the morning paper

an interesting/influential/important paper from the world of CS every weekday morning, as selected by Adrian Colyer

Chapar: Certified Causally Consistent Distributed Key-Value Stores

FEBRUARY 8, 2016

tags: Consistency, Distributed Systems, Formal methods

Chapar: Certified Causally Consistent Distributed Key-Value Stores (http://adam.chlipala.net/papers/ChaparPOPL16/ChaparPOPL16.pdf) – Lesani et al. 2016

Another POPL '16 paper today. The Chapar framework provides for *modular verification* of causal consistency for both causally consistent key-value store implementations and for client programs written to use them. §1 also wins the prize for best use of emojis in a research paper!

> *…precise specification of causal consistency and verification techniques that check the compliance of key-value store implementations with the specification enhance the reliability of applications that are built using these data stores. In addition, causal consistency provides weaker guarantees than serializability to clients. Thus, the client programs are exposed to less consistent data and are prone to more bugs. Therefore, automatic checkers are a useful aid for client programmers to verify that their programs preserve their application invariants if executed with causally consistent stores. Although there have been recent efforts on verification of eventual consistency, to the best of our knowledge, our work presented here is the first to address causal consistency.*

It's quite amazing to think how much energy has been collectively spent discussing, implementing, specifying, and verifying distributed implementations of the humble map!

Causally-content Client Programs

Starting on the client side, consider a program containing application invariants – the classic picture and announcement of picture example maybe:

Alice:

```
put(Pic, <picture>)
put(Post, "Hey, check out my picture...")
```

Bob:

```
post ← get(Post)
photo ← get(Photo)
```

We want it to be the case that Bob can never see the announcement of the picture before he can see the picture itself. This leads to an application assertion:

```
assert(post = Post ⇒ pic ≠ ⊥)
```

Lesani et al. define a simple model for expressing client programs (essentially put, get and assert, executed across N nodes), and an abstract key-value store interface for implementations. Beyond init, put, and get, this interface also contains an update operation for propagating updates to other nodes, and a *guard* function that is used to delay the application of updates until certain conditions hold (by this means, causal consistency can be implemented).

The next step is to define what it means for a key-value store to be causally consistent (without referring to the details of any one specific implementation). The authors present an *abstract operational semantics* for causal consistency that defines what it means for an execution of a program to be causally consistent. The semantics define a happens-before relation (though that phrase never actually appears in the paper). Put operations are uniquely identified, and they track the identifiers of other put operations on which they depend. A put cannot be applied until all of the operations it depends on have been applied. An operation at a node n depends on all preceding operations at that node. If a get operation returns a value that a put operation has put, then it is dependent on that put operation. Dependencies are transitive.

For client programs, this is enough to determine whether or not they are *causally content*. A causally content program is one that sees no assertion failures when executed with the abstract causal consistency operational semantics.

> *A program is causally content if and only if every abstract causal execution of the program is assertion-fail-free.*

By proving causal content therefore, we can prove that a program is safe to execute against a causally-consistent store.

Faithful implementations of Causal Consistency using ‘well-reception’

Taking the same abstract operational semantics, we can define an actual implementation to be causally consistent iff its *concrete* operational semantics refine the abstract operational semantics.

> *An operational semantics is a refinement of another operational semantics if and only if every external trace of the former is an external trace of the latter.*

And if we can show this refinement relation, we also know that any causally-content client program will be assertion free when executed against the store.

> *The immediate implication of this lemma is that the implementations and the clients can be verified separately to be causally consistent and causally content respectively, and the combination of every pair of verified implementation and verified program is safe.*

The *well-reception* condition helps simplify the proof obligation that a concrete implementation satisfies the abstract semantics. Any implementation that meets the well-reception condition is causally consistent. So what is this condition? To give you a flavour of some of the detail I'm eliding, here's the full definition from the paper:

We must be able to define some function *Rec* for the implementation, that satisfies four conditions: an *initial* condition, a *step* condition, a *causality* condition, and a *sequential* condition.

Rec :: NodeState → NodeId → Int

For some node n in a given state σ, Rec(σ,id) returns the count (*clock*) of the number of updates that n has received from the node with id *id*.

The Initial Condition

The initial state returned by the implementation's *init* function has a 0 count for updates received from every node.

The Step Condition

When performing a put, the Rec count for the current (originating) node should be incremented (and that of all other nodes remains unchanged). Get operations do not affect the Rec mapping. An update received from node n increments the counter for that node, and all other mappings remain unchanged.

The Causality Condition

If an update is received for some put operation p , then the updates for all put operations that p is causally dependent on must have been received already. We know this if the number of updates received from the originating node of p (the clock value) is greater than or equal to the update clock value.

The Sequential Condition

The functions put and update should update the mapping for the given key to the given value, and leave all other mappings unchanged. If we treat updates as simple puts, this means that the implementation refines a sequential map.

Implementation Verification and Extraction

Two implementations from the literature are verified using the well-reception condition and Coq. The first uses vector clocks for each node and tracks all dependencies, the second stores only *one-hop* (immediate) dependencies. This turns out to be sufficient because of the transitivity of dependencies.

> *In contrast to many verification efforts that work on abstract models of code, our development leads to executable code… We extracted the two implementations from Coq to OCaml using Coq's built-in extraction mechanism. We wrote a shim in OCaml that implements network primitives such as UDP message passing, queuing, and event handling. We compiled and linked the extracted implementations together with the shim to obtain executable key-value stores. The trusted computing base of these stores includes the following components: the assumption that the concrete semantics matches the physical network and the shim, Coq's proof checker and OCaml code extractor, and the OCaml compiler, runtime, and libraries.*

from → Uncategorized

One Comment leave one →

Trackbacks

1. An empirical study on the correctness of formally verified distributed systems | the morning paper