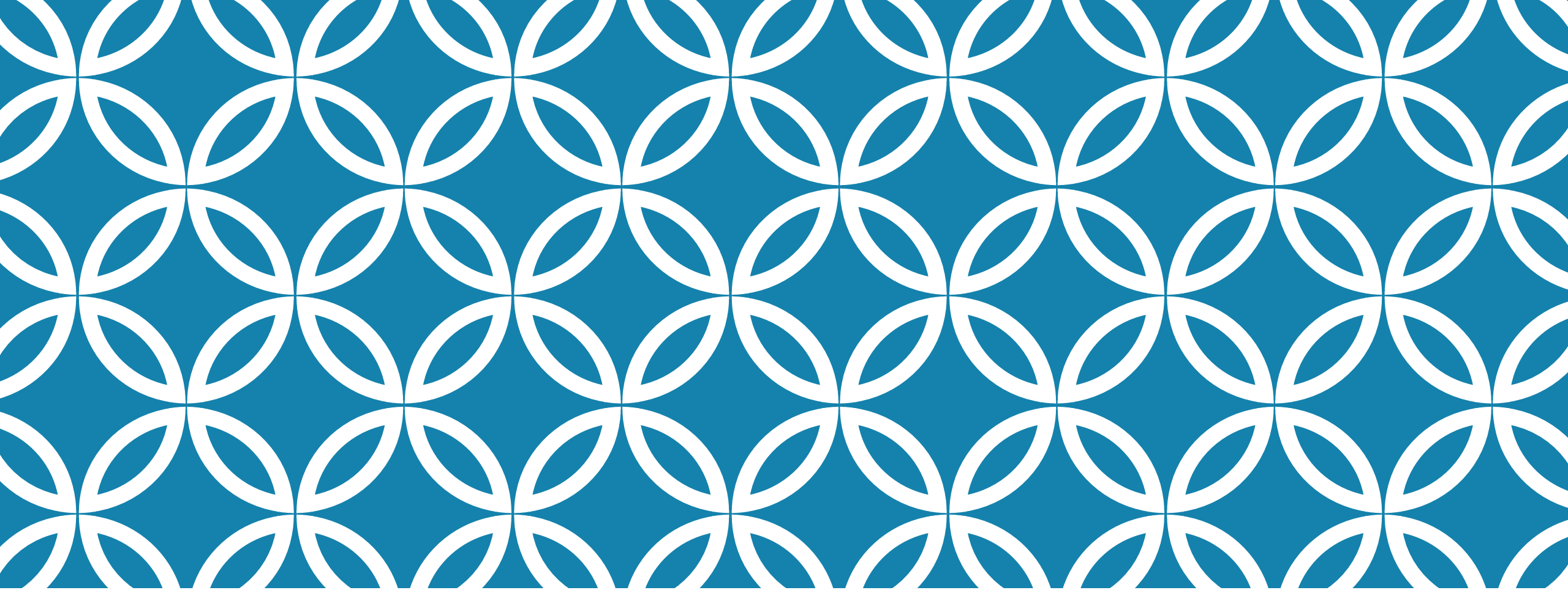


CS 5412/LECTURE 24 FAULT TOLERANCE IN PRACTICE

Ken Birman

Fall, 2022

HOW DO APACHE SERVICES HANDLE FAILURE?

We've heard about some of the main "tools"

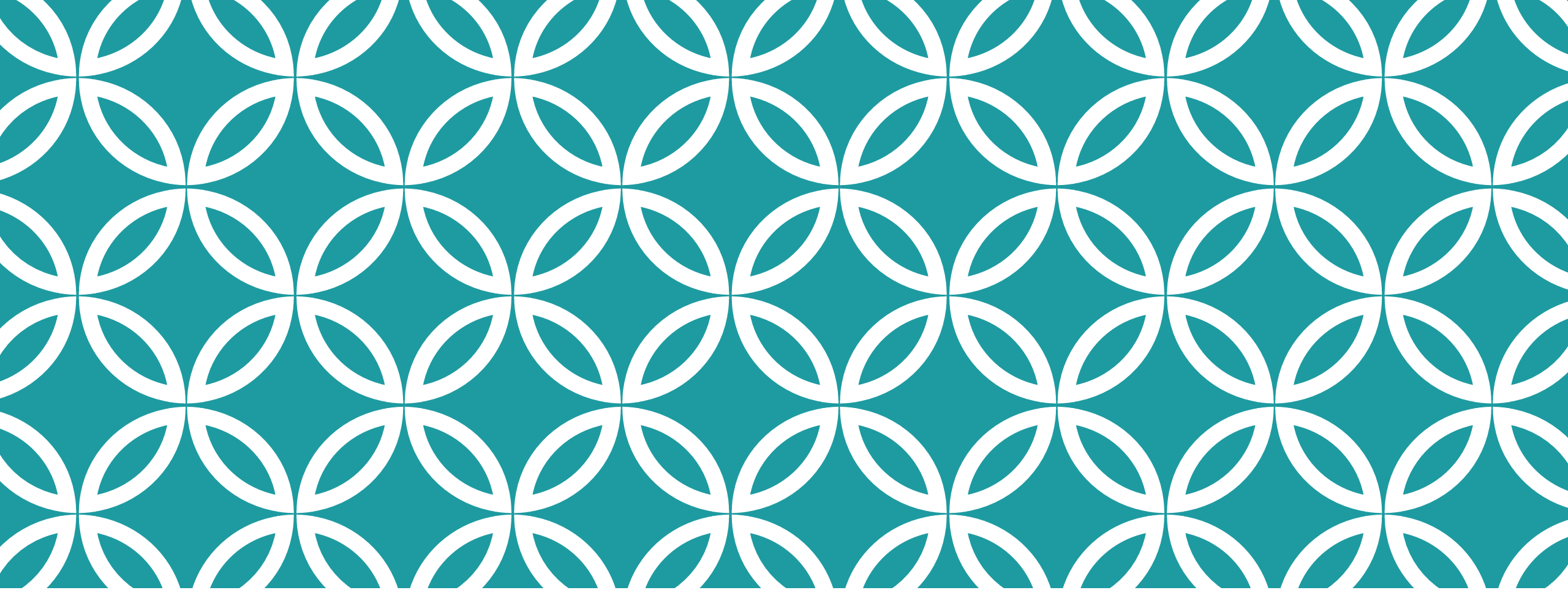
- Zookeeper, to manage configuration
- HDFS file system, to hold files and unstructured data
- HBASE to manage "structured" data
- Hadoop to run massively parallel computing tasks
- Hive and Pig to do NoSQL database tasks over HBASE, and then to create a nicely formatted (set of) output files

BASIC STEPS

Detecting the failure

Reporting the failure to the elements of the application software

Synchronizing so that each layer is fully self-repaired before any other layer manages to “see” it or tries to update it



HOW TO DETECT A FAILURE

It isn't as easy as you might expect

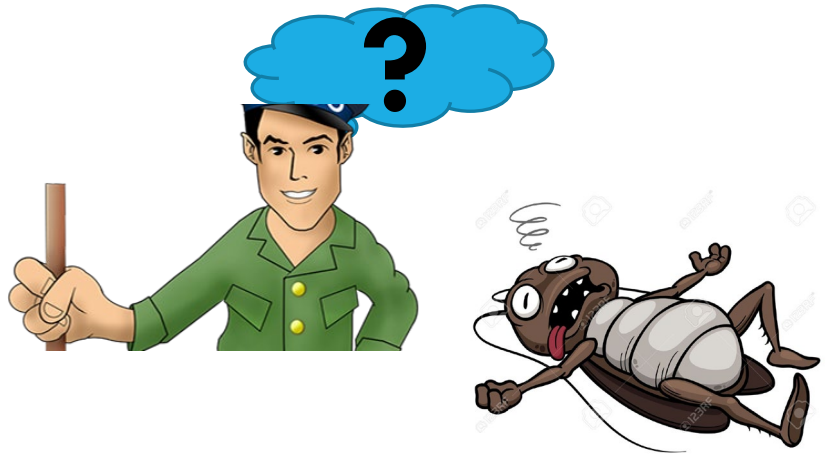
REMINDER: FAILURES AND "TRANSIENTS" LOOK THE SAME

A network or scheduling delay can mimic a crash

FLP was built around the idea of always needing to be certain. If a process was healthy, FLP insists that its vote must be considered.

In "real" systems like Apache, we don't want the to wait while one machine deals with a broken link. We need to move on!

WAYS TO DETECT FAILURES



Something segment faults or throws an exception, then exits

A process freezes up (like waiting on a lock) and never resumes

A machine crashes and reboots

SOME REALLY WEIRD "FAILURE" CASES

When clocks "resynchronize" they can jump ahead or backwards by many seconds or even several minutes.

- What would that do to timeouts?
- Could TCP connections "instantly" break?

Clocks can also freeze up and not move at all during resynchronization.

SOME REALLY WEIRD "FAILURE" CASES

Suppose we just trust TCP timeouts, but have 2 connections to a process.

- What if one connection breaks but the other doesn't?

  … can you think of a way to easily cause this?

This actually can happen!

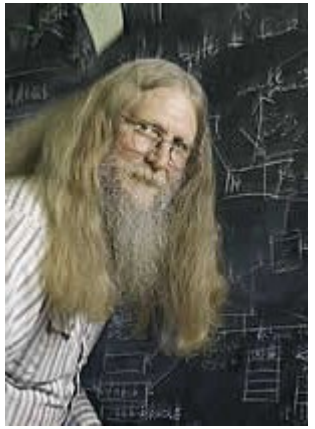
- FTP uses two connections, one for control and one for data movement
- It will crash if either of them goes down

USEFUL PERSPECTIVE FROM STUDIES

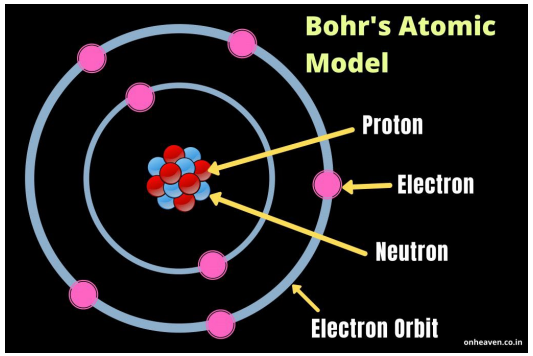
Why Do Computers Stop and What Can Be Done About It? Written around 1980 by Jim Gray.

He blamed operator errors and software bugs. Hardware crashes were rare! This triggered a lot of work on software engineering for reliability in complex distributed systems

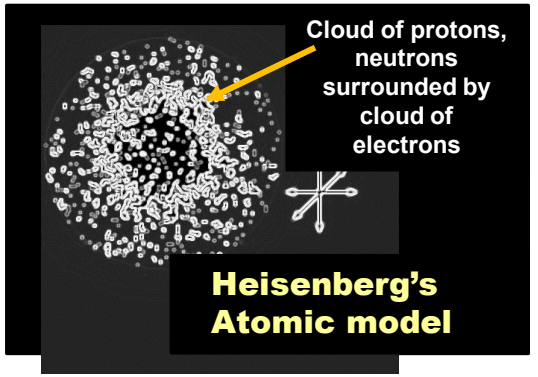
BOHRBUGS AND HEISENBUGS



Bruce Lindsay



Bohrbug is a solid mistake. Easy to reproduce and hence to fix



Heisenbug is often a side-effect of some error earlier in the execution.
Shifts due to concurrency.
Hard to pin it down and fix it.

SAVED BY SOFTWARE ENGINEERING!

The “clean room” development and testing process helps a lot

Design patterns for correct concurrent programming also make a huge difference, such as the “monitor” coordination model in Java and C++

With these methods, software (and even hardware) bug rates have dramatically improved

MORE INSIGHT FROM JIM GRAY'S STUDY

He also found that operators often make errors. Leads to a lot of modern attention on automated self-configuration and cleaner GUIs

And he found that the people who patch software often introduce new bugs! Leads to a modern focus on *regression testing* to make sure patches don't create issues, or just "shift" issues.

MORE RECENT PAPER

Study of datacenter failures by Yahoo:**BFT for the Skeptics**. 2009 SOSP, Flavio Junqueira, Yee Jiun Song, Benjamin Reed

The authors discovered that Byzantine faults basically never occur in datacenters. Similarly, FLP-style problems never occur

But they often saw network outages that caused transient partitioning

BOTTOM LINE?

For better reliability, don't route your internet through a goat pen

Software bugs used to cause most crashes, but are less and less of an issue.

Improvement due to a "cleanroom" coding style, unit testing, release testing, quality assurance team, acceptance testing, integration testing.

Hardware reliability is better too. **Operator error and physical events such as roof leaks remain common causes of major outages**

KEEP-ALIVE MESSAGES

The idea here is to have healthy systems do something actively to report "still ok". They could even run an internal self-check first, periodically.

We call these "keep-alives", meaning "I am still alive, keep me in your list of healthy members.

Each server could send a keep-alive to every other server, or to just a few neighbors. When we talked about gossip, we saw this in Kelips.

WHAT IF A TIMEOUT GOES OFF "INCORRECTLY?"

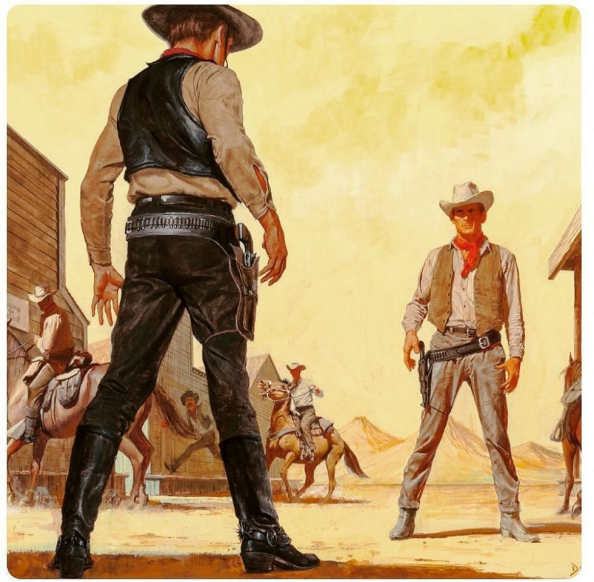
Recall James Hamilton's 3 R's

- Reboot, then
- Reimage, then
- Replace

Most data centers today offer a way that one machine can toggle the power switch for some other machine! This guarantees a reboot

DUELING REBOOTS

Obviously, we could have a shoot-out



But this won't happen often

James Hamilton would say that if something like this occurs, there is probably a root cause like a broken network connection, and very likely we actually should exclude those machines in any case!

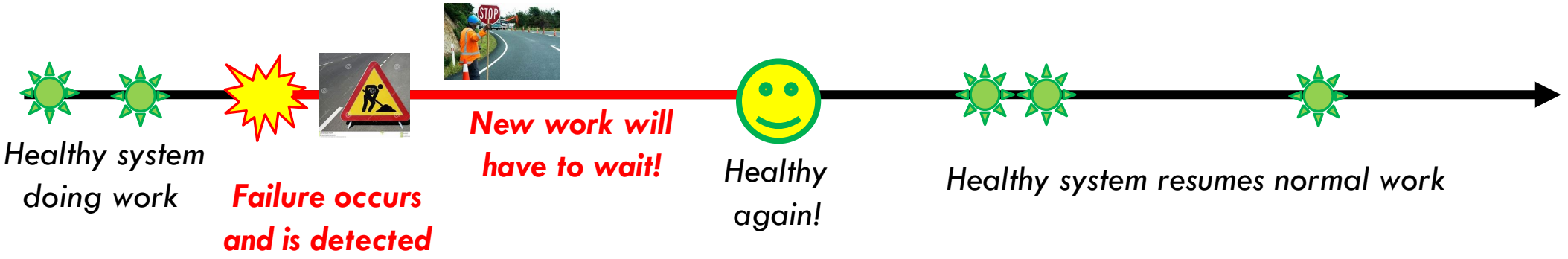
BROADER PHILOSOPHY: THE CLOUD MUST STAY ON

Never let the fate of one or two machines somehow control the cloud!

A cloud system is replicated and elastic and large

We should always keep the majority up and running, even if this trims away a bunch of machines because they seem to be broken. Their "opinion" that they are healthy is not relevant!

AS A TIME-LINE



Any system needs to go through a series of stages to deal with failues

If the failure could have damaged data or left an execution in a disrupted state, cleaning up will be important.

THREE KINDS OF ISSUES TO THINK ABOUT

How does each element work when things are healthy?

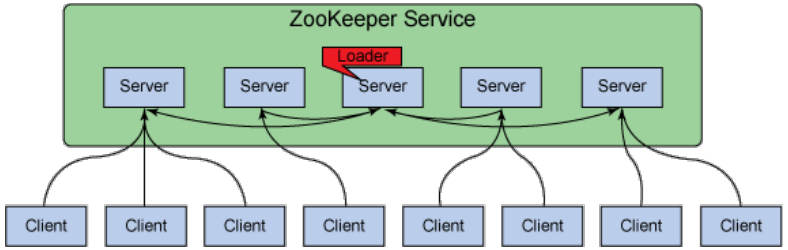
How does each element detect failures, and if needed, repair itself to recover from damage the fault might have caused (such as a file that wasn't fully written, and should be deleted and regenerated)?

How do the layers synchronize? If layer A lives on layer B, when layer A is ready to restart after a crash, can it be sure that B is already repaired?

ZOOKEEPER ROLE

One ring to rule them all

ZOOKEEPER "REMINDER"



Zookeeper is an implementation of a virtual-synchrony service (they don't use these words, but they do use this model!) to track system state and configuration data.

The basic idea is that your application processes connect to Zookeeper via TCP. It monitors their health. And it reports membership via a small file.

ZOOKEEPER FILES

Limited size (currently 1MB). Version replacement feature eliminates the need for costly locking.

- Notification when a file you are watching is updated.
- Zookeeper itself has a file holding membership data (process IP addresses). Detects failures if a TCP connection breaks, auto-updates.
- Applications can also create files. They store configuration data in them.
- Files are held in memory.

GUARANTEES?

Zookeeper is a lot like any virtual synchrony system

- The service itself is replicated over 3-5 nodes for availability
- It will never suffer split-brain problems
- State machine replication for the files it manages.

In effect, a POSIX file system but just used for configuration data

TWO FORMS OF MEMBERSHIP!

Zookeeper tracks its own membership… in this example, the five servers

When the Zookeeper service itself is not reconfiguring from failure, Zookeeper is tracking the status of clients (≡ members of the application)

- Client joins? Zookeeper uses atomic multicast to update the list of active client processes and to track any metadata the client supplies
- Client fails? Zookeeper learns this (as we discussed) and then notifies all healthy clients that the membership file has been updated

CLIENTS WILL SEE THESE UPDATES

They reread the file and adjust their internal data structures

For example, any client with a TCP connection to a failed client closes that connection.

During this transitioning period, some clients might have switched to the new membership and others could be lagging. Developers must plan for this and tolerate this period of asynchronous switchover!

ATOMIC MULTICAST: WEAKER THAN PAXOS!

Paxos also has a durability guarantee… and it applies to every update.

Zookeeper doesn't have true durability. It does periodic checkpoints once every K seconds, where K must be ≥ 5s.

- For their protocols, making each update durable immediately is slow
- Derecho doesn't have this issue, but our protocol is more efficient.

KEY INSIGHT

Atomic multicast protocols are designed to:

1. Send copies of messages to all participants… *but they won't yet be delivered to anyone*. Called **pending** messages

2. Wait until every healthy receiver has a copy
   a) This status might be detected by a leader: the 2-phase commit pattern
   b) Or it might be independently/directly sensed by members: Derecho SST approach

3. Only at when all have a copy do deliveries occur.

CONTRAST WITH PAXOS

Paxos does durable log updates, but has also been proposed as the basis of multicast protocols. In Paxos, pending messages arise in phase 1

Recall that in this phase leader(s) contend to get their proposed message into a quorum Q_W of slots using a series of ballots

Phase 2 commits the successful message, but at this point it is already certain to be in Q_W logs, saved to disk.

FAILURE DISRUPTS A MULTICAST?

Virtually synchronous systems like Zookeeper and Derecho "self-repair, then advance" with no real rollback (no delivered message is ever withdrawn)

Pattern seen in these:

- Freeze everything. Leader collects current status from healthy members
- For each pending multicast, decide if it should be delivered or not. Leader commits this outcome via 2-phase protocol involving healthy members.
- Pending multicasts that were not delivered are resent in the next epoch (view)

FAILURE DISRUPTS A MULTICAST?

In Cascade, layered over Derecho, the effect is that failures are "erased" and completely hidden from higher-level application logic.

But more complex outcomes can still arise because the failure may also have disrupted applications running on other nodes.

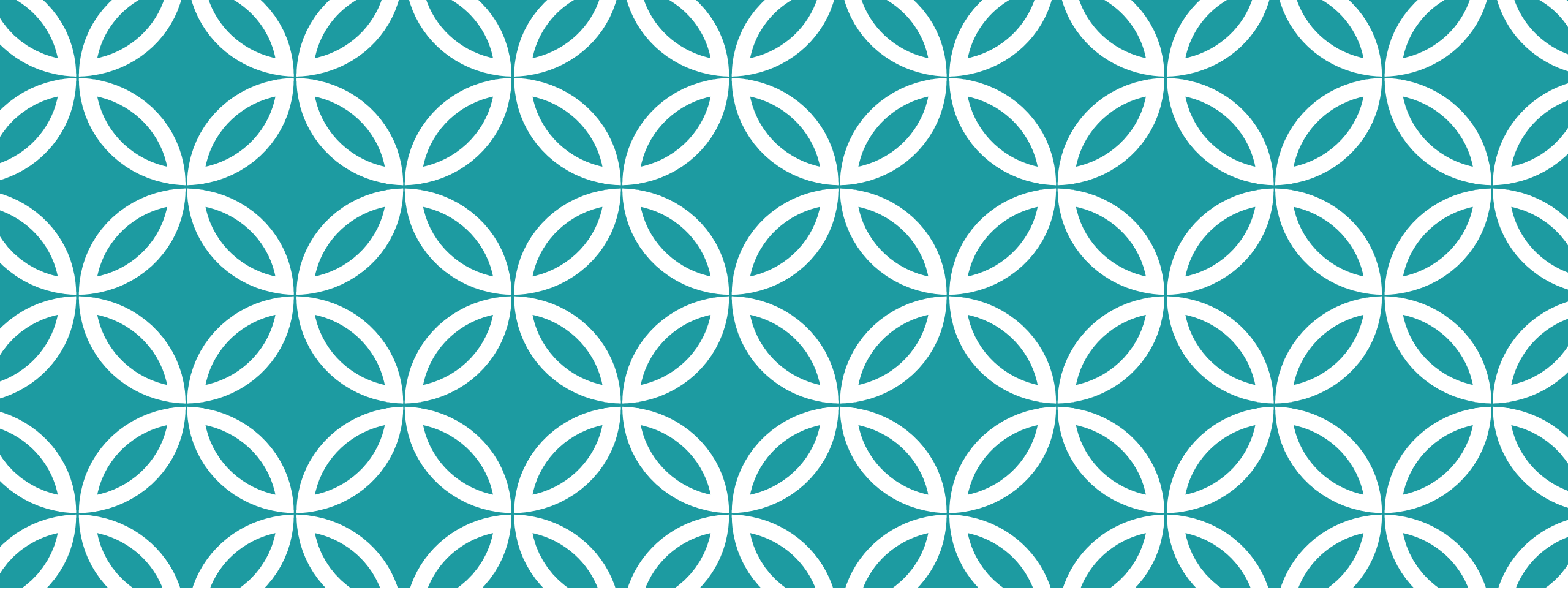
Apache examples illustrate this and how it is resolved.

WHAT IF A FAILURE DISRUPTS PAXOS?

This can become extremely complex! Not a topic for CS 5412

But there are quite a few cases one must consider. In the worse case,Paxos can "commit" to leave a log entry empty – it commits $\perp$ in that slot

Requires a case-by-case analysis and proofs. Paxos was so hard to reason about that it led Lamport to create the TLA+ theorem prover! At Cornell, we favor Coq/VST, Dafny/Z3 or Ivy/Z3 for this kind of protocol verification.



LAYERED RECOVERY

Unfortunately, this is very complex

THE PUZZLE

Suppose that Zookeeper notifies HDFS, HBASE, Hadoop and Apache SQL that some nodes have failed. They might *all* need to self-repair.

But HBASE can't self-repair until HDFS has finished! And Hadoop can't restart jobs until HBASE is self-repaired, etc.

In effect, we need a layer-by-layer recovery

THE APACHE SELF-REPAIR LOGIC IS TRICKY

A good explanation of HDFS self-repair can be found here.

It is many pages long, and very detailed, requires a lot of internal knowledge about how HDFS is implemented.

We won't do a deep dive…. Layer by layer synchronization isn't standardized in Apache. The HDFS version involves "refreshing" something called a "block lease" and is integrated with the HDFS replication logic

IT CENTERS ON A CONCEPT CALLED A "LEASE"

The term means "time-bounded lock"

HDFS has an internal design in which data actually is stored into replicated blocks. While doing a write, the application must have a block write lease. To read the block, the application must have a read lease.

HDFS uses leases as its main tool to buy time for self-repair when a crash disrupts it.

HOW WOULD HDFS SELF-REPAIR? HBASE?

HDFS learns that a failure has occurred when Zookeeper updates the map of system components and roles. HDFS then invalids active leases while it repairs any damaged blocks.

Now, think about HBASE. All HBASE state is stored in HDFS, so if HBASE is running and HDFS has a crash, HBASE must refresh its block leases.

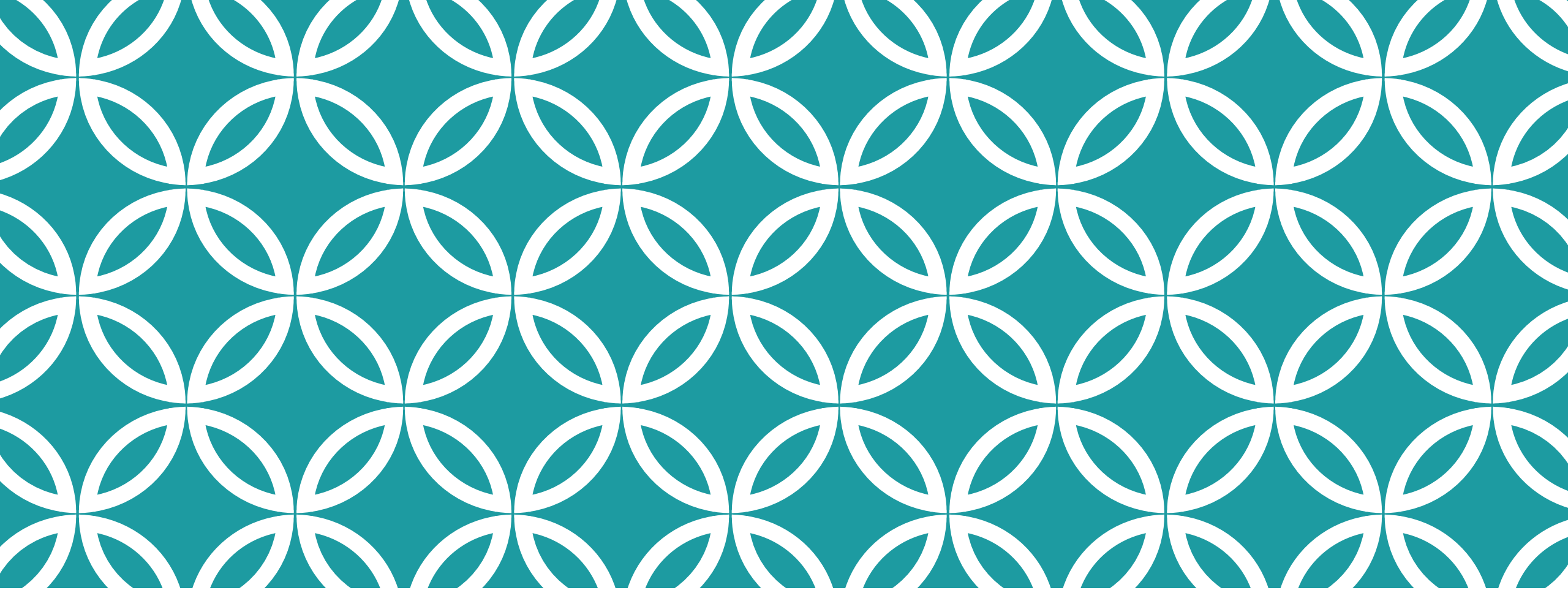
This refresh waits until HDFS self-repair completes. HBASE itself is stateless (HBASE state lives in HDFS files), so at this point HBASE has been self-repaired too.

WHAT ABOUT HADOOP?

Hadoop itself is stateless too, like HBASE. So in some sense the same policy works for Hadoop.

But now we have layers of higher level applications! Each needs services from layers below it. This stack "terminates" in HDFS leases.

In more complex systems, we might have more than one stateful service. That makes recovery trickier!



EXAMPLE OF A COMPLEX RECOVERY

Hadoop job disrupted by a crash when HDFS itself was also disrupted by the crash

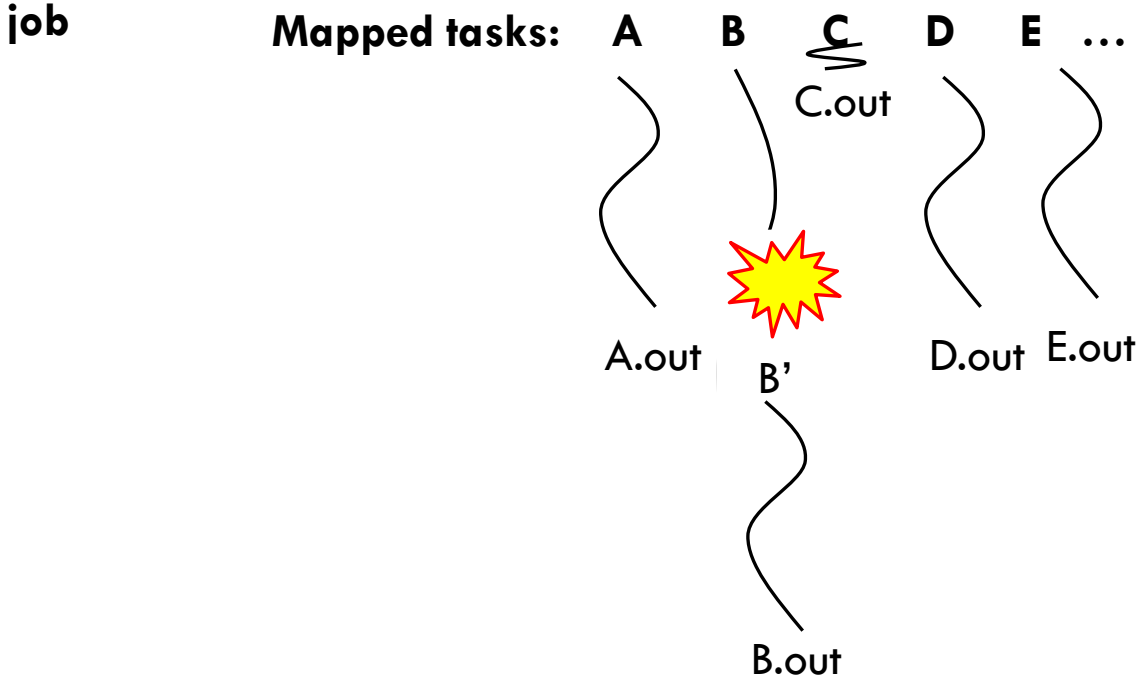
REMEMINDER: HDFS CHECKPOINTS

HDFS adds a "checkpoint" feature to what POSIX normally can support.

The checkpoint is just a file that contains the names, version numbers and lengths of the files your Hadoop application is using. To "roll back" it just truncates files back to the size they had and restores any deleted files.

This doesn't work for files you deleted or replaced. So Hadoop jobs need to delay deleting or replacing files until the very end – the last action.

VISUALIZING HADOOP FAILURES IN IMAGES



Normal case: A, B… E just run, create output (key,value collections in HDFS files), then the reduce step can run.

Failure case (B crashes). Now Hadoop just rolls back any files B was appending to and runs B', to repeat the task.

OTHER ASPECTS OF ROLLBACK

Hadoop also has to worry about jobs that might be running exactly when the failure occurred.

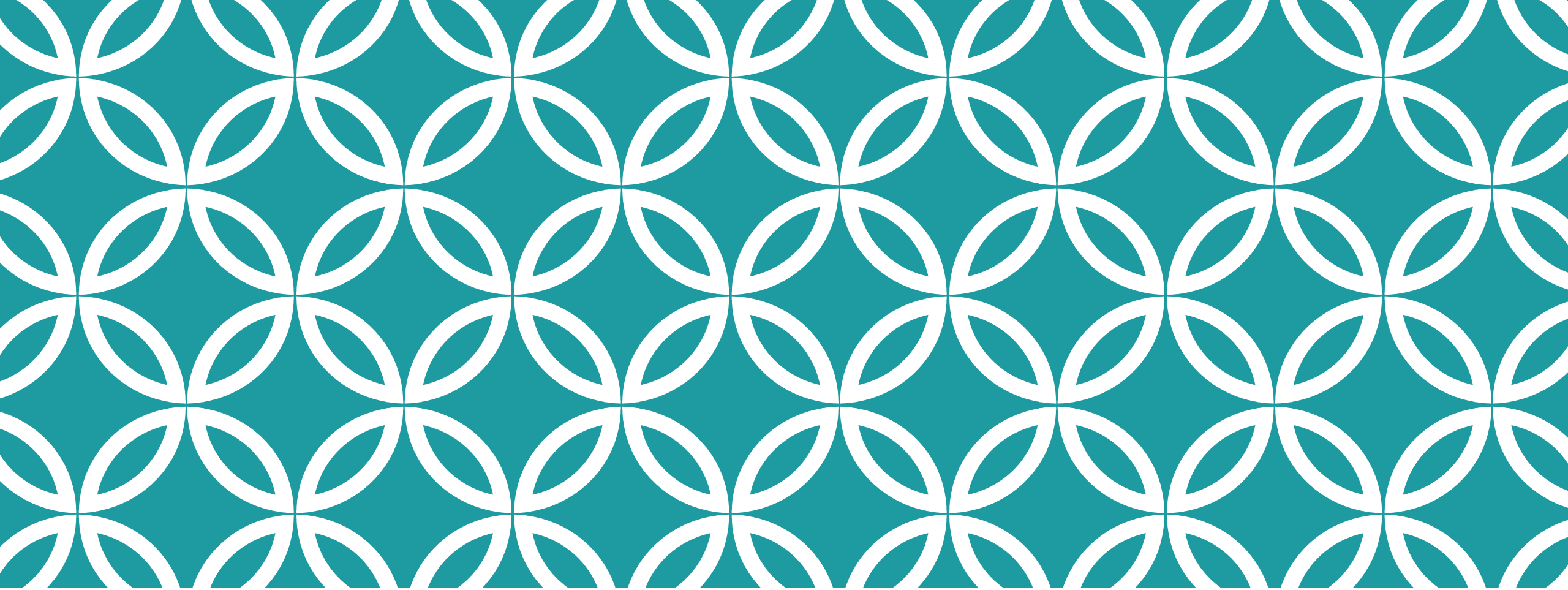
Very often such a job could be disrupted in some way, hence active tasks shouldn't be allowed to continue running "as is".

Hadoop kills all the user-generated tasks, removes any files they may have created and restores any then deleted, then reruns the failed tasks.

THIS CONCEPT WORKS IN ALL OF APACHE

The whole Apache infrastructure centers on mapping all forms of failure handling to Zookeeper, HDFS files with this form of "rollback", and task restart!

It has similar effect to an abort/restart in a database system, but doesn't involve contention for locks and transactions, so Jim Gray's observations wouldn't apply. Apache tools scale well (except for Zookeeper itself, but it is fast enough for the ways it gets used).



AIR TRAFFIC CONTROL FAULT TOLERANCE

A non-Apache example

WHAT ABOUT AIR TRAFFIC CONTROL?

These systems have a system-wide virtual synchrony view manager.

The role of the view manager is to atomically report to all components when any failure disrupts any component. When a view changes, all components instantly "wedge" and adjust to the new view

The system briefly (seconds… not minutes) freezes up and repairs itself, and when it resumes, every component is back to a healthy state.

WHAT ABOUT AIR TRAFFIC CONTROL?

These systems have a system-wide virtual synchrony view manager.

The role of the view manager is to atomically report to all components when any failure disrupts any component. When a view changes, all components instantly "wedge" and adjust to the new view

Rollback is not an option!

WHAT ABOUT AIR TRAFFIC CONTROL?

These systems have a system-wide virtual synchrony view manager.

The role of the view manager is to atomically report to all components when any failure disrupts any component. When a view changes, all components instantly "wedge" and adjust to the new view

The system briefly (seconds… not minutes) freezes up and repairs itself, and when it resumes, every component is back to a healthy state.

FAILURE IN AN ATC SYSTEM

A modern air traffic control system might have a structure like this:

Air traffic controllers update flight plans

Flight plan manager tracks current and past flight plan versions

Message bus

. . .

Microservices for various tasks, such as checking future plane separations, scheduling landing times, predicting weather issues, offering services to the airlines

Flight plan update broadcast service

WAN link to other ATC centers

COMPARE WITH APACHE?

Air traffic control might have state in a few places, but it helps that the flight plan records reside in a single database that every other process simply mirrors, read-only.

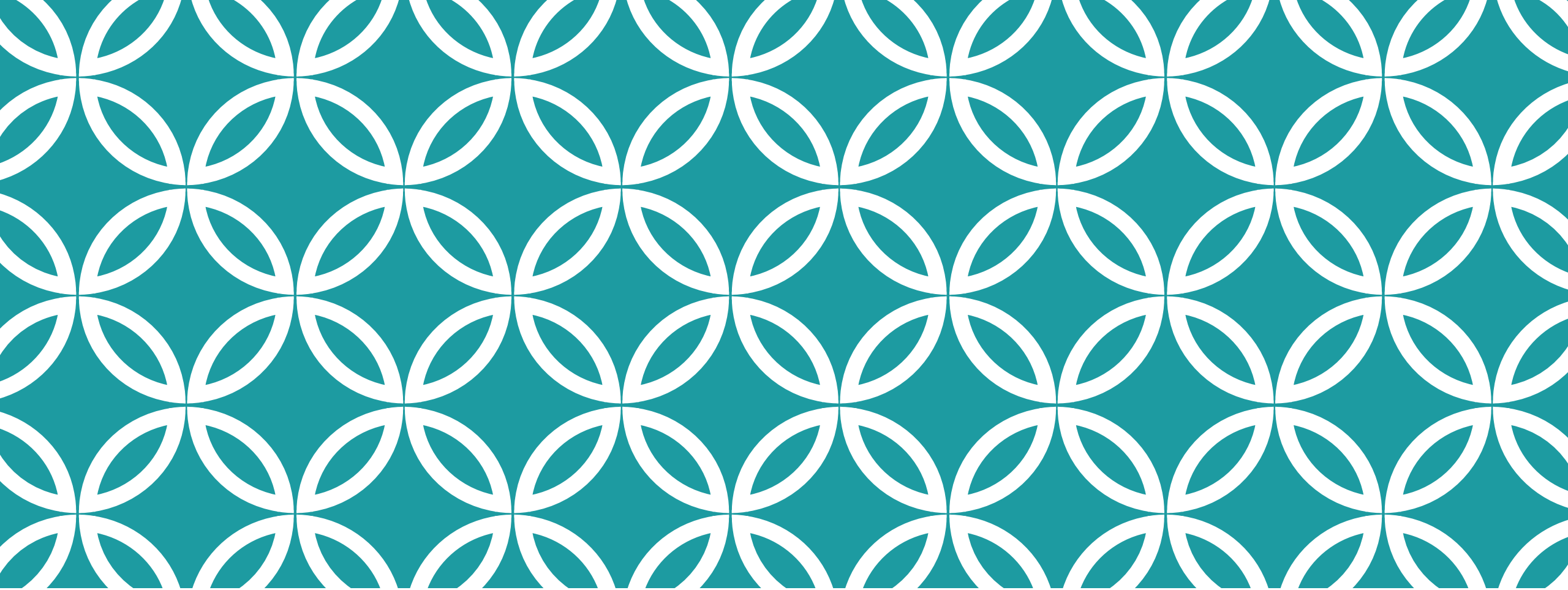
The key thing is to ensure that we never have parts of the system using one set of flight plans, or one set of configuration files, while the remainder is using a different set. And this property is very carefully verified.

CONNECTION WITH DERECHO/CASCADE

The ATC architecture we are looking at was literally built using the Isis Toolkit, an older version of Derecho

So Derecho can be viewed as using this exact same approach!

- Failure sensing via timeouts or "application detected issues"
- Spread-the-word: Uses a gossip approach, "shun" the failed process
- Freeze and self-repair, then restart and reissue any requests that got wiped out by the rollback



WRAPUP

Sense failures, force reboots if needed. Do a clean rollback, resume from checkpoint

BOTTOM LINE?

The cloud is highly available, because it has layers of backups– even backup datacenters and backups at geographic scale.

IoT data managed by the cloud c*an* be strongly consistent. This doesn't really reduce availability and in fact doesn't even reduce performance.

It leads to a style of coding in which membership is managed for you. But many parts of the existing cloud are using weaker consistency today, and you need to be aware of the risks when you use those tools.

EFFECTS OF BOTTOM LINE?

Today's cloud is remarkably robust.

We use CAP and weaken consistency in outer layers, but this is partly because doing so actually simplifies the solutions we create. Fault tolerance is easy when you don't worry about consistency.

Systems that do need consistency use the "timeline": they have a standard way to detect failure. Every component learns of any fault relevant to it. Disrupted components pause their work queues while they self-repair.