Distributed Programming in Argus

Barbara Liskov, CACM 1988

538B projects

- Thank you for the proposal *drafts*
- *Finalized* project proposals due Friday at 6pm Vancouver time
- Happy to chat about the project proposal
- Please share a version I can edit/comment on

Argus abstractions

- CLU (CLUsters) introduced : *iterators, exceptions, parallel assignment, abstract data type, objects, parametrized types.*

- What about C++? Earlier/Later depends on private/public. Eventually merged in many of the same concepts

- Argus builds on CLU, and adds : better state management, message passing, guardian *objects (actors with fault tolerance)*, action with strict distributed consistency and atomicity guarantees, top-action to coordinate nested *distributed transaction* using 2 phase commit (2PC)

- *Stable* keyword: explicitly denotes state that will persist across failures (at top of guardian definition)

- Argus … ahead of its time! Influenced a ton of work in the PL + distributed system space.

Argus

- What's hard or ridiculous about Argus?
- *Deadlocks* : Argus doesn't prevent deadlocks, and doesn't detect them… so they'll just happen when programmers make mistakes.
    - Well, locks are the same!
    - e.g., why not have Argus detect deadlocks like Go?
- What are the locks/how many locks in an Argus program? **EVERY** data object that you can reference (read or mutate) has a lock.
    - *Fine-grained locking* in Argus increases chance of deadlocks
    - Optimistic: create as many locks as may ever be necessary
    - Pessimistic: many locks!
- Is deadlocking proclivity a fair criticism?
    - *Deadlocks escape the abstraction :* which makes them the programmer's problem, and difficult to deal with (unlike machine failures : handled as part of the abstractions in Argus).
    - Humans have to reason about deadlocks: reason about the compiler's locking strategy + many many locks
    - Really loads the human capacity for reasoning about concurrency
- Finn Trivia: Java supports *synchronized.* The intent behind these was to introduce locks to implement synchronized. The reality of the implementation is not to do this.

More generally

- The central question: *how much concurrency control do you introduce into a language/compiler? And how much distribution control?*
- *Pros of introducing more*:
    - less code to write (less boilerplate)
    - compiler does more work for you (can optimize common cases)
    - handles the complexity (correctness! Only implement once, correctly by construction)
    - global compiler reasoning over the entire system, compiler can choose appropriate optimizations (can even target networking environments)
- *Cons of introducing more*:
    - need a *really* complex compiler
    - compilers must generalize *(it will never be as fast as special purpose custom code; but requires really smart human for this specialization)*
    - bottleneck to a compiler is expressiveness of the language (the more you can convey, the better, but this makes the language complex)
- Junfeng: Can we separate the concurrency/distribution notions from the PL, and integrate them independently into libraries/tools to benefit all PLs? PonyLang language with actors
    - ZooKeeper (written in Java)? Chubby? Kafka? Spark? MapReduce? …
    - OR just wait for C++ v1024 (or Perl v10) ?
- Why isn't ZooKeeper written in *something like Argus*?
    - It needs open source developers that know the language
    - It needs ability to use the "best" compiler and evolve over time
- **Bottomline: language popularity determined by features, libraries, familiarity, crowd effects, supported platforms (see JavaScript)**

Argus implementation/ design

- *Strict two phase locking*: concurrency within a guardian handler.
- *Nested Two phase commit*: once or zero semantics for actions (including top-actions, sub-actions)
  - Requires all nodes to commit (no notion of quorum)
  - Runs on all the replicas hosting guardians that are involved in a distributed transaction (servicing an action)
  - Safe but not live: coordinator failure in a particular state causes the transaction to stall until coordinator comes back up (top action, or the caller of the action)
  - There is no view change (not like PBFT/Paxos)
- Fault tolerant objects that are coordinated by 2PC (via nested actions)
- What about ordering? Does it provide an ordering that a client would actually want?
  - Serializable ordering : looks like a linear sequence of events against a collections of objects.
  - Bad ordering:
        1. accounts.total() called by clientA at 1PM
        2. For ac in accounts : ac.deposit(1) by clientB…clientZ at 1:01PM   (this might take a long time)
    - The eventual ordering: reverse of the above (deposits finish first, total finishes last).
- Very much like distributed database operations (e.g., SQL-like): commands issued to an ACID database have the same ordering concerns

Argus

- Isn’t weird how it sort of looks like a database built into a PL?
  - Argus = enriched SQL
  - Argus = SQL notions (of txns) built into a general purpose PL (CLU)
- Doesn’t really come close to SQL (declarative, and isn’t backed by a relational algebra); pales in comparison?
- Modern research languages that are declarative and DB-like for constructing distributed systems (e.g., Bloom http://bloom-lang.net/)

Next: Emerald

- A more fleshed out *object-based* distributed programming system. Perhaps the culmination of such systems (late 80s).
- Focus on mobility and compilation