Genie

Distributed Systems Synthesis and Verification

Marc Rosen

EN.600.667: Advanced Distributed Systems and Networks

May 1, 2017

Outline

- Introduction
  - Problem Statement
  - Prior Art
- Demo
- How does it work? Code Generation
- How does it work? Distributed Systems
- How does it work? Programming Languages
- Conclusion

Problem Statement (Background)

Distributed Systems are Useful

- Partition tolerant (i.e. offline-capable)
- Scalable

Distributed Systems are Hard

- Typically requires formal training or study
- Even then, it's easy to make mistakes
- Even simple systems can be time-consuming to implement properly

Problem Statement (Goals)

- Can we come up with a way to *specify* the semantics of a distributed system, and then *generate* the code for the specified system?
- Can we also make it *fool-proof*, and accessible to users without formal distributed systems training?

Prior Art (Industry Solutions)

- Apache Cassandra has: [8]
  - 9 write consistency levels
  - 10 read consistency levels
- Apache CouchDB lets the developer choose between: [10]
  - Using a CAS-loop for strict consistency
  - Arbitrarily picking a “winner” on conflict. All conflicting versions are stored. The developer should manually resolve the conflict.

Prior Art (Interactive Theorem Provers)

- The Coq Proof Assistant [13]
- SAML (System Analysis Modelling Language) [5]
- Constable's EventML [4]
- ...and many others

Prior Art (Model Checking Solutions)

- Leslie Lamport's TLA+ [12]
- CISE [6] & Indigo [2]
- The Leon Verification System [3]
- ...and many others

Outline

Demo: Mail

- This is the final project from the Fall 2016 Distributed Systems Course
- Users can connect to one of five mail servers, and "login" as a specific user
- The following operations are supported:
  1. List email messages
  2. Send an email message
  3. Delete an email message
  4. Mark an email message as read
- The entire system must be partition-tolerant and crash-tolerant

Demo: Chat

- This is the final project from the Fall 2014 Distributed Systems Course
- Users can connect to one of five chat servers, and "login" as a specific user
- The following operations are supported:
  1. Join a room
  2. Send a message to the room
  3. Like a message
  4. Unlike a message
- The entire system must be partition-tolerant and crash-tolerant

Demo: Simple verification example

- In order to be able have meaningful verification, we need a way to express domain-specific invariants about the system...

Questions (before the next part)?

Outline

Code Generation: Overview

- The AST is type-checked
- We generate C++ code from Twirl templates (a templating language for Scala)
- We generate one struct per class in the source to hold the properties.
- We generate one struct per exposed method.

Outline

Algorithmic Overview

- All updates/operations have a (Lamport timestamp, server, per-server monotonic counter) triple for an ID.
- Servers maintain and exchange the matrices consisting of the highest ID update that they've received from another server.
- Servers maintain lists (by server of origin) of all updates they've ever received. These lists are used for reconciliation.
- The current state of objects are stored as a mapping from ID to object.

Reconciliation Algorithm

1. On partition change, start queuing any incoming client requests. (In the even that a partition occurs during this algorithm, keep adding to the current queue, but otherwise reset other state associated with this algorithm.)
2. Buffer (into an array) any server-to-server updates that come in during this reconciliation period.
3. Once a server has sent out all of its updates for reconciliation, it sends out a “finished reconciliation message” to all the other servers
4. Once the server has received a “finished reconciliation message“ from every server in the partition, then it sorts the updates that came in by (ℓ, s, c) and then applies them in that order (which is guaranteed to be causal).
5. Process any queued incoming client requests, and stop queuing future client requests. Instead, process them immediately.

CRDTs [9]

- a Commutative (or Convergent) Replicated Data Type
- In general, they're data types that have the properties that you'd want for eventual consistency in a distributed system.

Two equivalent formulations [11]

State-based (CvRDT)

Operations are homomorphisms on a join semilattice.
(Operations respect a partial ordering on the set of possible states. Any two states in the semilattice have a least upper bound.)

Operation-based (CmRDT)

Operations are transmitted in causal order. Any operations that can happen concurrently must commute.

Object Model

- The *Universe* consists of several mappings (classes) from identifiers to fields of atomic type (i.e. they're not class instances).
- There are two kinds of IDs:

  Unique IDs
  : Totally ordered (in a causal order), but opaque otherwise. You get a fresh Unique ID every time you ask for one.

  Primary Key
  : Meaningful keys that can be used to tie an object to some quantity

- The ID of an object witnesses its existence

Operations Model

- Each operation has a precondition that must be met in order for the operation to take effect
- Non-strict consistency operations must commute with all other operations

Proof Rules

Let

1. The invariants are satisfiable (they don't conflict)
2. $\forall u, o$ where u is a universe and o is an operation $\textbf{Pre}(u) \wedge \textbf{Inv}(u) \implies \textbf{Inv}(op(u))$
3. The default values for objects with primary keys don't violate invariants
4. $\forall u, o, o'$ where u is a universe, o is an operation, and o' is an operation that doesn't require strict consistency. Then:
   4.1 $\textbf{Inv}(u) \wedge \textbf{Pre}(u) \implies \textbf{Pre}'(op(u))$
   4.2 $\textbf{Inv}(u) \wedge \textbf{Pre}'(u) \implies \textbf{Pre}(op'(u))$
   4.3 $\textbf{Inv}(u) \wedge \textbf{Pre}(u) \wedge \textbf{Pre}'(u) \implies op(op'(u)) = op'(op(u))$

Outline

The Specification Language: Some Highlights

- Is highly inspired by C# (it's not C#, though)
- The query syntax is very similar to LINQ in C# [7]
- No recursion. No higher-order functions. Strongly normalizing.
- The query and iteration syntax is rigged such that there's no way to access the i-th element of a list, which means that we can reason about lists as sets.
- The type of a Unique ID is tagged with the class that it is identifying (since otherwise it wouldn't act as a witness)

How Verification Works (an overview)

- We encode a given proof rule into SMTLIB2 format for the Z3 SMT solver [14]
- This amounts to converting the program, as viewed by the proof rule into a logical expression
- Objects are represented by sets (i.e. arrays from the object to booleans)
- Lists are represented as triples of (class to quantify over, map, filter). Lists get encoded to $\forall x \in C$ such that $\phi(x), f(x)$.
- `for(x in l) {assert f(x);}` get encoded in the same way

Outline

Conclusion

We've created a basic prototype to meet the ease-of-use goals that we set out to solve.

We think that this prototype should help to demonstrate the viability and the utility of having an easy-to-use tool to generate distributed systems. The key idea that makes such tooling viable is that specifying the system all at once makes these sorts of analyses possible.

Ideas for Future Work

- Finishing this prototype
- Determine the optimal (especially state-based) CRDT to use in a given situation based on the specification.
- Add support for other CRDTs like a numeric escrow CRDT [1]
- Can we automatically determine when we can relax the constraints of causal consistency in reconciliation to weak consistency, so that we can improve performance?
- And much much more...

Questions?

References I

[1] Valter Balegas et al. "Extending eventually consistent cloud databases for enforcing numeric invariants". In: *Reliable Distributed Systems (SRDS), 2015 IEEE 34th Symposium on*. IEEE. 2015, pp. 31–36.

[2] Valter Balegas et al. "Putting consistency back into eventual consistency". In: *Proceedings of the Tenth European Conference on Computer Systems*. ACM. 2015, p. 6.

References II

[3] Régis Blanc et al. "An overview of the Leon verification system: Verification by translation to recursive functions". In: *Proceedings of the 4th Workshop on Scala*. ACM. 2013, p. 1.

[4] *EventML*. http://www.nuprl.org/software/. Accessed: 2017-05-01.

[5] *Getting started with SAML*. http://rise4fun.com/SAML/tutorial/tutorial. Accessed: 2017-05-01.

References III

[6] Alexey Gotsman et al. "'Cause I'm strong enough: Reasoning about consistency choices in distributed systems". In: *ACM SIGPLAN Notices* 51.1 (2016), pp. 371–384.

[7] Anders Hejlsberg et al. *C# Programming Language*. Addison-Wesley Professional, 2010.

References IV

[8] *How is the consistency level configured?*. http://docs.datastax.com/en/dse/5.1/dse-arch/datastax_enterprise/dbInternals/dbIntConfigConsistency.html. Accessed: 2017-05-01.

[9] Marc Shapiro—Nuno Preguiça. "Designing a commutative replicated data type". In: *arXiv preprint arXiv:0710.1784* (2007).

References V

[10] *Replication and conflict model.* http://docs.couchdb.org/en/2.0.0/replication/conflicts.html. Accessed: 2017-05-01.

[11] Marc Shapiro, Carlos Baquero, and Marek Zawirski. "A comprehensive study of Convergent and Commutative Replicated Data Types". In: (2011).

References VI

[12] *The TLA Home Page*. http://lamport.azurewebsites.net/tla/tla.html. Accessed: 2017-05-01.

[13] *Welcome! — The Coq Proof Assistant*. https://coq.inria.fr/. Accessed: 2017-05-01.

[14] *Z3 SMT Solver*. https://github.com/Z3Prover/z3. Accessed: 2017-05-01.