LECTURE 5:
MAKING DHTS DO MAGIC TRICKS!

Ken Birman
Spring, 2020

TODAY’S AGENDA: TWO PARTS

Understanding how to put “anything at all” into a DHT for:

- Scalability: the capacity is determined only by how many servers we have (and how replicated the shards are)
- Performance: DHTs can hold data *in memory*. When deciding between a DHT approach and a big disk, because modern datacenter networks are much faster than disk I/O, we can count on much faster access to DHT data

In the second half of the lecture, we’ll look at other issues a DHT creates

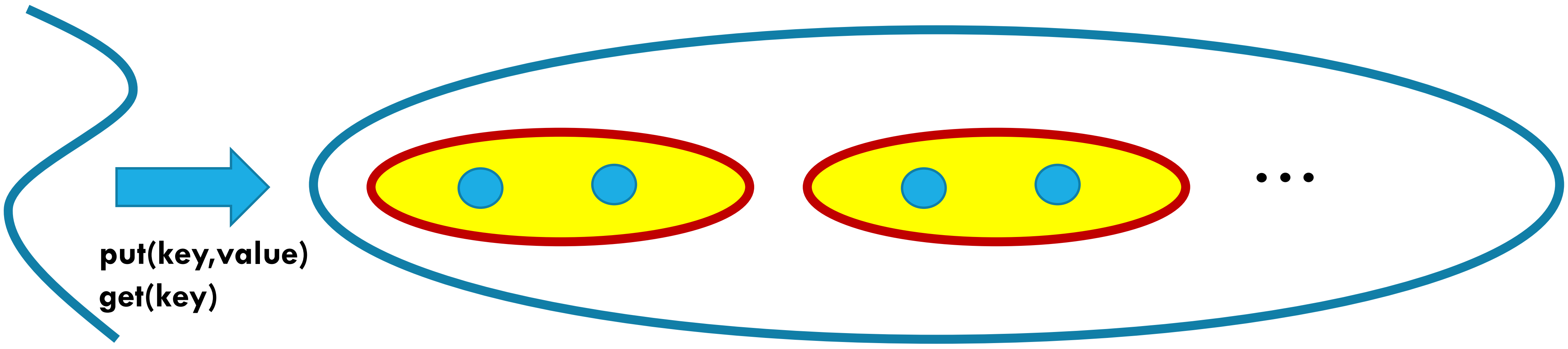
REMINDER: DHT BENEFITS (AND WHY)

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.

We need some way to know where to place each data item. We use a key here: the key is a kind of unique name for the data item, and by turning it into an integer modulo the number of shards, we find the target shard.

DHT PICTURE

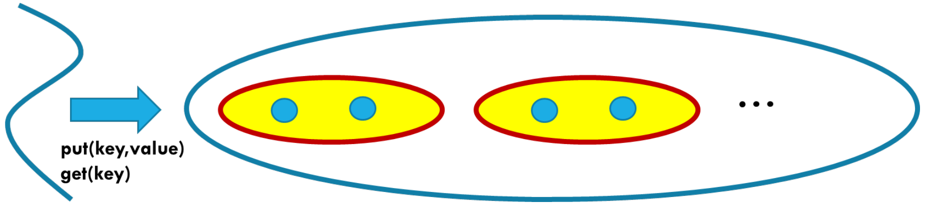


put(key,value)
get(key)

Client application

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

AN ANNOYING LIMITATION

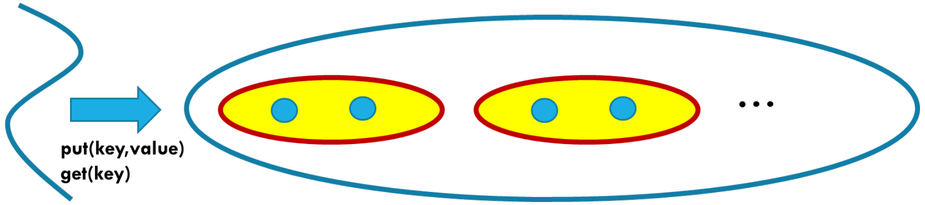


We are not given any way to do locking or 2-phase commit. In fact Jim Gray showed us that locking across shards would be ineffective.

A **get** or **put** is an atomic action on a <u>single</u> shard. For fault-tolerance, the shard update can use state machine replication (atomic multicast or Paxos).

With this approach we get "unlimited" scaling, and we can even keep all the data in memory (as long as the number of shards is big enough).

ANOTHER ISSUE

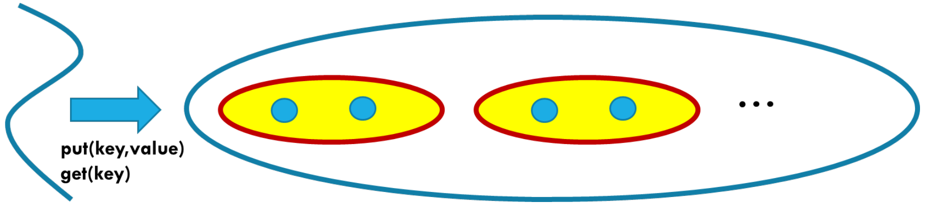


Data can be scattered around. In fact, this is the whole idea!

With the basic **get** and **put** API, this forces us to access each item separately. If related data was clustered at one shard, we could design fancier APIs that could get more work done in one atomic step.

Core issue: The DHT was created by a cloud operator and uses a hashing scheme you don’t control. Different keys tend to go to different shards.

SIDE REMARK



In fact it isn't horrendously costly that the items are scattered around

- Those 100 us retrieval delays are very small and you might be able to fetch all your data in parallel, by issuing concurrent **put/get** requests.
- Moreover, if the DHT was created by someone else, you probably can't extend the API with your own fancier operations.

So this limitation is really only relevant if you are building your own μService using a DHT sharding approach.

DHTS WORK BEST FOR DATA THAT DOESN'T CHANGE AFTER IT IS INITIALLY STORED

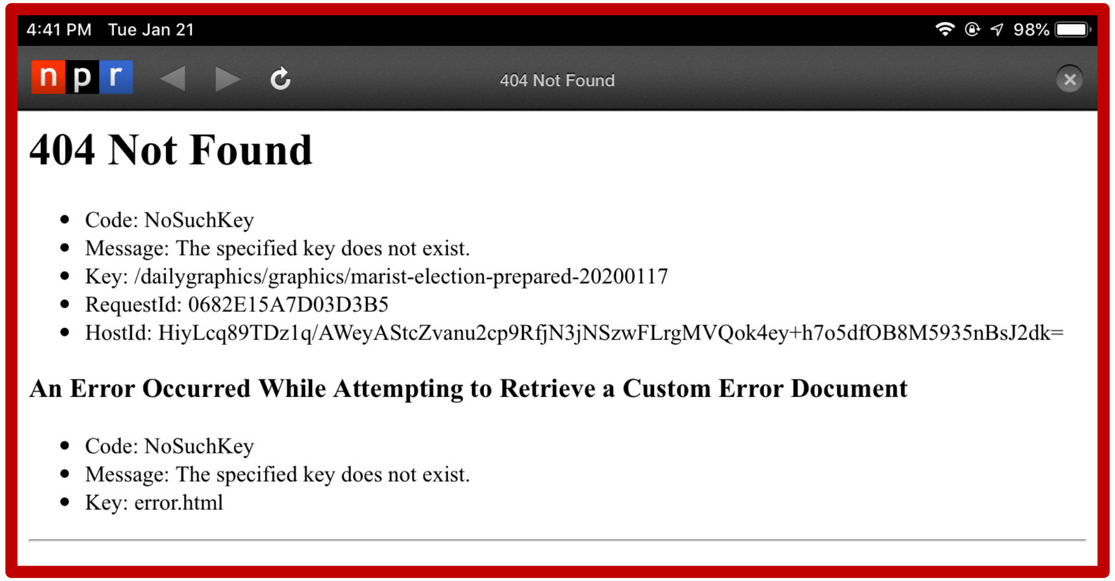
Once a web page has been uploaded, we probably won't update it again. A web page that won't change is an example of *immutable data*.

A DHT is ideal for this kind of data. Locking isn't useful for a big read-only data set even if we didn't have sharding!

Web pages are mostly built by reading immutable data. This is one reason DHTs became so popular: they work flawlessly here!

EXAMPLE: AN NPR NEWS ARTICLE

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



404 Not Found

- Code: NoSuchKey
- Message: The specified key does not exist.
- Key: /dailygraphics/graphics/marist-election-prepared-20200117
- RequestId: 0682E15A7D03D3B5
- HostId: HiyLcq89TDz1q/AWeyAStcZvanu2cp9RfjN3jNSzwFLrgMVQok4ey+h7o5dfOB8M5935nBsJ2dk=

An Error Occurred While Attempting to Retrieve a Custom Error Document

- Code: NoSuchKey
- Message: The specified key does not exist.
- Key: error.html

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…

BUT AN NPR NEWS STORY IS TOO EASY.

For the first few years, big search companies focused on just downloading snapshots of web pages and offering quick ways to find things.

Over time, however, there was an appreciation that the social network is the bigger opportunity. And these evolve rapidly over time.

So we saw a growing need to store data that *does* change.

HOW TO STORE "ANYTHING" AT LARGE SCALE

These data sets are huge – MUCH too big for any single computer.

Yet not only do we want to hold the data for access, we want superfast access: we want the data to be in memory, not on disk!

A DHT can solve this for us. The network I/O cost is a factor, but is still much faster than disk I/O. And modern datacenter networks, with the fastest software, can push network delays down to the 1-2us range.

HOW TO STORE "ANYTHING" AT LARGE SCALE

Puzzle: A DHT officially just holds (key, value) data:

- The key is an integer. Some permit various sizes: 64 bits, 128, 256.
- The value is generally either another integer, or a byte array.
- Some DHTs are specialized for (integer, byte[]), and some for (integer,integer). These often are used "together" for flexibility.

So how can we come up with a key for "anything at all"? And how can we use this value model to store "anything at all"?

Solution: We use ***serialization.*** This converts an object to a byte array.

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 not a unique name. The DHT could have some other object with that name too.

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

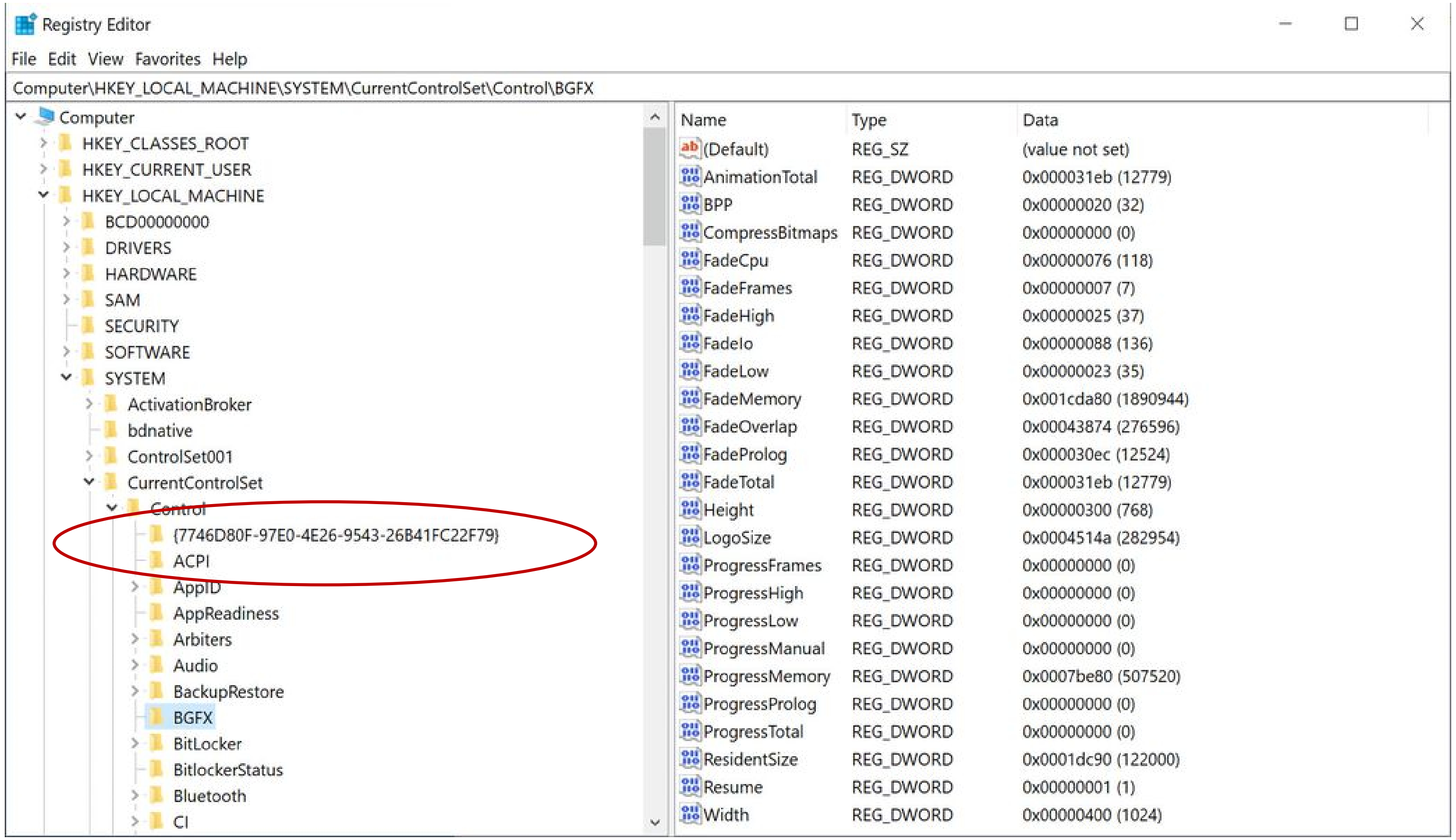
… BUT THAT RULE MIGHT NOT WORK IN GENERAL.

Some pet owners really love to use the same name for every pet.

How would you come up with a genuinely unique key?

- Microsoft and AWS both have “registry” services that are able to generate them.
- But now you run into the problem of having a key that has no obvious connection to the name of the object.

EXAMPLE: MICROSOFT REGISTRY



Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\BGFX

- Computer
  - HKEY_CLASSES_ROOT
  - HKEY_CURRENT_USER
  - HKEY_LOCAL_MACHINE
    - BCD00000000
    - DRIVERS
    - HARDWARE
    - SAM
    - SECURITY
    - SOFTWARE
    - SYSTEM
      - ActivationBroker
      - bdnative
      - ControlSet001
      - CurrentControlSet
        - Control
          - {7746D80F-97E0-4E26-9543-26B41FC22F79}
          - ACPI
          - AppID
          - AppReadiness
          - Arbiters
          - Audio
          - BackupRestore
          - BGFX
          - BitLocker
          - BitlockerStatus
          - Bluetooth
          - CI

Name	Type	Data
(Default)	REG_SZ	(value not set)
AnimationTotal	REG_DWORD	0x000031eb (12779)
BPP	REG_DWORD	0x00000020 (32)
CompressBitmaps	REG_DWORD	0x00000000 (0)
FadeCpu	REG_DWORD	0x00000076 (118)
FadeFrames	REG_DWORD	0x00000007 (7)
FadeHigh	REG_DWORD	0x00000025 (37)
FadeIo	REG_DWORD	0x00000088 (136)
FadeLow	REG_DWORD	0x00000023 (35)
FadeMemory	REG_DWORD	0x001cda80 (1890944)
FadeOverlap	REG_DWORD	0x00043874 (276596)
FadeProlog	REG_DWORD	0x000030ec (12524)
FadeTotal	REG_DWORD	0x000031eb (12779)
Height	REG_DWORD	0x00000300 (768)
LogoSize	REG_DWORD	0x0004514a (282954)
ProgressFrames	REG_DWORD	0x00000000 (0)
ProgressHigh	REG_DWORD	0x00000000 (0)
ProgressLow	REG_DWORD	0x00000000 (0)
ProgressManual	REG_DWORD	0x00000000 (0)
ProgressMemory	REG_DWORD	0x0007be80 (507520)
ProgressProlog	REG_DWORD	0x00000000 (0)
ProgressTotal	REG_DWORD	0x00000000 (0)
ResidentSize	REG_DWORD	0x0001dc90 (122000)
Resume	REG_DWORD	0x00000001 (1)
Width	REG_DWORD	0x00000400 (1024)

NAME SPACES AND KEYS

> "What is in a name? That which we call a rose
> By any other name would smell as sweet…"
>
> *Juliet, in Shakespeare's Romeo and Juliet*

A ***name space*** is some sort of user-oriented, semantically sensible, place to store names of objects. We could actually have one object in many name spaces, if the same object makes sense in different situations.

The namespace is used as a "service" to map from a name that makes sense to the user, to a unique internal key that makes sense in a DHT.

A cloud file system always has a namespace server as one component. We think of the storage servers as a separate, distinct, component.

KEN'S PETS

So we could, for example, have a kind of table listing all the pets Ken has had, with information about them

Pet Name	Period	Species	Photo List	Health Status
Nerd	1961-1962	Gerbil	*Empty*	*Deceased*
Susie	1970-1986	Keeshund	IMG-17171, …	*Deceased*
Biscuit	2003-2013	Golden Retriever	IMG-22187, …	Deceased

This table would be a "name space" if we the photo list was a list of keys

MANY THINGS CAN BE GIVEN UNIQUE KEYS

We could have a unique key for each row in a table.

We could have a unique key for each photo in a photo album. The album itself could be "named" but also have a key: its value would be a list of the keys for photos in the album.

Cloud systems use this approach very broadly!

COULD KEYS “COLLIDE”?

Yes, if the keys don’t have a large enough range of values, or the random number generator isn’t very effective.

Most cloud systems favor fairly large keys, like 256 bits. And some key generators use a variety of tricks to make sure that they won’t give out the same key twice.   A random number generator wouldn’t necessarily work.

Collisions would cause problems because two different objects would end up sharing what should be a unique name – a serious inconsistency.

WHAT PROBABLY HAPPENED IN THE NPR SITE?

It probably wasn’t a key collision.

In fact, I get news alerts for certain kinds of news stories, like confirmed first-encounters with space aliens. So… NPR posts a first-encounter story. And I receive an immediate alert!

The story was saved into a DHT, but maybe the DHT replication scheme is a bit slow, or it was resizing just at that moment for elasticity reasons. Until it “settles”, the key is correct but the story just can’t be found.

HOW DID THIS RELATE BACK TO CAP?

CAP was all about how we often face these choices:

- Give a super fast response on a web page, or an instant notification
- But this means not waiting for the elastic reconfiguration to finish, or for the replicated update to fully propagate to all the replicas.
- We might operate in an inconsistent way, briefly!

CAP says: you can’t have all three from {C,A,P} at the same time.

Relax consistency to get better availability and respond immediately even if a service you would have liked to check with is temporarily not responsive.

SUPPOSE WE HAD A LIST OF PHOTOS

The list itself is a data structure of objects that are linked by pointers.

The objects could be something like a photo description, and the photo (or other kinds of photo properties: “meta-data”).

HOW TO STORE “ANYTHING” AT LARGE SCALE

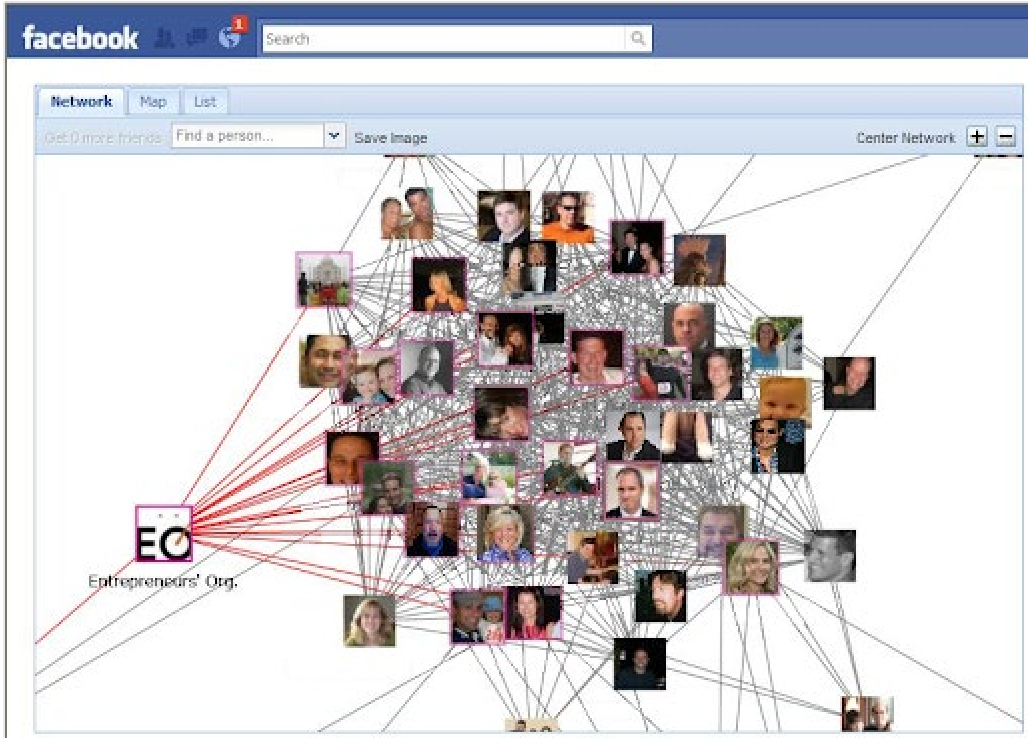
```
class myPet {
    int uid;
    animal species;
    hashset<string,photo> photo_collection;
    ….
}
```

```
myPet biscuit {
    uid := 5731,
    species := animal::dog,
    photo_collection := { [“on a rug”, ●],
                          [“in the woods”, ●]}
}
```

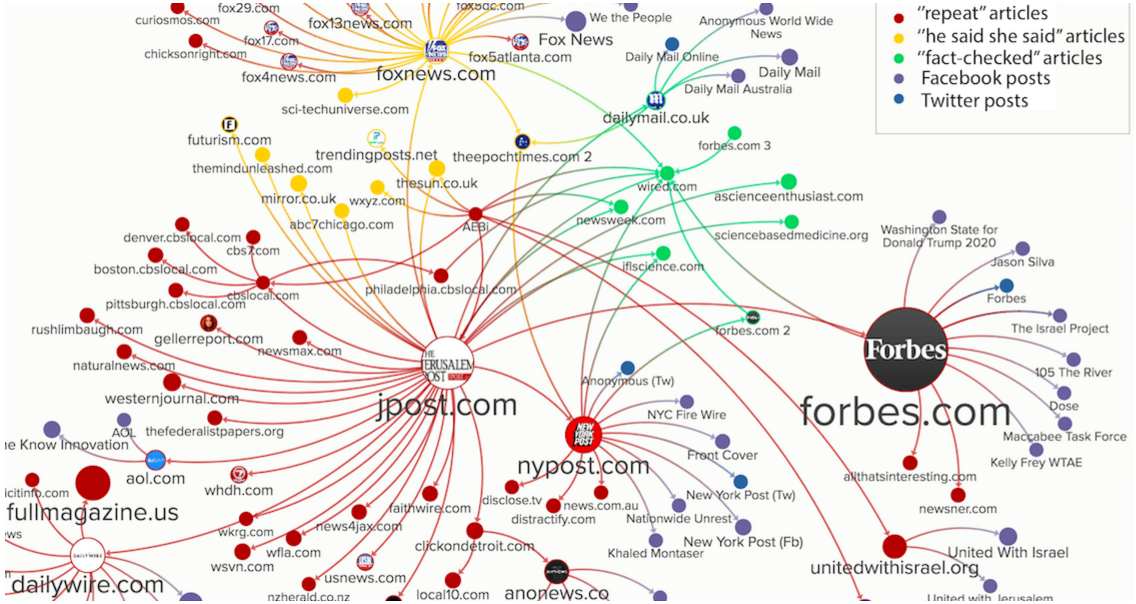
These objects all have type names the application understands, data that matches.

But the DHT wouldn’t know these object types! Amazon created the DHT long before you wrote the code defining this class.

NEXT ISSUE: 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.

SERIALIZATION CONCEPT

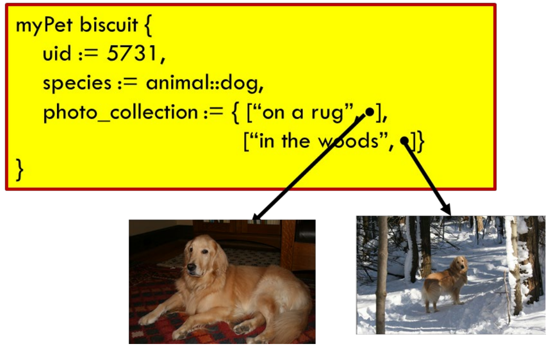
With a single object, a serializer just emits a self-describing data structure as a byte array, listing the field-types and their values.

If an object has sub-fields, it uses a recursive descent to serialize the inner object too. The self-describing byte array has a way to represent: “this is an object too, and the next 184 bytes describe it”.

The serializer output is often larger than the original object

ALSO, RECURSION ONLY HELPS FOR WHAT IS LOGICALLY A "SINGLE NODE"

The serializer knows how to do recursive serialization but only for what is logically a single object (and its subobjects).



```
myPet biscuit {
    uid := 5731,
    species := animal::dog,
    photo_collection := { ["on a rug", •],
                          ["in the woods", •]}
}
```

Key = "Ken Birman/pets/Photos of Biscuit"
Value = [0xFF 0xA6 0x1B 0x00 0x99 0x11 0x03 0xFF 0xFF …]

A deserializer is used to recreate the data structure. This is also sometimes called marshalling and demarshalling.

REASONS SERIALIZATION CAN BE COSTLY

We generally like the byte array to be in a “hardwareindependent format”, meaning that both Intel and ARM (and other devices) can reconstruct the object into their local data representation.

- There are several opinions about the best byte-order for integers
- Floating point formats are hardware-specific
- Some systems null-terminate strings; others just view a string as bytes
- Memory “alignment” rules differ from machine to machine.
- Compilers can make additional optimization choices

WHAT ABOUT A MORE COMPLEX STRUCTURE?

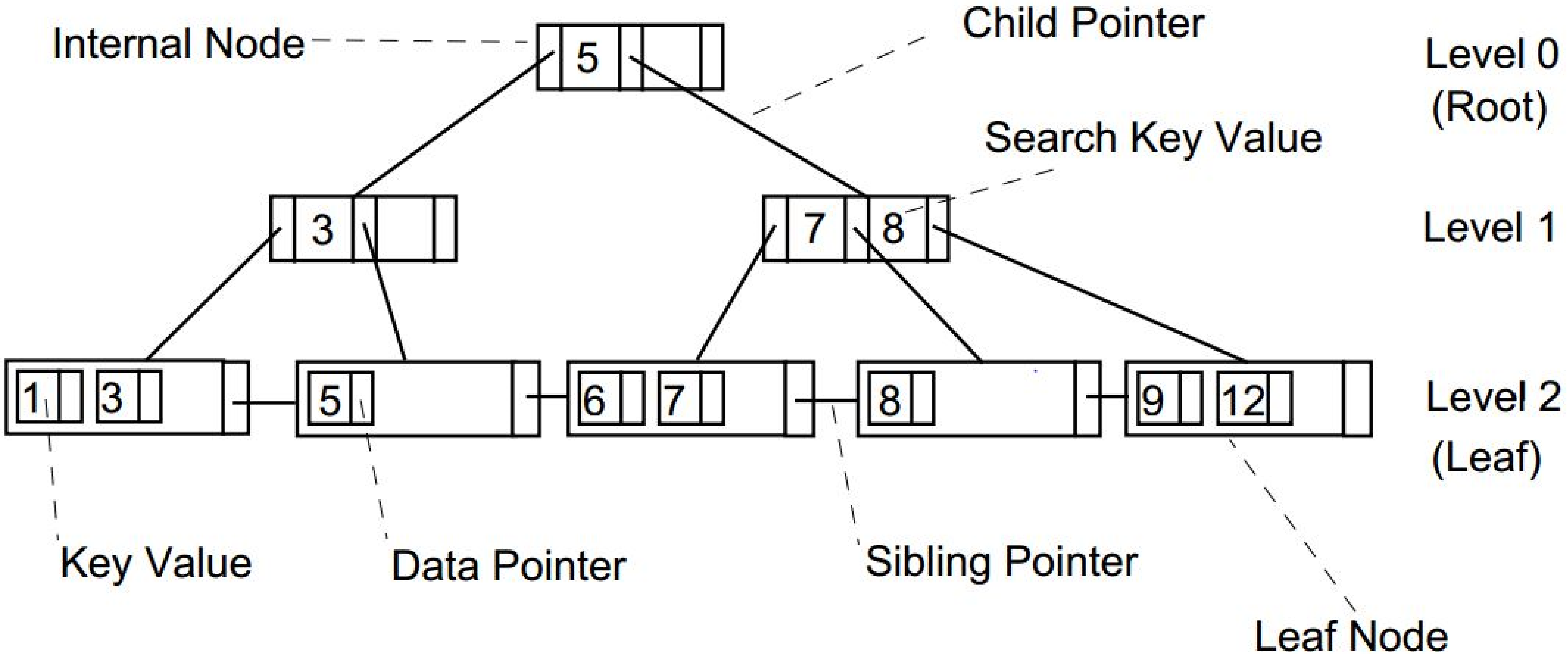
In homework 2 we are looking at using a DHT to hold a massive B-Tree

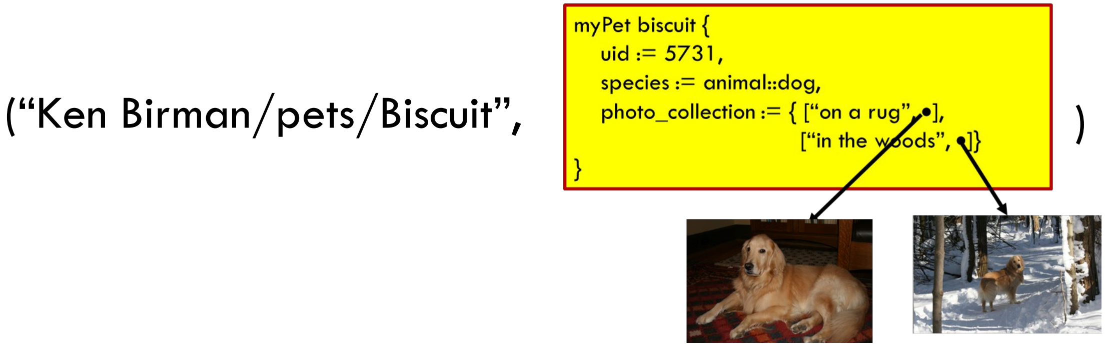
- They keep data sorted, which is useful, and you can do “range queries”
- You can also estimate the “statistics” of a data set, needed in AI/ML

The B-Tree is *much* too large to hold on one computer.

The data can be something complicated, but in addition, inner nodes have “pointers” to child nodes.

A 2,3 B-TREE EXAMPLE (WIKIPEDIA)





(“Ken Birman/pets/Biscuit”,

```
myPet biscuit {
   uid := 5731,
   species := animal::dog,
   photo_collection := { [“on a rug”, •],
                         [“in the woods”, •]}
}
```

)

GOOD NEWS

We can use the idea of replacing pointers with some form of key or node-id. Pick an approach that makes sense in the given situation.

- A node-id is probably just a 32-bit integer.
- A key might need to be 256 bits in size: wasteful, if not needed.

If a B-Tree uses memory pointers, it only can make sense on one single computer (server). But if it uses node-ids as “pointers”, this works even if the nodes are stored in different DHT shards!

HOW CAN AN INTEGER NODE-ID BE USED AS IF IT WAS A POINTER?

How does a pointer work in Java or Python or C++?

- We have an object in memory at some location.
- The pointer might point to that location, or to a structure that describes the object (in Java and Python, the structure approach is used. C++ uses a genuine pointer to the actual data and code).
- At runtime, we make sure the operation is legal (in Java and Python, this is for runtime error detection and polymorphism). Then we access the data directly.

MODIFIED BEHAVIOR?

We would need to check: ***is this node-id for a local node, or one in some other DHT shard?***

- If local, return a pointer to it.
- If remote, fetch it via a network RPC. Allocate memory (temporarily) to hold the copy. If the application modifies the object, write it back using another RPC. Free the memory when the access is finished.

In homework 2, you will "hide" this in a getter/setter

WHAT WILL THIS COST?

Now every node access might require a DHT **get** operation.

We can do a bit better: if we run our B-Tree logic on the same computers that run our DHT, then **local** get operations will be free. Only remote ones will be costly.

In Homework 2 we do this and even go one step further: we imagine building a kind of smart B-Tree search that can do a “piece” of a search. It talks to the local DHT shard, and goes as far as it can, then tells where to check next.

WHAT WILL THIS COST?

This idea of a combined DHT + B-Tree can go even further.

You can do a smarter node-id numbering scheme, or even a smarter hash function for the DHT, or both.

In Homework 2 we challenge you to come up with the best possible mix of node-id numbering scheme and hashing scheme, to minimize network costs.

DHTS CAN BE “TOO GENERAL” FOR SOME USES

At Facebook, the early work on social networking used one shared DHT.

But the social networking graph was huge, very complicated, and very heavily used. Facebook decided that it was just not an efficient solution.

They redesigned it, and later in the course we will learn about their solution: Facebook TAO, a specialist for social network graph storage.

A WORRY ABOUT GARBAGE COLLECTION!

When we use a distributed solution, such as a DHT, we often need to make copies of objects. For example, server A uses **get** to fetch a copy of some node that resides on server B.

But these copies then occupy memory, and we need to free that memory when finished with the copy. Otherwise, memory will leak and the server will eventually crash.

Fortunately, modern languages automate this step (in C++, use shared pointer template for the same behavior).

WHAT ABOUT STORAGE IN THE DHT ITSELF?

A DHT doesn't necessarily know how to garbage collect the (key,value) objects stored into it!

It does keep track: each DHT object has a user-id (who created it), and the user's account gets charged for the space consumed.

- Benefits? Total control, plus keys only need to be unique for the user or the application, not across the whole cloud.
- Problem: Many people are careless about freeing up the space!

DHT CLEANUP SUGGESTIONS

Always give an expiration time for every (key,value) tuple. If you want to keep an object longer, use a longer expiration time, but not infinite.

In your code, try to explicitly delete any temporary content you load into a DHT. Put it there, run for a while, but then delete it.

Commercial products offer fancier and powerful features, such as "delete these objects when such-and-such a server shuts down." Use them!

THE CLOUD CAN BE A VERY EXPENSIVE PLACE

When used properly, a cloud is often cheaper than owning hardware.

This is because you are sharing costs such as buying it and managing it with other users, and your share is potentially much lower.

But carelessness in storage management can leave all sorts of junk that might never be deleted, and you *will* be charged for it!

OTHER THINGS TO THINK ABOUT

Some DHT products allow you to control the hash function they will use. But this is not a standard thing – many do not.

Every DHT allows you to pick your own keys. And most DHTs tell you what hash function they will use, so you can pick keys “intelligently” if you wish.

Hot spots can be an issue. Even uniformly random inserts might be clumpy. And you can’t know which keys will be queried: some may be very popular

ONE DHT? OR ONE PER APPLICATION?

Amazon and Azure both urge you to use their DHT products: AWS Dynamo and Azure Cosmos.

But this means that many applications and even many users would potentially share the same storage infrastructure!

Is this a bad thing?

ISSUES TO THINK ABOUT

The DHT server will be more efficient in its use of space if shared by many users, so it will be ecologically greener and hence cheaper to use.

It will also stay busy all the time. If we plan to own a server and power it up, keeping it busy makes a lot of sense. But a shared server could become a hot spot because of some other user who pounds on some specific DHT item and overloads that shard.

- Your performance would suffer, and yet you have no way to know why!

SHARING CAN CREATE SECURITY ISSUES

Another form of leakage arises if data from one application or one user becomes visible to some other application or user, without permissions.

A DHT with distinct key spaces shouldn't leak information, but there could be software bugs or even performance behaviors that actually do reveal sensitive content, unintentionally.

We will discuss this in a future lecture. It is not a huge risk, but it is worth being aware of it.

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!

In homework 2, one of the challenge questions at the end focuses on minimizing access costs. This is typical of the issues seen in cloud systems that need to sort huge data sets, and that put the B-Tree nodes in a DHT.