Verifying Distributed Programs via Canonical Sequentialization

Klaus von Gleissenthall
Joint work with
Alexander Bakst, Ranjit Jhala and Rami Gökhan Kıçı
Writing distributed programs

A bug appears…

Issue: random hangs / deadlock in mono
Writing distributed programs

... haunts you ...

Issue:
random hangs /
deadlock in
mono

occurs in about
10% of our runs
Writing distributed programs

... then you write some more code...

moved to version 4.8.0.483.
Writing distributed programs

... and the bug disappears...

moved to version 4.8.0.483.

yet to reproduce the issue in 4.8.0.483
Writing distributed programs

...leaving you hoping it stays gone.

moved to version 4.8.0.483.

yet to reproduce the issue in 4.8.0.483

should be more confident in a few weeks
A better world

Can we catch all deadlocks during compile-edit cycle?
A better world

let's fix it

unmatched receive

unmatched send

sent wrong response address

check

coord :: Transaction -> Int -> SymSet ProcessId -> Process ()
coord transaction n nodes = do
  fold query () nodes
  n_ <- fold countVotes 0 nodes
  if n == n_
    then
      forEach nodes commit ()
    else
      forEach nodes abort ()
      forEach nodes expect :: Ack

query () pid = do { me <- myPid; send pid (pid, transaction) }

countVotes init nodes = do
  msg <- expect :: Vote
  case msg of
    Accept _ -> return (x + 1)
    Reject _ -> return x

acceptor :: Process ()
acceptor = do
  me <- myPid
  (who, transaction) <- expect :: (ProcessId, Transaction)
  vote <- chooseVote transaction
  send who vote
coord :: Transaction -> Int -> SymSet ProcessId -> Process ()
coord transaction n nodes = do
    fold query () nodes
    n_ <- fold countVotes 0 nodes
    if n == n_ then
        foreach nodes commit ()
    else
        foreach nodes abort ()
        foreach nodes expect :: Ack

    where
        query () pid = do { me <- myPid; send pid (me, transaction) }
        countVotes init nodes = do
            msg <- expect :: Vote
            case msg of
                Accept _ -> return (x + 1)
                Reject  -> return x

acceptor :: Process ()
acceptor = do
    me <- myPid
    (who, transaction) <- expect :: (ProcessId, Transaction)
vote <- chooseVote transaction
send who vote

A better world

proof
No deadlocks can occur!
This talk: Brisk

- Proves absence of deadlocks
- Provides counterexamples
- Fast enough for interactive use
- Restricted computation model
Restricted computation model

But Expressive Enough to Implement:

- Work Stealing
- Map Reduce
- Distributed File System
Outline

The Problems
The Key Idea
The Implementation
The Evaluation
The Problems
Example: Two phase commit (2PC)

Goal: Commit Transaction to all nodes
Example: Two phase commit (2PC)

Phase 1

data

depending on the value, votes to commit or abort

sends data
Example: Two phase commit (2PC)

Phase 1

- **commits** if no one voted to abort
- **aborts** otherwise

Depending on the value, votes to **commit** or **abort**

Commit

Commit

Commit

Commit
Example: Two phase commit (2PC)

Phase 2

sends decision to commit (or abort)
Example: Two phase commit (2PC)

Phase 2

- ACK
- send acknowledgement
- done
How to verify 2PC?

```haskell
coord :: Transaction -> Int -> SymSet ProcessId -> Process ()
coord transaction n nodes = do
  fold () query nodes
  n_ <- fold 0 countVotes nodes
  where
    query () pid = do { me <- myPid; send pid (me, transaction) }
    countVotes c _ = do
      msg <- expect :: Vote
      case msg of
        Accept _ -> return (c + 1)
        Reject _ -> return c

acceptor :: Process ()
acceptor = do
  me <- myPid
  (who, transaction) <- expect :: (ProcessId, Transaction)
  vote <- chooseVote transaction
  send who vote
```

Does Implementation Deadlock?
How to verify 2PC?
How to verify 2PC?

Problem: Asynchrony

- Messages may travel at different speeds
- Processes execute at different speeds
- Races trigger different behaviors
How to verify 2PC?

Problem: Unbounded Processes

don’t know how many nodes at runtime
How to verify 2PC?

Testing?
No guarantees

Proofs?
High user burden

Model checking…?
Infinite number of states
Outline

The Problems
The Key Idea
The Implementation
The Evaluation
Outline

The Problems
The Key Idea
The Implementation
The Evaluation
The Key Idea
Canonical Sequentialization
Canonical Sequentialization

Don’t enumerate execution orders…

… Reason about single representative execution
Canonical Sequentialization

Example 2PC

1. Sends transaction it wants to commit
2. Send votes
3. Relay decision
4. Send acknowledgments
Canonical Sequentialization

A Trickier Example

Work stealing queue
Work stealing queue

- Queue assigns work
- Workers perform tasks
- Coordinator collects results
idle workers ask for work
queue assigns an item

queue sends result to the coordinator
compute results

Work stealing queue
Sequentialized

for each item

queue assigns task to arbitrary worker

who computes result

writes to result set

arbitrary worker picks result from set

sends it to master

for each item
How can sequentialization help verify programs?
How can sequentialization help verify programs?

- no sequentialization means likely wrong
- implies deadlock freedom
- compute its canonical sequentialization
- same halting states
- use to prove additional properties
- on simpler, sequential program
Outline

The Problems
The Key Idea
The Implementation
The Evaluation
Outline

The Problems
The Key Idea
The Implementation
The Evaluation
The Implementation
The Implementation

1. Restrict Computation Model

2. Sequentialize by Rewriting
1. Restrict Computation Model

Symmetric Nondeterminism

Races yield equivalent outcomes
Symmetric Nondeterminism

Example: Phase 1 of 2PC

coordinator sends transaction

no race
Symmetric Nondeterminism

Example: Phase 1 of 2PC

Race

Send vote

commit

commit

commit

same outcome?

processes are symmetric
Symmetry means **invariance** under transformation.

- Look at from above
- Invariant under rotation
- Not this one
Symmetry
In Distributed Systems

Permuting Process Identifiers
Yields equivalent halting states

[Norris and Dill 1996]
Symmetry
Example: Phase 1 of 2PC

Permuting n1 and n2
equivalent halting states

Name the processes
Symmetric Nondeterminism

Example: Phase 1 of 2PC

pick \( n_1 \)

choose between picking \( n_1 \) and \( n_2 \)

did we lose any states?

Example: Phase 1 of 2PC

\((\text{commit}\!, n_1)\)
Symmetric Nondeterminism

Example: Phase 1 of 2PC

- No!
- if we pick n2
- we can permute ids
- to end up in same state
- so the states have the same behavior

(commit, n1)

(commit, n2)

(commit, n3)
How can we use symmetry to sequentialize?
Symmetric Nondeterminism

Example: Phase 1 of 2PC

coordinator sends transaction

no race

receive directly after sending

[Lipton75]
Symmetric Nondeterminism

Example: Phase 1 of 2PC
Symmetric Nondeterminism

Example: Phase 1 of 2PC

- processes are symmetric
- equivalent outcomes
- pick any!
- what now?
- Race

Send vote

Front:
- commit
- commit
- commit
Symmetric Nondeterminism

Example: Phase 1 of 2PC
The Implementation

1. Restrict Computation Model

2. Sequentialize by Rewriting
2. Sequentialize by Rewriting

(by example)
2. Sequentialize by Rewriting

Example 1

\[ \text{send } q \text{ ping } \]
\[ w \leftarrow \text{recv } q ; \]
\[ \text{send } p \text{ pong } \]
\[ v \leftarrow \text{ping } ; \quad q \]

\( p, q \) are in parallel
2. Sequentialize by Rewriting

Example 1

w <- recv q  \parallel  \textcolor{red}{\textbf{send} p \textbf{pong}}

\[ w \leftarrow \text{recv } q, \quad \text{send } p \text{ pong} \]

\[ v \leftarrow \text{ping }; \quad w \leftarrow \text{pong} \]

\textcolor{blue}{p, q \text{ are in parallel}}
2. Sequentialize by Rewriting

Example 2

\[
\text{p, } \text{qs}=\{q_1 \ldots q_n\} \text{ are in parallel}
\]
2. Sequentialize by Rewriting

Example 2

\[ \text{for } q \text{ in } qs \text{ do} \]
\[ \text{send } q \text{ ping} \]
\[ w \leftarrow \text{recv } q \]
\[ \text{end} \]

\[ \prod_{q \in qs} \]
\[ v \leftarrow \text{recv } p; \]
\[ \text{send } p \text{ pong} \]
\[ \text{end} \]

\[ v \leftarrow \text{ping }; \]
\[ w \leftarrow \text{pong } q \]
\[ \text{end} \]

\( p, \) qs=\{q1…qn\} are in parallel
2. Sequentialize by Rewriting

Example 3

\[
\prod_{q \in qs}
\]

\[
\forall q \in qs
do
send q ping
end
\]

\[
\forall q \in qs
do
w <- recv qs
end
\]

\[
v <- recv p;
send p pong
\]
2. Sequentialize by Rewriting

Example 3

for q in qs do
    send q ping
end

for q in qs do
    w <- recv qs
end

\[\prod_{q \in qs} q\]

\[v <- \text{recv } p;\]

\[\text{send } p \text{ pong}\]

for q in qs do
    v <- ping;
end

for q in qs do
    v <- ping;
end
2. Sequentialize by Rewriting

Example 3
The Implementation

1. Restrict Computation Model

2. Sequentialize by Rewriting
Outline

The Problems
The Key Idea
The Implementation
The Evaluation
Outline

The Problems
The Key Idea
The Implementation
The Evaluation
The Evaluation
The Evaluation

Implemented in a Haskell library.

Brisk computes canonical sequentialization.

Communication primitives like send / receive / foreach.

Provides counterexample to sequentialization.
The Evaluation

<table>
<thead>
<tr>
<th>Name</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConcDB</td>
<td>20</td>
</tr>
<tr>
<td>DistDB</td>
<td>20</td>
</tr>
<tr>
<td>Firewall</td>
<td>30</td>
</tr>
<tr>
<td>LockServer</td>
<td>30</td>
</tr>
<tr>
<td>MapReduce</td>
<td>30</td>
</tr>
<tr>
<td>Parikh</td>
<td>20</td>
</tr>
<tr>
<td>Registry</td>
<td>30</td>
</tr>
<tr>
<td>TwoBuyers</td>
<td>20</td>
</tr>
<tr>
<td>2PC</td>
<td>50</td>
</tr>
<tr>
<td>WorkSteal</td>
<td>40</td>
</tr>
<tr>
<td>Theque</td>
<td>100</td>
</tr>
</tbody>
</table>

Textbook algorithms

Map/Reduce framework

Variant of DISCO distributed filesystem

fast enough for interactive use
Summary

Reason about **representative sequentialization**

**symmetric races produce equivalent outcomes**

**symmetric races + sequentialization =**

verify deadlock freedom in tens of milliseconds
What’s next

- Faults
- larger program class
- larger class of properties
Backup slides
2PC: Faults

- May go down back up later
- Crash/recover
- Same state
- Back up later

What about faults?

Just more asynchrony
2PC: Faults

- Agents operate at arbitrary speed, may fail by stopping, and may restart. Since all agents may fail after a value is chosen and then restart, a solution is impossible unless some information can be remembered by an agent that has failed and restarted.
File System

- **Master**:
  - AllocBlob(name)
  - PutBlob(name, data)
  - GetBlob(name)
  - AddTag(tag, refs)
  - GetTag(tag)

- **Tag Server**:
  - AddTag(name)
  - GetTag(tag)

- **Data Server**:
  - PutBlob(name, data)
  - GetBlob(name)

**Types of Files**
- **mutable**
- **immutable**
Real consequences

We have tried for 6 months to narrow down … … to no avail.

It is random.

Issue tracker: random hangs / deadlock in mono

It occurs in about 10% of our runs.

https://bugzilla.xamarin.com/show_bug.cgi?id=42665
What happened to the bug?

Normally, 100,000 runs would hang 20% of the runs.

So far, 100,000 runs has produced no hangs.

I should be more confident in a few more weeks (fingers are still crossed right now).

This issue still occurs in mono-4.6.2.16 however we have yet to reproduce the issue in 4.8.0.483.

https://bugzilla.xamarin.com/show_bug.cgi?id=42665
Programmers don’t case split on execution orders.

Our approach:
- Compute sequentialization fast (~20-100 ms)
- Automated proofs

Correct programs often have an equivalent sequentialization.

Our approach: compute sequentialization.
Help writing distributed programs

We want to prove absence of deadlocks quickly while compiling. No manual proofs.
Canonical Sequentialization in Brisk

Our approach: compute canonical sequentialization

- no sequentialization = likely wrong
- existence implies deadlock freedom
- same halting states
- check additional safety properties
- on simpler, sequential program

Our approach: compute canonical sequentialization
## Results

<table>
<thead>
<tr>
<th>Name</th>
<th>#Param</th>
<th>#LOC</th>
<th>SPIN</th>
<th>ICET #Term</th>
<th>Brisk time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>EX2</td>
<td>1</td>
<td>14</td>
<td>-</td>
<td>69</td>
<td>20</td>
</tr>
<tr>
<td>EX3</td>
<td>1</td>
<td>13</td>
<td>11</td>
<td>57</td>
<td>20</td>
</tr>
<tr>
<td>PINGDET</td>
<td>1</td>
<td>17</td>
<td>13</td>
<td>83</td>
<td>20</td>
</tr>
<tr>
<td>PINGITER</td>
<td>1</td>
<td>19</td>
<td>11</td>
<td>63</td>
<td>20</td>
</tr>
<tr>
<td>PINGSYM</td>
<td>1</td>
<td>13</td>
<td>10</td>
<td>44</td>
<td>30</td>
</tr>
<tr>
<td>PINGSYM2</td>
<td>1</td>
<td>43</td>
<td>7</td>
<td>140</td>
<td>30</td>
</tr>
<tr>
<td>CONCDB</td>
<td>1</td>
<td>54</td>
<td>6</td>
<td>265</td>
<td>20</td>
</tr>
<tr>
<td>DistDB</td>
<td>2</td>
<td>42</td>
<td>2</td>
<td>218</td>
<td>20</td>
</tr>
<tr>
<td>FIREWALL</td>
<td>1</td>
<td>45</td>
<td>9</td>
<td>201</td>
<td>30</td>
</tr>
<tr>
<td>LockServer</td>
<td>1</td>
<td>28</td>
<td>12</td>
<td>109</td>
<td>30</td>
</tr>
<tr>
<td>MapReduce</td>
<td>2</td>
<td>64</td>
<td>4</td>
<td>205</td>
<td>30</td>
</tr>
<tr>
<td>Parikh</td>
<td>0</td>
<td>35</td>
<td>-</td>
<td>173</td>
<td>20</td>
</tr>
<tr>
<td>Registry</td>
<td>1</td>
<td>40</td>
<td>10</td>
<td>171</td>
<td>30</td>
</tr>
<tr>
<td>TwoBuyers</td>
<td>0</td>
<td>59</td>
<td>-</td>
<td>332</td>
<td>20</td>
</tr>
<tr>
<td>2PCommit</td>
<td>1</td>
<td>47</td>
<td>6</td>
<td>281</td>
<td>50</td>
</tr>
<tr>
<td>WorkSteal</td>
<td>2</td>
<td>39</td>
<td>5</td>
<td>141</td>
<td>40</td>
</tr>
<tr>
<td>Theque</td>
<td>3</td>
<td>576</td>
<td>3</td>
<td>1443</td>
<td>100</td>
</tr>
</tbody>
</table>

<= 100ms

- **micro benchmarks**
- **Firewall, Map Reduce, 2PC**
- **Distributed file system**
Outline

The Dream

The Problems

The Key Idea

The Implementation

The Evaluation