DOI:10.1145/3369736

In distributed systems theory, CALM presents a result that delineates the frontier of the possible.

BY JOSEPH M. HELLERSTEIN AND PETER ALVARO

Keeping CALM: When Distributed Consistency Is Easy

DISTRIBUTED SYSTEMS ARE tricky. Multiple unreliable machines are running in parallel, sending messages to each other across network links with arbitrary delays. How can we be confident these systems do what we want despite this chaos?

This issue should concern us because nearly all of the software we use today is part of a distributed system. Apps on your phone participate with hosted services in the cloud; together they form a distributed system. Hosted services themselves are massively distributed systems, often running on machines spread across the globe. Big data systems and large-scale databases are distributed across many machines. Most scientific computing and machine learning systems work in parallel across multiple processors. Even legacy desktop operating systems and applications like spreadsheets and word processors are tightly integrated with distributed backend services.

The challenge of building correct distributed systems is increasingly urgent, but it is not new. One traditional answer has been to reduce this complexity with *memory consistency* guarantees—assurances that accesses to memory (heap variables, database keys, and so on) occur in a controlled fashion. However, the mechanisms used to enforce these guarantees—*coordination protocols*—are often criticized as barriers to high performance, scale, and availability of distributed systems.

The high cost of coordination. Coordination protocols enable autonomous, loosely coupled machines to jointly decide how to control basic behaviors, including the order of access to shared memory. These protocols are among the most clever and widely cited ideas in distributed computing. Some well-known techniques include the Paxos[33] and Two-Phase Commit (2PC)[25,34] protocols, and global barriers underly-

key insights

- **Coordination is often a limiting factor in system performance. While sometimes necessary for consistent outcomes, coordination often needlessly stands in the way of interactivity, scalability, and availability.**
- **Distributed systems deserve a computability theory: When is coordination required for consistency, and when can it be avoided?**
- **The CALM Theorem shows that monotonicity is the answer to this question. Monotonic problems have consistent, coordination-free implementations; non-monotonic problems require coordination for consistency.**
- **The CALM Theorem emerges by shifting the definition of consistency to one of deterministic program outcomes rather than ordered histories of events. CALM thinking is also constructive: it informs the design of new distributed programming languages, program analysis tools, and application design patterns.**

IMAGE BY DABOOST

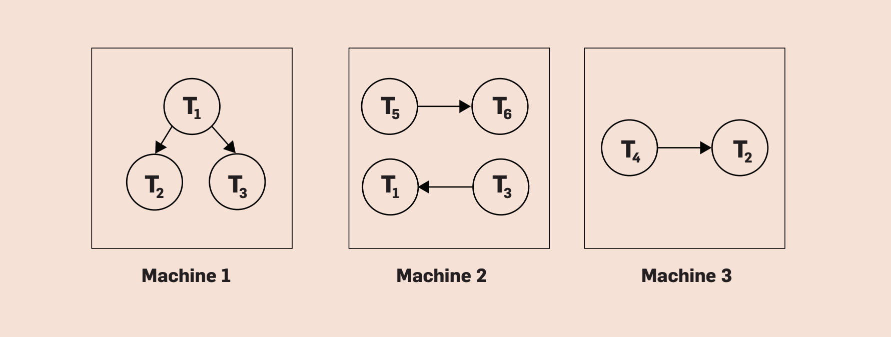
ing computational models like Bulk Synchronous Parallel computing.[40]

Unfortunately, the expense of coordination protocols can make them "forbidden fruit" for programmers. James Hamilton from Amazon Web Services made this point forcefully, using the phrase "consistency mechanisms" where we use coordination:

"The first principle of successful scalability is to batter the consistency mechanisms down to a minimum, move them off the critical path, hide them in a rarely visited corner of the system, and then make it as hard as possible for application developers to get permission to use them."[26]

The issue is not that coordination is tricky to implement, though that is true. The main problem is that coordination can dramatically slow down computation or stop it altogether. Some modern global-scale systems utilize coordination protocols; the Google Spanner transactional database[18] is a notable example that uses both Paxos and 2PC. However, these protocols suffer from high latencies, on the order of 10ms–100ms. Global-scale systems that rely on these protocols are not meant to be used in the fast path of an application. Coordination latency problems translate to the micro scale as well. Recent

Figure 1. A distributed waits-for graph with replicated nodes and partitioned edges. There is a cycle that spans Machines 1 and 2 ($\{T_1, T_3\}$).



work showed that state-of-the-art multiprocessor key-value stores can spend 90% of their time waiting for coordination; a coordination-free implementation called Anna achieves over two orders of magnitude speedup by eliminating that coordination.[43] Can we avoid coordination more generally, as Hamilton recommends? When?

The bigger picture: Program consistency. The general question of when coordination is necessary to achieve consistency was not addressed until relatively recently. Traditional work on consistency focused on properties like linearizability[30] and conflict serializability,[20] which ensure memory consistency by constraining the order of conflicting memory accesses. This tradition obscured the underlying question of whether coordination is required for the consistency of a particular program's outcomes. To attack the problem holistically we need to move up the stack, setting aside low-level details in favor of program semantics.

Traffic intersections provide a useful analogy from the real world. To avoid accidents at busy intersections, we often install stop lights to coordinate traffic across two intersecting roads. However, coordination is not a necessary evil in this scenario: we can also prevent accidents by building an overpass or tunnel for one of the roads. The "traffic intersection problem" is an example with a coordination-free solution. Importantly, the solution is not found by cleverly controlling the order of access to the critical section where the roads overlap on a map. The solution involves engineering the roads to avoid the need for coordination entirely.

For the traffic intersection problem, it turns out there is a solution that avoided coordination altogether. Not all problems have such a solution. For any given computational problem, how do we know if it has a coordination free solution, or if it requires coordination for consistency? To sharpen our intuition, we consider two nearly identical problems from the distributed systems canon. Both involve graph reachability, but one is coordination free and the other is not.

Distributed deadlock detection. Distributed databases identify cycles in a distributed graph in order to detect and remediate deadlocks. In a traditional database system, a transaction T_i may be waiting for a lock held by another transaction T_j, which may in turn be waiting for a second lock held by T_i. The deadlock detector identifies such "waits-for" cycles by analyzing a directed graph in which nodes represent transactions, and edges represent one transaction waiting for another on a lock queue. Deadlock is a stable property: the transactions on a waits-for cycle cannot make progress, so all edges on the cycle persist indefinitely.

In a distributed database, a "local" (single-machine) view of the waits-for graph contains only a subset of the edges in the global waits-for graph. In this scenario, how do local deadlock detectors work together to identify global deadlocks?

Figure 1 shows a waits-for cycle that spans multiple machines. To identify such distributed deadlocks, each machine exchanges copies of its edges with other machines to accumulate more information about the global graph. Any time a machine observes a cycle in the information it has received so far, it can declare a deadlock among the transactions on that cycle.

We might be concerned about transient errors due to delayed or reordered messages in this distributed computation. Do local detectors have to coordinate with other machines to be sure of a deadlock they have observed? In this case, no coordination is required. To see this, note that once we know a cycle exists in a graph, learning about a new edge can never make the cycle go away. For example, once Machine 1 and Machine 2 jointly identify a deadlock between T_1 and T_3, new information from Machine 3 will not change that fact. Additional facts can only result in additional cycles being detected: the output at each machine grows monotonically with the input. Finally, if all the edges are eventually shared across all machines, the machines will agree upon the outcome, which is based on the full graph.

Distributed garbage collection. Garbage collectors in distributed systems must identify unreachable objects in a distributed graph of memory references. Garbage collection works by identifying graph components that are disconnected from the "root" of a system runtime. The property of being "garbage" is also stable: once a graph component's connection to the root is removed, the objects in that component will not be re-referenced.

In a distributed system, references to objects can span machines. A local view of the reference graph contains only a subset of the edges in the global graph. How can multiple local garbage collectors work together to identify objects that are truly unreachable?

Note that a machine may have a local object and no knowledge whether the object is connected to the root; Machine 3 and object O_4 in Figure 2 form an example. Yet there still may be a path to that object from the root that consists of edges distributed across other machines. Hence, each machine exchanges copies of edges with other machines to accumulate more information about the graph.

As before, we might be concerned about errors due to message delays or reordering. Can local collectors autonomously declare and deallocate garbage? Here, the answer is different: coordination is indeed required! To see this, note that a decision based on incomplete information—for example, Machine 3 deciding that object O_4 is unreachable in Figure

2—can be invalidated by the subsequent arrival of new information that demonstrates reachability (for example, the edges Root $\rightarrow O_1$, $O_1 \rightarrow O_3$, $O_3 \rightarrow O_4$). The output does not grow monotonically with the input: provisional "answers" may need to be retracted. To avoid this, a machine must ensure it has heard *everything there is to hear* before it declares an object unreachable. The only way to know it has heard everything is to coordinate with all the other machines—even machines that have no reference edges to report—to establish that fact. As we will discuss, a hallmark of coordination is this requirement to communicate even in the absence of data dependencies.

The crux of consistency: Monotonicity. These examples bring us back to our fundamental question, which applies to any concurrent computing framework.

QUESTION: *We say that a computational problem is coordination-free if there exists a distributed implementation (that is, a program solving the problem) that computes a consistent output without using coordination. What is the family of coordination-free problems, and what problems lie outside that family?*

There is a difference between an incidental use of coordination and an intrinsic need for coordination: the former is the result of an implementation choice; the latter is a property of a computational problem. Hence our Question is one of computability, like P vs. NP or Decidability. It asks what is (im)possible for a clever programmer to achieve.

Note that the question assumes some definition of "consistency." Where traditional work focused narrowly on memory consistency (that is, reads and writes produce agreed-upon values), we want to focus on program consistency: does the implementation produce the *outcome* we expect (for example, deadlocks detected, garbage collected), despite any race conditions across messages and computation that might arise?

Our examples provide clues for answering our question. Both examples accumulate a *set* of directed edges E, and depend on reachability predicates—that is, tests for pairs of nodes in the transitive closure $E*$. But they differ in one key aspect. A node participates in a deadlock if there exists a path to itself in $E*$: $\{n \mid \exists(n,n) \in E*\}$. A node n is garbage if there does *not* exist a path from root to n: $\{n \mid \neg\exists(root,n) \in E*\}$.

Logical predicates clarify the distinction between the examples. For deadlock detection's existential predicate, the set of satisfying paths that exist is *monotonic* in the information received:

DEFINITION 1. *A problem P is monotonic if for any input sets S, T where $S \subseteq T$, $P(S) \subseteq P(T)$.*

By contrast, the set of satisfying paths that do *not* exist in the garbage collection example is non-monotonic: conclusions made on partial information about E may not hold in eventuality as counterexamples appear to revoke prior beliefs about what "did not exist" previously.

Monotonicity is the key property underlying the need for coordination to establish consistency, as captured in the CALM Theorem:

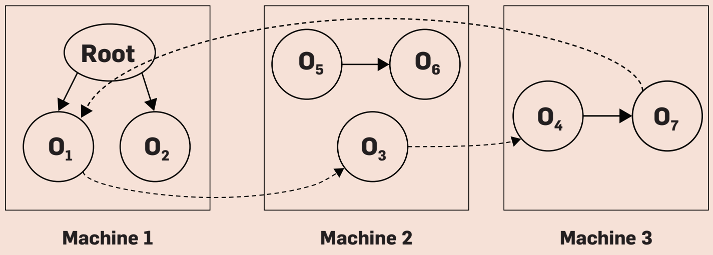
THEOREM 1. Consistency As Logical Monotonicity (CALM). *A problem has a consistent, coordination-free distributed implementation if and only if it is monotonic.*

Intuitively, monotonic problems are "safe" in the face of missing information and can proceed without coordination. Non-monotonic problems, by contrast, must be concerned that truth of a property *could change in the face of new information*. Therefore, they cannot proceed until they know all information has arrived, requiring them to coordinate.

Additionally, because they "change their mind," non-monotonic problems are order-sensitive: the order in which they receive information determines how they toggle state back and forth, which can in turn determine their final state (as we will see in the example of shopping carts). By contrast, monotonic problems simply accumulate beliefs; their output depends only on the content of their input, not the order in which it arrives.

Our discussion so far has remained at the level of intuition. The next section provides a sketch of a proof of the CALM Theorem, including further discussion of definitions for consistency and coordination. The proof uses logic formalisms from database theory and demonstrates the benefits of bringing the theory of databases (ACM PODS) and distributed systems (ACM PODC) closer together. Problems can be defined as families of declarative queries over relations (sets of records) running across multiple machines. As in our examples, the monotonicity of these queries can often be checked statically via their syntax: for example, $\exists(n,n) \in E*$ is monotonic, but $\neg\exists(root,n) \in E*$ is non-monotonic, as evidenced by the use of the negated existential quantifier $\neg\exists$ ("not exists"). Readers seeking a complete proof are directed to the papers by Ameloot, et al.[8,9]

Figure 2. A distributed object reference graph with remote references (dotted arrows). The fact that object O_3 is reachable from Root can be established without any information from Machine 3. Objects O_5 and O_6 are garbage, which can only be established by knowing the entire graph.



CALM: A Proof Sketch

Our first challenge in formalizing the CALM Theorem is to define program consistency in a manner that allows us to reason about program outcomes, rather than mutations to storage. Having done that, we can move on to a discussion of consistent computability with and without coordination.

Program consistency: Confluence. Distributed systems introduce significant non-determinism to our programs. Sources of non-determinism include unsynchronized parallelism, unreliable components, and networks with unpredictable delays. As a result, a distributed program can exhibit a large space of possible behaviors on a given input.

While we may not control all the behavior of a distributed program, our true concern is with its *observable* behavior: the program outcomes. To this end, we want to assess how distributed nondeterminism affects program outcomes. A practical consistency question is this: "Does my program produce deterministic outcomes despite nondeterminism in the runtime system?"

This is a question of program *confluence*. In the context of nondeterministic message delivery, an operation on a single machine is confluent if it produces the same set of output responses for any non-deterministic ordering and batching of a set of input requests. In this vein, a confluent single-machine operation can be viewed as *a deterministic function from sets to sets*, abstracting away the nondeterministic order in which its inputs happen to appear in a particular run of a distributed system. Confluent operations compose: if the output set of one confluent operation is consumed by another, the resulting composite operation is confluent. Hence, confluence can be applied to individual operations, components in a dataflow, or even entire distributed programs.[2] If we restrict ourselves to building programs by composing confluent operations, our programs are confluent by construction, despite orderings of messages or execution races within and across components.

Unlike traditional memory consistency properties such as linearizability,[30] 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*.

Confluence is a powerful yet permissive correctness criterion for distributed applications. It rules out application-level inconsistency due to races and non-deterministic delivery, while permitting nondeterministic ordering and timings of lower-level operations that may be costly (or sometimes impossible) to prevent in practice.

Confluent shopping carts. To illustrate the utility of reasoning about confluence, we consider an example of a higher-level application. In their paper on the Dynamo key-value store,[19] researchers from Amazon describe a shopping cart application that achieves confluence without coordination. In their scenario, a client Web browser requests items to `add` and `delete` from an online shopping cart. For availability and performance, the state of the cart is tracked by a distributed set of server replicas, which may receive requests in different orders. In the Amazon implementation, no coordination is needed while shopping, yet all server replicas eventually agree on the same final state of the shopping cart. This is a prime example of the class of program that interests us: eventually consistent, even when implemented atop a non-deterministic distributed substrate that does no coordination.

Program consistency is possible in this case because the fundamental operations performed on the cart (for example, `add`) commute, so long as the contents of the cart are represented as a set and the internal ordering of its elements is ignored. If two replicas learn along the way they disagree about the contents of the cart, their differing views can be merged simply by issuing a logical "query" that returns the union of their respective sets.

Unfortunately, if we allow a `delete` operation in addition to `add`, the set neither monotonically grows nor shrinks, which causes consistency trouble. If instructions to `add` item *I* and `delete` item *I* arrive in different orders at different machines, the machines may disagree on whether *I* should be in the cart. As mentioned earlier, this is reflected in the way the existence of *I* toggles on the nodes. On one machine the presence of *I* might start in the state `not-exists`, but a series of messages `<add(I); delete(I)>` will cause the state to toggle to `exists` and then to `not-exists`; on another machine the messages might arrive in the order `<delete(I);add(I)>`, causing *I*'s state to transition from `not-exists` to `not-exists` to `exists`. Even after the two machines have each received all the messages, they disagree on the final outcome. As a traditional approach to avoid such "race conditions," we might bracket every non-monotonic `delete` operation with a global coordination to agree on which `add` requests come before it. Can we do better?

As a creative application-level use of monotonicity, a common technique is for `deletes` to be handled separately from `adds` via two separate monotonically growing sets: `Added` items and `Deleted` items.[19,39] The `Added` and `Deleted` sets are both insert-only, and insertions across the two commute. The final cart contents can be determined by unioning up the `Added` sets across nodes, as well as unioning up the `Deleted` sets across nodes, and computing the set-difference of the results. This would seem to solve our problem: it removes the need to coordinate while shopping—that is, while issuing `add` and `delete` requests to the cart.

Unfortunately, neither the `add` nor `delete` operation commutes with `checkout`—if a `checkout` message arrives before some insertions into either the `Added` or `Deleted` sets, those insertions will be lost. In a replicated setting like Dynamo's, the order of `checkout` with respect to other messages needs to be globally controlled, or it could lead to different decisions about what was actually in the cart when the `checkout` request was processed.

Even if we stop here, our lens provided a win: monotonicity allows *shopping* to be coordination free, even though *checkout* still requires coordination.

This design evolution illustrates the technical focus we seek to clarify. Rather than micro-optimize protocols like Paxos or 2PC to protect race conditions in procedural code, modern distributed systems creativity often involves minimizing the use of such protocols.

A sketch of the proof. The CALM conjecture was presented in a keynote talk at PODS 2010 and written up shortly thereafter alongside a number of corollaries.[28] In a subsequent series of papers,[8,9,44] Ameloot and colleagues presented a formalization and proof of the CALM Theorem, which remains the reference formalism at this time. Here, we briefly review the structure of the argument from Ameloot et al.

Proofs of distributed computability require some formal model of distributed computation: a notion of disparate machines each supporting some local model of computation, data partitioned across the machines, and an ability for the machines to communicate over time. To capture the notion of a distributed system composed out of monotonic (or non-monotonic) logic, Ameloot uses the formalism of a *relational transducer*[1] running on each machine in a network. This formalism matches our use of logical expressions in our graph examples; it also matches the design pattern of sets of items with additions, deletions and queries in Dynamo.

Simply put, a relational transducer is an event-driven server with a relational database as its memory and programs written declaratively as queries. Each transducer runs a sequential event loop as follows:

1. **Ingest and apply** an unordered batch of requests to insert and delete records in local relations. Requests may come from other machines or a distinguished input relation.
2. **Query** the (now-updated) local relations to compute batches of records that should be sent somewhere (possibly locally) for handling in future.
3. **Send** the results of the query phase to relevant machines in the network as requests to be handled. Results sent locally are ingested in the very next iteration of the event loop. Results can also be "sent" to a distinguished output.

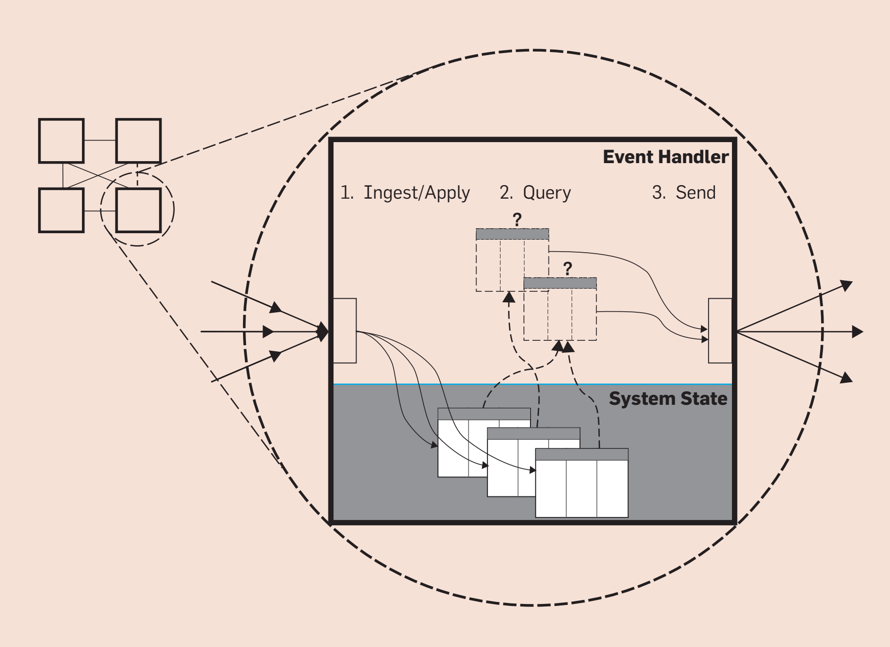
In this computational model, the state at each machine is represented via sets of records (that is, relations), and messages are represented via records that are inserted into or deleted from the relations at the receiving machine. Computation at each machine is specified via declarative (logic) queries over the current local relations at each iteration of the event loop.

The next challenge is to define monotonicity carefully. The query languages used by Ameloot are variants of Datalog, but we remind the reader that classical database query languages—relational calculus and algebra, SQL, Datalog—are all based on first-order logic. In all of these languages, including first-order logic, most common expressions are monotonic; the syntax reveals the potentially nonmonotonic expressions. Hence "programs expressed in monotonic logic" are easy to define and identify: they are the transducer networks in which every machine's queries use only monotonic syntax. For instance, in the relational algebra, we can allow each machine to employ selection, projection, intersection, join and transitive closure (the monotonic operators of relational algebra), but not set difference (the sole non-monotonic operator). If we use relational logic, we disallow the use of universal quantifiers (∀) and their negation-centric equivalent (¬ ∃)—precisely the construct that tripped us up in the garbage collection example noted earlier. If we model our programs with mutable relations, insertions are allowable, but in general updates and deletions are not.[5,35] These informal descriptions elide a number of clever exceptions to these rules that still achieve semantic monotonicity despite syntactic non-monotonicity,[8,17] but they give a sense of how the formalism is defined.

Now that we have a formal execution model (relational transducers), a definition of consistency (confluence), and a definition of monotonic programs, we are prepared to prove a version of the CALM Theorem. The forward "if" direction of the CALM Theorem is quite straightforward and similar to our previous discussion: in a monotonic relational transducer network, it is easy to show that any machine will eventually Ingest and Send a deterministic set of messages and generate a deterministic output. As a side benefit, at any time during execution, the messages output by any machine form a valid subset of the final output.

The reverse "only if" direction is quite a bit trickier, as it requires ruling out any possible scheme for avoiding coordination. The first challenge is to formally separate the communication needed to construct outputs (essentially, dataflow messages) from other communication (coordination messages). Intuitively, dataflow messages are those that arise to assemble data whose components are not co-located. To isolate the coordination messages, Ameloot et al. consider all possible ways to partition data across machines in the network at program start. From each of these starting points, a messaging pattern is produced during execution of the program. We say that a program contains coordination if it requires messages to be sent

Figure 3. A simple four-machine relational transducer network with one machine's state and event loop shown in detail.



under *all possible partitionings*—including partitionings that co-locate all data at a single machine. A message that is sent in every partitioning is not related to dataflow; it is a coordination message. As an example, consider how a distributed garbage collector decides if a locally disconnected object O_g is garbage. Even if all the data is placed at a single machine, that machine needs to exchange messages with the other machines to check that they have no more additional edges—it needs to "coordinate," not just communicate data dependencies. The proof then proceeds to show that non-monotonic operations require this kind of coordination.

This brief description elides many interesting aspects of the original article. In addition to the connections established between monotonicity and coordination-freeness, connections are also made to other key distributed systems properties. One classic challenge is to achieve distributed agreement on network membership (represented by Ameloot et al. as the *All* relation). It turns out that not only are the monotonic problems precisely the coordination-free problems, they are also precisely those that do not require knowledge of network membership—they need not query *All*. A similar connection is shown with the property of a machine being aware of its own identity/address (querying the *Id* relation).

CALM Perspective on the State of the Art

The CALM Theorem describes what is and is not possible. But can we use it practically? In this section, we address the implications of CALM with respect to the state of the art in distributed systems practice. It turns out that many patterns for maintaining consistency follow directly from the theorem.

CAP and CALM: Going positive. Brewer's CAP Theorem[14] informally states that a system can exhibit only two out of the three following properties: Consistency, Availability, and Partition-tolerance. CAP is a negative result: it captures properties that cannot be achieved in general. But CAP only holds if we assume the system in question is required to execute arbitrary programs. It does not ask whether there are specific subclasses of programs that can enjoy all three properties! In a retrospective, Brewer reframes his discussion of CAP along these very lines:

[The original] "expression of CAP served its purpose, which was to open the minds of designers to a wider range of systems and trade-offs ... The modern CAP goal should be to maximize combinations of consistency and availability that make sense for the specific application."[14]

CALM is a positive result in this arena: it circumscribes the class of problems for which all three of the CAP properties can indeed be achieved simultaneously. To see this, note the following:

OBSERVATION 1. *Coordination-freeness is equivalent to availability under partition.*

In the forward direction, a coordination-free program is by definition available under partition: all machines can proceed independently. When and if the partition heals, state merger is monotonic and consistent. In the reverse direction, a program that employs coordination will stall (become unavailable) during coordination protocols if the machines involved in the coordination span the partition.

In that frame, CALM asks and answers the underlying question of CAP: "Which problems can be consistently computed while remaining available under partition?" CALM does not contradict CAP. Instead, CALM approaches distributed consistency from a wider frame of reference:

1. First, CAP is a negative result over the space of *all problems*: CALM confirms this coarse result, but delineates at a finer grain the negative and positive cases. Using confluence as the definition of consistency, CALM shows that monotone problems can in fact satisfy all three of the CAP properties at once; non-monotone problems are the ones that cannot.
2. The key insight in CALM is to focus on consistency from the viewpoint of *program outcomes* rather than the traditional *ordered histories* of conflicting actions—typically storage mutation. The emphasis on the problem being computed shifts focus from imperative implementation to declarative specification of outputs; that allows us to ask questions about what computations are possible.

The latter point is what motivated our outcome-oriented definition of program consistency. Note that Gilbert and Lynch[23] choose to prove the CAP Theorem using a rubric of linearizability (that is, agreement on a total order of conflicting actions), while Ameloot's CALM Theorem proofs choose confluence (agreement on program outcomes.) We note that confluence is both more permissive and closer to user-observable properties. CALM provides the formal framework for the widespread intuition that we can indeed "work around CAP"—for monotone problems—even if we violate traditional systems-level notions of storage consistency.

Distributed design patterns. Our shift of focus from mutable storage to program semantics has implications beyond proofs. It also informs the design of better programming paradigms for distributed computing.

Traditional programming languages model the world as a collection of named variables whose values change over time. Bare assignment[10] is a nonmonotonic programming construct: outputs based on a prefix of assignments may have to be retracted when new assignments come in. Similarly, assignments make final program states dependent upon the arrival order of inputs. This makes it extremely hard to take advantage of the CALM Theorem to analyze systems written in traditional imperative languages!

Functional programming has long promoted the use of *immutable* variables, which are constrained to take on only a single value during a computation. Viewed through the lens of CALM, an immutable variable is a simple monotonic pattern: it transitions from being undefined to holding its final value, and never goes back. Immutable variables generalize to immutable data structures; techniques such as deforestation[41] make programming with immutable trees, lists and graphs more practical.

Monotonic programming patterns are common in the design of distributed storage systems. We already discussed the Amazon shopping cart for Dynamo, which models cart state as two growing sets. A related pattern in storage systems is the use of *tombstones*: special data values that mark a data item as deleted. Instead of explicitly allowing deletion (a non-monotonic construct), tombstones mask immutable values with corresponding immutable tombstone values. Taken together, a data item with tombstone monotonically transitions from undefined, to a defined value, and ultimately to tombstoned.

Conflict-free replicated data types (CRDTs)[39] provide an object-oriented framework for monotonic programming patterns like tombstones, typically for use in the context of replicated state. A CRDT is an abstract data type whose possible internal states form a lattice and evolve monotonically according to the lattice's associated partial order, such as the partial order of set containment under $\subseteq$ or of integers under $\leq$. Two instances of a CRDT can be merged using the commutative, associative, idempotent join function from the associated internal lattice. Eventually, the states of two CRDT replicas that may have seen different inputs and orders can always be deterministically merged into a new final state that incorporates all the inputs seen by both.

The issue is not that coordination is tricky to implement, though that is true. The main problem is that coordination can dramatically slow down computation or stop it altogether.

CRDTs are an object-oriented lens on a long tradition of prior work that exploits commutativity to achieve determinism under concurrency. This goes back at least to long-running transactions,[15,22] continuing through recent work on the Linux kernel.[16] A problem with CRDTs is that their guarantees apply only to individual objects. The benefits of commutativity have been extended to composable libraries and languages, enabling programmers to reason about correctness of whole programs in languages like Bloom,[3] the LVish library for Haskell,[32] Lasp,[37] and Gallifrey.[38] We turn to an example of that idea next.

The Bloom programming language. One way to encourage good distributed design patterns is to use a language specifically centered around those patterns. Bloom is a programming language we designed in that vein; indeed, the CALM conjecture and Bloom language were developed together.[3]

The main goal of Bloom is to make distributed systems easier to reason about and program. We felt that a good language for a domain is one that obscures irrelevant details and brings into sharp focus those that matter. Given that data consistency is a core challenge in distributed computing, we designed Bloom to be *data-centric*: both system state and events are represented as named data, and computation is expressed as queries over that data. The programming model of Bloom closely resembles that of the relational transducers described previously. This is no coincidence: both Bloom and Ameloot's transducer work are based on a logic language for distributed systems we designed called Dedalus.[5] From the programmer's perspective, Bloom resembles event-driven or actor-oriented programming—Bloom programs use reorderable query-like handler statements to describe how an agent responds to messages (represented as data) by reading and modifying local state and by sending messages.

Because Bloom programs are written in a relational-style query language, monotonicity is easy to spot just as it was in relational transducers. The relatively uncommon non-monotonic relational operations—for example, set difference—stand out in the language's syntax. In addition, Bloom's type system includes CRDT-like lattices that provide object-level commutativity, associativity and idempotence, which can be composed into larger monotonic structures.[17]

The advantages of the Bloom design are twofold. First, Bloom makes set-oriented, monotonic (and hence confluent) programming the *easiest constructs for programmers to work with in the language*. Contrast this with imperative languages, in which assignment and explicit sequencing of instructions—two non-monotone constructs—are the most natural and familiar building blocks for programs. Second, Bloom can leverage simple forms of static analysis—syntactic checks for non-monotonicity and dataflow analysis for the taint of nonmonotonicity—to certify when programs provide the eventual consistency properties desired for CRDTs, as well as confirming when those properties are preserved across *compositions* of modules. This is the power of a language-based approach to monotonic programming: local, state-centric guarantees can be verified and automatically composed into global, outcome-oriented, program-level guarantees.

With Bloom as a base, we have developed tools including declarative testing frameworks,[4] verification tools,[6] and program transformation libraries that add coordination to programs that cannot be statically proven to be confluent.[2]

Coordination in its place. Pragmatically, it can sometimes be difficult to find a monotonic implementation of a full-featured application. Instead, a good strategy is to keep coordination off the critical path. In the shopping cart example, coordination was limited to checkout, when user performance expectations are lower. In the garbage collection example (assuming adequate resources) the non-monotonic task can run in the background without affecting users.

It can take creativity to move coordination off the critical path and into a background task. The most telling example is the use of tombstoning for low-latency deletion. In practice, memory for tombstoned items must be reclaimed, so eventually all machines need to agree to delete certain tombstoned items. Like garbage collection, this distributed deletion can be coordinated lazily in the background on a rolling basis. In this case, monotonic design does not stamp out coordination entirely, it moves it off the critical path.

Another non-obvious use of CALM analysis is to identify when to *compensate* ("apologize"[27]) for inconsistency, rather than prevent it via coordination. For example, when a retail site allows you to purchase an item, it should decrement the count of items in inventory. This non-monotonic action suggests that coordination is required, for example, to ensure that the supply is not depleted before an item is allocated to you. In practice, this requires too much integration between systems for inventory, supply chain, and shopping. In the absence of such coordination, your purchase may fail non-deterministically after checkout. To account for this possibility, additional compensation code must be written to detect the out-of-stock exception and handle it by—for example—sending you an apologetic email with a loyalty coupon. Note that a coupon is not a clear mathematical inverse of any action in the original program; domain-aware compensation often goes beyond typical type system logic.

In short, we do not advocate pure monotonic programming as the only way to build efficient distributed systems. Monotonicity also has utility as an analysis framework for identifying nondeterminism so that programmers can address it creatively.

Our question is one of computability … it asks what is (im)possible for a clever programmer to achieve.

Additional Results

Many questions remain open in understanding the implications of the CALM Theorem on both theory and practice; we overview these in a longer version of this article.[29] The deeper questions include whether all PTIME is practically computable without coordination, and whether monotonicity in the CALM sense maps to stochastic guarantees for machine learning and scientific computation.

The PODS keynote talk that introduced the CALM conjecture included a number of related conjectures regarding coordination, consistency and declarative semantics.[28] Following the CALM Theorem result,[9] the database theory community continued to explore these relationships, as summarized by Ameloot.[7] For example, in the batch processing domain, Koutris and Suciu,[31] and Beame et al.[12] examine massively parallel computations with rounds of global coordination, considering not only the number of coordination rounds needed for different algorithms, but also communication costs and skew.

In a different direction, a number of papers discuss tolerating memory inconsistency while maintaining program invariants. Bailis et al. define a notion of Invariant Confluence[11,42] for replicated transactional databases, given a set of database invariants. Many of the invariants they propose are monotonic in flavor and echo intuition from CALM. Gotsman et al.[24] present program analyses that identify which pairs of potentially concurrent operations must be synchronized to avoid invariant violations. Li et al. define RedBlue Consistency,[36] requiring that users "color" operations based on their ordering requirements; given a coloring they choose a synchronization regime that satisfies the requirements.

Blazes[2] similarly elicits programmer-provided labels to more efficiently avoid coordination, but with the goal of guaranteeing full program consistency as in CALM.

Conclusion

Distributed systems theory is dominated by fearsome negative results, such as the Fischer/Lynch/Patterson impossibility proof,[21] the CAP Theorem,[23] and the two generals problem.[25] These results identify things that are not possible to achieve in general in a distributed system. System builders, of course, are more interested in the complement of this space—those things that *can* be achieved, and, importantly, how they can be achieved while minimizing complexity and cost.

The CALM Theorem presents a positive result that delineates the frontier of the possible. CALM proves that if a problem is monotonic, it has a coordination-free program that guarantees consistency—a property of all possible executions of that program. The inverse is also true: any program for a non-monotonic problem will require runtime enforcement (coordination) to ensure consistent outcomes. CALM enables reasoning via static analysis, and limits or eliminates the use of runtime consistency checks. This is in contrast to storage consistency like linearizability or serializability, which requires expensive runtime enforcement.

CALM falls short of being a constructive result—it does not actually tell us how to write consistent, coordination-free distributed systems. Even armed with the CALM Theorem, a system builder must answer two key questions. First, and most difficult, is whether the problem they are trying to solve has a monotonic specification. Most programmers begin with pseudocode of some implementation in mind, and the theory behind CALM would appear to provide no guidance on how to extract a monotone specification from a candidate implementation. The second question is equally important: given a monotonic specification for a problem, how can I implement it in practice? Languages such as Bloom point the way to new paradigms for programming distributed systems that favor and (conservatively) test for monotonic specification. There is remaining work to do making these languages attractive to developers and efficient at runtime.

Acknowledgments. Thanks to Jeffrey Chase, our reviewers, as well as Eric Brewer, Jose Faleiro, Pat Helland, Frank Neven, Chris Ré, and Jan Van den Bussche for their feedback and encouragement.

References

1. Abiteboul, S., Vianu, V., Fordham, B. and Yesha, Y. Relational transducers for electronic commerce. *J. Computer and System Sciences 61*, 2 (2000), 236–269.
2. Alvaro, P., Conway, N., Hellerstein, J. and Maier, D. Blazes: Coordination analysis for distributed programs. In *Proceedings of the IEEE 30th Intern. Conf. on Data Engineering*, 2014, 52–63.
3. Alvaro, P., Conway, N., Hellerstein, J. and Marczak W. Consistency analysis in Bloom: A CALM and collected approach. In *Proceedings of the 5th Biennial Conf. Inno*vative Data Systems Research (Asilomar, CA, USA, Jan. 9–12, 2011) 249–260.
4. Alvaro, P., Hutchinson, A., Conway, N., Marczak, W. and Hellerstein, J. BloomUnit: Declarative testing for distributed programs. In *Proceedings of the 5th Intern. Workshop on Testing Database Systems.* ACM, 2012., 1.
5. Alvaro, P., Marczak, W., Conw ay, N., Hellerstein, J., Maier, D. and Sears, R. Dedalus: Datalog in time and space. Datalog Reloaded. Springer, 2011, 262–281.
6. Alvaro, P., Rosen, J. and Hellerstein, J. Lineage-driven fault injection. In *Proceedings of the 2015 ACM SIGMOD Intern. Conf. Management of Data.* ACM, 2015, 331–346.
7. Ameloot. T. Declarative networking: Recent theoretical work on coordination, correctness, and declarative semantics. *ACM SIGMOD Record 43*, 2 (2014), 5–16.
8. Ameloot, T., Ketsman, B., Neven, F. and Zinn, D. Weaker forms of monotonicity for declarative networking: A more fine-grained answer to the CALM-conjecture. *ACM Trans. Database Systems 40*, 4 (2016), 21.
9. Ameloot, T., Neven, F. and den Bussche, J.V. Relational transducers for declarative networking. *J. ACM 60*, 2 (2013), 15.
10. Backus, J. Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs. *Commun. ACM 21*, 8 (Aug. 1978).
11. Bailis, P., Fekete, A., Franklin, M., Ghodsi, A., Hellerstein, J. and Stoica, I. Coordination avoidance in database systems. In *Proceedings of the VLDB Endowment 8*, 3 (2014), 185–196.
12. Beame, P., Koutris, P. and Suciu, D. Communication steps for parallel query processing. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI Symp. Principles of Database Systems.* ACM, 2013, 273–284.
13. Birman, K., Chockler, G. and van Renesse, R. Toward a cloud computing research agenda. *SIGACT News 40*, 2 (2009).
14. Brewer, E. CAP twelve years later: How the" rules" have changed. *Computer 45*, 2 (2012), 23–29.
15. Chrysanthis, P.K. and Ramamritham, K. Acta: A framework for specifying and reasoning about transaction structure and behavior. *ACM SIGMOD Record 19*, 2 (1990), 194–203.
16. Clements, A.T., Kaashoek, M.F., Zeldovich, N., Morris, R.T., and Kohler, E. The scalable commutativity rule: Designing scalable software for multicore processors. *ACM Trans. Computer Systems 32*, 4 (2015), 10.
17. Conway, N., Marczak, W., Alvaro, P., Hellerstein, J. and Maier, D. Logic and lattices for distributed programming. In *Proceedings of the 3rd ACM Symp. Cloud Computing.* ACM, 2012, 1.
18. Corbett, J. et al. Spanner: Google's globally distributed database. *ACM Trans. Computer Systems 31*, 3 (2013), 8.
19. DeCandia, G. et al. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS Operating Systems Rev. 41*, 6 (2007), 205–220.
20. Eswaran, K., Gray, J., Lorie, R. and Traiger, I. The notions of consistency and predicate locks in a database system. *Commun. ACM 19*, 11 (1976), 624–633.
21. Fischer, M., Lynch, N. and Paterson, M. Impossibility of distributed consensus with one faulty process. *J. ACM 32*, 2 (1985), 374–382.
22. Garcia-Molina, H. and Salem, K. Sagas. In *Proceedings of the 1987 ACM SIGMOD Intern. Conf. Management of Data.* ACM, 249–259.
23. Gilbert, S. and Lynch, N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News 33*, 2 (2002), 51–59.
24. Gotsman, A.,Yang, H., Ferreira, C., Najafzadeh, M. and Shapiro, M. 'Cause I'm strong enough: Reasoning about consistency choices in distributed systems. *ACM SIGPLAN Notices 51*, 1 (2016), 371–384.
25. Gray, J. Notes on data base operating systems. *Operating Systems.* Springer, 1978, 393–481.
26. Hamilton, J. Keynote talk. The 3rd ACM SIGOPS Workshop on Large-Scale Distributed Systems and Middleware. ACM, 2009.
27. Helland, P. and Campbell, D. Building on quicksand. In *Proceedings of the Conference on Innovative Data Systems Research.* ACM, 2009.
28. Hellerstein, J. The Declarative Imperative: Experiences and conjectures in distributed logic. *SIGMOD Record 39*, 1 (2010), 5–19.
29. Hellerstein, J. and Alvaro, P. Keeping CALM: When distributed consistency is easy. 2019; arXiv:1901.01930.
30. Herlihy, M. and Wing, J. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Programming Languages and Systems 12*, 3 (1990), 463–492.
31. Koutris, P. and Suciu, D. Parallel evaluation of conjunctive queries. In *Proceedings of the 30th ACM SIGMOD-SIGACT-SIGART Symp. Principles of Database Systems.* ACM, 2011, 223–234.
32. Kuper, L. and Newton, R. LVARS: Lattice-based data structures for deterministic parallelism. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Functional High-Performance Computing.* ACM, 2013, 71–84.
33. Lamport, L. The part-time parliament. *ACM Trans. Computer Systems 16*, 2 (1998), 133–169.
34. Lampson, B. and Sturgis, H. Crash recovery in a distributed system. Technical report, Xerox PARC Research Report, 1976.
35. Lausen, G., Ludäscher, B. and May, W. On active deductive databases: The state log approach. In *Workshop on (Trans) Actions and Change in Logic Programming and Deductive Databases*. Springer, 1997, 69–106.
36. Li, C., Porto, D., Clement, A., Gehrke, J., Preguiça, N. and Rodrigues, R. Making geo-replicated systems fast as possible, consistent when necessary. *OSDI 12* (2012), 265–278.
37. Meiklejohn, C. and Van Roy, P. Lasp: A language for distributed, coordination-free programming. In *Proceedings of the 17th Intern. Symp. Principles and Practice of Declarative Programming.* ACM, 2015, 184–195.
38. Milano, M., Recto, R., Magrino, T. and Myers, A. A tour of Gallifrey, a language for geo-distributed programming. In *Proceedings of the 3rd Summit on Advances in Programming Languages.* Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.
39. Shapiro, M., Preguiça, N., Baquero, C. and Zawirski, M. Conflict-free replicated data types. In *Proceedings of the Symp. Self-Stabilizing Systems.* Springer, 2011, 386–400.
40. Valiant, L. A bridging model for parallel computation. *Commun. ACM 33*, 8 (Aug. 1990), 103–111.
41. Wadler, P. Deforestation: Transforming programs to eliminate trees. In *Proceedings of the 2nd European Symp. Programming*, 1988.
42. Whittaker, M. and Hellerstein, J. Interactive checks for coordination avoidance. In *Proceedings of the VLDB Endowment 12*, 1 (2018), 14–27.
43. Wu, C., Faleiro, J., Lin, Y. and Hellerstein, J. Anna: A KVS for any scale. In *Proceedings of the 34th IEEE Intern. Conf. on Data Engineering*, 2018.
44. Zinn, D., Green, T. and Ludäscher, B. Win-move is coordination-free (sometimes). In *Proceedings of the 15th Intern. Conf. Database Theory*. ACM, 2012, 99–113.

Joseph M. Hellerstein (hellerstein@berkeley.edu) is the Jim Gray Professor of Computer Science at the University of California at Berkeley, CA, USA.

Peter Alvaro (palvaro@cs.ucsc.edu) is an assistant professor at the University of California at Santa Cruz, CA, USA.

Watch the authors discuss this work in the exclusive *Communications* video. https://cacm.acm.org/videos/keeping-calm