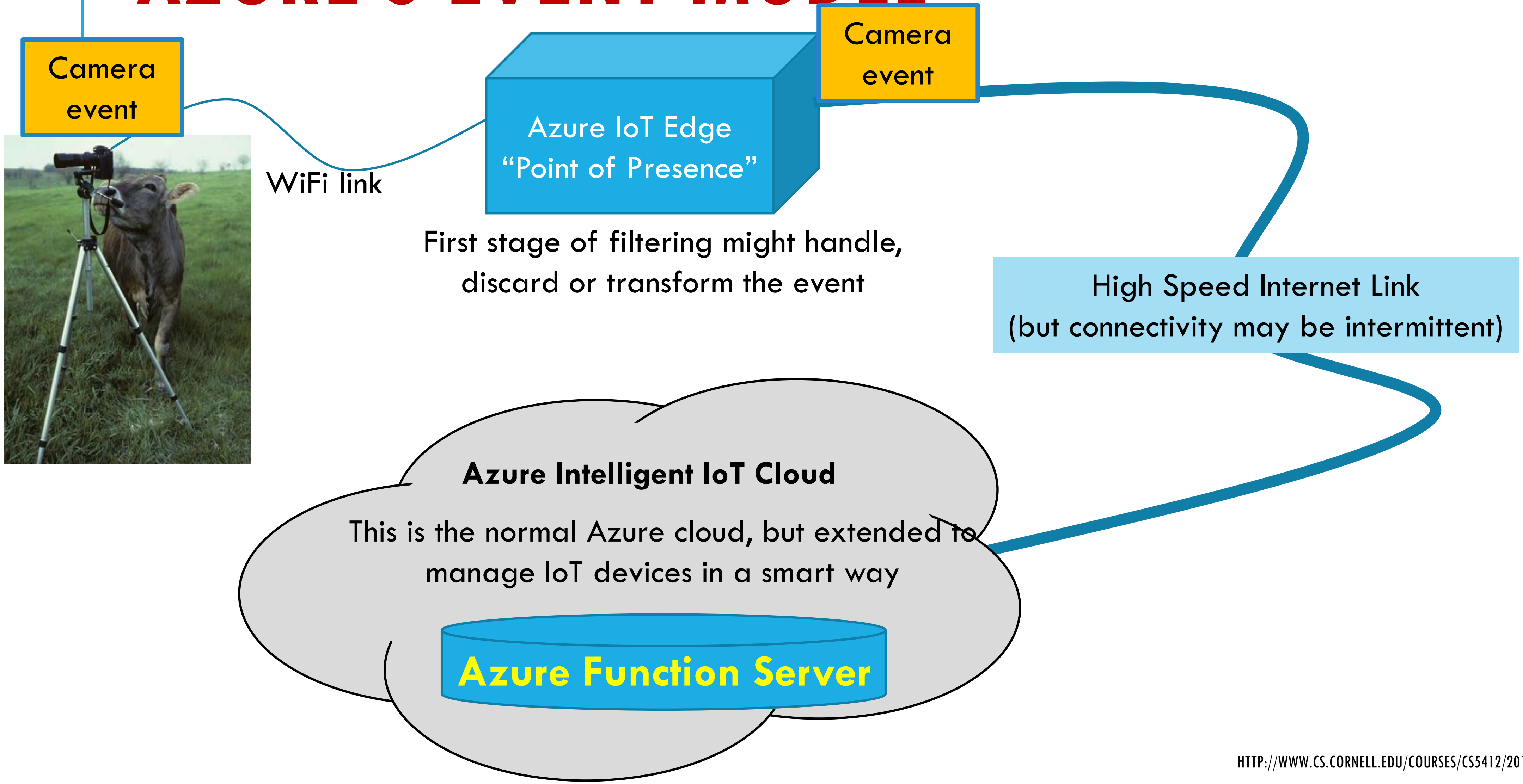
EVENT-DRIVEN IOT MODEL

So… we securely bind Ken's camera to our cloud, and register it under Ken's account (he'll be billed for the resources used). We use Azure IoT Hub.

Now the camera is accessible and we get events, and can send it events too:

- Cloud-to-camera: Turn on camera. Take one photo per second.
- Camera-to-cloud: PowerUp. {Status=success,13GB free space}
- Camera-to-cloud: Photo {UID=IMG-2546.jpg, GPS=42.44940,-76.48280,…}
- Cloud-to-camera: DownloadPhoto {UID=IMG-2546.jpg,Quality=ThumbNail}
- Cloud-to-camera: DeletePhoto {UID=IMG-2546.jpg}

AZURE'S EVENT MODEL



FUNCTION SERVERS

A very simplified way to create event-driven first-tier solutions.

- In Amazon, called Amazon Lambda
- Microsoft calls it the Azure Function Server

The server is an elastic set of machines that host very inexpensive containers running Linux or some other operating system of your choice.

You provide simple programs that get triggered by events.

FUNCTION MODEL

Each function is a small program that will be launched with arguments extracted from the event.

The function runs on some machine selected by the Function server, which has a pool of machines that it manages elastically.

To make things simple, the function and any files it needs are wrapped up into a container: a kind of virtual machine, very cheap to launch.

FUNCTIONS ARE "STATELESS"

This term is extremely confusing for many people.

A stateless function is simply a program you can launch with arguments that will be shut down after the event. Local files would be discarded.

The kind of state we lack is "long term storage". But the program is just a normal, with variables. It can even have files packaged with it – the only issue is that if it modifies those files, the changes are lost when it finishes.

WHY CAN'T THE FUNCTION HOLD STATE?

Every time an event occurs, we launch a new instance of the function, always in the identical initial state, the one defined by the container.

So in effect, if two events occur, for the identical device, they will be handled by different programs, maybe on different machines, and those programs have no "knowledge" of one-another.

Any data they try to retain from the event is erased when the function finishes handling the event: the VM (the container) will be shut down.

SO HOW CAN FUNCTIONS STORE DATA?

Many options! You get to decide…

- In cookies, on the sensor itself. This is how web sites track their users.
- In files, in the "global" file system. Every cloud offers one.
- In a database. Every cloud offers many. But remember Jim's advice!
- In a scalable (key,value) store. **For simple state, the best choice.**
- In a new stateful μ-service that you create. **In complex situations, this is the only realistic choice.**

FUNCTION MODEL

The idea, then, is to write programs to handle device events, for "both sides" of the connection.

Since the device itself is probably quite specialized and "dumb", one side behaves like a device driver that knows how to talk to this kind of device.

The device itself is the other side of the connection.

HOW THE EVENT IS PASSED TO THE FUNCTION

When an event occurs, a new instance of the event handling function you registered will be launched in a “clean” state.

The event itself is available either as program arguments, or via an API

You can also register a shell script if you wish.

EXAMPLE: PASSING PARAMETERS TO AZURE FUNCTIONS (AWS LAMBDA IS QUITE SIMILAR)

In Azure, a function *trigger* defines how a function is invoked. A function must have exactly one trigger. Triggers have associated data, which is usually the payload that triggered the function.

Input and output *bindings* provide a declarative way to connect to external data or μ-services from within your code. Bindings are optional and a function can have multiple input and output bindings.

Triggers and bindings let you avoid hardcoding many details that would involve complicated "boilerplate." You can arrange to receive data (for example, the content of a queue message) via parameters in the trigger.

MORE DETAILS, PLEASE?

This becomes kind of complex, and in CS5412 we won't dive into full details. Configuring a trigger or binding involves editing a JSON file.

For the specific question of writing a function that uses triggers and bindings, read this web page.

We are arranging for a Microsoft Azure training session early in February for those who would find it valuable.

WHAT IF A DEVICE CAN GENERATE MANY KINDS OF EVENTS?

A single "function program" will handle all of them:

switch(event-type) { …. }

The event type would be passed as one of the event parameters. This way there is still just one trigger for the function.

Your logic for dealing with a single event should be short and simple.

HOW DO FUNCTIONS TALK TO μ-SERVICES

Three main options:

- Remote method invocation (for example over the RESTful RPC layer, or JNI, or WCF).
- Via a message “bus” (no storage: like a “broadcast”)
- Via a message “queue” (stores messages, like an email)

Use the remote method approach for immediate actions with immediate responses. The other two “decouple” the source and receiver.

EXAMPLE LOGIC?

Your function could be passed a thumbnail photo, then use a photo-analysis μ-service to decide whether the photo

- Has any "content". A farm photo with no animals might not be useful.
- Is of "interest". Here, you could use a pre-trained machine learning classifier that has learned which kinds of photos interest you.

Then if the photo seems to be interesting, you could download the full version. If not, you might still leave the dull ones on the camera just in case.

FANCIER CASE: POINT, FOCUS AND SHOOT

Suppose that some event occurs: “Animal motion detected”.

This will cause us to swivel the camera.

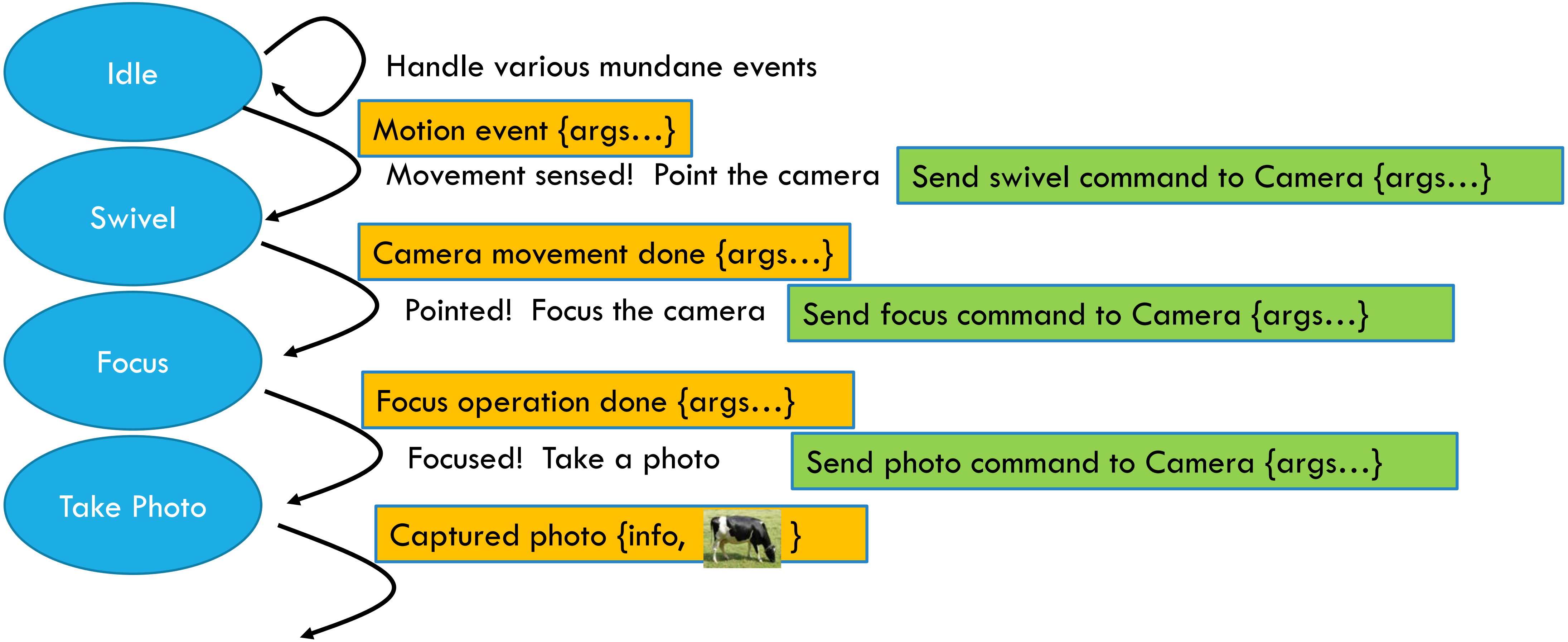
When the camera is pointed towards the location, we should focus.

When the focus converges, we shoot a photo.

Now the thumbnail is sent to the server.

If the photo is considered interesting, we’ll download it.

THIS IS A FORM OF STATE MACHINE!



Orange: “camera to Azure IoT”

Green: “Azure IoT to camera”

DEFINITION: STATE MACHINE

A state machine is a program that is in some "state", corresponding to the nodes in the figure. The states form a directed graph.

Events cause some action (label on the arrow) and also a transition to the same state (loop back) or some other state.

In an IoT setting, we favor *deterministic* state machines: The same events, handed to the state machine in the same state, produce the same effect.

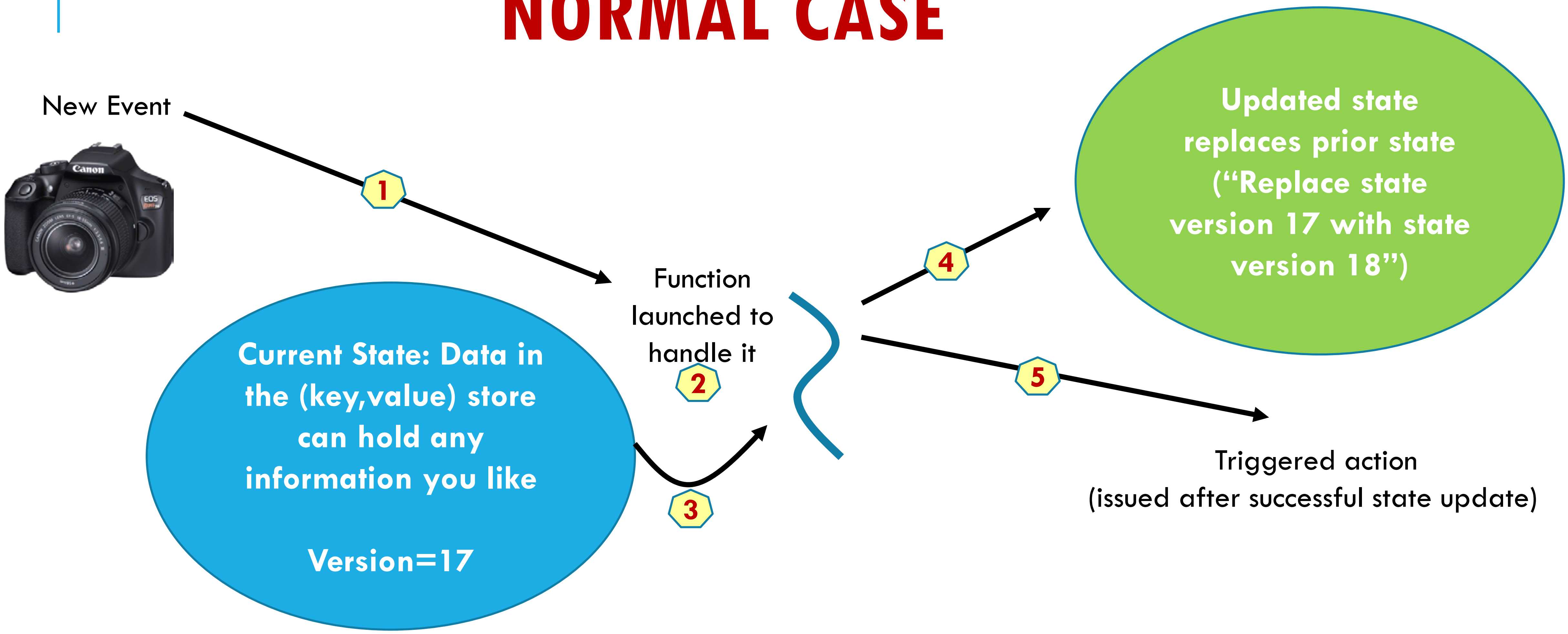
… THAT STATE IS HELD IN A KEY-VALUE STORE

The function loads it, mutates it, then stores it back.

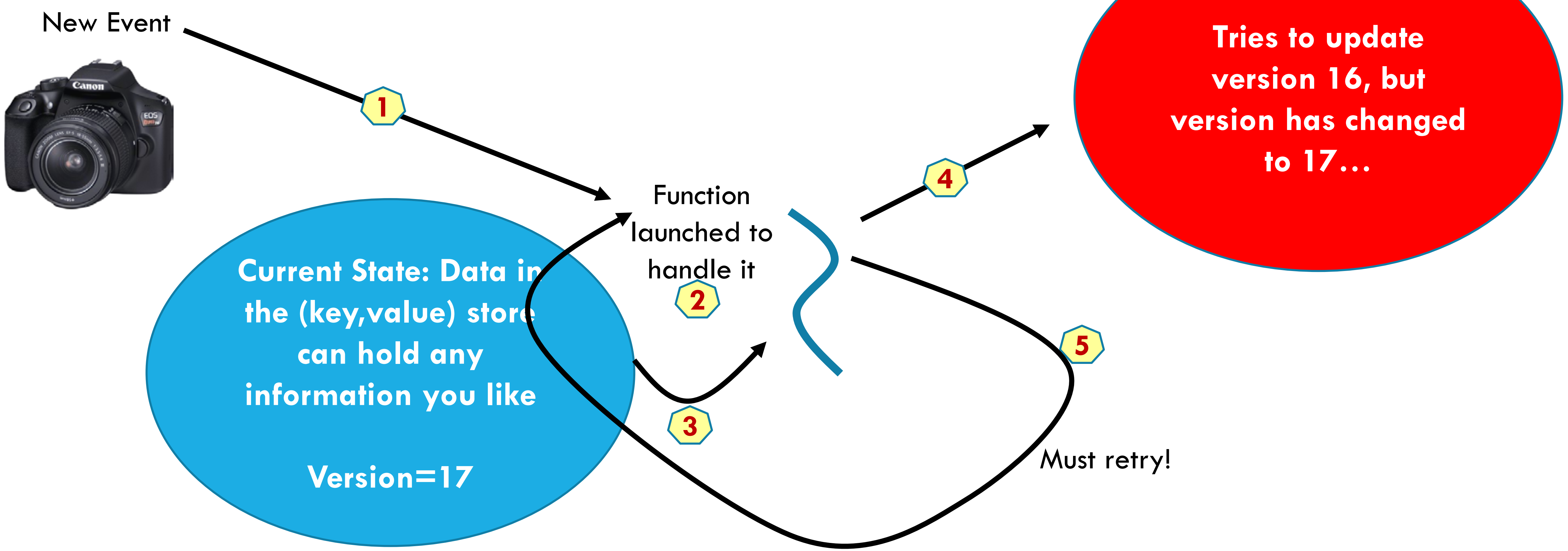
An atomicity issue arises if concurrent events trigger two functions that try to make conflicting updates to the state machine state.

If we solve this by adding locks to the key-value store, Jim Gray's scalability warning applies! To avoid locks, we use a kind of "atomic" *conditional key-value put.*

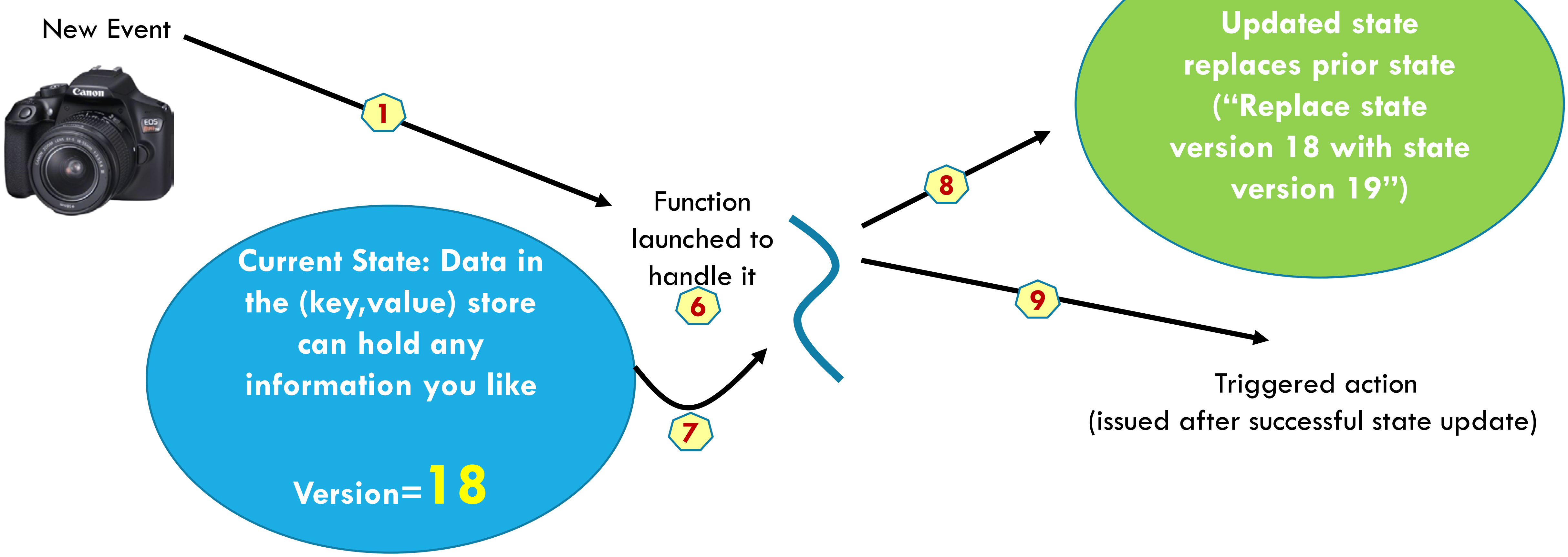
STATEFUL BEHAVIOR WITH A FUNCTION NORMAL CASE



STATEFUL BEHAVIOR WITH A FUNCTION CONCURRENCY CONFLICT



STATEFUL BEHAVIOR WITH A FUNCTION RETRY CASE



THIS IS EASY, IN SOME SENSE. YET IT BREAKS UP THE FLOW OF LOGIC

You wouldn't encode long sequences of complex logic this way.

So a function service has handlers limited to very simple direct actions, or very simple sequences.

Anything more complex needs to be implemented in a μ-service

GREAT MATCHES FOR FUNCTION MODEL

When someone swipes their card, check to see if they are authorized and if so, unlock the door. *Use a μ-service to store authorization records.*

When the drone reaches the end of the field, send it instructions for its next sweep. *Use a separate μ-service to compute the overall flight path.*

When the camera captures a photo, examine the thumbnail. *Ask the "photo is interesting" μ-service if we should pull it down. If so, send a "download" req.*

FUNCTIONS OFTEN USE EXISTING μ-SERVICES

Key-value storage (with conditional updates!)

Message bus (pure notification) and queues (stored until deleted)

File systems, which may include services for compressing big objects, photo segmentation & tagging

Deduplication services ("Only store this if it includes something new")

Blockchains (used for a tamper-proof audit trail, append-only)

.... Easy to dream up more of them specialized for particular needs, but then you need to implement them on your own. **This will be common in IoT.**

EXAMPLES OF SPECIALIZED ONES

Recognize farm animals by coloration, or RFID, or other means.

Compute trajectory for a drone

Compute the direction of the wind

Decide what seeds to plant in a particular parcel

Study a photo to decide if a wilted leaf suggests new insect infestation

… all of these might keep their own long term storage (state)

HOW WILL YOU KNOW WHICH TO USE?

Draw a diagram of the state machine you'll need.

If it only has a very small sequences of simple steps, implementing them purely with functions and saving state in a μ-service will be fine.

But if the state machine seems at all complex, you'll probably need to either build your own μ-service, or build a new μ-service that leverages existing ones but still "holds" the longer-term stateful interactions.

BIG OPEN PUZZLE: HOW TO ADD NEW ONES

Many applications would seem to need specialized new μ-services.

But those would be hard to implement without significant distributed computing skills. They:

- Certainly must scale to handle load, and might need "elasticity" options.
- Probably need fault-tolerance, hence replication features
- Will often be built by people with expertise primarily in other domains, like machine learning / artificial intelligence.

μ-SERVICE “DEVELOPER TOOLS”

Over time these are likely to become common, but today they are mostly lacking.

Cornell’s Derecho system is aimed at this use case, and is an option for skilled developers to consider.

A tool or framework would implement a basic set of template features and then be customizable in some way using developer logic.

EASIEST OPTION TODAY?

Most people seem to be trying to shoehorn application-specific logic into some form of key-value store.

So in effect we end up with two "services": a deployed key-value store, and then a second service that talks to the key-value store but is implemented separately. The customized logic would be in the second one.

But this is (obviously) clumsy and could also be quite inefficient.

BUILDING NEW μ-SERVICES

You could also use a tool like Cornell's Derecho software library.

Derecho is a powerful infrastructure and we will learn much more about it soon. It lets you build complex self-managed services to run for long periods of time and self-heal if failures or other disruptions occur.

Many companies will probably μ-Service development kits over time

DOWN THE ROAD: TENSOR FLOW AND JULIA?

Within the machine learning community, many machine learning tasks are currently coded in a version of Python extended with features for big-data matrix and tensor computing: Tensor Flow.

Julia is a machine learning language from MIT with similar options but also supporting graphical data structures like Bayesian belief networks.

Neither is used in the cloud edge today, but perhaps someone will find a way to move them to the edge in the future.

WHY THESE LANGUAGES?

They are very good for expressing machine learning algorithms, A.I. data structures, and image processing tasks.

In our examples, many required some form of “machine intelligence”.

But today, we really don’t often see either used in an IoT edge setting. Perhaps, never at all! So this is a future opportunity.

HOW FLEXIBLE IS THIS PARADIGM?

The claim is that it can cover a vast range of use cases!

One puzzle is to decide what belongs in a μ-service and what belongs in a function, since this "trick" of saving state allows the functions themselves to do fairly elaborate things.

The general model is to keep functions very simple and fast and put complex logic into the μ-service layer (adding new ones, if needed)

WHY CAN WE BE CONFIDENT THESE PROBLEMS WILL BE SOLVED?

The key thing is to follow the money.

When there is a really big business opportunity, technology companies always jump in and contribute their own solutions.

So there will be many prebuilt services, and this will include tools for people who want to create new ones… eventually.

"The Internet of Things is transforming the everyday physical objects that surround us into an ecosystem of information that will enrich our lives. From refrigerators to parking spaces to houses, the Internet of Things is bringing more and more things into the digital fold every day, which will likely make the Internet of Things a **multi-trillion dollar industry** in the near future." — Price-Waterhouse-Cooper report

FARMING IS JUST ONE OF MANY "MARKETS"

- Smart food delivery "pipeline" from farm to table.
- Smart homes, and cities. Smart highways and traffic light patterns.
- Smart "apps" for this smart world (like Elder Care, or Healthy Living)
- Smart power grid.
- Smart corporate campuses that adapt heating, lighting, humidity
- Smart protective systems to crack down on theft and other crimes

CONCLUSIONS?

Azure IoT is a powerful infrastructure but to tackle these smart applications, you still need to do a fair amount of coding.

Function servers offer a great way to handle high volume but simple events, and they can even be somewhat stateful.

But fancier functionality probably needs to occur either in an existing μ-service or in a new μ-service you would design and implement.