Implementing STM

Misha Kozik
@mishadoff

What is STM?

Software **T**ransactional **M**emory

just another one
concurrency control mechanism
like locks or actors

What is STM?

Software **T**ransactional **M**emory

Algorithm for **Database**-like concurrency

just another one
concurrency control mechanism
like locks or actors

Database Transactions

Atomicity — “all or nothing”

Consistency — data is valid at the end

Isolation — not affected by other txs

Durability — ~persistence

Memory ~~Database~~ Transactions

Atomicity — "all or nothing"

Consistency — data is valid at the end

Isolation — not affected by other txs

Durability — ~~persistence~~

Bank Accounts Transfer

```
public class Account {
    private long money;

    void add(long amount) {
        money += amount;
    }
}

public class Bank {
    private Account[] accounts;

    void transfer(Account a1, Account a2, int amount) {
        a1.add(-amount);
        a2.add(amount);
    }

    public long sum() {
        long sum = 0;
        for (Account a : accounts) sum += a.getMoney();
    }
}
```

Simulate Process

```
public void simulate(int threads, int num) {
    ExecutorService service =
                Executors.newFixedThreadPool(threads);
    for (int i = 0; i < threads; i++) {
        service.submit(new BankThread(this, num));
    }
    service.shutdown();
    service.awaitTermination(1, TimeUnit.MINUTES);
}

public class BankThread implements Runnable {
    private Bank bank;
    private int num;

    public void run() {
        for (int i = 0; i < num; i++) {
            bank.transfer(bank.getRandomAccount(),
                          bank.getRandomAccount(),
                          bank.getRandomValue());
        }
    }
}
```

Where is my money?!

```
Bank bank = new Bank();
System.out.println("Bank sum before: " + bank.sum());
bank.simulate(100, 100000);
System.out.println("Bank sum after: " + bank.sum());
```

```
Bank sum before:   10000000
Bank sum after:     9970464
```

-20000$ = 

49634

How to solve it?

synchronized

What the difference?

```
public class Bank {
    synchronized void transfer(Account a1, Account a2, int amt) {
        a1.add(-amt);
        a2.add(amt);
    }
}
```

VS

```
public class Account {
    synchronized void transfer(Account a, int amt) {
        this.add(-amt);
        a.add(amt);
    }

    synchronized void add(int amt) {
       value += amt;
    }
}
```

synchronized: cons

→ no read/write distinction

→ slow/error-prone

→ low-level

→ protect code rather than data

Locks

still use locks in 2013

```
RingBuffer buffer = null;
readCpsLock.lock();
try {
    if (ticks.containsKey(accountId)) {
        buffer = ticks.get(accountId);
        // check if this value changed
        if (buffer.getMaxCallCount() != cpsValue) {
            readCpsLock.unlock();
            try {
               writeCpsLock.lock();
               try {
                   ticks.remove(accountId);
                   buffer = new RingBuffer(cpsValue, DEFAULT_TIME_UNIT);
                   ticks.put(accountId, buffer);
               }
               finally {
                   writeCpsLock.unlock();
               }
            }
            finally {
                readCpsLock.lock();
            }
        }
    }
    else {
        buffer = new RingBuffer(cpsValue, DEFAULT_TIME_UNIT);
        readCpsLock.unlock();
        try {
            writeCpsLock.lock();
            try {
                ticks.put(accountId, buffer);
            }
            finally {
                writeCpsLock.unlock();
            }
        }
        finally {
            readCpsLock.lock();
        }
    }
}
finally {
    readCpsLock.unlock();
}
```

useful code

Locks: cons

→ tricky

→ error-prone

→ not composable

→ protect code rather than data

Actors

Actors in Erlang


```
ping(0, PID) ->
  PID ! finished,
  io:format("Ping stop~n", []);

ping(N, PID) ->
  PID ! {ping, self()},
  receive
    pong ->
      io:format("Ping~n", [])
  end,
  ping(N - 1, PID).
```

```
pong() ->
  receive
    finished ->
      io:format("Pong stop~n", []);
      {ping, PID} ->
        io:format("Pong~n", []),
        PID ! pong,
        pong()
  end.
```

```
start() ->
    PID = spawn(pingpong, pong, []),
    spawn(pingpong, ping, [3, PID]).
```

Actors: cons

→ code restructuring

→ deadlocks still possible

STM

STM in Clojure

```
(def account (ref 100))
```

```
(defn transfer [acc1 acc2 amount]
  (dosync
    (alter acc1 - amount)
    (alter acc2 + amount)))
```

STM in Java

```
transaction {
  acc1.add(-value);
  acc2.add(value);
}
```

STM in Java

```
STM.transaction(
    new Runnable() {
        public void run() {
            acc1.add(-value);
            acc2.add(value);
        }
    });
```

STM in Java 8

```
STM.transaction(() → {
    acc1.add(-value);
    acc2.add(value);
});
```

STM: pros

→ natural approach

→ coordinated state

→ no deadlocks, no livelocks

Implementing STM

Intuition

→ start transaction

→ get ref snapshot

→ set of reads/writes* on snapshot

→ CAS initial and current snapshot

→ if CAS succeeds commit

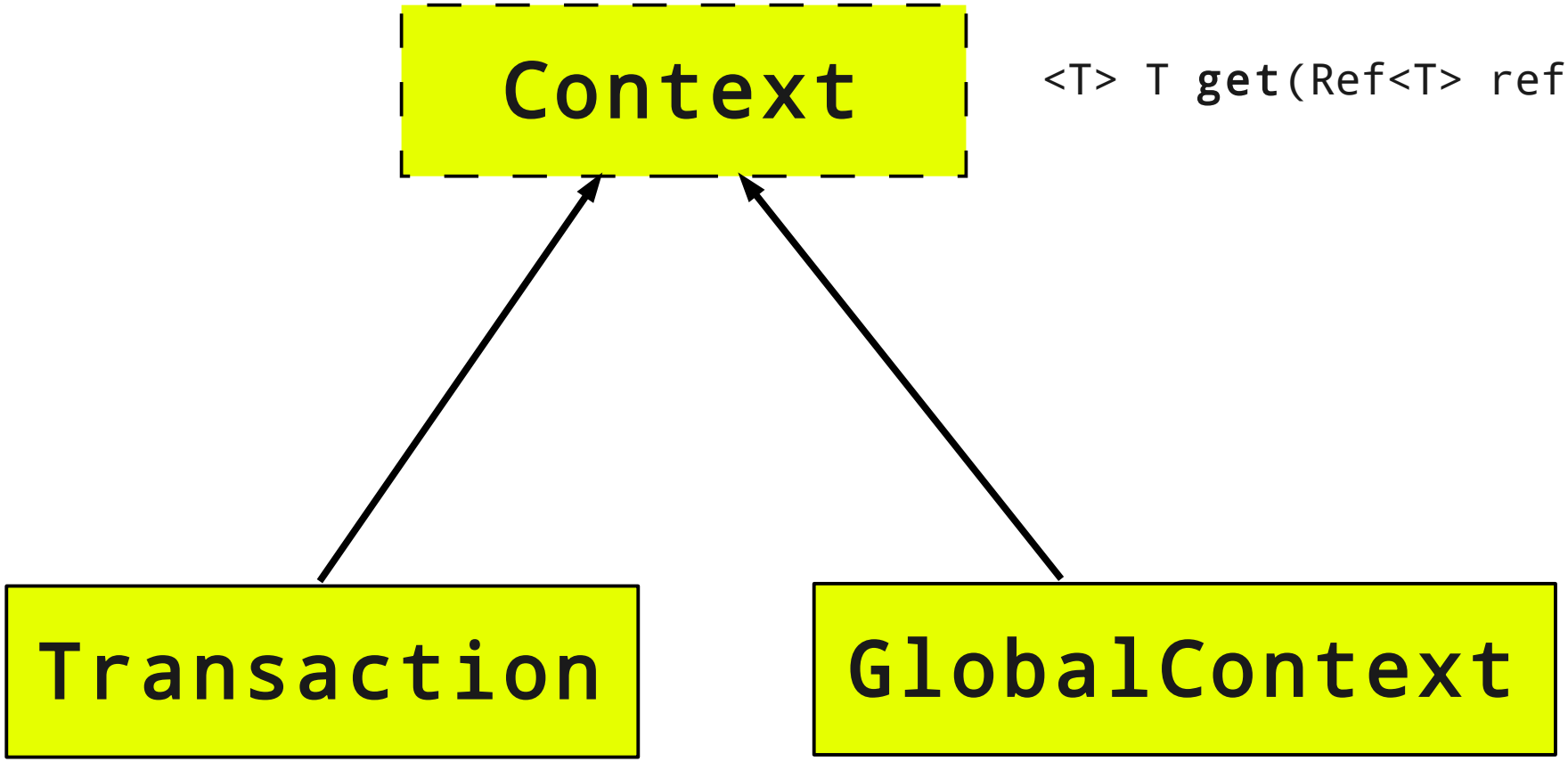
→ else retry

Ref v0.1

```
public final class Ref<T> {
    private T value;

    public T getValue() {
        return value;
    }

    public void setValue(T value) {
        this.value = value;
    }
}
```

Ref v0.1

```
public final class Ref<T> {
    private T value;

    public T getValue() {
        return value;
    }                                   // from anywhere

    public void setValue(T value) {
        this.value = value;
    }                                   // from transaction
}
```


Context
<T> T get(Ref<T> ref);
Transaction
GlobalContext
<T> void set(Ref<T> ref, T value);
singleton

Ref v0.2

```
public final class Ref<T> {
    private T value;

    public T getValue(Context ctx) {
        return ctx.get(this);
    }

    public void setValue(T value, Transaction tx) {
        tx.set(this, value);
    }
}
```

STM API

```
public final class STM {
    private STM() {}

    public static void transaction(TransactionBlock block) {
        Transaction tx = new Transaction();
        block.setTx(tx);
        block.run();
    }

}

public abstract class TransactionBlock implements Runnable {
    private Transaction tx;

    void setTx(Transaction tx) {
        this.tx = tx;
    }

    public Transaction getTx() {
        return tx;
    }
}
```

STM API

```
public final class STM {
    private STM() {}

    public static void transaction(TransactionBlock block) {
        Transaction tx = new Transaction();
        block.setTx(tx);
        block.run();
    }

}

public abstract class TransactionBlock implements Runnable {
    private Transaction tx;

    void setTx(Transaction tx) {
        this.tx = tx;
    }

    public Transaction getTx() {
        return tx;
    }
}
```

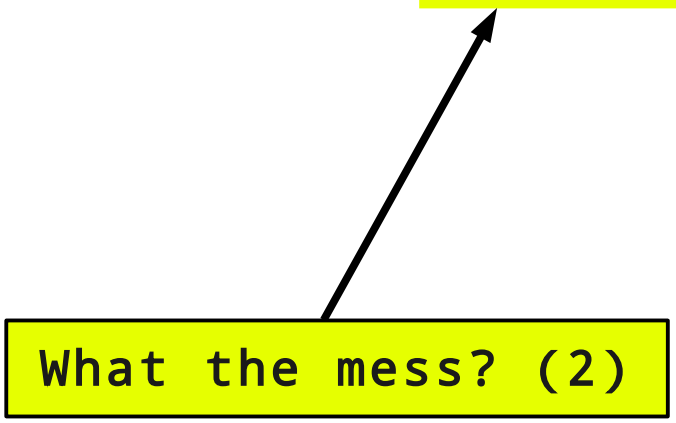
NO return value (1)

STM Usage

```
STM.transaction(new TransactionBlock() {

      @Override
      public void run() {
          int old1 = a1.getValue(this.getTx());
          a1.setValue(old1 - value, this.getTx());
          int old2 = a2.getValue(this.getTx());
          a2.setValue(old2 + value, this.getTx());
      }

  });
```

STM Usage

```
STM.transaction(new TransactionBlock() {

      @Override
      public void run() {
          int old1 = a1.getValue(this.getTx());
          a1.setValue(old1 - value, this.getTx());
          int old2 = a2.getValue(this.getTx());
          a2.setValue(old2 + value, this.getTx());
      }

  });
```

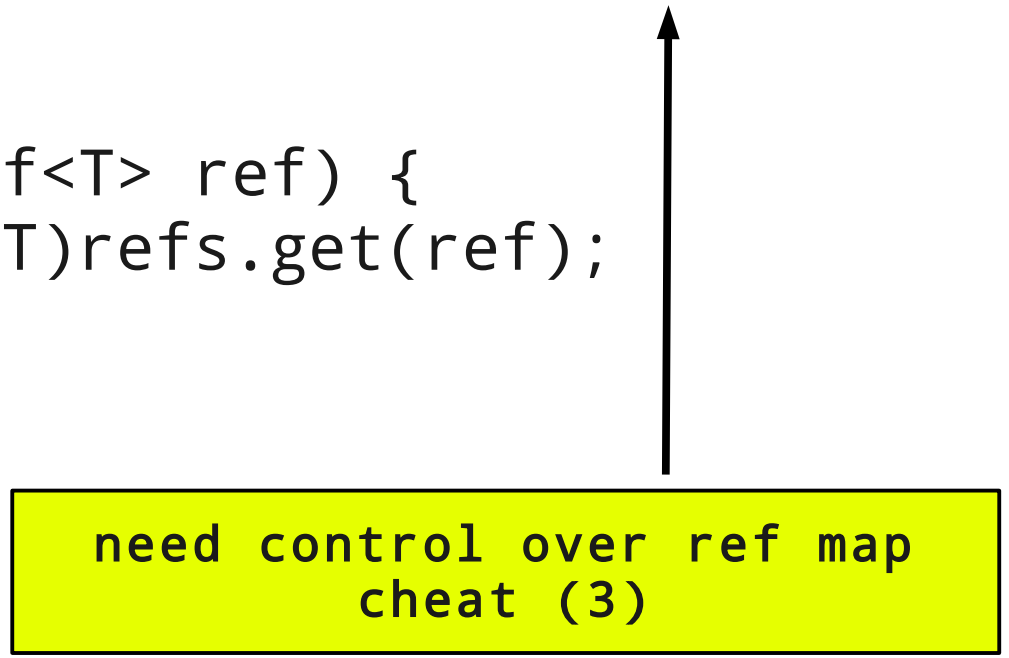


GlobalContext v0.1

```
public class GlobalContext extends Context {
    private HashMap<Ref, Object> refs;

    @Override
    <T> T get(Ref<T> ref) {
        return (T)refs.get(ref);
    }
}
```

GlobalContext v0.1

```
public class GlobalContext extends Context {
    private HashMap<Ref, Object> refs;

    @Override
    <T> T get(Ref<T> ref) {
        return (T)refs.get(ref);
    }
}
```



Ref v0.3

```
public final class Ref<T> {
    private T value;

    public T getValue(Context ctx) {
        return ctx.get(this);
    }

    public void setValue(T value, Transaction tx) {
        tx.set(this, value);
    }
}
```

GlobalContext v0.2

```
public class GlobalContext extends Context {
    private HashMap<Ref, Object> refs;

    @Override
    <T> T get(Ref<T> ref) {
        return ref.value;
    }
}
```

Transaction v0.1

```
public final class Transaction extends Context {
    private HashMap<Ref, Object> inTxMap;

    @Override
    <T> T get(Ref<T> ref) {
        if (!inTxMap.containsKey(ref)) {
            inTxMap.put(ref, ref.value);
        }
        return (T)inTxMap.get(ref);
    }

    <T> void set(Ref<T> ref, T value) {
        inTxMap.put(ref, value);
    }
}
```

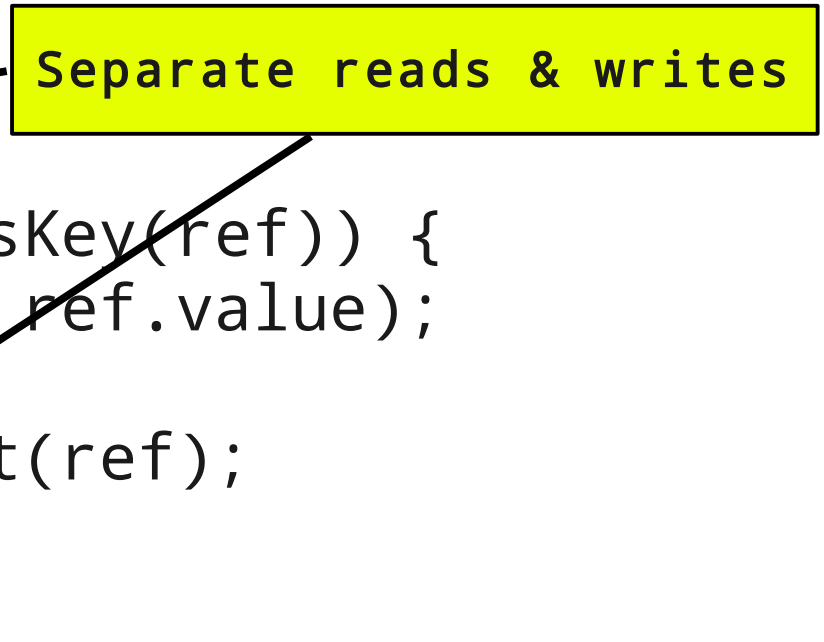
Transaction v0.1

```
public final class Transaction extends Context {
    private HashMap<Ref, Object> inTxMap;

    @Override
    <T> T get(Ref<T> ref) {
        if (!inTxMap.containsKey(ref)) {
            inTxMap.put(ref, ref.value);
        }
        return (T)inTxMap.get(ref);
    }

    <T> void set(Ref<T> ref, T value) {
        inTxMap.put(ref, value);
    }
}
```

NO Snapshot Isolation (4)

Transaction v0.1

```
public final class Transaction extends Context {
    private HashMap<Ref, Object> inTxMap;

    @Override
    <T> T get(Ref<T> ref) {
        if (!inTxMap.containsKey(ref)) {
            inTxMap.put(ref, ref.value);
        }
        return (T)inTxMap.get(ref);
    }

    <T> void set(Ref<T> ref, T value) {
        inTxMap.put(ref, value);
    }
}
```



Transaction v0.2

```
public final class Transaction extends Context {
    private HashMap<Ref, Object> inTxMap;
    private HashSet<Ref> toUpdate;

    @Override
    <T> T get(Ref<T> ref) {
        if (!inTxMap.containsKey(ref)) {
            inTxMap.put(ref, ref.value);
        }
        return (T)inTxMap.get(ref);
    }

    <T> void set(Ref<T> ref, T value) {
        inTxMap.put(ref, value);
        toUpdate.add(ref);
    }
}
```

Commit

```
public final class Transaction extends Context {
    // …

    void commit() {
        synchronized (STM.commitLock) {
            // TODO validate
            // TODO write values
        }
    }

}
```

```
public final class STM {
    private STM() {}

    public static Object commitLock = new Object();

    // ...
}
```

Commit: Write Values

```
void commit() {
    synchronized (STM.commitLock) {
        // TODO validate
        for (Ref ref : toUpdate) {
            ref.value = inTxMap.get(ref);
        }
    }
}
```

Validation Idea

→ get all transaction-local refs

→ compare version with transaction revision

→ if all versions the same
allow commit

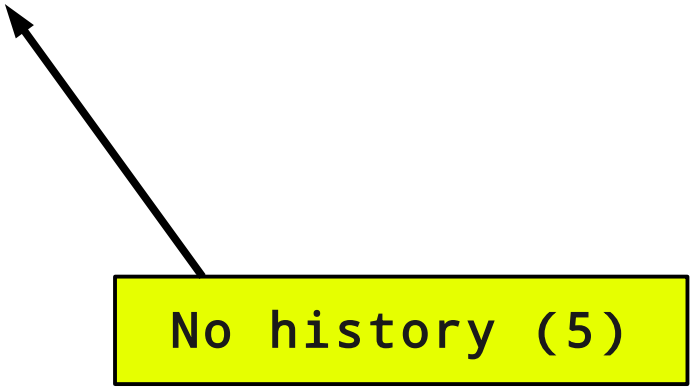
→ else retry

Ref v0.4

```
public final class Ref<T> {
    T value;
    long revision = 0;


    // ...
}
```

Ref v0.4

```
public final class Ref<T> {
    T value;
    long revision = 0;


    // ...
}
```



Transaction v0.3

```
public final class Transaction extends Context {
    // ...

    private long revision;
    private static AtomicLong transactionNum =
                                    new AtomicLong(0);

    Transaction() {
        revision = transactionNum.incrementAndGet();
    }

    // ...
}
```

Transaction v0.3

```
public final class Transaction extends Context {
    private HashMap<Ref, Long> version;

    <T> T get(Ref<T> ref) {
        if (!inTxMap.containsKey(ref)) {
            inTxMap.put(ref, ref.value);
            if (!version.containsKey(ref)) {
                version.put(ref, ref.revision);
            }
        }
        return (T)inTxMap.get(ref);
    }

    <T> void set(Ref<T> ref, T value) {
        inTxMap.put(ref, value);
        toUpdate.add(ref);
        if (!version.containsKey(ref)) {
            version.put(ref, ref.revision);
        }
    }
}
```

Commit: Validation

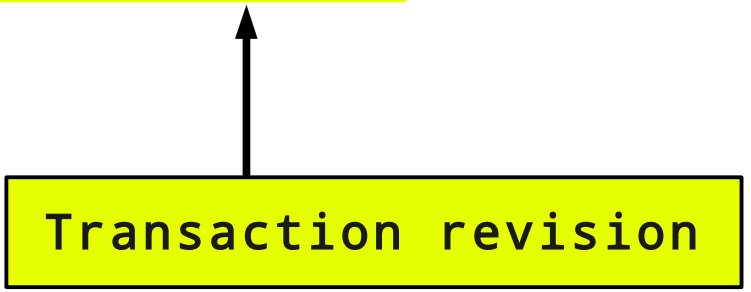
```
boolean commit() {
    synchronized (STM.commitLock) {
        boolean isValid = true;
        for (Ref ref : inTxMap.keySet()) {
            if (ref.revision != version.get(ref)) {
                isValid = false;
                break;
            }
        }

        if (isValid) {
            for (Ref ref : toUpdate) {
                ref.value = inTxMap.get(ref);
                ref.revision = revision;
            }
        }
        return isValid;
    }
}
```

Commit: Validation

```
boolean commit() {
    synchronized (STM.commitLock) {
        boolean isValid = true;
        for (Ref ref : inTxMap.keySet()) {
            if (ref.revision != version.get(ref)) {
                isValid = false;
                break;
            }
        }

        if (isValid) {
            for (Ref ref : toUpdate) {
                ref.value = inTxMap.get(ref);
                ref.revision = revision;
            }
        }
        return isValid;
    }
}
```

Commit: Validation

```
boolean commit() {
    synchronized (STM.commitLock) {
        boolean isValid = true;
        for (Ref ref : inTxMap.keySet()) {
            if (ref.revision != version.get(ref)) {
                isValid = false;
                break;
            }
        }

        if (isValid) {
            for (Ref ref : toUpdate) {
                ref.value = inTxMap.get(ref);
                ref.revision = revision;
            }
        }
        return isValid;
    }
}
```



Commit: Validation

```
boolean commit() {
    synchronized (STM.commitLock) {
        boolean isValid = true;
        for (Ref ref : inTxMap.keySet()) {
            if (ref.revision != version.get(ref)) {
                isValid = false;
                break;
            }
        }

        if (isValid) {
            for (Ref ref : toUpdate) {
                ref.value = inTxMap.get(ref);
                ref.revision = revision;
            }
        }
        return isValid;
    }
}
```

STM Transaction

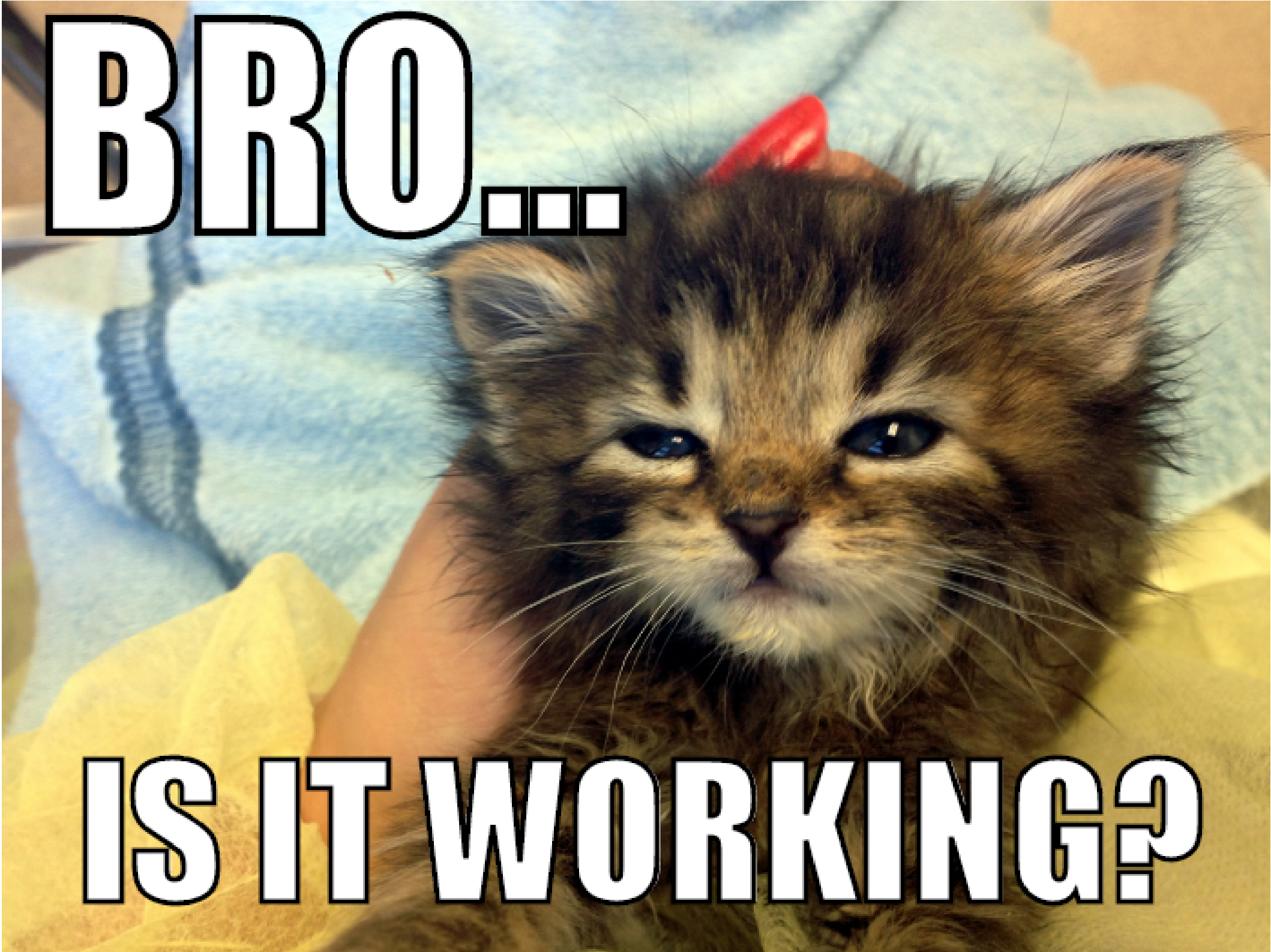
```
public final class STM {
    private STM() {}

    public static Object commitLock = new Object();

    public static void transaction(TransactionBlock block) {
        boolean committed = false;
        while (!committed) {
            Transaction tx = new Transaction();
            block.setTx(tx);
            block.run();
            committed = tx.commit();
        }
    }

}
```

STM Transaction

```
public final class STM {
    private STM() {}

    public static Object commitLock = new Object();

    public static void transaction(TransactionBlock block) {
        boolean committed = false;
        while (!committed) {
            Transaction tx = new Transaction();
            block.setTx(tx);
            block.run();
            committed = tx.commit();
        }
    }

}
```

No exceptions handling (6)

No nested transactions (7)


BRO...
IS IT WORKING?

What the problem?

```
boolean commit() {
    synchronized (STM.commitLock) {
        // ...
        if (isValid) {
            for (Ref ref : toUpdate) {
                ref.value = inTxMap.get(ref);
                ref.revision = revision;
            }
        }
        // ...
    }
}
```

What the problem?

```
boolean commit() {
    synchronized (STM.commitLock) {
        // ...
        if (isValid) {
            for (Ref ref : toUpdate) {
                ref.value = inTxMap.get(ref);
                ref.revision = revision;
            }
        }
        // ...
    }
}
```

Memory switch is atomic

Two Memory switches are not atomic

RefTuple

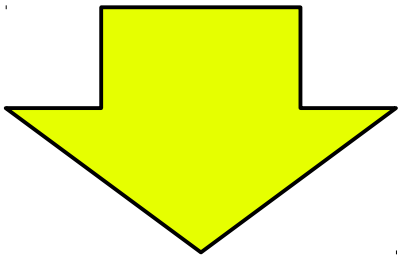
```
public class RefTuple<V, R> {
    V value;
    R revision;

    public RefTuple(V v, R r) {
        this.value = v;
        this.revision = r;
    }

    static <V, R>  RefTuple get(V v, R r) {
        return new RefTuple<V, R>(v, r);
    }
}
```

Ref v0.5

```
public final class Ref<T> {
    RefTuple<T, Long> content;

    public Ref(T value) {
        content = RefTuple.get(value, 0);
    }

    // ...
}
```

Transaction.commit()

```
for (Ref ref : toUpdate) {
    ref.value = inTxMap.get(ref);
    ref.revision = revision;
}
```

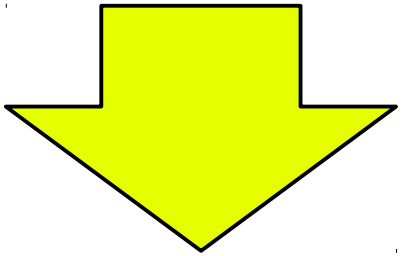


```
for (Ref ref : toUpdate) {
    ref.content =
        RefTuple.get(inTxMap.get(ref), revision);
}
```

Transaction.get()

```
<T> T get(Ref<T> ref) {
   if (!inTxMap.containsKey(ref)) {
       inTxMap.put(ref, ref.value);
       if (!version.containsKey(ref)) {
           version.put(ref, ref.revision);
       }
   }
   return (T)inTxMap.get(ref);
}
```

2 non-sync reads



```
<T> T get(Ref<T> ref) {
    if (!inTxMap.containsKey(ref)) {
        RefTuple<T, Long> tuple = ref.content;
        inTxMap.put(ref, tuple.value);
        if (!version.containsKey(ref)) {
            version.put(ref, tuple.revision);
        }
    }
    return (T)inTxMap.get(ref);
}
```

// TODO

1. Support return value in TX
2. Avoid **this.getTx()** repetition
3. Fair **GlobalContext**
4. Snapshot Isolation
5. Support ref history
6. Support exceptions
7. Support nested transactions

STM Algorithms

Multi-**V**ersion **C**oncurrency **C**ontrol

Lazy **S**napshot **A**lgorithm

Transactional **L**ocking **2**

Practical Advices

→ avoid side-effects

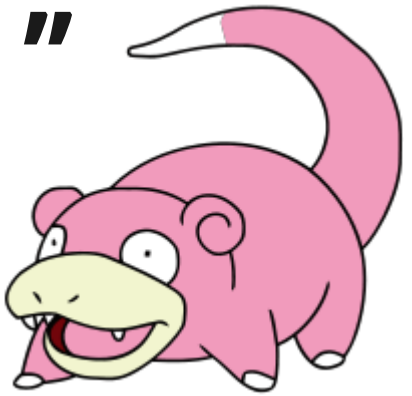
→ avoid short/long transactions

→ separate as much as possible outside of transaction

→ ref value must be immutable

OR STM WON'T HELP YOU

“But STM is slow...”



No!

TM/GC Analogy

Transactional Memory is to shared-memory concurrency

as

Garbage Collection is to memory management.

Non-STM

→ atomics

→ immutability

→ avoid shared state

→ lock-free data structures

Links

- STM In Clojure
- Locks, Actors and STM in pictures
- The Dining Philosophers Problem
- Clojure refs
- TM/GC Analogy [PDF]
- STM in Scala
- 7 Concurrency Models in 7 weeks