Keeping CALM: When Distributed Consistency is Easy

Introduction:

- In Distributed system, the coordination protocols are known to be barriers to high performance, scale and high availability of distributed systems. For instance: A coordination-free implementation of key value store called Anna (http://db.cs.berkeley.edu/jmh/papers/anna_ieee18.pdf) beats Redis by over 10x on a single AWS instance, and beats Cassandra by 10x across the globe on a standard interactive benchmark.

- If every process remains in their `lane`, then these coordination can be avoided (assume processes are cars on a highway and coordination protocols are stop lights. Intersection can be removed by taking underpass or overpass).
  The perfect freeway is an idealistic analogy but is not always possible to achieve. To demonstrate this author has given two examples: Distributed Deadlock Detection and Distributed Garbage Collection.
  Lets say there are 3 participating machines.
  Distributed Deadlock Detection: If machine M1 and M2 decides there is a deadlock then they do not need any new info from M3 to decide that there is a deadlock in system. New info from M3 will only tell about additional deadlocks, it will not change the earlier established fact.
  Distributed Garbage Collection: Here once a machine or subset of machine detects there is an unreachable object O1, can they declare O1 unreachable ? No, as new info from a new machine may demonstrate that O1 is reachable.

Distributed Deadlock Detection is different than Distributed Garbage Collection in the sense that output does not grow monotonically with input. In case of Distributed Deadlock Detection any new info will not change earlier facts, new info will only establish new fact. While in case of Distributed Garbage Collection, arrival of new info may invalidate earlier knowledge. Distributed Deadlock Detection is monotonic while Distributed Garbage Collection is not.

Monotonic programs are "safe" in the face of missing information, and can proceed without coordination. A program P is monotonic if for any input sets S,T where $S \subseteq T$, $P(S) \subseteq P(T)$. **A program has a consistent, coordination-free distributed implementation if and only if it is monotonic.**

CALM: A PROOF SKETCH

CALM approaches program consistency in terms of application level semantics instead of storage semantics:

- If we recall, a binary operation is commutative if changing the order of the operands does not change the result. ***Confluence*** is generalization of same idea in the context of nondeterministic message delivery. An operation on a single machine is confluent if it produces the same set of outputs for any nondeterministic ordering and batching of a set of inputs.
- Confluent operations compose: if the outputs of one confluent operation are 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.
- Confluence 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 operations are the building blocks of monotonic systems. We still need to take care to avoid negation though. This paper gives an example of confluent shopping cart. In a shopping cart deletes are not monotonic and seem to cause consistency trouble. A common technique is for deletes to be handled separately from inserts as another monotonically growing set of items. Unfortunately, while additions and deletions commute, neither operation commutes with checkout—if a checkout message arrives before some updates, those updates will be lost.

Ameloot and colleagues have presented a formalization and proof of the CALM theorem in their series of papers. They have used a formal execution model (relational transducers), confluence and definition of monotonic programs for the proof. Rest of this section gives a brief overview of that proof (not covered in the slide).

CALM PERSPECTIVE ON THE STATE OF THE ART:

This section discusses connection between calm and state of art in Distributed systems practice like CAP and other distributed design patterns.

- In Brewer's (The CAP theorem Guy) word ''….The modern CAP goal should be to maximize combinations of consistency and availability that make sense for the specific application." Calm includes those sets of applications for which all of three CAP properties can be achieved simultaneously.
- In Functional programming variables are immutable. An immutable variable is a simple monotonic pattern of CALM. Monotonic programming patterns are common in the design of distributed storage systems. What are other monotone pattern we can think of ? (hints: tombstone, monotonic semi-lattice of CRDT)

- Bloom is programming language that encourages CALM programming. Bloom was designed to make distributed systems easier to reason about and program. Bloom's types include CRDT-like lattices that provide object-level commutativity, associativity and idempotence.
- CALM is a noco (No Coordination) way of developing app. As it is difficult to provide a monotonic implementation of a full featured app, its better to move coordination off critical path e.g. in GC, task can run in background without affecting user. Another approach suggested by this paper is to provide compensation (apology) for inconsistencies instead of preventing it via coordination e.g. if purchase fail during checkout non-deterministically, the buyer can be sent an email with loyalty coupon.

Questions:

CALM raises a number of questions at the heart of distributed systems theory and practice. This section basically tries to expand CALM capability by giving reference to the result of some of the other works done in distributed system.

- What is the expressive power of the monotone distributed programs from the CALM Theorem? There are some theorems which tell that monotone logic programs can express all of PTIME under certain assumption.

- Calm provides no assistance in finding monotonic implementations of programs. There are work being done to convert imperative code fragments to monotonic SQL code.
- Recent excitement about machine learning at scale has brought statistical programming concerns to distributed systems. The author believes that CALM definition of consistency can be extended to encompass statistical equivalences like convergence to a near-optimum.

ADDITIONAL RESULTS

Various research work has been done which drives inspiration or intuition from CALM. List of those works are present in `Additional result` section of paper.

Conclusion:

In distributed system there are theorems like CAP and two generals problem which identify things which are not possible in distributed system. In contrast, CALM Theorem presents a positive result that identifies a space where things are possible.
CALM falls short of being a constructive result—it does not actually tell us how to write consistent, coordination-free distributed systems.