Fachhochschule Nordwestschweiz

Software Transactional Memory (STM)

- Motivation
- ScalaSTM
- Implementation

The state of the art in concurrent programming

```
public void transfer(Account from, Account to, double amount)
                                throws InactiveException, OverdrawException {
    Account x, y;
    if (from.getNumber().compareTo(to.getNumber()) < 0) {
        x = from; y = to;
    } else {
        x = to; y = from;
    }
    synchronized (x) {
        synchronized (y) {
            from.withdraw(amount);
            try {
                to.deposit(amount);
            } catch (InactiveException e) {
                from.deposit(amount);
                throw e;
            }
        }
    }
}
```

Software Transactional Memory (STM)

- **Declaratively express what to do atomically, not how to do it!**

```
def transfer(from: Account, to: Account, amount: Double): Unit = {
  atomic { implicit tx =>
    from.withdraw(amount)
    to.deposit(amount)
  }
}
```

Software Transactional Memory (STM)

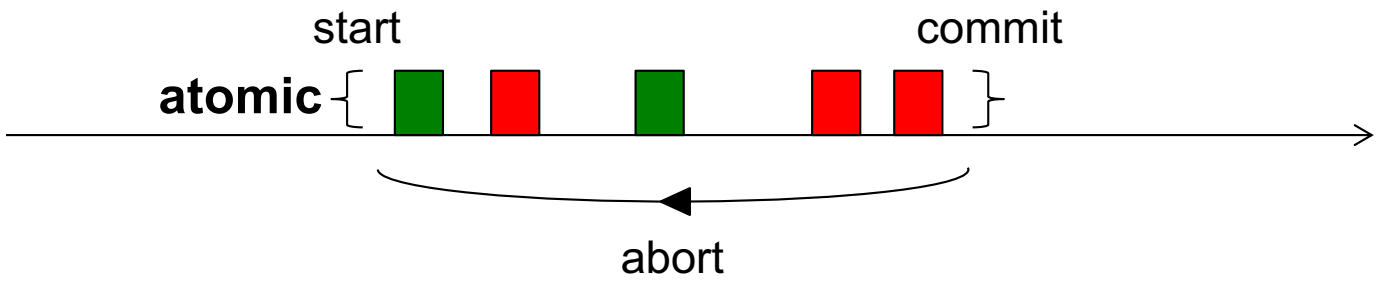
- **Coordination mechanism for shared memory concurrency**
- **Coordinates access to heap locations (as opposed to db tx)**

A transaction is

- **a sequence of read and write operations to shared memory**
- **which occurs logically at a single instant in time**
- **whereby intermediate states are not visible to other transactions**

Semantics of a transaction:
- **A**tomic: All or nothing
- **C**onsistent: Preserves invariants (programmers responsibility)
- **I**solation: No interference with concurrent txs

Optimistic Concurrency Control



1. **Begin transaction**
2. **Tentative state changes during execution**
3. **Conflict encountered?**
   - **Yes: Abort and retry**
   - **No: Commit, make changes permanent and visible**

Atomics: Single Reference Transactions

```
class Account {
    @volatile
    private bal: Int = 0;

    def deposit(amount: Int): Unit = {
        while (true) {
            val oldBal = bal;                // read current value
            val newBal = oldBal + amount; // compute new value
            if (compareAndSet(addr(bal), oldBal, newBal)) {
                return; // commit successful -> return
            }
            // conflict -> retry
        }
    }

    def withdraw(int amount): Unit = {/* analogous to deposit */}
}
```

STM: Multi Reference Transactions

```
public class STMConceptually {
   public void atomicTransfer(double amount,
                              Account from, Account to) {
      while(true) {
         int oldFromBal = from.bal;
         int newFromBal = oldFromBal - amount;

         int oldToBal = to.bal;
         int newToBal = oldToBal + amount;

         if(STM.multiCAS(addr(from.bal), oldFromBal, newFromBal,
                         addr(to.bal)  , oldToBal  , newToBal)) {
            return;
         }
         // retry
      }
   }
}
```

Software Transactional Memory (STM)

- Motivation
- ScalaSTM
- Implementation

ScalaSTM: Ref-s and atomic

- **ScalaSTM consists of two fundamental parts**
  - **Ref**: Mutable cell (wrapper), access is coordinated by the STM system

```
val as = Ref(Map[String, Account]()) /* Factory */
```

  - **atomic**: Executes the given function within a transaction

```
def atomic[Z](block: InTxn => Z): Z  /* simplified */
```

- **Example**

```
private val bal: Ref[Double] = Ref(0.0)

def withdraw(a: Double) {
  atomic { implicit tx => bal.set(bal.get - a) }
}
```

- **API: http://nbronson.github.io/scala-stm/**

ScalaSTM: Ref

- **A Ref is a mutable reference to immutable state**
  - Access to referenced state is coordinated by STM system

Refs have to be the only mutable abstraction within a program

- **Creation**

```
val ref = Ref[Type](initValue)
```

- **Read**

```
val insideRef = ref() | val insideRef = ref.get
```

- **Write**

```
ref() = newValue      | ref.set(newValue)
```

- **Transform**

```
ref.transform(insideRef => f(insideRef))
```

(read|write|transform) compile only within an atomic block!

ScalaSTM: atomic

- **atomic takes a function and executes it transactionally**

```
import scala.concurrent.stm._

atomic {
  implicit txn => code
}
```

context: `implicit txn`

Code block: `code`

- **The function takes a parameter of type InTxn**
  - This parameter
    - provides a context for the transaction (consumed by Ref accessors)
    - has to be marked implicit

ScalaSTM: Single Operation Transactions

- **Ref.View provides convenient single operation transactions**

```
val ref: Ref[Int] = Ref(1)
val refView: Ref.View[Int] = ref.single
```

- **Mental model**

```
val ref = Ref(1)
ref.single.operation(args)
```

acts like

```
val ref = Ref(1)
atomic{ implicit tx =>
  ref.operation(args)
}
```

- **Example**

```
val ref = Ref(1)
ref.single.set(2)
```

acts like

```
val ref = Ref(1)
atomic{ implicit tx =>
  ref.set(2)
}
```

Atomic vs Synchronized

```
atomic { implicit txn =>
  doA(); doB()
}
```

- delimits a transaction
- no explicit locking
- atomic to all other atomic blocks executed by another thread
- no deadlocks are possible
- composable

```
mutex.synchronized {
  doA(); doB()
}
```

- delimits a critical section
- acquires a specific lock
- atomic to other synchronized blocks that acquire the same lock
- nested synchronized blocks may deadlock
- not composable

Non composability of locks

```
class LockAccount(val id: Int) {
  private var balance = 0d

  def withdraw(a: Double) {
    synchronized {
      balance = balance - a
    }
  }
  def deposit(a: Double) {
    synchronized {
      balance = balance + a
    }
  }
}
```

```
class LockBank {
  def transfer(amount: Double,
   from: LockAccount, to: LockAccount) {

    val (fst, snd) =
      if(from.id > to.id) (from,to)
      else (to,from)

    fst.synchronized {
      snd.synchronized {
        from.withdraw(amount)
        to.deposit(amount)
      }
    }
  }
}
```

- **LockBank must know locking convention of Account**
- **Account must expose internals to be reusable**

Composability of atomic

```
class STMAccount(val id: Int) {
  private val balance = Ref(0d)

  def withdraw(a: Double) {
    atomic { implicit txn =>
      balance() = balance() – a
    }
  }
  def deposit(a: Double) {
    atomic { implicit txn =>
      balance() = balance() + a
    }
  }
}
```

```
class STMBank {

def transfer(amount: Double,
   from: STMAccount, to: STMAccount) {
    atomic { implicit txn =>
      to.deposit(amount)
      from.withdraw(amount)
    }
  }
}
```

- **STMBank knows nothing about internals of STMAccount**
- **Composable tx propagation strategy:**
  - **When arriving at an atomic block: Check if tx is already running:**

    Yes: join existing tx

    No: start new tx

Atomicity and Exceptions

```
atomic { implicit txn =>
  to.deposit(amount)
  from.withdraw(amount)
}
```

- **Atomicity makes error recovery easy**

```
def withdraw(amount: Double) {
  if(amount < 0) throw new IllegalArgumentException
  atomic { implicit txn =>
    if(!active()) throw new InactiveException
    if(balance() - amount < 0) throw new OverdrawException
    balance.transform(b => b - amount)
  }
}
```

- **If an exception is raised inside an atomic block**
  - it may be caught inside the atomic block, that's ok
  - if it is not caught, rollback and re-throw
- **Because of atomicity, changes are rolled back in case of an exception, no need to cleanup**

Lifecycle callbacks

- **Txn.afterCommit(handler: Status => Unit)**
  - Executed after a successful commit
  - Used for side effects which should only happen once

```
def transfer(from: STMAccount, to: STMAccount, amount: Double) {
  atomic { implicit txn =>
    to.deposit(amount)
    from.withdraw(amount)
    Txn.afterCommit { _ =>  sendMail(to.email, "You've got $" + amount) }
  }
}
```

- **Txn.afterRollback(handler: Status => Unit)**
  - Executed after rollback
  - Used for compensating actions

Software Transactional Memory (STM)

- Motivation
- ScalaSTM
- Implementation

Transactional Locking II

Setup:

- **Global version counter: Stores version of latest successfully committed tx**
- **Local version stamp: Each Ref is marked with the version of the last successfully committed tx which modified this reference**

1. **Tx start: Store the value of the global version counter local to the new tx (read version [rv])**
2. **Tx body:**
   a. **Before first reading / writing a Ref, make a local working copy of it**
      - **Abort and retry if version(Ref) > read version (NOT ONLY ON FIRST ACCESS)**
   b. **Read and write only to local copy**
3. **Tx commit:**
   a. **Lock all modified Refs (use a timeout to avoid deadlocks)**
   b. **Increment global version counter, store copy local to transaction (write version [wv])**
   c. **Check all Refs again. Abort and retry if**
      - **version(Ref) > read version [rv]**
      - **accessed object is locked (currently updated)**
   d. **Write values and write version [wv] back to the modified Refs**
   e. **Release locks**

http://www.cs.tau.ac.il/~shanir/nir-pubs-web/Papers/Transactional_Locking.pdf

Commit time locking

atomic { withdraw(A,2); deposit(B,2); }

Tx0

global version | 6

A: bal=5 [3]

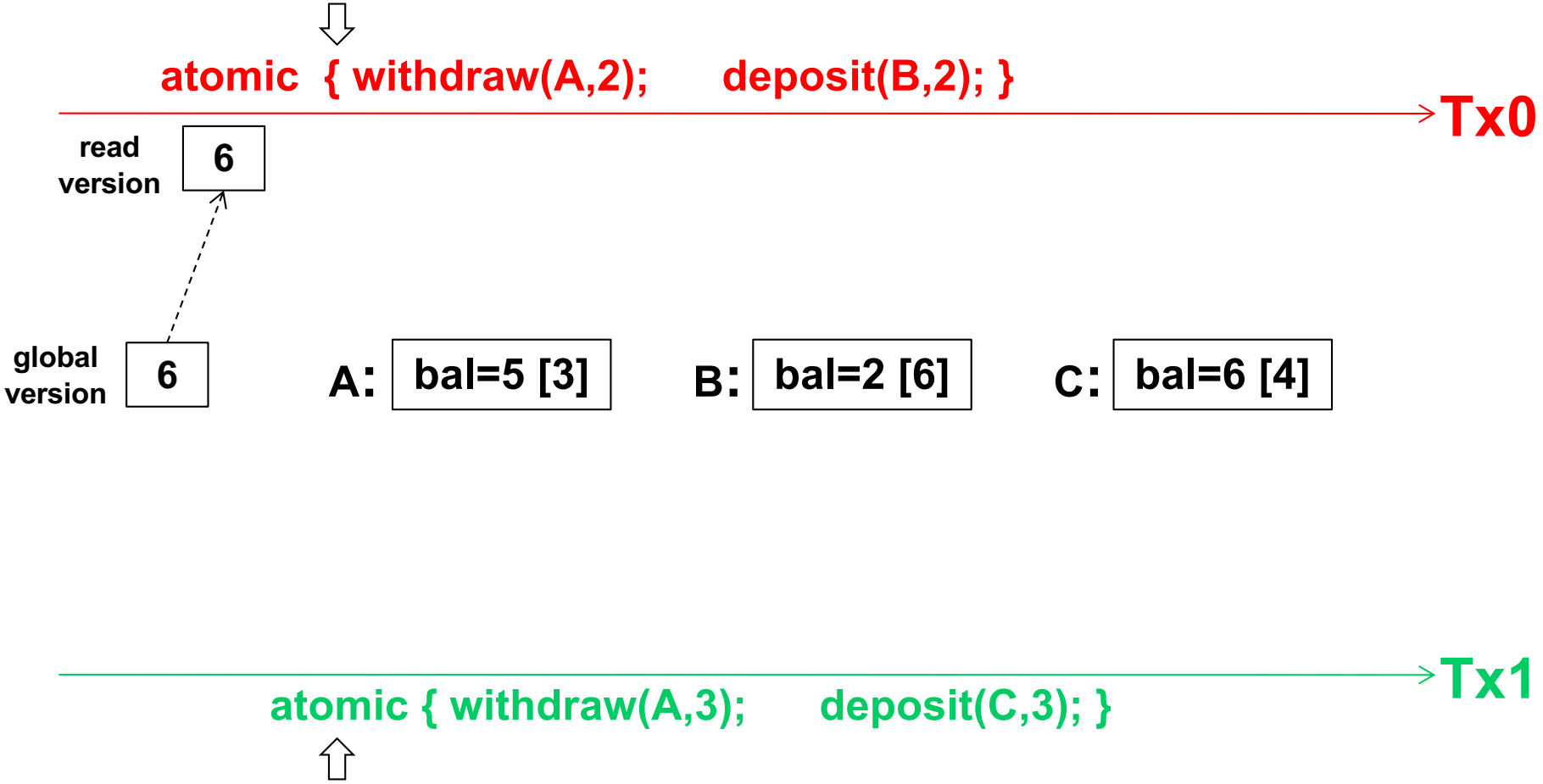
B: bal=2 [6]

C: bal=6 [4]

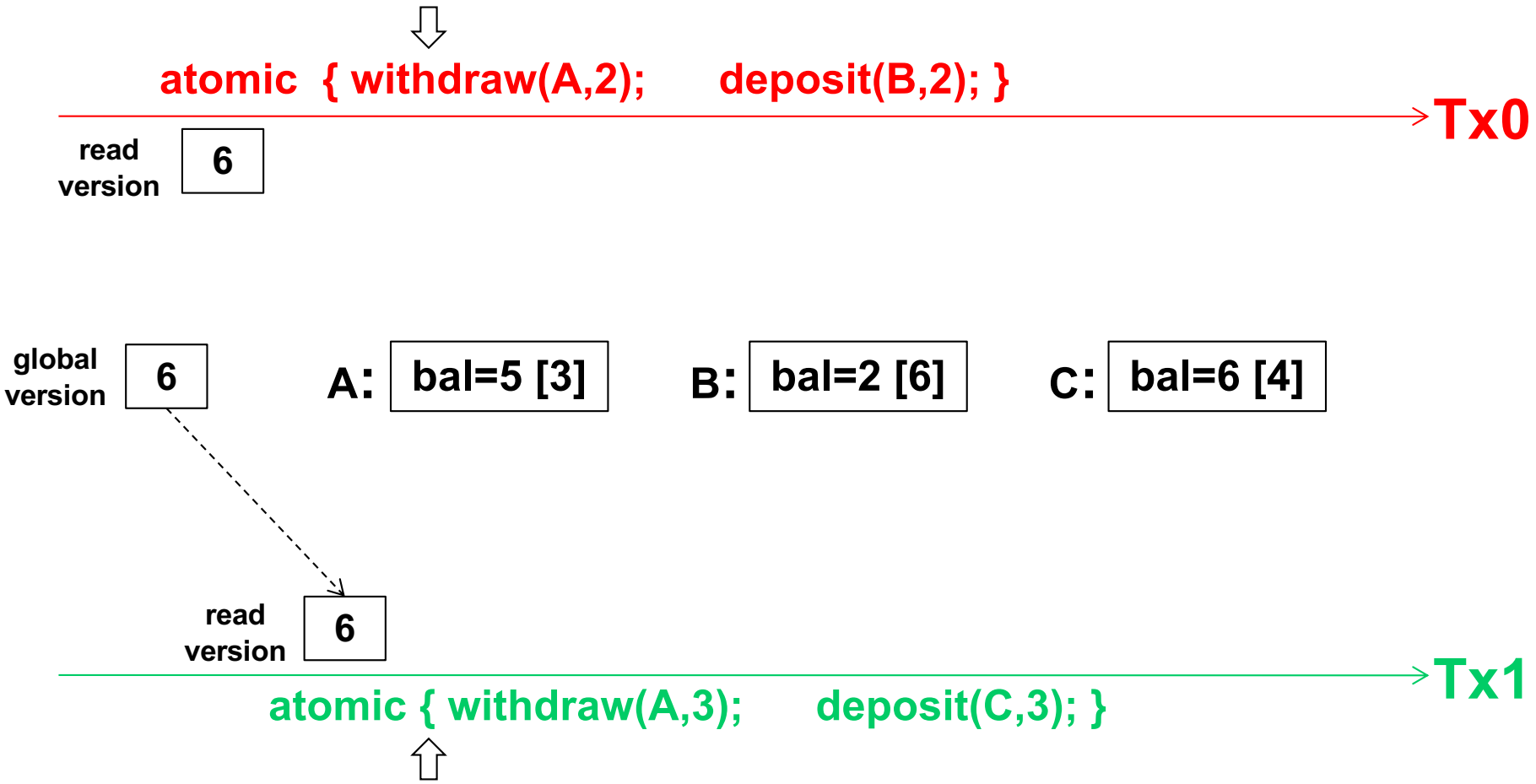
Tx1

atomic { withdraw(A,3); deposit(C,3); }

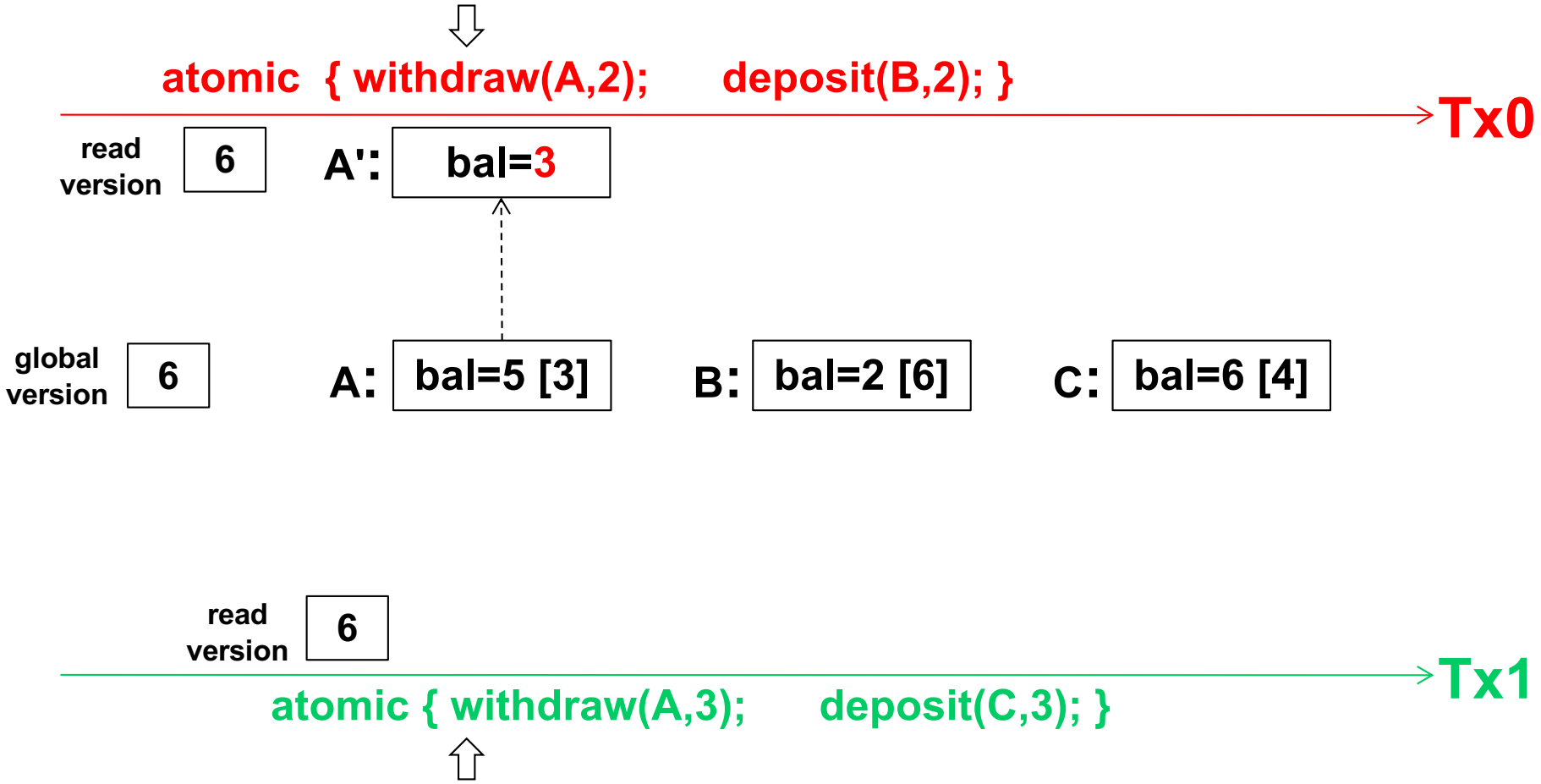
Commit time locking



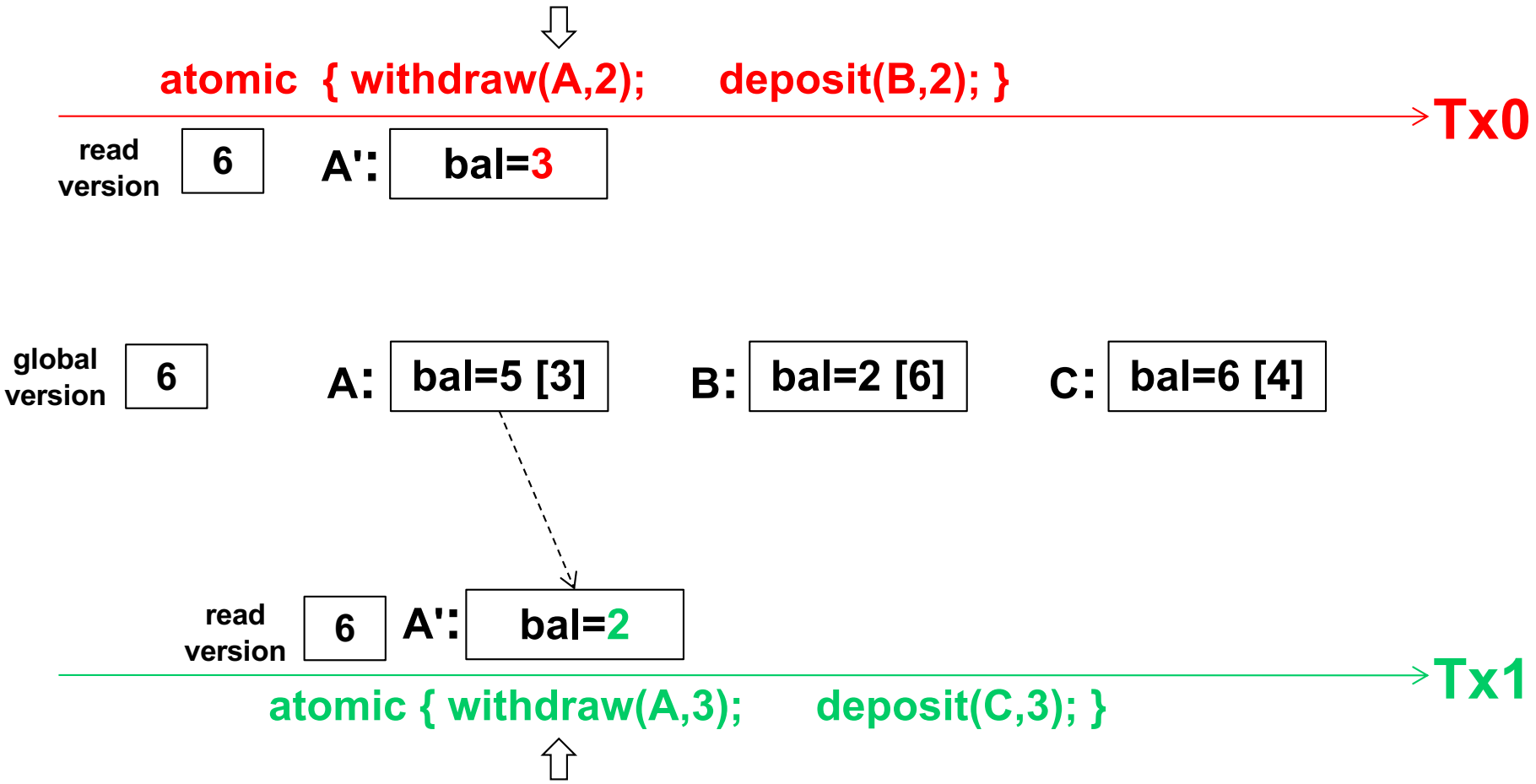
Commit time locking



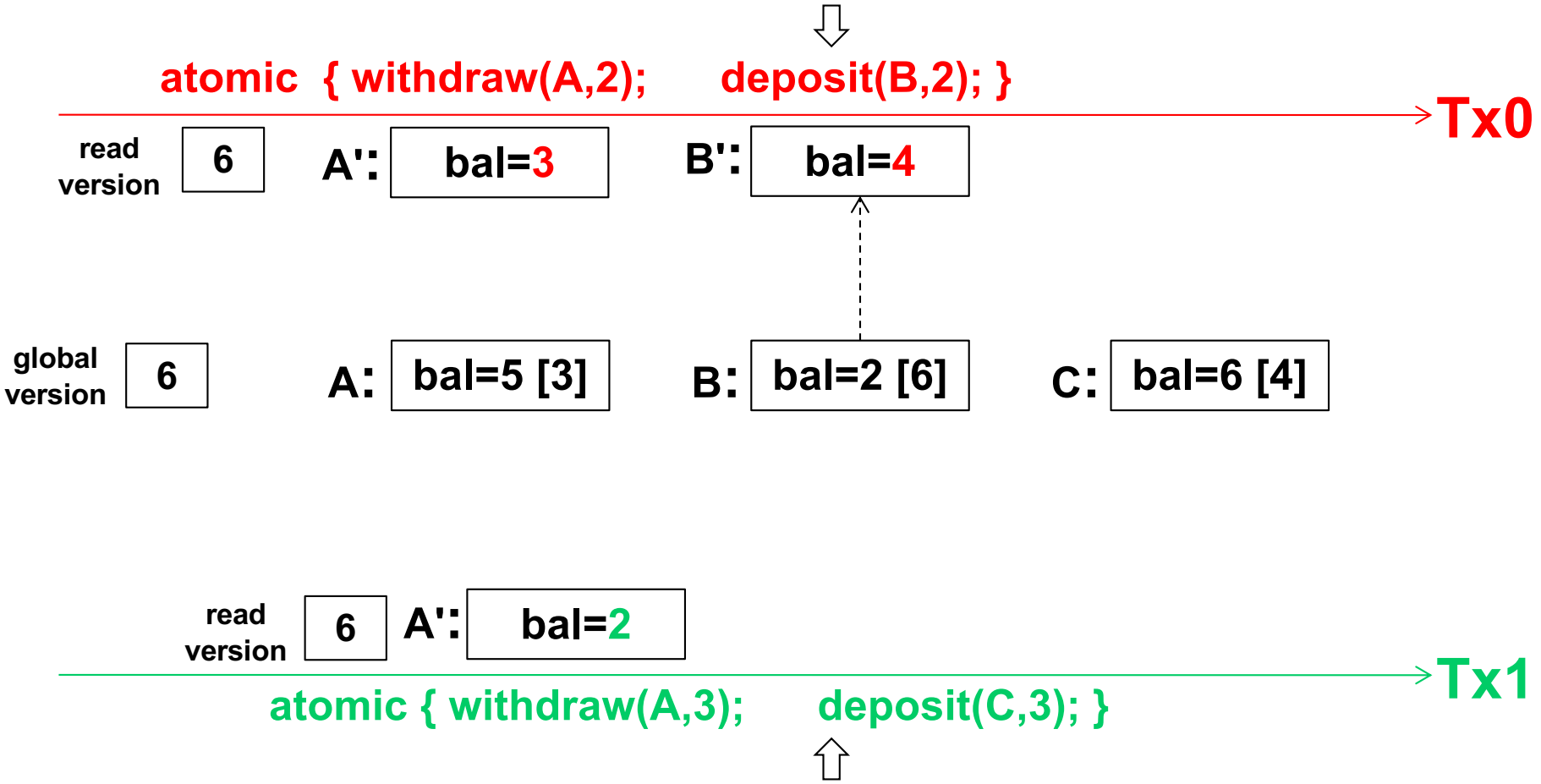
Commit time locking



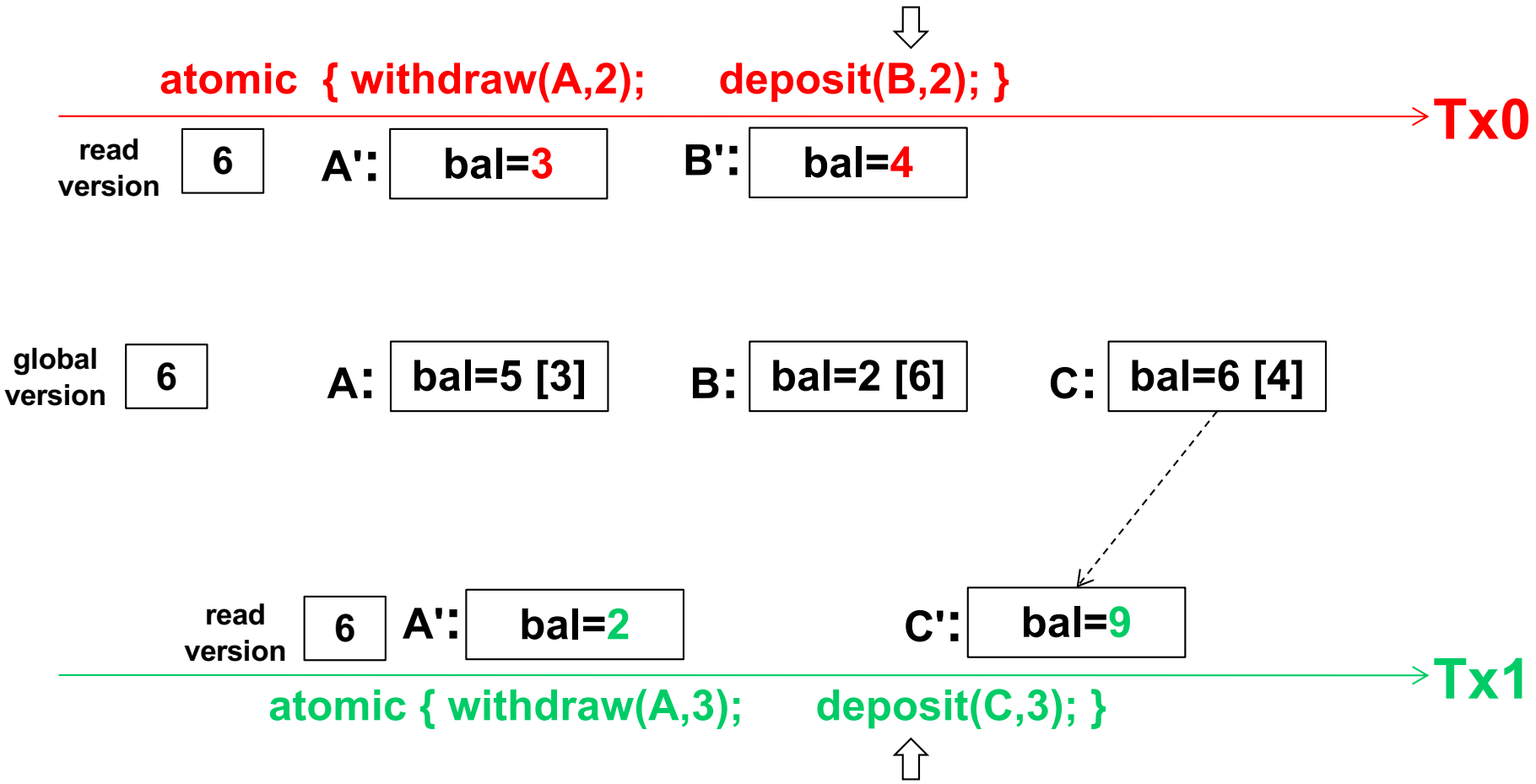
Commit time locking



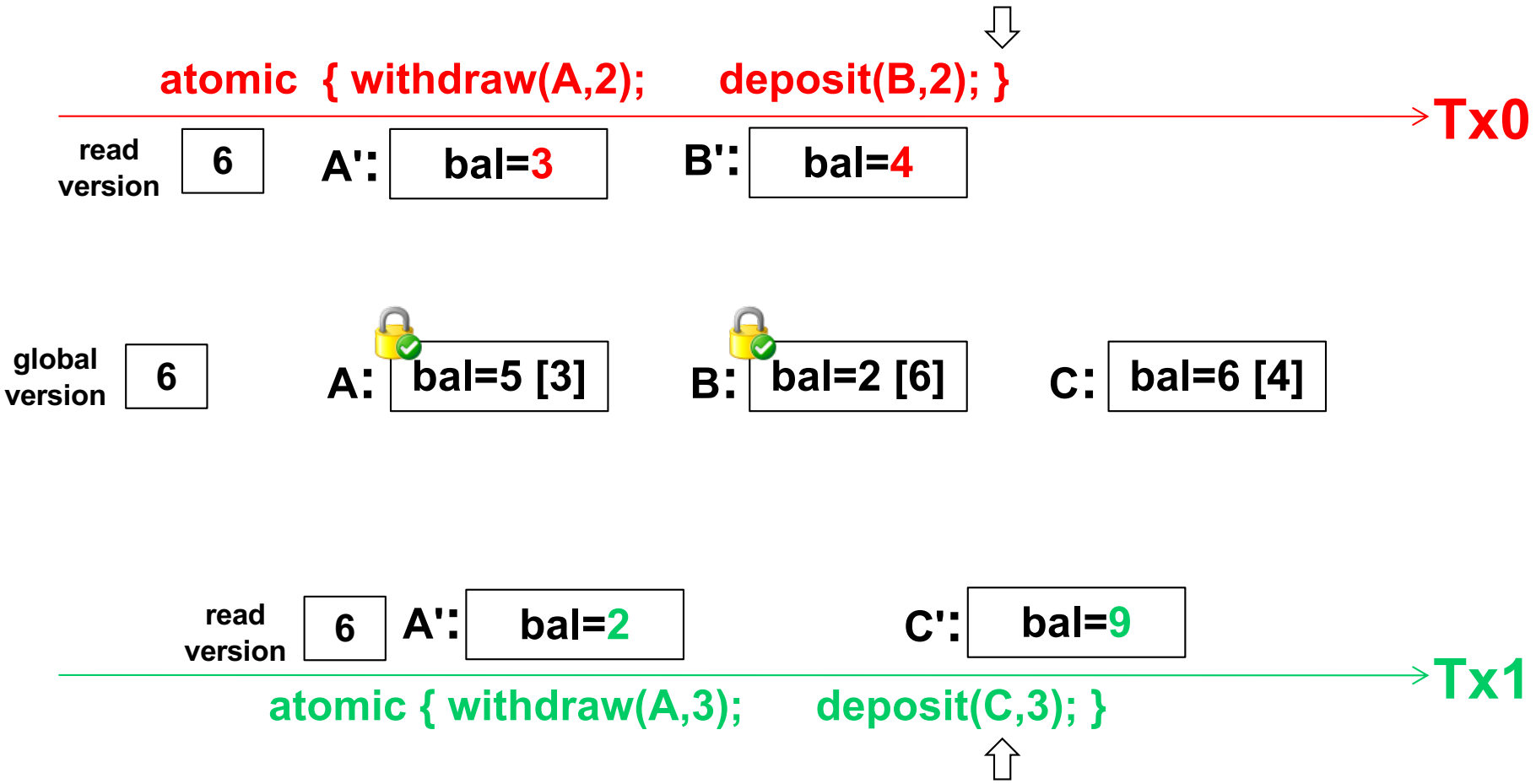
Commit time locking



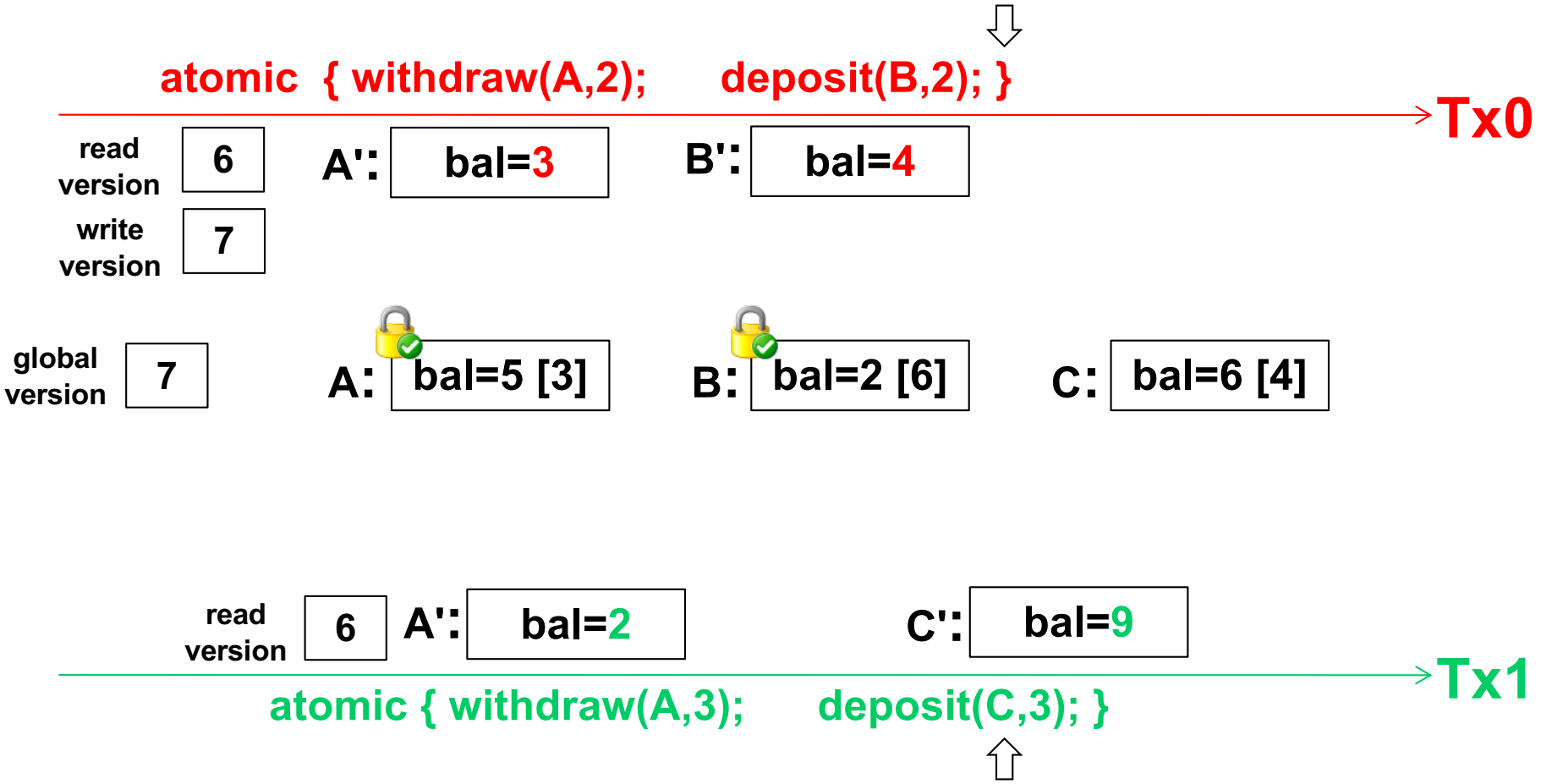
Commit time locking



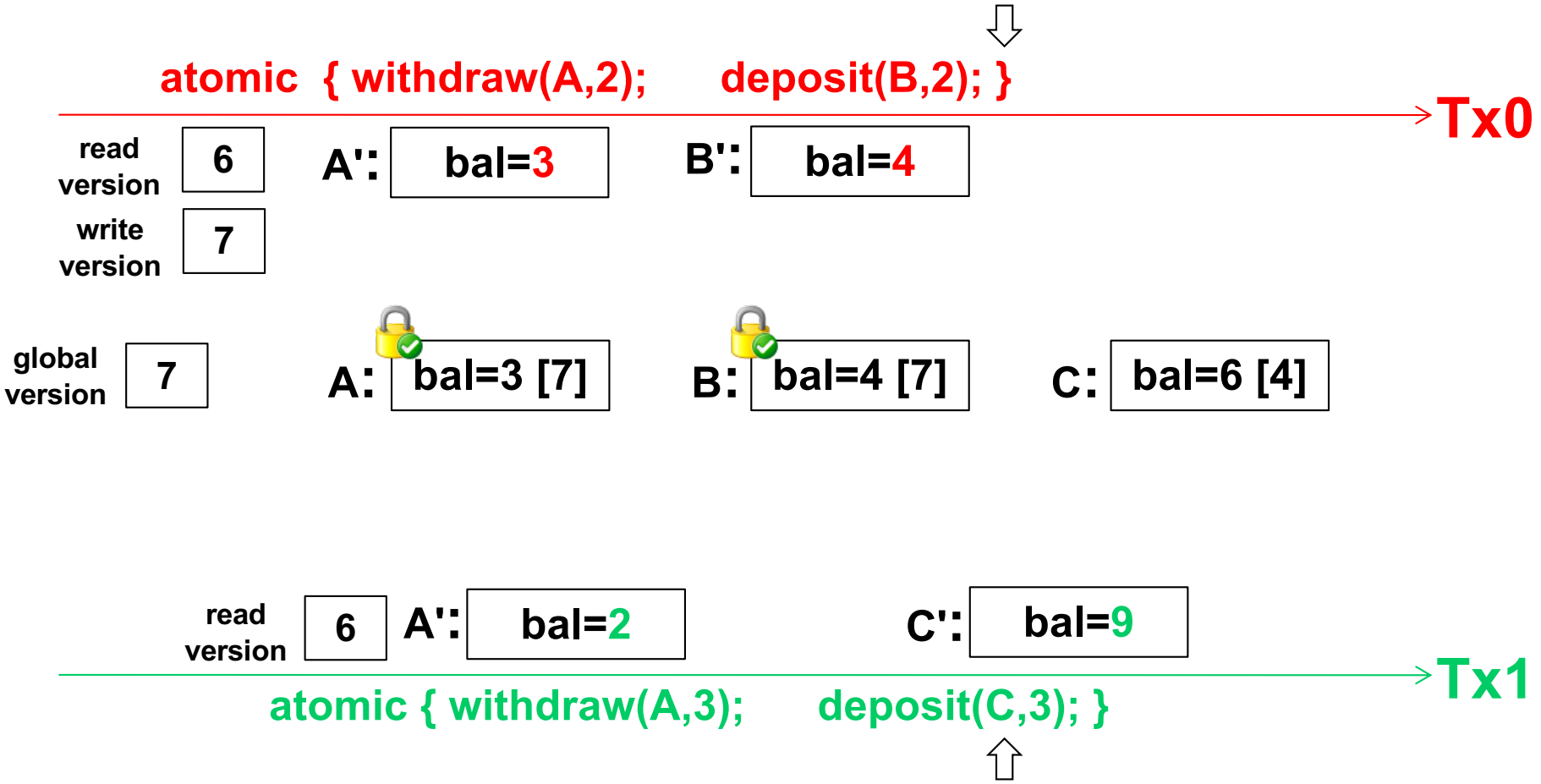
Commit time locking



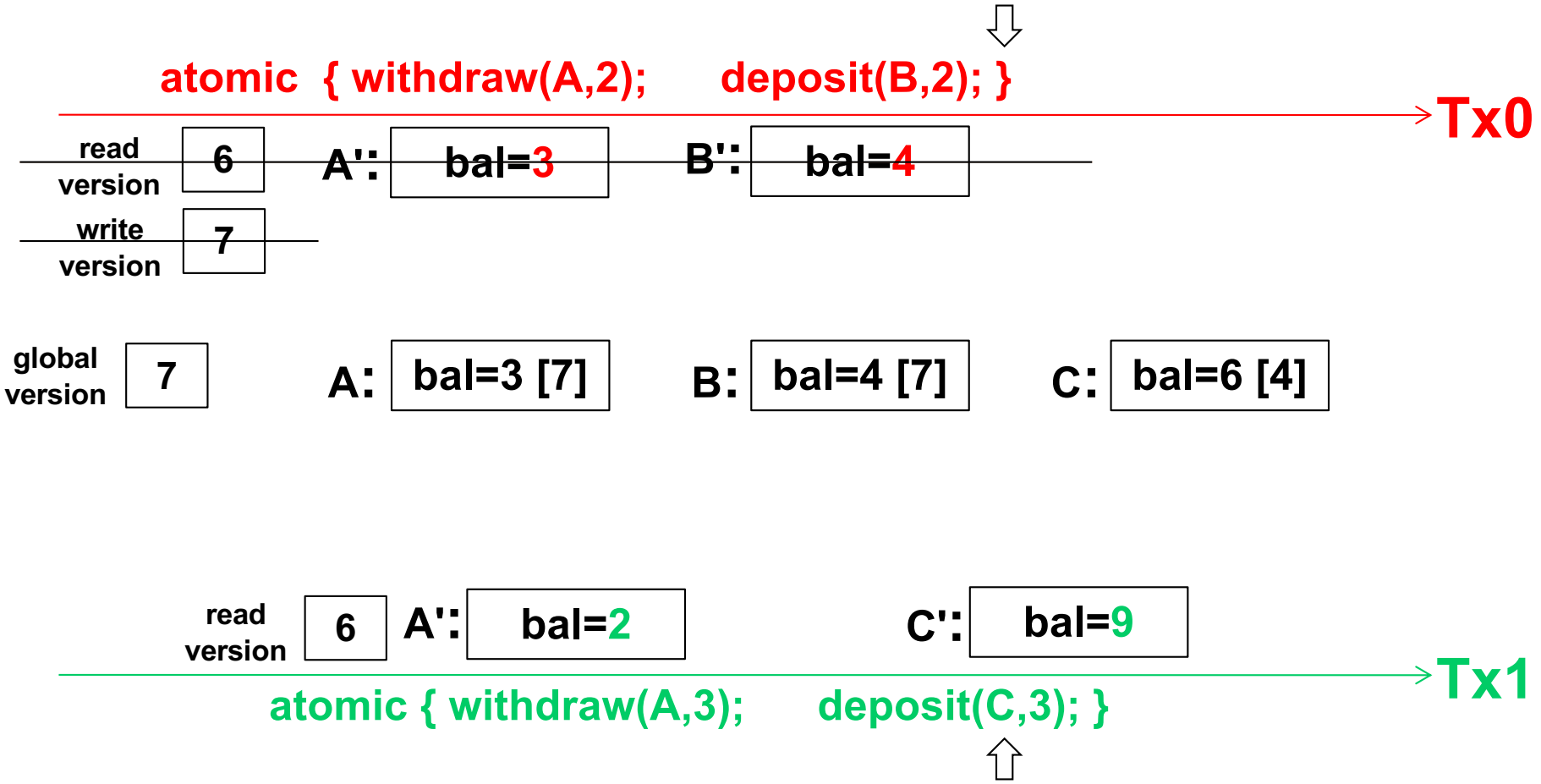
Commit time locking



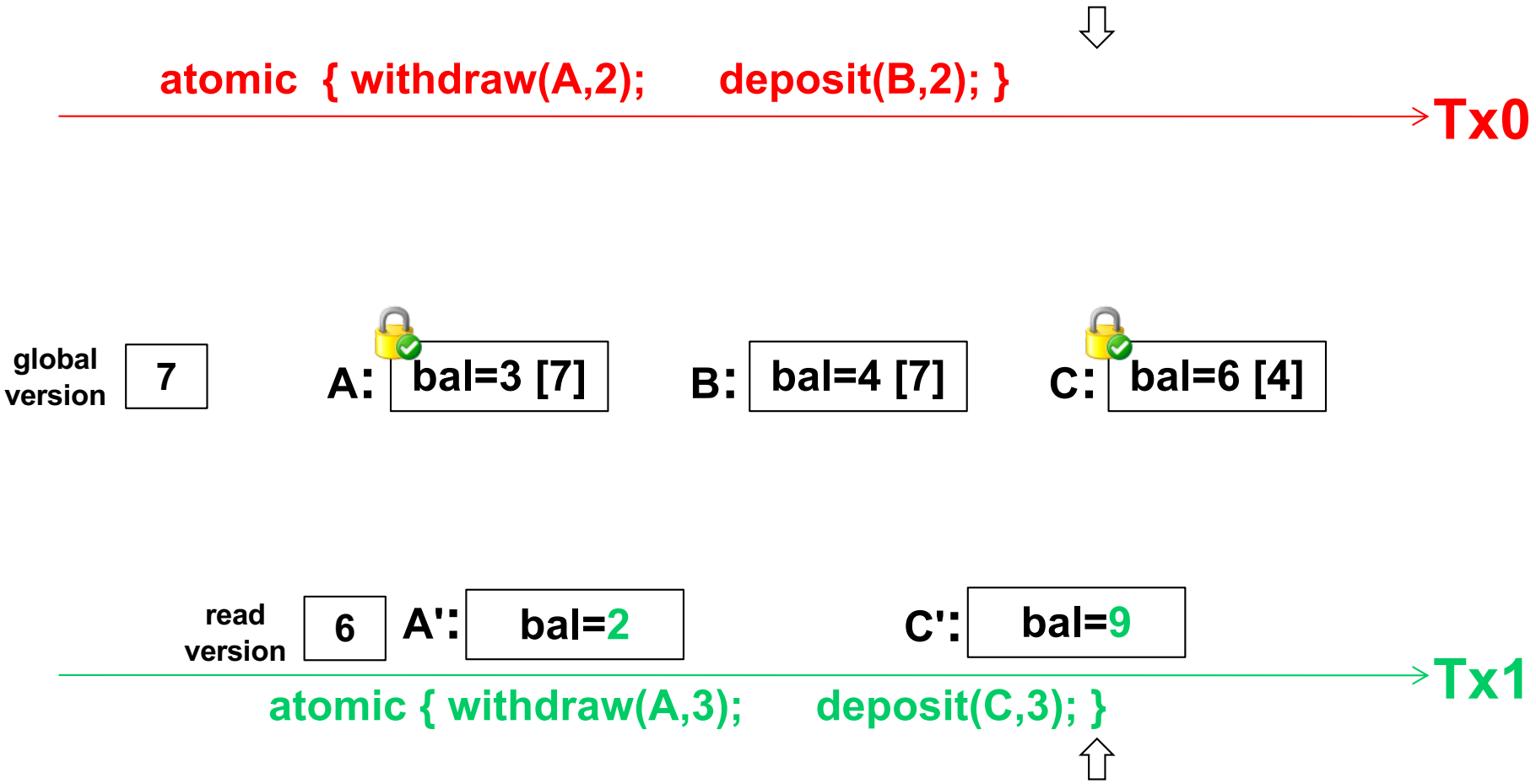
Commit time locking



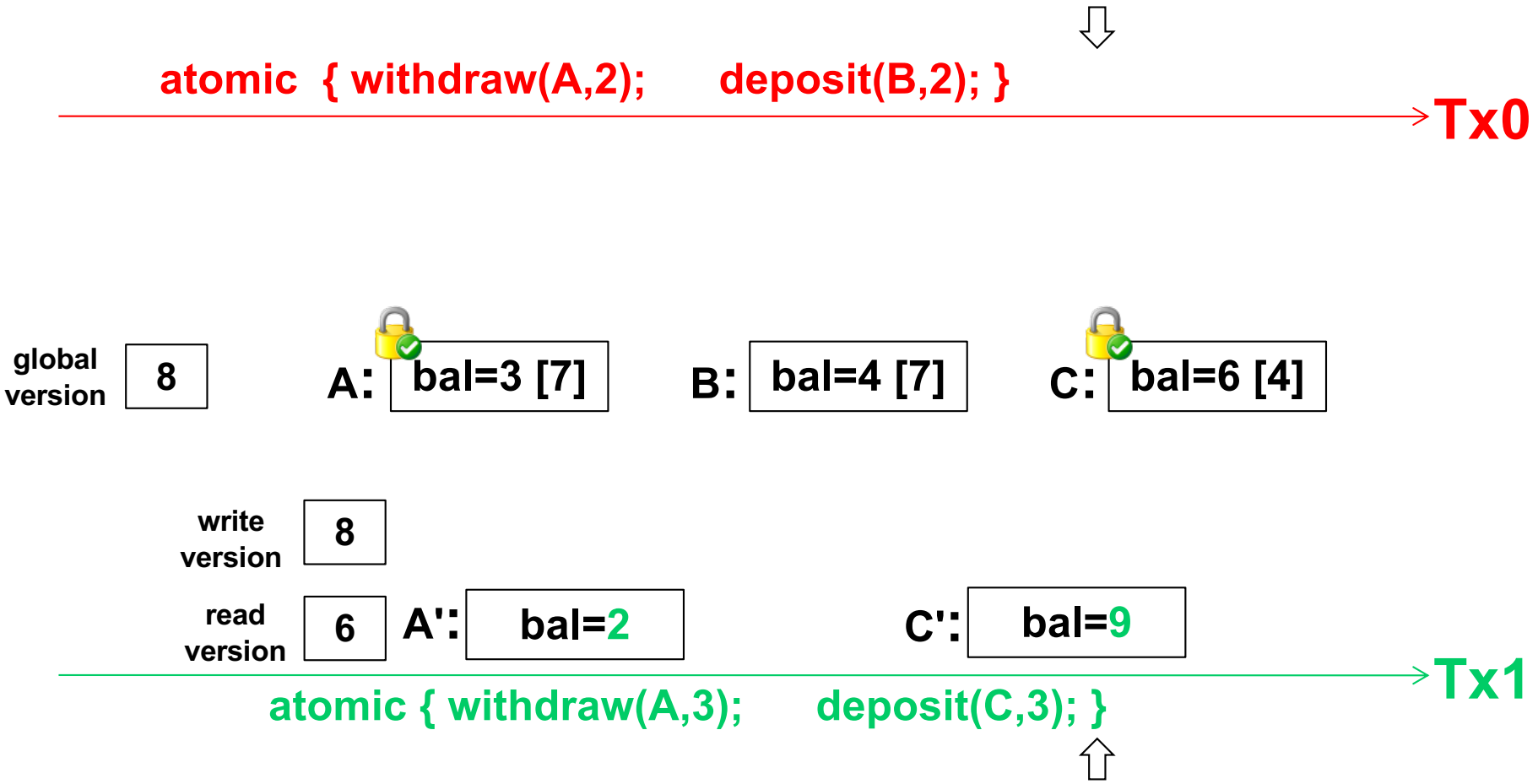
Commit time locking



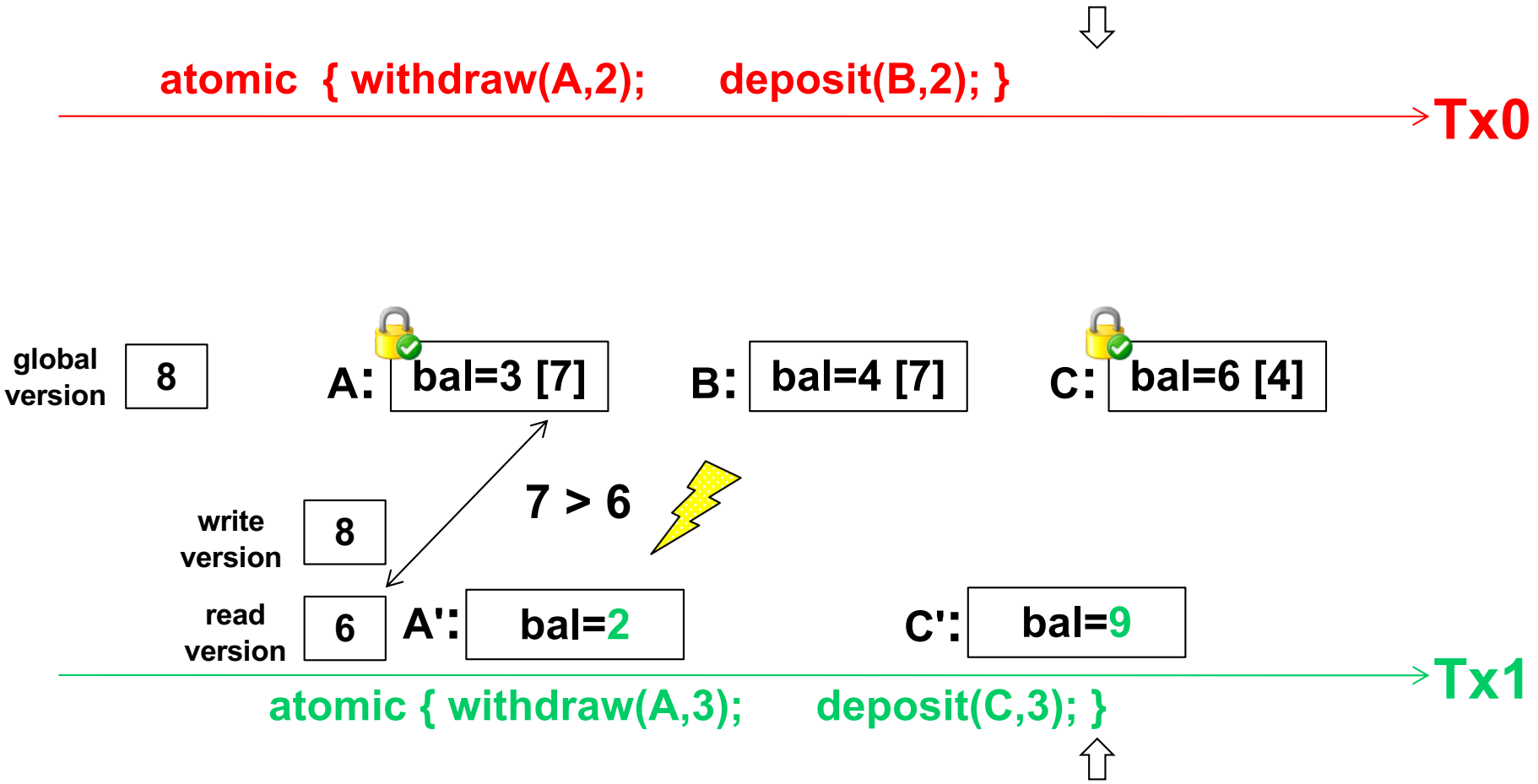
Commit time locking



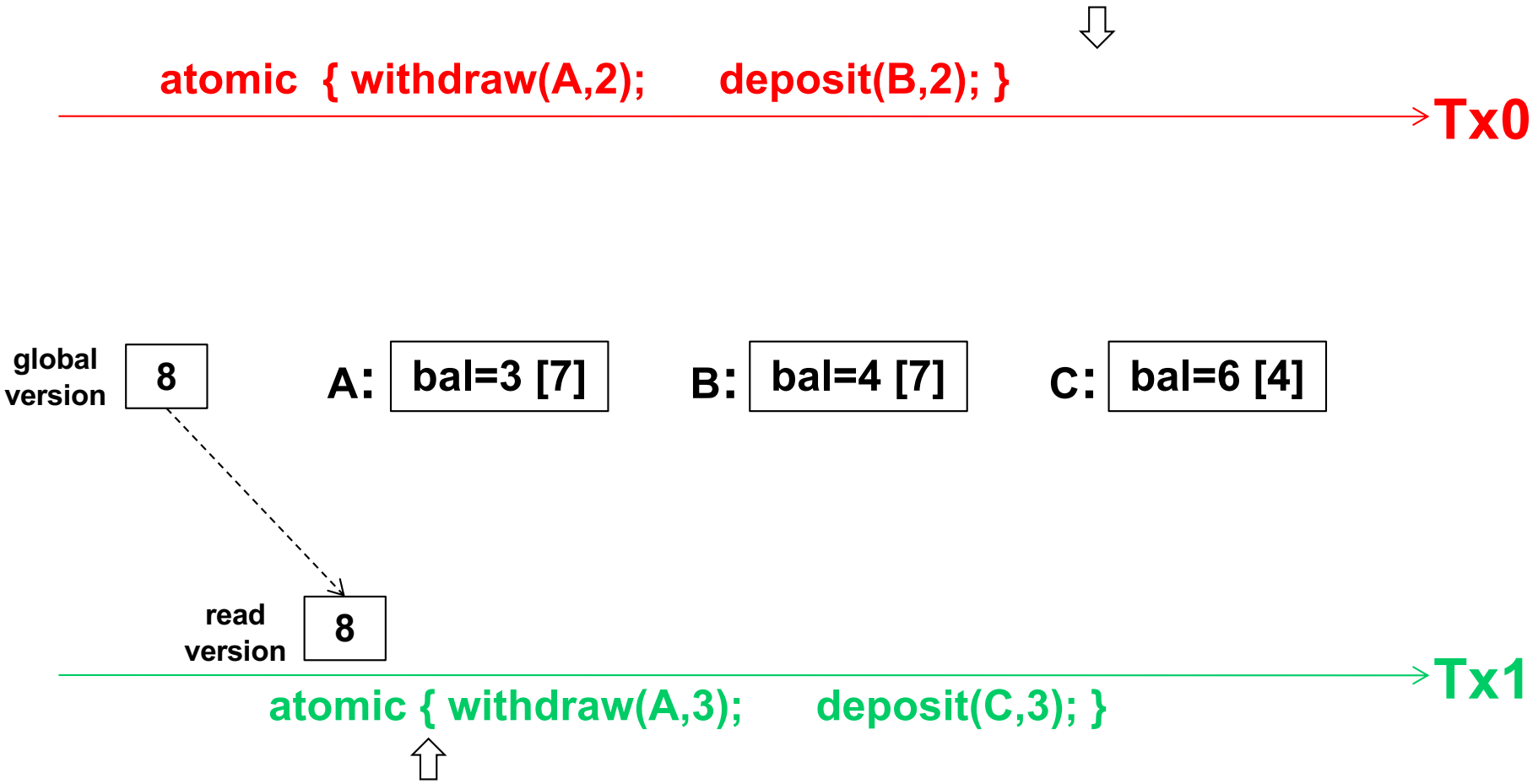
Commit time locking



Commit time locking



Commit time locking



Summary: STM principles

- **A transaction is a sequence of steps executed by a single thread**
  - Atomic: All or nothing semantics
  - Consistent: Preserves invariants (programmers' responsibility)
  - Isolation: Changes made by concurrent transactions are invisible
- **Building blocks:**
  - atomic: delimits a transaction, atomics can be nested
  - Refs: mutable cells, managed by STM library

- **STM eliminates whole classes of low-level errors**
  - It's maybe a bit slower than lowlevel hacker code
  - But it's much easier to get right!