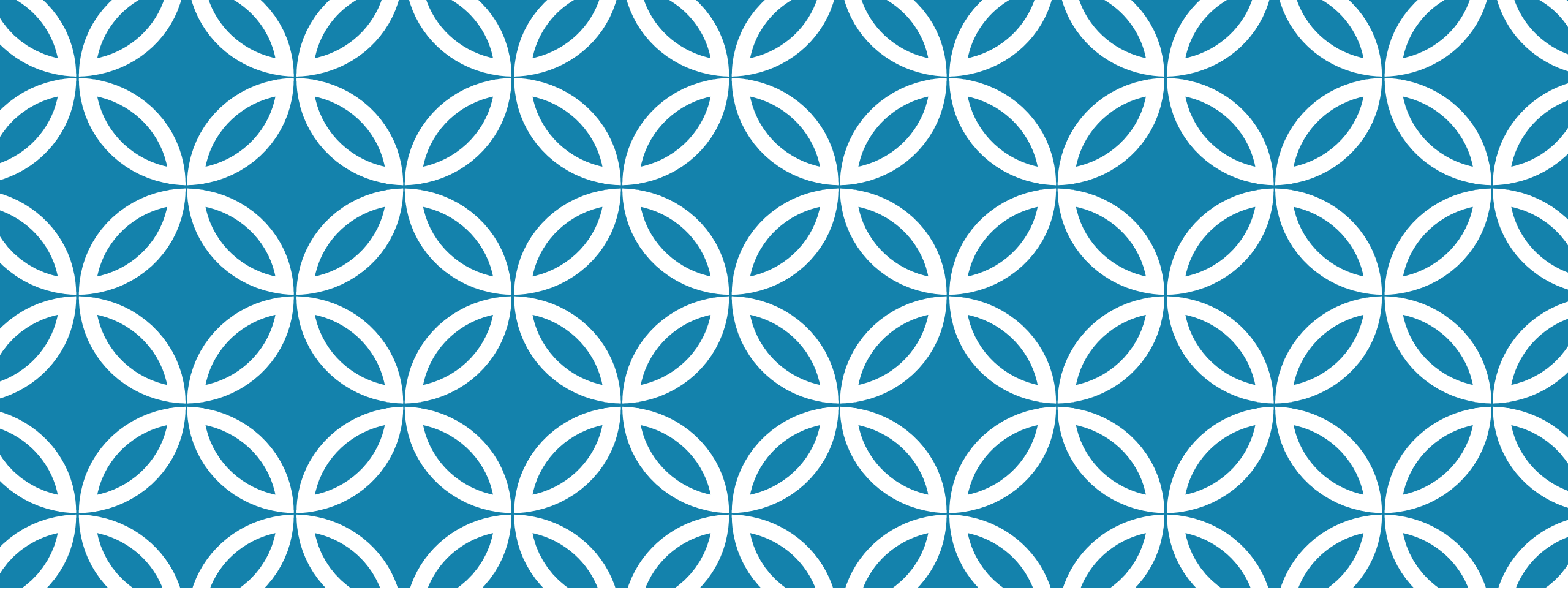
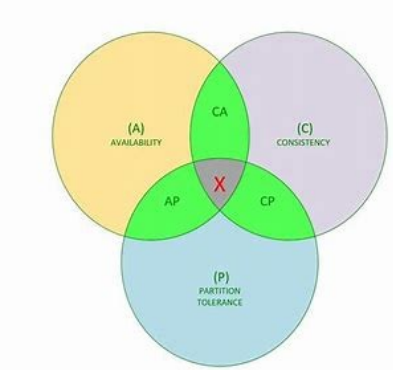


CS5412 / REPLICATION AND CONSISTENCY (ENRICHMENT/REVIEW)

Ken Birman

Fall, 2022

CAP USES "PECULIAR" DEFINITIONS

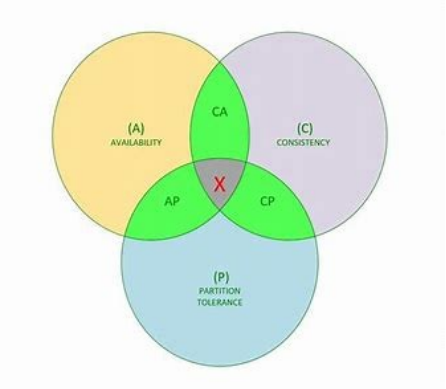


Consistency (for Eric) means: *Uses the most current versions of data (he isn't worried about update ordering, but probably should!).*

Availability (…): *Able to give a fast response without waiting for anything else (should also include fault-tolerance, but Eric didn't)*

Partition Tolerance (…): *No need to worry if you can't get a rapid reply from a back-end storage service. (Should include "no split brain" too)*

CAP (FOLK) THEOREM



You can only have two of C+A+P, so favor A+P… don't worry about C? CAP and BASE work for many cloud computing web sites, like Amazon.com

But you don't always need to abandon C. If *a μ-service does the whole job* –holds its own state and does the computing, using primary-partition membership – then P won't matter. We can opt for C+A.

This motivated state machine replication and virtual synchrony.

RECAP: STATE MACHINE REPLICATION

This is a model in which we have *deterministic programs*

They see one update at a time and apply the updates in the same order. There is a communications version of this (atomic multicast) and a log-append version (persistent replicated logging).

If our replicas start up in sync, they stay in sync.

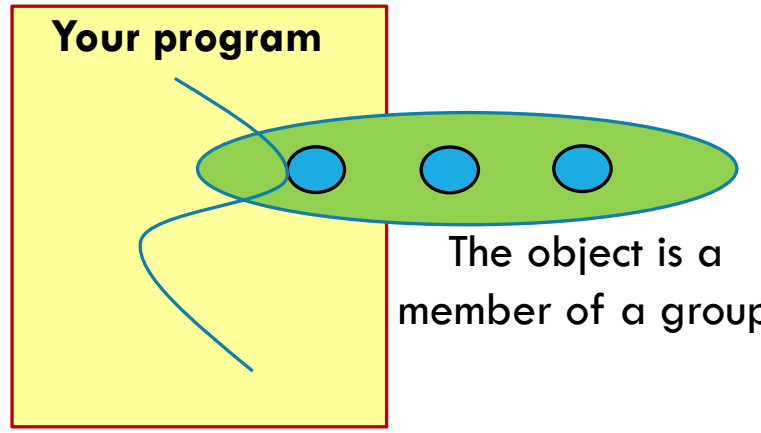
DETERMINISM ISSUE, AND SOLUTION

Programs are rarely deterministic

But it is possible to build fully deterministic data types (classes), and then use state machine replication just in those kinds of objects

You end up with server instances doing different things, yet with a subsystem that can maintain consistent state across them!

OBJECT ORIENTED COMPUTING



The idea is to think of your main program as thread that shares the address space with one or more objects, like the blue circle.

… This blue object is a member of a replicated group: the green oval

This is why your program can be non-deterministic and yet the object it created and uses can be deterministic and use state machine replication.

RECAP: ATOMIC MULTICAST AND DURABLE REPLICATION

Both ideas center on applying the same updates in the same order

With atomic multicast updates are carried in messages, and delivered to all receivers (or none), in the identical order, even despite failures.

With durable replication each receiver also keeps a log of updates or of the result after applying them.

- In some solutions each replica (each log) is itself a complete history.
- In other solutions logs must be merged to know the full update sequence.

PAXOS FOCUSES ON QUORUMS IN A FIXED SET

In Paxos we normally assume the set of servers is fixed

If it gets updated, that happens offline

So there are N servers – but some of them might be crashed, or not responsive (overloaded, network outage, etc).

PAXOS WITH N MEMBERS, F POTENTIAL CRASHES

We want to be sure that any two write quorums overlap:

- QW+QW > N

We also want any read to "see" all the prior committed writes

- QW+QR > N

For fault-tolerance, we need QW ≤ N-F and QR ≤ N-F.

For maximum speed, QR should be as small as possible

RECAP: CLASSIC PAXOS

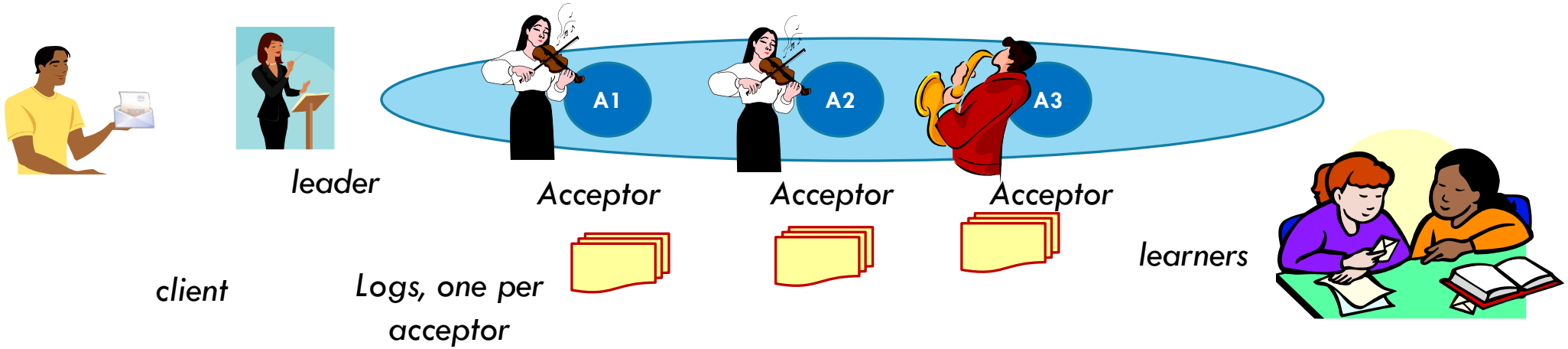
Runs on a fixed set of machines, usually N=3 or N=5

Updates should go to a write quorum, QW < N. QW+QW > N.

Reads must merge QR logs, also QR as small as possible, QW+QR > N

Roles: client, leader, acceptor, log, learner

VISUALIZING PAXOS SETUP



The client asks the leader to add a message to the Paxos logs. Paxos is like a "postal system". The leader will be in charge of this request.

The system "discusses" the letter for a while (the first phase, which picks the slot in the log, stores the letter in the log, and reaches QW acceptors).

Once the update is "committed" the learners can execute the command

PAXOS BALLOT NUMBERS

Paxos log has "slots" but two or more leaders could try to put different proposed messages into the same slot.

This leads to the idea of "ballots" – the first phase loops, with the leader trying to get QW successes on a series of proposals, with an increasing ballot numbers

Ideally, with one leader, the mechanism decides on the first ballot. But with multiple contending leaders it could loop for a while.

THOUGHT QUESTIONS (WE WON'T ANSWER THEM)

Suppose F becomes bigger, and this causes us to make N larger. One benefit of having more servers is that reads should be faster.

- Will this advantage be seen in Paxos? Why or why not?

Give a concrete example of how P and Q in some Paxos group might have different logs. Does this mean that one of them is having a Byzantine fault?

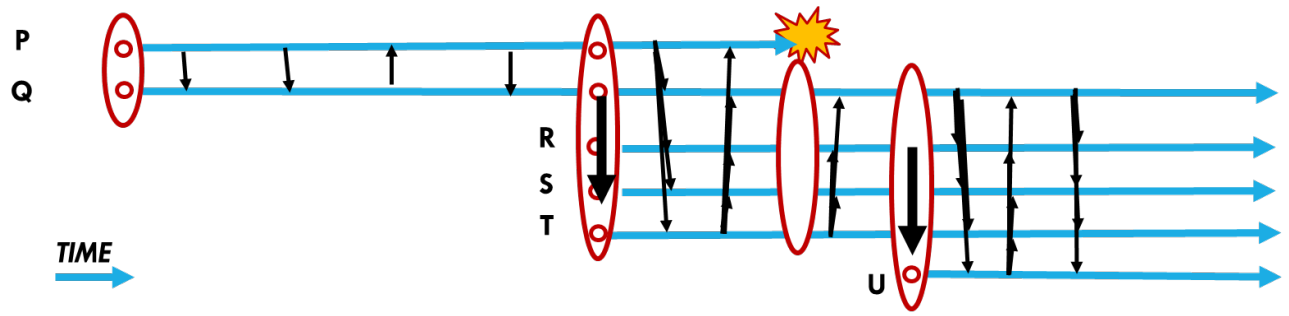
THOUGHT QUESTIONS (WE WON'T ANSWER THEM)

Suppose that some learner merges QR logs, maybe the logs from P and Q. Later more updates have occurred. Should the learner restart from scratch and merge a new set of QR logs, or can it "cache" the answers and just update the tail of its merged log the next time a read query is issued?

Must every read check to see if the log has gotten longer?

When we do need to update a log, does the learner need to read from the same logs it used the first time? Or can it read some other set of QR logs?

RECAP: VIRTUAL SYNCHRONY



The idea was to break down a distributed systems into a

- **Multicast or durable update protocol.** Sends messages only while membership is stable.
- **Membership service.** Tracks which processes are in the system, and what role each process is playing (like which shard it is in). Design it to never experience a split-brain outage.
- **State transfer mechanism.** Uses checkpointing to initialize a joining process.

In virtual synchrony, membership changes only when updates are frozen and vice-versa. The multicast and state transfer logic becomes much simpler.

VIRTUAL SYNCHRONY HAS COMPLETE REPLICAS

Every member of the current view sees every update in the epoch

Thought questions:

- Does this mean we no longer need to merge QR logs?
- Is this saying that Paxos would always be slower than virtual synchrony for doing reads (queries)?
- With an atomic multicast that updates some variable, like a counter, can an object instance just use its local replica safely? Or does it have to first check to see if other replicas have the same value in their versions?

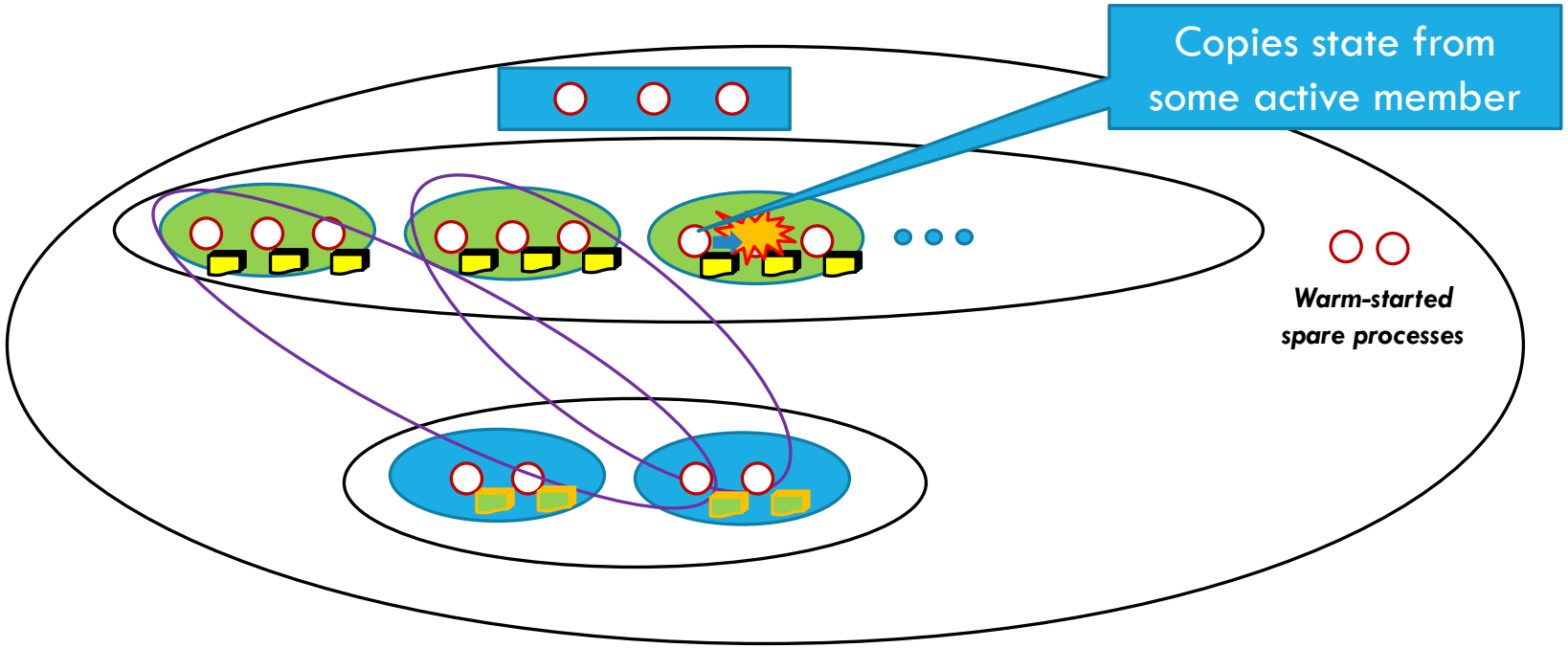
NOTE: CLASSIC PAXOS ≠ VIRTUALLY SYNCHRONOUS PAXOS

Virtual synchrony adjusts membership to drop a failed process. But updates always reach every process in the current view.

Paxos just omits the failed or slow process from the update, but a result is that logs can have gaps. Reading always must merge logs.

Paxos runs in stages: a 3-stage commit. Virtually synchronous Paxos (Derecho) streams data, separately streams control information.

DERECHO'S EPOCH MECHANISM IN ACTION



…. If a failure occurs, Derecho automatically repairs it.

THOUGHT QUESTIONS

Suppose that Derecho is being used to replicate data in some persistent (logged) shard.

A crash causes *all* the members of that shard to fail – not just one like in the illustration on slide 18

Can Derecho replace *all the failed members* with standbys? Or will it force the application to wait until one or more failed members recovers? Why?

IS PAXOS MORE AVAILABLE THAN DERECHO?

Can you identify a case where Derecho can make progress, but Paxos is stuck?

Can you identify a case where Paxos can make progress, but Derecho is stuck?

Which is faster for doing reads, if a system has mostly read-only queries?

What does this tell us about which is “best”?

DERECHO IS…

Quite efficient. Because its data plane aligns better with fast networks and the control plane offers a simple, natural form of batching, Derecho is much faster than classic Paxos (as much as 10,000x)

But not so easy to use: You need to be a C++ programmer

Basis for Cornell Cascade project, which we will look at in Lecture 10

MODERN CHOICES

Most systems are built today using Zookeeper, RaFT or LibPaxos (in that order).

If you joined a big company like Facebook, and discovered this, would you urge them to migrate their platform to Derecho?

What would be some reasons in favor of that? What would be some reasons for *not* migrating?