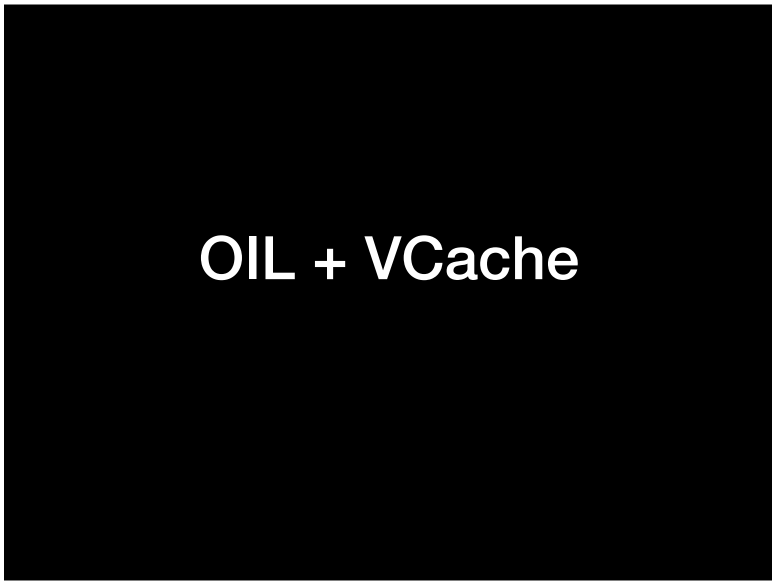
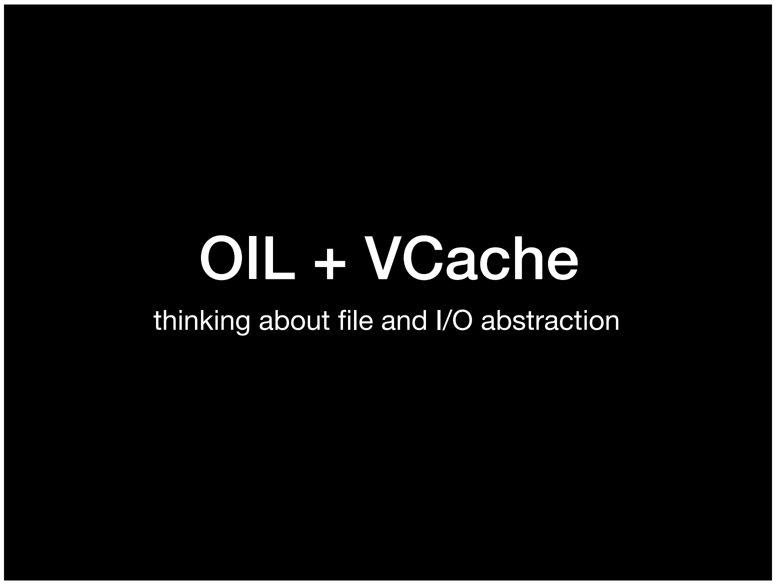
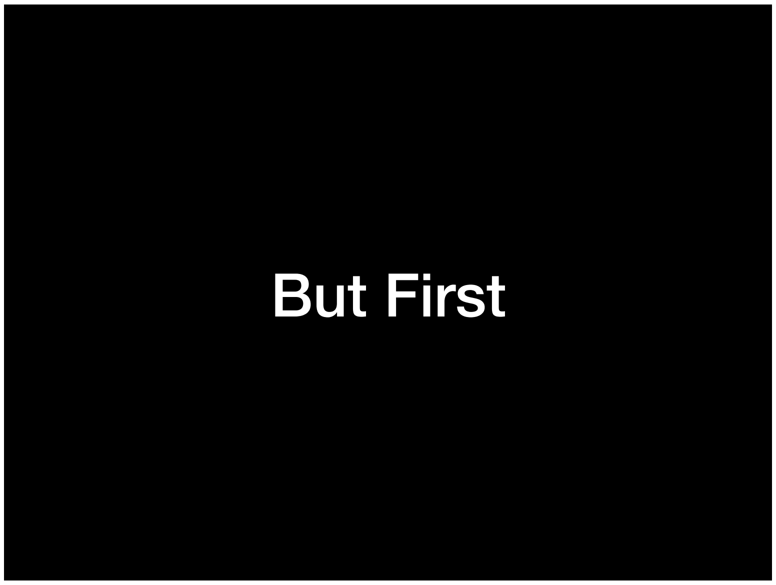

OIL + VCache

Hello!
I'm Roberto Peon!


OIL + VCache
thinking about file and I/O abstraction

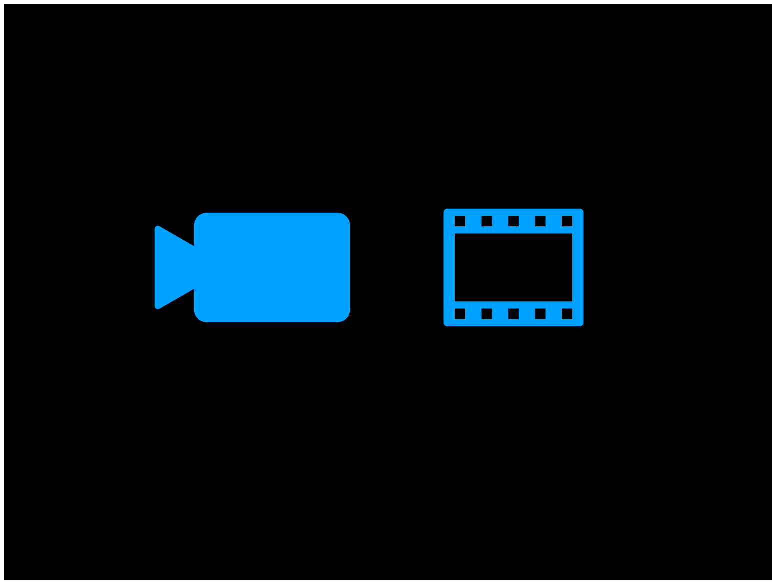
I'm here to talk to you today about abstractions, after all,
"All problems in computer science can be solved by another level of indirection" er abstraction!


But First

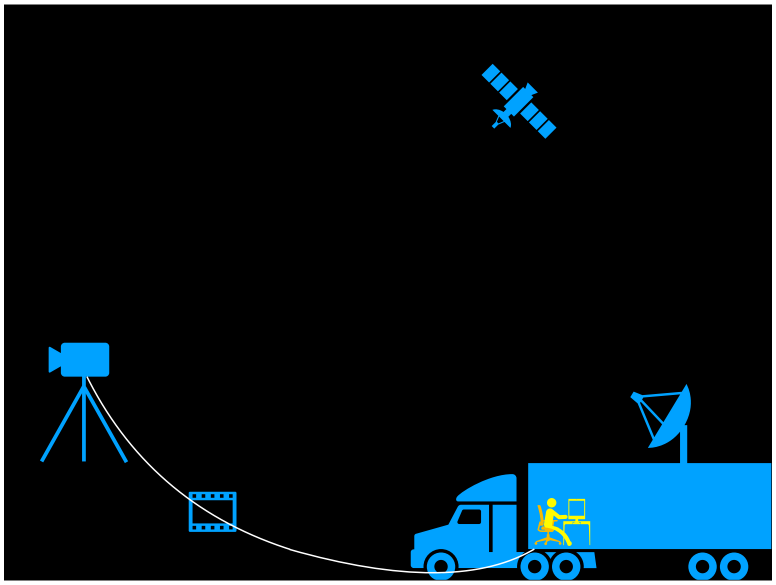
Let me tell you to story about how I came to care about abstractions

History

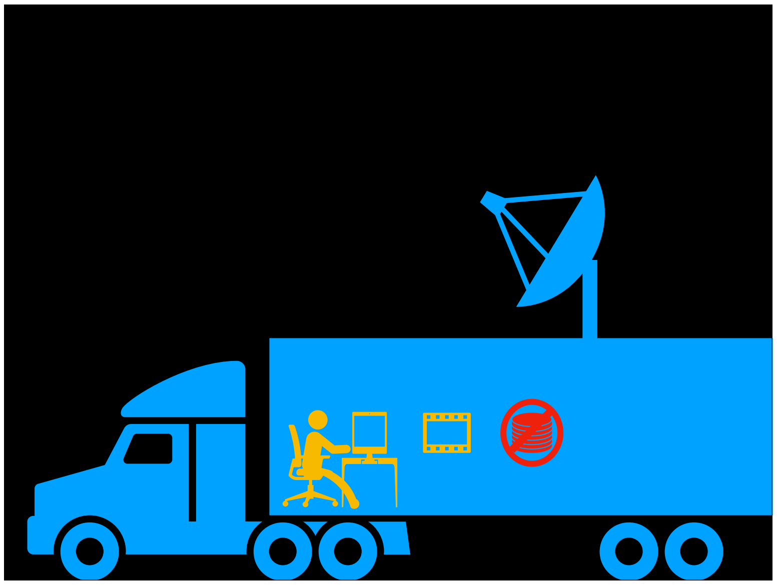
My first job after college was doing real-time special effects for live sports television.



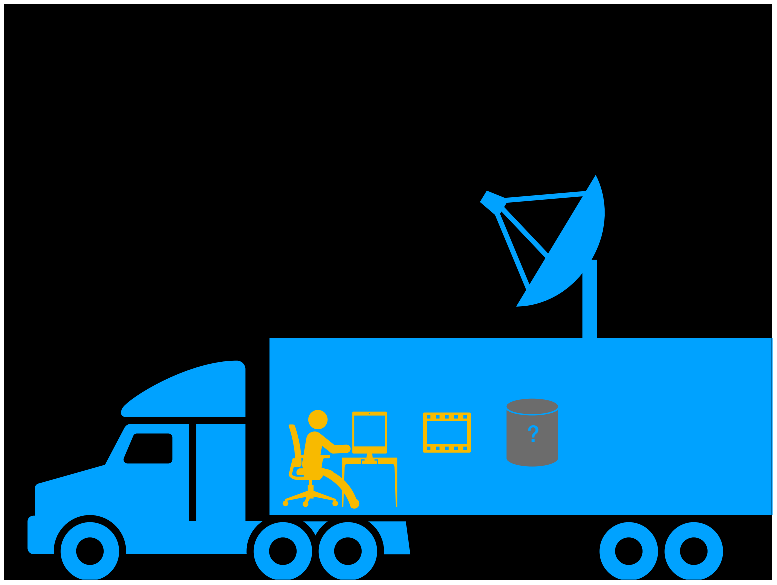
This was a very high-stress job. Nothing like recompiling your system during a commercial break of indeterminate length along with an SLA of 1/30th of a second for hours at a time with 10s of millions of people getting upset within a few hundreds of milliseconds when you mess it up...



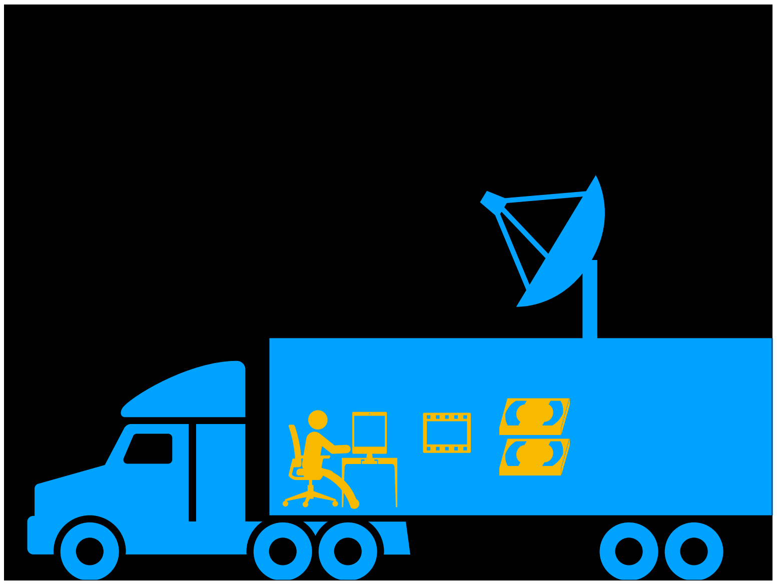
While TV had made a big shift to digital, the bandwidths simply weren't there to treat video the same as data
And ya, I had to sit in a truck trailer to get that done. Again, not enough bandwidth at the time to do it elsewhere!



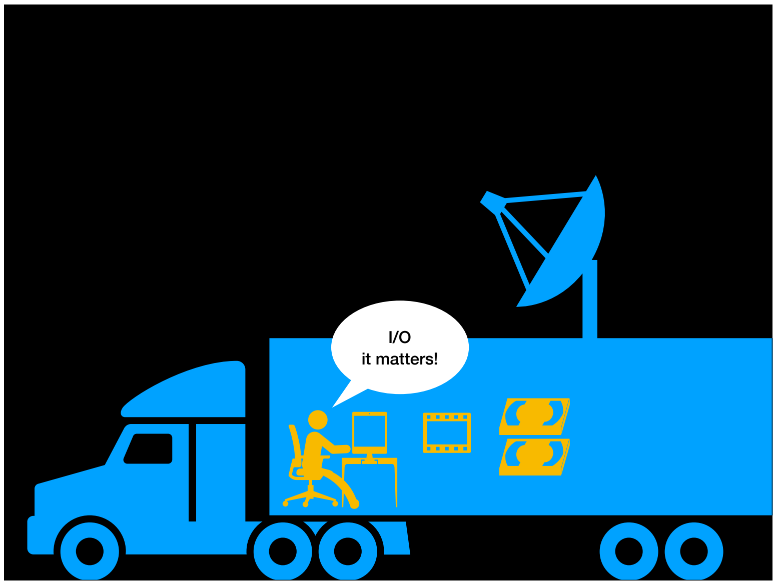
Inside the truck, we had to deal with the video somehow.
While one could build an array of disks that could handle the bandwidth, it was prohibitively expensive to do at any reasonable scale.
Disks, thus, were relegated to the still-important task of dealing with metadata: Where were the cameras pointed at that particular time, what was the accelerator-position of the number-17 car at that time, etc.



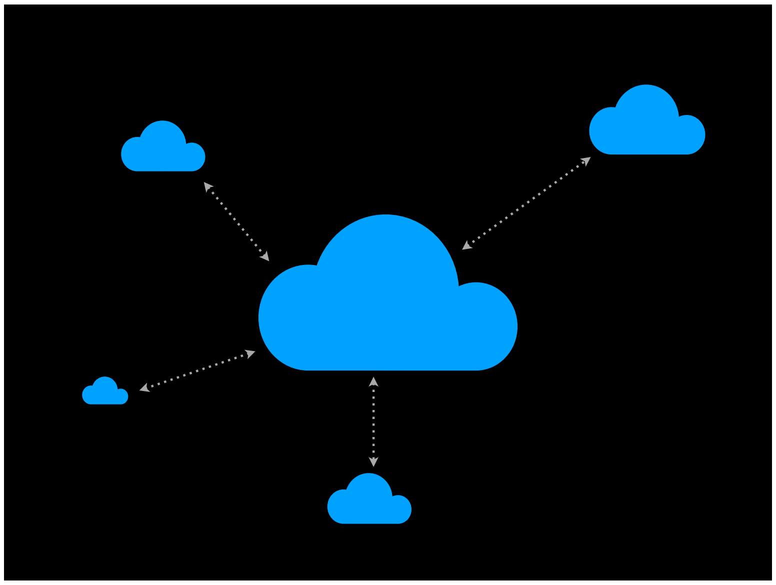
So, on what did we store the video?



We used DigiBeta tape. Video, it turns out, is primarily accessed linearly.
At 270Mb/s, it was challenging to move the video around inside the computers themselves, as the bus bandwidths of the computers of the time weren't that much larger.

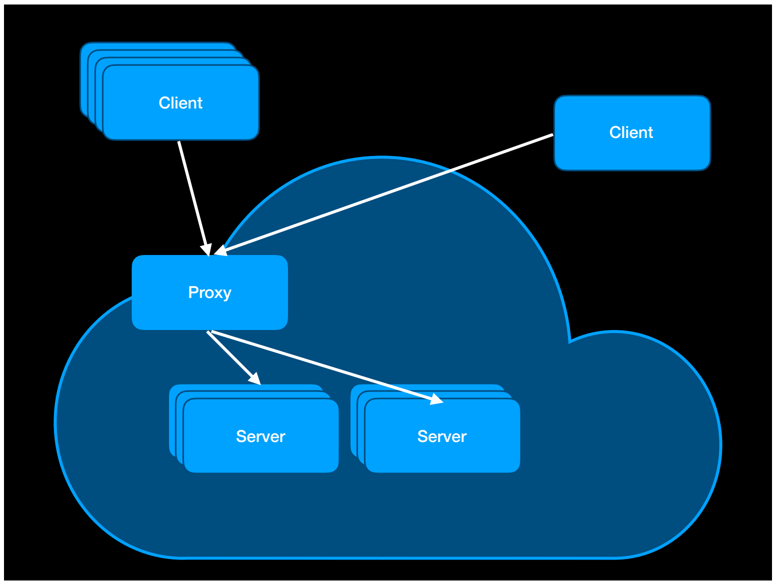


In both the data-at-rest and data-in-flight ways, the means by which the I/O was done mattered directly to whether or not we could meet the SLA.
(aside: This was stressful given multiple millions of people saw your mistakes within 2/3rds of a second, even when your mistake lasted only 1/30th of sec)

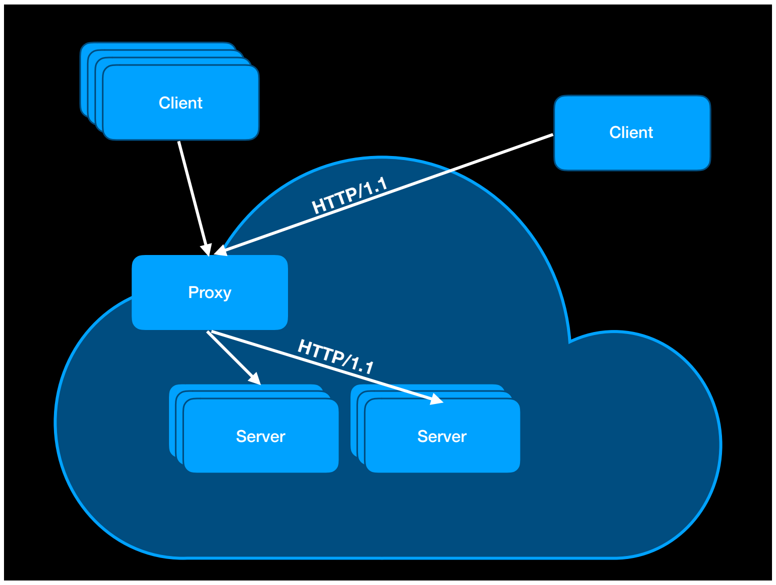


Fast forward, no pun intended, and I've moved to working at my new job, helping route HTTP requests to the correct servers, at "Internet" scale.

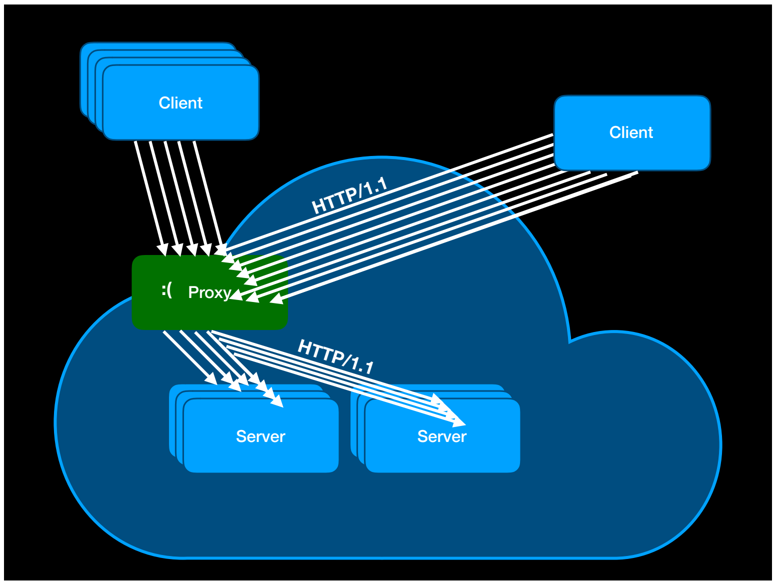
Fast forward, no pun intended, and I've moved to working at my new job, helping route HTTP requests to the correct servers, at "Internet" scale.



Fast forward, no pun intended, and I've moved to working at my new job, helping route HTTP requests to the correct servers, at "Internet" scale.

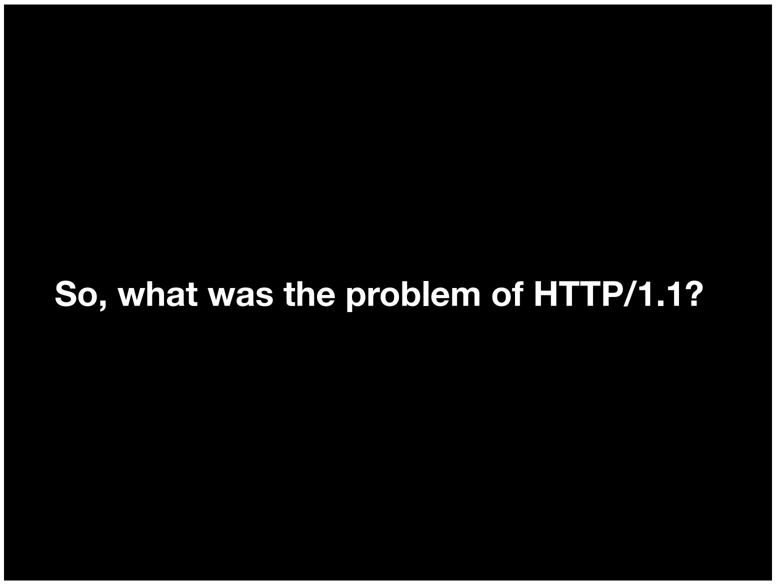


The world then was HTTP/1.1 That was the state of the art.



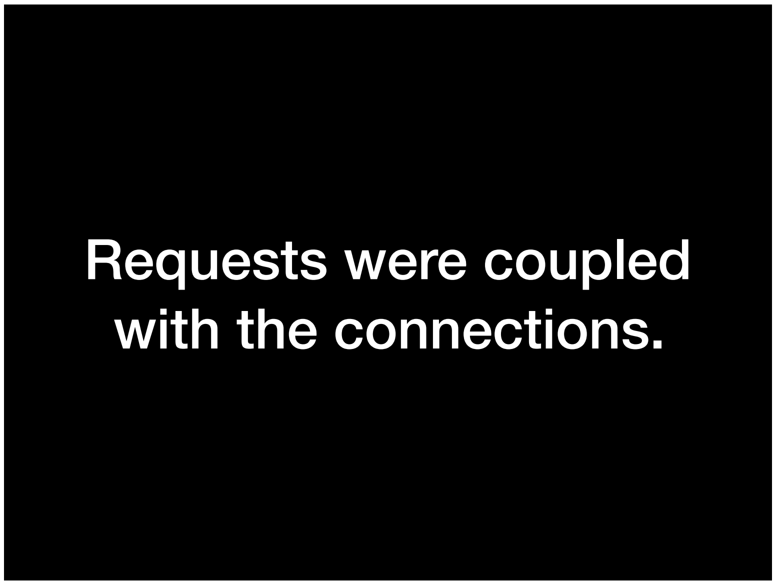
After a while it was apparent that there were issues.

After a while it was apparent that there were issues. Serious issues.



So, what was the problem of HTTP/1.1?

.. and that little animation was understating the issues.



Requests were coupled with the connections.

and, as a result, you'd see 60 connections needed to load a single site.
If you didn't have many IPs that you could use when phoning home, and you were running a CDN, you'd run out of ephemeral ports!
That meant you'd HoL block many people.

There was an abstraction missing.

To make a long story a short story...

The problems we were experiencing suggested...

...we needed a new session-layer abstraction.

and we weren't the only ones to realize that there were issues here. There were other standards/protocols/technologies

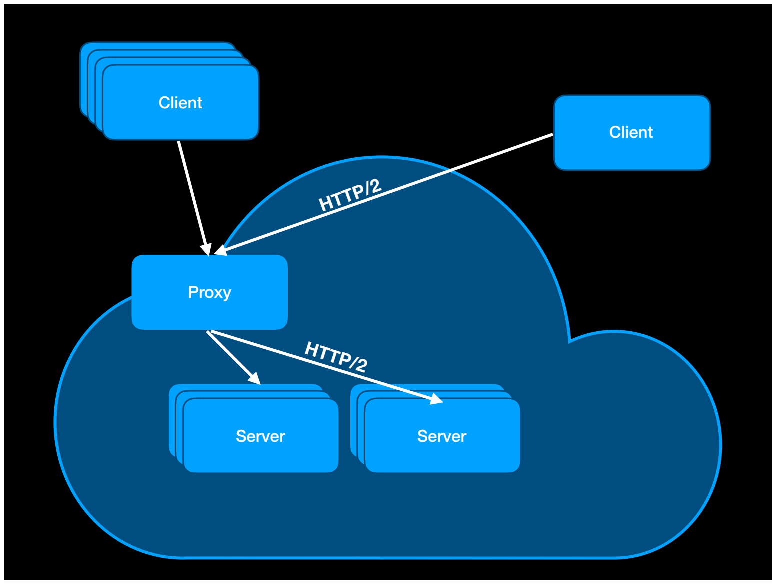
---

Enter SPDY

That is why we started work on what eventually became HTTP2.

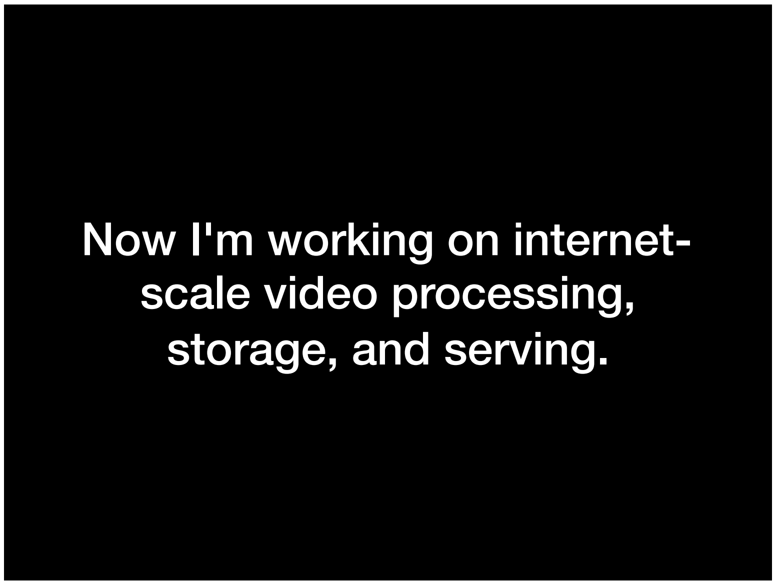
Enter HTTP/2

SPDY -> HTTP2
While I'm not going to talk about that much more, I will say that we knew that HTTP2 was going to be a springboard for further innovation, and we knew it was imperfect. We moved forward anyway.



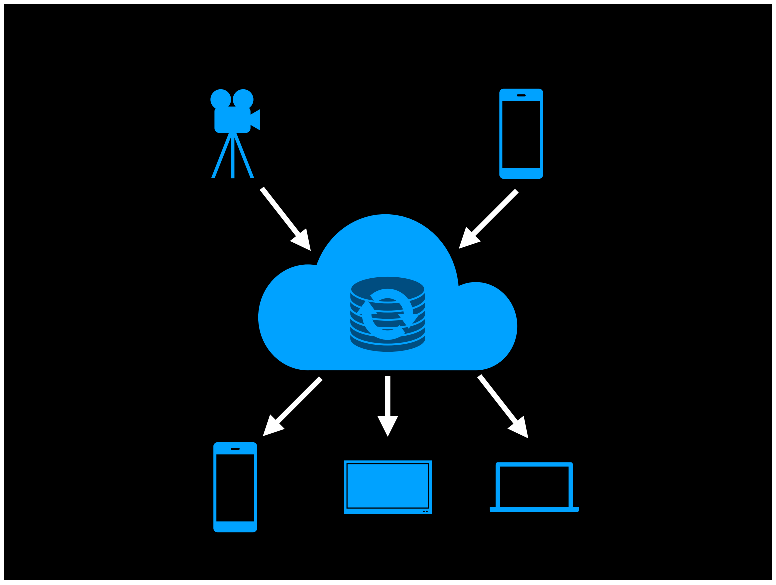
HTTP/2 was, in effect a session layer with the things that were needed to make multiplexing work well.
With HTTP/2, we added a session layer, de-coupling connections and requests.
Adding the session abstraction enabled compression, reuse of the TCP state, and allowed prioritization between requests.
Because we had multiplexing, the ephemeral port limit was no longer a big issue (that became the interplay of packet loss and HoL blocking on TCP, but that is another story, and it isn't QUIC... or

Fast forward again.

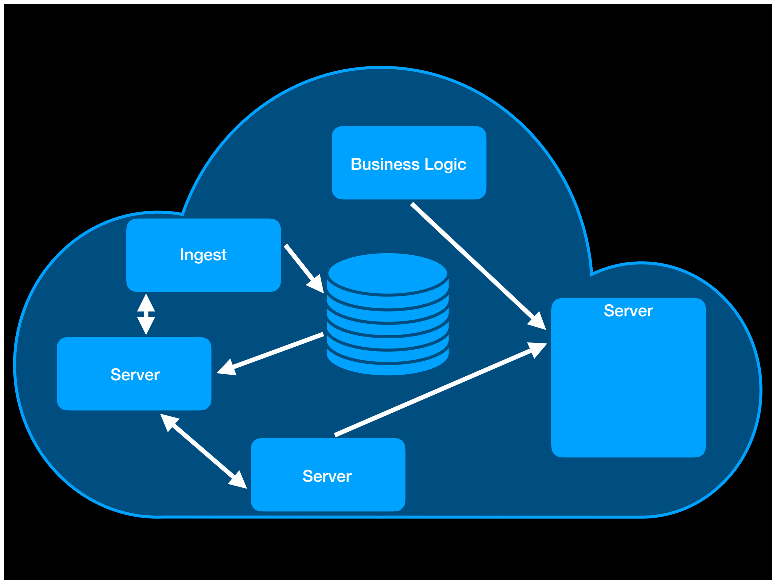


Now I'm working on internet-scale video processing, storage, and serving.

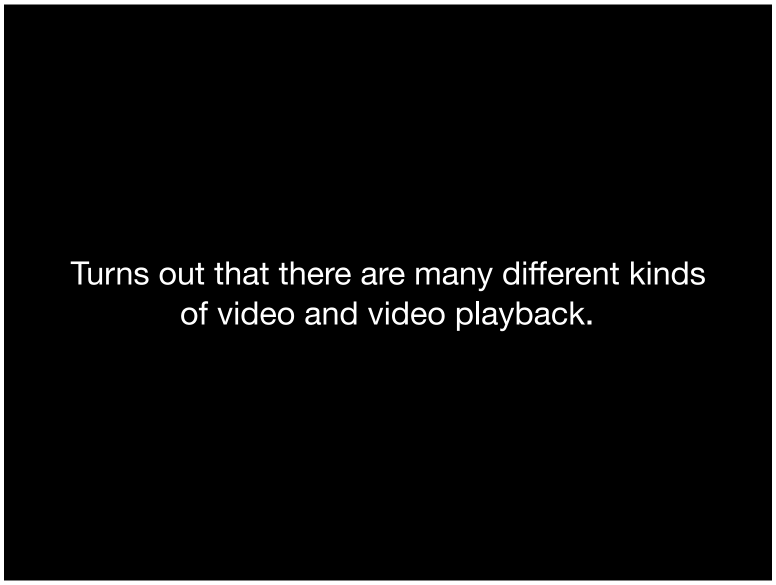
What is "internet-scale" storage?
If we take public numbers from a major video-sharing site, and if my arithmetic is correct (it often isn't), then on the order of a significant fraction of a Petabyte of storage ingested/day.
What does internet-scale serving mean?
It means you can induce significant packet loss if you mess it up.



Here, in this new world, we're worried about managing processing from many different inputs into many different outputs.
And while it looks peaceful and simple on the inside...



In reality, there were many different things going on inside, as one probably expects today.



Turns out that there are many different kinds of video and video playback.

Chatting with mom on the VC

has vastly different requirements than

playing a new (VoD) movie on the TV.

j/k ... Chatting with my dad on the VC also has different requirements.
Even more fun, the audio part has different requirements than the video part, and not all of the video parts are the same either...

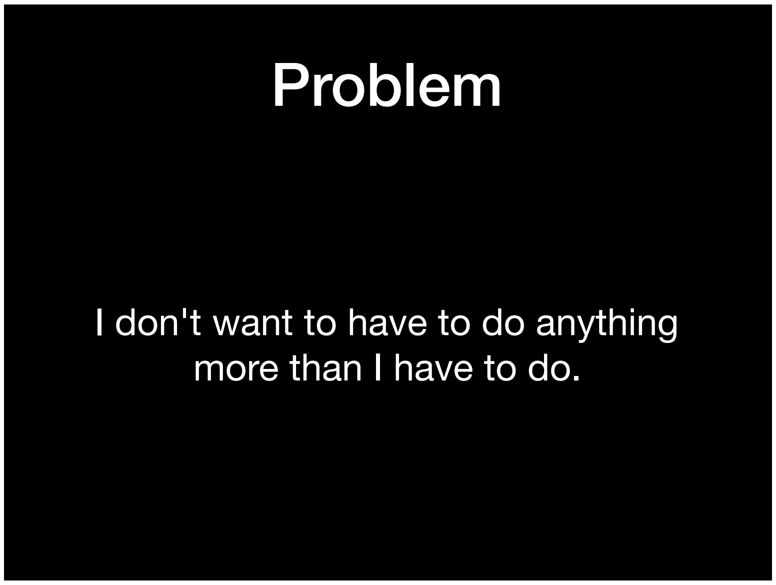
So much so that there are different protocols for VC as compared to VoD playback.

and Live too. Off the top of my head, some of the popular ones are RTMP, WebRTC, and there are variants of the DASH and HLS protocols (which are normally VoD, but have been changed to use mp4).

Problem

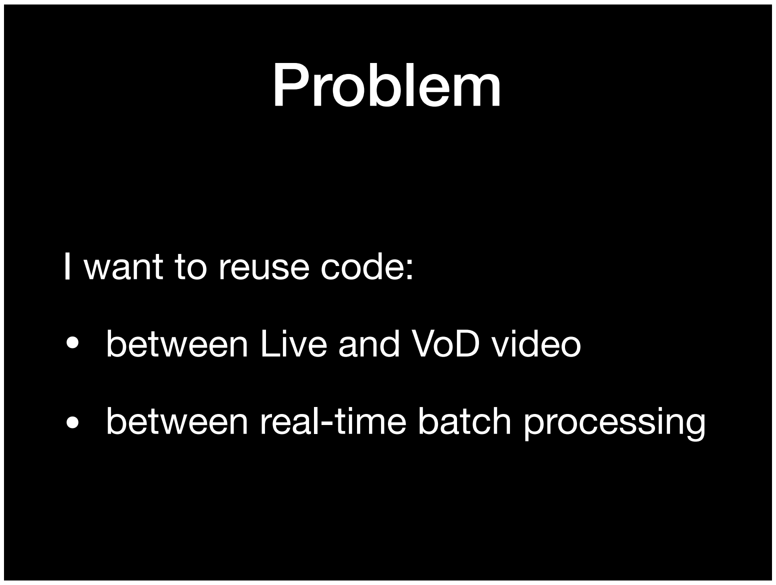
I'm lazy.

though hopefully not in a bad way...


Problem

I don't want to have to do anything more than I have to do.

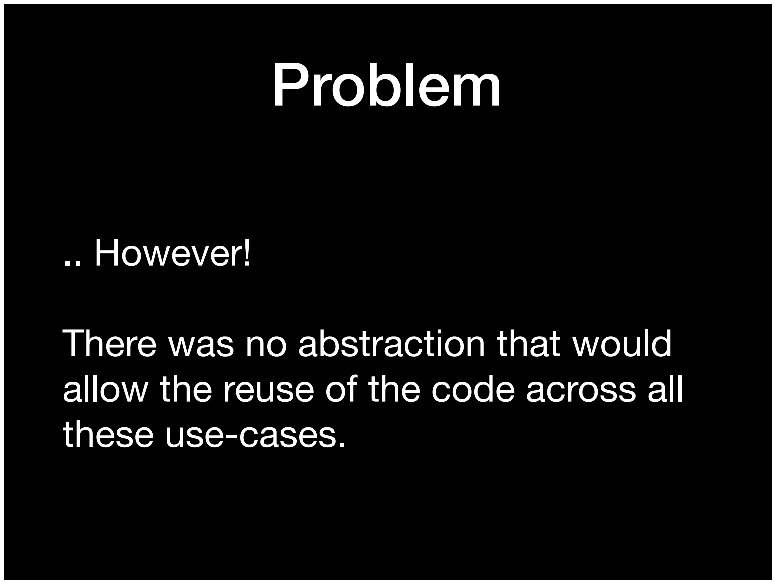
to get a particular job done.


Problem

I want to reuse code:

- between Live and VoD video
- between real-time batch processing

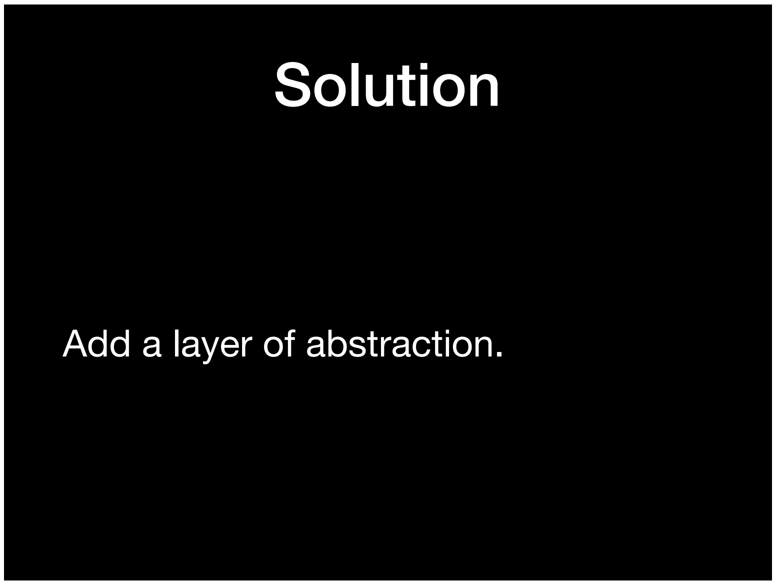
Doing the same thing over and over again is pretty boring, and also error prone.
From a statistical sense it also increases the amount of time necessary to achieve a particular amount of certainty about a result.
There is a lot of value in reuse, and I'd like to be able to extract it.


Problem

.. However!

There was no abstraction that would allow the reuse of the code across all these use-cases.

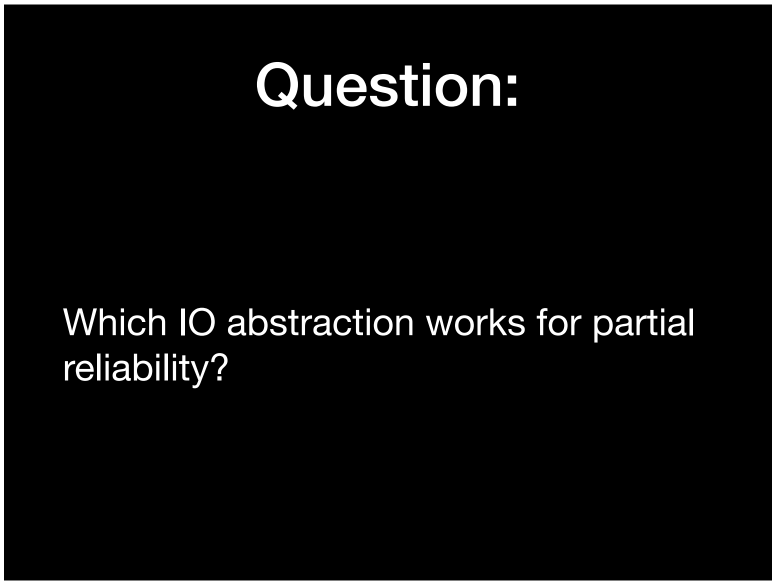
There were abstractions across the different use-cases, sure. Not good enough!
Things like VC emphasize low-jitter and transmitting the new data moreso than the old data.
When ingesting a new movie or other show, getting *all* the bytes reliably matters more than the jitter.
... and we want to be able to use/reuse the tools across all of these different use-cases.



Solution

Add a layer of abstraction.

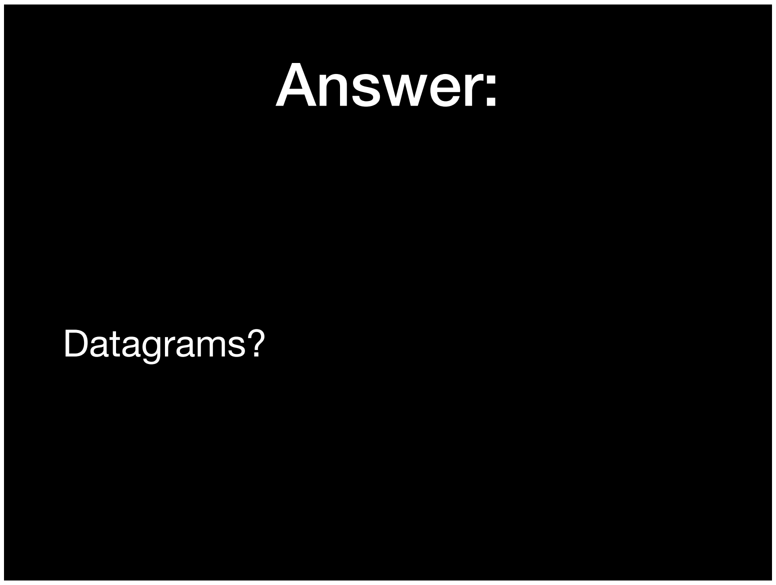
as usual...
.. after all every problem is solved with another layer of indirection, er, abstraciton!



Question:

Which IO abstraction works for partial reliability?

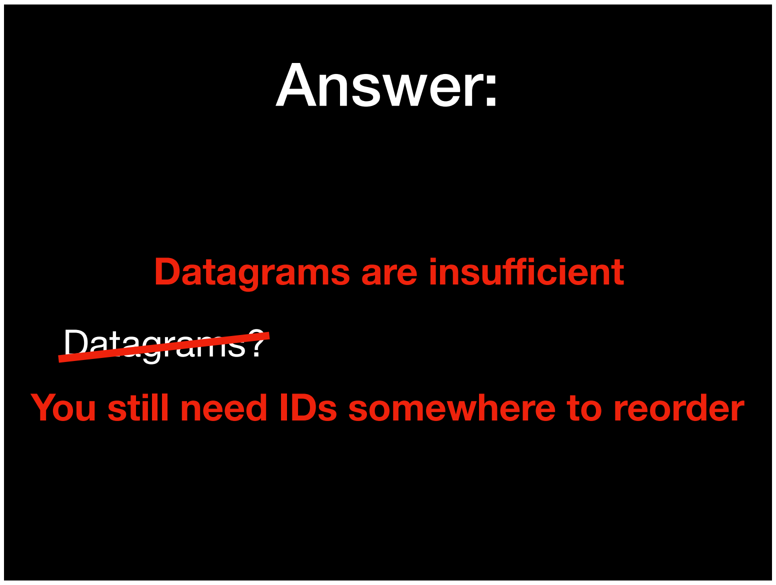
Lemme ask you a question.
Off the top of your head...



Answer:

Datagrams?

I'm guessing that many folks though of this one:



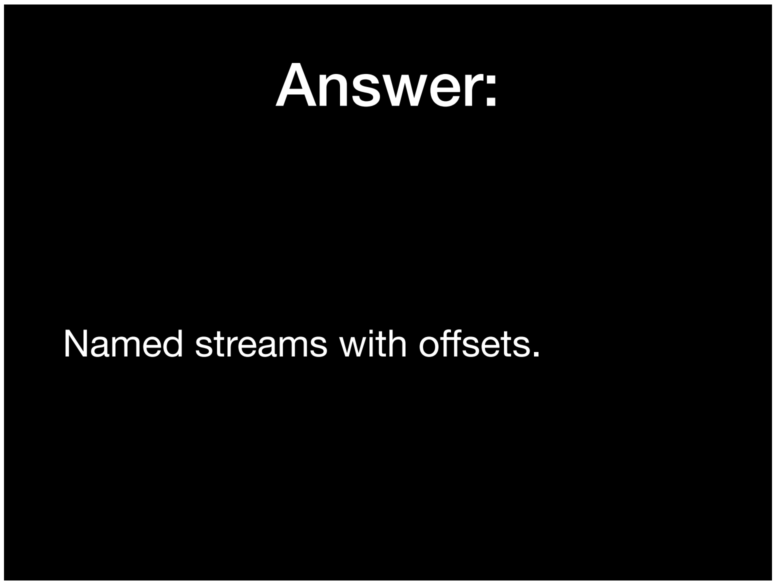
Answer:

Datagrams are insufficient

~~Datagrams?~~

You still need IDs somewhere to reorder

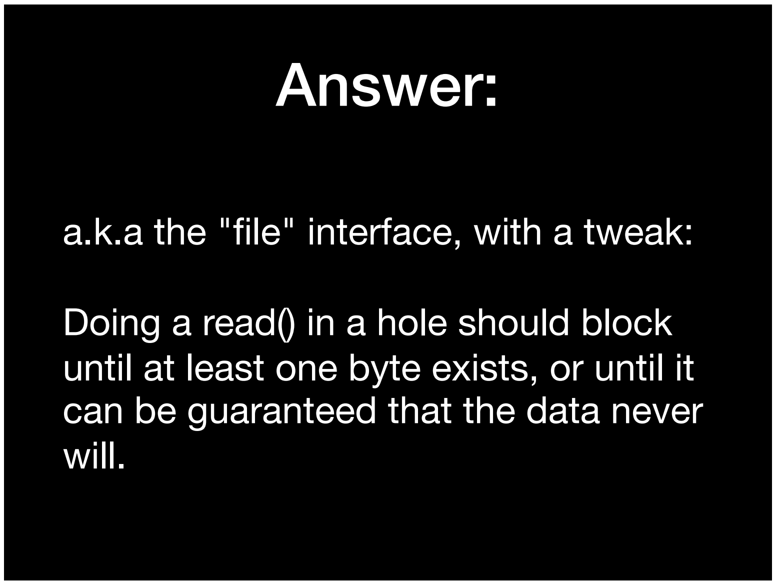
If partial reliability admits the possibility of reordering, then you need some kind of sequence-number or ID so you can put things back into the correct order.
Things like Video are inherently stateful. Try to play it out-of-order and you'll get things you probably didn't expect.
Just as partitioning is a fact of life, reordering is a fact of life, and data-loss is a fact of life.



Answer:

Named streams with offsets.

So, I think instead that a superior method is to use.. named streams with offsets. Named so it can be addressed appropriately. Crossing the streams here should be avoided.



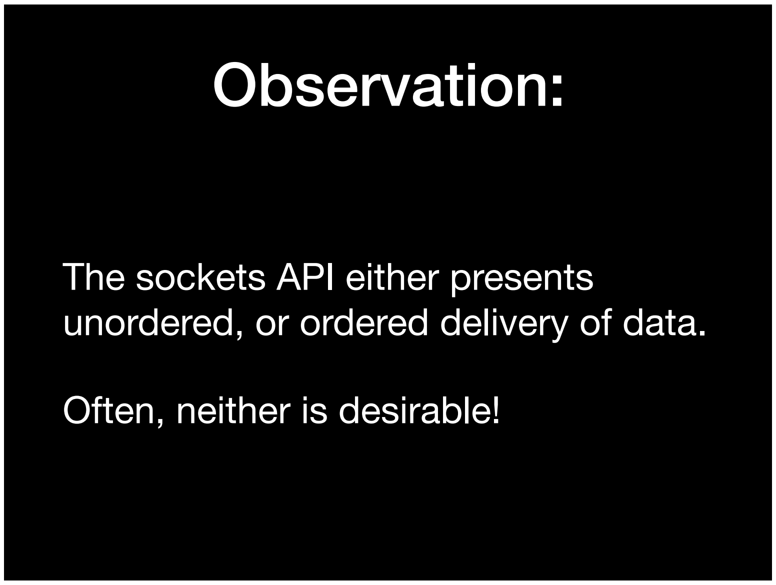
Answer:

a.k.a the "file" interface, with a tweak:

Doing a read() in a hole should block until at least one byte exists, or until it can be guaranteed that the data never will.

Returning OOB should happen only when the max-offset of the file is already known.
Note I didn't say "file size", but am being more specific, because what is the size of a file that has a max-offset of 1MB, but only has one byte written and the others are all unknown?

There are *details*. In cases where your doing things like TCP, you could get OOB because of reading out of the range of available addresses, which is not quite the same as not having data, and it is different from getting an error suggesting the connection is terminated, but I digress a bit...

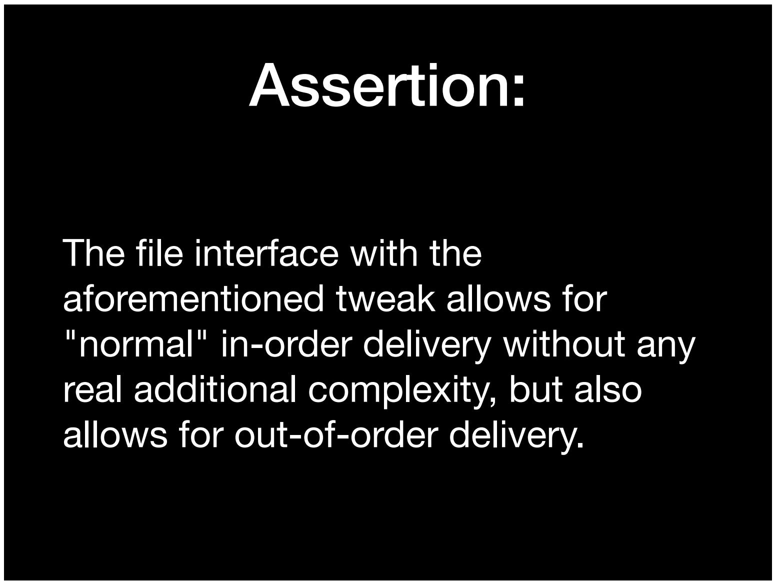


Observation:

The sockets API either presents unordered, or ordered delivery of data.

Often, neither is desirable!

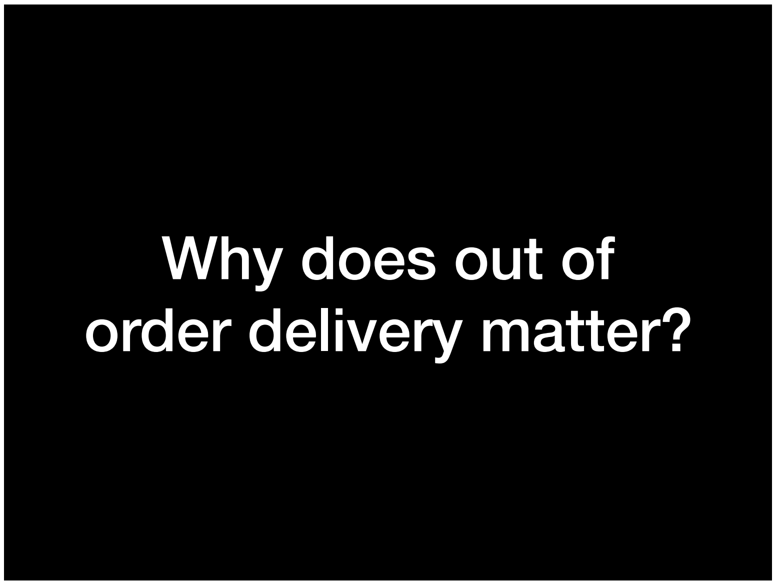
Most data which encompasses multiple packets in most sessions is ordered. Presenting it out-of-order, i.e. scatter-gather potentially makes sense, and is often a superior tradeoff to presenting a HoL blocking/high-jitter interface.
As a reminder, variance is the bane of existence for many latency-sensitive applications!



Assertion:

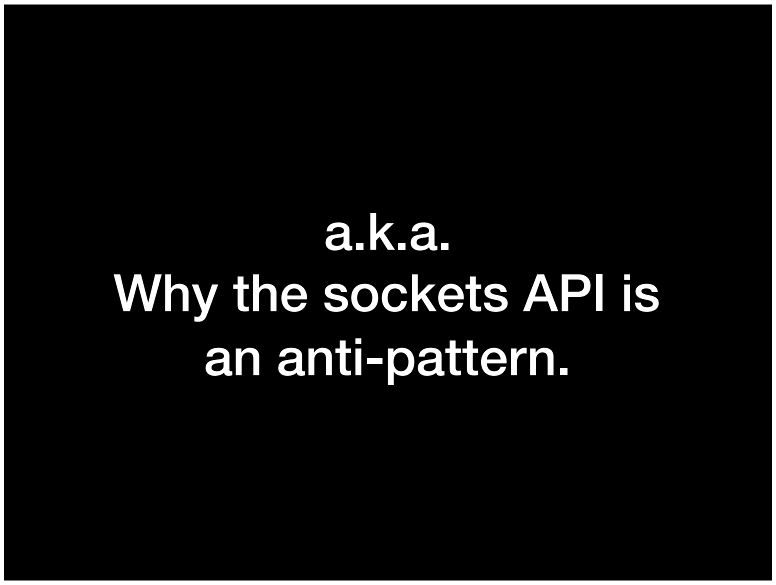
The file interface with the aforementioned tweak allows for "normal" in-order delivery without any real additional complexity, but also allows for out-of-order delivery.

If it can do what sockets can do (i.e. telling me what is new), and what files can do (scatter-gather/out-of-order or random access), then I have an abstraction that can work across all of those lovely video use-cases.



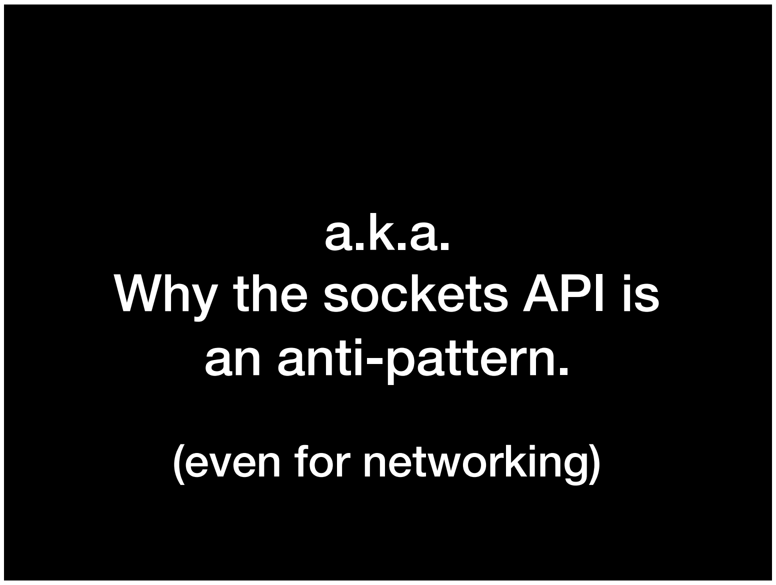
Why does out of order delivery matter?

In many cases of for video, multicast isn't available.
You get unicast, and but thankfully we have CDNs which means we have ways of reducing the total network work via caching.



a.k.a. Why the sockets API is an anti-pattern.

<hit next, then talk again>

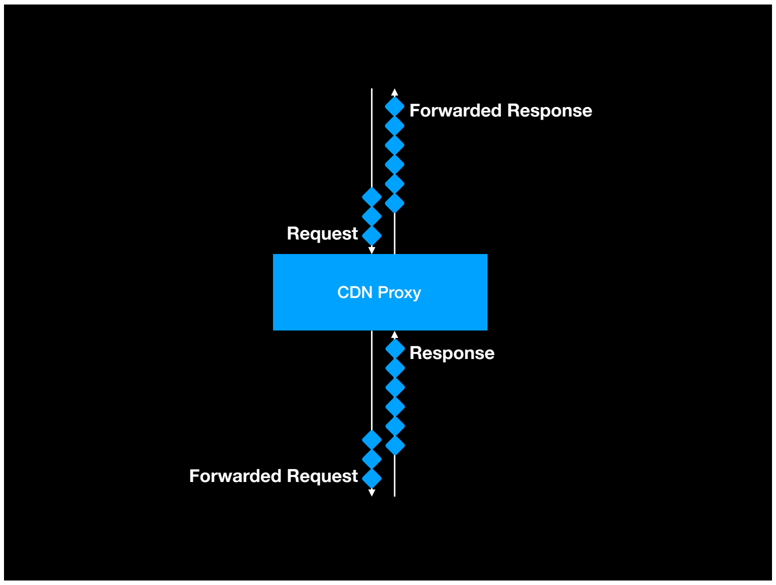


a.k.a. Why the sockets API is an anti-pattern.

(even for networking)

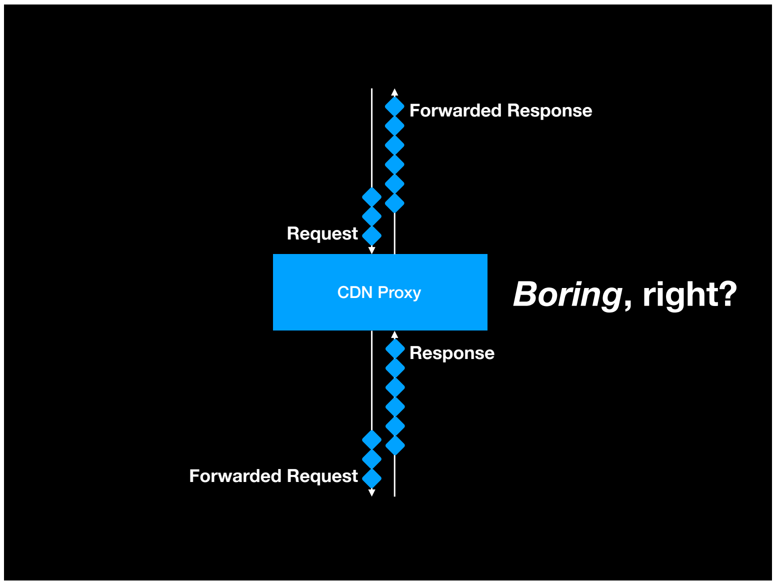
If you wouldn't use an in-order API for reading files, why do it for the network? This is why I have problems with the sockets API. It conflates having an ordering (which we'll often call having an address or defining an address space) with ordered delivery.

To put it another way, imagine that to read k bytes at offset N in a file you are required to read (and potentially discard) all N-1 bytes first. Wouldn't it be far cheaper to just start reading at offset N?

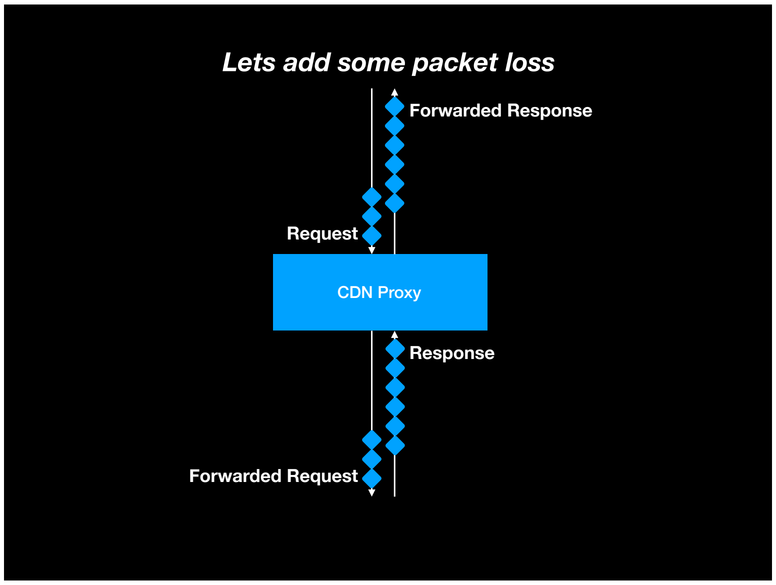


Lets zoom into a proxy and talk packets. As a reminder, these proxies make the internet work with reasonable cost/efficiency/latency.

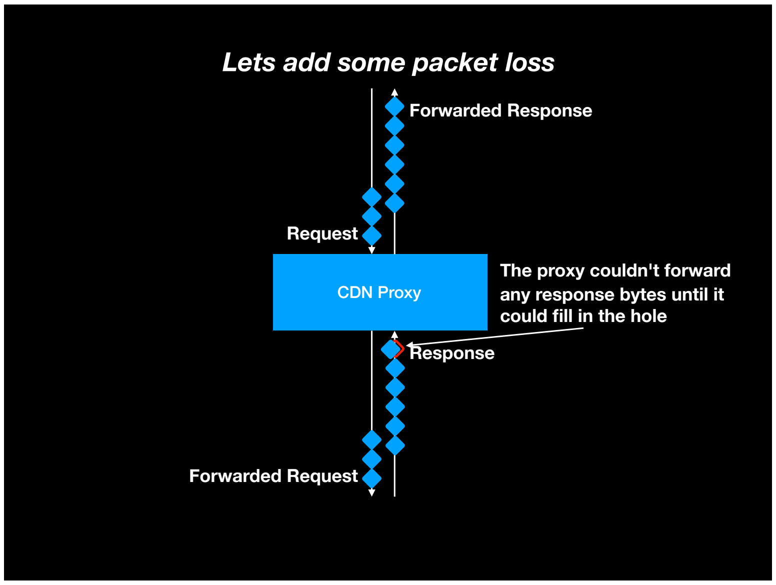
These caches, however, can represent bottlenecks when you do L7 interpretation in the real world via a sockets (in-order) API.



Boring because if the world was that easy, we wouldn't be talking about "the edge"...
The reality is that there is more going on here.

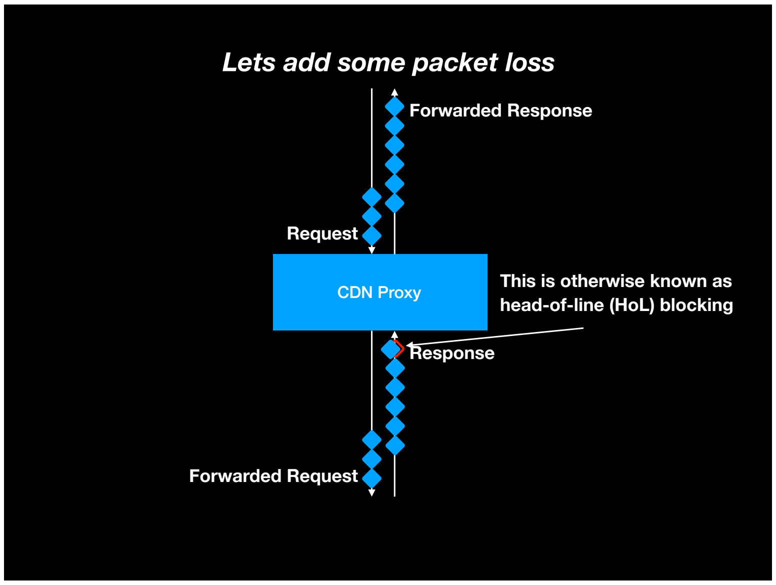


In the real world, there is packet loss. In many cases this can happen even when there is no channel contention based on how the congestion controller does bandwidth probing...


Lets add some packet loss

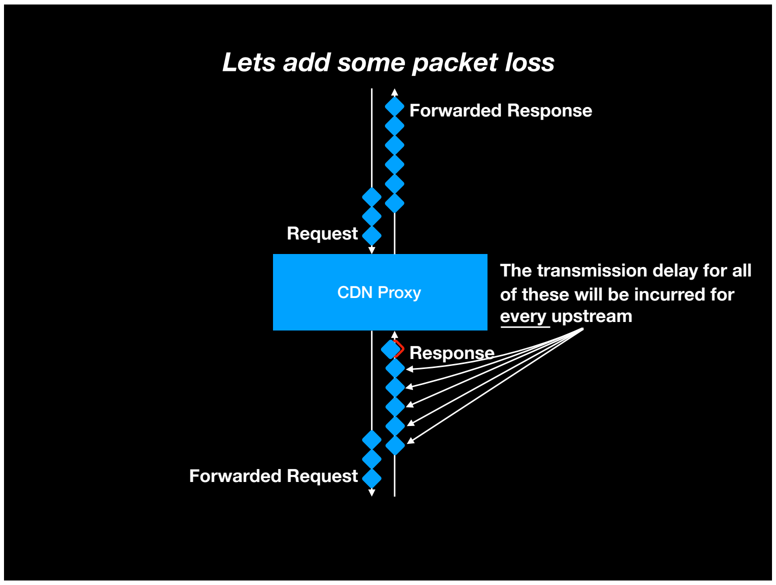
The proxy couldn't forward any response bytes until it could fill in the hole

---


Lets add some packet loss

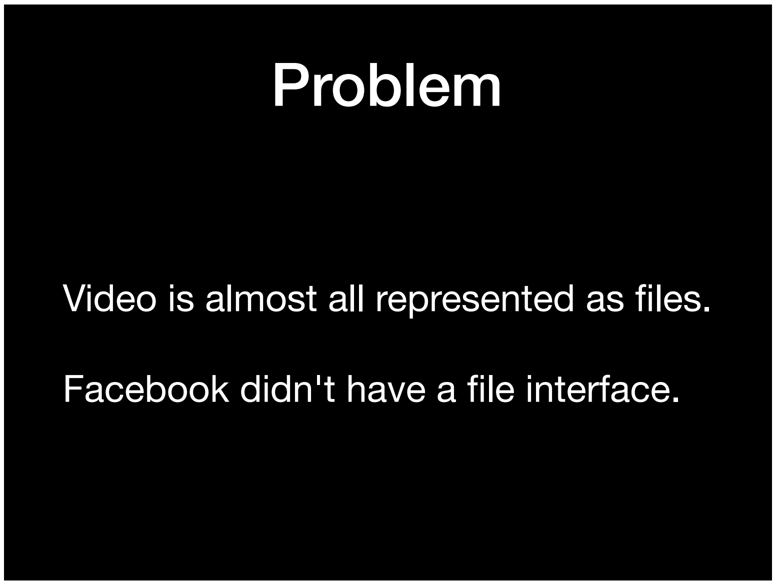
This is otherwise known as head-of-line (HoL) blocking

And HoL blocking causes jitter/variance, which causes us to fail to use the full channel goodput towards<next>


Lets add some packet loss

The transmission delay for all of these will be incurred for every upstream

and any such HoL blocking will impact all upstream (i.e. downloading) clients.

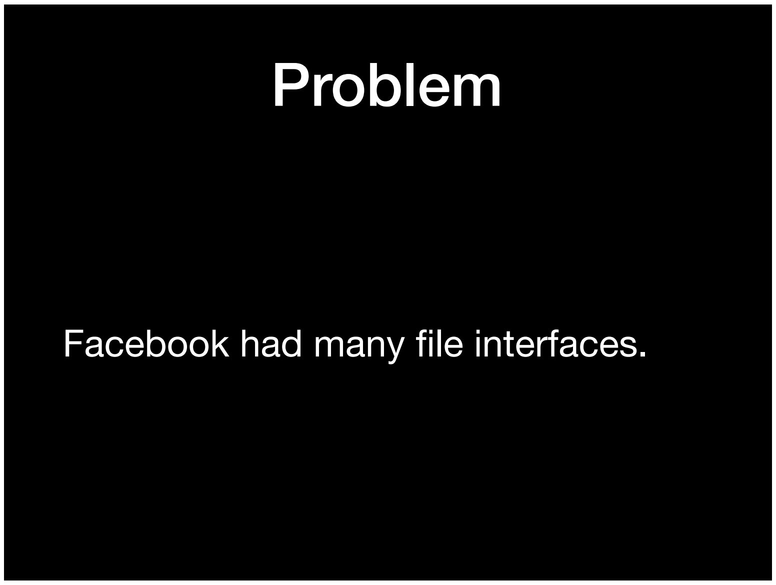


Problem

Video is almost all represented as files.

Facebook didn't have a file interface.

Fundamental issue if you're doing video where most video is represented as streams or files.

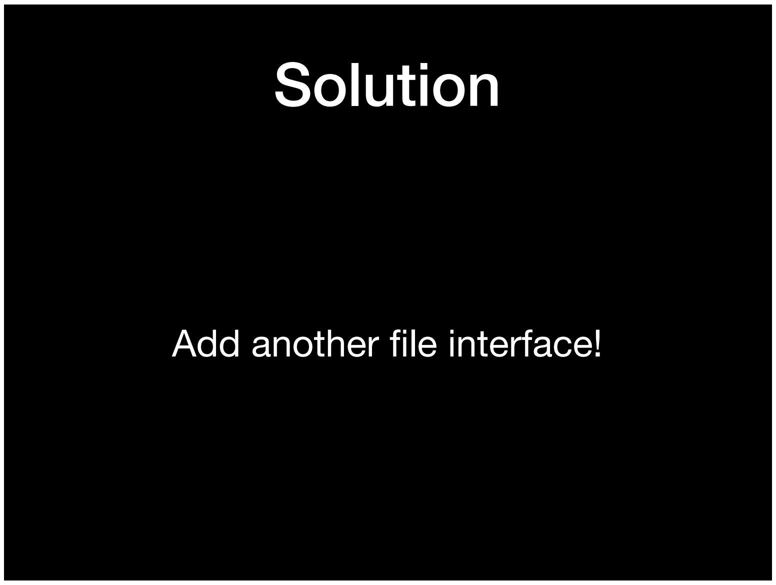


Problem

Facebook had many file interfaces.

No, instead... <many file interfaces>

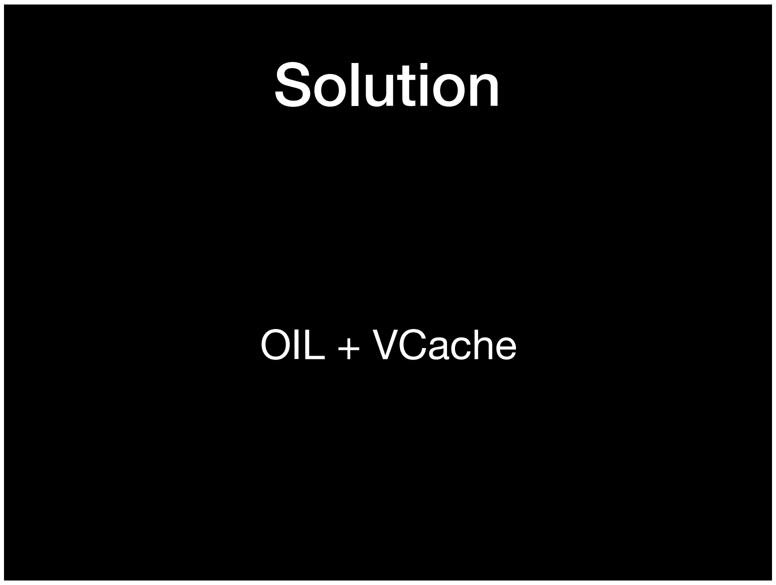
Haystack, HDFS, F4, local filesystem...



Solution

Add another file interface!

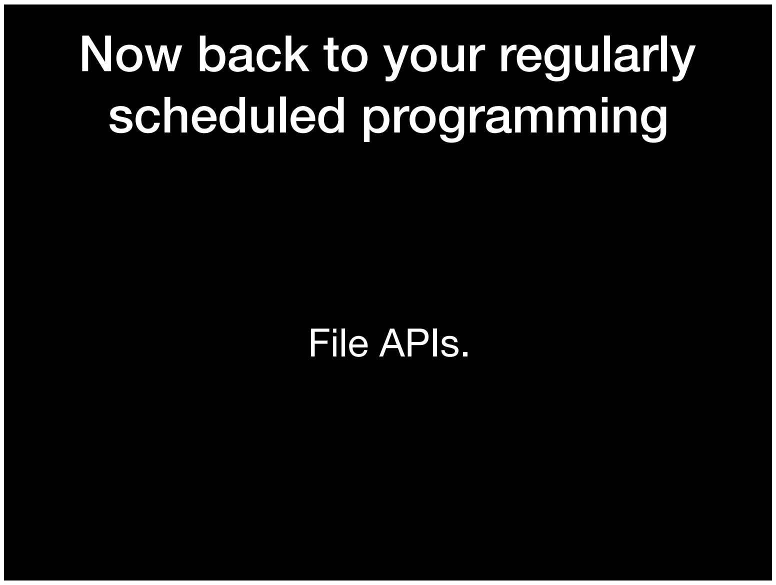
Yet another file interface!



Solution

OIL + VCache

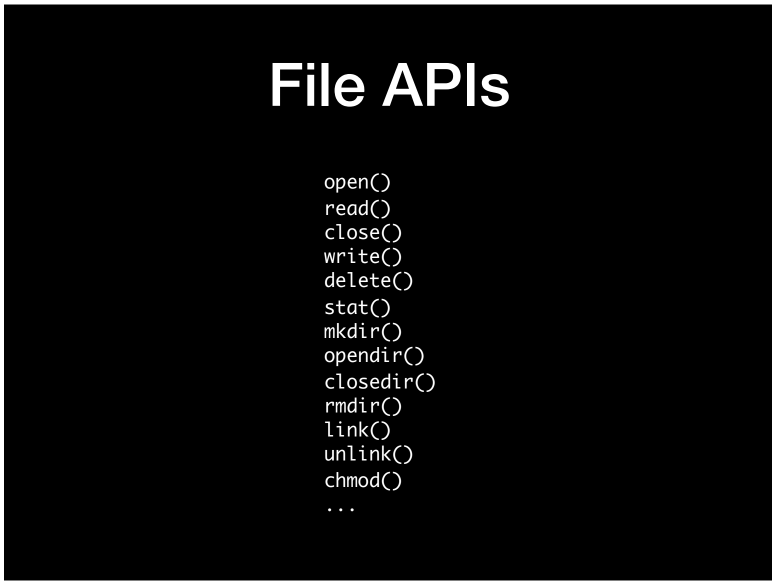
Hopefully this one is different... we'll get into a bunch of the differences, but one of the interesting things is that we hope that this is expressive enough to wrap/ encapsulate most public interfaces for most filesystems and/or object stores.



Now back to your regularly scheduled programming

File APIs.

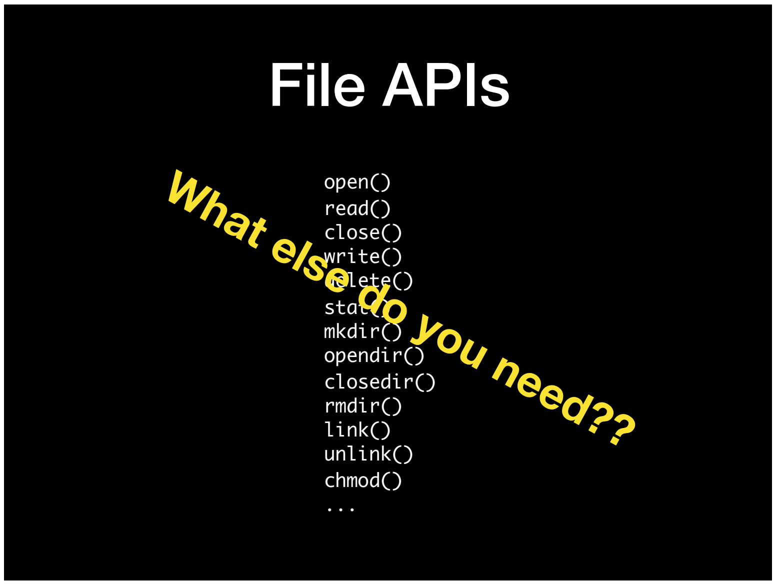
I heard that there are folks here who like file APIs?
Lets talk about file APIs.



File APIs

```
open()
read()
close()
write()
delete()
stat()
mkdir()
opendir()
closedir()
rmdir()
link()
unlink()
chmod()
...
```

here are some sample calls we're probably all familiar with...
and my question to you is...

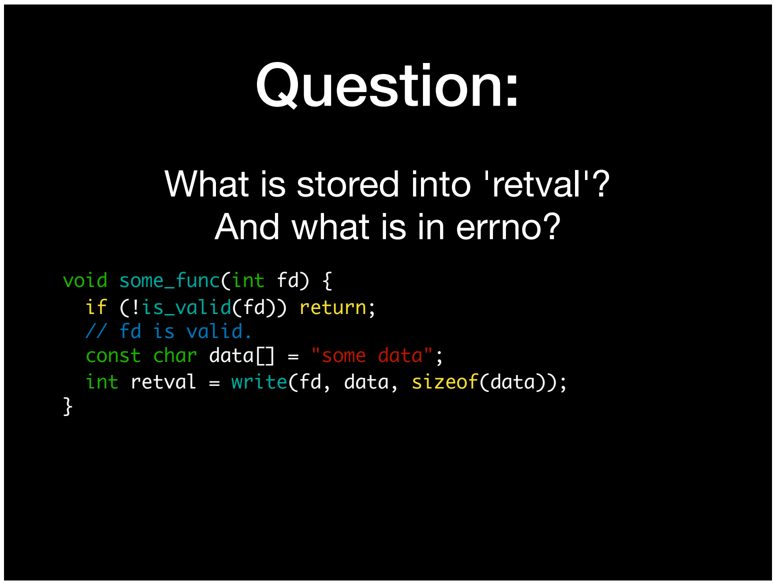


File APIs

```
open()
read()
close()
write()
delete()
stat()
mkdir()
opendir()
closedir()
rmdir()
link()
unlink()
chmod()
...
```

What else do you need??

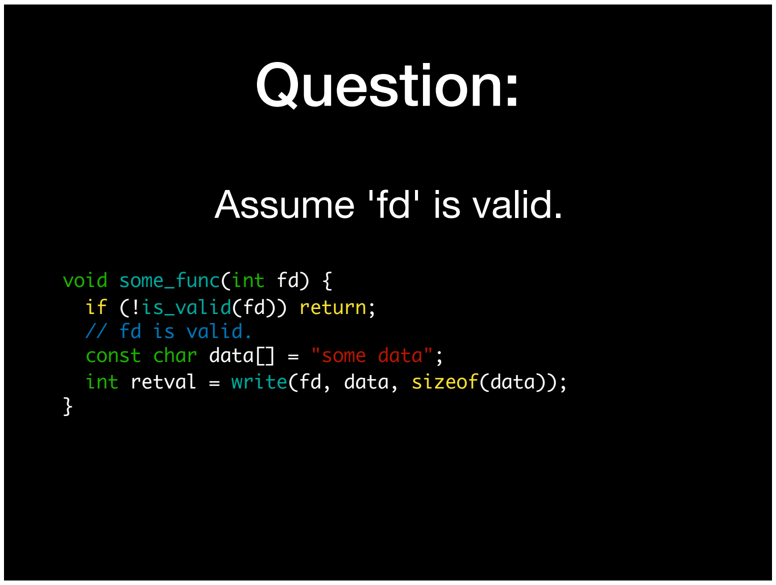
Why would you want to add more? Isn't that enough?
Lets answer that question with a question...



Question:

What is stored into 'retval'?
And what is in errno?

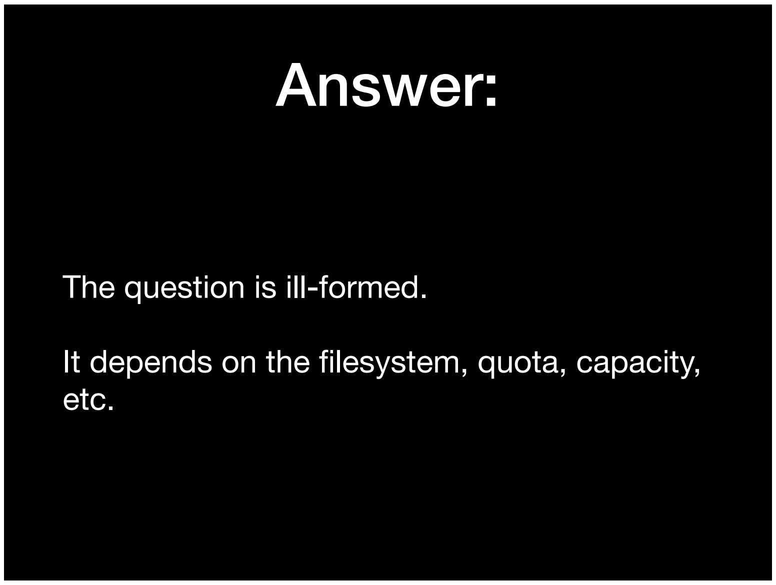
```c
void some_func(int fd) {
  if (!is_valid(fd)) return;
  // fd is valid.
  const char data[] = "some data";
  int retval = write(fd, data, sizeof(data));
}
```



Question:

Assume 'fd' is valid.

```c
void some_func(int fd) {
  if (!is_valid(fd)) return;
  // fd is valid.
  const char data[] = "some data";
  int retval = write(fd, data, sizeof(data));
}
```

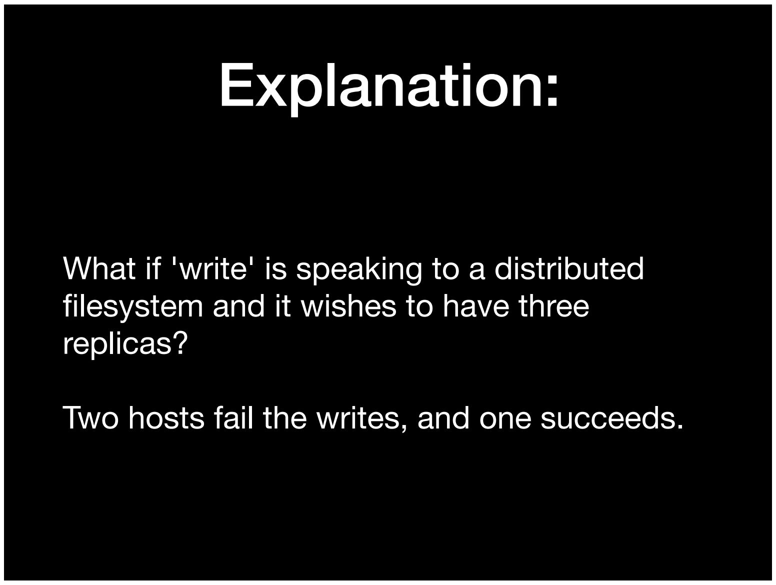


Answer:

The question is ill-formed.

It depends on the filesystem, quota, capacity, etc.

Arguably, this is a trick question. There are many possibilities depending on filesystem tradeoffs, capacity, etc.



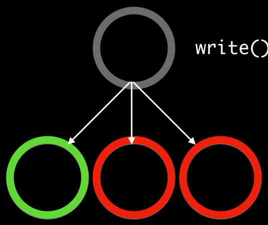
Explanation:

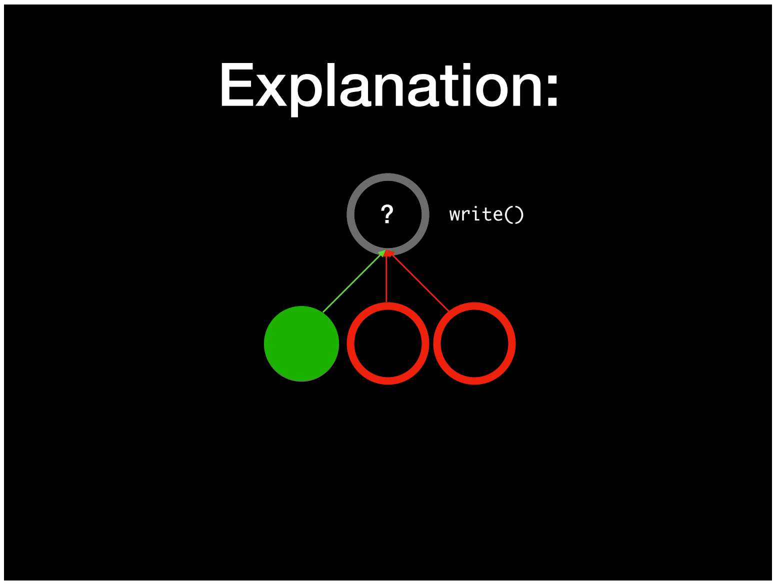
What if 'write' is speaking to a distributed filesystem and it wishes to have three replicas?

Two hosts fail the writes, and one succeeds.

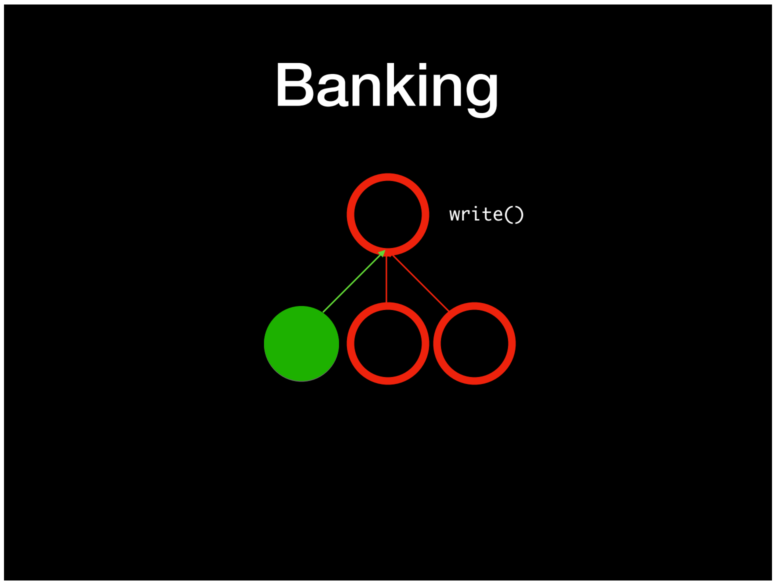
Here is a diagram <next>

Explanation:

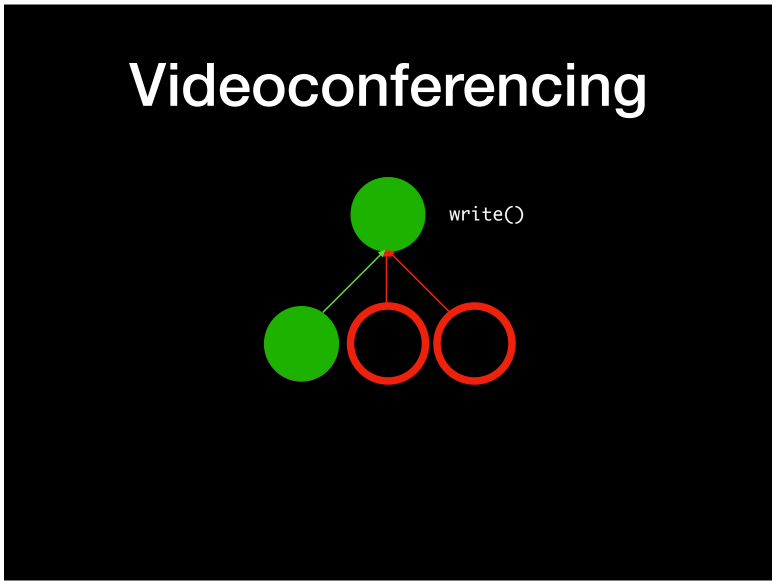



Explanation:

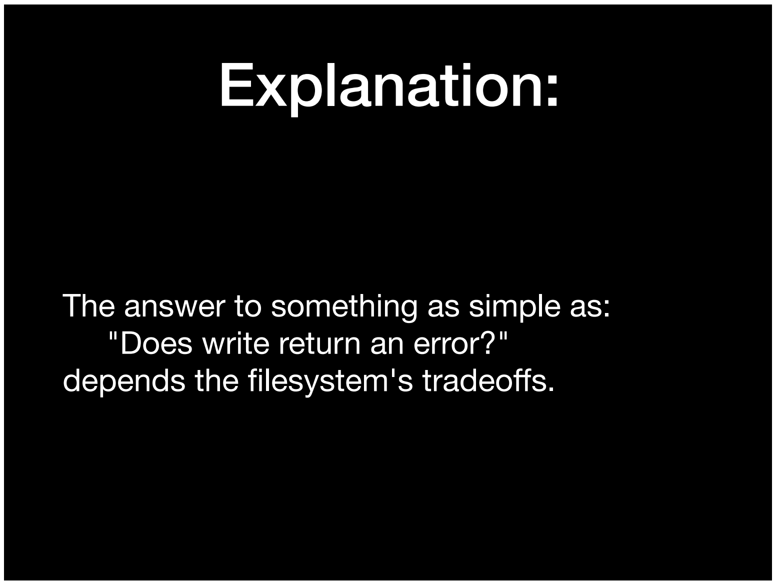
what will the return value be??


Banking

In the banking use-case, we didn't get quorum.
Since we probably care more about consistency than availability, this is a no-go.
Failing the write (returning an error) makes the most sense.


Videoconferencing

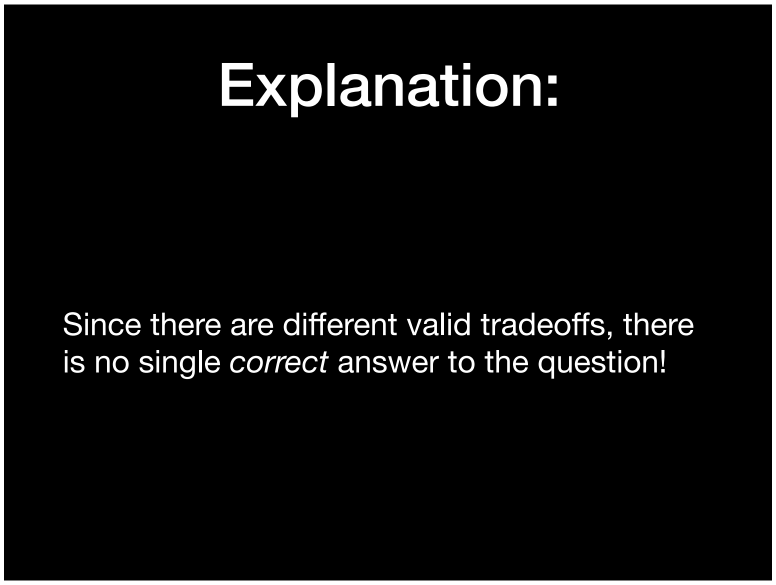
In the videoconferencing case, however, you're good to go if you get even one out there. Availability trumps consistency most of the time.
Returning success in this case makes sense.



Explanation:

The answer to something as simple as:
"Does write return an error?"
depends the filesystem's tradeoffs.

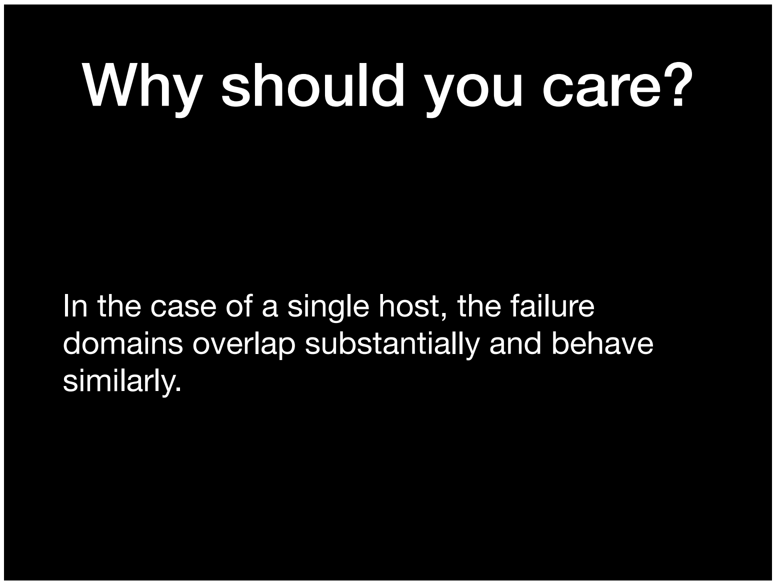
... Or, the application tradeoffs.
And we know given CAP that we'll have to make some tradeoffs given that the "p" in CAP isn't really optional.



Explanation:

Since there are different valid tradeoffs, there is no single *correct* answer to the question!

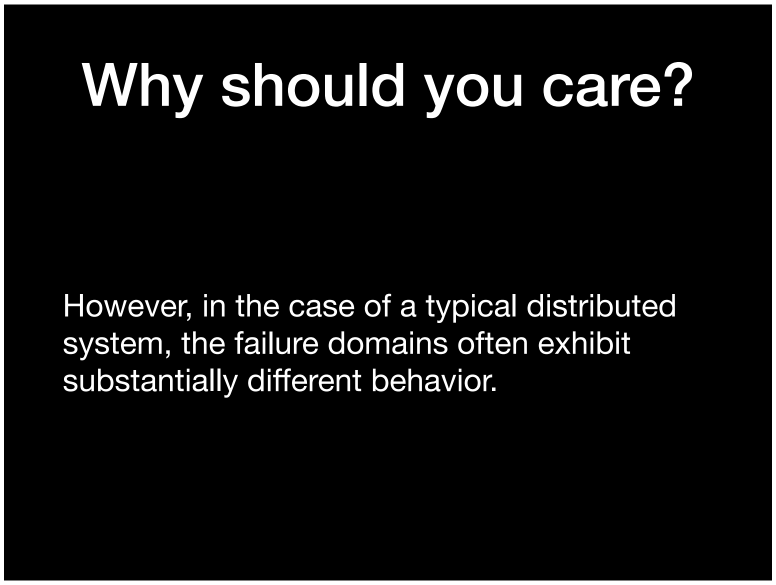
as we already saw with banking vs VC use-cases.



Why should you care?

In the case of a single host, the failure domains overlap substantially and behave similarly.

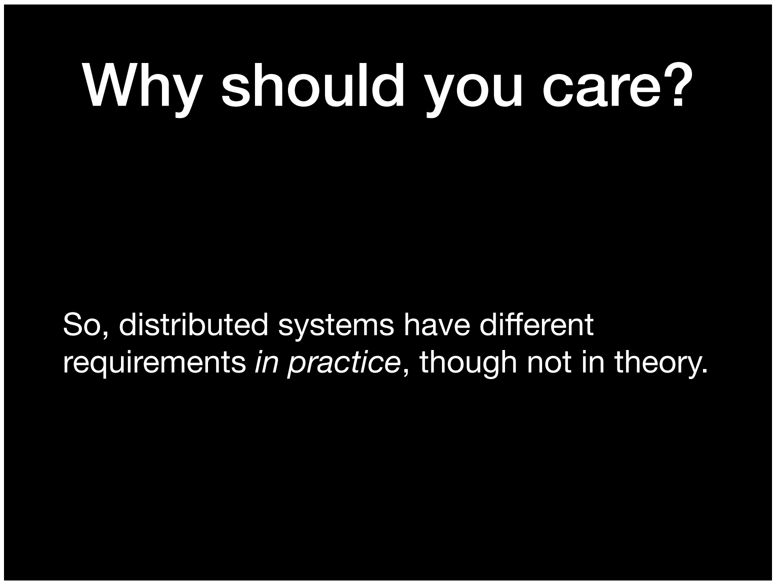
aaaand You should care about this because CAP suggests you can't have it all at the same time.



Why should you care?

However, in the case of a typical distributed system, the failure domains often exhibit substantially different behavior.

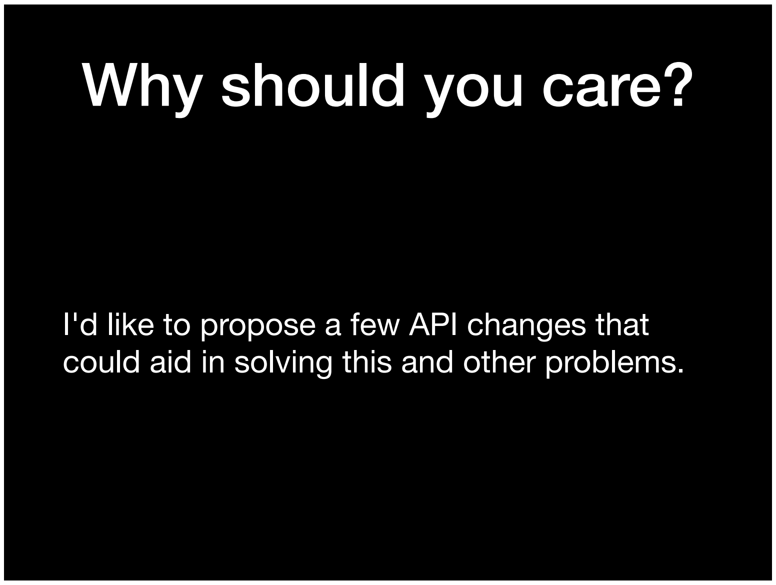
We worry about the 'P' in cap a lot, as it seems to be mostly unavoidable at large-scale deployments.



Why should you care?

So, distributed systems have different requirements *in practice*, though not in theory.

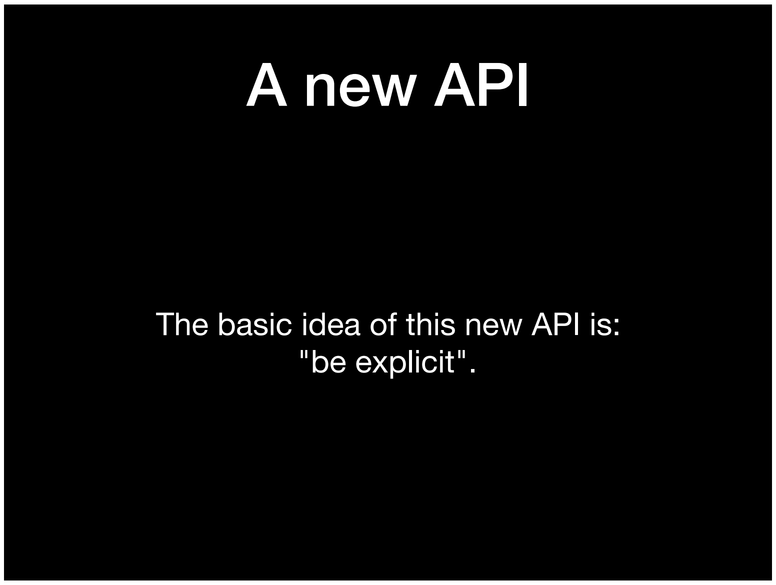
probability of failure in distributed systems is non-theoretical. For the system to be practical, you must address it.



Why should you care?

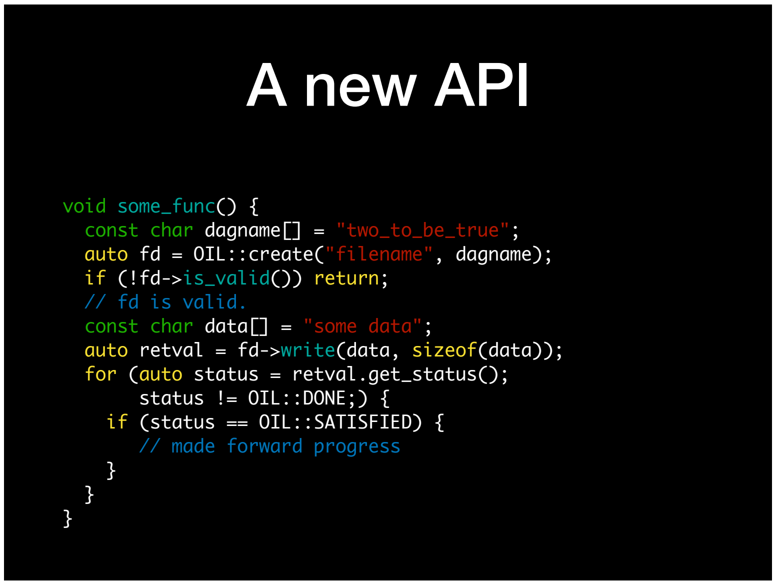
I'd like to propose a few API changes that could aid in solving this and other problems.

More than just having read() block when the data is missing (though that is still cool)...



A new API

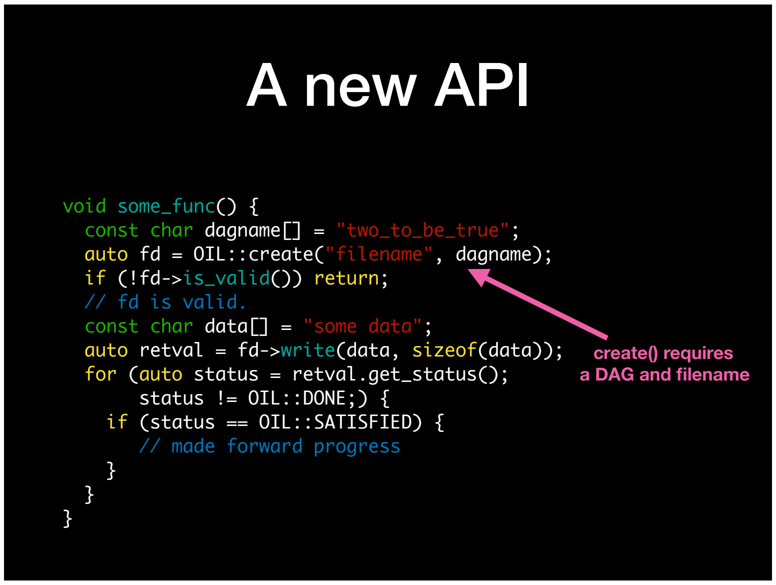
The basic idea of this new API is:
"be explicit".



A new API

```
void some_func() {
  const char dagname[] = "two_to_be_true";
  auto fd = OIL::create("filename", dagname);
  if (!fd->is_valid()) return;
  // fd is valid.
  const char data[] = "some data";
  auto retval = fd->write(data, sizeof(data));
  for (auto status = retval.get_status();
       status != OIL::DONE;) {
    if (status == OIL::SATISFIED) {
      // made forward progress
    }
  }
}
```

I'll point out that you can have trivial shims/adapters which make this look like "Ye Olde File API" that everyone is used to, but I'm interested in the slightly-lower layer here, and so that is what I'm showing.

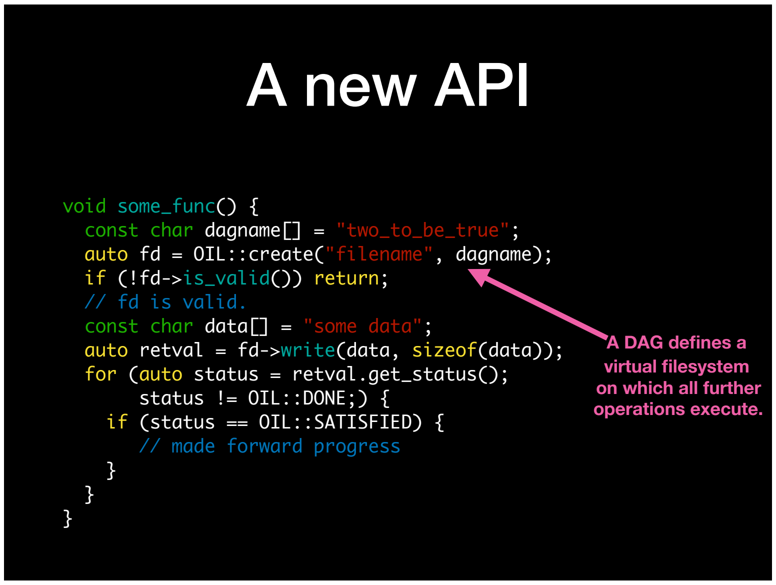


A new API

```
void some_func() {
  const char dagname[] = "two_to_be_true";
  auto fd = OIL::create("filename", dagname);
  if (!fd->is_valid()) return;
  // fd is valid.
  const char data[] = "some data";
  auto retval = fd->write(data, sizeof(data));
  for (auto status = retval.get_status();
       status != OIL::DONE;) {
    if (status == OIL::SATISFIED) {
      // made forward progress
    }
  }
}
```

create() requires a DAG and filename

Pointing out that the create/open call is a bit different.
It requires a specification-- in this case a name -- of a "dag", which represents the I/O policy that will be used.
.. in particular <next>

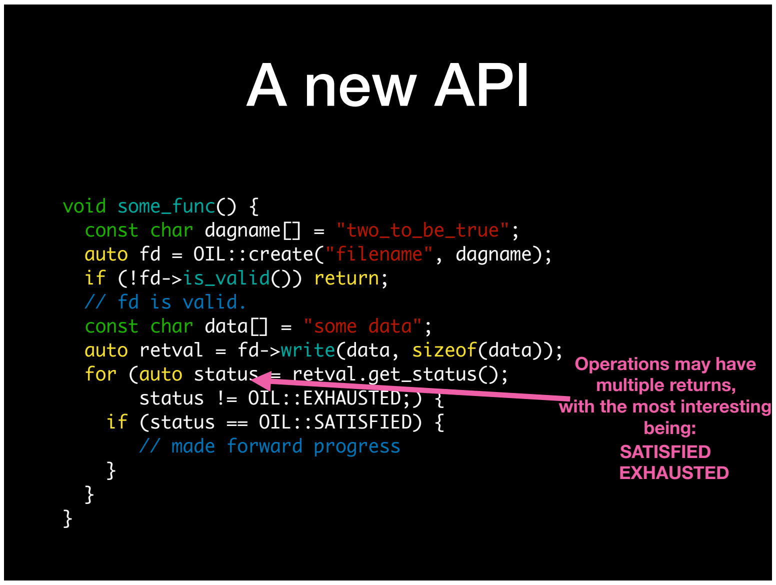


A new API

```
void some_func() {
  const char dagname[] = "two_to_be_true";
  auto fd = OIL::create("filename", dagname);
  if (!fd->is_valid()) return;
  // fd is valid.
  const char data[] = "some data";
  auto retval = fd->write(data, sizeof(data));
  for (auto status = retval.get_status();
       status != OIL::DONE;) {
    if (status == OIL::SATISFIED) {
      // made forward progress
    }
  }
}
```

A DAG defines a virtual filesystem on which all further operations execute.

So, what is a virtual filesystem? In other places it is defined as something which doesn't itself store data, but relies on other things to store the data.
I'd describe it as something which is a filesystem from the application-standpoint, but delegates to other storage systems.
In particular, with such a system the number of virtual-filesystems could be proportional to the number of files.

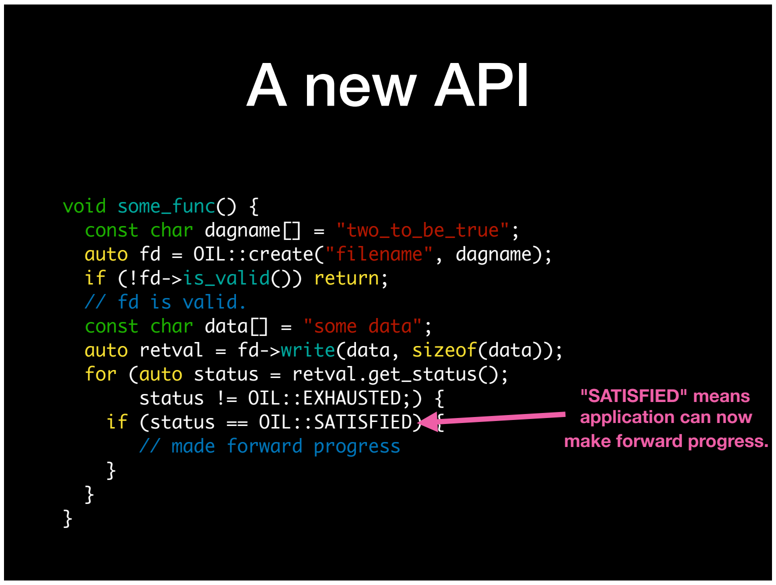


A new API

```
void some_func() {
  const char dagname[] = "two_to_be_true";
  auto fd = OIL::create("filename", dagname);
  if (!fd->is_valid()) return;
  // fd is valid.
  const char data[] = "some data";
  auto retval = fd->write(data, sizeof(data));
  for (auto status = retval.get_status();
       status != OIL::EXHAUSTED;) {
    if (status == OIL::SATISFIED) {
      // made forward progress
    }
  }
}
```

Operations may have multiple returns, with the most interesting being:
SATISFIED
EXHAUSTED

This is a key observation: the POSIX API didn't really (signals don't count) provide for multiple returns, but having multiple returns is key to solving a number of real-world problems.

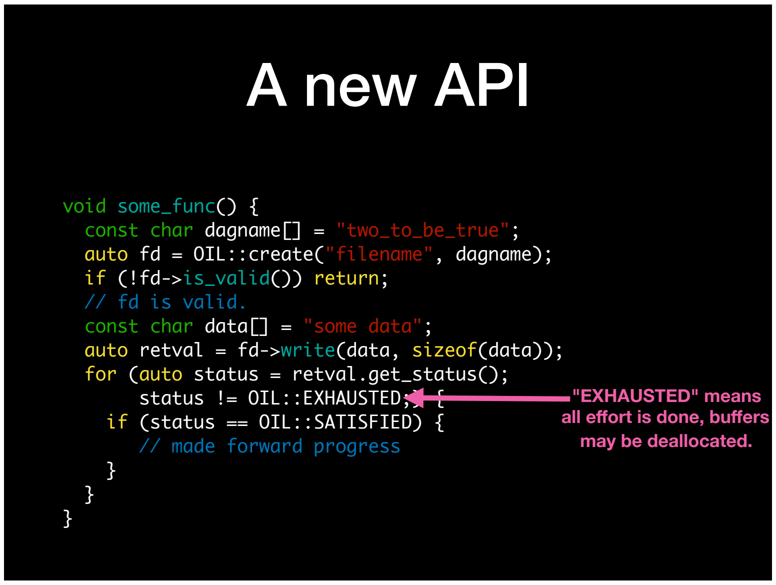


A new API

```
void some_func() {
  const char dagname[] = "two_to_be_true";
  auto fd = OIL::create("filename", dagname);
  if (!fd->is_valid()) return;
  // fd is valid.
  const char data[] = "some data";
  auto retval = fd->write(data, sizeof(data));
  for (auto status = retval.get_status();
       status != OIL::EXHAUSTED;) {
    if (status == OIL::SATISFIED) {
      // made forward progress
    }
  }
}
```

"SATISFIED" means application can now make forward progress.

In the case of a read, some of these returns may be a range of data and the version of that returned data, but I'm not going to go into that here as this is mostly just a teaser!
<describe SATISFIED as per pink>

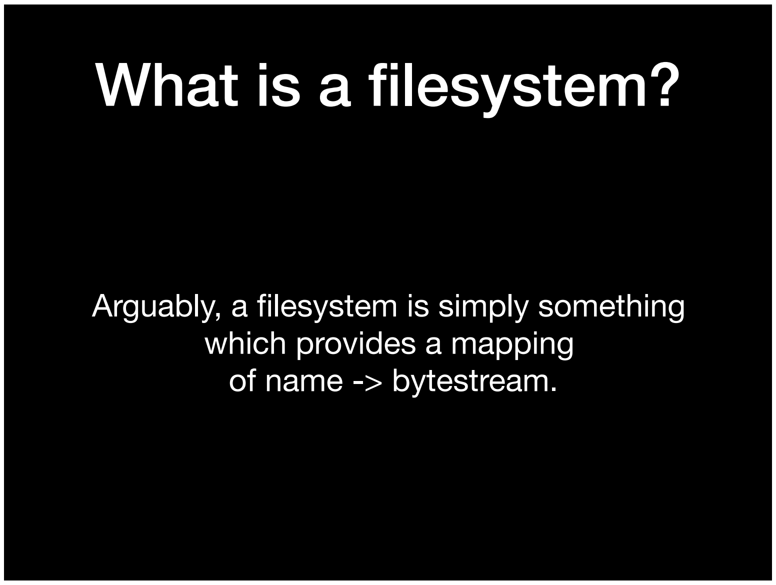


A new API

```
void some_func() {
  const char dagname[] = "two_to_be_true";
  auto fd = OIL::create("filename", dagname);
  if (!fd->is_valid()) return;
  // fd is valid.
  const char data[] = "some data";
  auto retval = fd->write(data, sizeof(data));
  for (auto status = retval.get_status();
       status != OIL::EXHAUSTED; ) {
    if (status == OIL::SATISFIED) {
      // made forward progress
    }
  }
}
```

"EXHAUSTED" means all effort is done, buffers may be deallocated.

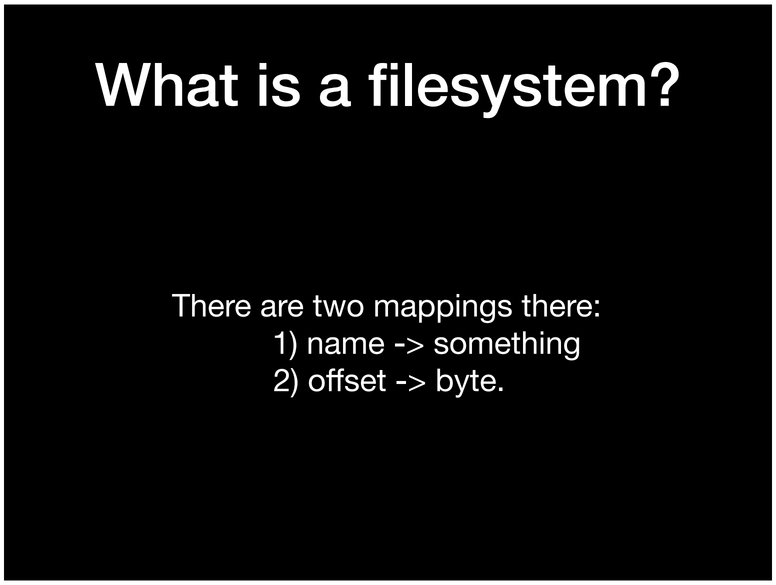
This is a key observation: the POSIX API didn't really (signals don't count) provide for multiple returns, but having multiple returns is key to solving a number of real-world problems.
<describe EXHAUSTED as per pink, how that helps with OOM, etc>



What is a filesystem?

Arguably, a filesystem is simply something which provides a mapping of name -> bytestream.

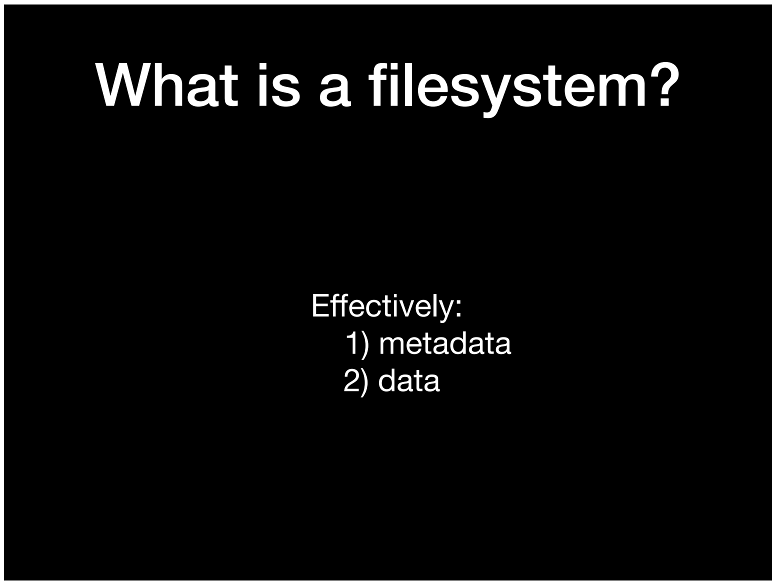
was a question we had to try to answer.



What is a filesystem?

There are two mappings there:
1) name -> something
2) offset -> byte.

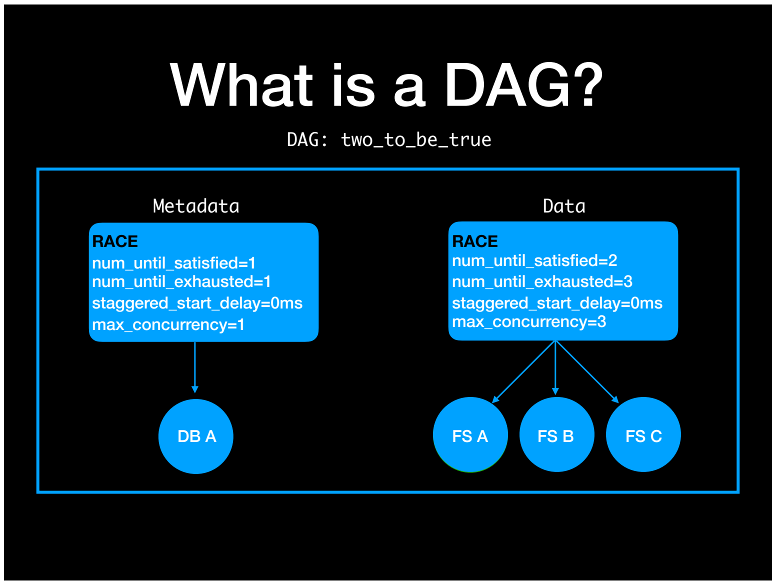
where "something" is some metadata, and then...<offset>



What is a filesystem?

Effectively:
1) metadata
2) data

So, if we want to provide something that looks/acts like a filesystem, for at least some definition of a filesystem, we probably need to handle metadata and data.
If we are being explicit about nearly everything, then we need to handle metadata and data explicitly and separately.
So, what is this dag thing that was referenced above?



What is a DAG?

DAG: two_to_be_true

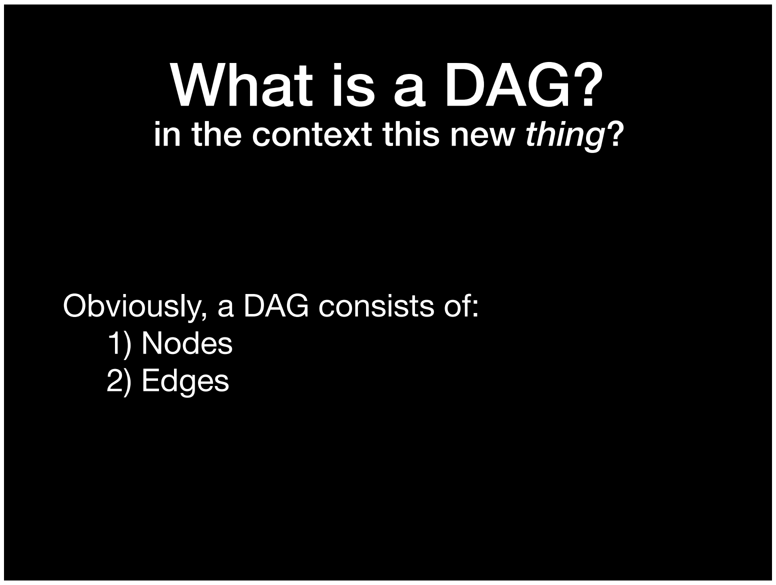
Metadata

RACE
num_until_satisfied=1
num_until_exhausted=1
staggered_start_delay=0ms
max_concurrency=1

Data

RACE
num_until_satisfied=2
num_until_exhausted=3
staggered_start_delay=0ms
max_concurrency=3

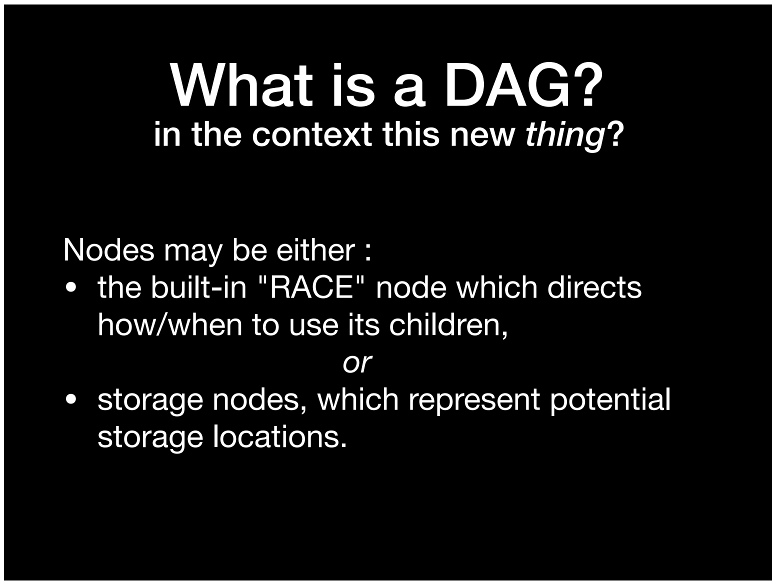
DAGs are I/O policies or plans. In this case there is a DAG for the metadata and one for the data.



What is a DAG?
in the context this new *thing*?

Obviously, a DAG consists of:
1) Nodes
2) Edges

Obviously I don't mean what is a "Directed Acyclic Graph", I mean a DAG in the context of this new API which defines a virtual filesystem.



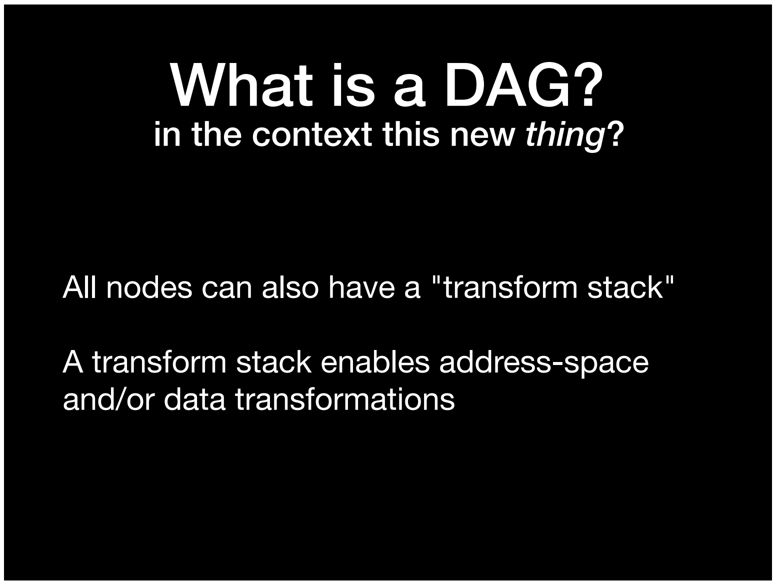
What is a DAG?
in the context this new *thing*?

Nodes may be either :
- the built-in "RACE" node which directs how/when to use its children,

or

- storage nodes, which represent potential storage locations.

we know a dag is nodes and edges (and further has no cycles).
.. what are the nodes?

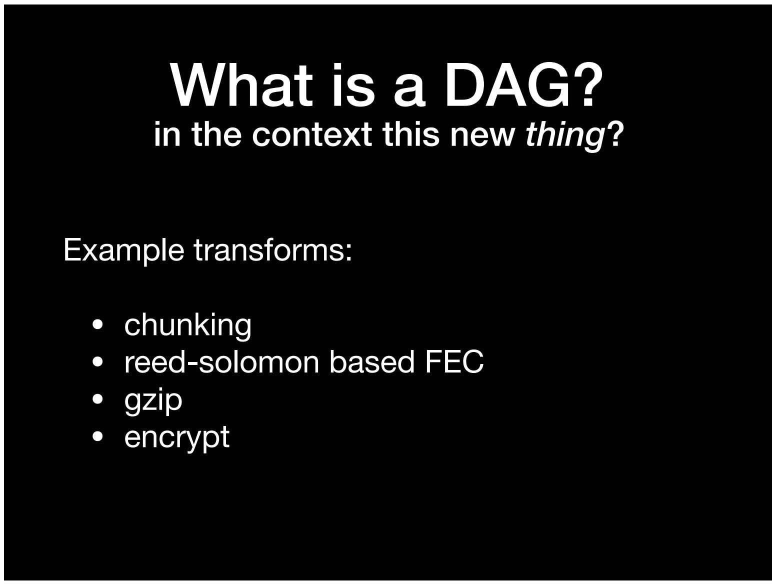


What is a DAG?
in the context this new *thing*?

All nodes can also have a "transform stack"

A transform stack enables address-space and/or data transformations

this is a potentially strong win for efficiency.
I'll also note that others have thought of some similar things in the past.
HTTP transfer-coding and content-coding, for instance..



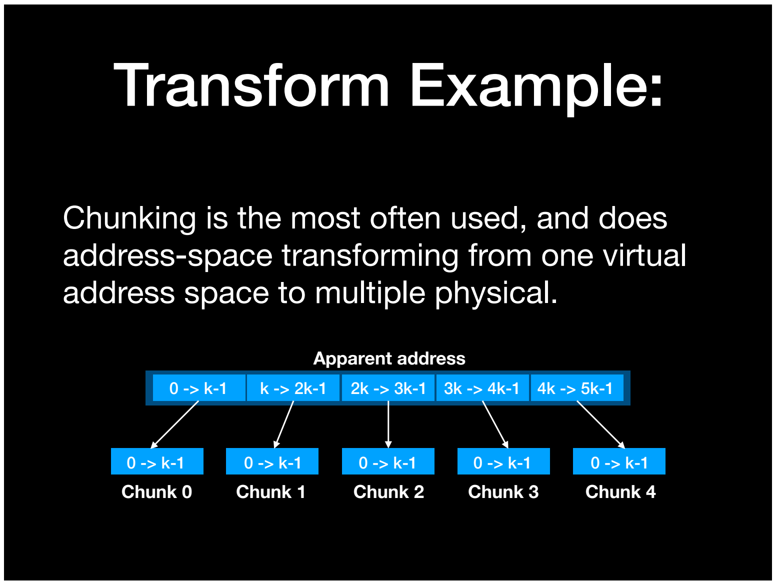
What is a DAG?
in the context this new *thing*?

Example transforms:

- chunking
- reed-solomon based FEC
- gzip
- encrypt

in the case of FEC, etc. failure-domains are represented as disparate children of the node with the transform.
You could apply this to a 'race' node.

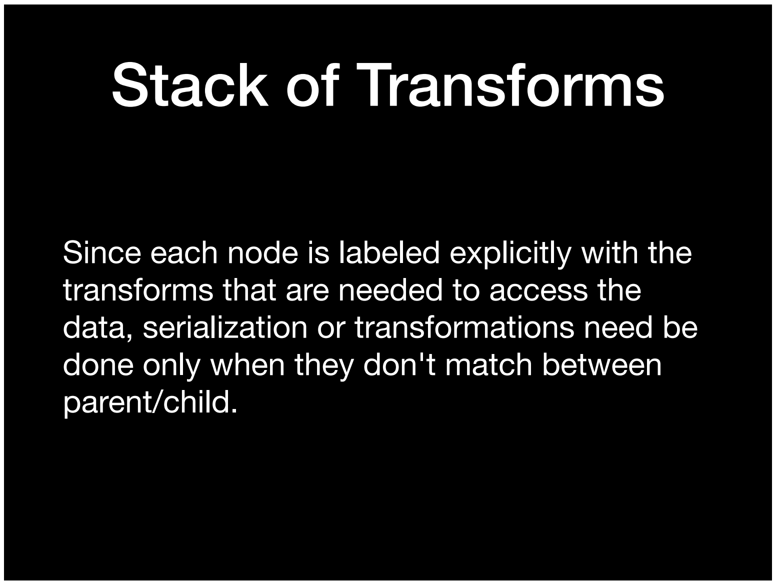
This is probably suboptimal, and is something I think can be improved.



Transform Example:

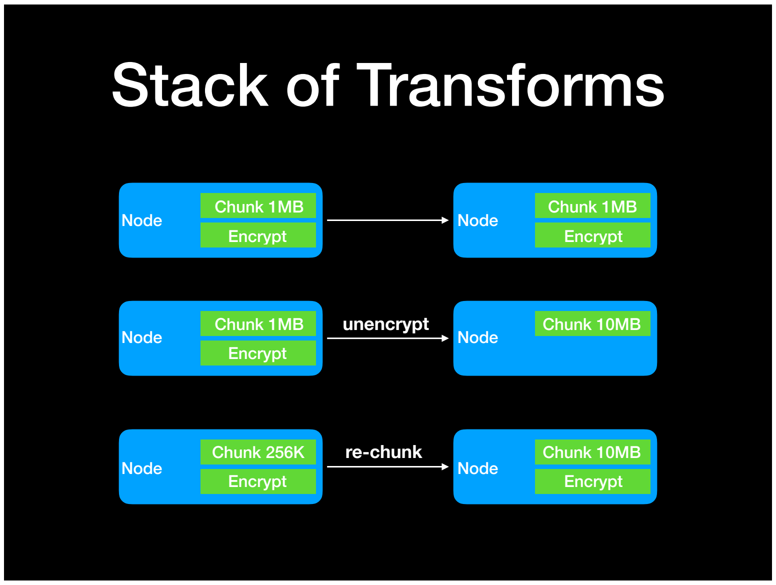
Chunking is the most often used, and does address-space transforming from one virtual address space to multiple physical.

This is very commonly used because the size of the chunk has very strong impacts on the amount of transactional or I/O overhead, it impacts the amount of data which shares fate on the same host, and affects the total amount of bandwidth that is on offer to satisfy the I/O.



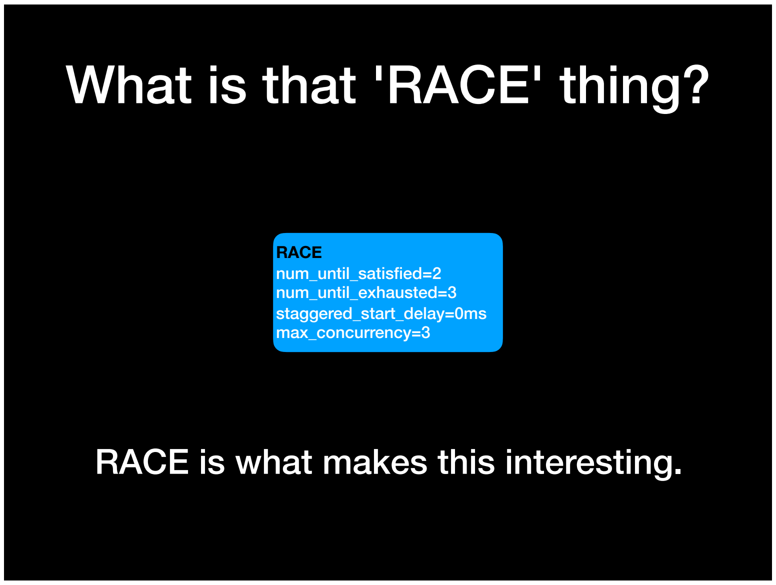
Stack of Transforms

Since each node is labeled explicitly with the transforms that are needed to access the data, serialization or transformations need be done only when they don't match between parent/child.



Stack of Transforms

Going left-to-right, with three different scenarios.

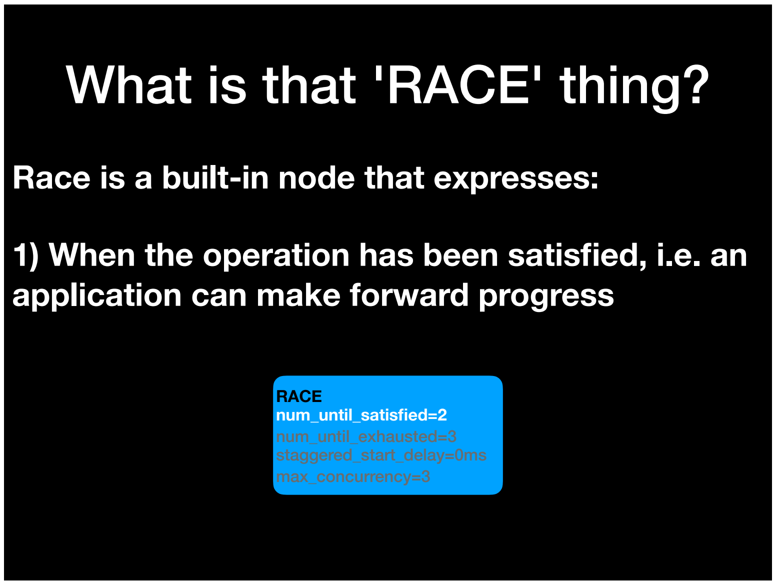


What is that 'RACE' thing?

RACE
num_until_satisfied=2
num_until_exhausted=3
staggered_start_delay=0ms
max_concurrency=3

RACE is what makes this interesting.

This is one of my favorite parts of this new OIL thing.



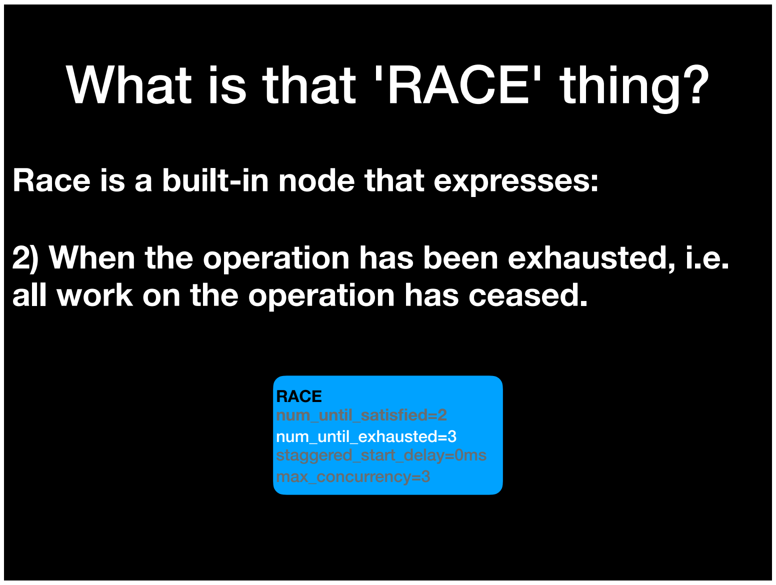
What is that 'RACE' thing?

Race is a built-in node that expresses:

1) When the operation has been satisfied, i.e. an application can make forward progress

RACE
num_until_satisfied=2
num_until_exhausted=3
staggered_start_delay=0ms
max_concurrency=3

reminder: 'Satisfied' generally means "application can make forward progress", as a convention. But it could mean whatever.
You can think of this node as expressing a map-reduce policy-- when/how to map, and how to reduce.



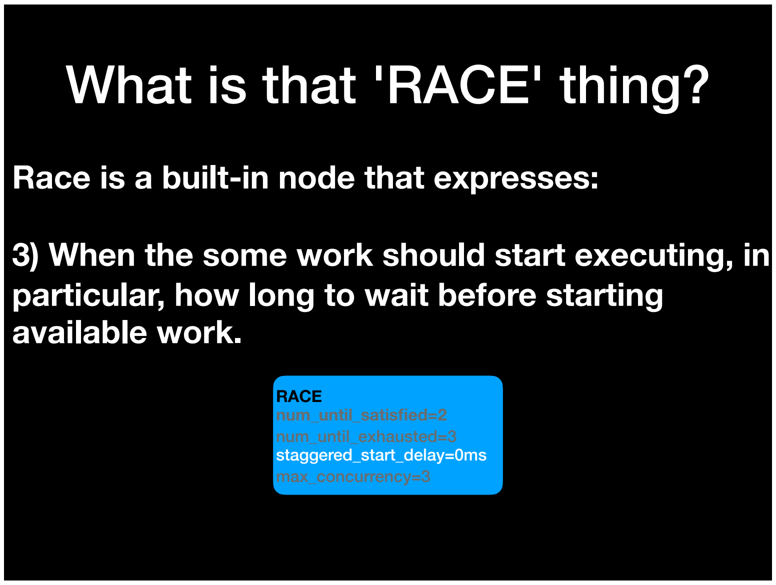
What is that 'RACE' thing?

Race is a built-in node that expresses:

2) When the operation has been exhausted, i.e. all work on the operation has ceased.

RACE
num_until_satisfied=2
num_until_exhausted=3
staggered_start_delay=0ms
max_concurrency=3

When the operation is exhausted, there is no more work ongoing. In cases where the application cares about overlap, this provides a mechanism for the application to do whatever it needs to do to resolve any remaining issues, including kicking off another I/O-write.



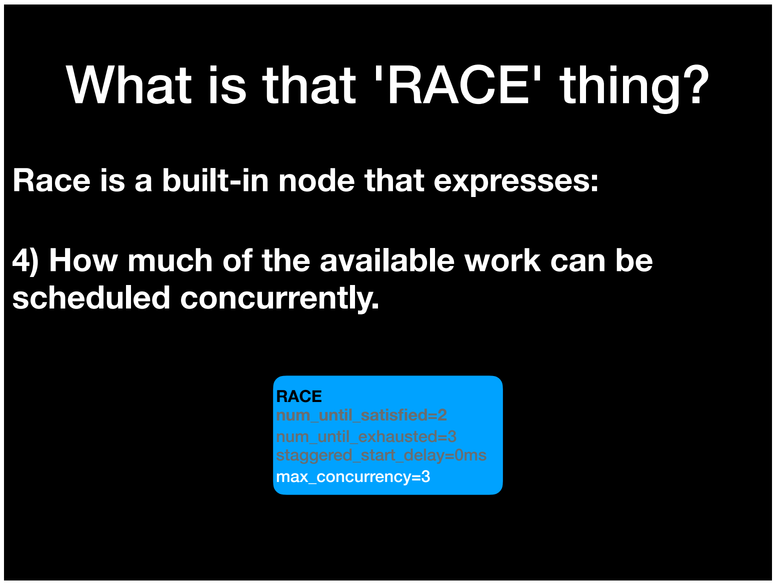
What is that 'RACE' thing?

Race is a built-in node that expresses:

3) When the some work should start executing, in particular, how long to wait before starting available work.

RACE
num_until_satisfied=2
num_until_exhausted=3
staggered_start_delay=0ms
max_concurrency=3

A common optimization in systems that are doing quorum-reads is to read only quorum, instead of from all potential replicas. You can imagine using this delay on those nodes that'd be the "spares". Or, in the case where you want any copy of the data, you can reduce the amount of effort in the common-case.



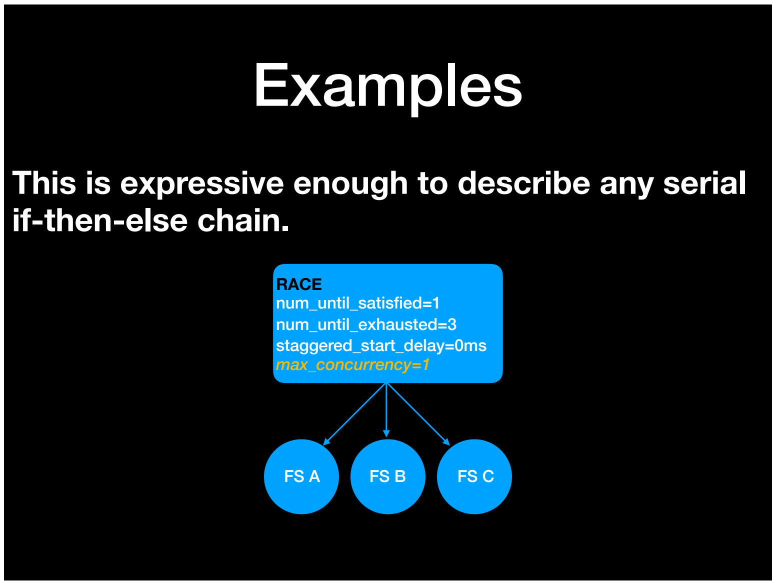
What is that 'RACE' thing?

Race is a built-in node that expresses:

4) How much of the available work can be scheduled concurrently.

RACE
num_until_satisfied=2
num_until_exhausted=3
staggered_start_delay=0ms
max_concurrency=3

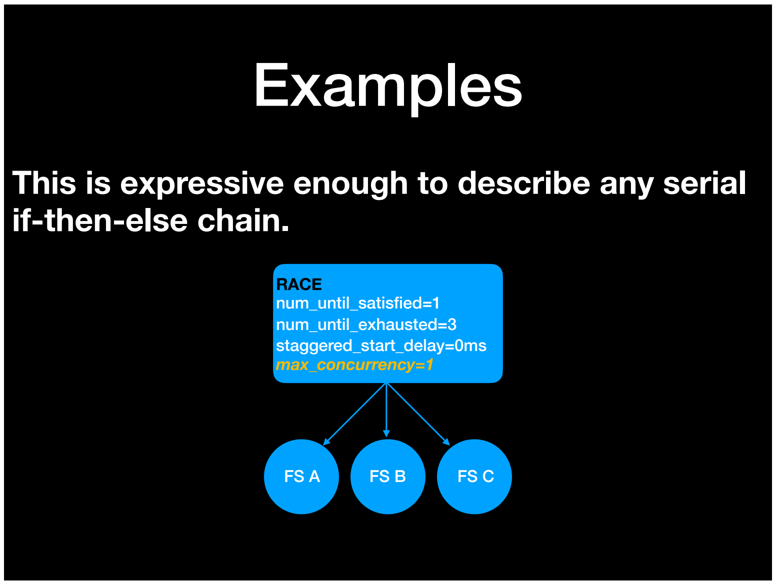
I hope this one is obvious-- the race node will only allow up-to 'max_concurrency' children to execute simultaneously.



Examples

This is expressive enough to describe any serial if-then-else chain.

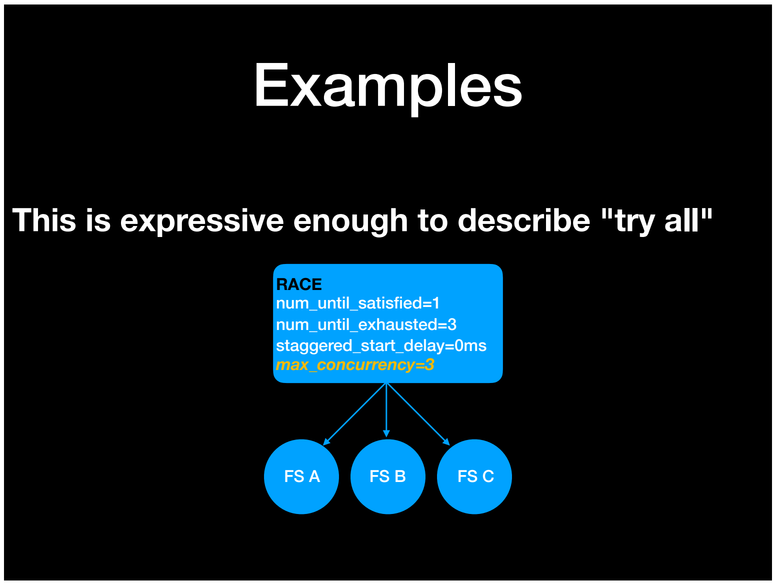
RACE
num_until_satisfied=1
num_until_exhausted=3
staggered_start_delay=0ms
max_concurrency=1



Examples

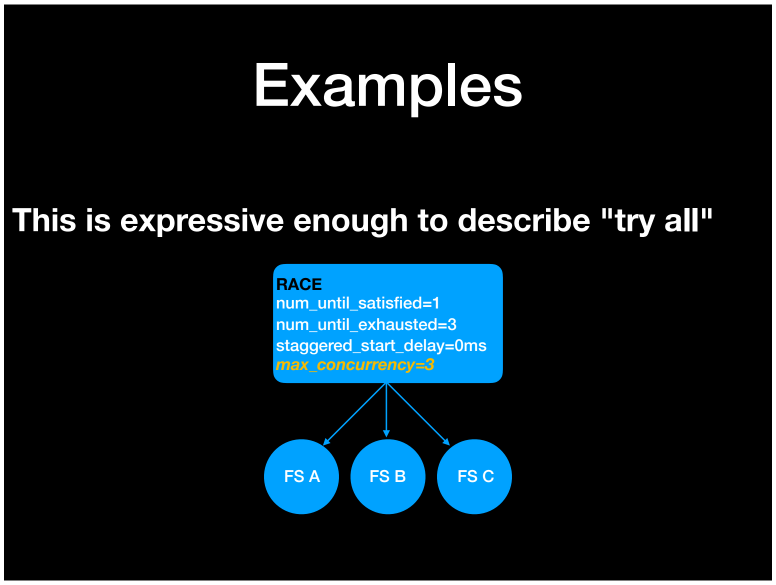
This is expressive enough to describe any serial if-then-else chain.

A max-concurrency of one implies serial behavior, where child nodes will be visited from left->right.



Examples

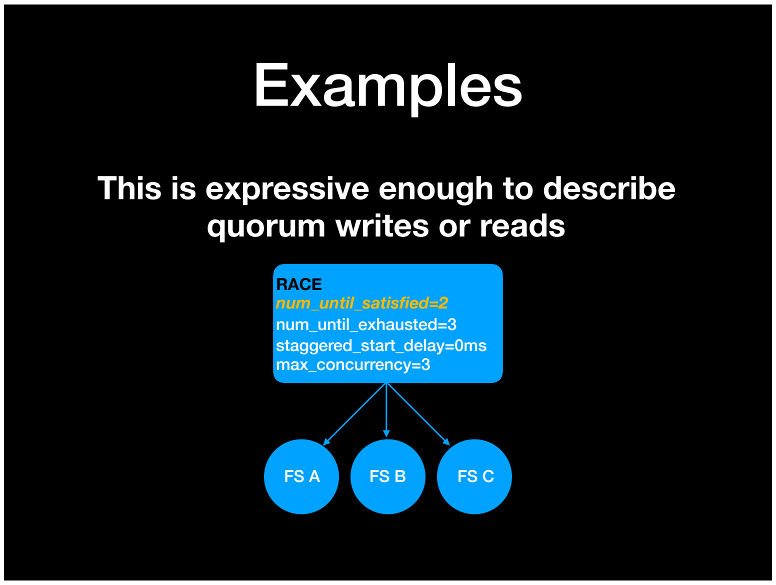
This is expressive enough to describe "try all"



Examples

This is expressive enough to describe "try all"

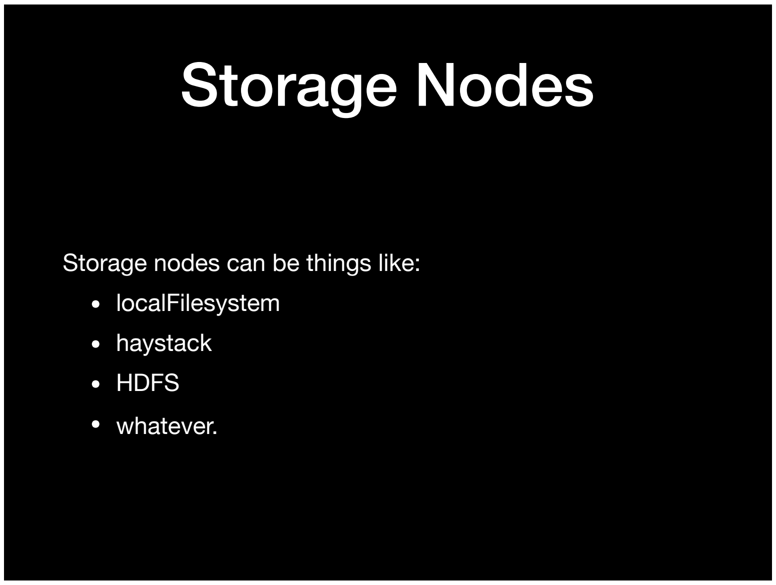
Note that max-concurrency is 3. As you'd expect means that we can try up-to-three things simultaneously. In this case that implies fully parallel behavior.



Examples

This is expressive enough to describe quorum writes or reads

As noted in the "quiz" before, there is not a single answer to the question of 'what to return' when there are multiple subordinate components with different answers... unless you can talk about that explicitly!
num_until_satisfied signals when forward progress can be signaled upwards to a parent (or the root, which is the application).
In this case 2 (out of the three) must be satisfied before the race node itself returns that it is satisfied.



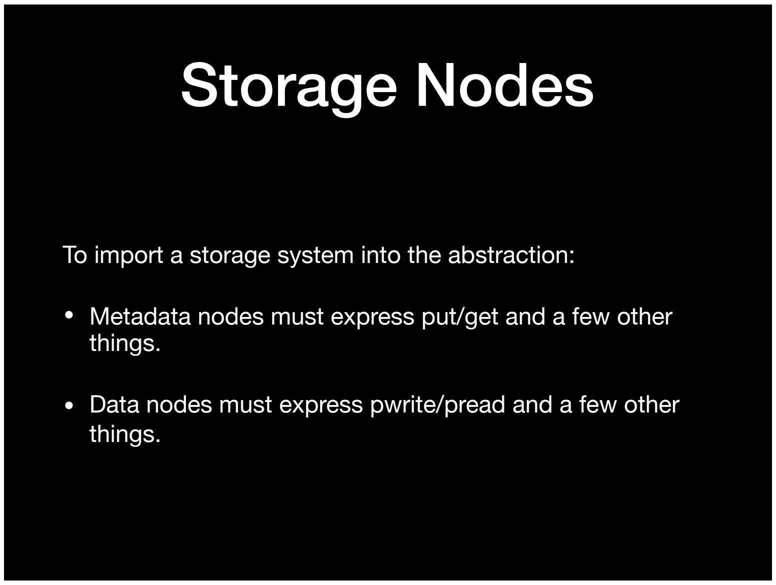
Storage Nodes

Storage nodes can be things like:

- localFilesystem
- haystack
- HDFS
- whatever.

What is a 'storage node'?

It is the place where the virtual becomes physical (or at least seems to).



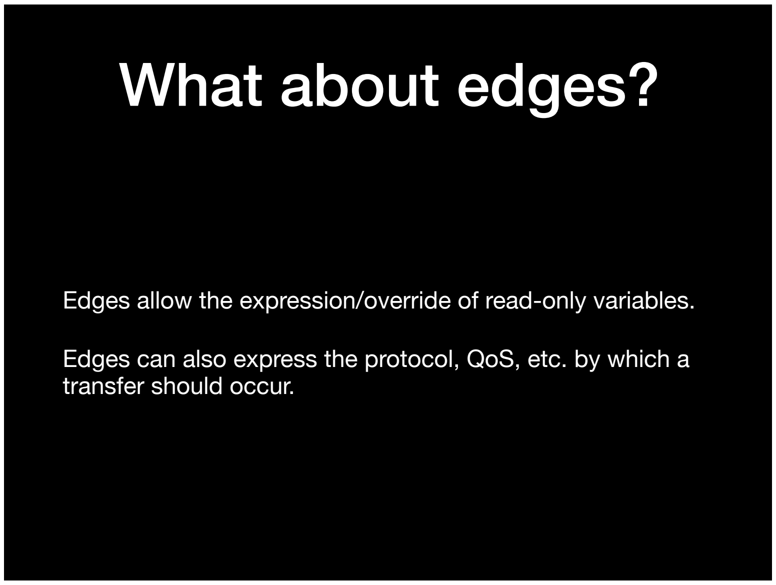
Storage Nodes

To import a storage system into the abstraction:

- Metadata nodes must express put/get and a few other things.
- Data nodes must express pwrite/pread and a few other things.

things like copy(), and similar things are stubbed out by doing a new open()->write(), but for filesystems that support such things directly, the writer/importer can provide filesystem-specific glue.

Typically adding a new filesystem is the work of 2-3 days for a single developer... not that we've done enough to say that with stat-sig certainty...



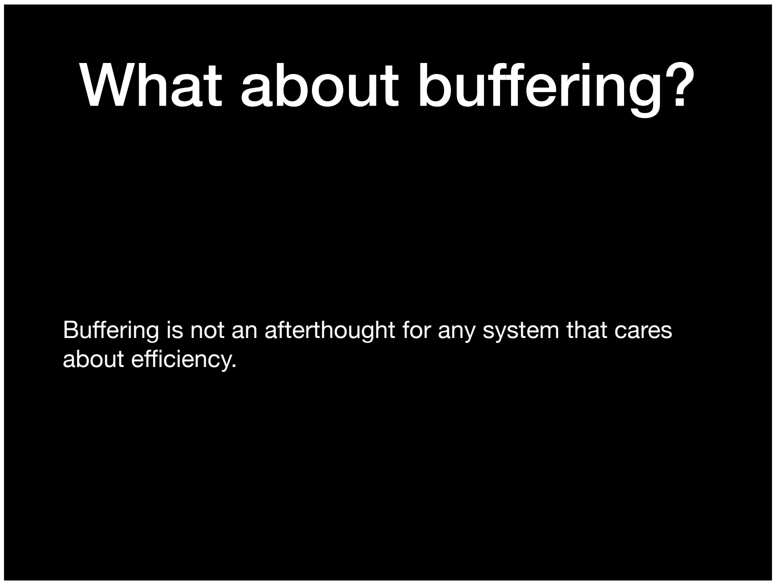
What about edges?

Edges allow the expression/override of read-only variables.

Edges can also express the protocol, QoS, etc. by which a transfer should occur.

We talked about nodes and edges... so what about the edges?
When a DAG is executed, a dictionary of read-only values is passed into each node. The node *cannot* modify this dictionary, but it can change its behavior based upon the values within.

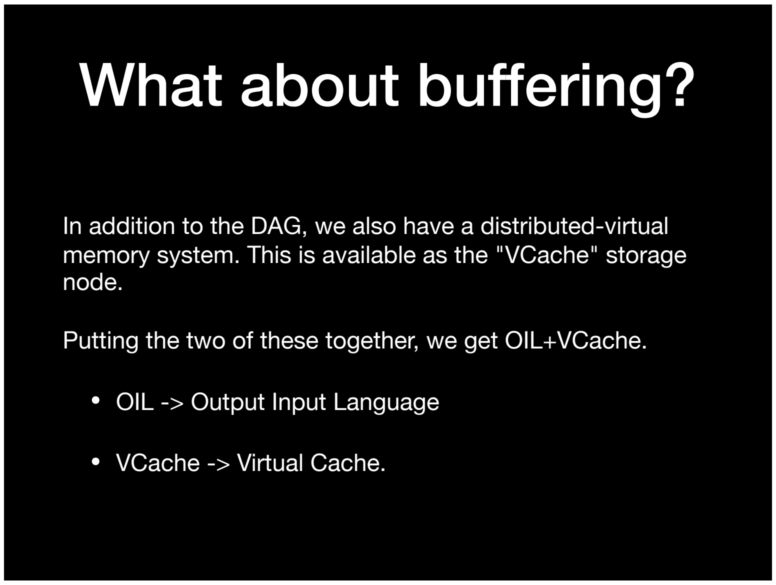
Edges can also express things that should happen during a transfer. QoS is something that might be commonly signaled here.



What about buffering?

Buffering is not an afterthought for any system that cares about efficiency.

If all there was was a single-layer, it'd be pretty boring, and I wouldn't be standing here.



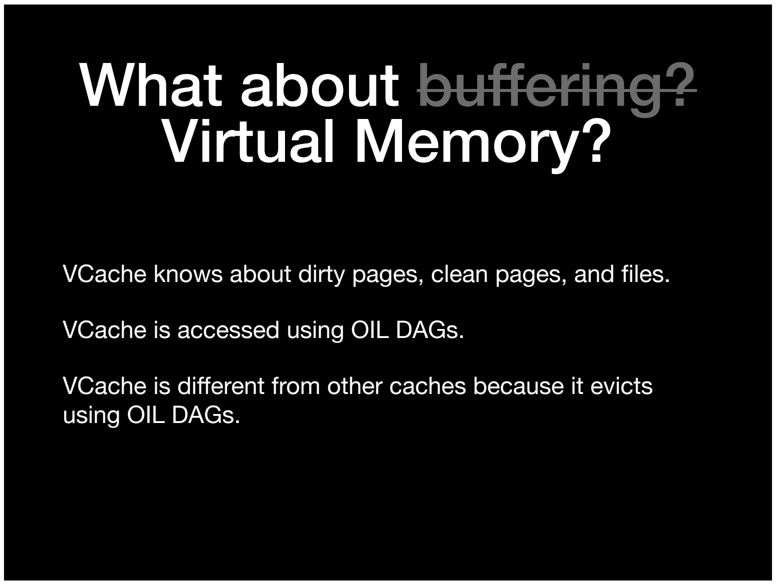
What about buffering?

In addition to the DAG, we also have a distributed-virtual memory system. This is available as the "VCache" storage node.

Putting the two of these together, we get OIL+VCache.

- OIL -> Output Input Language
- VCache -> Virtual Cache.

OIL is the part of this which is the pure API. It defines how the DAGs are interpreted, and what the code surface looks like.
VCache is the catch-all cache and buffer.



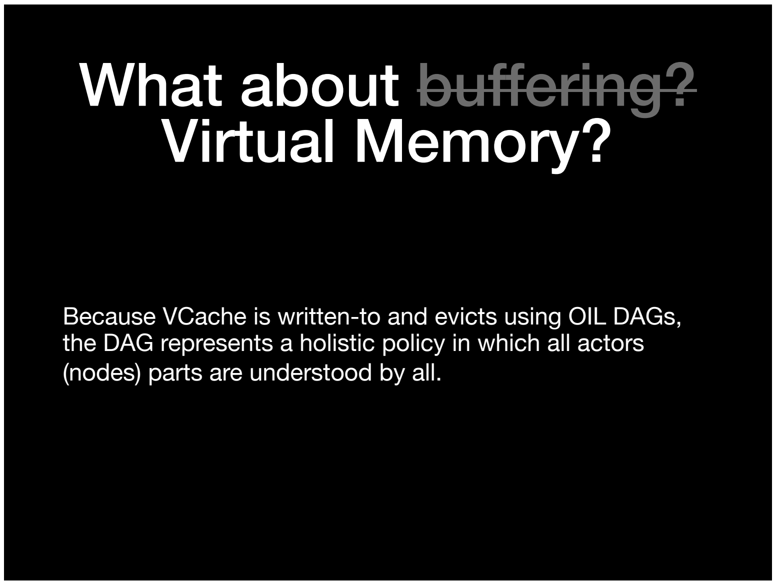
What about ~~buffering?~~ Virtual Memory?

VCache knows about dirty pages, clean pages, and files.

VCache is accessed using OIL DAGs.

VCache is different from other caches because it evicts using OIL DAGs.

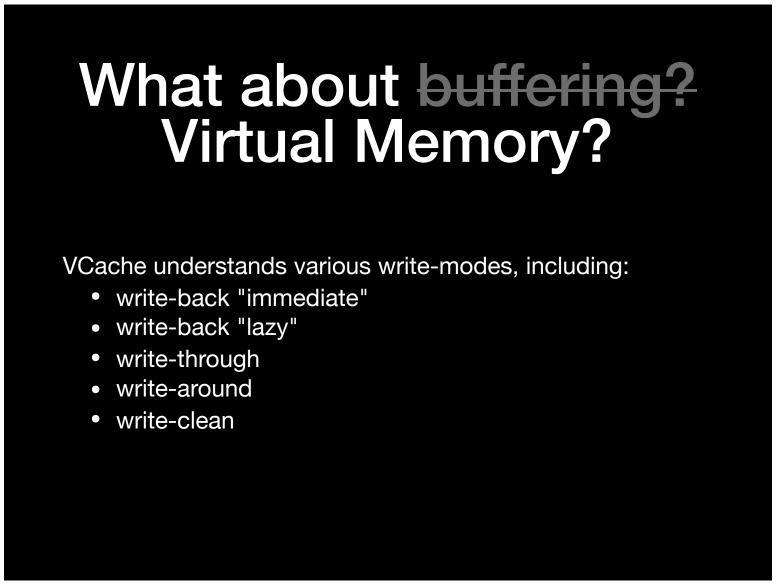
Unlike a number of caches, VCache is meant to look/act like a virtual memory subsystem, at least as viewed externally. This means that it handles write-back, not just look-aside.



What about ~~buffering?~~ Virtual Memory?

Because VCache is written-to and evicts using OIL DAGs, the DAG represents a holistic policy in which all actors (nodes) parts are understood by all.

Why would you care?
I'll say it again, the real-world performance is going to be dependent on how you buffer/cache.

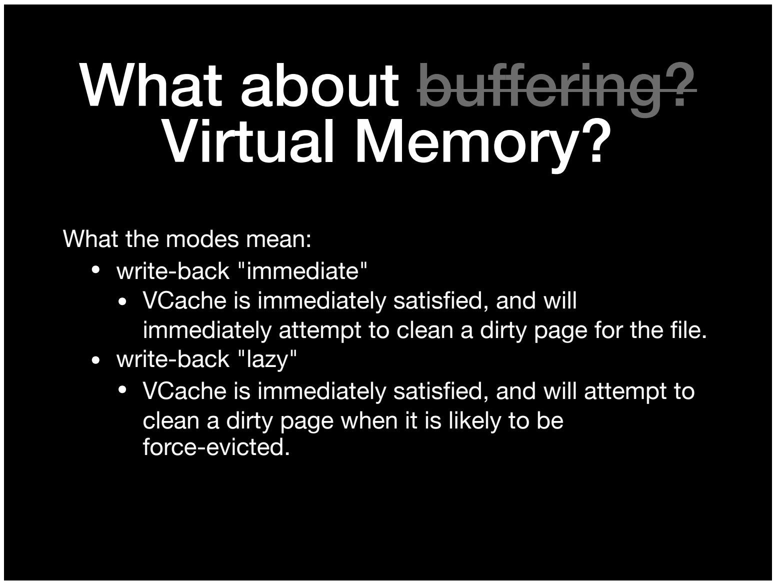


What about ~~buffering?~~ Virtual Memory?

VCache understands various write-modes, including:

- write-back "immediate"
- write-back "lazy"
- write-through
- write-around
- write-clean

will describe these in a second.

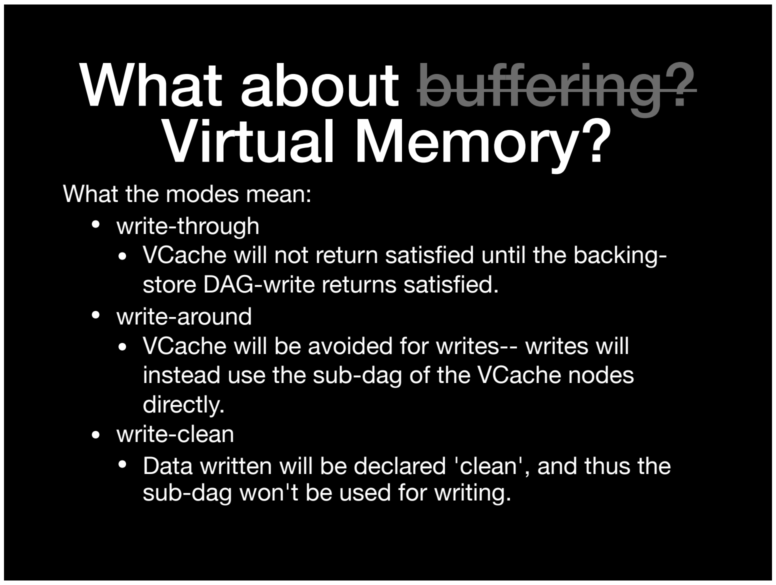


What about ~~buffering?~~ Virtual Memory?

What the modes mean:

- write-back "immediate"
  - VCache is immediately satisfied, and will immediately attempt to clean a dirty page for the file.
- write-back "lazy"
  - VCache is immediately satisfied, and will attempt to clean a dirty page when it is likely to be force-evicted.

immediate -- asynchronous, but not trying to save backing-store IOPs.
lazy -- async and attempting to save backing-store IOPs. great for tmp things.

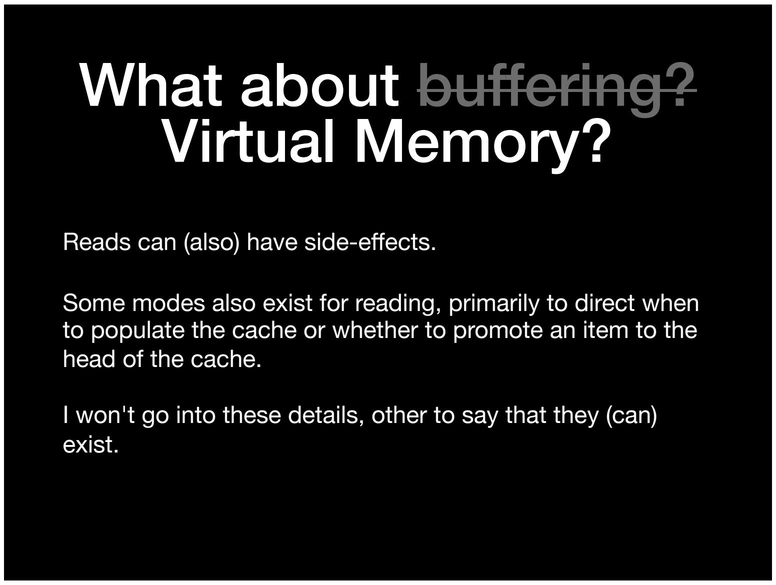


What about ~~buffering?~~ Virtual Memory?

What the modes mean:

- write-through
  - VCache will not return satisfied until the backing-store DAG-write returns satisfied.
- write-around
  - VCache will be avoided for writes-- writes will instead use the sub-dag of the VCache nodes directly.
- write-clean
  - Data written will be declared 'clean', and thus the sub-dag won't be used for writing.

write-through: satisfied happens when done happens.
write-around: This is useful when the read topology != the write topology, but the DAG authors are too lazy to write two entirely different DAGs.
write-clean: allows VCache to act as a look-aside cache.
Did I mention the 'being lazy' bit??



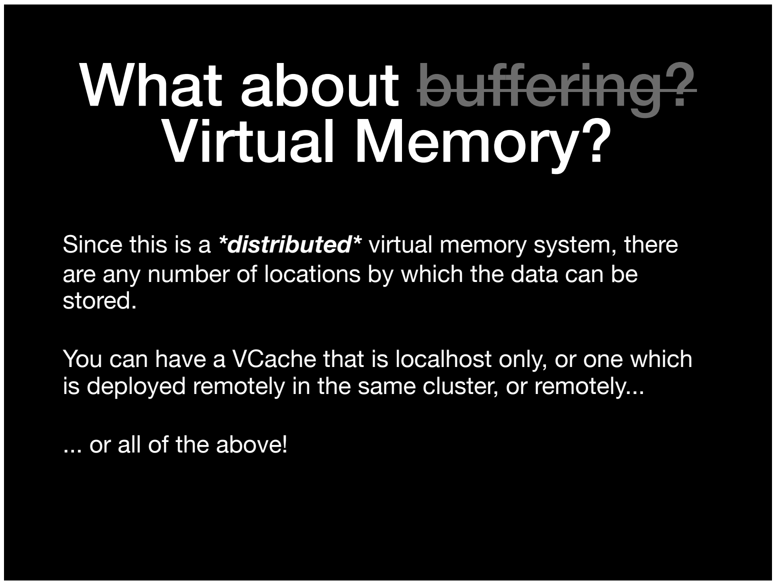
What about ~~buffering?~~ Virtual Memory?

Reads can (also) have side-effects.

Some modes also exist for reading, primarily to direct when to populate the cache or whether to promote an item to the head of the cache.

I won't go into these details, other to say that they (can) exist.

.. and not only do they exist, they can have significant impacts on real-world performance.



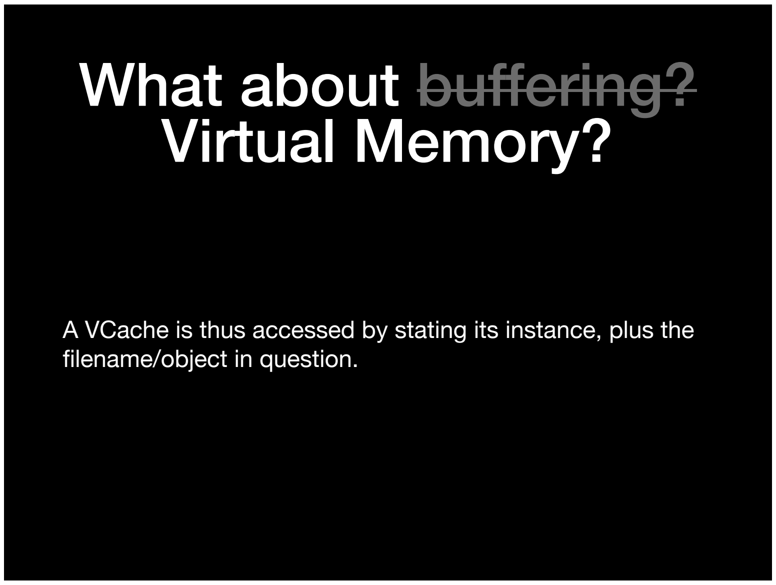
What about ~~buffering?~~ Virtual Memory?

Since this is a ***distributed*** virtual memory system, there are any number of locations by which the data can be stored.

You can have a VCache that is localhost only, or one which is deployed remotely in the same cluster, or remotely...

... or all of the above!

and each v-cache has its own idea about whether pages are dirty/clean, etc. So the localhost cache can believe the pages are clean (it did its job of writing to backing-store), while the remote caches may still have dirty pages.

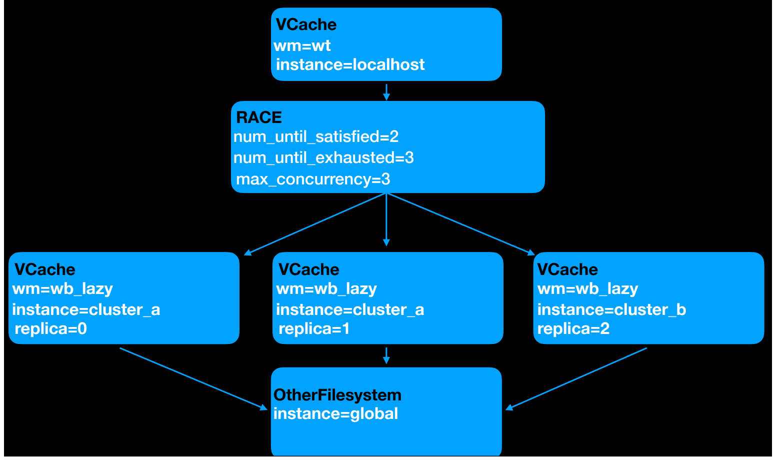


What about ~~buffering?~~ Virtual Memory?

A VCache is thus accessed by stating its instance, plus the filename/object in question.

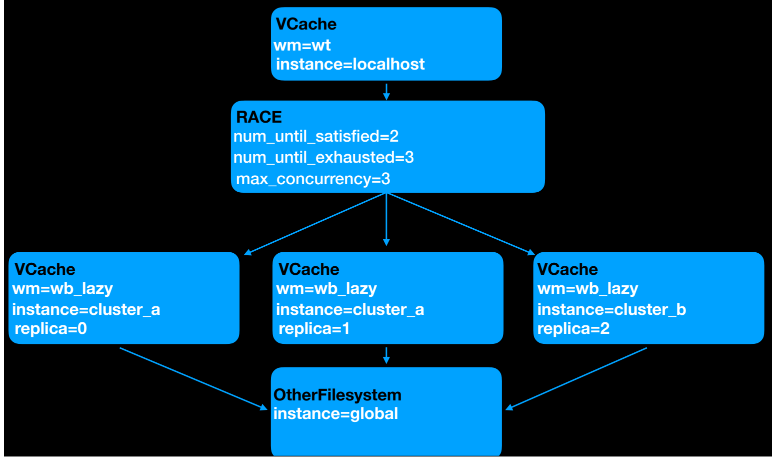
There is a difference between a 'localhost' VCache and a VCache using the machine's IP:port.
The former is private, while the latter is network accessible.

Real-World Example

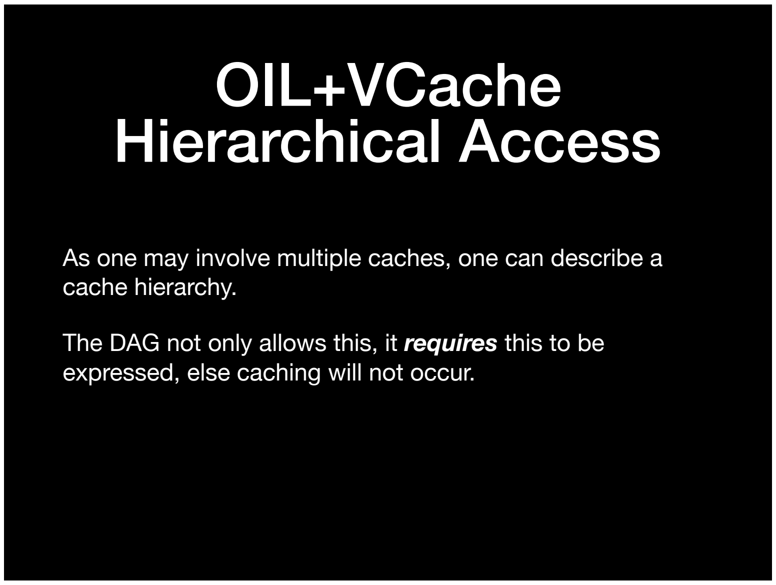


The localhost vcache takes ownership of the data and,then propagated ASAP to three different VCaches, two of which are in the same instance, and another is in another instance. When two remote VCache writes succeed, the write is satisfied.

Real-World Example



Finally, the remote VCaches evict to OtherFilesystem before the data would be lost.



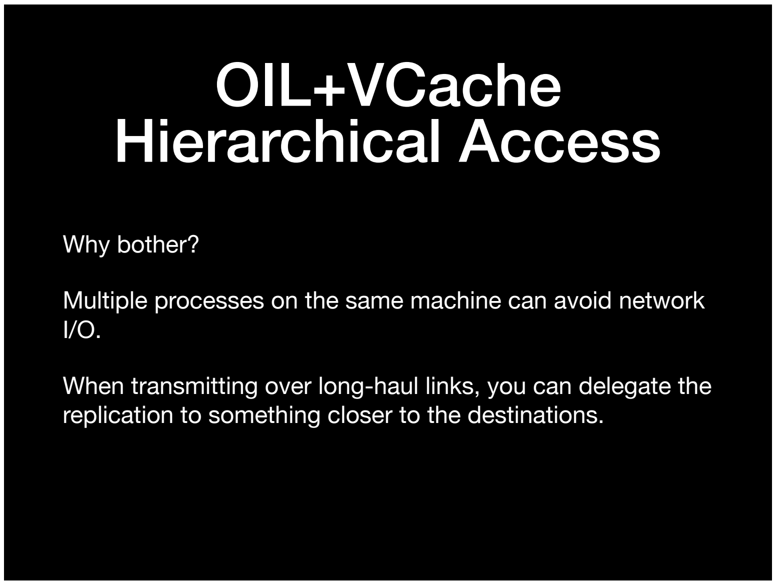
OIL+VCache Hierarchical Access

As one may involve multiple caches, one can describe a cache hierarchy.

The DAG not only allows this, it ***requires*** this to be expressed, else caching will not occur.

as was seen in the prior diagram which *had* a hierarchical caching description!

.. and yes, individual storage nodes may have their own caching.
... so it is probably more correct to say that caching won't occur in a semantical-interesting way/won't be explicitly addressable, manipulatable or shareable.



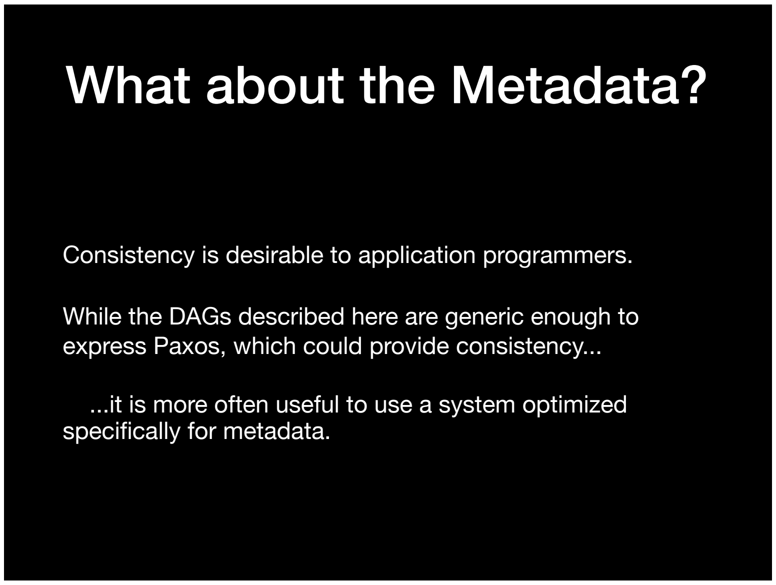
OIL+VCache Hierarchical Access

Why bother?

Multiple processes on the same machine can avoid network I/O.

When transmitting over long-haul links, you can delegate the replication to something closer to the destinations.

.. and with long-haul links being a fact of life in this new world of clouds, and with those long-haul links having significantly less bandwidth, this can provide for significant latency and overall cost benefits at scale.



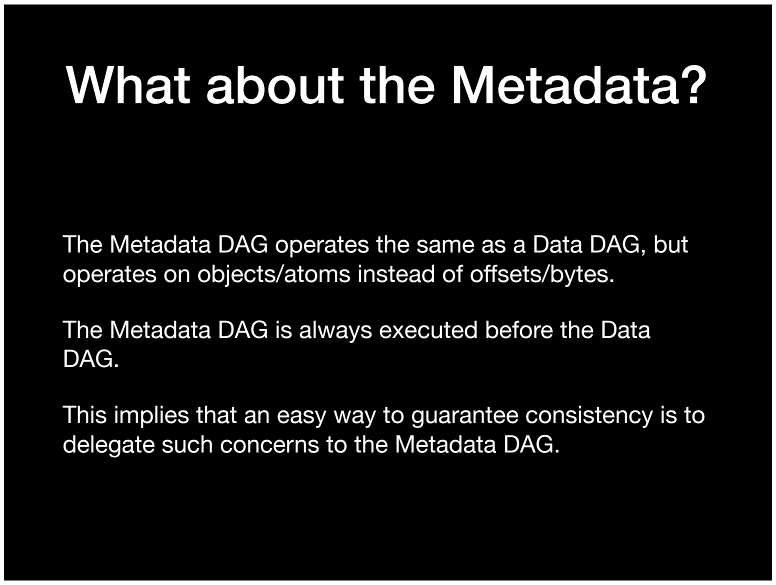
What about the Metadata?

Consistency is desirable to application programmers.

While the DAGs described here are generic enough to express Paxos, which could provide consistency...

...it is more often useful to use a system optimized specifically for metadata.

I described both a metadata dag and a data dag as part of the OIL policy dag.
I'll get back to the Paxos thing in a while...



What about the Metadata?

The Metadata DAG operates the same as a Data DAG, but operates on objects/atoms instead of offsets/bytes.

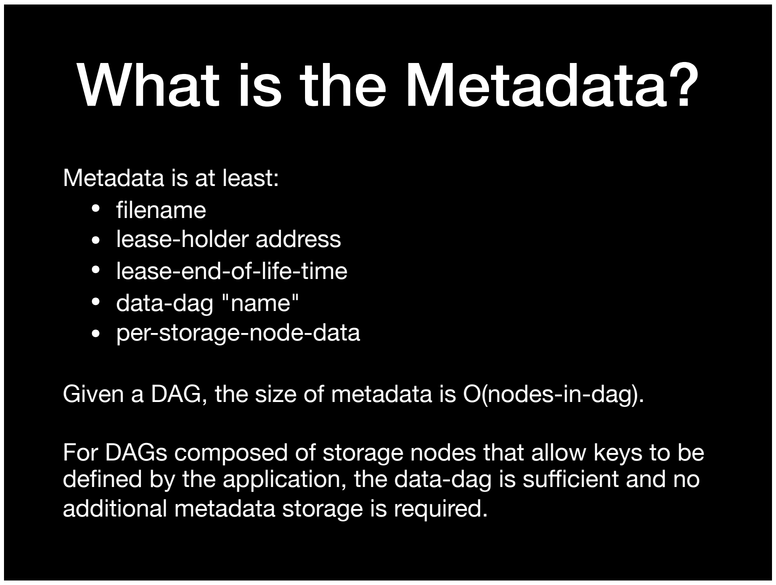
The Metadata DAG is always executed before the Data DAG.

This implies that an easy way to guarantee consistency is to delegate such concerns to the Metadata DAG.

The metadata DAG can also communicate an 'authority' to the data dag. This is a means by which cases where there's been a network partition leading to a lack-of-quorum can be understood. This changes the name of the data-dag files, and would require an application to understand how to merge things.

When things come back up, the data under the weaker authority name can be promoted to the strong-consistency authority by renaming the data.

Unlike the data dag, which can change over the lifetime of a file, the metadata dag cannot. Thus, it probably makes sense to have it be separate.



What is the Metadata?

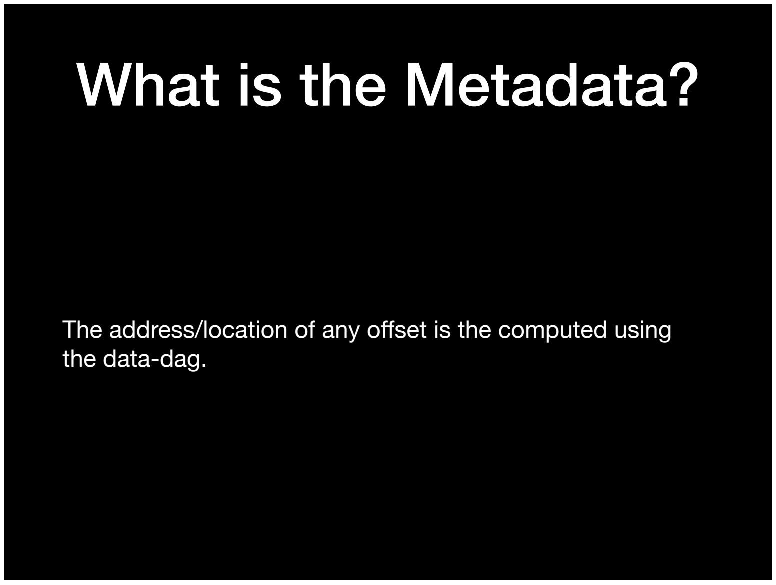
Metadata is at least:

- filename
- lease-holder address
- lease-end-of-life-time
- data-dag "name"
- per-storage-node-data

Given a DAG, the size of metadata is O(nodes-in-dag).

For DAGs composed of storage nodes that allow keys to be defined by the application, the data-dag is sufficient and no additional metadata storage is required.

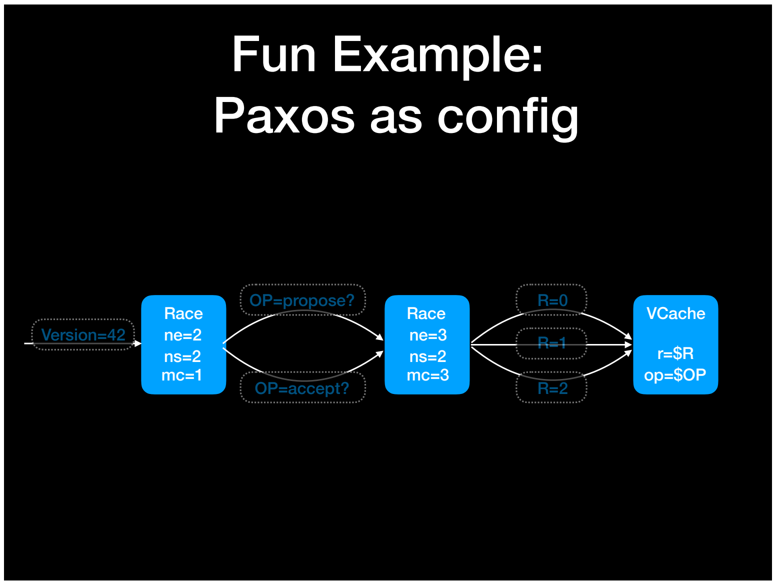
There are storage systems that give you a handle and don't let you provide a name. These will require some additional metadata storage (i.e. to store the handle).



What is the Metadata?

The address/location of any offset is the computed using the data-dag.

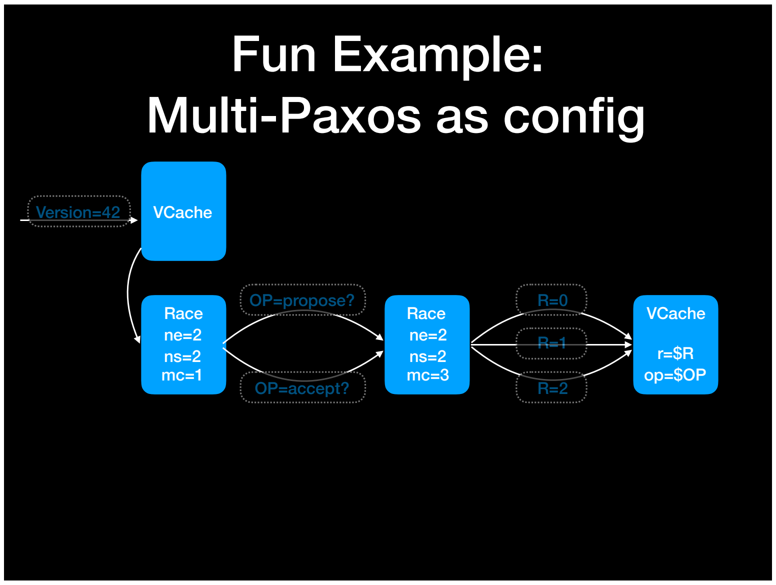
Fun Example: Paxos as config



This would be executed left-to-right, top-to-bottom.

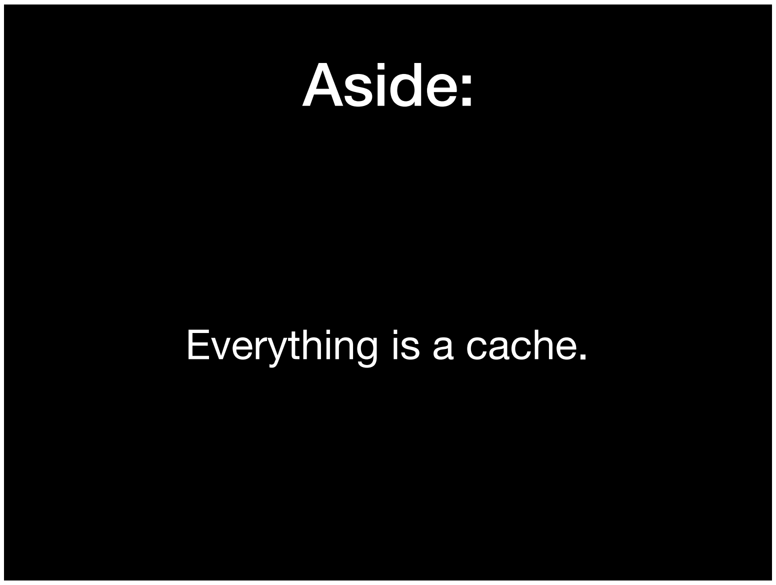
The application would supply that the version is 42.
The first race node will serially request "propose" of its child.
That child (the middle node) will then talk to three different VCaches, and declare 'satisfied' when a quorum of 2 (out of three) has been successful (i.e. were OK with proposal).
Then, the same thing will happen with

accept.



Fun Example: Multi-Paxos as config

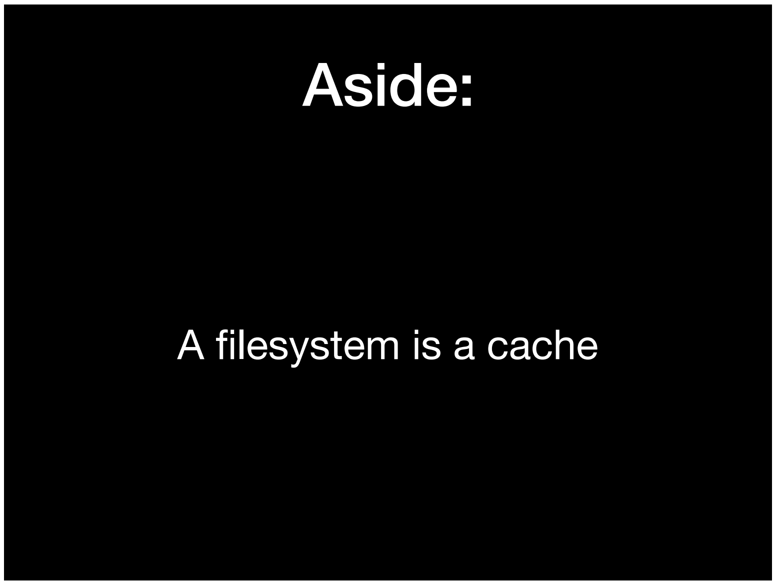
To have a leader, just add another VCache node.
The trick there will be electing it, but we can leave that to the metadata-dag, which could do a paxes-round to elect/discover the leader...



Aside:

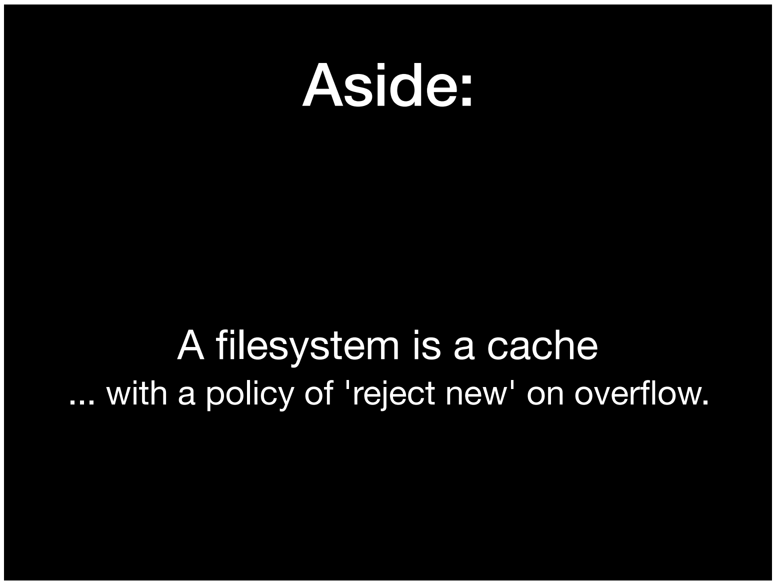
Everything is a cache.

potentially controversial/you may disagree... but bear with me!
Perhaps another way to say this is that everything should be thought of as a cache...



Aside:

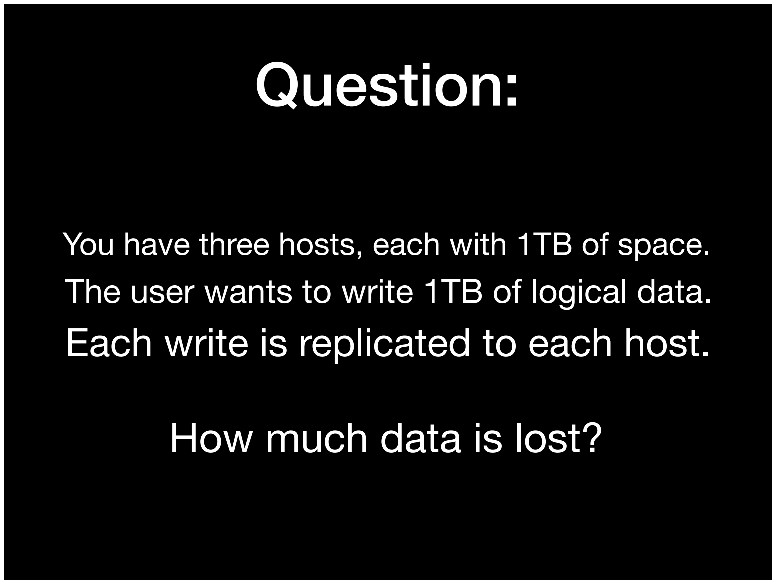
A filesystem is a cache



Aside:

A filesystem is a cache
... with a policy of 'reject new' on overflow.

For instance, what happens when your filesystem can store 1TB, and you offer it 3TB of data?
You'd reject the newest 2TB out of the 3TB offered.



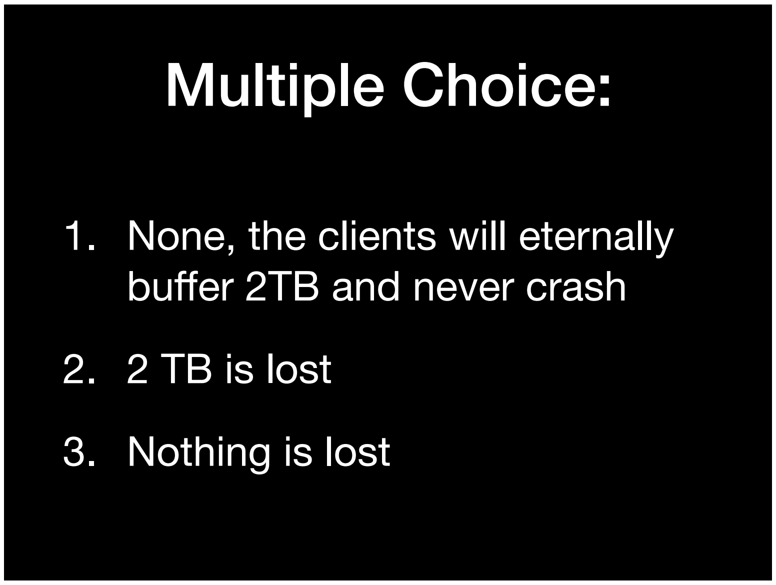
Question:

You have three hosts, each with 1TB of space.
The user wants to write 1TB of logical data.
Each write is replicated to each host.

How much data is lost?

another fun scenario...

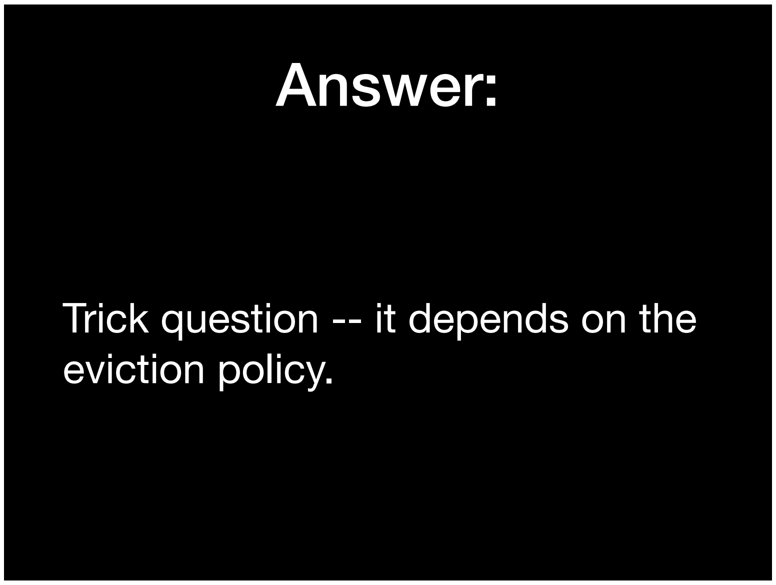
Another way to say this is that you have 1X physical storage, and 1X logical data, but the logical -> physical mapping is 1:3.



Multiple Choice:

1. None, the clients will eternally buffer 2TB and never crash
2. 2 TB is lost
3. Nothing is lost

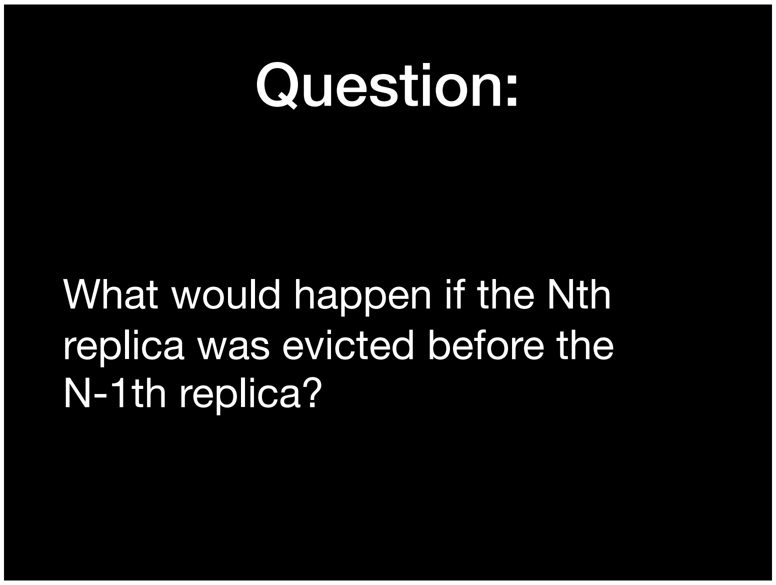
2/3 means 2-out-of-three here.
Why did we need a storage system at all if we could delegate all storage to the clients?



Answer:

Trick question -- it depends on the eviction policy.

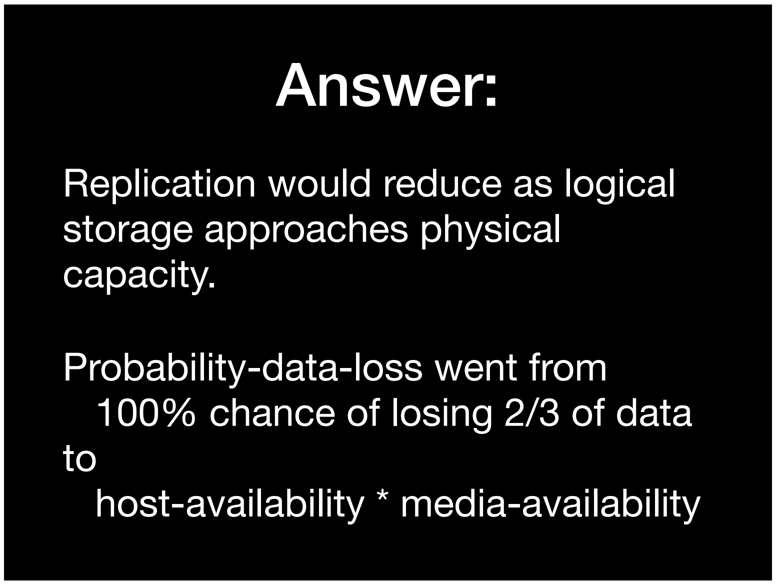
everything is a cache!
After all that just means having an explicit plan for when you've run out of capacity!



Question:

What would happen if the Nth replica was evicted before the N-1th replica?

For each host, if it had a 1th replica, all other things being equal, it'd be sure to evict any 2th replicas prior to evicting the 1th replicas.



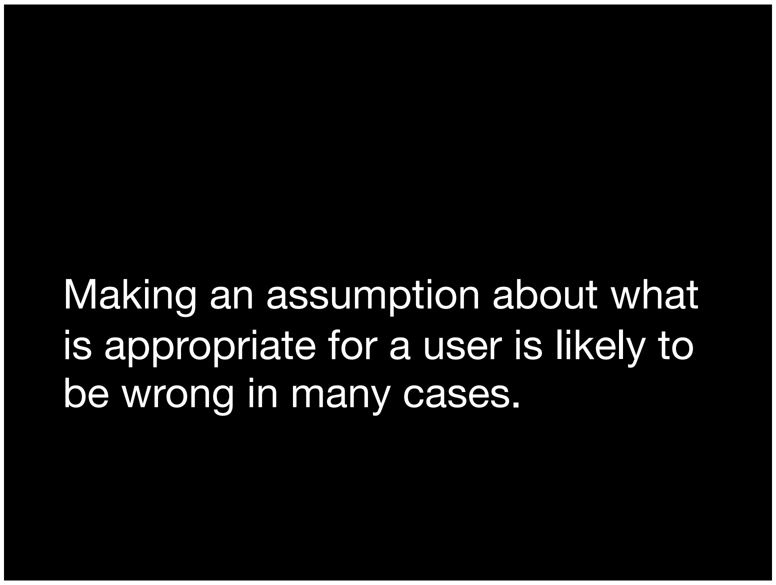
Answer:

Replication would reduce as logical storage approaches physical capacity.

Probability-data-loss went from
   100% chance of losing 2/3 of data
to
   host-availability * media-availability

Surprisingly, by treating everything as a cache, we *increase* reliability.

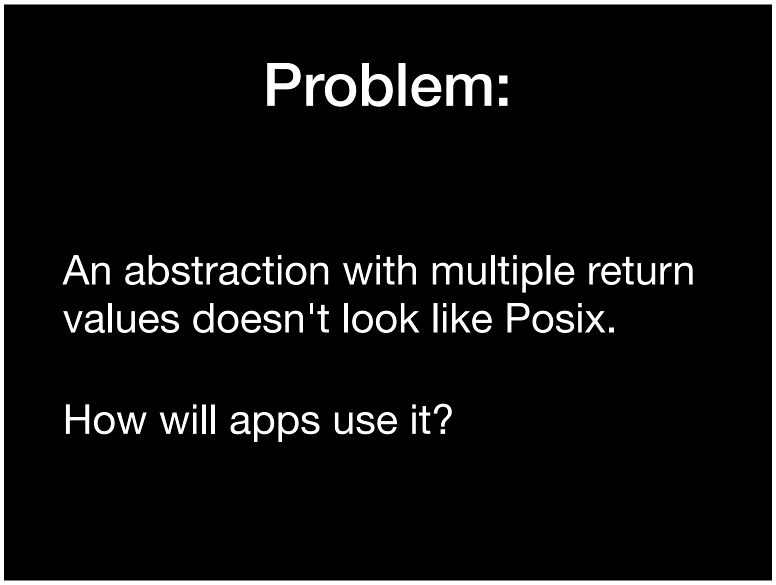
This would not be OK for some data-- it'd be better to lose the new data instead of the old (e.g. financial transactions), but in many cases such tradeoffs make sense.



Making an assumption about what is appropriate for a user is likely to be wrong in many cases.

Or at least it will make application-programmers implement work-arounds that'll be difficult or expensive and time consuming to find and back-out.

Moving the complexity of the system out to the edges almost always means more code and total system complexity...



Problem:

An abstraction with multiple return values doesn't look like Posix.

How will apps use it?

People like the Posix API, or are at least familiar with it.
.. and if not people, then there is a vast body of prior code and binaries that expect to be able to use it.

Answer:

New applications can use the new API.

Answer:

Make a FUSE mount, and have a local VCache manage the memory for async operations to ensure no OOMing.

Old applications can use the FUSE mount.

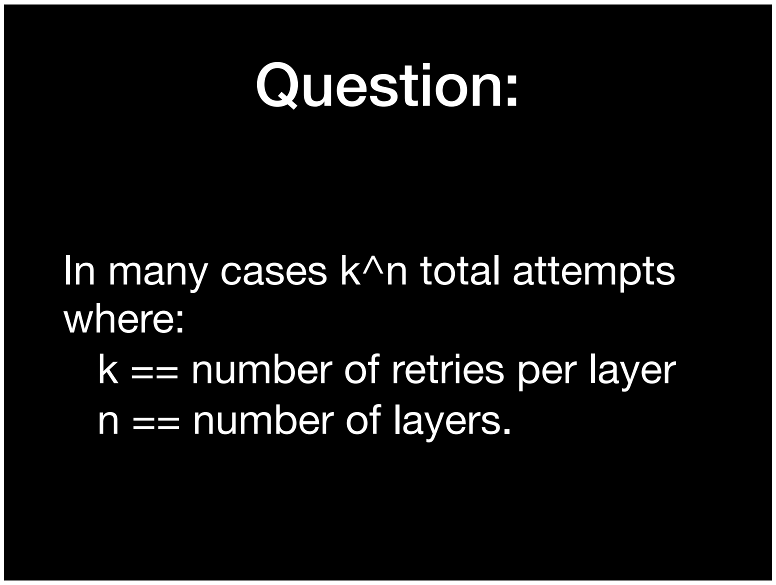
This isn't perfect. FUSE isn't as performant as one would like, but getting the application-layer expectations right often matter more to performance that the IOBench or other uBenchmarks would imply.
I'm sure there is plenty of future-work here in seeing if one couldn't efficiently express such things to the kernel.
I wonder what we'd call that language?

Question:

You have a hierarchy involving multiple layers of systems.

How many total I/O attempts will occur?

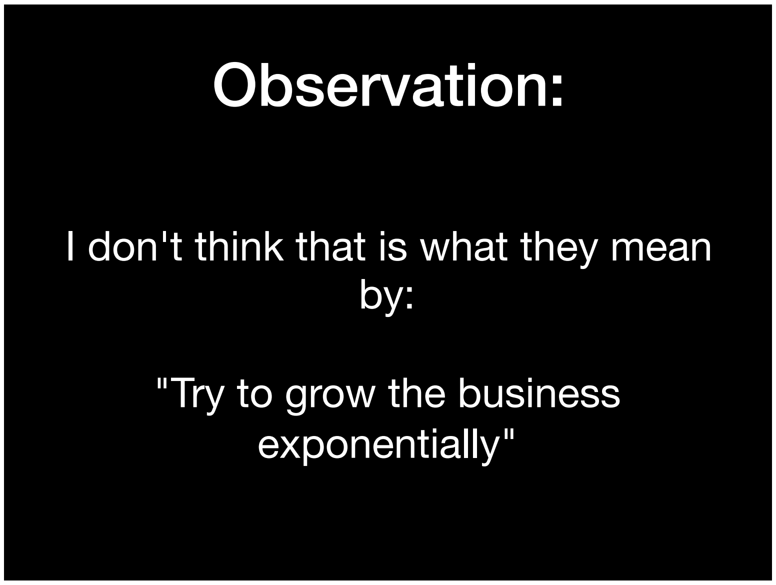
There is the answer for when things are succeeding vs failing.
Failing is the interesting case...



Question:

In many cases k^n total attempts where:
k == number of retries per layer
n == number of layers.

hmm... k^n. I think we call that exponential.

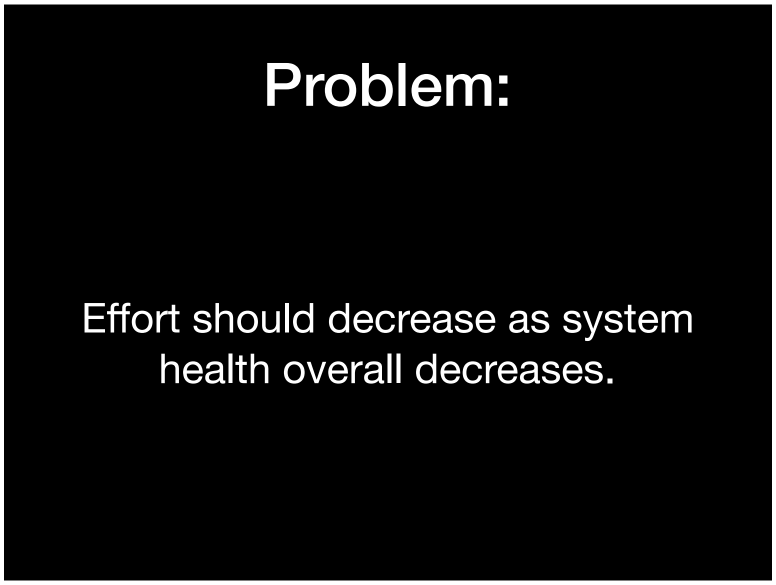


Observation:

I don't think that is what they mean by:

"Try to grow the business exponentially"

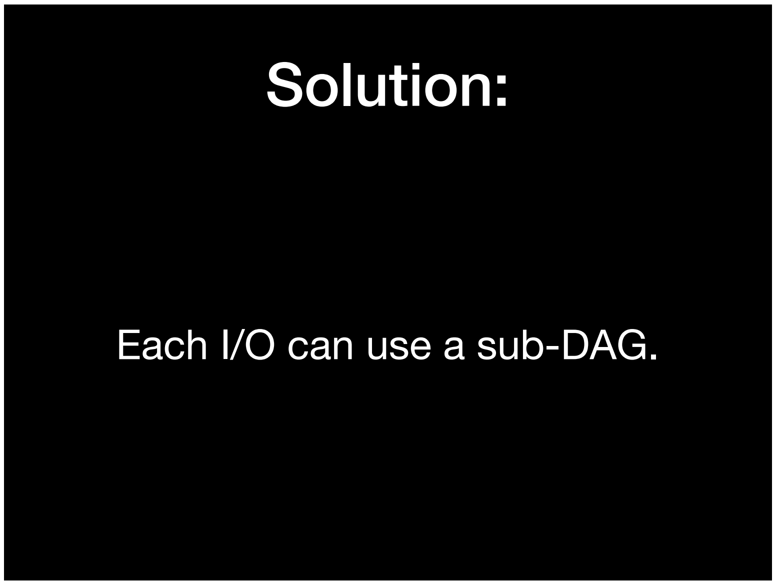
Ouch.



Problem:

Effort should decrease as system health overall decreases.

Not deceasing effort as things get bad, at least in many shared/distributed systems, can result in cascade failure.
I've been there, and that sucks.
This is why we have TCP (or other network) congestion control, for instance.



Solution:

Each I/O can use a sub-DAG.

If you have three replicas, and the first one is always dead, it wouldn't be an efficient use of time/effort to attempt to schedule the I/O to that server.

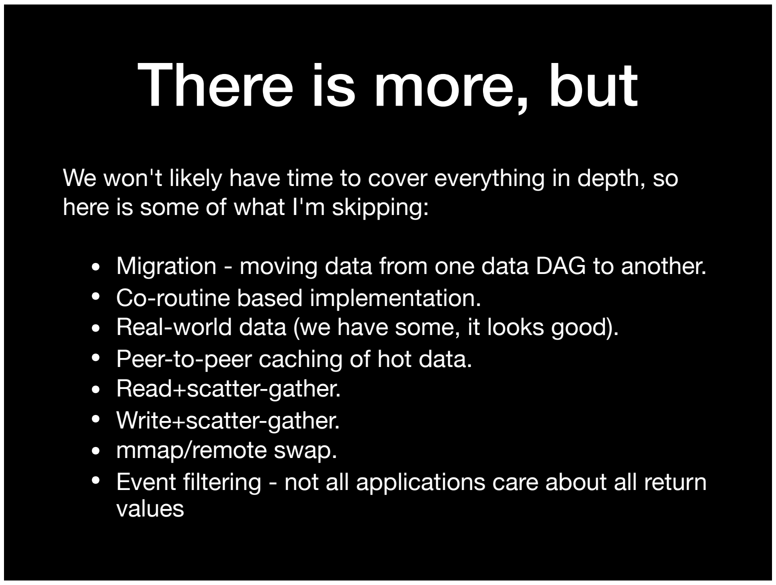
OIL allows per-IO customization

When you're doing a filesystem scrub, or a heal of a known-missing replica...

OIL allows per-IO customization

.. you probably want to be targeting specific parts of a DAG.

and again, though you can do per-IO customization, the per-IO subdag is required to be composed of storage nodes, unaltered, from the original DAG plus new/different RACE nodes.

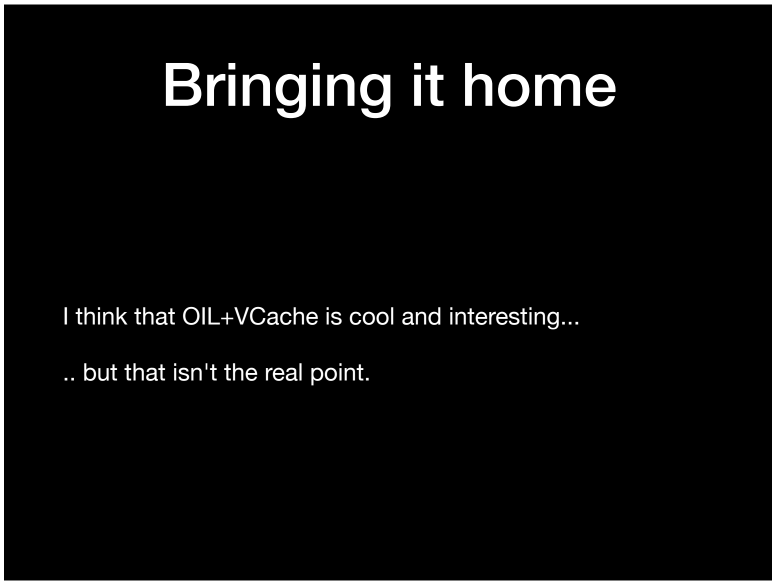


There is more, but

We won't likely have time to cover everything in depth, so here is some of what I'm skipping:

- Migration - moving data from one data DAG to another.
- Co-routine based implementation.
- Real-world data (we have some, it looks good).
- Peer-to-peer caching of hot data.
- Read+scatter-gather.
- Write+scatter-gather.
- mmap/remote swap.
- Event filtering - not all applications care about all return values

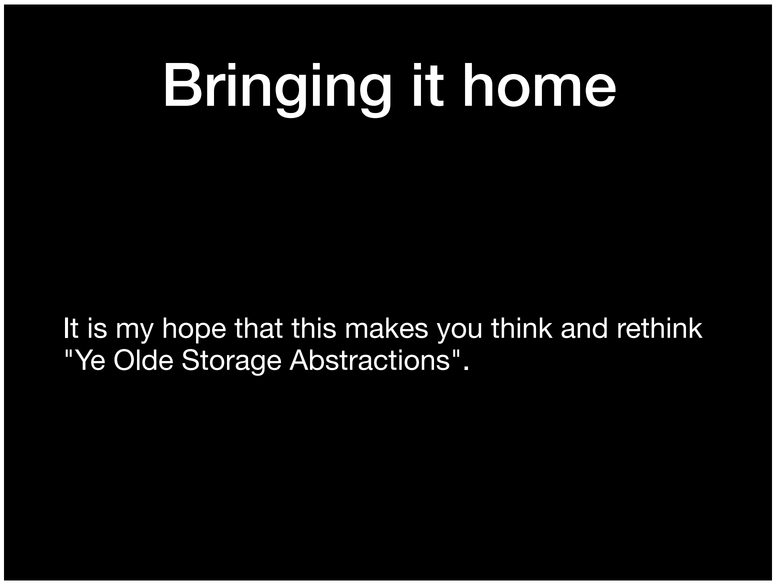
Read DAGs can be different from write dags, can be different from 'storage' dags.
Also not getting into that, but it is pretty useful in some cases.



Bringing it home

I think that OIL+VCache is cool and interesting...

.. but that isn't the real point.



Bringing it home

It is my hope that this makes you think and rethink "Ye Olde Storage Abstractions".

It is our hope that OIL+VCache inspires further innovation of abstractions and APIs across the industry and in academia.
Hey, we probably got it wrong. I look forward to hearing how it can be done better in the future.
We chose a non-turing complete description/language to describe these DAGs so that we could more easily reason about the surfaces. Maybe that was right, maybe that was wrong?

Thanks!

fenix@fb.com