Don't Get Stuck in the "Con" Game

CONSISTENCY, CONVERGENCE, AND CONFLUENCE ARE NOT THE SAME!

PAT HELLAND

EVENTUAL CONSISTENCY AND EVENTUAL CONVERGENCE AREN'T THE SAME AS CONFLUENCE, EITHER.

Like many others, I've fallen victim to using the phrase *eventual consistency*. It's a popular phrase, even though its meaning is fuzzy. Different communities within computer science use the word *consistency* in varying ways. Even within these different communities, people are inconsistent in their use of *consistency*. That fuzziness has gotten many of us tangled up in misunderstanding.

It turns out that there are other terms, *convergence* and *confluence*, that have crisper definitions and are more easily understood than *consistency*.

WHAT IS MEANT BY CONSISTENCY? WHAT ABOUT EVENTUAL CONSISTENCY?

When database people refer to consistency, they loosely mean ACID (atomicity, consistency, isolation, durability) transactional consistency

*It's Not Your Grandmother's Database Anymore

(approximately the same as serializability). My dear friend, Andreas Reuter coined the term *ACID* in 1983 in Principles of Transaction-Oriented Database Recovery. Recently, when I asked him what the C meant, he said, "THE APP CAN CONTROL WHEN THE DATABASE TRIES TO COMMIT."

In other words, don't commit a proper subset of the updates for the transaction. Don't commit early. That, combined with isolation, means the application can control its very own consistency of the data. That's not QUITE the same way most database people interpret consistency. (See ACID: My Personal "C" Change.) Database people (including me) are even confused about their own meaning of consistency.

Distributed-systems people sometimes use the word in a different fashion. When asking them what consistency means, the answer has historically been, "EACH OF THE REPLICATED VALUES FOR OBJECT-X HAVE THE SAME VALUE." There is no notion of work across the objects, just the individual objects and their values. Today, this is more commonly called *convergence*. When my friend Doug Terry coined the phrase *eventual consistency* in the Bayou paper in 1995, he meant that for each object in a collection of objects, all the replicas of that object will eventually have the same value.

When I asked him about it years later, he said, "YEAH, I SHOULD HAVE CALLED IT *EVENTUAL CONVERGENCE*." The distributed-systems people are also confused about their interpretation of consistency.

Looking at the proof of the CAP theorem in Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services, by Seth Gilbert and Nancy Lynch (2002), shows yet another meaning of

consistency. Of course, the proof has a different definition from the other descriptions of CAP that I located.

Distributed-systems people and database people get wrapped around the axle when discussing what is meant by eventual consistency within their own communities; it gets even uglier when they talk to people in the OTHER community.

CONVERGENCE AND EVENTUAL CONVERGENCE

Let's consider the definition of *convergence*. My friend Peter Alvaro addresses convergence in his 2015 Ph.D. dissertation at UC Berkeley, Data-centric Programming for Distributed Systems (page 76):

> A SYSTEM IS CONVERGENT OR 'EVENTUALLY CONSISTENT' IF, WHEN ALL MESSAGES HAVE BEEN DELIVERED, ALL REPLICAS AGREE ON THE SET OF STORED VALUES.

This is essentially the same as seen in a 2011 paper by Marc Shapiro et al. about CRDTs (conflict-free replicated data types), although Alvaro focuses on a system with a set of eventually convergent values and Shapiro et al. focus on a single object. Shapiro et al. say:

> STRONG CONVERGENCE: CORRECT REPLICAS THAT HAVE DELIVERED THE SAME UPDATES HAVE EQUIVALENT STATE.

Note that this is also how Doug Terry used *convergence* and *eventual convergence* in my chat with him. Convergence is a property of individual objects and their merge function.

Eventual convergence is a phrase with "too much extra redundancy." Convergence means convergent, eventually or not. Still, eventual convergence sounds cool, so I'll go with it.

LINEARIZABILITY

Linearizability is not the same as convergence.

Every distributed-systems geek should study Linearizability: a Correctness Condition for Concurrent Objects, by Maurice Herlihy and Jeannette Wing (1990). I did not STUDY it until recently. It is an amazingly clear and crisp definition of a correctness criterion for some objects within distributed systems. After reading this paper, I can easily and clearly explain linearizability.

To quote Peter Bailis from his famous blog post, Linearizability versus Serializability:

> LINEARIZABILITY IS A GUARANTEE ABOUT SINGLE OPERATIONS ON SINGLE OBJECTS. IT PROVIDES A REAL-TIME (I.E., WALL-CLOCK) GUARANTEE ON THE BEHAVIOR OF A SET OF SINGLE OPERATIONS (OFTEN READS AND WRITES) ON A SINGLE OBJECT (E.G., DISTRIBUTED REGISTER OR DATA ITEM).

As I learned from the Herlihy and Wing paper:

> UNDER LINEARIZABILITY, OPERATIONS SHOULD *APPEAR TO BE INSTANTANEOUS FROM THE PERSPECTIVE OF THE OBJECT.* OPERATIONS HAPPEN ONE AT A TIME (AS SEEN BY THE OBJECT) AND *EACH OPERATION ACTS ON THE STATE OF THE OBJECT AS DERIVED FROM ITS LOCAL HISTORY.*
>
> CLIENTS SEE *INVOCATIONS OF AN OPERATION* ON A LINEARIZABLE OBJECT FOLLOWED LATER BY *RESPONSES TO THESE OPERATIONS*. FROM THE

PERSPECTIVE OF THE CLIENT, THE OPERATION OCCURS AT THE OBJECT SOMETIME BETWEEN THE INVOCATION AND THE RESPONSE.

Linearizability defines a relationship between the partial order (or history) of operations on an object and the partial order (or history) of operations as seen at each client. While wall-clock time provides a dandy intuitive description, the true relationship between the histories of the object and its clients is a "happened-before" relationship (as defined by Leslie Lamport in 1978 in Time, Clocks, and the Ordering of Events in a Distributed System).

Linearizability does not imply read/write operations, although those are possible. A linearizable object may provide any set of operations, and each of these operations appears to occur one at a time in the history of the object. These operations appear to occur at each client sometime between the client's invocation and the client's receipt of the response.

Many times, the word *consistency* is used to describe linearizable read/write operations over a set of objects. Read is defined to return the latest value seen in a write to the object. If Client A does a write followed by a read to a linearizable object, the value returned to the read invocation must be either the one previously written by Client A or the value of a write from another Client B processed by the object after Client A's write and before Client A's subsequent read. It is the interaction of the read operation with the history of the object (at the time of the read) that poses challenges under partitioning.

Again, linearizability defines a relationship between the

partial order (or history) of the object and the partial order (or history) of each client. It supports any operation that can be invoked in a single request to one object.

CONVERGENCE VERSUS LINEARIZABILITY

Convergence speaks to what happens when you stop tormenting multiple disjoint replicas of the same object and bring the replicas together. Specifically, if you have many replicas of an object and *each receives different subsets of the updates*, a convergent object will merge with its replicas and yield the same object. This is independent of the order the updates were applied to the various replicas.

Linearizability is a property of the history of a single object and that history's relation to the histories of the clients sending operations to that single object. The linearizable object's crisp history must look like it lives in one location doing one operation at a time.

Linearizability and convergence are very similar in that they both refer to single objects (whatever that means). They are existentially different in that convergence assumes multiple replicas, but linearizability *must appear to be one replica with a linear history.* Below, I point out that *the appearance of one replica with a linear history can be spoofed by constraining the types of operations performed.*

A FEW MORE MEANINGS OF CONSISTENCY: THE "C" IN THE CAP THEOREM

CAP was famously posed as a conjecture in 2000 by Eric Brewer at the PODC (Principles of Distributed Computing)

symposium in his keynote called Towards Robust Distributed Systems. It was later proved in Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services and is now called the CAP theorem. The conjecture and theorem get their name from "Consistent reads, Availability, and Partition-tolerance: Pick two."

According to Brewer in CAP Twelve Years Later: How the Rules Have Changed (2012), consistency is defined as "CONSISTENCY (C) EQUIVALENT TO HAVING A SINGLE UP-TO-DATE COPY OF THE DATA." He further states:

CONSISTENCY (C). IN ACID, THE C MEANS THAT A TRANSACTION PRESERVES ALL THE DATABASE RULES, SUCH AS UNIQUE KEYS. IN CONTRAST, THE C IN CAP REFERS ONLY TO SINGLE-COPY CONSISTENCY, A STRICT SUBSET OF ACID CONSISTENCY. ACID CONSISTENCY ALSO CANNOT BE MAINTAINED ACROSS PARTITION. PARTITION-RECOVERY WILL NEED TO RESTORE ACID CONSISTENCY. MORE GENERALLY, MAINTAINING INVARIANTS DURING PARTITIONS MIGHT BE IMPOSSIBLE, THUS THE NEED FOR CAREFUL THOUGHT ABOUT WHICH OPERATIONS TO DISALLOW AND HOW TO RESTORE INVARIANTS DURING RECOVERY.

Then, we see:

ASPECTS OF THE CAP THEOREM ARE OFTEN MISUNDERSTOOD, PARTICULARLY THE SCOPE OF AVAILABILITY AND CONSISTENCY, WHICH CAN LEAD TO UNDESIRABLE RESULTS.

Followed by:

SCOPE OF CONSISTENCY REFLECTS THE IDEA THAT, WITHIN SOME BOUNDARY,

STATE IS CONSISTENT, BUT OUTSIDE THAT BOUNDARY ALL BETS ARE OFF. FOR EXAMPLE, WITHIN A PRIMARY PARTITION, IT IS POSSIBLE TO ENSURE COMPLETE CONSISTENCY AND AVAILABILITY, WHILE OUTSIDE THE PARTITION, SERVICE IS NOT AVAILABLE.

Confused, I dug deeper. In the 2012 article, Brewer indicated that the CAP conjecture was first discussed in 1999 in Harvest, Yield, and Scalable Tolerant Systems, by Armando Fox and Brewer. How do they define consistency in that paper?

IN THIS DISCUSSION, STRONG CONSISTENCY MEANS SINGLE-COPY ACID CONSISTENCY; BY ASSUMPTION A STRONGLY CONSISTENT SYSTEM PROVIDES THE ABILITY TO PERFORM UPDATES, OTHERWISE DISCUSSING CONSISTENCY IS IRRELEVANT.

What about in Brewer's keynote at PODC 2000? Here the definition of C as strong consistency falls under the category of ACID transactions, versus weak consistency under the category of BASE (basically available soft-state eventually consistent).

Finally, I checked out the proof of the theorem in Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. Here, I found a concise definition for consistency, even if it was different from any others I found in this spelunking of CAP:

THE MOST NATURAL WAY OF FORMALIZING THE IDEA OF A CONSISTENT SERVICE IS AS AN ATOMIC DATA OBJECT. ATOMIC, OR LINEARIZABLE, CONSISTENCY IS THE CONDITION EXPECTED BY MOST WEB SERVICES TODAY. UNDER THIS CONSISTENCY GUARANTEE, THERE MUST EXIST A TOTAL ORDER

ON ALL OPERATIONS SUCH THAT EACH OPERATION LOOKS AS IF IT WERE COMPLETED AT A SINGLE INSTANT. THIS IS EQUIVALENT TO REQUIRING REQUESTS OF THE DISTRIBUTED SHARED MEMORY TO ACT AS IF THEY WERE EXECUTING ON A SINGLE NODE, RESPONDING TO OPERATIONS ONE AT A TIME. THIS IS THE CONSISTENCY GUARANTEE THAT GENERALLY PROVIDES THE EASIEST MODEL FOR USERS TO UNDERSTAND, AND IS MOST CONVENIENT FOR THOSE ATTEMPTING TO DESIGN A CLIENT APPLICATION THAT USES THE DISTRIBUTED SERVICE.

OK, cool! This is linearizability for operations against a single object, as so crisply described by Herlihy and Wing. Reading further, I found the theorem they proved.

THEOREM 1. IT IS IMPOSSIBLE IN THE ASYNCHRONOUS NETWORK MODEL TO IMPLEMENT A READ/WRITE DATA OBJECT THAT GUARANTEES THE FOLLOWING PROPERTIES:

- AVAILABILITY
- ATOMIC CONSISTENCY

IN ALL FAIR EXECUTIONS (INCLUDING THOSE IN WHICH MESSAGES ARE LOST).

Studying this paper, I don't believe they've proved CAP for *atomic* or *linearizable consistency* unless *also constraining the operations to be read and write*. So, their proof is not applicable to the consistency model they cited within the very same paper!

It's an interesting proof but it only applies to consistent reads combined with atomic or linearizable consistency. This is a subset of the broader atomic or linearizable consistency.

CAP IS REALLY CrAP!

Consistent reads, Availability, and Partition-tolerance: Pick two!

The exact meaning of consistency in the conjecture is not clear. Only consistent reads are proven in the theorem!

Neither the database community nor the distributed systems community seem to have a consistent consistency. I am completely comfortable saying that consistent reads can only be achieved if you are willing to sacrifice either availability or partition tolerance. That clarification of the CAP conjecture and proof works for me!

CONFLUENCE: DEALING WITH COMPONENT INPUT AND OUTPUT

Feeling like I'd been *con*-ned, I decided to look at *confluence*, a property of the inputs and outputs of components.

Alvaro's dissertation defines confluence as follows:

> We call a dataflow component confluent if it produces the same set of outputs for all orderings of its inputs. At any time, the output of a confluent component (and any redundant copies of that component) is a subset of the unique, "final" output.

Alvaro continues to contrast convergence and confluence:

> Convergence is a local guarantee about component state; by contrast, confluence provides guarantees about component outputs, which (because they can become the inputs to downstream components) compose into global guarantees about dataflows.

In summary:

- **Convergence** is a definition based on eventual state.
- **Confluence** is a definition based on behavior and program outputs.

Thus, given a set of inputs to a confluent component, you get a set of outputs from that component. No matter what legal subset of the inputs you see, you will never retract any output emitted.

Confluence is written about both in Alvaro's dissertation and in a recent paper by Alvaro and Joe Hellerstein: Keeping CALM: When Distributed Consistency Is Easy. In that paper, they say:

> Unlike traditional memory consistency properties such as linearizability, confluence makes no requirements or promises regarding notions of recency (for example, a read is not guaranteed to return the result of the latest write request issued) or ordering of operations (for example, writes are not guaranteed to be applied in the same order at all replicas).
>
> Nevertheless, if an application is confluent, we know that any such anomalies at the memory or storage level *do not affect the application outcomes.*

In other words, confluence is NOT about memory reads and writes.

ACHIEVING MONOTONICITY BY SQUINTING AT THE DETAILS

So, confluence is a property of inputs to and outputs from a function. It means that, for a given set of inputs, you will

create only those outputs that are never retracted when you get new inputs. Basically, you never change your mind about an output.

Monotonicity means you move only forward, never backward. In other words, you never need to retract an output once you've said it.

CALM stands for consistency as logical monotonicity. The logical part of this means that you can squint at the inputs and the outputs in such a way that you have monotonicity. Can you omit enough details that you don't need to back away from any output messages?

If monotonicity means *always move forward*, how do you think about what moving forward means? At one level, it's related to equivalence (or fungibility) of the outputs seen from a confluent function. If you have a high-enough view of the operation and can see many answers as just the same, there's not much need to back up and retract an output.

Another way of saying this is that two confluent replicas might give two different answers, but when they come together, they can be viewed as one answer. If I'm in charge of picking an adequate hotel for a night's stay, just about anything will be fine. If my wife is selecting a hotel, there will be many additional criteria. My hotel-picking function is sometimes going to be confluent when my wife's is not. The notion of *the same output* depends on what's meant by *the same*.

Confluence speaks about the inputs and outputs from a function. That's the view from the outside. From the inside, we frequently think about side effects. In Side Effects, Front and Center!, I point out that side effects

are subjective. One person (or subsystem's) side effect is another's mainstream business purpose. Hence, interpreting monotonicity seems to depend on what's important to the observer.

SEALING OF A CONFLUENT STREAM

In their CALM paper, Hellerstein and Alvaro point out that confluent functions can answer questions such as, "Does this exist?" but have a hard time answering questions such as, "Does this NOT exist?". This is because the confluent function may still get later knowledge, and that new knowledge may add the thing that so far does not exist. Hence, you'd better not say it doesn't exist until you know you won't be getting new facts as new inputs.

There is a concept called *sealing of a confluent stream*. Sealing is a way to say no more confluent input operations will arrive. When you seal a confluent stream, you CAN get an output that says: "I can attest that this does NOT exist." (See Alvaro's dissertation.) Sealing of a confluent stream is funky. By definition, a seal operation is ordered and, hence, not a part of the confluent set of inputs (because the order of the seal matters).

You can meaningfully seal an individual replica of a confluent function and say there will be no more inputs to that replica. You can meaningfully seal a bounded set of replicas of a function if you know the bounded set, track them all down, and seal them. If you don't precisely know the set of replicas, you can't ensure all their inputs are sealed. When that happens, you can't answer "does not exist" questions about the set's outputs.

Confluence allows for composition of arbitrary sets of

confluent functions. If you want to think about "does not exist" questions, you'd better be able to control the bounds on this set of functions.

CON-FUSION: ARE CONFLUENCE AND CONVERGENCE THE SAME?

For a database person, I am surprisingly annoyed when folks see data as the be-all and end-all. It's common to think of data as stuff supporting only reads and writes. The focus should be on the collection of defined operations on a function and its internal state. Reads and writes are just two types of operations (and particularly wretched ones at that).

Confluence is about inputs and outputs to a function. A function is confluent if the outputs it produces do not change when the order of the inputs changes.

Convergence is about the state of a replica when it is merged with other replicas. But what does state mean? From my perspective, the state of a replica is defined by the outputs it will produce in response to future inputs.

Convergent replicas are allowed to wander around, accept inputs, and generate outputs. When seeing another member of the family, they do a Vulcan mind meld and become one thing. New outputs in the output set are dependent on the inputs received and the outputs emitted by ANY of the replicas. The output set for replicas of a convergent object grows monotonically without regard to the order of the inputs seen at any of the replicas. You can infer this based on the definition of convergence. Shapiro et al. say:

STRONG CONVERGENCE: CORRECT REPLICAS THAT HAVE DELIVERED THE SAME UPDATES HAVE EQUIVALENT STATE.

To me, equivalent state means that future outputs will be based on future inputs. So, the merged replicas of a convergent object will emit predictable outputs based on future inputs without regard to the order in which their previous inputs were seen.

Confluence is a way of describing the externally viewed outputs of a set of convergent replicas. Convergence speaks to the expected behavior of a set of confluent functions.

Of course, this is just a different way of talking about CRDTs as defined by Shapiro et al.

- CvRDTs (convergent replicated data types) are state based. Merging two or more replicas in any order gives equivalent state.
- CmRDTs (commutative replicated data types) are operation based. Any replicas seeing the same set of inputs without regard to their order will be equivalent.

Since each convergent replica saw its own set of inputs, the merged replicas will have seen the union of the inputs seen by the replicas before merging. If this is equivalent, their future outputs will be the same. The proof that CvRDTs are equivalent to CmRDTS is provided in Shapiro et al.

LINEARIZABILITY IS IN THE EYE OF THE BEHOLDER

Recall that linearizability defines a relationship between the partial order (or history) of operations on an object and the partial order (or history) of operations as seen at each client.

For some operations, linearizability can *appear* to be the same as confluence, at least from the client's perspective. If subsequent responses (outputs) need not be retracted, the responses (or outputs) appear linearizable for objects with confluent behavior. The order of the outputs was just fine no matter what other clients may have seen.

To make a confluent object appear to be linearizable, the "linear" order seen by the object itself must be relaxed. While this object looks linearizable from the outside (because of confluent operations), the operations have a DAG (directed acyclic graph) for its history. Clients on the outside won't be able to tell that things didn't happen in a linear order. This works because the confluent operations tolerate reordering. Linear from the outside can be a DAG on the inside if the operations are confluent.

Confluent functions and convergent objects offer behavior that appears linearizable to their clients. Each client sends an input that, in turn, generates an output. The output response is seen by the client after the input request is performed.

Loosening up how linearizable behavior is interpreted provides essential flexibility over both time (e.g., long-running work) and space (e.g., replication), assuming the operations are confluent and they never need to retract any output they've emitted. Never having a reason to say you're sorry is an excellent strategy in distributed computing AND a happy married life!

INCONSISTENT AND CONSISTENT "CONSISTENCY"

There seem to be at least three popular uses of the word *consistency*:

- **Database consistency.** This is a conflation of complete transactions with some unstated enforcement of unstated rules. Because the set of updates within a transaction must be bounded by the application in cahoots with the upper part of the database, the application and upper database can enforce some rules that the transaction system doesn't understand. I think of this as complete from the transaction's perspective.
- **Eventual consistency of replicated objects (convergent).** When all the replicas of an object are merged, they all have the same value.
- **Consistency as linearizable read/write operations.** Updates to EACH object in a distributed system must appear as if they occurred within a single history as seen by the object, AND the invocations and responses from the client must form a well-defined happened-before history at each client.

None of these popular uses makes any sense to me. But there ARE usages of consistency (with an adjective in front of them) that I crisply understand:

- **Strict consistency.** This is exactly the same as linearizable read/write operations as used by Gilbert and Lynch in their proof of the CAP conjecture. Usually, it is interpreted as linear order for possibly nonconfluent operations (e.g., read and write).
- **Sequential consistency.** Defined by Lamport in How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs (1979), sequential consistency is weaker than strict consistency. Each client's writes to a variable appear in order from the perspective of the writer. Reads by a client do not guarantee visibility to other

clients' concurrent writes. I understand this definition.

➡ **Causal consistency.** One of my favorite forms of consistency, causal consistency is a weakened form of sequential consistency. It categorizes operations based on their causal dependence.

Suppose I receive Operation Y performed on Object X by Client A. Causal consistency says that before I receive Operation Y performed by Client A, I will first receive all operations performed on Object X that were visible to Client A at the time it requested Operation Y.

This can be done by carefully tracking the delivery of operations to Client A and squirting these operations around the network as a bundle capturing the order of delivery as seen by Client A. As I set out to receive the next operation on Object X, the plumbing delivering these operations to me FIRST ensures I see all the operations that were received by Client A before it requested Operation Y.

The ordering of operations as seen by each client shows the causal order. This is tractably implementable in a distributed system. It is also incredibly useful in occasionally connected systems.

EXTERNAL CONSISTENCY

This paper doesn't delve into the phrase "External Consistency". Near as I can tell, that means you should provide the same behavior *outside a distributed cluster* and you would *outside a single node*. Since this is tangled up in what was meant by the word *consistency*, I find this confusing and imprecise, too, The "external" part

seems OK. The "consistency" part is frequently used inconsistently, though.

CONCLUSION

This article is about nomenclature and HOW to talk about stuff. I am going to work hard to make my usage of words crisper and more concise in the future.

- **Convergence** is a property of an object. It means you have a *merge algorithm* that allows you to smash divergent replicas together and get the identical result.

  A convergent object's merge algorithm must be associative, commutative, and idempotent. If you merge any two replicas in any order, as long as you've captured the same set of changes on any replica in any order, you will get the same result. This makes it easy to distribute the work across space and time. Then, you can have ACID 2.0 (associative, commutative, idempotent, and distributed) changes to the object as described in Building on Quicksand.
- **Eventual convergence** is the same as plain-old convergence but sounds cooler and may help you be successful when meeting new people in a bar.
- **Confluence** is a property of a dataflow component. Alvaro says:

> We call a dataflow component *confluent* if it produces the same set of outputs for all orderings of its inputs. At any time, the output of a confluent component (and any redundant copies of that component) is a subset of the unique, "final" output.

Confluent dataflow components compose. In my opinion, confluence is the new and exciting space for distributed-systems research.

- **Consistency:** The word *consistent* is not consistent. I pretty much think it means nothing useful unless preceded by *strong*, *sequential*, or *causal*.
- **Eventual consistency:** *Eventual consistency* means nothing both now AND later. It does, however, confuse the heck out of a lot of people.

So, except for *strict consistency*, *sequential consistently*, or *causal consistency*, I am going to consistently avoid using the word *consistent* in the future. Most of the time.

Open questions

- When should a designer choose an operation-centric approach to replication (e.g., confluence) and an object-centric approach to replication (e.g., convergence)?
- When, where, and how is causal consistency an amazing gift to any system's designer?
- Consensus is a mechanism to allow an agreed-upon new value for an object. Is that simply a distributed write operation? Why do we focus on consensus as a form of writing a new value? Is that the same as strong consistency?
- How should you react on a first date when your date talks about *eventual consistency*? Is that a more difficult discussion than talking about politics?
- Is *linearizability* truly in the eye of the beholder? Can we make robust distributed systems by not caring about a lack of order that is hidden by another system? Is "can't tell if it's linear" the same as "truly linear"?

Pat Helland *has been implementing transaction systems, databases, application platforms, distributed systems, fault-tolerant systems, and messaging systems since 1978. For recreation, he occasionally writes technical papers. He works at Salesforce. His blog is at pathelland.substack.com.*

CONTENTS