Pretend Synchrony

Klaus von Gleissenthall

UC San Diego
Software Systems
Shouldn’t Fail
Ariane 5 Crash

Crashed due to float to int conversion bug

1996
Marriott Breach

Data from 500 Mio customers (2018)

Name, Passport No., Credit Card Numbers
The Nightmare
Katie Warren @KayEllDoubleYoo · 5 Oct 2017
@NetflixUK is down! What do I do now? 😞 What kind of life is this?
#netflixdown
The Nightmare
The Nightmare

Hello darkness, my old friend..

I feel so damn lost.

elguitarro 49 points · 5 years ago
ToxicSandwich 2 points · 4 years ago
Software Systems
Shouldn’t Fail
How to Make Sure Distributed Systems Don’t Crash?
Distributed Systems

Nodes run protocol
Send & receive asynchronously
Testing?
Testing?

To fix a run, we need

Input

Schedule
Testing?

Schedule: order on message delivery
Testing?

Schedule: order on message delivery

Schedule 2

![Diagram showing the order of message delivery with nodes A, B, and C connected to a computer.](image-url)
Testing?
Schedule: order on message delivery

Schedule 3

A
B
C

A

B

C
Testing?

Given *input* & *schedule* check property $\Psi$
Asynchrony: too many schedules

Testing?

☹
Testing?

Asynchrony: too many schedules
Model Checking?
Model Checking?

Enumerate all *inputs* & *schedules*
Model Checking?

Problem: Unbounded State
Model Checking

Problem: Unbounded state space

MC
Model Checking?
Problem: Unbounded State
Deductive Verification?
Deductive Verification?

Prove Protocol Correctness

SMT ✅ ❌
Deductive Verification?

Can handle Unbounded State

... but needs Auxiliary Invariants
Deductive Verification?

... but needs Auxiliary Invariants
Deductive Verification?

... but needs Auxiliary Invariants

that enumerate schedules and network state
Deductive Verification?

Too many Invariants!
Pretend Synchrony: Make Proofs Easier!
Pretend Synchrony

Programmers *don’t* case-split on *schedules* & *network*
Pretend Synchrony
... they think about a representative schedule
Pretend Synchrony

... where messages are delivered instantly
Pretend Synchrony
we call this schedule *synchronization*
Pretend Synchrony

To verify a protocol

1.

2.
Pretend Synchrony

To verify a protocol

1. compute its synchronization
Pretend Synchrony

To verify a protocol

... 2. verify synchronization
Pretend Synchrony

Synchronizations don’t case-split on schedules & network
Pretend Synchrony

Synchronizations don’t case-split on schedules & network

\[
\begin{array}{c}
\Psi \\
θ_1 \quad θ_2
\end{array}
\]

... which significantly reduces invariants
Synchronizations

Example 1: Two-Phase Commit
Two-Phase Commit

Phase 1

Coordinator

Commit a value

Storage node

Storage node

Storage node
Two-Phase Commit

Phase 1

Send Value

Save Value

Save Value

Save Value
Two-Phase Commit

Phase 1

Use value to commit or abort

Abort if any node aborts
Two-Phase Commit

Phase 1

Use value to commit or abort

Commit if all nodes commit
Two-Phase Commit

Phase 2

If

commit finalize

Send
commit/abort decision
Two-Phase Commit

Phase 2

Send acks

Receive acks; done
Two-Phase Commit

Send value to nodes

Respond commit/abort

Relay decision

Gather acks
Synchronizations

Example 2: Work stealing Queue
Work stealing Queue

Queue

Workers

Workers

Workers

Collector
Work stealing Queue

Worker requests a task
Work stealing Queue

Assigns task to worker
Work stealing Queue

Other workers request task

[Diagram showing a work stealing queue with computer and database symbols]
Work stealing Queue

Other workers request task
Work stealing Queue

Sends result to coordinator
Work stealing Queue

The other workers finish
Work stealing Queue

Done!
Work stealing Queue

Synchronization

1. **Queue** assigns tasks to workers that write result to set

2. **Workers** pick results from set and send to collector
Outline

Key Idea: Pretend Synchrony

1. Computing Synchronizations
2. Verifying the Synchronization

Extensions
Evaluation
1. Computing Synchronizations

(By Rewriting and By Example)
Synchronize by Rewriting

Example 1: Loop Free
Synchronize by Rewriting

Example 1
Synchronize by Rewriting

Example 1

\[
\begin{align*}
\text{send } q \text{ Ping;} & \quad v \gets \text{recv } p; \\
w \gets \text{recv } q; & \quad \text{send } p \text{ Pong;}
\end{align*}
\]
Synchronize by Rewriting

Example 1

Since there is only a single order

\[
\begin{align*}
\text{send } q \text{ Ping}; & \quad v \gets \text{recv } p; \\
\text{w } \gets \text{recv } q; & \quad \text{send } p \text{ Pong};
\end{align*}
\]
Synchronize by Rewriting

Example 1

Since there is only a single order

```
send q Ping;

v <- recv p;

w <- recv q; || send p Pong;
```

... we can \textit{sequentialize} the \textit{send} & \textit{receive} (Lipton’75)
Synchronize by Rewriting

Example 1

... we can **sequentialize** the send & receive (Lipton’75)

```
send q Ping;
v <- recv p;
w <- recv q;  ||  send p Pong;
```
Synchronize by Rewriting

Example 1

... we can *sequentialize* the send & receive (Lipton’75)

\[
\begin{align*}
q.v & \leftarrow \text{Ping}; \\
\text{w} & \leftarrow \text{recv } q; \parallel \text{send } p \text{ Pong};
\end{align*}
\]

... and replace them by an *assignments*
Synchronize by Rewriting

Example 1

… we can *sequentialize the send & receive* (Lipton’75)

q.v <- Ping;
send p Pong;
w <- recv q;

… and replace them by an *assignments*
Synchronize by Rewriting

Example 1

… we can sequentialize the send & receive (Lipton’75) …

\begin{verbatim}
q.v <- Ping;
p.w <- Pong;
\end{verbatim}

… and replace them by an assignments
Synchronize by Rewriting

Example 1

Synchronization

q.v <- Ping;
p.w <- Pong;
Synchronize by Rewriting

Example 2: Loop over Processes
Synchronize by Rewriting

Example 2
Synchronize by Rewriting

Example 2
Synchronize by Rewriting

Example 2

\[
\prod_{q \in qs} \begin{array}{l}
\text{for } q \text{ in } qs \text{ do} \\
\quad \text{send } q \text{ Ping;} \\
\quad w \leftarrow \text{recv } q; \\
\quad \text{end}
\end{array} \quad \parallel \quad \begin{array}{l}
\forall \ v \leftarrow \text{recv } p; \\
\quad \text{send } p \text{ Pong;}
\end{array}
\]

\( p \) loops over \( qs \)
Synchronize by Rewriting

Example 2

Since iterations are *sequential*

\[
\text{for } q \text{ in } qs \text{ do}
\]

\[
\begin{align*}
&\text{send } q \text{ Ping;} \\
&w \leftarrow \text{recv } q; \\
\end{align*}
\]

\[
\text{end}
\]

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

\[
\begin{align*}
&v \leftarrow \text{recv } p; \\
&\text{send } p \text{ Pong;} \\
\end{align*}
\]

... and each iteration talks to *a single process*
Synchronize by Rewriting

Example 2

... focus on arbitrary iteration

\[
\begin{align*}
\text{for } q \text{ in } qs & \text{ do} \\
\text{send } q \text{ Ping;} \\
w & \leftarrow \text{recv } q; \\
\text{end} \\
\prod_{q \in qs} \text{ v } & \leftarrow \text{recv } p; \\
\text{send } p \text{ Pong;}
\end{align*}
\]
Synchronize by Rewriting

Example 2

… focus on arbitrary iteration, synchronize

```plaintext
for q in qs do
    q.v <- Ping;
    p.w <- Pong;
end
```
Synchronize by Rewriting

Example 2

… focus on arbitrary iteration, synchronize

```
for q in qs do
  q.v <- Ping;
  p.w <- Pong;
end
```

… and generalize (Materialization, Sagiv’99)
Synchronize by Rewriting

Example 2

for q in qs do
    q.v <- Ping;
    p.w <- Pong;
end

Synchronization
Synchronize by Rewriting

Example 3: Symmetric Races
Synchronize by Rewriting

Example 3
Synchronize by Rewriting

Example 3

two loops

\[
\begin{align*}
\text{for } q \text{ in } qs & \text{ do} \\
& \text{send } q \text{ Ping;} \\
& \text{end}
\end{align*}
\]

\[
\begin{align*}
\text{for } _\_ \text{ in } qs & \text{ do} \\
& w \leftarrow \text{recv } qs; \\
& \text{end}
\end{align*}
\]

\[
\begin{align*}
\Pi_{q \in qs} & \ \\
v & \leftarrow \text{recv } p; \\
& \text{send } p \text{ Pong;}
\end{align*}
\]
Synchronize by Rewriting

Example 3

Split the rewrite into two *sequential* steps

\[
\begin{align*}
\text{for } q \text{ in } qs & \text{ do} \\
& \text{send } q \text{ Ping}; \\
\text{end} \\
\end{align*}
\]

\[
\begin{align*}
\prod_{q \in qs} v & \leftarrow \text{recv } p; \\
\end{align*}
\]

\[
\begin{align*}
\text{for } w \text{ in } qs & \text{ do} \\
& \text{w }\leftarrow \text{recv } qs; \\
\text{end} \\
\end{align*}
\]

\[
\begin{align*}
\end{align*}
\]

... the first loop and its *receive*, then the rest
Synchronize by Rewriting

Example 3

... some $qs$ might send during the first loop

$$\prod_{q \in qs} v \leftarrow \text{recv } p;$$

we can pretend the sends happen later (Lipton’75)
Synchronize by Rewriting

Example 3

Since the loop is \textit{sequential}

\[
\begin{align*}
\text{for } q \in qs & \text{ do} \\
\text{send } q \text{ Ping; } \\
\text{end} \\
\text{for } _\_ \text{ in qs do} \\
\text{w }\leftarrow \text{recv } qs; \\
\text{end} \\
\prod_{q \in qs} v & \leftarrow \text{recv } p; \\
\text{send } p \text{ Pong; }
\end{align*}
\]
Synchronize by Rewriting

Example 3

Since the loop is *sequential*

... synchronize arbitrary an iteration
Synchronize by Rewriting

Example 3

Since the loop is *sequential*

\[
\text{for } q \text{ in } qs \text{ do} \\
q.v \leftarrow \text{Ping}; \\
\text{end}
\]

\[
\begin{align*}
\text{for } \_ \text{ in } qs \text{ do} & \quad \| \quad \Pi_{q \in qs} \\
w \leftarrow \text{recv } qs; & \\
\text{end} \\
\text{send } p \text{ Pong;}
\end{align*}
\]

... and generalize
Synchronize by Rewriting

Example 3

Problem: Iterations are no longer *sequential*

... there is a *race* between the processes in $qs$
Synchronize by Rewriting

Example 3

Problem: Iterations are no longer sequential

... how can we synchronize?

\[
\begin{align*}
\text{for } _\_ \text{ in } qs & \text{ do} \\
\text{w } & \leftarrow \text{ recv } qs; \\
\text{end} \\
\prod_{q \in qs} & \text{ send } p \text{ Pong;}
\end{align*}
\]
Synchronize by Rewriting

Exploit

Sequence

Symmetry
Synchronize by Rewriting

*Exploit*

**Sequence**

**Symmetry**
Symmetry

Invariance under transformation
Symmetry

Invariance under *transformation*
Symmetry

Invariance under rotation
Symmetry

Invariance under rotation
Symmetry in Distributed Systems

Invariance under process-id permutation

... doesn’t affect halting states (Ip & Dill 1996)
Symmetry in Distributed Systems

Name nodes ...
Symmetry in Distributed Systems

permuting process names

... does not affect halting states
Use **Symmetry** to Synchronize!
Synchronize by Rewriting

Example 3

Since the processes in $qs$ are symmetric

\[
\begin{align*}
&\text{for } _- \text{ in } qs \text{ do} \\
&w \leftarrow \text{recv } qs; \\
&\text{end} \\
\end{align*}
\]

\[
\begin{align*}
&\prod_{q \in qs} \text{send } p \text{ Pong};
\end{align*}
\]

... we can receive the messages in any order
Synchronize by Rewriting

Example 3

... we can receive the messages in any order

\[
\begin{align*}
\text{for } _ & \text{ in qs do} \\
w & \gets \text{recv } qs; \\
\text{end} \\
\prod_{q\in qs} & \quad \text{send } p \ Pong;
\end{align*}
\]
Synchronize by Rewriting

Example 3

... we can receive the messages in *any* order

\[
\text{for } q \text{ in } qs \text{ do } \\
\quad w \leftarrow \text{recv } q; \\
\text{end}
\]

\[
\prod_{q \in qs} \text{send } p \text{ Pong;}
\]

... in particular, we the *iteration order* of the loop
Synchronize by Rewriting

Example 3

... in particular, we the iteration order of the loop

\[
\text{for } q \text{ in } qs \text{ do } \quad \Pi_{q \in qs} \quad \text{send } p \text{ Pong;}
\]

... since the loop is now \textit{sequential}
Synchronize by Rewriting

Example 3

... since the loop is now \textit{sequential}

\[
\begin{align*}
\text{for } q \text{ in } qs & \text{ do} \\
& w \leftarrow \text{recv } q; \\
& \text{end}
\end{align*}
\]

\[\prod_{q \in qs} \text{send } p \text{ Pong;}\]
Synchronize by Rewriting

Example 3

... since the loop is now \textit{sequential}

\begin{verbatim}
for _ in qs do
  p.w <- Pong;
end
\end{verbatim}

... we can synchronize, as before
Synchronize by Rewriting

Example 3

... since the loop is now *sequential*

```haskell
for _ in qs do
  p.w <- Pong;
end
```

... we can synchronize, as before
Synchronize by Rewriting

Example 3

... we can synchronize, as before

```
for _ in qs do
  p.w <- Pong;
end
```
Synchronize by Rewriting

Example 3

... we can synchronize, as before

```plaintext
for q in qs do
    q.v <- Ping;
end

for _ in qs do
    p.w <- Pong;
end
```

... and get the overall synchronization
Synchronize by Rewriting

Example 3

Synchronization

```
for q in qs do
  q.v <- Ping;
end

for _ in qs do
  p.w <- Pong;
end
```
Synchronize by Rewriting

Example 4: Two Phase Commit
for $p$ in $\text{dbs}$ do
    send $p$ ($c, val$)

abort <- False

for $p$ in $\text{dbs}$ do
    msg <- recv
    if msg == $A$
        abort <- True

$(id, val)$ <- recv

$\prod_{p \in \text{dbs}}$

vote <- * ? $C : A$

send id vote
Synchronize by Rewriting

Two Phase Commit: Phase 2

dec <- abort ? A : C

for p in dbs do
send p dec

for p in dbs do
_ <- recv

\[ \prod_{p \in \text{dbs}} \]

\[
\begin{align*}
\text{dec} & \leftarrow \text{recv} \\
\text{if } \text{dec} = C \\
\text{value} & \leftarrow \text{val} \\
\text{send id Ack}\end{align*}
\]
Synchronize by Rewriting

Two Phase Commit: Synchronizing Phase 1

```
for p in dbs do
    send p (c,val)

abort <- False

for p in dbs do
    msg <- recv
    if msg == A
        abort <- True
```

```
(id,val) <- recv
vote <- * ? C : A
send id vote

Π_{p ∈ dbs} ...
```

... by example 3
Synchronize by Rewriting

Two Phase Commit: Synchronizing Phase 1

\[
\begin{align*}
&\text{abort} \gets \text{False} \\
&\text{for } p \text{ in } \text{dbs} \text{ do} \\
&\quad p.\text{id} \gets c; \\
&\quad p.\text{val} \gets c.\text{val}; \\
&\text{vote} \gets * \text{ ? C : A} \\
&\text{send} \text{ id vote}
\end{align*}
\]

... by example 3
Synchronize by Rewriting

Two Phase Commit: Synchronizing Phase 1

\[
\begin{align*}
\text{for } p \text{ in } \text{dbs} \text{ do} \\
p.\text{id} &\leftarrow c; \\
p.\text{val} &\leftarrow c.\text{val}; \\
c.\text{abort} &\leftarrow \text{False}; \\
\end{align*}
\]

\[
\begin{align*}
\text{for } p \text{ in } \text{dbs} \text{ do} \\
\text{msg} &\leftarrow \text{recv} \\
\text{if } \text{msg} &\text{ == A} \\
\text{abort} &\leftarrow \text{True} \\
\end{align*}
\]

\[
\begin{align*}
\text{vote} &\leftarrow * \text{ ? C : A} \\
\text{send id vote} \\
\end{align*}
\]

... by example 3
Synchronize by Rewriting

```
for p in dbs do
  p.id<-c;
  p.val<-c.val;
  c.abort<-False;
for p in dbs do
  vote <-* ? C : A
  c.msg<-p.vote;
  if msg == A
    abort <- True
```

Synchronized Phase 1
Outline

Key Idea: Pretend Synchrony

1. Computing Synchronizations

2. Verifying the Synchronization

Extensions

Evaluation
2. Verifying the Synchronization

Synchronous Proofs Are Easy!
2PC: Correctness

\[ \Psi = \text{Nodes agree on same value} \]
Asynchronous Proofs are *Ugly!*
Asynchronous Proofs
Asynchronous Proofs

Enumerate \textit{schedules} and \textit{network state}
Asynchronous Proofs

Enumerate schedules and network state

\[ \Psi \]

\[ \theta_1 \quad \theta_2 \]

\[ \Phi_1 \quad \Phi_2 \quad \Phi_3 \quad \Phi_4 \quad \Phi_5 \quad \Phi_6 \quad \Phi_7 \quad \Phi_8 \quad \Phi_9 \]

\[ \text{did send to} \]
Asynchronous Proofs

Enumerate schedules and network state

\[ \Psi \]

\[ \theta_1 \quad \theta_2 \]

\[ \Phi_1 \quad \Phi_2 \]

\[ \Phi_4 \quad \Phi_6 \]

\[ \Phi_7 \quad \Phi_8 \quad \Phi_9 \]

\[ \Phi_3 \quad \text{did send to} \quad \text{database} \]

\[ \lor \]

\[ \Phi_5 \quad \text{didn’t execute its receive} \]

☹
Asynchronous Proofs

Enumerate schedules and network state

∀ Φ₃, did send to

∀ Φ₅, didn’t execute its receive

∀ Φ₆, there is messages from to containing ’s ID and value
Synchronous Proofs are Nice!
Asynchronous Proofs
Synchronous Proofs

\[ \Psi \theta_1 \theta_2 \]
Synchronous Proofs

\[ \Psi_{\theta_1 \theta_2} \]
Synchronous Proofs

\[ \theta_1 = \forall p \in \text{dbs} . \ p \in \text{done} \rightarrow p . \text{val} = c . \text{val} \]
Synchronous Proofs

$\theta_2 = \forall p \in dbs. p \in done \land c.dec = C \Rightarrow p.value = c.val$
Recap

Compute synchronizations using *sequence* and *symmetry*
Recap

Makes Deductive Proofs Easier
Recap

Makes Deductive Proofs Easier

\[ \Psi \]

\[ \theta_1 \theta_2 \]

😊
Outline

Key Idea: Pretend Synchrony
1. Computing Synchronizations
2. Verifying the Synchronization

Extensions
Evaluation
Extensions

Multicasts

Message Drops

Rounds
Multicasts
Multicasts

\[ \prod_{p \in \mathcal{P}} \prod_{q \in \mathcal{Q}} \]
Multicasts

Problem: neither sequential nor symmetric

\[
\prod_{p \in ps} \text{send } q \ (\text{Ping}, p); \\
\text{w <- recv } q; \\
\text{end}
\]

\[
\prod_{q \in qs} \text{for } q \text{ in } qs \text{ do} \\
\text{send } q \ (\text{Ping}, p); \\
\text{w <- recv } q; \\
\text{end}
\]

\[
\prod_{v, id \in ps} \text{for } _, \text{ in } ps \text{ do} \\
(v, id) \text{ <- recv } ps; \\
\text{send id Pong; } \\
\text{end}
\]

… can’t compose in sequence? Compose in parallel!
Multicasts

… can’t compose in sequence? Compose in parallel!

\[
\begin{align*}
\text{for } q \text{ in } qs & \text{ do} \\
\prod_{p \in ps} & \text{send } q \ (\text{Ping}, p); \\
& w \leftarrow \text{recv } q; \\
& \text{end} \\
\end{align*}
\]

\[
\begin{align*}
\text{for } _{} \text{ in } ps & \text{ do} \\
\prod_{q \in qs} & (v, id) \leftarrow \text{recv } ps; \\
& \text{send id Pong;} \\
& \text{end} \\
\end{align*}
\]
Multicasts

... can’t compose \textit{in sequence}? Compose \textit{in parallel}!

\begin{align*}
\text{for } q \text{ in } qs\text{ do} & \quad \text{for } _{\_} \text{ in } ps\text{ do} \\
\text{send } q \text{ (Ping, } p) & \quad (v, id) \leftarrow \text{recv } p \\
w \leftarrow \text{recv } q & \\
\text{send } id \text{ Pong} & \\
\text{end} & \quad \text{end}
\end{align*}

... \textit{focus on arbitrary process } p
Multicasts

... focus on arbitrary process $p$

\[ \prod_{p \in \mathcal{P}} \sum_{q \in \mathcal{Q}_p} \text{send } q \ (\text{Ping}, p) ; \]
\[ w \leftarrow \text{recv } q ; \]
\[ \text{end} \]

\[ \prod_{p \in \mathcal{P}} \sum_{q \in \mathcal{Q}_p} \text{for } _{\_} \text{ in } \mathcal{P} \text{ do} \]
\[ (v, id) \leftarrow \text{recv } p ; \]
\[ \text{send id Pong} ; \]
\[ \text{end} \]

... the interaction is \emph{sequential}!
Multicasts

... the interaction is *sequential*!
Multicasts

... the interaction is *sequential*!

\[
\prod_{p \in ps} (q.v, q.id) \leftarrow (Ping, p); \\
p.w \leftarrow Pong
\]

... synchronize (by example 2)
Multicasts

... synchronize (by example 2)

for q in qs do
  (q.v,q.id)<-(Ping,p);
  p.w<-Pong
end
Multicasts

... synchronize (by example 2)

... and generalize!
Multicasts

... and generalize!

\[
\prod_{p \in ps} (q.v, q.id) \leftarrow (Ping, p);
\]
\[
p.w \leftarrow \text{Pong}
\]
\[
\text{end}
\]

results in a concurrent, shared memory program
Extensions

Multicasts

Message Drops

Rounds
Extensions
Message Drops
Extensions
Message Drops

\[
\begin{align*}
\text{for } q \text{ in } qs \text{ do} \\
& \quad \text{send } q \text{ Ping;} \\
\text{end}
\end{align*}
\]

\[
\prod_{q \in qs} v \leftarrow \text{recvTO } p;
\]

Receive non-deterministically times out
Extensions
Message Drops

The interaction is *sequential*

\[
\text{for } q \text{ in } qs \text{ do }
\begin{align*}
\text{send } q \text{ Ping;}
\end{align*}
\]  
\[
\prod_{q \in qs} v \leftarrow \text{recvT0 } p;
\]
Extensions
Message Drops

The interaction is \textit{sequential}

\begin{align*}
\text{for } q \text{ in } qs \text{ do} & \\
\quad & \text{send } q \text{ Ping; } \parallel \quad \prod_{q \in qs} v \leftarrow \text{recvT0 } p; \\
\text{end}
\end{align*}

… focus on a single iteration
Extensions

Message Drops

... focus on a single iteration

\[
\text{for } q \text{ in } qs \text{ do}
\text{send } q \text{ Ping}; \quad \big\| \quad \Pi_{q \in qs} v \gets \text{recvT0} p;
\text{end}
\]
Extensions
Message Drops

... focus on a single iteration

```
for q in qs do
    send q Ping;
    v <- recvT0 p;
end
```

... match up the send and receive
Extensions
Message Drops

... match up the send and receive

for q in qs do
  send q Ping;
  v <- recvTO p;
end
Extensions
Message Drops

... match up the send and receive

... case-split whether the message was received
Extensions
Message Drops

... case-split whether the message was received
Extensions
Message Drops

... case-split whether the message was received

```
for q in qs do
  q.v <-* ?
    Just Ping:
    None
end
```

... and generalize
Extensions
Message Drops

for q in qs do
  q.v <- *?
  Just Ping:
  None
end

Synchronization
Extensions

Multicasts

Message Drops

Rounds
Extensions

Rounds

Instead of running only once
Extensions

Rounds

... repeat protocol in multiple rounds
... repeat protocol in multiple rounds
Extensions

Rounds

To repeat the protocol (from Ex. 2)

\[
\text{for } q \text{ in } qs \text{ do}
\]
\[
\begin{array}{c}
\text{send } q \text{ Ping} \\
\leftarrow \text{recv } q;
\end{array}
\]
\[
\text{end}
\]

\[
\prod_{q \in qs}
\]
\[
\begin{array}{c}
w \leftarrow \text{recv } p; \\
\text{send } p \text{ Ack;}
\end{array}
\]
Extensions
Rounds
To repeat the protocol (from Ex. 2)

\[
\begin{align*}
\text{for } q \text{ in } qs & \text{ do} \\
\text{send } q (\text{Ping}, r) & \\
_\leftarrow \text{recv } q & \\
\text{end}
\end{align*}
\]
Extensions
Rounds
To repeat the protocol (from Ex. 2)

\[
\text{for } q \in qs \text{ do}
\]

\[
\begin{align*}
&\text{send } q \ (\text{Ping, } r) \\
&\_<- \text{ recv } q;
\end{align*}
\]

\[
\text{end}
\]

\[
(\prod_{q \in qs} (w, r)<-\text{recv } p;)
\]

\[
\text{send } p \ \text{Ack;}
\]

… bind the round number at the receive
Extensions
Rounds
To repeat the protocol (from Ex. 2)

\[
\begin{align*}
\text{send } q \text{ (Ping, } r) \\
\text{\_\_\_\_ recv } q; \\
\text{end}
\end{align*}
\]

\[
\begin{align*}
\prod_{q \in qs} (w[r], r) & \gets \text{recv } p; \\
\text{send } p \text{ Ack;}
\end{align*}
\]

... and turn \( w \) into an array
Extensions

Rounds

To repeat the protocol (from Ex. 2)

\[
\begin{align*}
\text{for } q \text{ in } q_s \text{ do} \\
\text{send } q \text{ (Ping, } r) \\
\text{ } \leftarrow \text{recv } q; \\
\text{end}
\end{align*}
\]

\[
\begin{align*}
\prod_{q \in q_s} (w[r], r) & \leftarrow \text{recv } p; \\
\text{send } p \ r \ \text{Ack;}
\end{align*}
\]

\[\text{… reply with a message for round number } r\]
Extensions

Rounds

To repeat the protocol (from Ex. 2)

\[
\prod_{q \in qs} (w[r], r) \leftarrow \text{recv } p;
\]

\[
\text{send } p \ r \ \text{Ack};
\]

... receive for round \( r \)
Extensions

Rounds

To repeat the protocol (from Ex. 2)

```plaintext
for r in rounds do
  for q in qs do
    send q (Ping, r)
    _<- recv q r;
  end
end
```

\[ \prod_{q \in qs} (w[r],r) \leftarrow \text{recv } p; \]

\[ \text{send } p \ r \text{ Ack}; \]

... repeat the for-loop, once for each round
Extensions

Rounds

To repeat the protocol (from Ex. 2)

... and repeat each process q, indefinitely
Extensions

Rounds

Show $\forall r \in \text{rounds}, \forall q \in qs : q . w[r] = Ping$

\[
\begin{align*}
\text{for } r \text{ in rounds do} \\
\quad \text{for } q \text{ in } qs \text{ do} \\
\quad \quad \text{send } q \ (\text{Ping, } r) \\
\quad \quad _{-}\text{recv } q \ r; \\
\quad \text{end} \\
\text{end}
\end{align*}
\]

\[
\begin{align*}
\text{repeat do} \\
\quad \prod_{q \in qs} (w[r], r)\text{-recv } p; \\
\quad \text{send } p \ r \text{ Ack; } \\
\quad \text{end}
\end{align*}
\]
Idea: Round Non-Interference
Idea: Round Non-Interference

No shared state or communication between rounds
Idea: *Round Non-Interference*

Show $\forall r \in \text{rounds} : \phi(r)$ by showing $\phi(r^*)$ for an arbitrary round $r^*$
Extensions

Rounds

Check Round Non-interference via Syntax

\[
\begin{align*}
\text{for } r \text{ in rounds do} \\
\text{for } q \text{ in } q_s \text{ do} \\
\text{send } q \text{ (Ping, r)} \\
\text{(_<- recv } q \text{ r)} \\
\text{end} \\
\text{end}
\end{align*}
\]

\[
\begin{align*}
\text{repeat do} \\
\prod_{q \in q_s} \text{ (w[r], r)<-recv } p; \\
\text{send } p \text{ r Ack; } \\
\text{end}
\end{align*}
\]
Extensions
Rounds

Check Round Non-interference via Syntax

... round id only bound once
Extensions

Rounds

Check Round Non-interference via Syntax

\[ \prod_{q \in qs} \text{for } q \text{ in } qs \text{ do} \]

\[ \_ \text{<- recv } q \_ ; \]

\[ \text{for } r \text{ in rounds do} \]

\[ \text{send } q \text{ (Ping, } r) ; \]

\[ \text{send } p \_ \text{ Ack;} \]

\[ \text{send only for bound round} \]
Extensions
Rounds

Check Round Non-interference via Syntax

\[
\text{for } r \text{ in rounds } \text{do}
\]
\[
\text{for } q \text{ in } qs \text{ do}
\]
\[
\text{send } q \text{ (Ping, r)} \quad \leftarrow \text{recv } q \text{ r;}
\]
\text{end}
\[
\text{end}
\]
\[
\text{repeat do}
\]
\[
\prod_{q \in qs} (w[r], r) \leftarrow \text{recv } p; \quad \text{send } p \text{ r Ack;}
\]
\text{end}

\ldots \text{ receive only for bound round}
Extensions
Rounds

Check Round Non-interference via Syntax

… array indexed only for bound round
Extensions

Rounds

To show $\forall r \in rounds : q \cdot w[r] = Ping$

```
for r in rounds do
  for q in qs do
    send q (Ping, r)
    _<- recv q r;
  end
end
```

```
repeat do
  $\Pi_{q\in qs} (w[r], r)<-recv p;$
  send p r Ack;
end
```
Extensions

Rounds

To show $\forall r \in \text{rounds} : q \cdot w[r] = Ping$

\[
\begin{align*}
\text{for } q \text{ in } qs \text{ do} \\
\quad \text{send } q \left( \text{Ping}, r^* \right) \\
\quad \leftarrow \text{recv } q \ r^*; \\
\text{end}
\end{align*}
\]

\[
\begin{align*}
\prod_{q \in qs} (w[r^*],r^*) \leftarrow \text{recv p;}
\quad \text{send } p \ r^* \text{ Ack;}
\end{align*}
\]

\[\ldots \text{ show } \forall q \in qs : q \cdot w[r^*] = Ping\text{ as before}\]
Evaluation
Evaluation

Brisk

Goolong
Haskell Library → Intermediate Language → Synchronization → Counterexample

checks if a synchronization exists
Brisk: OOPSLA’17

On Github

Haskell Library ➔ Intermediate Language ➔ Synchronization

Counterexample

…through *rewrites* implemented in Prolog
... no deadlocks, spurious sends, etc.
Brisk: OOPSLA’17

<table>
<thead>
<tr>
<th>Name</th>
<th>Time</th>
<th>Brisk</th>
<th>Spin</th>
<th>TO</th>
<th>#n</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConcDB</td>
<td>20 ms</td>
<td></td>
<td></td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>DistDB</td>
<td>20 ms</td>
<td></td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Firewall</td>
<td>30 ms</td>
<td></td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>LockServer</td>
<td>30 ms</td>
<td></td>
<td></td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>MapReduce</td>
<td>30 ms</td>
<td></td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Parikh</td>
<td>20 ms</td>
<td></td>
<td></td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Registry</td>
<td>30 ms</td>
<td></td>
<td></td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>TwoBuyers</td>
<td>20 ms</td>
<td></td>
<td></td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>2PC</td>
<td>50 ms</td>
<td></td>
<td></td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

From the literature

Really fast

Use interactively
<table>
<thead>
<tr>
<th>Name</th>
<th>Time</th>
<th>Brisk</th>
<th>Spin</th>
<th>TO</th>
<th>#n</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map/Reduce</td>
<td>40 ms</td>
<td></td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>Theque</td>
<td>100 ms</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>Filesystem</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Evaluation

Brisk

Goolong
Goolong: POPL’19

Go Library → Intermediate Language → Synchronization → Checker

Counter-example × × ☑️

On Github
Go Library

Declared protocol variables:
\[ x \leftarrow \text{NewVar()} \]
\[ v \leftarrow x.\text{Get()} \]
\[ x.\text{Set}(v) \]

Iteration over sets/invariants:
\[
\{ 
\text{for } q \text{ in } qs @\text{Inv do} \\
\text{send } q \text{ v} \\
\text{w} \leftarrow \text{recv } q;
\}
\]
Goolong: 2PC Phase 1

```plaintext

// Declarations

proposal := gochai.NewVar()
vote := gochai.NewVar()
reply := gochai.NewVar()
abort := gochai.NewVar()
committed := gochai.NewVar()
ack := gochai.NewVar()

committed.Assign(0)
abort.Assign(0)

/*@ invariant: forall((decl(i,int)), implies( and(( elem(i,done) ]), ref(val,i)=proposal) ) -@}*/

for ID := range n.PeerIds {
    n.Send(ID, proposal)
}

for ID := range n.PeerIds {
    vote = n.RecvAll()
    if vote.Get() == 0 {
        abort.Assign(1)
    }
}
```

// Send proposals

// Receive Votes
Goolong: POPL’19

Case-Studies

2PC
Raft Leader Election
Single Decree Paxos
Multi-Paxos KV Store
Goolong: POPL’19

Case-Studies

- If committed, all nodes have same value
- At most one candidate elected leader, per term
- Proposers agree on same value
- Proposers agree on same value, per instance

2PC

Raft Leader Election

Single Decree Paxos

Multi-Paxos KV Store
Goolong: POPL’19

Goolong vs. other verified KVstores

Throughput (req/ms)

Goolong: *not able to run
PSync
Ivy-Raft*
IronKV*

Multi-Paxos KV Store

Goolong vs. other verified KVstores
Does Synchrony Simplify Proofs?
Goolong: POPL’19

Number of Invariants Dafny vs. Goolong

Reduce Invariants by 6x
## Goolong: POPL’19

<table>
<thead>
<tr>
<th>Name</th>
<th>Time Dafny</th>
<th>Time Goolong</th>
</tr>
</thead>
<tbody>
<tr>
<td>2PC</td>
<td>12.8s</td>
<td>0.04s</td>
</tr>
<tr>
<td>Raft</td>
<td>301.6s</td>
<td>0.18s</td>
</tr>
<tr>
<td>Paxos</td>
<td>1141.3s</td>
<td>1.51s</td>
</tr>
<tr>
<td>Total</td>
<td>1455.8s</td>
<td>1.73s</td>
</tr>
</tbody>
</table>

Reduce checking time by 3 orders of magnitude.
Recap: Evaluation

Brisk: Synchronization in ms

Goolong: Go Library

Competitive with verified KV-stores

Reduces Invariants and Checking Time
Inference Rules

R-Send
\[ \Delta, \Sigma \vdash x = q \quad m \text{ fresh} \quad q \text{ is a PID} \]
\[ m' = m \cup \{(p, q, t, r, m)\} \]
\[ \Gamma, \Delta, \Sigma, [send(x, n, t, r)]_p \leadsto \Gamma', (\Delta; [m \leftarrow n]_p), \Sigma, \text{skip} \]

R-Recv
\[ \Delta, \Sigma \vdash x = p \quad (p, q, t, r, m) \in \Gamma \quad p \text{ is a PID} \]
\[ \Gamma' = \Gamma - \{(p, q, t, r, m)\} \]
\[ \Gamma, \Delta, \Sigma, [y \leftarrow recv(t, x, r)]_q \leadsto \Gamma', (\Delta; [y \leftarrow p.m]_q), \Sigma, \text{skip} \]

R-RecvTO
\[ \Gamma, \Delta, \Sigma, [y \leftarrow recvTO(t, x, r)]_q \leadsto \Gamma', (\Delta; [y \leftarrow p.m]_q), \Sigma, \text{skip} \]
\[ \Gamma, \Delta, \Sigma, [y \leftarrow recvTO(t, x, r)]_q \leadsto \Gamma', (\Delta; [y \leftarrow \text{Just p.m}]_q \oplus [y \leftarrow \text{None}]_q), \Sigma, \text{skip} \]

R-Choice
\[ \Gamma, \Delta, \Sigma, A \leadsto \Gamma, (\Delta; \Delta_A), \Sigma, \text{skip} \]
\[ \Gamma, \Delta, \Sigma, B \leadsto \Gamma, (\Delta; \Delta_B), \Sigma, \text{skip} \]
\[ \Gamma, \Delta, \Sigma, A \oplus B \leadsto \Gamma, (\Delta; \Delta_A \oplus \Delta_B), \Sigma, \text{skip} \]

R-False
\[ \Delta \vdash \text{false} \]
\[ \Gamma, \Delta, \Sigma, A \leadsto \Gamma, \Delta', \Sigma', A' \]

R-Context
\[ \Gamma, \Delta, \Sigma, A \circ B \leadsto \Gamma, \Delta', \Sigma', A' \circ B \]

R-Send-Unfold
\[ \Gamma \vdash \text{unfold}(u, x, ps) \quad \Gamma' \triangleq \Gamma - \{\text{unfold}(u, x, ps)\} \]
\[ \Gamma, \Delta, \Sigma, \text{send}(t, x, n) \leadsto \Gamma', (\Delta; \text{assume}(x = u)), \Sigma, \text{send}(t, x, n) \]

R-Recv-Unfold
\[ \Gamma \vdash \text{unfold}(u, x, ps) \quad \Gamma' \triangleq \Gamma - \{\text{unfold}(u, x, ps)\} \]
\[ \Gamma, \Delta, \Sigma, y \leftarrow \text{recv}(ps, t) \leadsto \Gamma', (\Delta; x \leftarrow \text{pick}(ps)), \Sigma', y \leftarrow \text{recv}(u, t) \]

R-Loop
(1) \(u, x\) fresh
(2) \(\Gamma_0 \triangleq \Gamma \cup \{\text{unfold}(u, x, ps)\} \text{ and } \Delta_0 \triangleq \text{assume}(I_C)\)
(3) \(\Delta, \Sigma \vdash I_C \text{ and } (\Delta_0; \langle \Delta^u \rangle), \Sigma \vdash I_C\)
\[ \Gamma_0, \Delta_0, \Sigma, [A]_u \parallel B[x/p] \leadsto \Gamma, (\Delta_0; \Delta^u), \Sigma, \text{skip} \]

\[ \Gamma, \Delta, \Sigma, \prod(p \in ps). [A]_p \parallel \prod(q \in qs). [B]_q \leadsto \Gamma, \Delta, \Sigma' \]

R-Focus
(1) \(u\) fresh
(2) \(\Gamma_0 \triangleq \Gamma \cup \{\text{unfold}(u, x, ps)\} \text{ and } \Delta_0 \triangleq \text{assume}(I_C)\)
(3) \(\Sigma' = (\Sigma \parallel \prod(p \in ps). \Delta^u[p/u])\)
(4) \(\Delta, \Sigma' \vdash I_C \text{ and } (\Delta_0; \Delta^u), \Sigma' \vdash I_C\)
\[ \Gamma_0, \Delta_0, \text{skip}, [A]_u \parallel \prod(q \in qs). [B]_q \leadsto \Gamma_0, (\Delta_0; \Delta^u), \text{skip}, \text{skip} \]

\[ \Gamma, \Delta, \Sigma, \prod(p \in ps). [A]_p \parallel \prod(q \in qs). \text{foreach ps do } B \}_q \leadsto \Gamma, \Delta, \Sigma', \text{skip} \]
Limitations

Structured Loops
Symmetric Non-Determinism
Round Non-Interference
Limitations: Structured Loops

- Loops over sets of processes
- In each iteration, talk to a single process, only
- Easy: transform to broadcast/gather
Limitations: Structured Loops

Loops over sets of processes

Hard: arbitrary loop carried state

Encode as rounds
Limitations

Structured Loops

Symmetric Non-Determinism

Round Non-Interference
Limitations: Symmetric Non-Determinism

Sends have matching receives!

Easy: check with static analysis

Easy: remove “deadlocks”/spurious sends
Limitations:
Symmetric Non-Determinism

- Sends have matching receives!
- Hard: Topologies e.g., Chord, Stoica et al., SIGCOMM '01.
- More inspiration from shape analysis!
Limitations

Structured Loops

Symmetric Non-Determinism

Round Non-Interference
Limitations: Round Non-Interference

- Rounds don’t share messages/state
  - Enough for Multi-Paxos, Raft-Leader Election, Zab
  - Prevents Optimization such as stable leader
  - Algorithms like Stoppable Paxos, 2008
Limitations: Round Non-Interference

Rounds don’t share messages/state

Generalize: HO-model, communication closed layers?
Future Work

Language Restrictions

(Distributed-system folks, really go read that Canonical Sequentialization paper. Big deal.)

12:12 AM · Sep 12, 2017 · TweetDeck
Future Work

Language Restrictions

"Figure out how programmers are currently reasoning, formalize as language restricting validity to that model, profit!" is A+ research plan

Graydon Hoare @graydon_pub • Sep 12, 2017
Yeah it's normal in many contexts, I'm just applauding a particularly pleasing example of it (that Canonical Sequentializatoon paper)
## Goolong: POPL’19

### Invariants

<table>
<thead>
<tr>
<th>Name</th>
<th>#Inv Async</th>
<th>Time Async/Dafny</th>
<th>#Inv Sync</th>
<th>Time Sync</th>
</tr>
</thead>
<tbody>
<tr>
<td>2PC</td>
<td>30</td>
<td>12.8s</td>
<td>3</td>
<td>0.04s</td>
</tr>
<tr>
<td>Raft</td>
<td>50</td>
<td>301.6s</td>
<td>6</td>
<td>0.18s</td>
</tr>
<tr>
<td>Paxos</td>
<td>72</td>
<td>1141.3s</td>
<td>14</td>
<td>1.51s</td>
</tr>
<tr>
<td>Total</td>
<td>152</td>
<td>1455.8s</td>
<td>23</td>
<td>1.73s</td>
</tr>
</tbody>
</table>

Reduce Invariants by 6x

Reduce checking time by 3 orders of magnitude
## Goolong: POPL’19

### Case-Studies

1.5-3x slowdown over unverified KV store

<table>
<thead>
<tr>
<th>System</th>
<th>Throughput (req/ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goolong</td>
<td>118.5</td>
</tr>
<tr>
<td>PSync</td>
<td>32.4</td>
</tr>
<tr>
<td>Ivy-Raft*</td>
<td>13.5</td>
</tr>
<tr>
<td>IronKV*</td>
<td>30</td>
</tr>
</tbody>
</table>

*not able to run

---

Multi-Paxos KV Store
Brisk: OOPSLA’17

Two Phase Commit in Brisk

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
Leftovers
Synchronous Proofs

for p in dbs do
  p.id<-c;
  p.val<-c.val;
  for p in dbs do
    c.abort<-False;
    vote <-* ?
    Commit : Abort
    c.msg<-p.vote;
    if msg == Abort
      abort <- True
for \( p \) in \( \text{dbs} \) do @\texttt{Inv1}

\[
p.\text{id} \leftarrow \text{c};
\]

\[
p.\text{val} \leftarrow \text{c.val};
\]

for \( p \) in \( \text{dbs} \) do @\texttt{Inv2}

\[
c.\text{abort} \leftarrow \text{False};
\]

\[
vote \leftarrow * ?
\]

\[
\text{Commit : Abort}
\]

\[
c.\text{msg} \leftarrow p.\text{vote};
\]

if \( \text{msg} = \text{Abort} \)

\[
\text{abort} \leftarrow \text{True}
\]
Synchronous Proofs

@Inv1=
\forall p \in dbs : p \in done \Rightarrow p.val = c.val

@Inv2 = true

```
for p in dbs do @Inv1
    p.id <- c;
p.val <- c.val;
for p in dbs do @Inv2
    c.abort <- False;
vote <- * ?
    Commit : Abort
    c.msg <- p.vote;
if msg == Abort
    abort <- True
```
Synchronous Proofs

c.dec <- c.abort ?
   Abort : Commit

for p in dbs do @Inv3
   p.dec <- c.dec;

for p in dbs do @Inv4
   if msg == Commit
      p.value <- p.val;
   _<-Ack;
c.dec <- c.abort?
    Abort : Commit

for p in dbs do @Inv3
    p.dec<-c.dec;

for p in dbs do @Inv4
    if msg == Commit
        p.value <- p.val;

<-Ack;

∀p∈dbs: (p∈done ∧ c.dec= Commit) => p.value=c.val

No case-splits!
@Inv3 = true

No network state!
@Inv4=

Synchronous Proofs
Outline

Key Idea: Pretend Synchrony
Implementation
From 2PC to Paxos
Evaluation
Limitations
Implementation

Key ingredients

Symmetric Nondeterminism

Synchronize by Rewriting
Two-phase Commit: Phase 1

No races!
Two-phase Commit: Phase 1

Need to case-split?

Race between storage nodes!
Two-phase Commit: Phase 1

No! Because races are symmetric
Two-phase Commit: Phase 1

Receive directly after sending

No races!

Theory of Movers [Lipton 1975]
Two-phase Commit: Phase 1

Theory of Movers [Lipton 1975]
Two-phase Commit: Phase 1

Need to case-split?

Race between storage nodes!
Two-phase Commit: Phase 1

Verify arbitrary interleaving

No! Because races are symmetric
Two-phase Commit: Phase 1
Implementation

Key ingredients
Symmetric Nondeterminism
Synchronize by Rewriting
Implementation

Example 2PC
Example: Two-phase Commit
Example: Two-phase Commit

Coordinator

Commit a value

Storage node

Storage node

Storage node
Two-phase Commit: Phase 1

Send Value

Save Value

Save Value

Save Value
Two-phase Commit: Phase 1

Use value to **commit** or **abort**

Abort if any node aborts
Two-phase Commit: Phase 1

Use value to commit or abort

Commit if all node commit
Two-phase Commit: Phase 2

Send commit/abort decision

If commit finalize
Two-phase Commit: Phase 2

Receive acks; done

Send acks
Problem: Asynchronous Proofs are Hard
Make Proofs Easier!
Synchronous Proofs are Easy
Verilog Primer
Example FPU

Given floats $x$ and $y$

... compute $x \times y$
Example FPU

Given floats $x$ and $y$

... but exhibits timing variability
always @ (posedge clk) begin
  if (iszero)
    out <= 0;
  else if (isNaN)
    ...
  else
    out <= flp_res;
end
Example FPU

```verilog
always @(posedge clk) begin
    if (iszero)
        out <= 0;
    else if (isNaN)
        ...
    else
        out <= flp_res;
end
```

```verilog
always @(posedge clk) begin
    ...
    flp_res <= // x*y;
end
```
Influence set:
cycles that influenced value
### Example FPU

**Influence set: for x=0**

<table>
<thead>
<tr>
<th>cycle</th>
<th>x</th>
<th>y</th>
<th>flp_res</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{0}</td>
<td>{0}</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>1</td>
<td>{1}</td>
<td>{1}</td>
<td>∅</td>
<td>{0}</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k-1</td>
<td>{k-1}</td>
<td>{k-1}</td>
<td>{0}</td>
<td>{k-2}</td>
</tr>
<tr>
<td>k</td>
<td>{k}</td>
<td>{k}</td>
<td>{1}</td>
<td>{k-1}</td>
</tr>
</tbody>
</table>
Example FPU

Influence set: \( x=1 \)

<table>
<thead>
<tr>
<th>cycle</th>
<th>( x )</th>
<th>( y )</th>
<th>( \text{flp_res} )</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( {0} )</td>
<td>( {0} )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>1</td>
<td>( {1} )</td>
<td>( {1} )</td>
<td>( \emptyset )</td>
<td>( {0} )</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( k-1 )</td>
<td>( {k-1} )</td>
<td>( {k-1} )</td>
<td>( {0} )</td>
<td>( {k-2} )</td>
</tr>
<tr>
<td>( k )</td>
<td>( {k} )</td>
<td>( {k} )</td>
<td>( {1} )</td>
<td>( {0,k-1} )</td>
</tr>
</tbody>
</table>

sets for out at \( k \) differ: \textbf{timing variability}
Example FPU

How to verify?

Produce product program, track equivalence of influence sets through equivalence of membership
Example FPU

The FPU takes a fast path, if x is 0

<table>
<thead>
<tr>
<th>cycle</th>
<th>x</th>
<th>y</th>
<th>flp_res</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k-1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>k</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Example FPU

The FPU takes the slow path, if x is 1

<table>
<thead>
<tr>
<th>cycle</th>
<th>x</th>
<th>y</th>
<th>flp_res</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k-1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>k</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Example FPU

Influence set: all cycles that influenced value

<table>
<thead>
<tr>
<th>cycle</th>
<th>x</th>
<th>y</th>
<th>flp_res</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{0}</td>
<td>{0}</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>1</td>
<td>{1}</td>
<td>{1}</td>
<td>∅</td>
<td>{0}</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k-1</td>
<td>{k-1}</td>
<td>{k-1}</td>
<td>{0}</td>
<td>{k-2}</td>
</tr>
<tr>
<td>k</td>
<td>{k}</td>
<td>{k}</td>
<td>{1}</td>
<td>{k-1}</td>
</tr>
</tbody>
</table>
the network is fine BUT

😊 hi

actually I'm going to garbage collect for 2 minutes no reply for you!
## Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Goolong</th>
<th></th>
<th>Dafny</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#LG</td>
<td>#LI</td>
<td>#A</td>
<td>#I</td>
</tr>
<tr>
<td>Two-Phase Commit</td>
<td>102</td>
<td>49</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Raft Leader Election</td>
<td>138</td>
<td>44</td>
<td>17</td>
<td>6</td>
</tr>
<tr>
<td>Single-Decree Paxos</td>
<td>504</td>
<td>65</td>
<td>23</td>
<td>14</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>744</td>
<td>158</td>
<td>42</td>
<td>23</td>
</tr>
<tr>
<td>Multi-Paxos KV</td>
<td>847</td>
<td>100</td>
<td>21</td>
<td>14</td>
</tr>
</tbody>
</table>
BACKUP

Tags
Mutable

Data
Immutable

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)
<table>
<thead>
<tr>
<th>System</th>
<th>Throughput (req/ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goolong</td>
<td>118.5</td>
</tr>
<tr>
<td>PSync</td>
<td>32.4</td>
</tr>
<tr>
<td>Ivy-Raft*</td>
<td>13.5</td>
</tr>
<tr>
<td>IronKV*</td>
<td>~ 30</td>
</tr>
</tbody>
</table>
BACKUP

clocks lie

It's 5pm

It's 5pm

2 minutes later...

5:02

5:17

Just got paged

Why...
Problem 1: Asynchrony

Processes/messages have different speed
Problem 2: Message Drops

Network may be unreliable
Problem 3: Parametrized

Number of nodes not known
How to make sure they don’t?

Too many possibilities

Infinite Number of States: No guarantees

Testing

Model Checking

Deductive Proofs