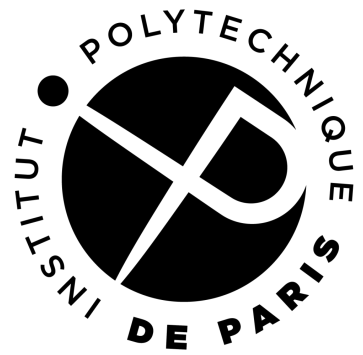


Transactional memory

Master in computer science of IP Paris

Master CHPS of Paris Saclay

Gaël Thomas

Limitation of lock-based algorithms

- Large critical section hampers performance

- Problems with fine-grain locking schemes or lock-free algorithms
  - Makes the code complex and hard to maintain
  - Makes the code difficult to reuse (invariant are often only in the mind of the initial developer, locks have to be taken in a given order)
  - Bugs are hard to find
  - Code is not composable (one data structure => one algorithm)
  - Makes de code difficult to prove

- Idea of transactional memory (TM)
  - Offers a high-level API that simplifies development
  - Tries to be as efficient as lock-free algorithms

Transactional memory: principle

- A single universal construct: the atomic block
  - A block of code that appears to be executed instantaneously

```
a.  atomic {                f.  atomic {
b.    tmp = x;              g.    tmp = x;
c.    tmp = tmp + 1;        h.    tmp = tmp * 2;
d.    x = tmp;              i.    x = tmp;
e.  }                       j.  }
```

Two possible schedulings:
[b,c,e] ***then*** *[g,h,i] (⇒ 44) or [g,h,i]* ***then*** *[b,c,e] (⇒ 43)*

- Advantages
  - Simplifies the code: we don't have to know which locks we have to take (and in which order we have to take the locks) to access a variable
  - Avoids many bugs (deadlocks, starvation)

From locks to TM

- At high-level
  - Transform each critical section by an atomic block
  - Remove the underlying locks

```
synchronized(o)
  if(!x) {
    x = true;
    doSomething();
  }
}
```

```
atomic {
  if(!x) {
    x = true;
    doSomething();
  }
}
```

From condition variables to TM

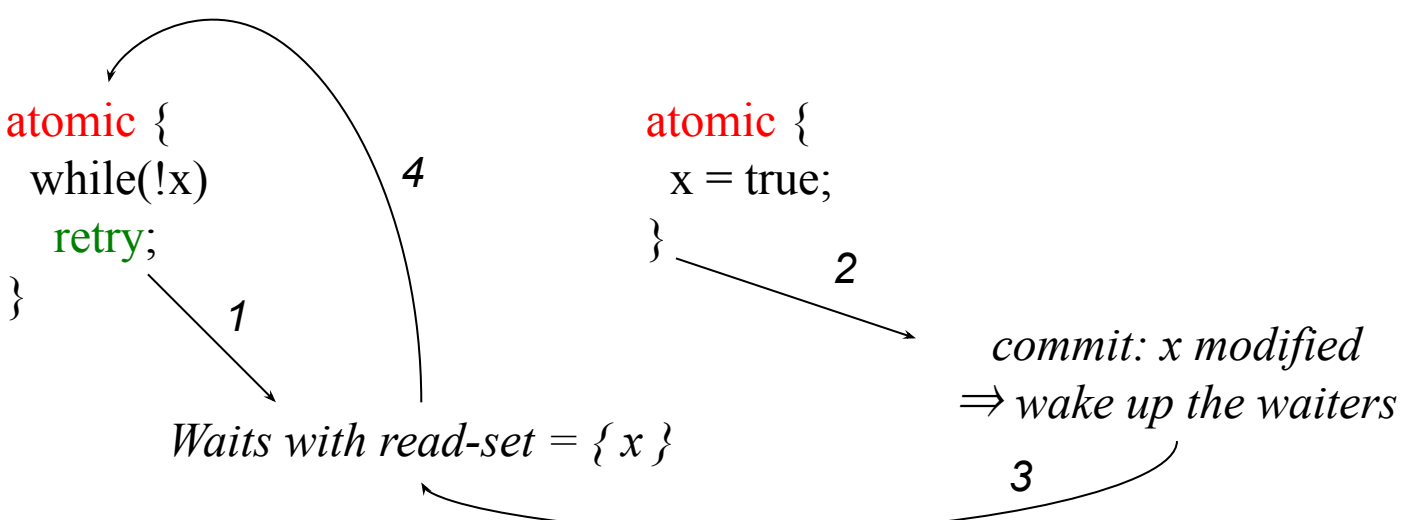
- We often use locks with variable conditions to wait for an event
- Transactional memory provides a notion of retry [Harris'05]
  - Wait until a read variable is modified

```
synchronized(o) {
  while(!x)
    o.wait()
}
```

```
synchronized(o) {
  x = true;
  o.notify();
}
```

```
atomic {
  while(!x)
    retry;
}
```

```
atomic {
  x = true;
}
```



Composability

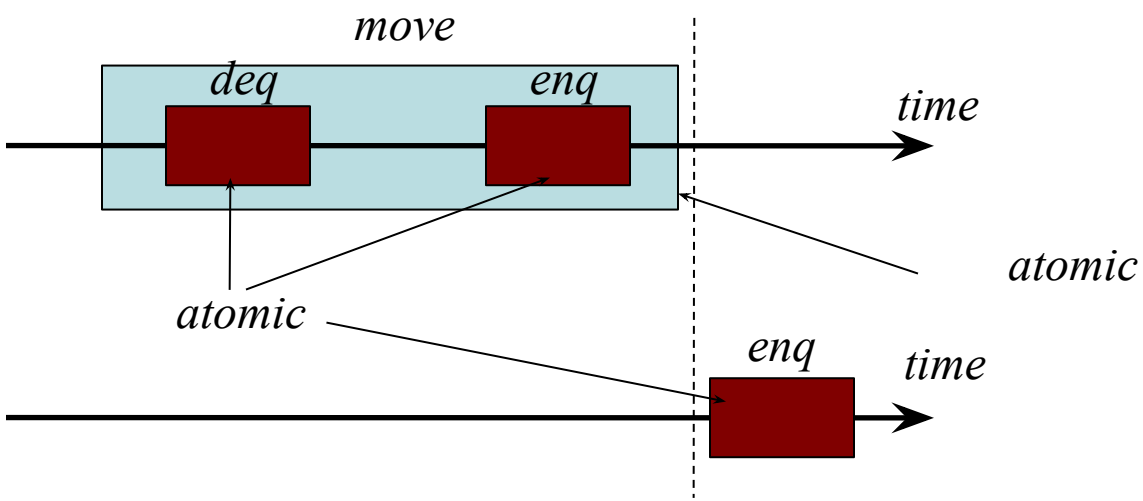
If A and B are two atomic blocks, we can easily compose them

Example: a queue

```
atomic move(Queue dst, Queue src) {
  Elmt e = src.deq();
  dst.enq(e);
}
```

atomic Elmt deq()

atomic void enq(Elmt e);



Composability and retry (1/2)

A queue implementation with retry

```
class Queue {
  LinkedList<Elmt> queue;
}
```

```
void enq(Elmt e) {
 atomic {
  if(queue.size() == MAX_SIZE)
   retry;
  queue.addLast()
 }
}
```

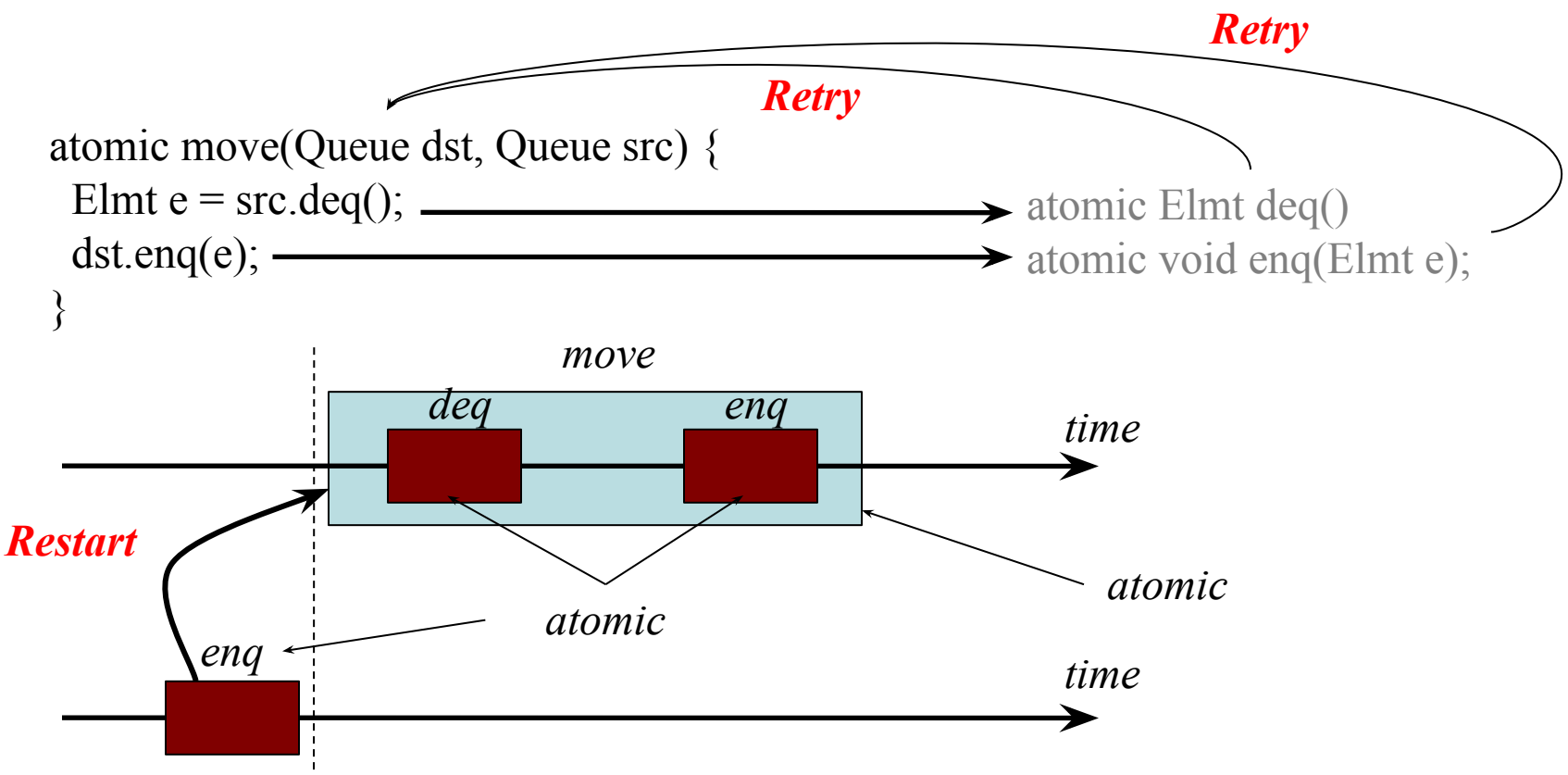
Wake up the waiting transaction

```
Elmt deq() {
 atomic {
  if(queue.empty())
   retry;
  return queue.removeLast()
 }
}
```

***Wait** until one of the read variable is modified and **restart** the transaction*

Composability and retry (2/2)

Retry is also composable: retry restarts from the outer transaction



Composability by alternative

Problem: how to reuse a blocking queue to implement a non-blocking one

Solution: the **orElse** construct

```
Elmt deqNoWait() {
  atomic {            Execute first block
    return deq();     In case of  retry
  } orElse {          Continue with the orElse block
    return null;      If retry again, continue with next orElse
  }                   block or restart from the beginning
}
```

Design of a TM runtime

- Pessimistic solution: use a single lock
  - Acquire the lock when the atomic block starts
  - Release the lock when the atomic blocks ends
  - => often especially inefficient!

- Optimistic solution: abort in case of conflict
  - Execute an atomic block without taking a lock, as a transaction in DB
  - In case of conflict, abort the transaction

- But, what is a conflict?
  - A conflict appears when a transaction cannot execute atomically
    - another transaction Y can observe an ephemeral state that only exists inside a transaction X
    - One of the variable read by a transaction X is modified by another transaction Y during the execution of X

Read-write conflict

- Read-write conflicts
  - Let X and Y be two transactions and A a variable
  - Double write with a reader
    - X writes b in A and then c in A
    - Y reads b from A, while b would never have existed if X had executed atomically
  - Double read with a writer
    - X writes b in A (writer)
    - Y reads a from A and then b from A, which means that Y didn't execute atomically (X is executed "during" Y, which is impossible if Y is atomic)

- In case of conflict, we can/have to abort the reader, the writer or both

Two main possible designs

- Deferred update (redo log)
  - X writes in a redo log
  - If X commits, applis the redo log to main memory
  - => more work in case of commit
  - => avoids by design the double write conflict, we only have to handle double read conflict (read twice a variable modified by another transaction)

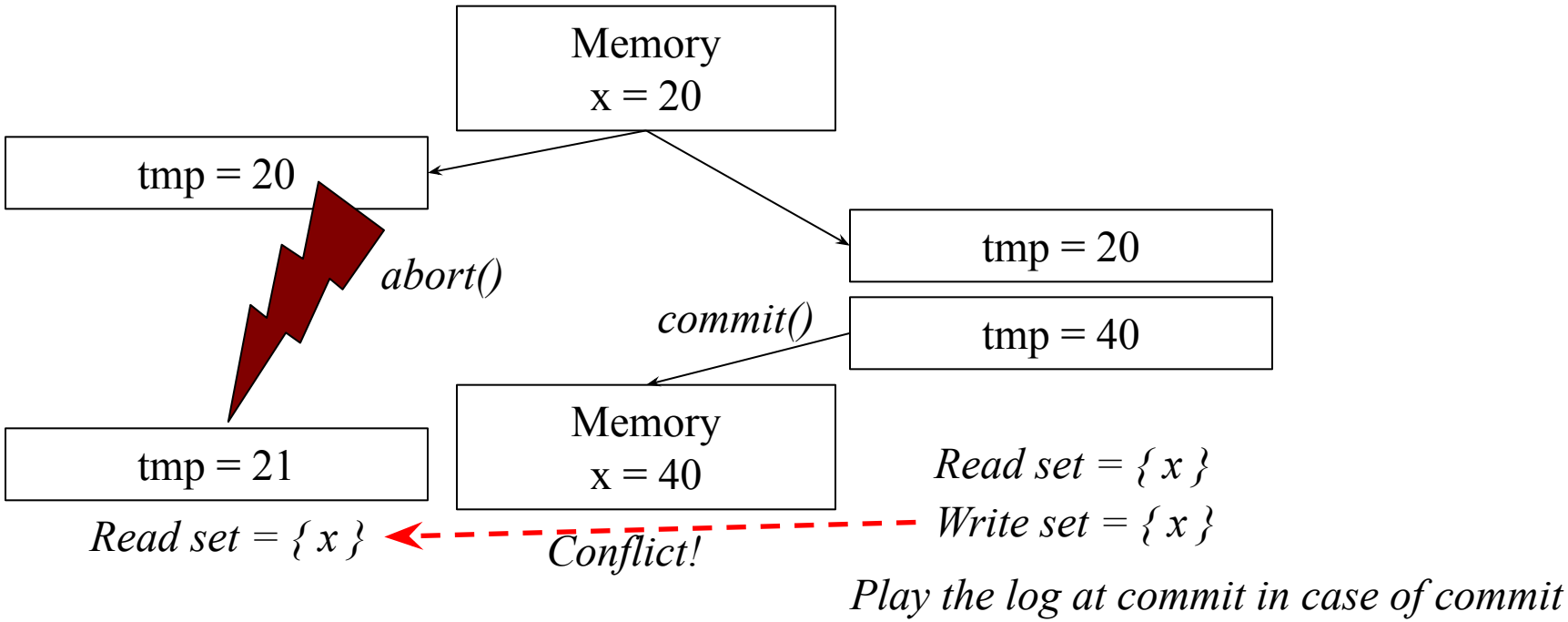
- Immediate (undo log)
  - X writes in main memory and in an undo log
  - If X aborts, undo the operations recorded in the undo log
  - => more work in case of abort
  - => subject to both double write conflicts and double read conflicts

Deferred-update TM

Efficient if many aborts

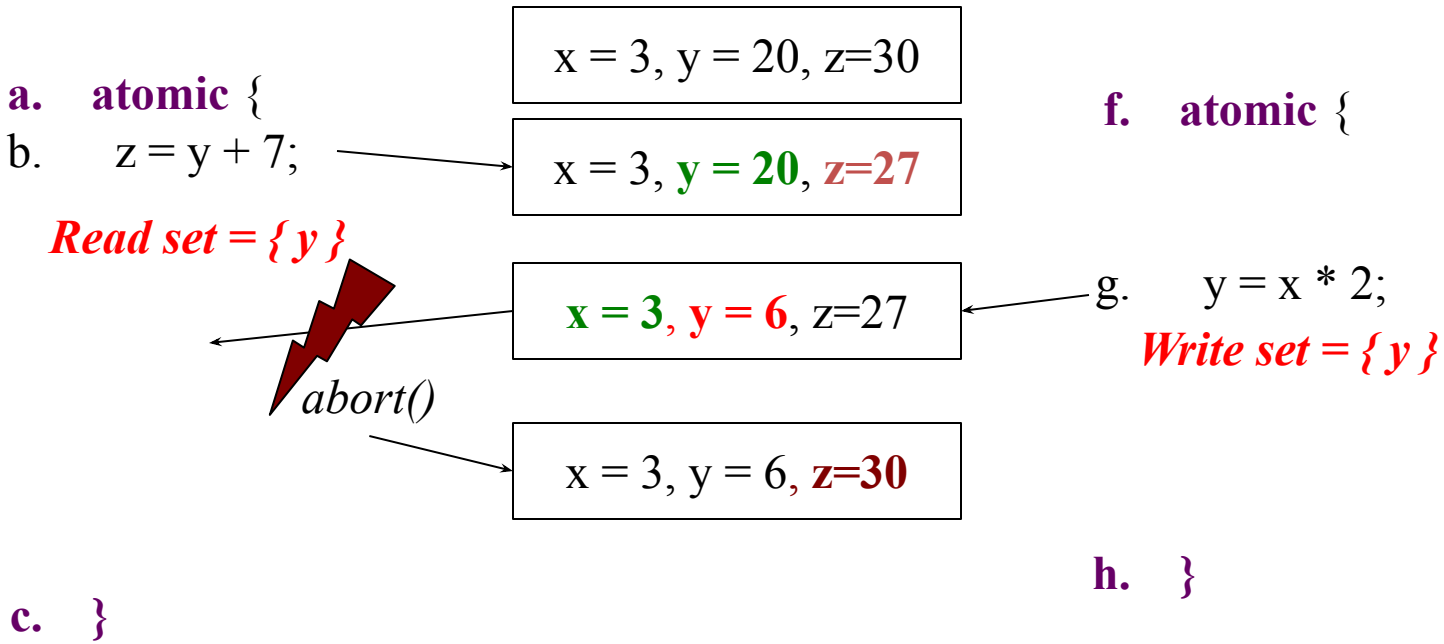
```
a.  atomic {
b.    tmp = x;
c.    tmp = tmp + 1;
d.    x = tmp;
e.  }
```

```
f.  atomic {
g.    tmp = x;
h.    tmp = tmp * 2;
i.    x = tmp;
j.  }
```



Immediate TM

Efficient if many commits



Note: the undo log is not represented

Conflict detection

Two possible solutions:

Eager: abort as soon as the runtime detects a conflict

- Code instrumentation for each read and each write

Lazy: check the conflict only at the end of the transaction

- Avoid instrumenting all the reads or all the writes
- Possible inconsistency if a transaction continues to run with invalid values (typically in case of double read conflicts)

Implementation techniques

- *Hardware transactional memory (HTM)*

  Use the processor cache to build a deferred-update TM

  Often use a lazy detection mechanism (explicit instruction to check the conflicts)

  + very efficient

  - size limited to the cache => inadequate for large transactions

- *Software transactional memory (STM)*

  Code instrumentation injected by a compiler

  - slower + can handle any size

- *Hybrid Transactional memory (HyTM)*

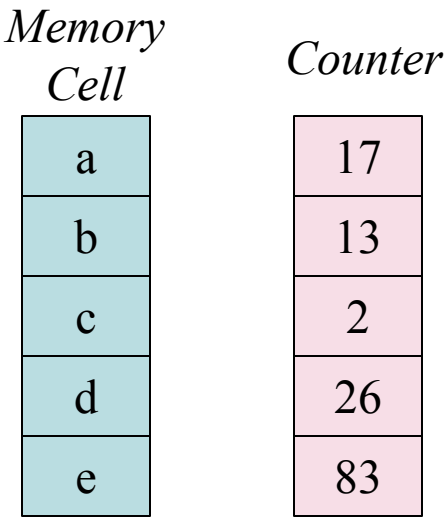
  In hardware if possible and switches to software otherwise

Naive algorithm

Deferred-update and pure lazy STM with a lock during commit

Principle :

- Associate a counter to each memory cell
- Read: record the counter in a local memory
- Write: write in a local memory
- At the end of the transaction
  - Ensure that the counters are not modified
  - In case of commit (counter not modified)
    - increments the counter in main memory
    - propagates the values in main memory
  - In case of abort (counter modified)
    - Simply ignores the local memory



Naive algorithm

Memory

Name	*Value*	*Counter*
a	10	17
b	21	13
c	7	2
d	83	26
e	8	83

Local log

a
b
c
d
e

```
a. atomic {
b.   a = a + b;
c.   c = a - e;
d.   b = c;
e. }
```

Naive algorithm

Memory

Name	Value	Counter
a	10	17
b	21	13
c	7	2
d	83	26
e	8	83

Local log

a	31	17
b		13
c		
d		
e		

```
a.  atomic {
b.   a = a + b;
c.   c = a - e;
d.   b = c;
e.  }
```

Naive algorithm

Memory

Name	Value	Counter
a	10	17
b	21	13
c	7	2
d	83	26
e	8	83

Local log

a	31	17
b		13
c	23	
d		
e		83

```
a.  atomic {
b.   a = a + b;
c.   c = a - e;
d.   b = c;
e.  }
```

Naive algorithm

Memory

Name	Value	Counter
a	10	17
b	21	13
c	7	2
d	83	26
e	8	83

Local log

Name	Value	Counter
a	31	17
b	23	13
c	23	
d		
e		83

```
a.  atomic {
b.    a = a + b;
c.    c = a - e;
d.    b = c;
e.  }
```

Naive algorithm

Memory

a	10	17
b	21	13
c	7	2
d	83	26
e	8	83

Local log

a	31	17
b	23	13
c	23	
d		
e		83

```
a.  atomic {
b.    a = a + b;
c.    c = a - e;
d.    b = c;
e.  }
```

Value

Name *Counter*

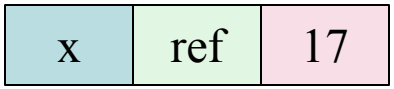
a	**31**	**18**
b	**23**	**14**
c	**23**	**3**
d	83	26
e	8	83

Memory state after commit

Issue: zombie transactions

```
a.  atomic {
b.    if(x != null)
c.      x.f();
d.  }
```

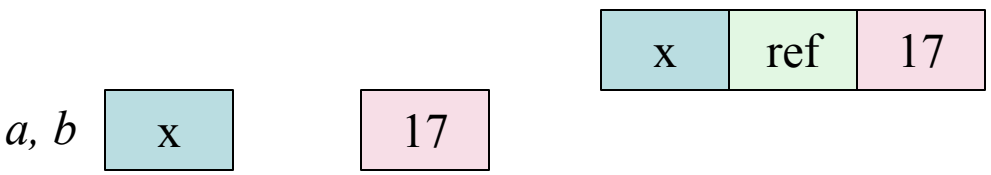
```
e.  atomic {
f.    x = null;
g.  }
```



Issue: zombie transactions

```
a.  atomic {
b.   if(x != null)
c.      x.f();
d.  }
```

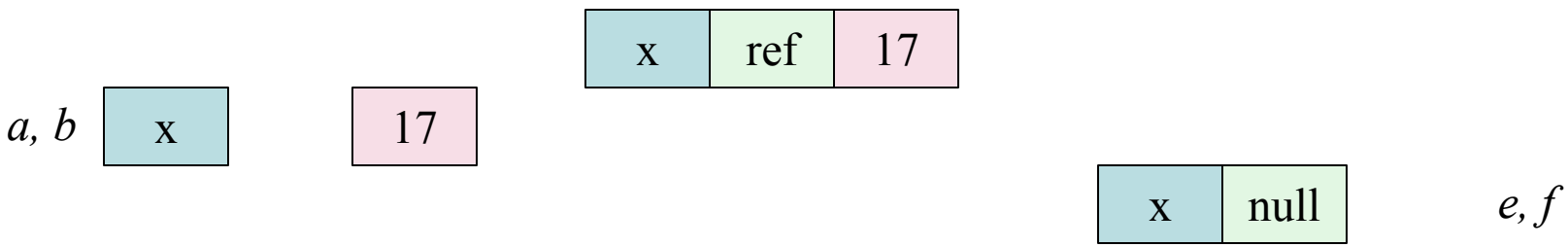
```
e.  atomic {
f.    x = null;
g.  }
```



Issue: zombie transactions

```
a.  atomic {
b.   if(x != null)
c.      x.f();
d.  }
```

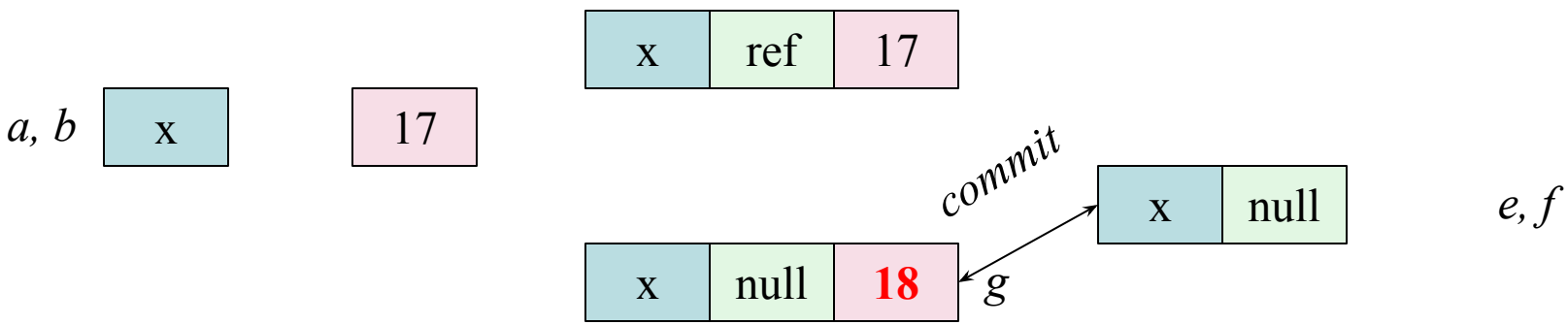
```
e.  atomic {
f.    x = null;
g.  }
```



Issue: zombie transactions

```
a.  atomic {
b.   if(x != null)
c.      x.f();
d.  }
```

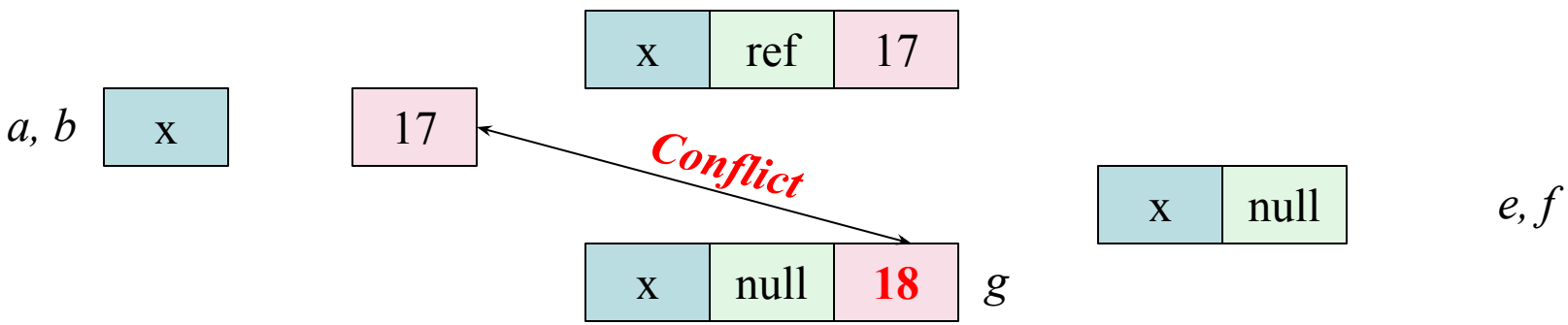
```
e.  atomic {
f.     x = null;
g.  }
```



Issue: zombie transactions

```
a.  atomic {
b.   if(x != null)
c.      x.f();
d.  }
```

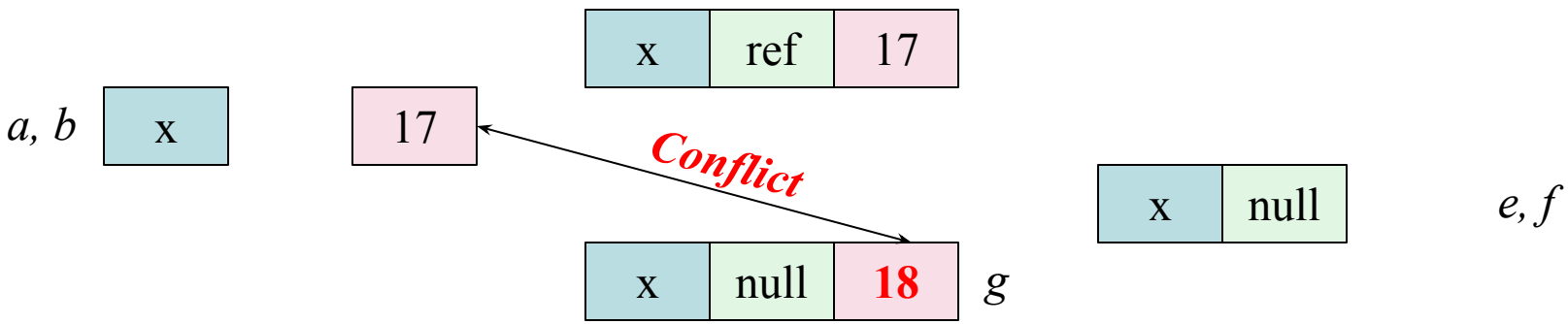
```
e.  atomic {
f.     x = null;
g.  }
```



Issue: zombie transactions

```
a.  atomic {
b.   if(x != null)
c.      x.f();
d.  }
```

```
e.  atomic {
f.    x = null;
g.  }
```



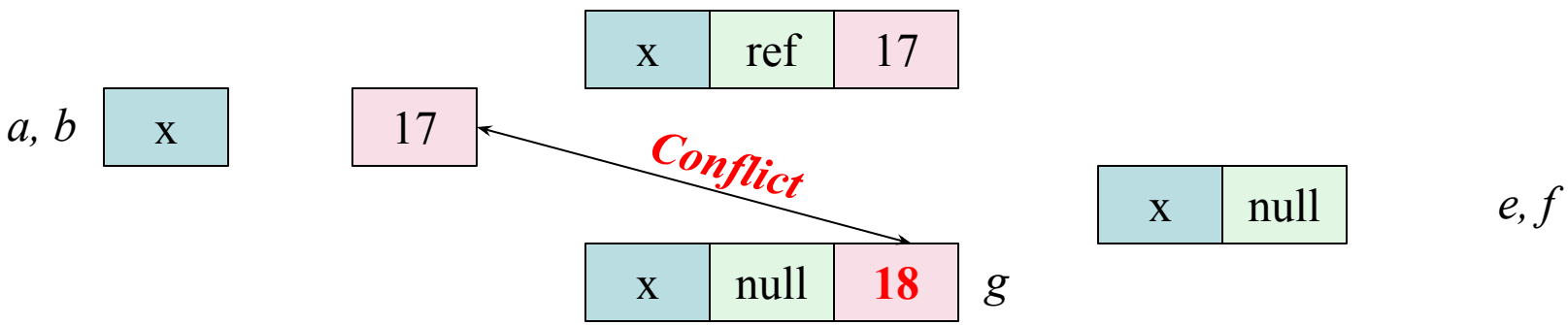
c: NullPointerException

Problem: the transaction does not read x again!

Issue: zombie transactions

```
a.  atomic {
b.   if(x != null)
c.      x.f();
d.  }
```

```
e.  atomic {
f.    x = null;
g.  }
```



c: NullPointerException

Naive solution: read again the x counter at line c and abort

Issue: another zombie transaction

Initially: x = 4, y = 5

```
a.  atomic {
b.    t1 = x;
c.    t2 = y;
d.    p = 1/(t1-t2)
e.  }
```

```
f.  atomic {
g.    x = 217;
h.    y = 4;
i.  }
```

x	4	17
y	5	83

Reading the counter at each read is not enough
Let suppose the invariant x != y

Issue: another zombie transaction

Initially: x = 4, y = 5

```
a.  atomic {
b.    t1 = x;
c.    t2 = y;
d.    p = 1/(t1-t2)
e.  }
```

```
f.  atomic {
g.    x = 217;
h.    y = 4;
i.  }
```

x	4	17
y	5	83

a,b : t1 = 4 x 17

Reading the counter at each read is not enough
Let suppose the invariant x != y

Issue: another zombie transaction

Initially: x = 4, y = 5

```
a.  atomic {
b.    t1 = x;
c.    t2 = y;
d.    p = 1/(t1-t2)
e.  }
```

a,b : t1 = 4

x

17

x	4	17
y	5	83

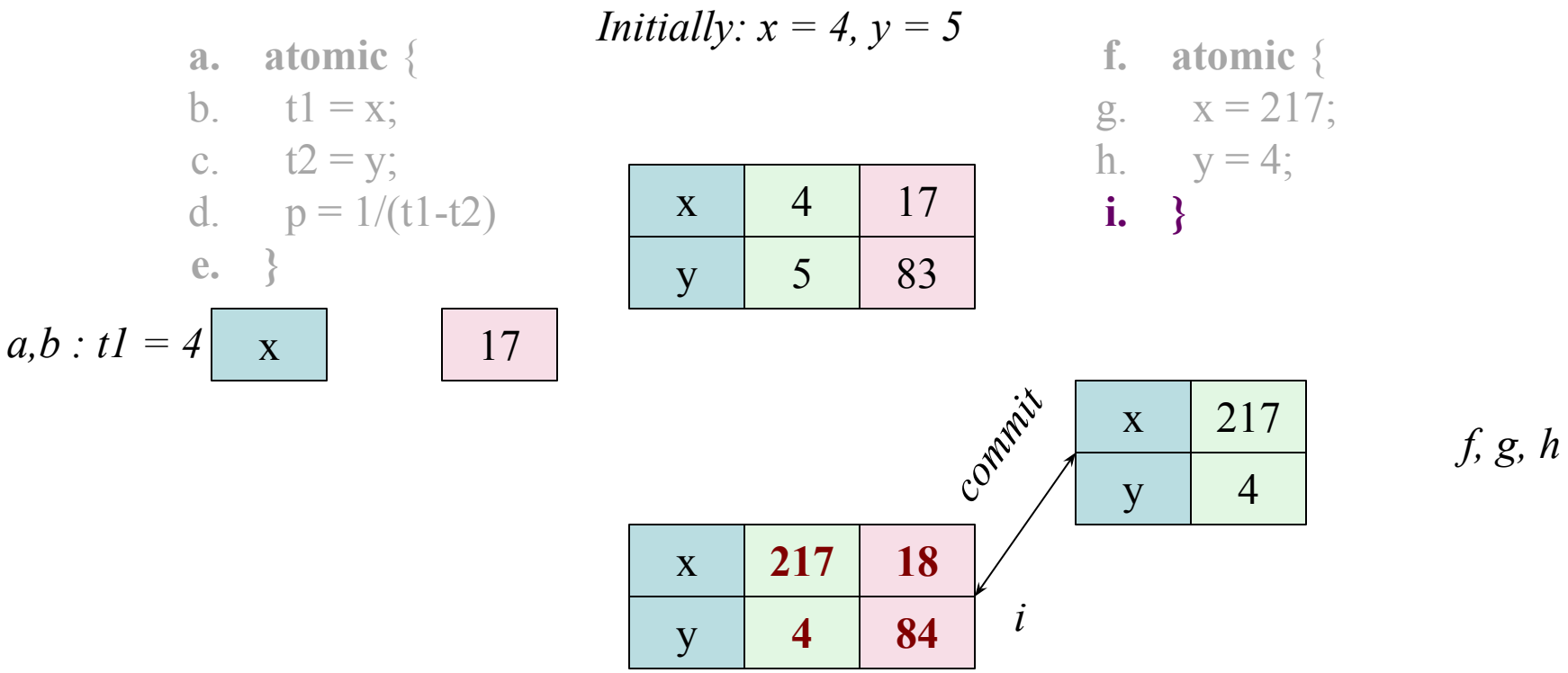
```
f.  atomic {
g.    x = 217;
h.    y = 4;
i.  }
```

x	217
y	4

f, g, h

Reading the counter at each read is not enough
Let suppose the invariant x != y

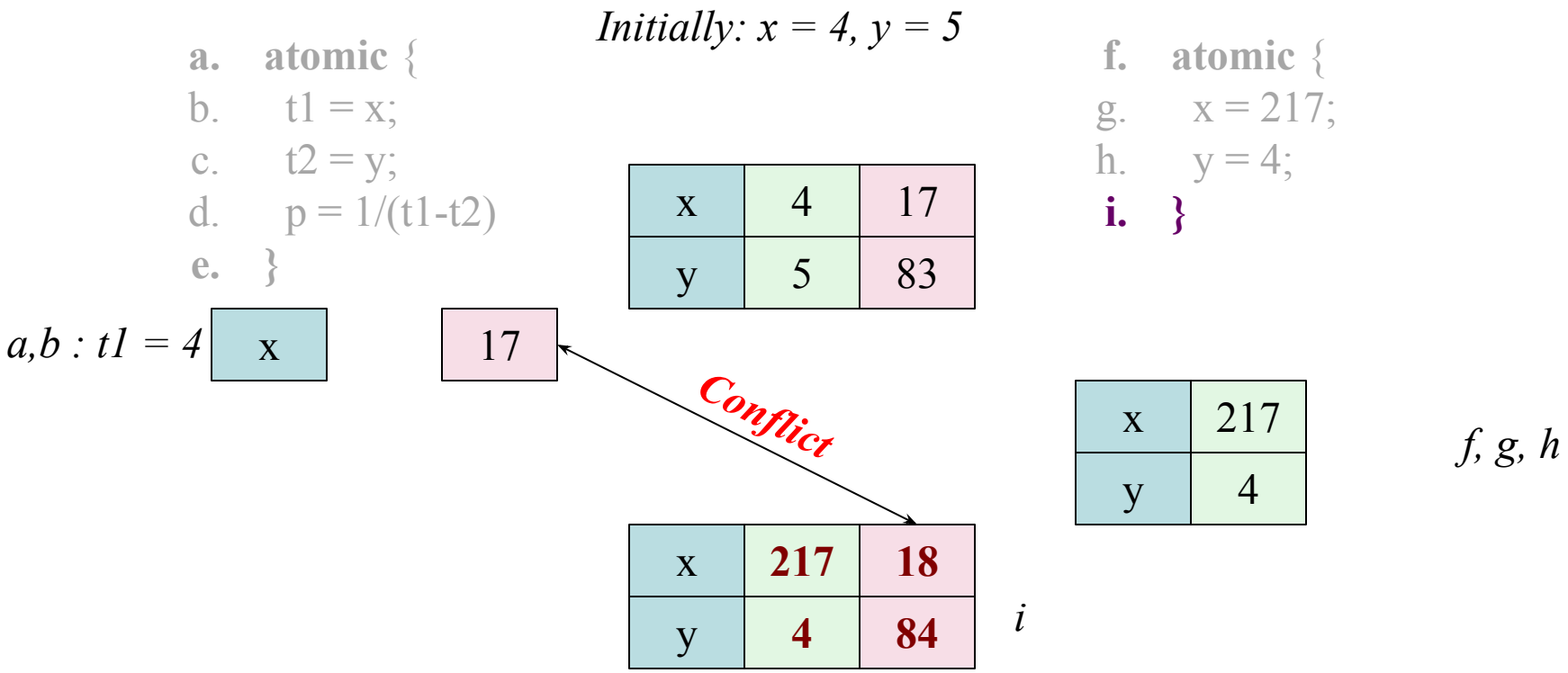
Issue: another zombie transaction



Reading the counter at each read is not enough

Let suppose the invariant x != y

Issue: another zombie transaction



Reading the counter at each read is not enough

Let suppose the invariant x != y

Issue: another zombie transaction

Initially: x = 4, y = 5

```
a.  atomic {
b.    t1 = x;
c.    t2 = y;
d.    p = 1/(t1-t2)
e.  }
```

```
f.  atomic {
g.    x = 217;
h.    y = 4;
i.  }
```

x	4	17
y	5	83

a,b : t1 = 4 | x | → 17

Conflict

x	**217**	**18**
y	**4**	**84**

i

x	217
y	4

f, g, h

c : t1 = 4, t2 = 4 | y | → 84

Problem: we cannot see that y was modified after the beginning of the transaction

Crash because t1 - t2 = 0

Reading the counter at each read is not enough

Let suppose the invariant x != y

Complete algorithm

Solution to avoid zombie transactions: a global clock

At each time, the counter of a variable has to be lower than the global clock

⇒ ensures that the variable was not modified after the beginning of the transaction

Full implementation

Start transaction

- Copy the global clock in a local clock

For each read

- Abort if the counter of the variable is greater or equal than the local clock
- Adds the variable to the read set otherwise

For each write

- Add the variable and its value in the write set

End transaction:

- If exists var in read set >=local clock, abort
- For each var in write set, update its value and its counter (to current global clock)
- Increment global clock

Full implementation

```
a.  atomic {
b.    t1 = x;
c.    t2 = y;
d.    p = 1/(t1-t2)
e.  }
```

90

```
f.  atomic {
g.    x = 217;
h.    y = 4;
i.  }
```

Full implementation

```
a.  atomic {
b.    t1 = x;
c.    t2 = y;
d.    p = 1/(t1-t2)
e.  }
```

90

```
f.  atomic {
g.    x = 217;
h.    y = 4;
i.  }
```

90

Full implementation

```
a.  atomic {
b.    t1 = x;
c.    t2 = y;
d.    p = 1/(t1-t2)
e.  }
```

90

Other transactions in //

100

x	4	17
y	5	83

```
f.  atomic {
g.    x = 217;
h.    y = 4;
i.  }
```

90

Full implementation

```
a.  atomic {
b.    t1 = x;
c.    t2 = y;
d.    p = 1/(t1-t2)
e.  }
```

```
f.  atomic {
g.    x = 217;
h.    y = 4;
i.  }
```

90

Other transactions in //

90

100

a

100

x	4	17
y	5	83

Full implementation

```
a.  atomic {
b.    t1 = x;
c.    t2 = y;
d.    p = 1/(t1-t2)
e.  }
```

90

Other transactions in //

```
f.  atomic {
g.    x = 217;
h.    y = 4;
i.  }
```

90

100

x	4	17
y	5	83

a

100

b : t1 = 4

x

Full implementation

```
a.  atomic {
b.    t1 = x;
c.    t2 = y;
d.    p = 1/(t1-t2)
e.  }
```

90

Other transactions in //

100

x	4	17
y	5	83

```
f.  atomic {
g.    x = 217;
h.    y = 4;
i.  }
```

90

a

100

b : t1 = 4

x

x	217
y	4

g, h

Full implementation

```
a.  atomic {
b.    t1 = x;
c.    t2 = y;
d.    p = 1/(t1-t2)
e.  }
```

```
f.  atomic {
g.    x = 217;
h.    y = 4;
i.  }
```

90

Other transactions in //

90

100

x	4	17
y	5	83

a

100

b : t1 = 4

x

101

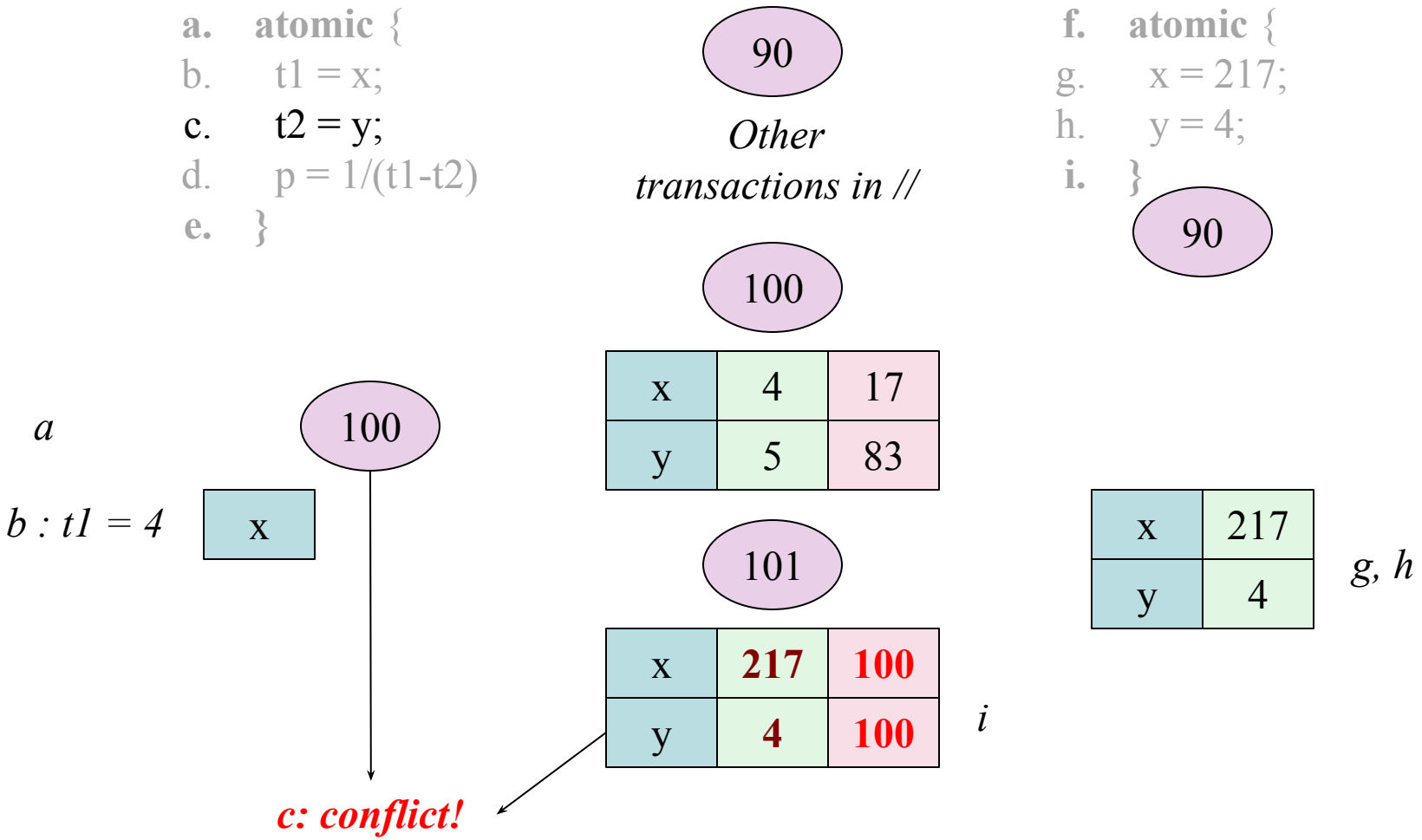
x	**217**	**100**
y	**4**	**100**

i

x	217
y	4

g, h

Full implementation



Implementation

- Memory is an array of pointers to (value, counter)
- Atomically update a pointer to a new (value, counter), but never modify a the value or the counter in an existing (value, counter)
- Don’t try to free a (value, counter): we need a garbage collector because we can not easily know if a (value, counter) is not still used by another thread

Implementation

```
class Value {
  int value;
  int counter;
}
```

```
class Memory {
  static Value values[];
  static int   clock;
}
```

```
class TX {
  HashSet<int>      readSet;
  HashMap<int, int> writeSet;
  int               clock;
}
```

Implementation

```
class Value {
  int value;
  int counter;
}
```

```
class Memory {
  static Value values[];
  static int   clock;
}
```

```
class TX {
  HashSet<int>       readSet;
  HashMap<int, int> writeSet;
  int                clock;
}
```

```
void TX.begin() {
  clock    = Memory.clock;
  readSet  = new HashSet();
  writeSet = new HashMap();
}
```

Start a transaction: copy the global clock

Implementation

```
class Value {
  int value;
  int counter;
}
```

```
class Memory {
  static Value values[];
  static int   clock;
}
```

```
class TX {
  HashSet<int>      readSet;
  HashMap<int, int> writeSet;
  int               clock;
}
```

```
void TX.write(int idx, int value) {
  writeSet.put(idx, value);
}
```

Deferred update: write in a local variable

Implementation

```
class Value {
  int value;
  int counter;
}
```

```
class Memory {
  static Value values[];
  static int   clock;
}
```

```
class TX {
  HashSet<int>       readSet;
  HashMap<int, int>  writeSet;
  int                clock;
}
```

```
int TX.read(int idx) {
  if(writeSet.contains(idx))
    return writeSet.get(idx);

  Value value = Memory.values[idx];

  if(value.counter >= clock)
    abort();

  readSet.add(idx);

  return value.value;
}
```

If a local write exists, use it

Abort of value was modified by another transaction

Implementation

```
class Value {
  int value;
  int counter;
}
```

```
class Memory {
  static Value values[];
  static int   clock;
}
```

```
class TX {
  HashSet<int>       readSet;
  HashMap<int, int>  writeSet;
  int                clock;
}
```

```
void TX.commit() {
  synchronized(Memory.values) { // Take a lock during a commit
    for(int idx : readSet)
      if(Memory.values[idx].counter >= clock) abort();

    // ok, commit!
    for(Map<int, Value> entry : writeSet.entrySet()) {
      Value v = new Value(entry.getValue(), Memory.clock);
      Memory.values[entry.getKey()] = v;
    }

    Memory.clock++;
  }
}
```

Reader/writer conflict?

Record the written values and updates the counters

For each transaction that begin after this line, the writes are consistent (counter < clock)

Implementation

Problem:

Two transactions abort each other

Restart ⇒ they will probably abort each other

Solution:

- Introduce a random dalay that increases exponentially (backoff)

```
        int backoff(int n) {
          Thread.sleep(1+(int)(n*Math.random()));
          return n < 512 ? n<<1 : n;
        }
void doTransaction() {
  n = 16;
  try {              tx.begin(); …; tx.commit();      }
  catch(TXAbort e) { n = backoff(n); doTransaction();  }
}
```

Transaction and Input/Output

```
atomic {
  if(x > 42)
    launchMissile();
}
```

Aborting an input/output is not always possible

Solution:

- Ensures that the transaction can still commit before the I/O
- Marks the transaction as unabortable

⇒ Complexify the code

To take away

Transactional memory simplifies the development of concurrent applications

- No deadlock, no starvation
- Composability (inner transactions, retry, orElse)

Implementation is difficult: performance are far from perfect

- STM: less efficient than fine grain locking schemes [Rossback07]
- HTM: only for corner case where the transaction fits in the L1 cache
- HyTM: switching from HTM to STM is costly

Performance evaluation:

- 100 threads increment 10’000 times a counter on a 2-core
- 3,0s in STM without backoff, **0, 48s in STM with backoff, 0,19s with a lock**)