A framework for verifying Conflict-free Replicated Data Types (CRDTs)

Dominic Mulligan
Joint work with Victor Gomes, Martin Kleppmann, and Alastair Beresford

S-REPLS 6, University College London, May 2017

Distributed systems: quick introduction

Distributed systems characterised by:

- A number of nodes/processes
- Potentially widely geographically distributed
- Performing local processing
- Passing messages back and forth across a network

Distributed systems: quick introduction

Distributed systems characterised by:

- A number of nodes/processes
- Potentially widely geographically distributed
- Performing local processing
- Passing messages back and forth across a network

Implicit shared global state

Strong consistency

Strong consistency model:

- Used in e.g. in relational databases
- Tries to make system behave like a single machine

Strong consistency

Strong consistency model:

- Used in e.g. in relational databases
- Tries to make system behave like a single machine

May be costly to achieve:

- Hundreds of thousands, or millions of replicas
- Replicas may be widely separated, or network slow

or not desired: e.g. in calendar applications

Relaxed consistency models

Observations motivate *relaxed* consistency models

Popular alternative is *eventual consistency*

Used in many NoSQL distributed database systems

Relaxed consistency models

Observations motivate *relaxed* consistency models

Popular alternative is *eventual consistency*

Used in many NoSQL distributed database systems

Informally:

1. If no changes are made to some component of shared global state
2. Eventually all replicas will converge to some consensus on value of that component

Eventual consistency

Eventual consistency is very weak consistency model!

Eventual consistency

Eventual consistency is very weak consistency model!

Model:

- makes no guarantees in case where updates never cease,
- and does not constrain return value of intermediate reads

Trade-offs

Range of consistency models exists because of a tradeoff:

Performance vs. guarantees offered by model

Trade-offs

Range of consistency models exists because of a tradeoff:

Performance vs. guarantees offered by model

Possible to find intermediate models:

- Obtain stronger guarantees than those provided by eventual consistency
- Without inheriting the inherent costs of strong consistency

Trade-offs

Range of consistency models exists because of a tradeoff:

Performance vs. guarantees offered by model

Possible to find intermediate models:

- Obtain stronger guarantees than those provided by eventual consistency
- Without inheriting the inherent costs of strong consistency

One such consistency model is *strong eventual consistency*

Strong eventual consistency

Strong eventual consistency (Shapiro, 2011) has:

- Performance and fault-tolerance characteristics similar to eventual consistency
- Provides stronger guarantees

Strong eventual consistency

Strong eventual consistency (Shapiro, 2011) has:

- Performance and fault-tolerance characteristics similar to eventual consistency
- Provides stronger guarantees

A system satisfying strong eventual consistency must satisfy:

- Correct replicas that have received same updates have same state,
- Update delivered at a replica is eventually delivered to all replicas,
- All executions must terminate

Strong eventual consistency

Strong eventual consistency (Shapiro, 2011) has:

- Performance and fault-tolerance characteristics similar to eventual consistency
- Provides stronger guarantees

A system satisfying strong eventual consistency must satisfy:

- Correct replicas that have received same updates have same state,
- Update delivered at a replica is eventually delivered to all replicas,
- All executions must terminate

First property (‘convergence’) most important

Conflict Free Replicated Datatypes (CRDTs)

Achieving consensus in SEC:

- Typically requires conflict resolution policy
- Simple policies cause data loss, complex ones are error prone

Conflict Free Replicated Datatypes (CRDTs)

Achieving consensus in SEC:

- Typically requires conflict resolution policy
- Simple policies cause data loss, complex ones are error prone

Observation:

- Some operations naturally commutative
- Concurrent operations can therefore be applied in any order

Conflict Free Replicated Datatypes (CRDTs)

Achieving consensus in SEC:

- Typically requires conflict resolution policy
- Simple policies cause data loss, complex ones are error prone

Observation:

- Some operations naturally commutative
- Concurrent operations can therefore be applied in any order

Generalise this to other data structures and operations

CRDTs = replicated data types with commutative (concurrent) operations

Three families of CRDT

Three different types of CRDT are known:

- Operation based CRDTs: construct operations so they are commutative
- State based CRDTs: broadcast entire state of replica when it changes
- Delta-CRDTs: broadcast changes to states, not entire states

Three families of CRDT

Three different types of CRDT are known:

- Operation based CRDTs: construct operations so they are commutative
- State based CRDTs: broadcast entire state of replica when it changes
- Delta-CRDTs: broadcast changes to states, not entire states

Operation based CRDTs require stronger network behaviour

We consider operation based CRDTs in this work

Why is verification necessary here?

Operational Transformation is a closely related technique:

Why is verification necessary here?

Operational Transformation is a closely related technique:

- Algorithms have been presented without any proof

Why is verification necessary here?

Operational Transformation is a closely related technique:

- Algorithms have been presented without any proof
- Hand-written proofs of convergence tend to be long and complex

Why is verification necessary here?

Operational Transformation is a closely related technique:

- Algorithms have been presented without any proof
- Hand-written proofs of convergence tend to be long and complex
- Algorithms have claimed to satisfy TP_2 but later shown incorrect

Why is verification necessary here?

Operational Transformation is a closely related technique:

- Algorithms have been presented without any proof
- Hand-written proofs of convergence tend to be long and complex
- Algorithms have claimed to satisfy TP_2 but later shown incorrect
- In classic formulation of OT, later shown impossible to achieve TP_2

Why is verification necessary here?

Operational Transformation is a closely related technique:

- Algorithms have been presented without any proof
- Hand-written proofs of convergence tend to be long and complex
- Algorithms have claimed to satisfy TP_2 but later shown incorrect
- In classic formulation of OT, later shown impossible to achieve TP_2
- Several wrong machine checked proofs presented (!)

Why is verification necessary here?

Operational Transformation is a closely related technique:

- Algorithms have been presented without any proof
- Hand-written proofs of convergence tend to be long and complex
- Algorithms have claimed to satisfy TP_2 but later shown incorrect
- In classic formulation of OT, later shown impossible to achieve TP_2
- Several wrong machine checked proofs presented (!)

See also (Fonseca et al):

An Empirical Study on the Correctness of Formally Verified Distributed Systems

More verification?

Rest of talk: overview of another distributed systems verification project

We use Isabelle/HOL to provide a mechanical proof of correctness

But why should you trust our proof?

Diagnosing bugs in previous proofs

Relied on high-level axioms stated in terms of data structure, and operations on it

Diagnosing bugs in previous proofs

Relied on high-level axioms stated in terms of data structure, and operations on it

But reasoning about data structure is hard, and unintuitive

(It's why there's a mechanical proof in the first place…)

Wrong axioms are therefore hard to spot

Need to think of all possible network behaviours to spot them

Diagnosing bugs in previous proofs

Relied on high-level axioms stated in terms of data structure, and operations on it

But reasoning about data structure is hard, and unintuitive

(It's why there's a mechanical proof in the first place...)

Wrong axioms are therefore hard to spot

Need to think of all possible network behaviours to spot them

Claim: need end-to-end verification to ensure correctness

Ensure required properties hold in all network executions

Our contributions: a framework for verifying CRDTs

We develop a stratified proof *framework* for CRDTs

Our contributions: a framework for verifying CRDTs

We develop a stratified proof *framework* for CRDTs

We have three different ‘components’:

- A network model
- An abstract convergence theorem
- Implementations of concrete CRDTs

Our contributions: a framework for verifying CRDTs

We develop a stratified proof *framework* for CRDTs

We have three different ‘components’:

- A network model
- An abstract convergence theorem
- Implementations of concrete CRDTs

Piecing all three together gives a concrete convergence theorem for the CRDT

Two components are reusable

Our contributions: modelling the network

In one component, we:

- Model a standard class of networks, axiomatically
- Our axioms are easy to defend, only 5 of them
- Use standard broadcast/deliver event model for network
- Contents of messages are abstract

Our contributions: modelling the network

In one component, we:

- Model a standard class of networks, axiomatically
- Our axioms are easy to defend, only 5 of them
- Use standard broadcast/deliver event model for network
- Contents of messages are abstract

By working axiomatically:

- Reason about all possible network behaviours
- Eliminate corner cases that may invalidate proofs

Our contributions: abstract convergence

In another component, we:

Parameterise our work with a strict partial order on abstract ‘events’ called *happens before*

Our contributions: abstract convergence

In another component, we:

Parameterise our work with a strict partial order on abstract ‘events’ called *happens before*

Parameterise by a method of lifting ‘events’ to ‘state transformers’

Our contributions: abstract convergence

In another component, we:

Parameterise our work with a strict partial order on abstract ‘events’ called *happens before*

Parameterise by a method of lifting ‘events’ to ‘state transformers’

For two lists of events *xs* and *ys*, if we assume:

- Concurrent events in both *xs* and *ys* commute
- All events in *xs* and *ys* respect the happens before relation

Our contributions: abstract convergence

In another component, we:

Parameterise our work with a strict partial order on abstract 'events' called *happens before*

Parameterise by a method of lifting 'events' to 'state transformers'

For two lists of events *xs* and *ys*, if we assume:

- Concurrent events in both *xs* and *ys* commute
- All events in *xs* and *ys* respect the happens before relation

Then: applying the state transformers in both *xs* and *ys* to the same initial state gives the same final state

Our contributions: Replicated Growable Array

In a final component, we:

- Formalise the Replicated Growable Array (RGA) CRDT
- “the reason why the RGA actually works has been a bit of a mystery”
- Show operations commute with themselves, and with each other, under assumptions
- Work in terms of insert/delete messages with concrete elements to insert into array

Composing components

We compose:

- Our network model has an associated concrete *happens before* relation
- Given in terms of broadcast and deliver events
- This relation is a strict partial order

Composing components

We compose:

- Our network model has an associated concrete *happens before* relation
- Given in terms of broadcast and deliver events
- This relation is a strict partial order

We can then replace the parameter in our abstract theorem with this relation

Composing components

As a corollary of abstract theorem, obtain a concrete convergence theorem for CRDT

Composing components

As a corollary of abstract theorem, obtain a concrete convergence theorem for CRDT

Assuming two finite lists of deliver events, with insert/delete messages

Composing components

As a corollary of abstract theorem, obtain a concrete convergence theorem for CRDT

Assuming two finite lists of deliver events, with insert/delete messages

Such that the messages delivered are the same (but not necessarily in same order)

Composing components

As a corollary of abstract theorem, obtain a concrete convergence theorem for CRDT

Assuming two finite lists of deliver events, with insert/delete messages

Such that the messages delivered are the same (but not necessarily in same order)

Then applying these insert/delete operations to an initial state yields the same list

Composing components

As a corollary of abstract theorem, obtain a concrete convergence theorem for CRDT

Assuming two finite lists of deliver events, with insert/delete messages

Such that the messages delivered are the same (but not necessarily in same order)

Then applying these insert/delete operations to an initial state yields the same list

Convergence, from definition of strong eventual consistency

A note

Abstract convergence theorem does not mention:

- Network,
- Concrete CRDT implementation

A note

Abstract convergence theorem does not mention:

- Network,
- Concrete CRDT implementation

Purely a theorem about orders, and lists of ordered elements

Claim: this theorem is ‘essence’ of CRDT convergence

A note

Abstract convergence theorem does not mention:

- Network,
- Concrete CRDT implementation

Purely a theorem about orders, and lists of ordered elements

Claim: this theorem is ‘essence’ of CRDT convergence

If true, other CRDTs should be verifiable using it

A note

Abstract convergence theorem does not mention:

- Network,
- Concrete CRDT implementation

Purely a theorem about orders, and lists of ordered elements

Claim: this theorem is ‘essence’ of CRDT convergence

If true, other CRDTs should be verifiable using it

1. Observed-removed set
2. Distributed increment-decrement counter

Conclusions

Distributed algorithms hard to verify

Many mistakes in previous proofs of distributed algorithms

Even those checked by machine!

Conclusions

Distributed algorithms hard to verify

Many mistakes in previous proofs of distributed algorithms

Even those checked by machine!

Must use end-to-end verification for confidence

Conclusions

Distributed algorithms hard to verify

Many mistakes in previous proofs of distributed algorithms

Even those checked by machine!

Must use end-to-end verification for confidence

Developed a framework for end-to-end verification of CRDTs

Verified RGA, ORSet, and counter CRDT implementations as examples

Our network axioms

Write $hist_i$ for local (totally ordered) history of node i

Write $m \sqsubset^i n$ for local message ordering at node i

Write mid_m for message identifier of message m

All events are either a *Broadcast* or a *Delivery* of message

Delivery has a cause

$$\mathit{Deliver}\ m \in \mathit{hist}_i \longrightarrow \exists j.\ \mathit{Broadcast}\ m \in \mathit{hist}_j$$

Deliver locally

$$\mathit{Broadcast}\ m \in \mathit{hist}_i \longrightarrow \mathit{Broadcast}\ m \sqsubset^i \mathit{Deliver}\ m$$

Message identifiers unique

$$Broadcast\ m \in hist_i \wedge$$
$$Broadcast\ n \in hist_j \wedge$$
$$mid_m = mid_n \longrightarrow m = n \wedge i = j$$

Causal networks

Write *hb m n* for happens before relation, defined by:

$$\frac{\mathit{Broadcast}\ m \sqsubset^{i} \mathit{Deliver}\ n}{\mathit{hb}\ m\ n}$$

$$\frac{\mathit{Deliver}\ m \sqsubset^{i} \mathit{Broadcast}\ n}{\mathit{hb}\ m\ n}$$

$$\frac{\mathit{hb}\ m\ n \quad \mathit{hb}\ n\ o}{\mathit{hb}\ m\ o}$$

Causality

$$\mathit{Deliver}\ m \in \mathit{hist}_j \wedge \mathit{hb}\ n\ m \longrightarrow \mathit{Deliver}\ n \sqsubset^j \mathit{Deliver}\ m$$