Repliss: A tool for building correct applications on top of weak consistency

Peter Zeller, Arnd Poetzsch-Heffter, Annette Bieniusa

May 2017

TU Kaiserslautern, AG Softech

Repliss

Repliss is a tool for developing correct applications on top of weakly consistent databases.

(Replicated information systems with strong guarantees)

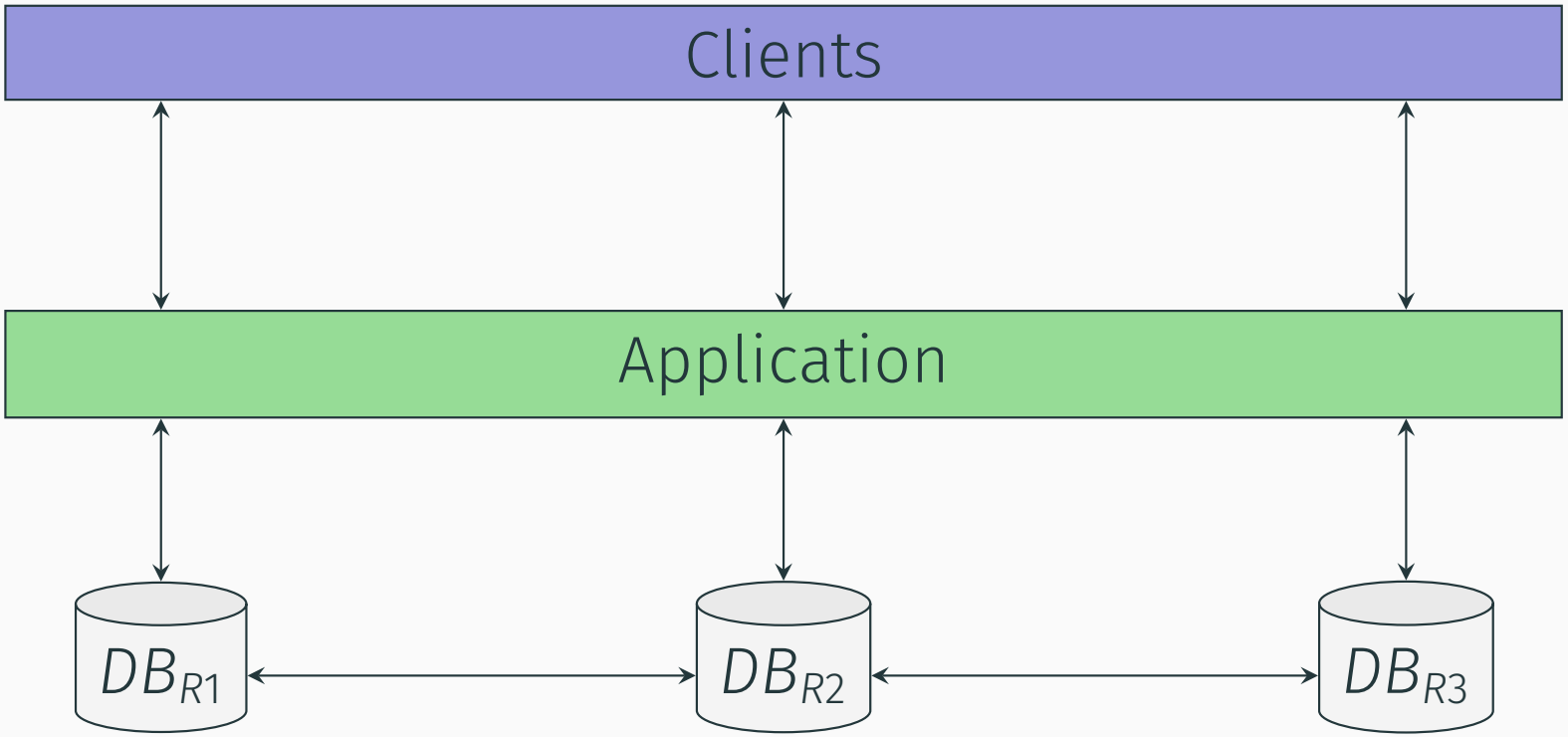
Repliss

Repliss is a tool for developing correct applications on top of weakly consistent databases.

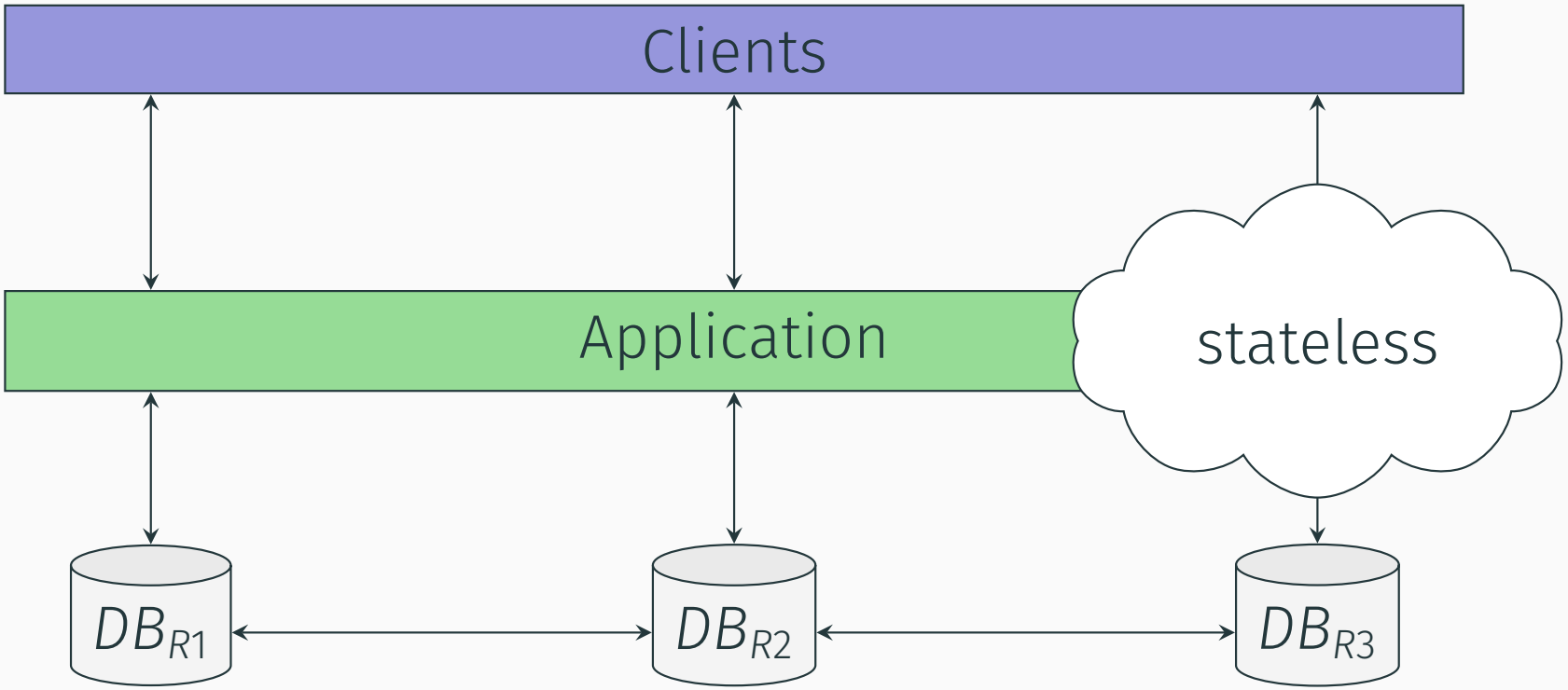
(Replicated information systems with strong guarantees)

- Specification of safety properties
- Automated and manual verification
- Automated testing

Setting



Setting



Repliss Workflow

1. Write Code, define database schema
2. Specify functional properties
   (run automated tests*)
3. Add stronger invariants for verification
4. Run automated verification tool (and tests*)
5. When failed:
   Adapt 1/2/3 or use manual verification (Isabelle)

* no additional input required

Repliss Demo

The foundation: database specification

- User provided data type specifications
- Repliss assumes those are correct
  - Already formally verified in [Forte 2014]
  - For *Antidote*: property based tests for CRDT library (with Vitor Enes Duarte and Georges Younes)
- Assuming transactional causal$^+$ consistency (as provided by Antidote)

The application

- Clients can invoke API arbitrarily
- Invocations executed concurrently
- No concurrency inside invocation
- Replicas exchange updates asynchronously

Application specifications

Data invariants:

```
invariant forall p: Player, t: Tournament ::
   enrolled_contains(p,t) ==>
     players_contains(p)
```

Invariants over the history of invocations:

```
invariant forall r, g: invocationId, u:
  UserId ::
     r.info == removeUser(u)
  && g.info == getUser(u)
  && r happened before g
  ==> g.result == getUser_res(notFound())
```

Automated Testing

Execution model

Required for testing:

- Reproducible executions ($\rightarrow$ deterministic)

Execution model

Required for testing:

- Reproducible executions ($\rightarrow$ deterministic)

Idea: Capture all nondeterministic choices in a trace

Trace Example

```
i1  invoc(updateMail, [user1, ”ab@c.de”])
i1  beginAtomic(tx4, [tx1, tx2])
i2  invoc(removeUser, [user1])
i2  beginAtomic(tx5, [tx1, tx3])
i1  invCheck
i2  return
i1  return
```

Trace Example

i1	**invoc(updateMail, [user1, ”ab@c.de”])**
i1	beginAtomic(tx4, [tx1, tx2])
i2	**invoc(removeUser, [user1])**
i2	beginAtomic(tx5, [tx1, tx3])
i1	invCheck
i2	return
i1	return

Invoke API

Trace Example

i1 invoc(updateMail, [user1, ”ab@c.de”])
i1 **beginAtomic(tx4, [tx1, tx2])**
i2 invoc(removeUser, [user1])
i2 **beginAtomic(tx5, [tx1, tx3])**
i1 invCheck
i2 return
i1 return

Start Transaction (includes pulled transactions)

Trace Example

i1	invoc(updateMail, [user1, ”ab@c.de”])
i1	beginAtomic(tx4, [tx1, tx2])
i2	invoc(removeUser, [user1])
i2	beginAtomic(tx5, [tx1, tx3])
i1	**invCheck**
i2	return
i1	return

Check invariants in context of invocation

Trace Example

i1	invoc(updateMail, [user1, ”ab@c.de”])
i1	beginAtomic(tx4, [tx1, tx2])
i2	invoc(removeUser, [user1])
i2	beginAtomic(tx5, [tx1, tx3])
i1	invCheck
i2	**return**
i1	**return**

Return from API invocation

Actions

- Client invokes API procedure
- Start transaction
- Return
- Invariant check
- **Node crash**
- **Generate unique id**

Actions

- Client invokes API procedure
- Start transaction
- Return
- Invariant check
- **Node crash**
- **Generate unique id**
- Local steps (local computations, database operations, commits)
  - Not recorded in trace
  - Not interfering with other invocations
  - Directly executed until next decision-point

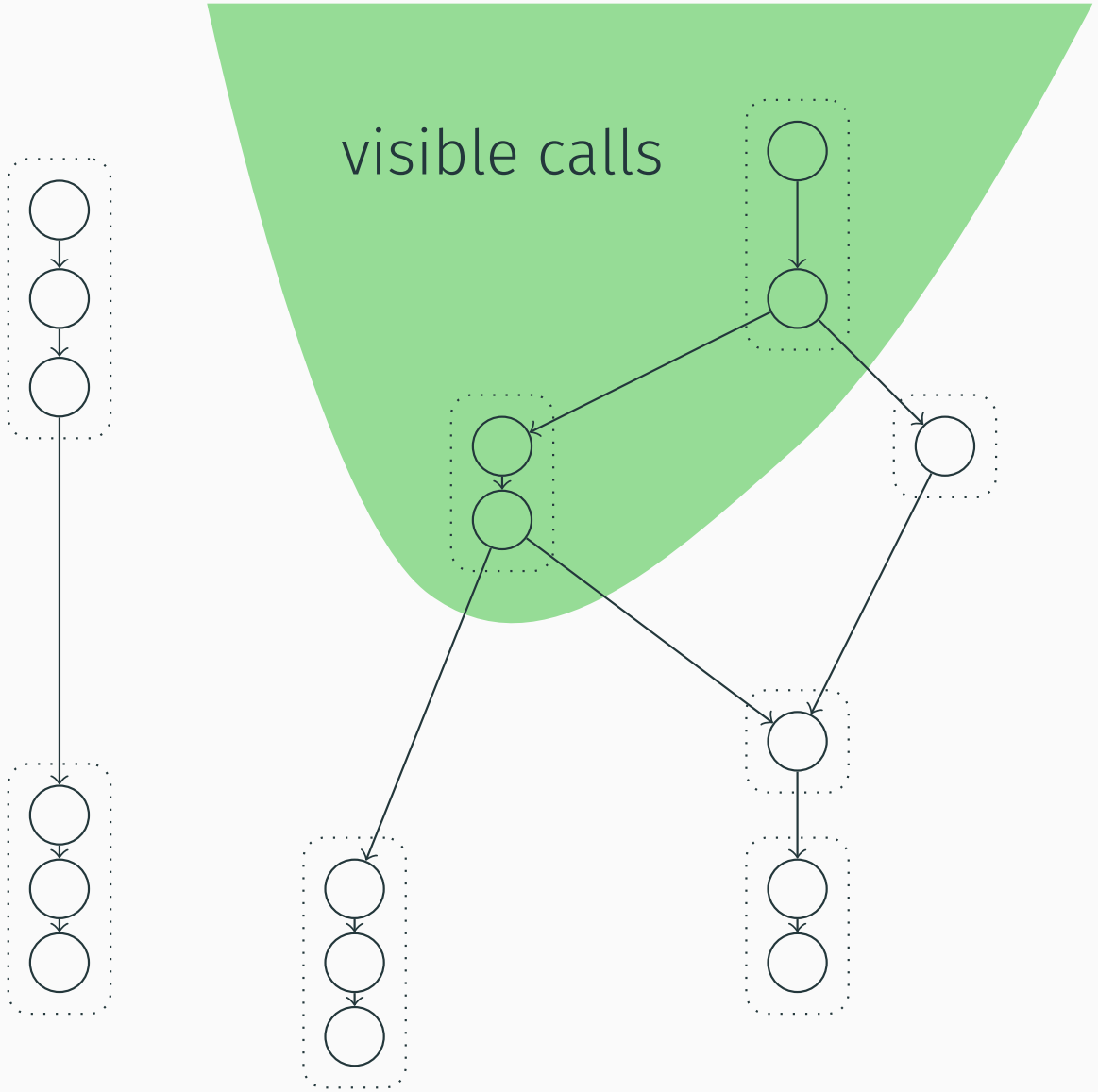
Executing database operations

Result of operation depends only on update-history visible at local replica.

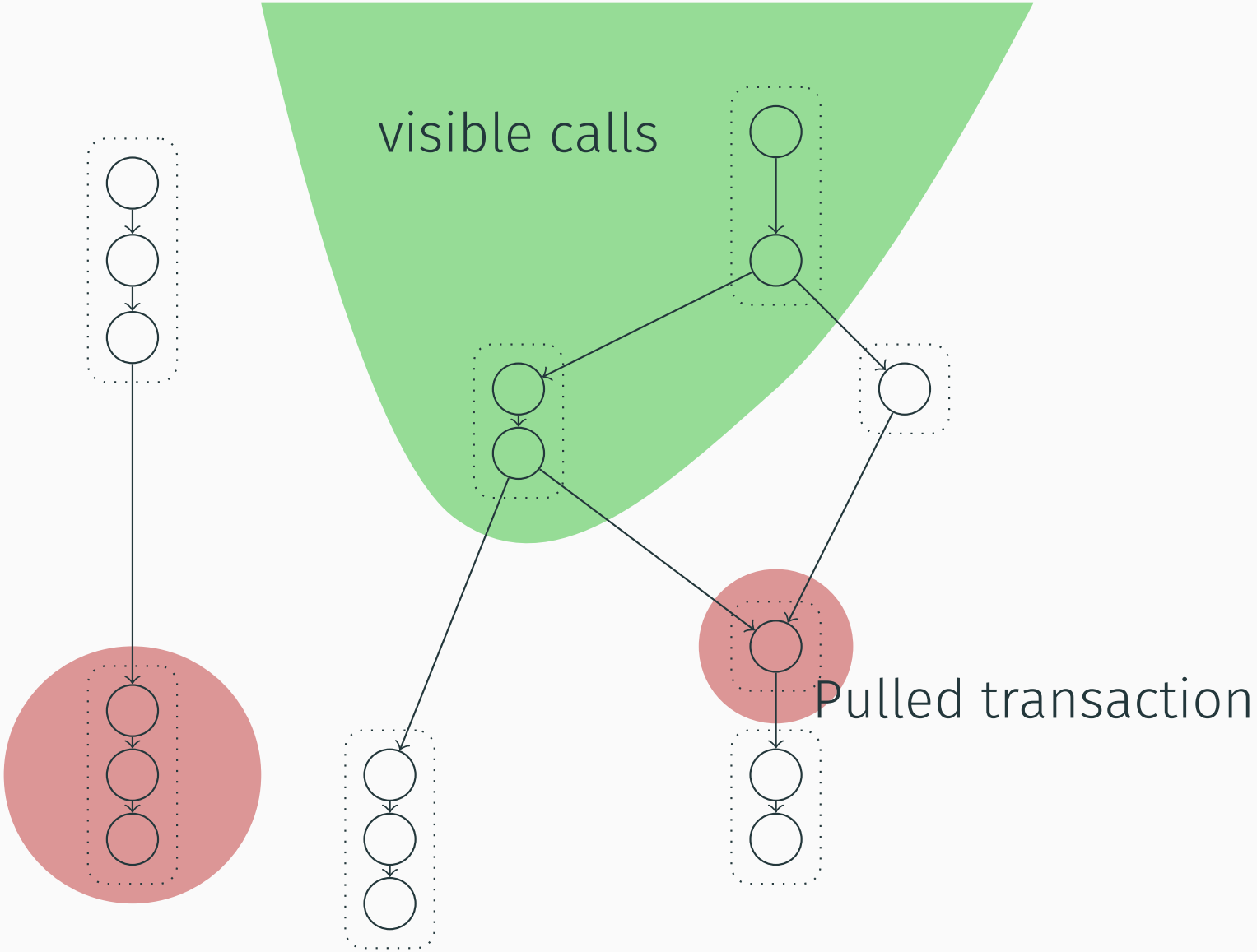
Executing an operation updates history:

- Arguments and result of operation are recorded
- Originating transaction and invocation are recorded
- New operation happens causally after currently visible events
- New operation is added to set of local visible events

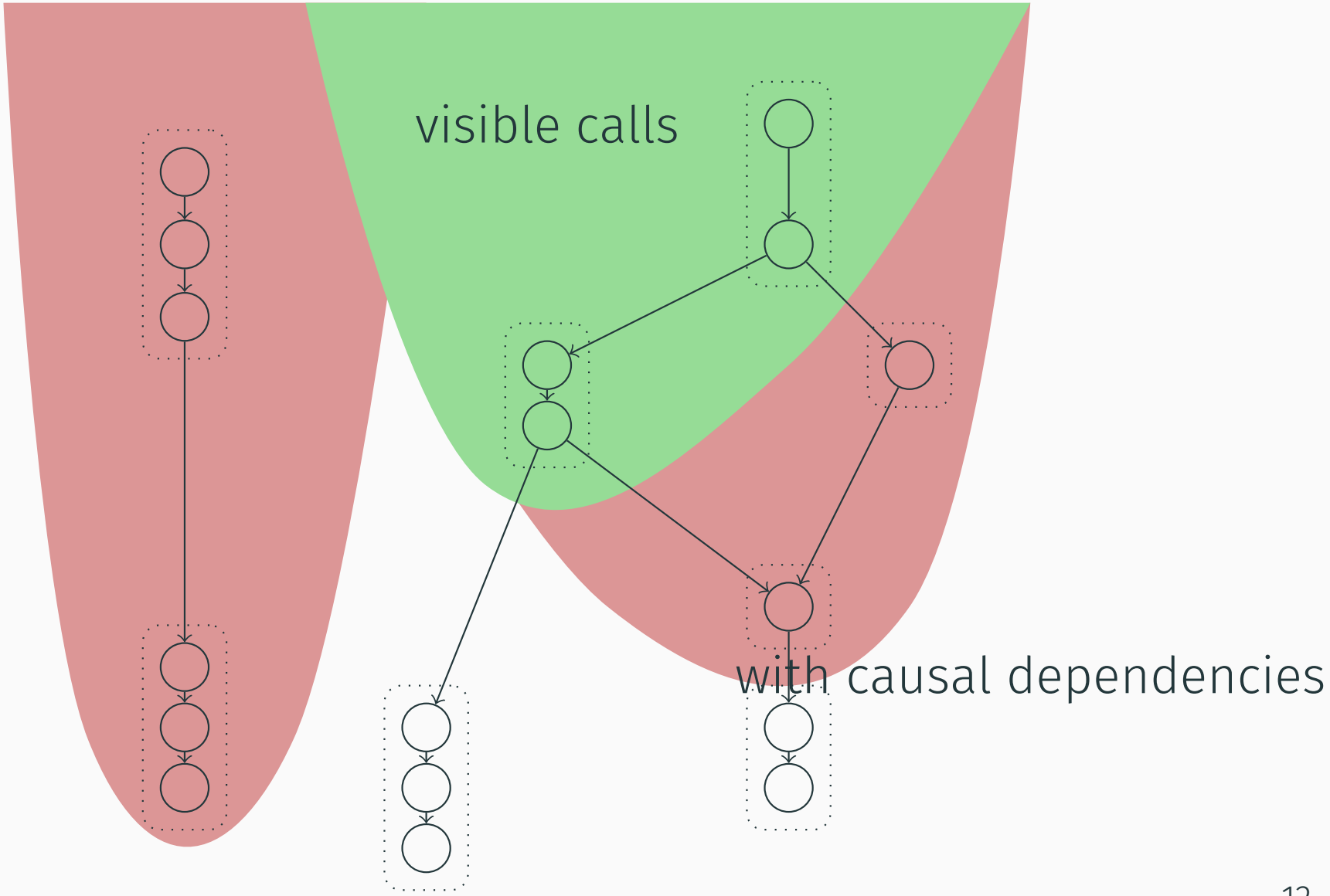
Starting Transactions (pulling updates)



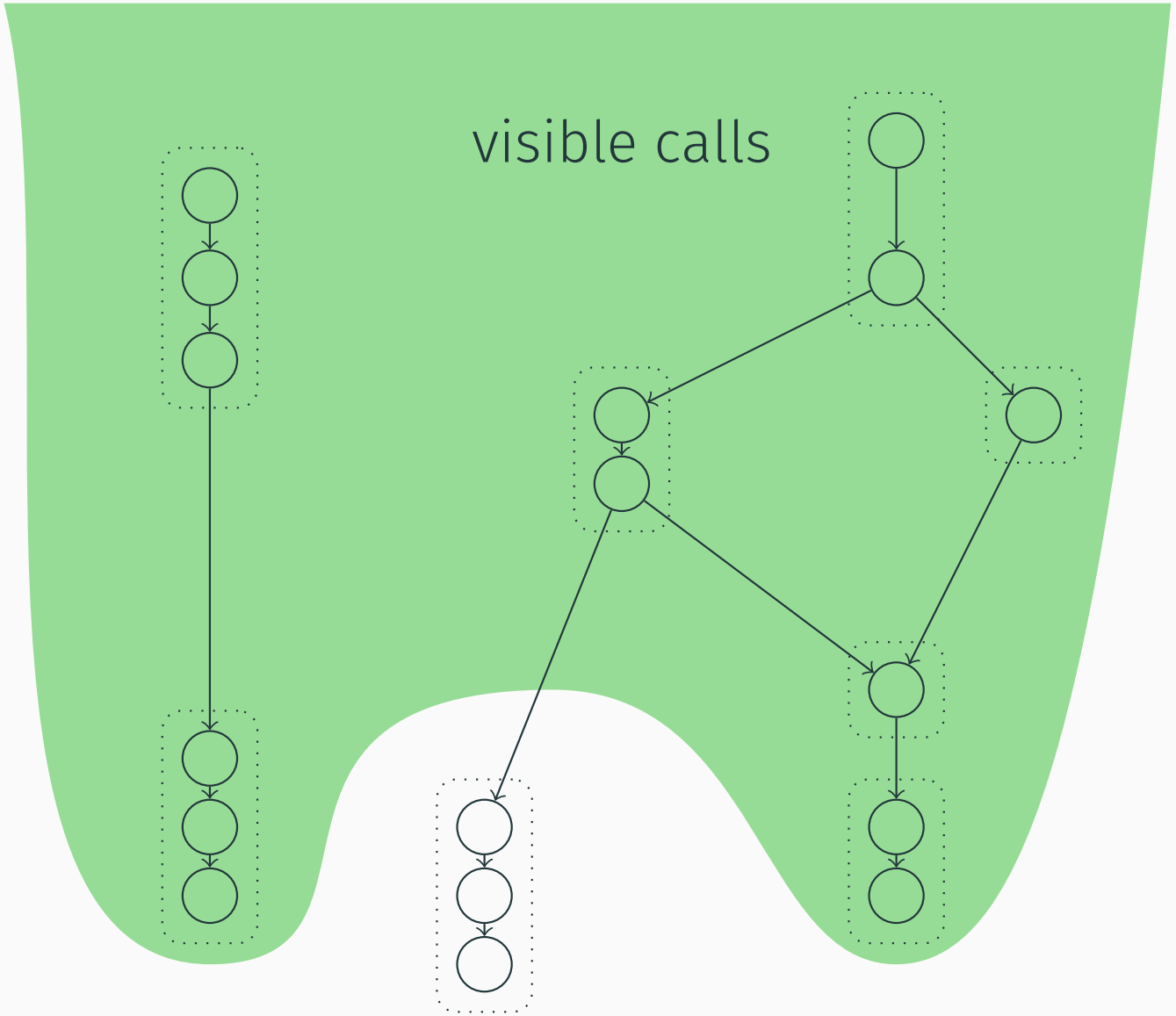
Starting Transactions (pulling updates)



Starting Transactions (pulling updates)



Starting Transactions (pulling updates)



Eliminating Nondeterminism

Nondeterministic steps:

- Choosing pulled transactions at start of transaction
- API invocations
- Order of interleaving

Eliminating Nondeterminism

Nondeterministic steps:

- Choosing pulled transactions at start of transaction
- API invocations
- Order of interleaving

All recorded in trace.

Shrinking Counter Examples

When invariant violation is found:

- Try to remove action from trace
- Adapt trace (e.g. remove dependent actions)
- Repeat if still failing, otherwise try next action

Shrinking Counter Examples

When invariant violation is found:

- Try to remove action from trace
- Adapt trace (e.g. remove dependent actions)
- Repeat if still failing, otherwise try next action

Important: Stability

- Execution semantic based on traces ensures that removing actions has little effect on other actions
- `beginAtomic` action includes set of all causal dependencies

Shrinking Counter Examples

When invariant violation is found:

- Try to remove action from trace
- Adapt trace (e.g. remove dependent actions)
- Repeat if still failing, otherwise try next action

Important: Stability

- Execution semantic based on traces ensures that removing actions has little effect on other actions
- `beginAtomic` action includes set of all causal dependencies

Example: 19 invocations, reduced to 4 invocations

Verification

Verification

Central aspects of verification approach:

- Using history of API-invocations and history of database-calls in invariants
- Isolation of causal$^+$ transactions
  $\Rightarrow$ Sufficient: transactions maintain invariants.
  $\Rightarrow$ No concurrent executions have to be considered.
  $\Rightarrow$ Can use tools for verification of sequential programs
- Concurrency covered by snapshot semantics

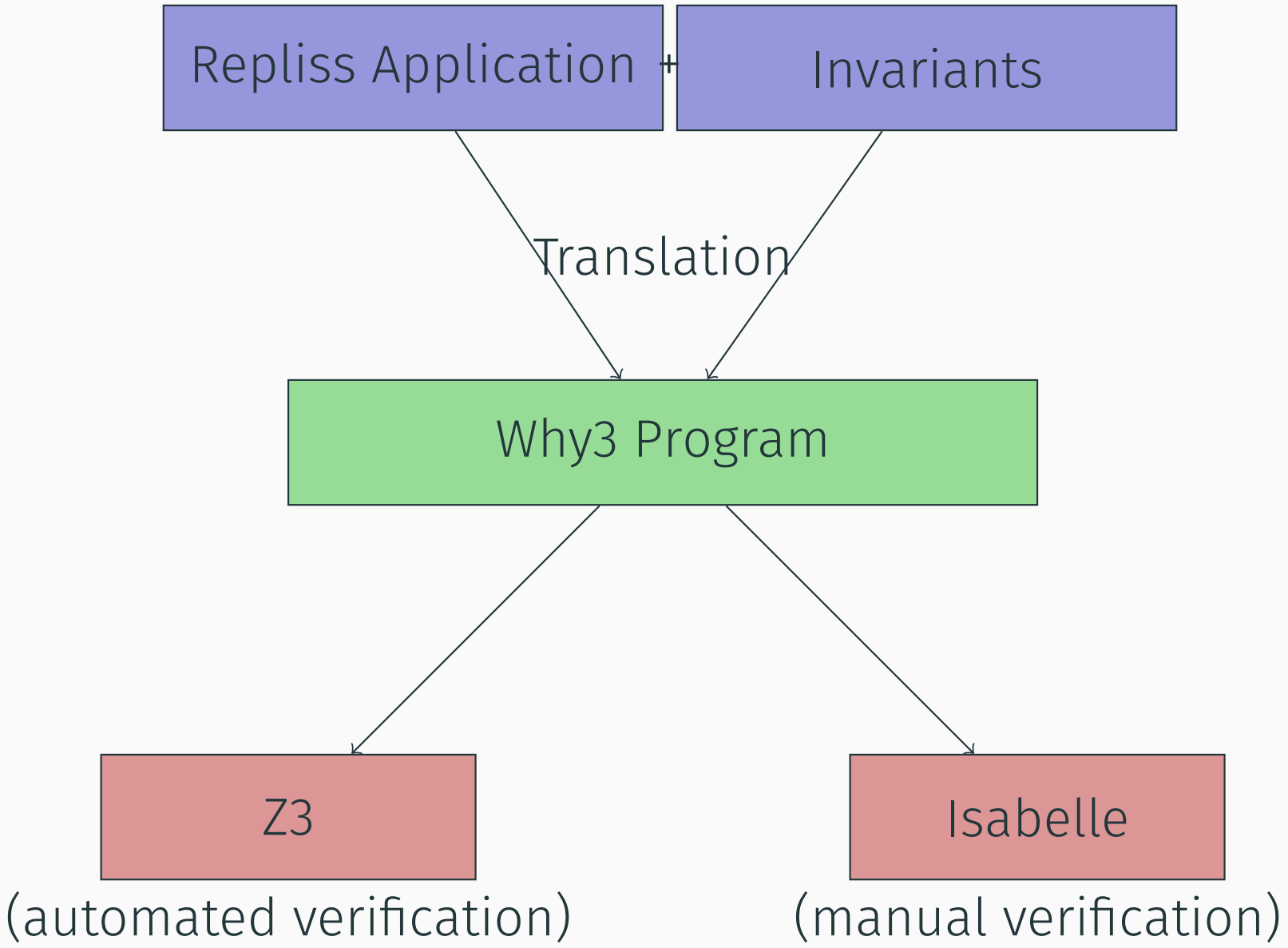
Interleaving

```
i1  beginAtomic(tx4, snapshot = [tx1, tx2])
i1  local
i2  beginAtomic(tx5, snapshot = [tx1, tx3])
i2  local
i2  commit
i1  local
i1  commit
```

⇔

```
i1  beginAtomic(tx4, snapshot = [tx1, tx2])
i1  local
i1  local
i1  commit
i2  beginAtomic(tx5, snapshot = [tx1, tx3])
i2  local
i2  commit
```

Repliss Verification Tool



Future work - Verification

- Proving soundness of proof approach
- Local assertions and loops
- More flexibility (weaker and stronger consistency)

Future Work - Testing

Performance
(2 seconds for executing 100 actions, 6 seconds for 200 action)

- Main problem: Evaluating logical formulas
  E.g.: CRDT query specification:

```
(exists c1: callId, f: userRecordField, v:
  String ::
      c1 is visible
    && c1.op == mapWrite(u, f, v)
    && (forall c2: callId :: (c2 is visible &&
      c2.op == mapDelete(u)) ==> c2 happened
      before c1))
```

- Try to use smarter algorithm for evaluation
  (e.g. SMT solver)

Future Work - Verification and testing

Tighter integration of testing and verification

- Use verification errors to guide tests
  $\rightarrow$ real counter examples for verification errors

This was Repliss!