CS286 2020

Hellerstein

CAP Theorem

Background: NOW Project and Inktomi

- Cluster-based Search Engine
- Distributed Web Cache

Eric Brewer's CAP Conjecture, PODC 2000 Keynote

- Need Highly Available systems on the web
- Persistent State is Hard
    - Classic Distributed Systems focus on the computation, not the data
        - This is WRONG, computation is the easy part
    - Data centers exist for a reason
        - Can't have consistency or availability without them
    - Other locations are for caching only
        - Proxies, phones, etc.
- First try: ACID vs. BASE
    - Basically Available, Soft-state Eventual consistency
        - Weak consistency (stale data OK)
        - Availability First
        - Best effort
        - Approximate answers OK
        - …
    - "But I *think* it's a *spectrum*"
- The CAP Theorem
    - Consistency, Availability, tolerance to network Partitions
    - Examples of CA (forfeit Partitions)
        - Tightly coupled DBMS
        - LDAP
        - Traits: 2PC, cache validation
    - Examples of CP (forfeit Availability)
        - Distributed DBMS
        - Distributed Locking
        - Quorums
        - Traits: locking, unavailable minorities

    - Examples of AP (forfeit Consistency)
        - Web caches
        - DNS
        - Traits: expirations/leases, conflict resolution, optimistic schemes
- Formalism is offered
    - Need to define C! It's not the C in ACID, alas.
    - Gilbert and Lynch define C as Linearizability
        - Individual request/response rather than transactions
        - Sorta fits HTTP (except many of the web examples hit databases)
        - Footnote of original paper has an incorrect characterization of ACID database transactions (sigh), assets that Linearizability subsumes the A and C in ACID which is silly.
    - Linearizability and Sequential Consistency
        - History of sequential processes issuing requests and receiving responses, in pairs
        - Sequential history: requests followed immediately by matching responses
        - **Sequential Consistency (Lamport):** History H is *sequentially consistent* if the result of the completed requests in H are the same as in some sequential history S
        - **Linearizability (Herlihy & Wing):** History H is *linearizable* if there is extension of H called H' with 0 or more additional events, such that:
            - H' is sequentially consistent
            - The order of operations in H' matches those of S -- i.e. if the response of op1 precedes request of op2 in H', then op1 precedes op2 in S
            - I.e. linearizability is more restrictive, requires order to be preserved.
        - Side-comment: the wikipedia discussion of Linearizability has all sorts of weird incorrect references/comparisons to Serializability.
    - Gilbert and Lynch's CAP Theorem in a nutshell
        - Consistency: Linearizability
        - Availability: every request received by a non-failing node must result in a response
        - Partition Tolerance: requests may be lost arbitrarily; responses are guaranteed, and guaranteed to be consistent.
        - Theorem: AC impossible under partition: It's impossible in this asynchronous network model to implement a read/write data object that guarantees Availability and Atomic Consistency in all fair executions (including those in which messages are lost).
        - Simple proof by contradiction: under partition, some write occurs on one side, and later a read occurs on the other side and cannot see the write.
- Nuance Ensues

    - Coda Hale's "You can't sacrifice partition tolerance". It's C or A, baby. Anything else is a dodge. OK sure. Call it the CA Theorem.
    - Dan Abadi's PACELC
        - In the case of Partitions:
            - Choose A or C
        - Else
            - Choose L or C
        - Removing the acronymization: a partition is just an example of very high latency. We knew that already.

CALM Theorem

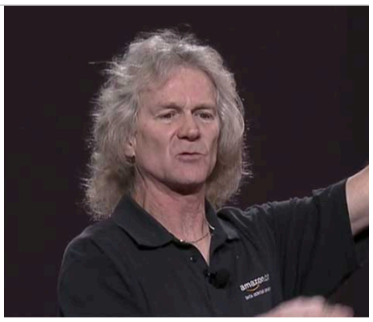
Background: Declarative Networking

- We had implemented a variety of network variants of Datalog (NDlog, Overlog)
    - Used them to implement various protocols and systems (e.g. Chord, Hadoop and HDFS Masters, Paxos, etc)
- Wanted to move on to a bigger challenge: Programming the Cloud!
- Semi-imperative race conditions seemed to occasionally cause trouble.
    - Example: elections
        - Voter registration: easy, just a set
        - Majority test: also easy, just counting and comparing
        - Not well defined if concurrent!!
        - Something is unstable about |votes|/|voters|!
    - Example: state update
        - What happens when we want to overwrite or delete the value of a key?
        - Datalog doesn't model this at all!
        - Not well defined if concurrent with queries, obviously
- Dedalus (Peter Alvaro): A network Datalog variant that provided a formal semantics for state update and asynchronous messaging
    - Cleaned up the previous questions via a notion of *time*.
    - But when do we need to use this *time* construct, really?

Joe Hellerstein's CALM Conjecture, PODS Keynote 2010

- We need to figure out how to program the cloud!! But it's hard
    - Parallel computing, message reordering, partial failure…
- Distributed Computing is all about coordination protocols. But coordination sucks!

Coordination Avoidance (a poem)



> “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”

—James Hamilton (IBM, MS, Amazon)
in Birman, Chockler: “Toward a Cloud Computing Research Agenda”, LADIS 2009

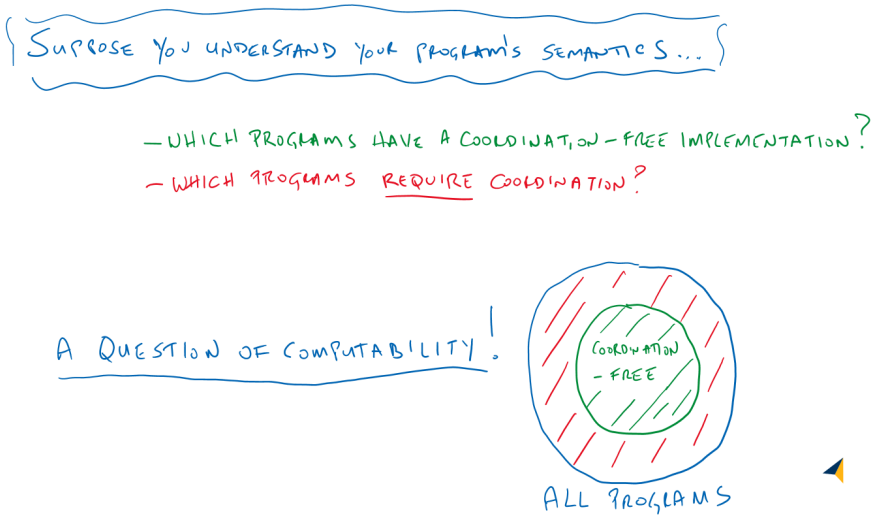
21

- It’s complicated
- Waiting is bad: tail latencies mean it gets worth with scale (straggler effect)
- Waiting leads to queuing: “slowdown cascades”, etc.
- The problem is the Read/Write abstraction

With thanks to Peter Bailis…

  - We need more semantic *Information* a la Kung/Papadimitriou!
  - Programming Languages to the rescue?
- Query Processing got this right -- those parallelized really nicely! Maybe a logic language is the answer? Databases to the rescue?!
  - Subsets of SQL are “embarrassingly” parallel: Unordered collection types
  - Embarrassing = Fast and Easy!
  - Embarrassing = Eventually Consistent too!

    - Let's not be embarrassed by what works!
- Hmm… but when can we get away with this?



- Intuition from Database queries
    - select, project, join all have *streaming* implementations
        - Major annoyance that MapReduce conflated shuffle with barrier!
        - Prevented the implementation of streaming joins, well-known in db literature (Wilschut and Apers)
    - set difference and negation require *blocking:* a dataflow barrier
        - These are non-monotonic operators!
        - I.e. suppose you emit an output. Then a new input arrives -- it may force you to retract the previous output!
    - Monotonicity means *never having to apologize*. Hence streaming! One direction seems obvious.
    - But does Non-Monotonicity require blocking? E.g. Not Exists?!
- CALM: A distributed program has a consistent, coordination free distributed implementation iff it is monotone.
- Now we need definitions:
    - Consistency (of programs, not of data!!)
    - Coordination
    - Monotonicity

Formalism is Offered

- Tom Ameloot, Frank Neven and Jan Van den Bussche. "Relational Transducers for Declarative Networking". PODS 2011, JACM 2013.
- Formalism of a Transducer Networks

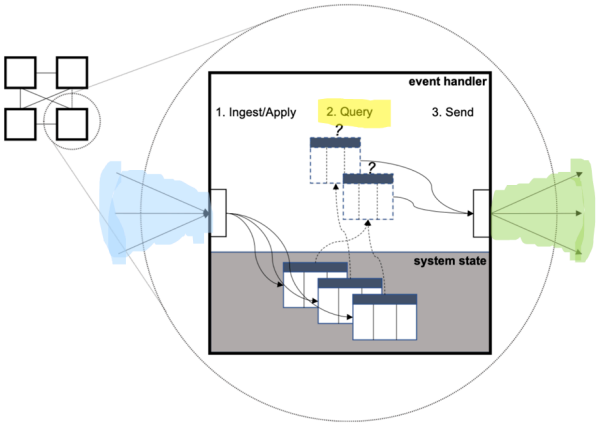
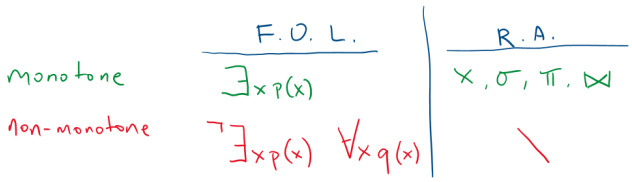
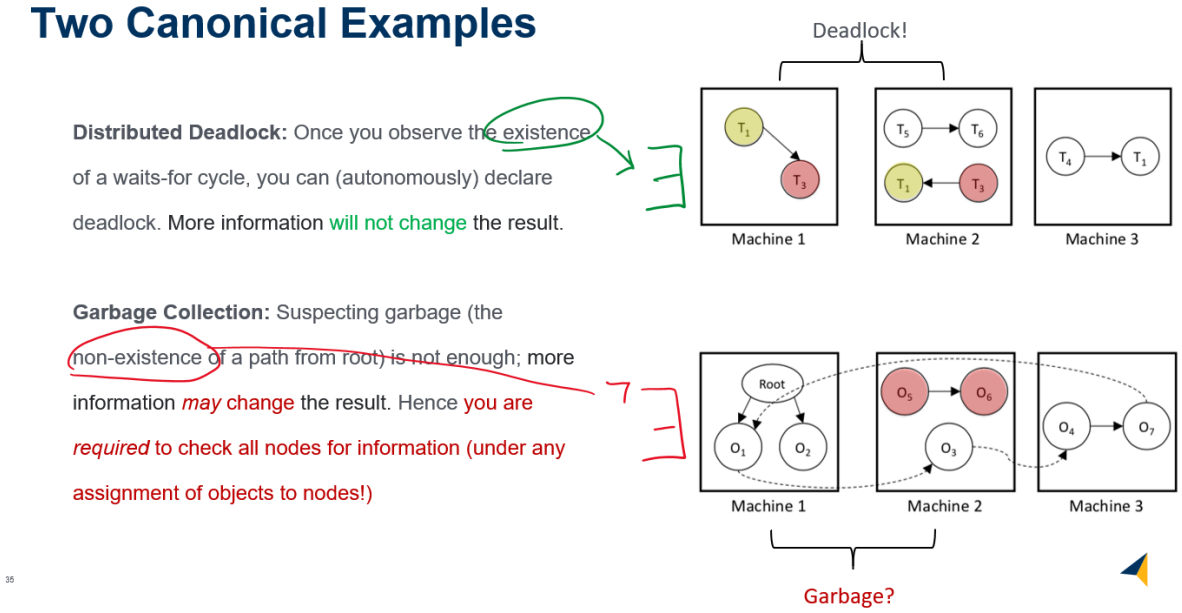


Figure 3: A simple four-machine relational transducer network with one machine's state and event loop shown in detail.

-
- **Consistency**: *Confluent* Distributed Execution
  Definition: A distributed program *P* is *consistent* if it is a deterministic function from sets to sets, regardless of non-deterministic message ordering and duplication.
- **Monotonicity.** In terms of the language at each transducer.
  Definition: Definition: A distributed program *P* is *monotonic* if for any input sets *S, T* if $S \subseteq T$, then $P(S) \subseteq P(T)$.



- **Coordination: Data-Independent Messaging**
  Definition: A distributed program *P(T)* uses *coordination* if it requires messages to be sent under all possible input partitionings of *T*. (Including partitionings where all the data is colocated at a single node!)
- **CALM Theorem** (full version):
  - Let L be a query language containing unions of conjunctive queries. For every query Q that is expressible in L, the following are equivalent:
    - Q can be distributedly computed by a coordination-free L-transducer
    - Q can be distributedly computed by an L-transducer that knows its own ID but not the membership of the network
    - Q can be distributedly computed by an L-transducer that knows the membership of the network but no its own ID
    - Q is monotonic
  - The proof is not an easy read



- Subsequent work adds nuance to the definition of monotonicity, as well as the information available to the system (e.g. partitioning metadata). Also additional work on decidability and other issues. Ameloot wrote a checkpoint survey in 2014 but work has been done since.

Can we compare CAP and CALM?

- Not exactly. The two theory papers have different definitions of Consistency!
- CAP is a *negative* result. CALM is differentiator between negative and positive cases
- But Gilbert/Lynch didn't really capture Brewer's intent. It's really not about linearizability. In his retrospective, Brewer wrote:
  - > [The original] expression of CAP served its purpose, which was to open the minds of designers to a wider range of systems and tradeoffs ... The modern CAP goal should be to maximize combinations of consistency and availability that make sense for the specific application. [15]
- What "combinations of consistency and availability" are possible? Remember Coda Hale? You gotta choose?
- CALM answers just this question
  - Monotone programs can have both consistency (deterministic outcomes) and availability, even under partition!

    - Non-monotone programs can*not*.

- For more, please see Keeping CALM: When Distributed Consistency is Easy. To appear in CACM.