LECTURE 5: MAKING DHTS DO MAGIC TRICKS!

Ken Birman

Fall, 2022

TODAY'S AGENDA: TWO PARTS

A DHT is a very simple μ-service, just offering **get**/**put**/**watch**. Why is this "enough" for the cloud? Today we will aim at:

- Understanding how to put "anything at all" into a DHT for scalability and high performance: DHTs can hold data *in memory*.
- Limitations and factors the DHT app developer must keep in mind.

CENTRAL QUESTIONS WE WILL FOCUS ON

How spread out do I want my data to be?

How do deal with very large, complex, data structures (with pointers!)

What to do about the small but non-zero risk of key collisions

START BY THINKING ABOUT DATA STRUCTURES

We all have worked with lists, trees, perhaps graphs. But our experience was with one program on one machine. Big cloud data sets can have a similar need – but with a *lot* of data, spread over a lot of machines.

Even a photo is actually a structure that has an image plus meta-data.

Our challenge is that to store data at huge scale, we want to use key-value products, but it isn't so obvious how to "map" from these standard representations to the forms that can match with a **put**/**get** API!

SIMPLEST DATA STRUCTURE: A LIST

Why lists? Usually because the application needs to associate multiple data fields with some item.

For example, this photo of Biscuit has an embedded list of meta-data such as location taken, date, camera settings, etc.

Access them via the menu option “properties” for the actual file.

EXAMPLE PHOTO METADATA FIELDS

Location taken

Data taken

Author

IP rights if any

Camera type

Lens settings

Photo encoding parameters…

BUT WHERE IS THAT LIST "LOCATED"?

Initially, the camera itself takes the photo and generates the initial version of these fields.

That list is baked right into the file holding the photo.

In fact the file is a bit like an archive (think of JAR files, or ZIP). The photo itself combines the pixel array with a type of JSON file listing attributes

SERIALIZATION/DESERIALIZATION IS USED TO CONVERT THIS "STRUCTURE" TO A BYTE ARRAY

Any object class can optionally be defined to also have a method that will convert the data in the object to a byte vector, and a constructor to create a new instance of the object, given a byte vector.

These byte vectors have a length, which varies, and we can put them in messages, key-value stores, files, etc.

The storage layer won't know what the serialization format is, so it just remembers that these 72,604 bytes "are" a photo object called "Biscuit"

WITHOUT THE OBJECT CLASS YOU CAN'T DESERIALIZE THE BYTE VECTOR

To understand the bytes requires a program compiled using the class definition and its logic for serialization and deserialization.

For common file types like .jpeg and .mpeg there are standard programs that can be asked to interpret the bytes for you.

- This is how file systems (and key-value stores, and databases) are able to generate thumbnails and list photo properties: they ask for help.
- You could extend that set to support your very own object types, but would need to learn how to properly register your helper program

BUT WHAT IF THE CLOUD WANTS TO ADD MORE LISTS, OR CREATE BIG ONES?

For example, we might track all the friends of a person, or all the likes for a photo or video.

That list is generated by the cloud, not the camera. And this is good: ideally, a photo or video should be immutable once we first store it.

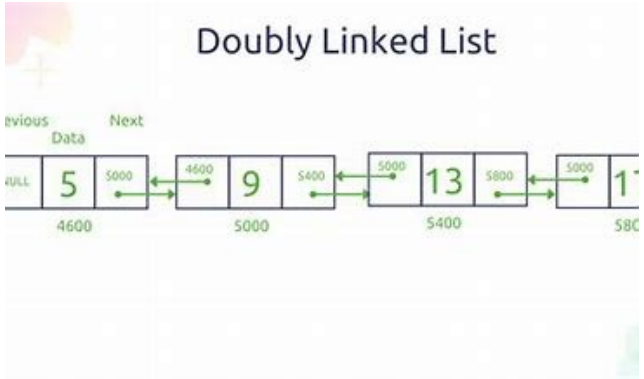
- Because an immutable object won't change, it is easy to cache
- It can never become stale. But this also means it can't hold the cloud-provided lists of additional properties, like "liked-by"

HOW CAN WE STORE A LIST INTO A DHT?

For example, we could have a list that starts out initialized with the same data extracted from the photo meta-data, but then that can be extended with things like "liked by" or "other-photos-of", or "map location"

This would allow us to leave the original photo object unmodified. We learned that immutable objects are convenient because they are easy to cache and never become stale. So by separating the list from the photo, we gain this form of scalable flexibility!

… LIST IN A DHT



Easy to just store the list "inside" the photo, give each node a unique id

Save each node on the list using (node-id, obj) representation. The object is just the byte vector obtained by serializing the node.

Instead of memory pointers, just have the next and prev fields contain node-ids for the next node and the previous node.

BUT NOW WE NEED TO THINK ABOUT COST

For small numbers of objects, just calling **get** a few times (maybe in parallel) is a reasonable option. Datacenter networks are fast!

… but if we form a really big list it gets very long and might be costly even to count the length of the list (like if we had a list of "likes" for a famous video of an iconic song).

And other operations could be even more expensive

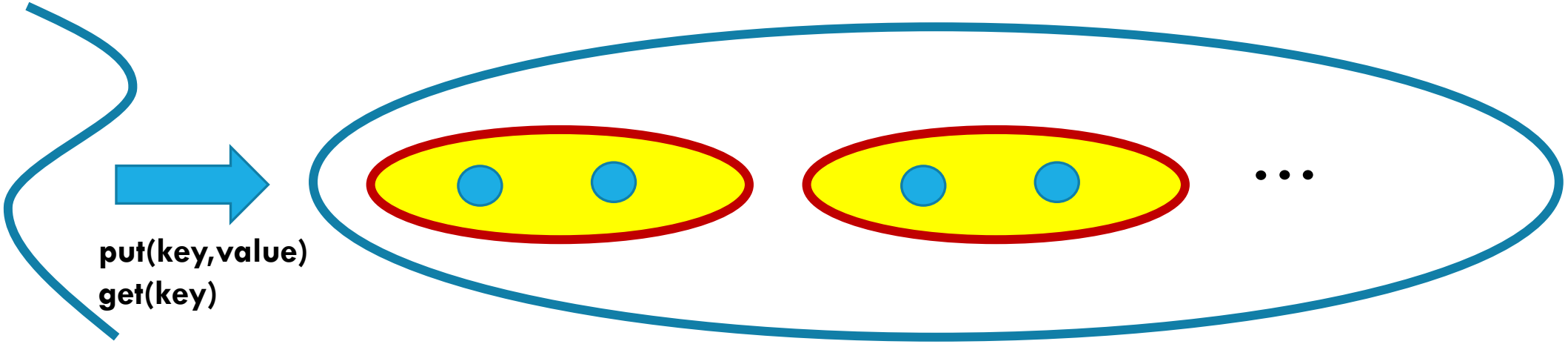
REMINDER: DHTS ARE SHARDED. BUT THIS MEANS ANY BIG LIST WILL BE TOO!

The DHT idea can be traced back to work by people at Google, and to papers like the Jim Gray paper on scalability.

We take some service and structure it into shards: sub-services with the identical API but handling disjoint subsets of the data.

Given a key and a value, the DHT will map the key to some shard and save the tuple at the member (or replicate it over the members)

DHT PICTURE



Client application

DHT Service on many nodes
Shards on 2-nodes each, using state-machine replication

A DHT assumes that the data-center network is very fast. 10us delays for a gRPC are typical – whereas the client application has 100ms to send a reply back to the human end-user (customer)

… PLUS, WE CAN EVEN TREAT THEM LIKE DATABASES, AND THIS IS IMPORTANT!

We haven't looked at the exact way to embed SQL into code yet, but you've heard about this idea (NoSQL)

Any list can be treated as a kind of database:

- As an example, I could "code" this in a few lines in C++, Python, Java:

  from my list of friends, find friends who are currently within half a mile from me, not busy in a class now, and like pizza.

- We will look at the SQL notation used in language settings later

CONCERNS WE MIGHT WANT TO THINK ABOUT

When we create a very big list, how will it be accessed?

Individual **put**/**get** operations should have O(1) cost.

But that query would need to search for friends. Will it do a sequence of operations, one per node? That could become very expensive!

- In fact for speed, we clearly need a way to do parallel search
- And we probably want to look at many nodes per "operation"

NETWORK SPEED / DELAY?

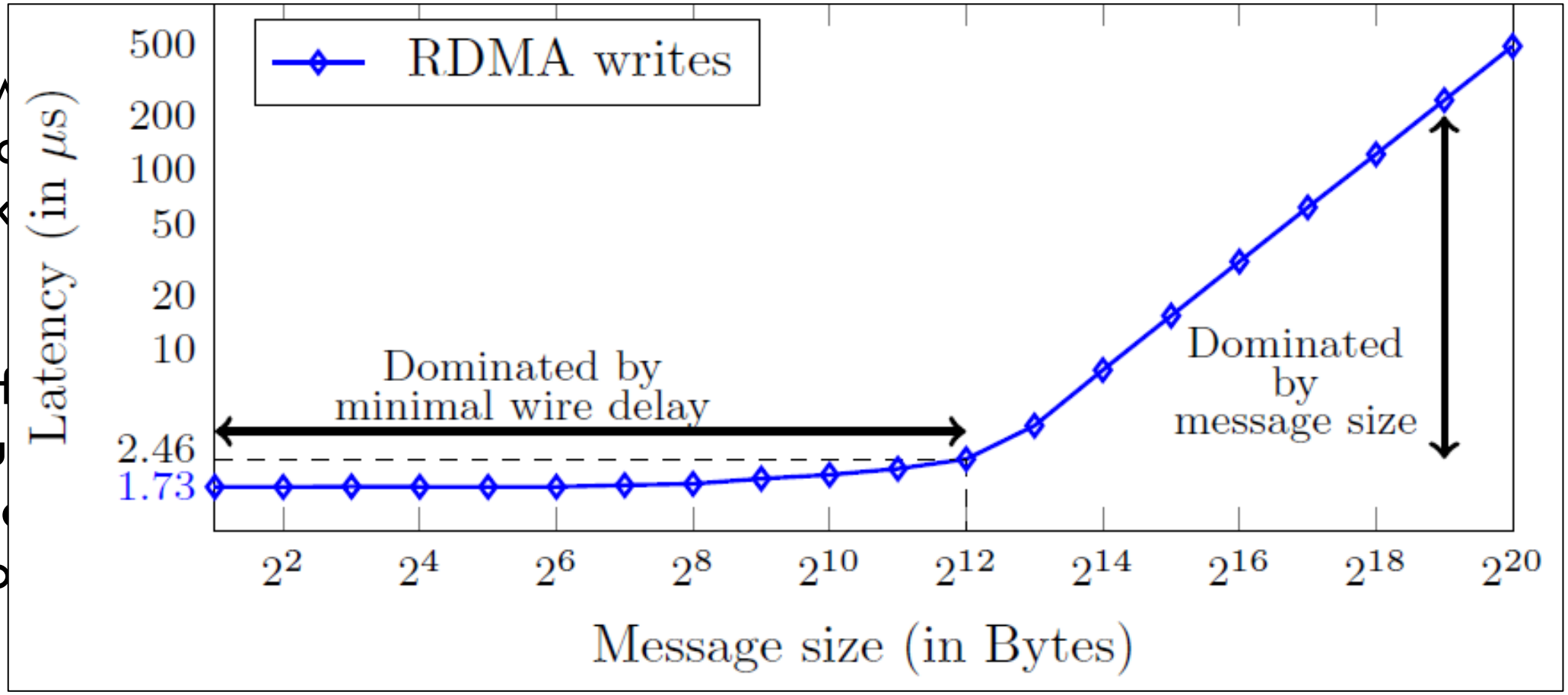
Networks operate in terms of "packets" with a maximum size (usually 4K, 8K, or 64K). Smaller objects can be inefficient because the network packets are not full

The first packet you request typically incurs one "delay" to send the request and one for the reply. But subsequent packets show up quickly in a chain, so we pay this first delay once, then additional cost as a function of total data size we transfer.

NETWORK SPEED / DELAY?

Netw
8K, c
pack

The f
requ
a che
of to



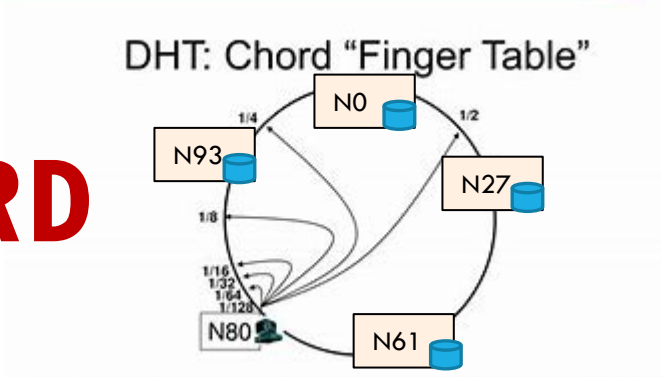
HOW CAN WE USE THIS INFORMATION?

When storing individual objects like photos it might already be useful

But cloud solution architects use concrete data even more when they think about complex operations, like that query searching for friends to get pizza

Understanding how a NoSQL query might be implemented lets us visualize the data access patterns that would be used, which in turn can let us design the data stored in the DHT in a way that matches nicely.

EXAMPLE: SEARCH IN MIT'S CHORD



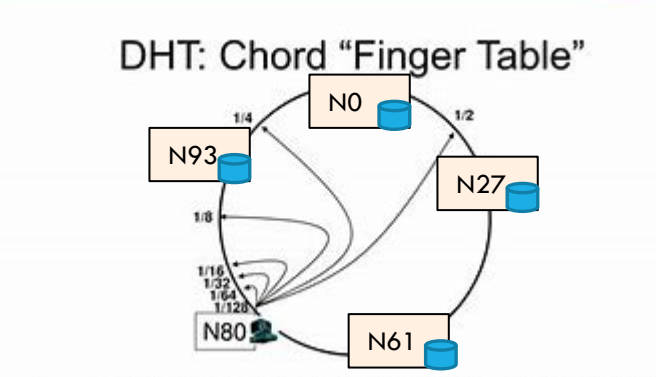
Chord treats the key-value store as a circular data structure representing the *hashed* key values: from 0 … max_int.

Each <u>server</u> is assigned to some location on the ring (as if we were giving the server a key). A server owns all keys falling between the prior server and its own hashed location. A uniform assignment spreads the load evenly on servers.

A shard would simply be each node and its K-1 neighbors in the clockwise direction. Chord replicated data within the shard. There is one shard per node in this approach.

Each server then would maintain a "finger pointer table" telling how to get from **here** to the server owning **here+max_int/2, here+max_int/4, ….**

HOW DID CHORD WORK?



Example: If we talk to N80 and ask it to please track down key 7, N80 doesn't own that portion of the space, but can look at the ring.

Halfway across is N0. So it forwards the request to N0.

N0 has its own finger table. It realizes that N27 is a better choice. It goes at most distance ¼ across the ring, clockwise, for this hop. This continues if needed.

N27 does own key 7. So it replies to the client: "here is your object"

BEEHIVE: CHORD PLUS RANGE SEARCHES

Beehive was one of a few DHTs that used *unhashed* keys instead of hashing them. It also varied the replication level, to make poplar keys easier to find

The benefit of not hashing the key is that data remains in sort order along the ring.

A disadvantage is that Beehive required very active placement management.

… BUT, CHORD DIDN'T "TAKE OFF"

The problem turns out to be that in a modern cloud, SQL range searches are fairly rare compared to fetching individual objects

So making a single lookup into an O(log N) operation in order to make range searching easier isn't a popular concept

Chord is used by the Akamai CDN but is not in wide use elsewhere

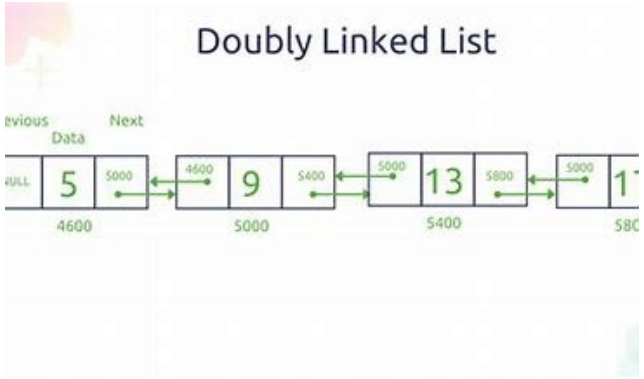
HOW DO REAL CLOUDS DO IT?

What other options do we have?

Parallel search on chunks of data is a popular tradeoff

Group key-value objects so that code can access many of them in one **get** operation. Then design the language-layer NoSQL API to efficiently perform parallel operations on chunks (a concept called MapReduce)

COLLOCATION



For example, in a list we could group neighboring nodes so that when we access one we also get its neighbors.

Design a chunk size efficient for the network.

This leads to a two-stage list design: each C successive nodes will be held in one chunk. The chunk gets its own key and has the successor and previous pointers, but the C values are in an array.

HOW TO COLLOCATE LOTS OF NODES

data	9 13 57 21 22
next	*2271*
prev	*1008*

More tradeoffs!

- A single **get** will tend to fetch the whole chunk
- Putting a new value into the middle of the list would be costly, and probably shouldn't be supported (the chunk might get too big). At best we would let the chunk grow, then split it.
- But now we could potentially scan multiple chunks in parallel

CONCURRENCY: RISK OF CONFLICTING UPDATES

For a shared list that collocates lots of data in larger objects, we might have many programs simultaneously doing **get** operations, which is great – a key-value DHT is made for that.

But we could also have a few concurrent inserts or delete operations: **put**

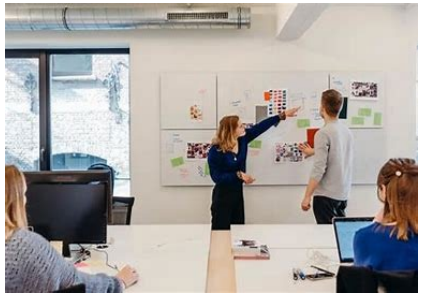
We would use versioned updates, since locks risk Jim Gray's issue

NOTICE ALL THE TRADEOFFS!

We have to think about:

- How long will our lists be?
- How big will the objects (and the network transfers) be?
- How often will updates conflict, and need versioned update operations?
- How many shards should we maintain?
- Would we be better off using a Chord-like approach?

IS THERE A SINGLE BEST SOLUTION?

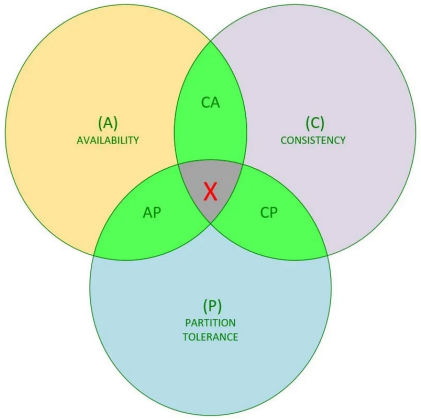


Unfortunately, no! Different applications need their own different mappings to the DHT model, for optimal efficiency

You are expected to just use a white board, estimate the percentage of read-only queries versus updates, and design a solution that spreads the loads and shouldn't have a lot of collisions between **get** operations

Most likely this analysis will be part of your decision process for figuring out how many nodes to collocate in one object on one shard.

ON TOP OF ALL THIS, CAP ENTERS THE PICTURE!



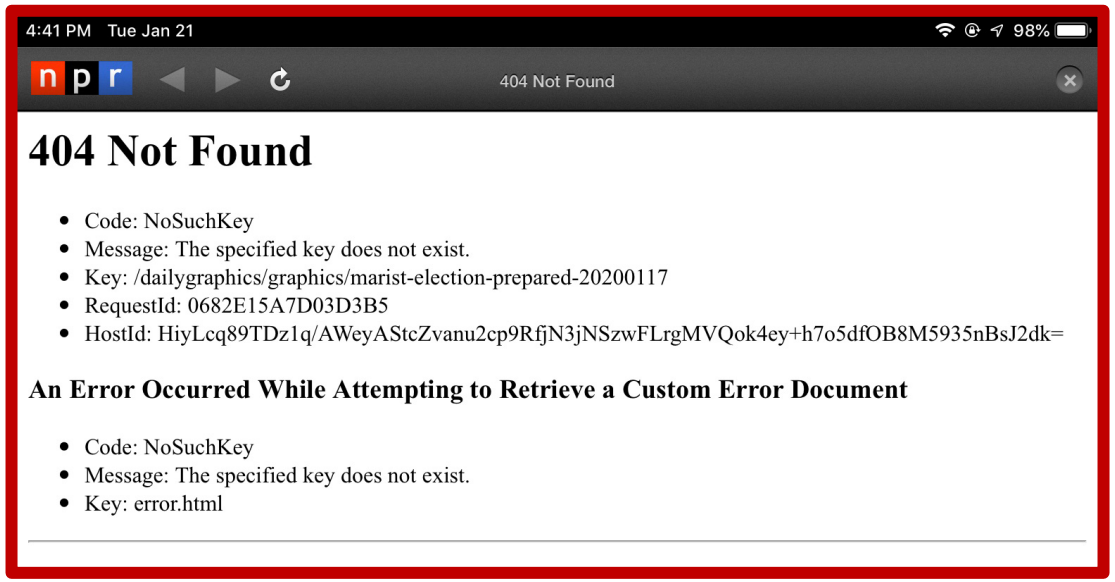
Recall: CAP was Eric Brewer's rule of thumb: *relax consistency, use AP*

Yes, sometimes web pages might be wrong. This is fine if it is rare

The goal is to convince yourself that it will be rare, have the underlying system be basically consistent, and have it converge (self-repair) if an inconsistency does arise (BASE methodology).

EXAMPLE: AN NPR NEWS ARTICLE

This error message from a popular news site, NPR, is clearly caused by not finding data in a DHT!



They probably stored their articles in the DHT, but somehow got an error when trying to fetch this article to build my web page. It could be an example of CAP: When a DHT resizes elastically, sometimes it makes errors for a few seconds afterwards…

WHAT DO YOU THINK HAPPENED AT NPR?

The data must have been changing when I accessed it!

So my web page builder in the first tier of the cloud saw inconsistent data

And it trusts CAP, so it still built "something" but not a very useful page. I refresh, and then it works… This is BASE: CAP in action

WHY WOULD IT HAVE BEEN CHANGING?

Perhaps the shard was still in the process of updating but I already tried to query it

Perhaps this DHT happened to have a crash and needed a moment to repair itself and recover

Perhaps elastic resizing was occurring and my data was temporarily on the wrong shard

COMING UP WITH SUITABLE DHT KEYS

You need a unique name for the objects you are storing.

For example: Ken's dog was named Biscuit. But "Biscuit" is <u>not</u> a unique name. The DHT could have some other object with that name too.

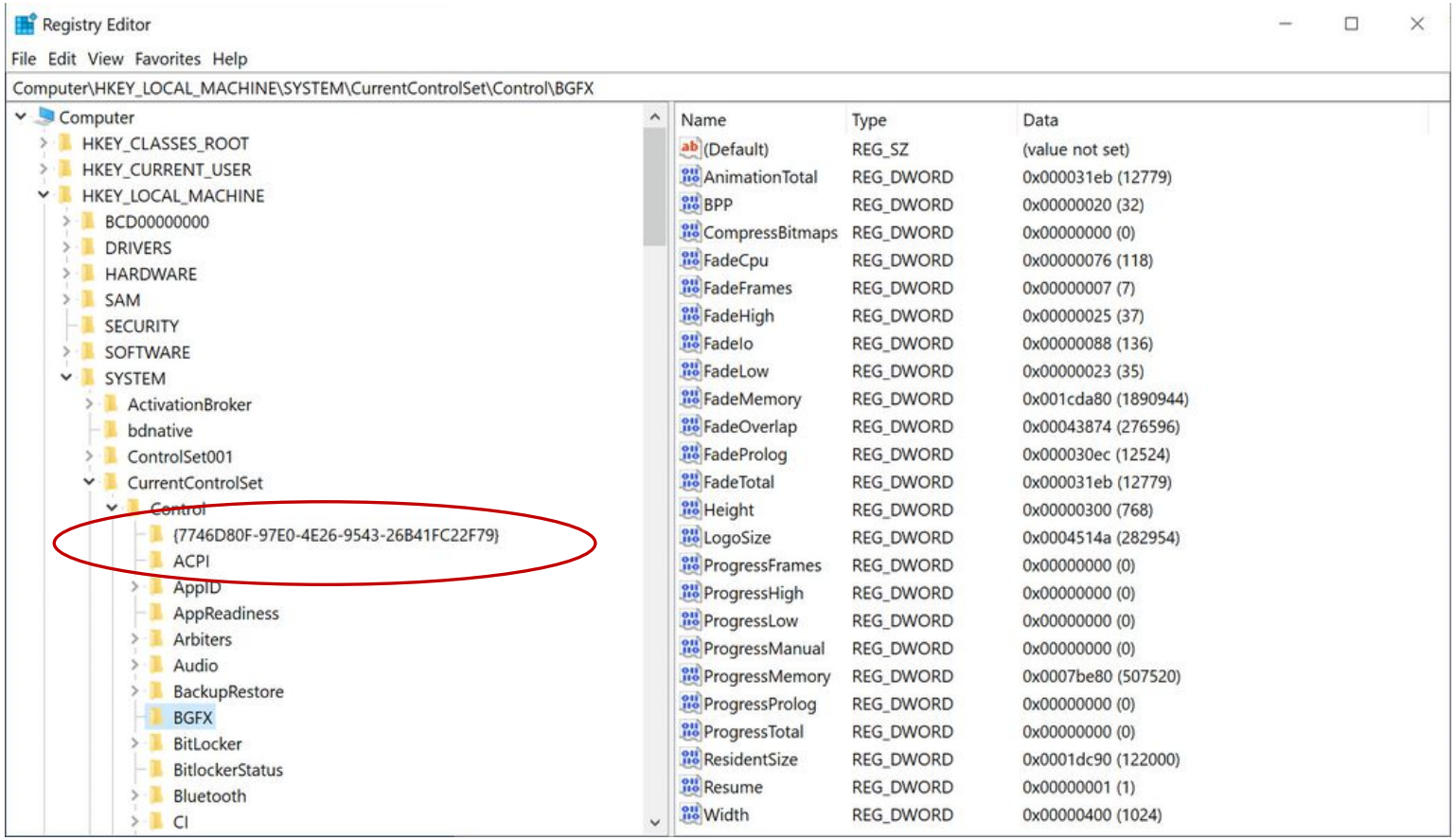
On the other hand, "/users/Ken Birman/pet/Biscuit" is a unique key, and we can hash it with SHA64 or SHA128 into a unique integer.

… UNIQUE KEY GENERATING SERVICES.

They provide a genuinely unique key

- Microsoft and AWS both have “registry” services.
- The resulting keys will have no obvious meaning to a human user, but is unique.
- These are for objects we will fetch “algorithmically”, not for external programs to use directly.

EXAMPLE: MICROSOFT REGISTRY



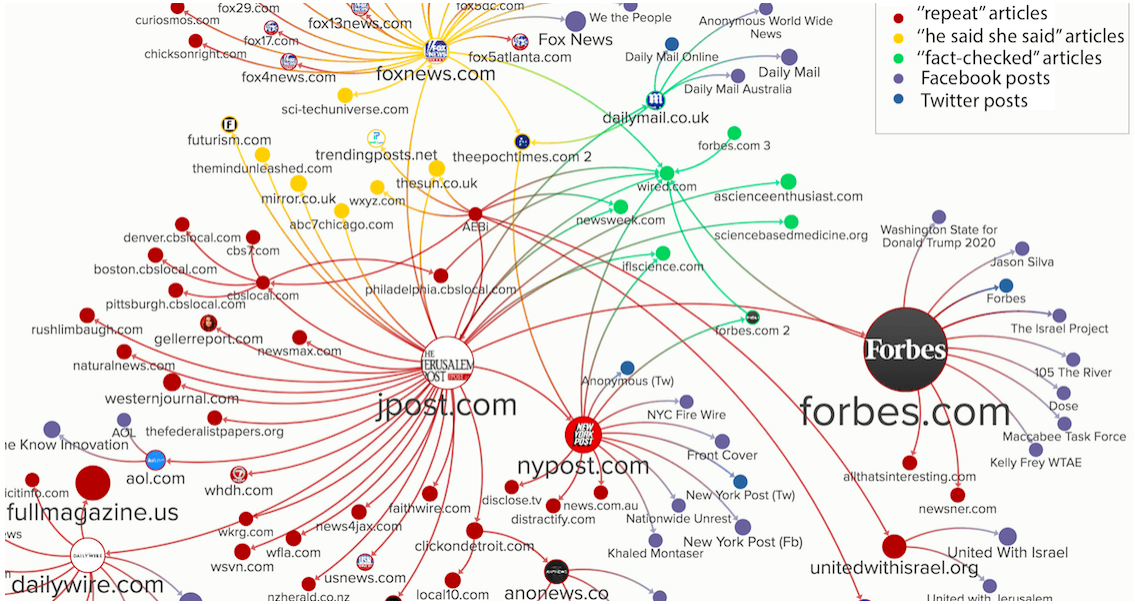
COULD KEYS "COLLIDE"?

No: Most cloud systems favor fairly large keys, like 64 bits. And key generators use a variety of tricks to make sure that they won't give out the same key twice.

For example the key could include the name of the server that generated the key, which ensures there can't be a tie.

ISSUE WE WILL TACKLE LATER: CLOUD DATA FORMS A HUGE, COMPLICATED GRAPH

Ken belongs to Entrepreneurs' Org



The entrepreneurs shared a viral (completely exaggerated, basically fake) story about a complete cure for cancer.

BIG GRAPHS HAVE MORE AND MORE ISSUES!

We will discuss Facebook TAO in a future lecture

It encodes the whole Facebook social network into a form of DHT

Many tricky decisions were required, but it works really well

SUMMARY

Almost anything can be stored into a DHT. The cloud does this. But it isn't free: you need to be clever to encode your application into a DHT.

Think about keys, object sizes, access patterns, costs. Be wary of "leakage" (neglecting to delete temporary data) or you will get a BIG monthly bill!