8th Central and Eastern European Software Engineering Conference in Russia - CEE-SECR 2012

November 1 - 2, Moscow

2012
CEE-SEC(R)
Software Engineering
Conference in Russia

Dynamic data race detection in concurrent Java programs

DEVEXPERTS

Vitaly Trifanov, Dmitry Tsitelov

Devexperts LLC

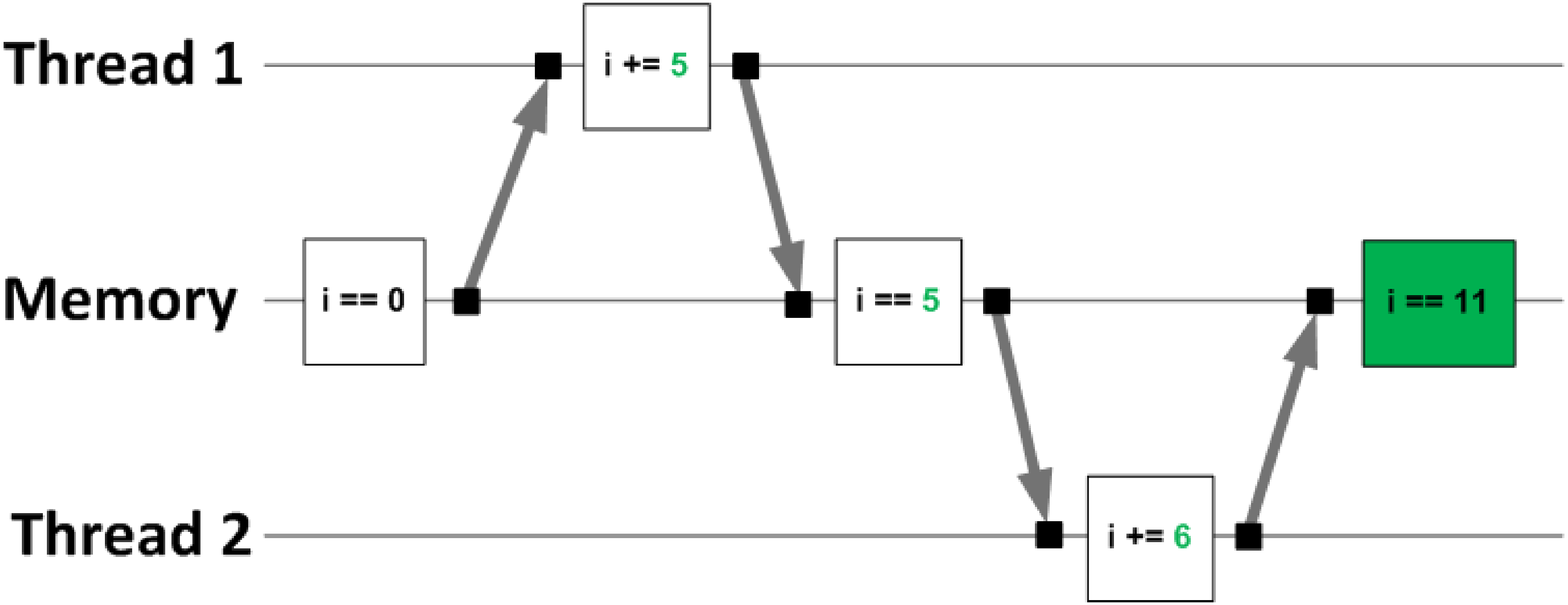
Agenda

- What are data races and why they are dangerous
- Automatic races detection
  - approaches, pros & cons
- Happens-before race detection algorithm
  - Vector clocks
- Our dynamic race detector
  - implementation
  - solved problems

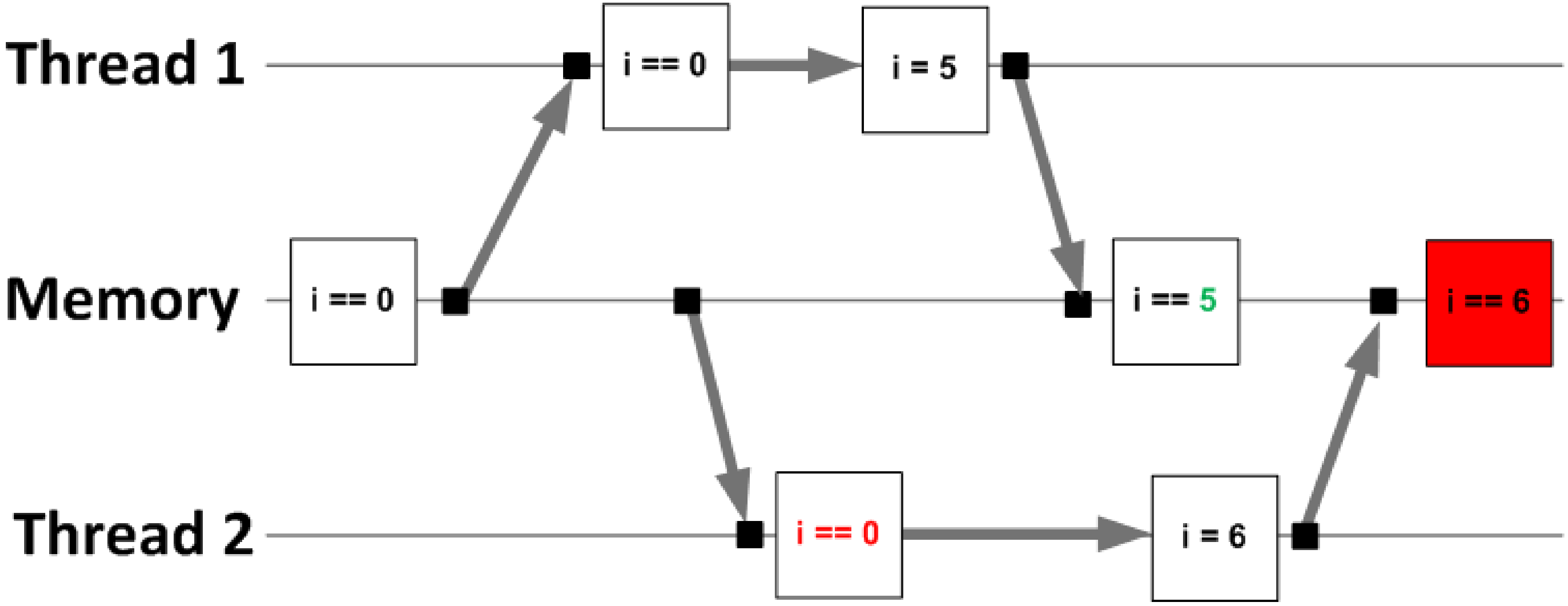
Data Race Example

```
public class Account {
  private int amount = 0;
  public void deposit(int x) {amount += x;}
  public int getAmount() {return amount;}
}

public class TestRace {
   public static void main (String[] args) {
     final Account a = new Account();
     Thread t1 = depositAccountInNewThread(a, 5);
     Thread t2 = depositAccountInNewThread(a, 6);
     t1.join();
     t2.join();
     System.out.println(account.getAmount()); //may print 5, 6, 11.
  }
}
```

Expected Execution



Racy Execution



Data Races

- Data race occurs when many threads access the same shared data concurrently; at least one writes

- Usually it’s a bug

Data Races Are Dangerous

- Hard to detect if occurred
  - no immediate effects
  - program continues to work
  - damage global data structures

- Hard to find manually
  - Not reproducible - depends on threads timing
  - Dev & QA platforms are not so multicore

Automatic Race Detection

- 20+ years of research
- Static
  - analyze program code offline
  - data races prevention (extend type system, annotations)
- Dynamic: analyze real program executions
  - On-the-fly
  - Post-mortem

Dynamic Detectors vs Static

Static Approach

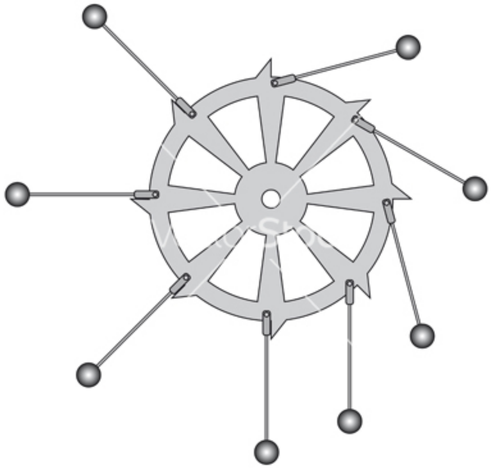
- Pros
  - Doesn’t require program execution
  - Analyzes all code
  - Doesn’t depend on program input, environment, etc.

- Cons
  - Unsolvable in common case
  - Has to reduce depth of analysis

- A lot of existing tools for Java
  - FindBugs, jChord, etc

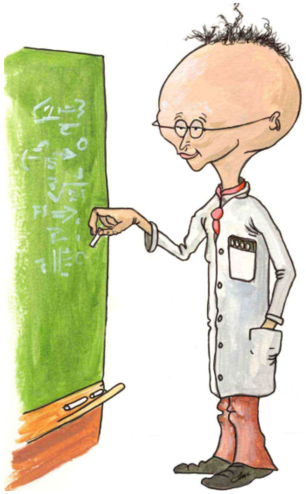
Dynamic Approach

- Pros
  - Complete information about program flow
  - Lower level of false alarms
- Cons
  - Very large overhead
- No existing stable dynamic detectors for Java



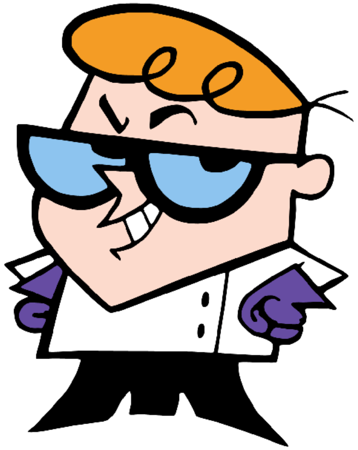
Static vs Dynamic: What To Do?

- Use both approaches ☺



- Static (FindBugs/Sonar, jChord, …)
  - Eliminate provable synchronization inconsistencies on the early stage

- Dynamic
  - Try existing tools, but they are unstable
    - IBM MSDK, Thread Sanitizer for Java
  - That’s why we’ve developed our own!



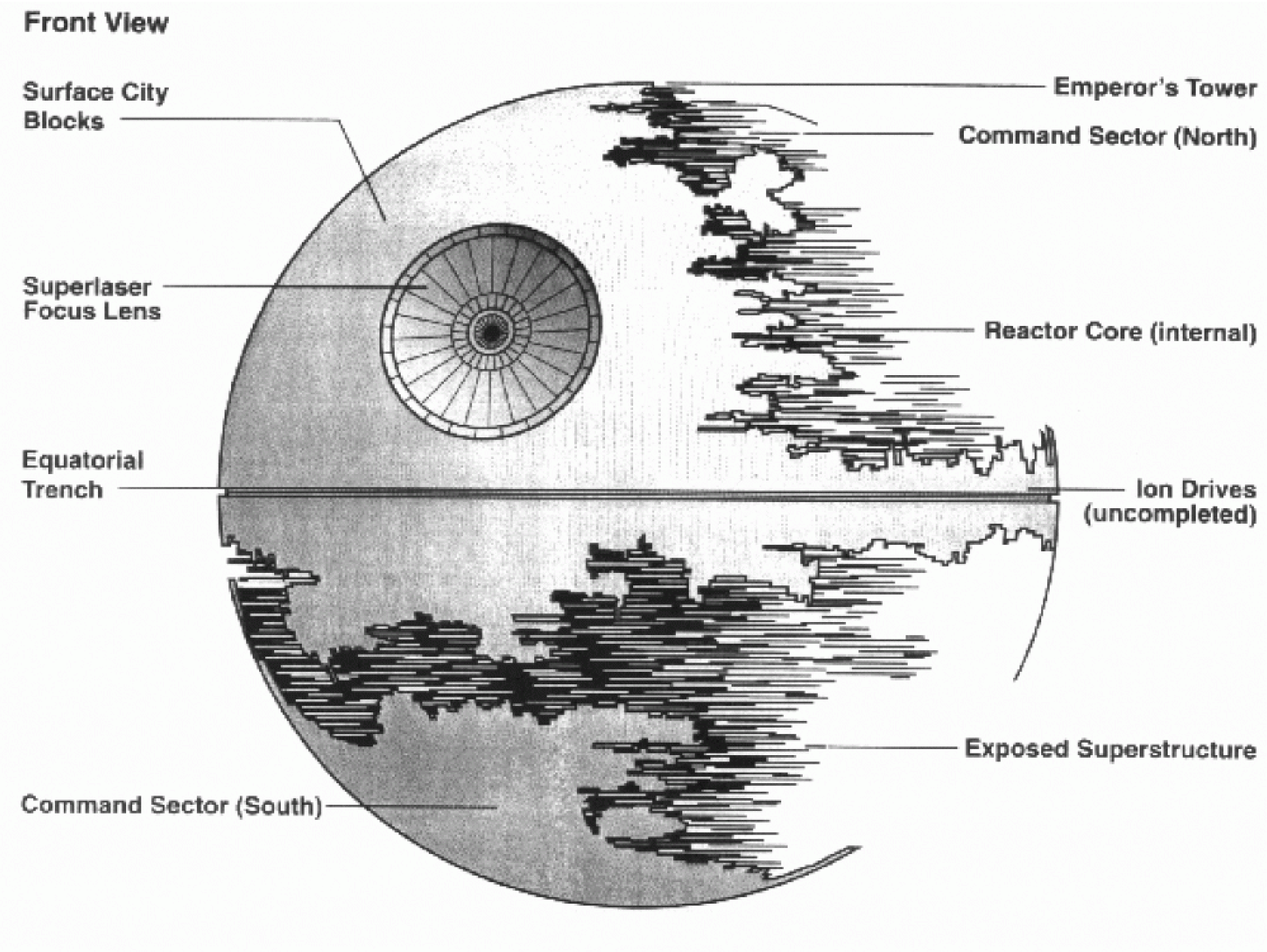
Data Race Detector Concept

- Application uses libraries and frameworks via API
  - At least JRE

- API is well documented
  - “Class XXX is thread-safe”
  - “Class YYY is not thread-safe”
  - “XXX.get() is synchronized with preceding call of XXX.set()”

- Describe behavior of API and exclude library from analysis

DRD: How It’s Organized

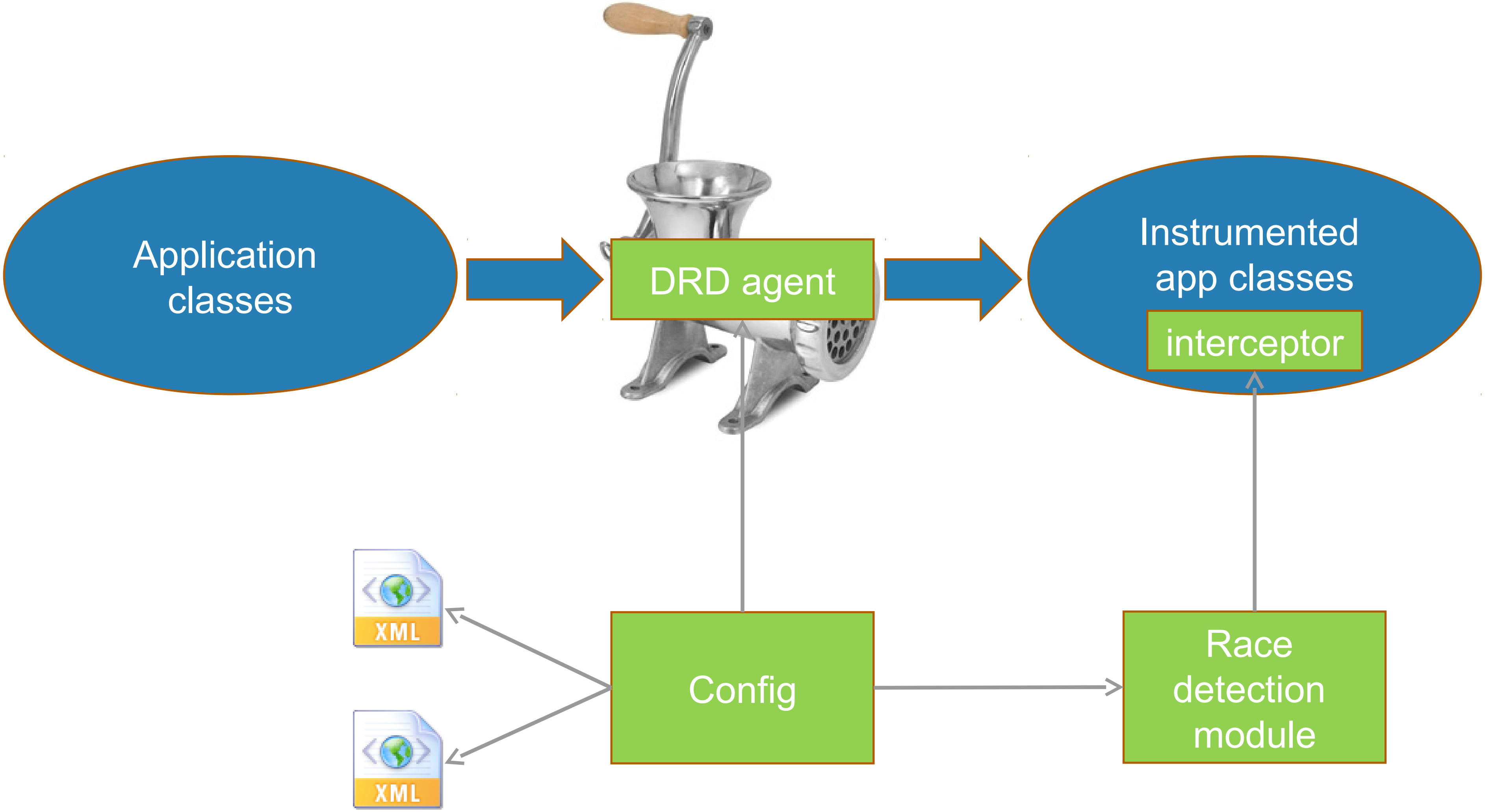


What Operations to Intercept?

- Synchronization operations
  - thread start/join/interrupt
  - synchronized
  - volatile read/write
  - java.util.concurrent

- Accesses to shared data
  - fields
  - objects

How It Works



Application classes

DRD agent

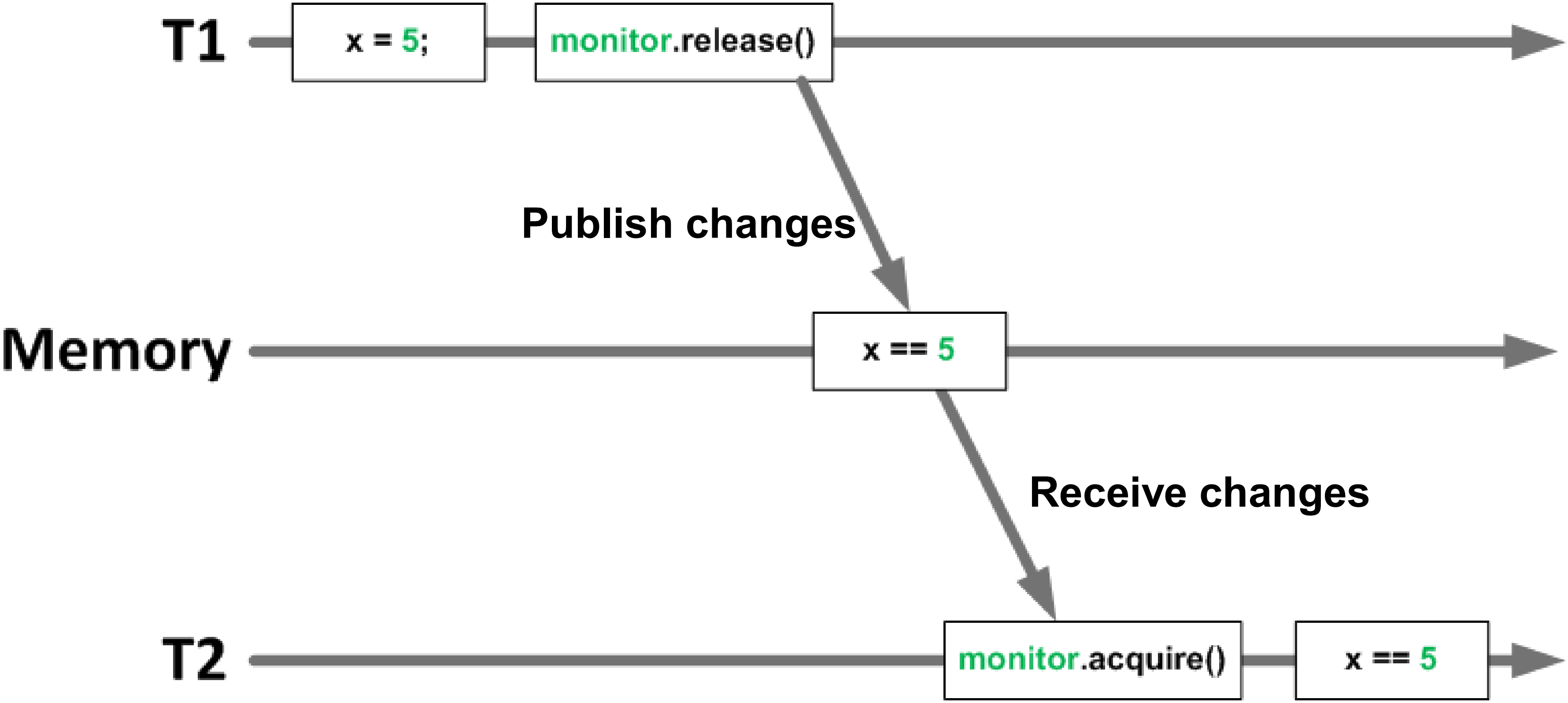
Instrumented app classes

interceptor

Config

Race detection module

JLS: Publishing Data



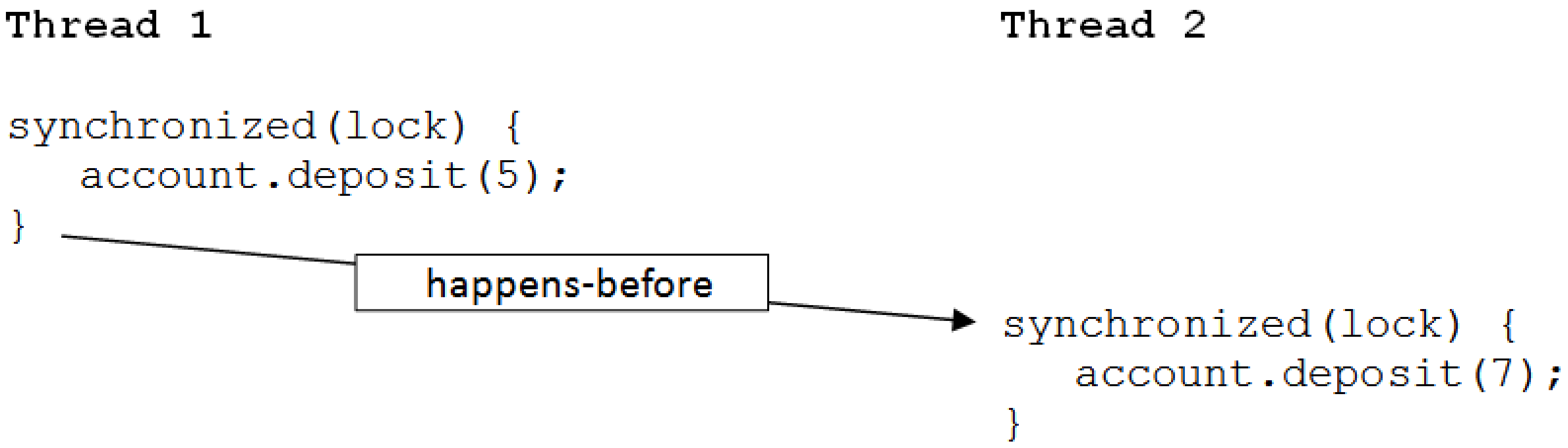
JLS: Synchronized-With Relation

- “Synchronized-with” relation
  - unlock monitor M ↦ all subsequent locks on M
  - volatile write ↦ all subsequent volatile reads
  - …
- Notation: **send** ↦ **receive**

JLS: Happens-Before & Data Races

- X happens-before Y, when
  - X, Y - in same thread, X before Y in program order
  - X is synchronized-with Y
  - Transitivity: exists Z: *hb(X, Z)* && *hb*(*Z, Y*)
- Data race: 2 conflicting accesses, not ordered by happens-before relation

Happens-Before Example



Thread 1

```
synchronized(lock) {
   account.deposit(5);
}
```

Thread 2

```
synchronized(lock) {
   account.deposit(7);
}
```

- No data race

Vector Clock

Time

A

A:0

B

B:0

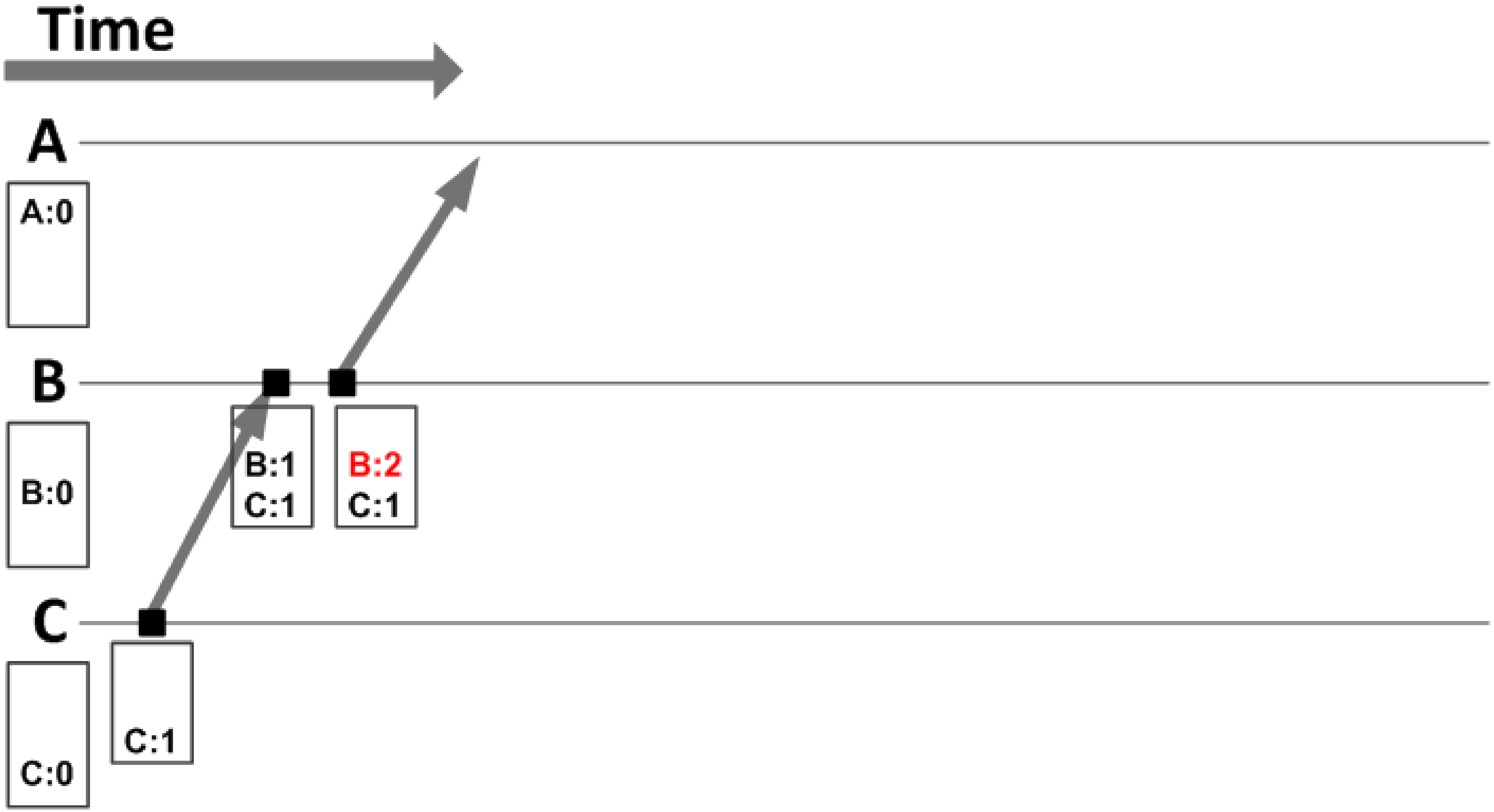
C

C:0

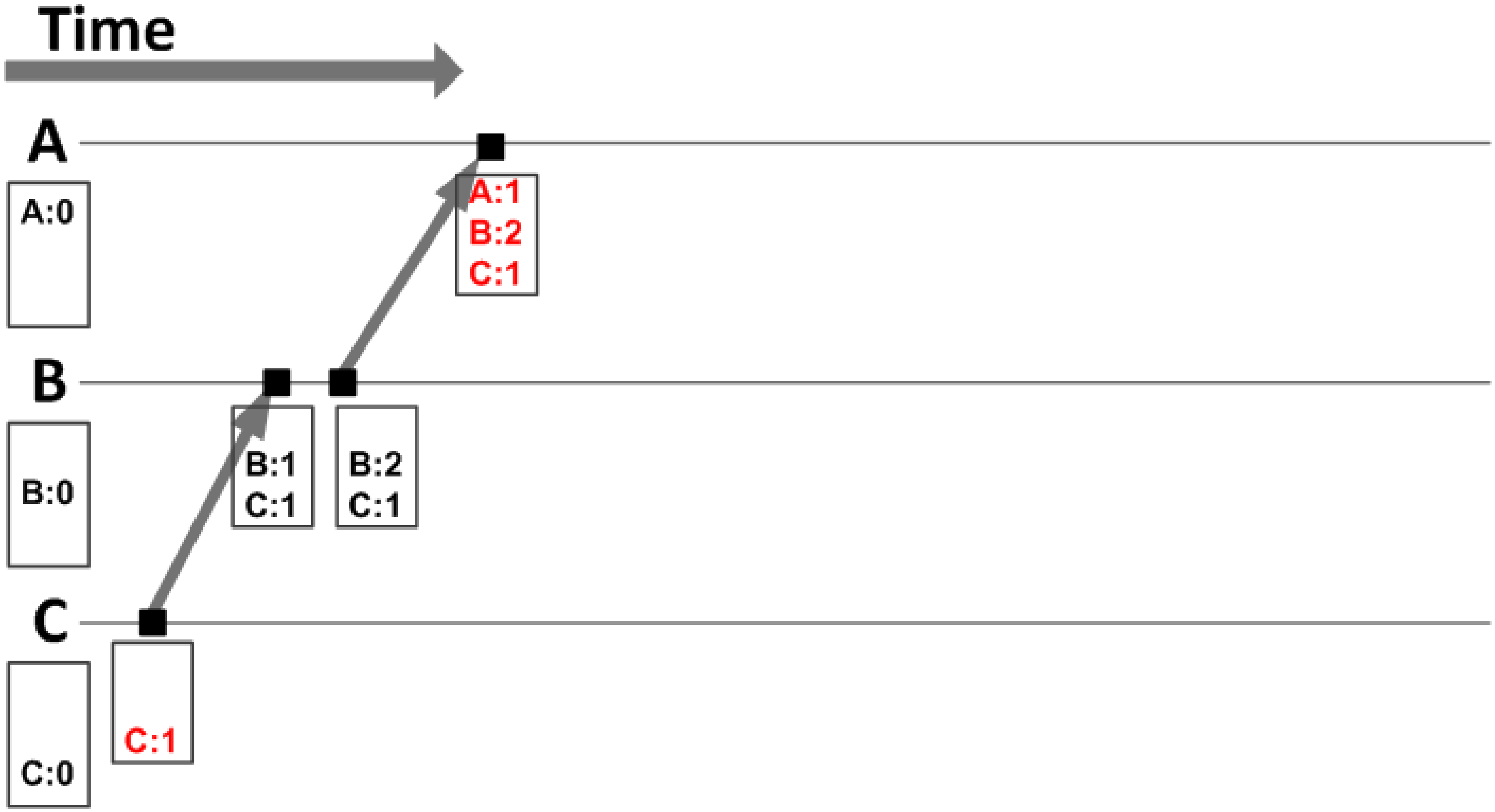
Vector Clock

Vector Clock

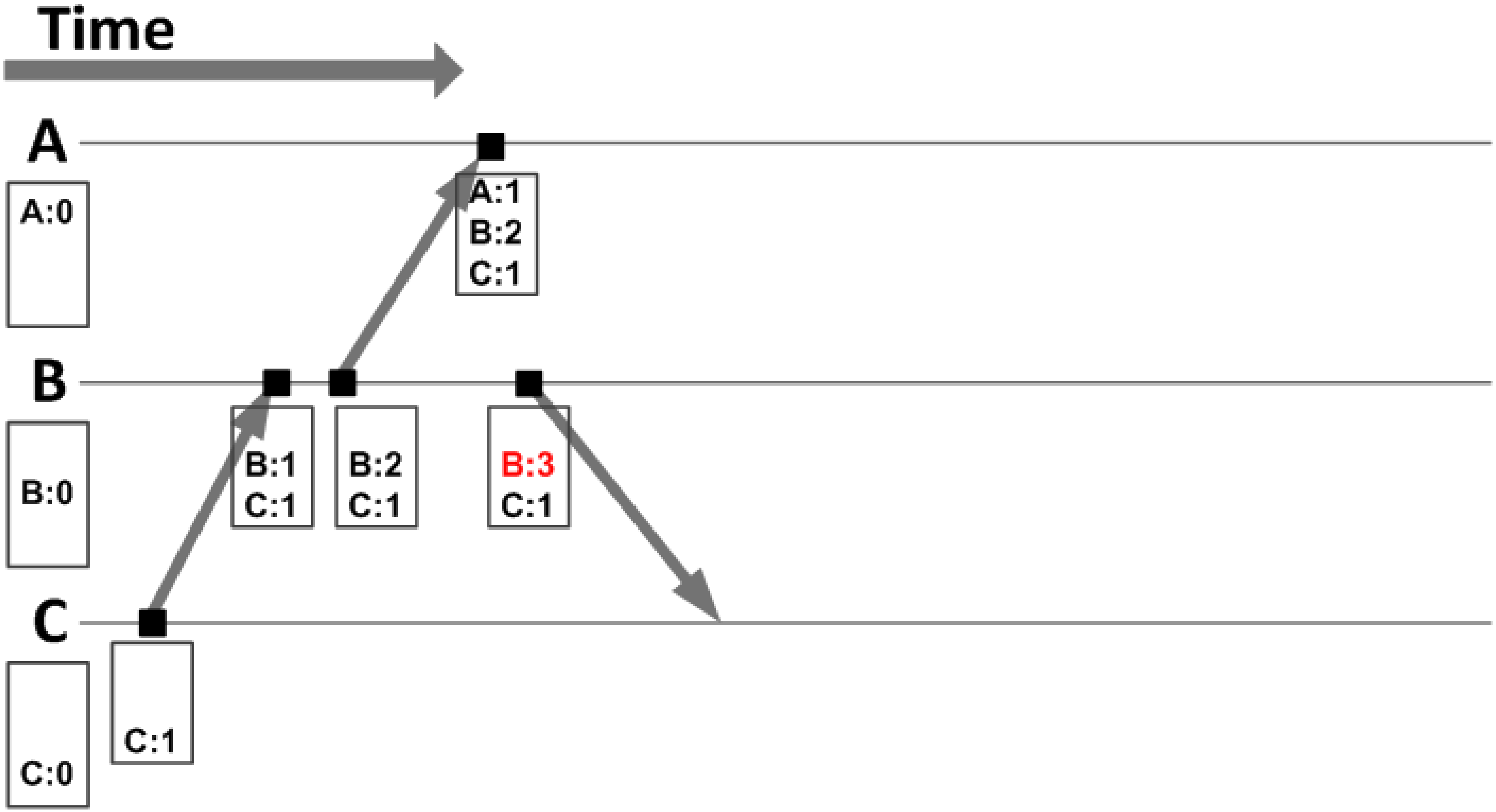
Vector Clock



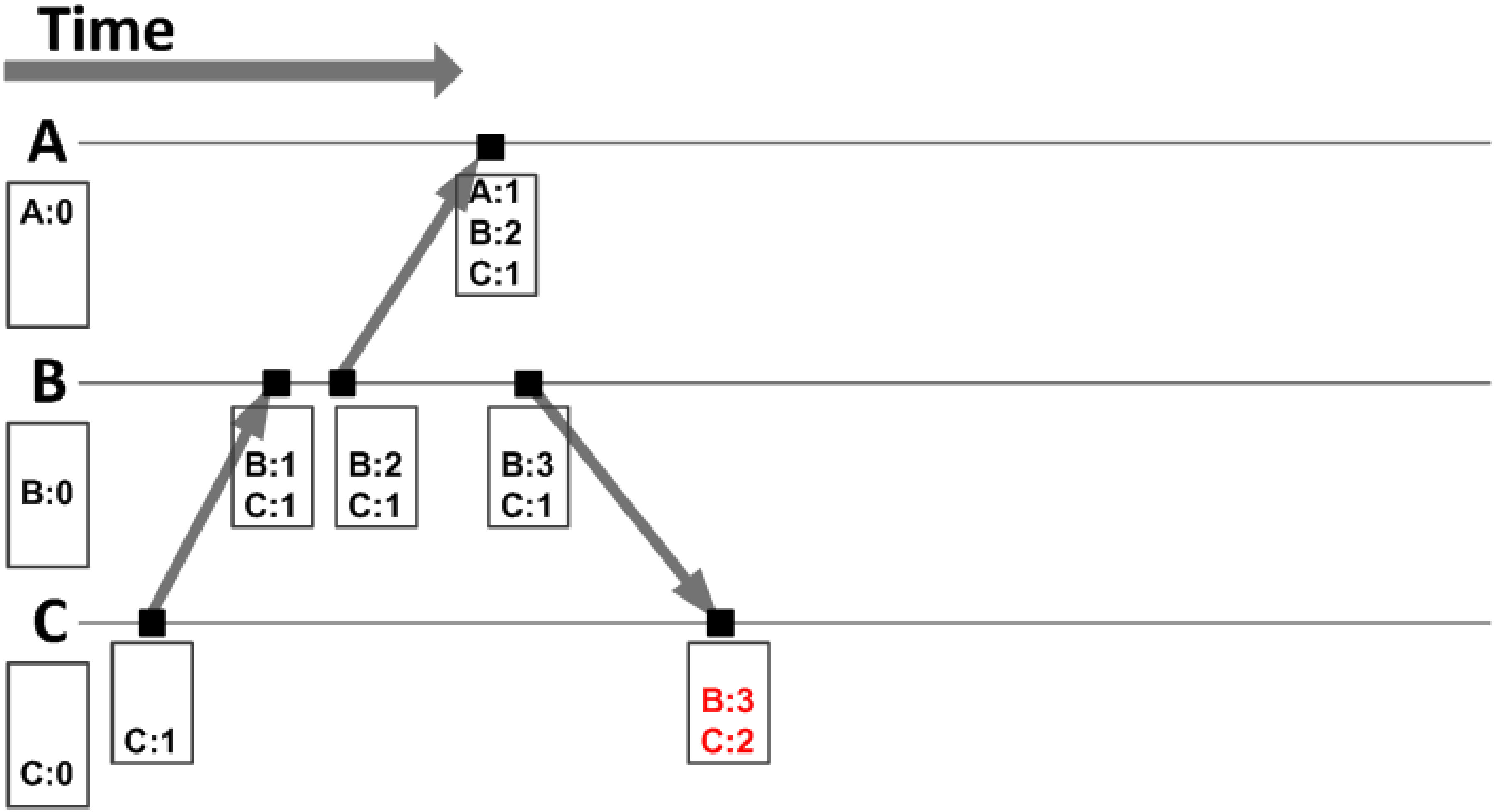
Vector Clock



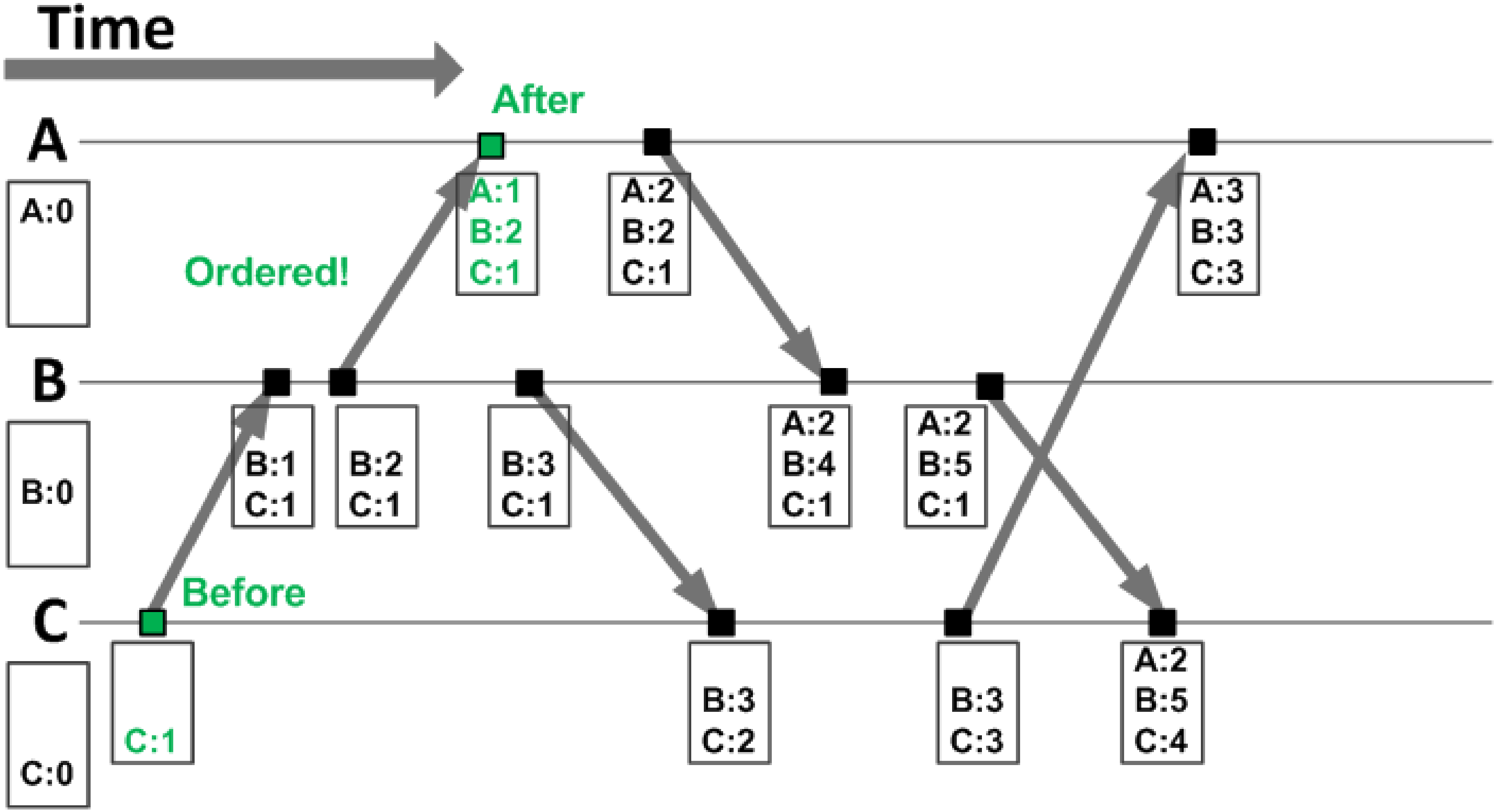
Vector Clock



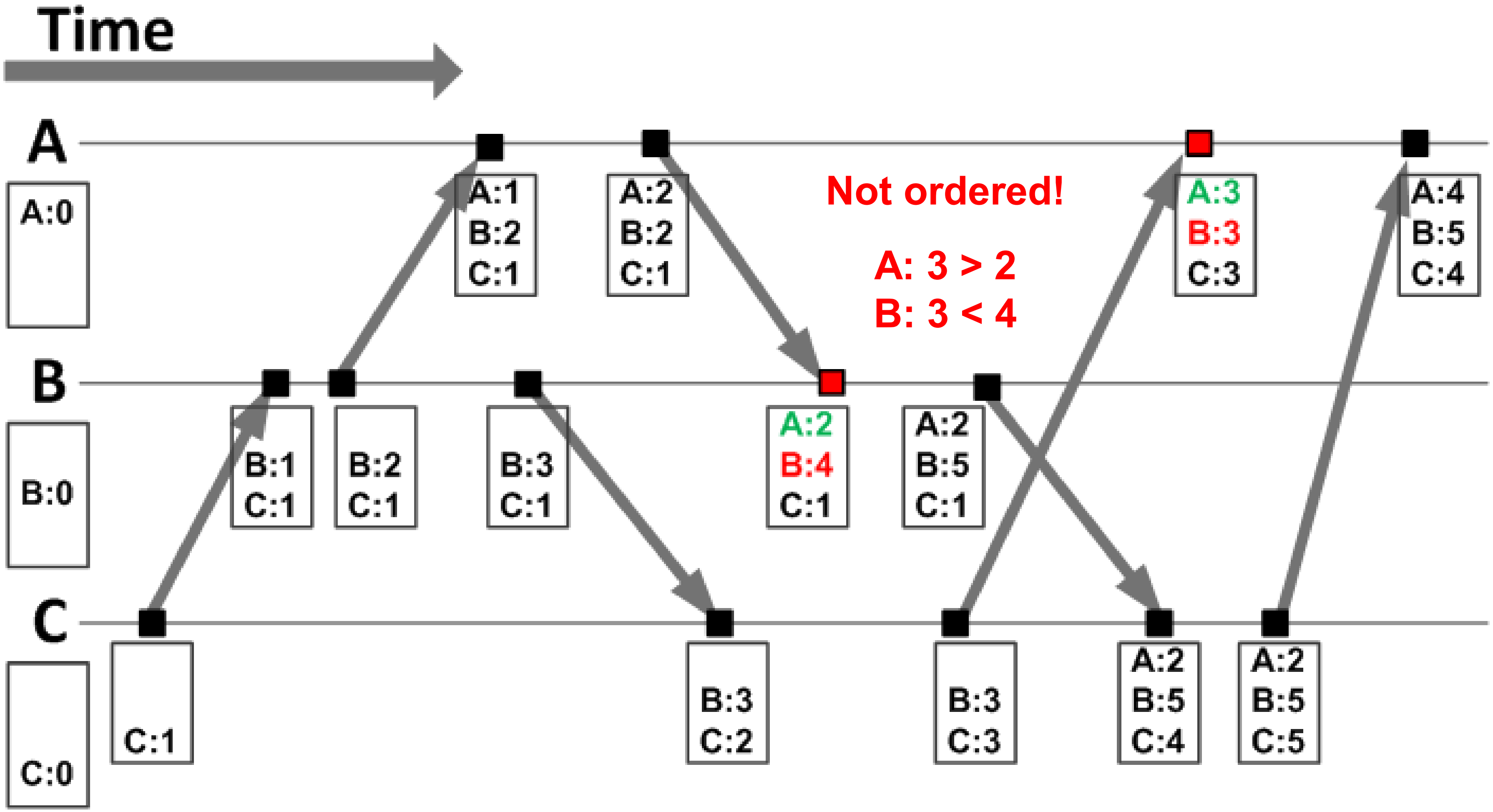
Vector Clock



Vector Clock



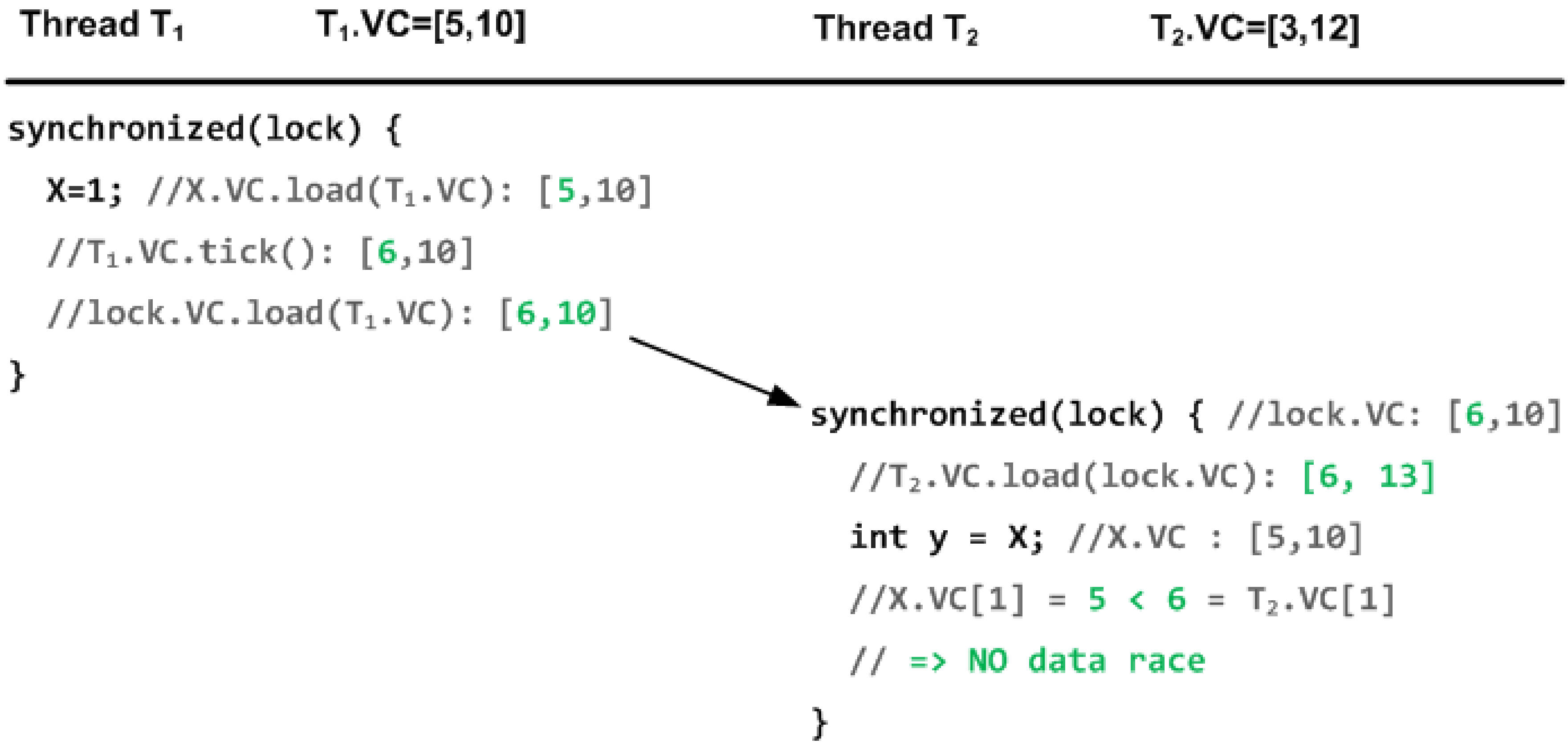
Vector Clock



Not ordered!

A: 3 > 2
B: 3 < 4

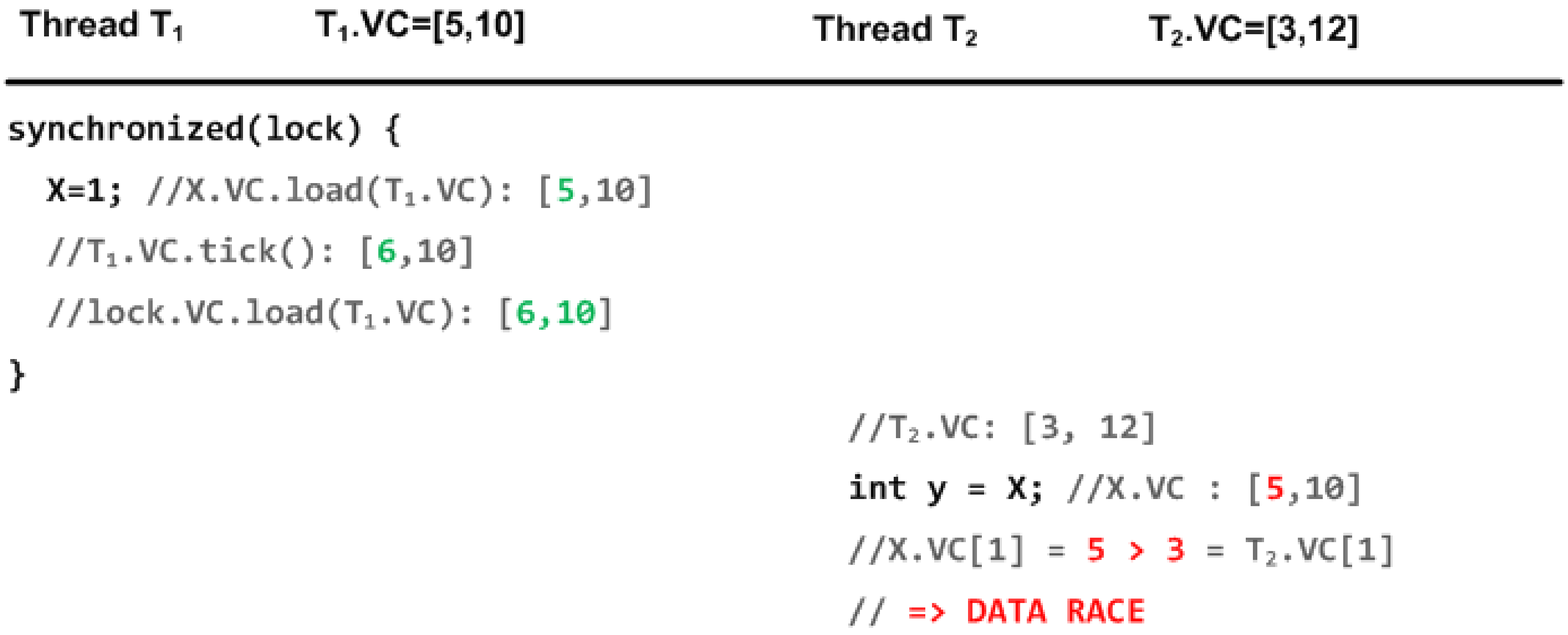
How It Works. No Data Race Example



Thread T_1 **T_1.VC=[5,10]** **Thread T_2** **T_2.VC=[3,12]**

```
synchronized(lock) {
  X=1; //X.VC.load(T1.VC): [5,10]
  //T1.VC.tick(): [6,10]
  //lock.VC.load(T1.VC): [6,10]
}
                                            synchronized(lock) { //lock.VC: [6,10]
                                              //T2.VC.load(lock.VC): [6, 13]
                                              int y = X; //X.VC : [5,10]
                                              //X.VC[1] = 5 < 6 = T2.VC[1]
                                              // => NO data race
                                            }
```

How It Works. Data Race Example



Thread T_1	T_1.VC=[5,10]	Thread T_2	T_2.VC=[3,12]

```
synchronized(lock) {
  X=1; //X.VC.load(T1.VC): [5,10]
  //T1.VC.tick(): [6,10]
  //lock.VC.load(T1.VC): [6,10]
}
```

```
//T2.VC: [3, 12]
int y = X; //X.VC : [5,10]
//X.VC[1] = 5 > 3 = T2.VC[1]
// => DATA RACE
```

Code Instrumentation

- Check everything => huge overhead

- Race detection scope
  - Accesses to our fields
  - Foreign calls (treat them as read or write)

- Sync scope
  - Detect sync events in our code
  - Describe contracts of excluded classes
  - Treat these contracts as synchronization events

Race Detection

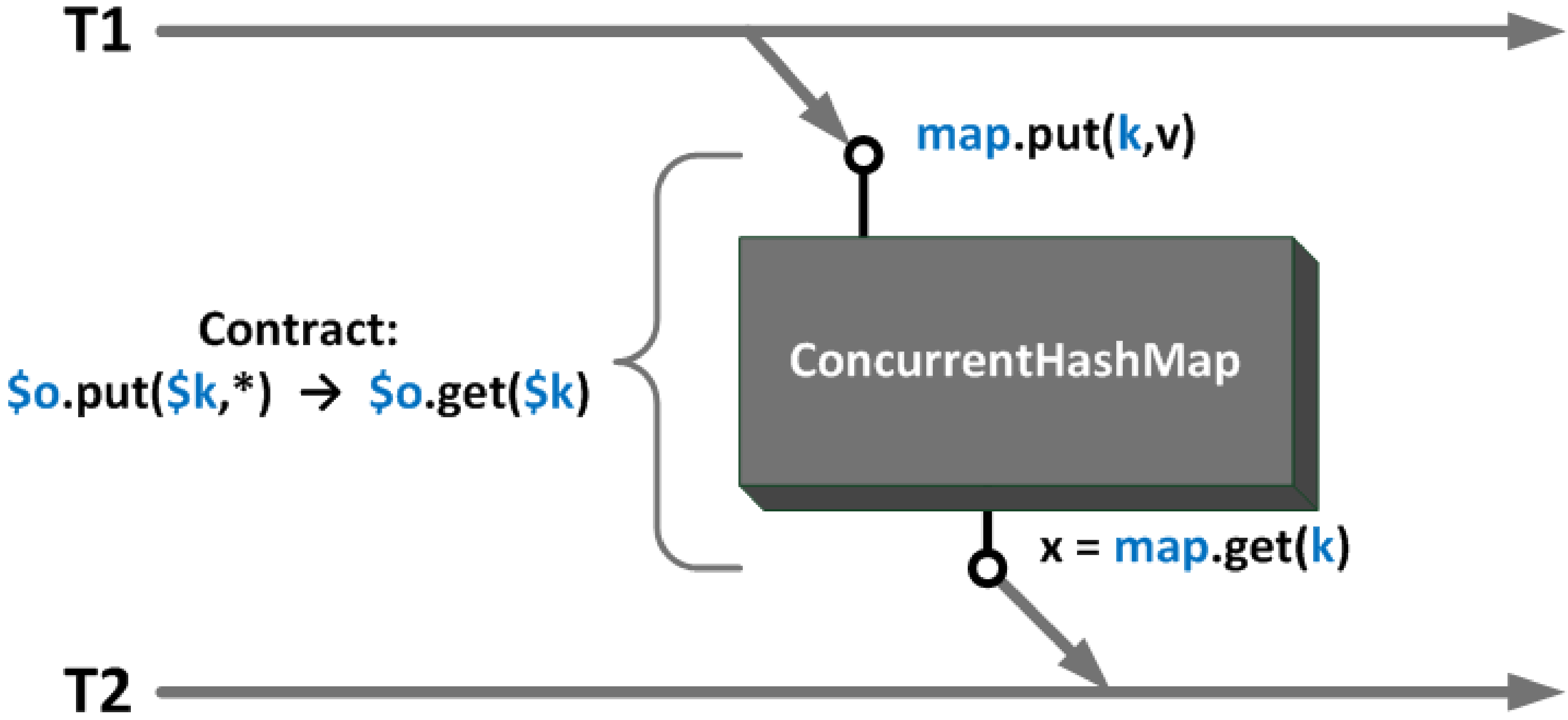
```
private class Storage {
   private Map<Integer, Item> items = new HashMap<Integer, Item> ();

   public void store(Item item) {
       items.put(item.getId(), item);
   }

   public void saveToDisk() {
       for (Item item : items.values()) {
              //serialize and save
              saveItem(item);
              //...
       }
   }

   public Item getItem(int id) {
       return items.get(id);
   }

   public void reload() {
      items = deserealizeFromFile();
   }
}
```

On each access of “items” field we check race on this **field**

On each call of “items” method we check race on this **object**

Each field of class Item is protected the same way as field “items” of class Storage

Synchronization Contract Example



T1

map.put(k,v)

Contract:

$o.put($k,*) → $o.get($k)

ConcurrentHashMap

x = map.get(k)

T2

Clocks Storing

- Thread clock
  - `ThreadLocal<VectorClock>`
- Field XXX
  - `volatile transient VectorClock XXX_vc;`
- Foreign objects, monitors
  - `WeakIdentityConcurrentHashMap<Object,VectorClock>`
- Volatiles, synchronization contracts
  - `ConcurrentHashMap <???, VectorClock>`

Composite Keys

- `AtomicLongFieldUpdater.CAS(Object o, long offset, long v)`
  - param 0 + param 1
- Volatile field “abc” of object o
  - object + field name
- `AtomicInteger.set() & AtomicInteger.get()`
  - object
- `ConcurrentMap.put(key, value) & ConcurrentMap.get(key)`
  - object + param 0

Solved Problems

- Composite keys for contracts and volatiles
  - Generate them on-the-fly

- Avoid unnecessary keys creation
  - ThreadLocal<MutableKeyXXX> for each CompositeKeyXXX

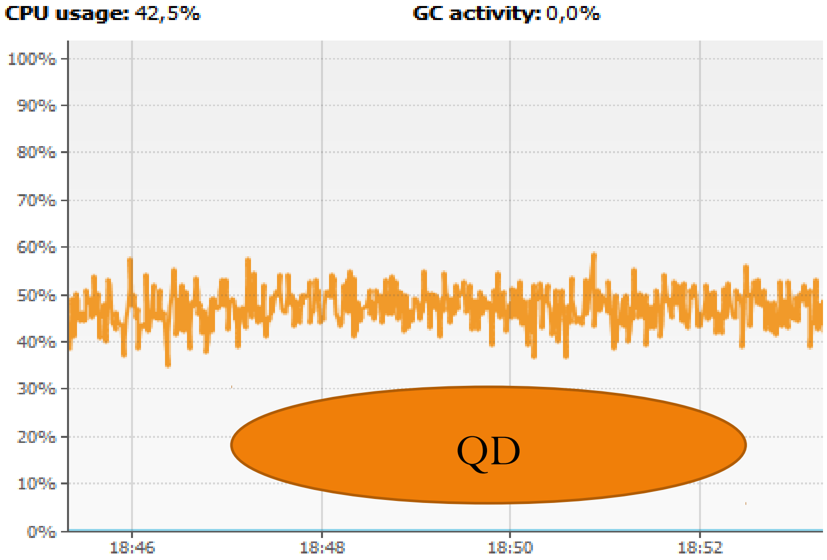
- Loading of classes, generated on-the-fly
  - Instrument ClassLoader.loadClass()

Solved Problems

- Don’t break serialization
  - compute `serialVersiodUid` before instrumentation

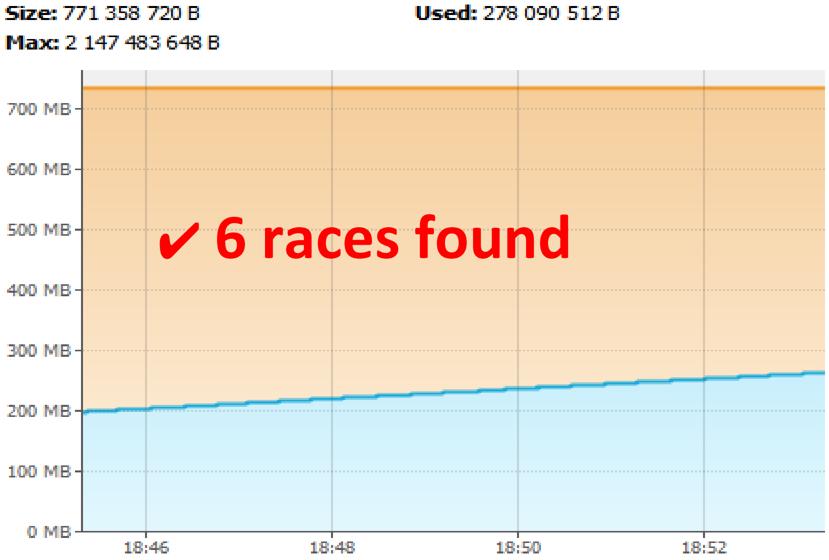
- Caching components of dead clocks
  - when thread dies, its time frames doesn’t grow anymore
  - cache frames of dead threads to avoid memory leaks
  - local last-known generation & global generation

DRD in Real Life: QD



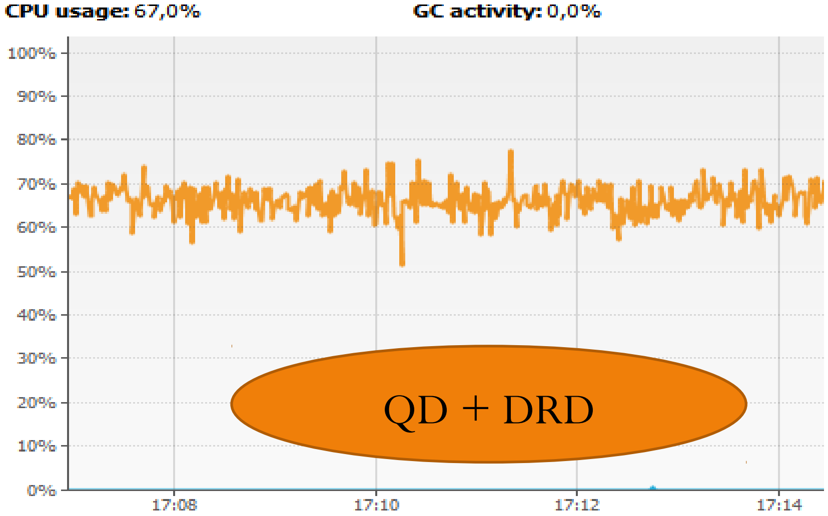
CPU usage: 42,5% **GC activity:** 0,0%

QD



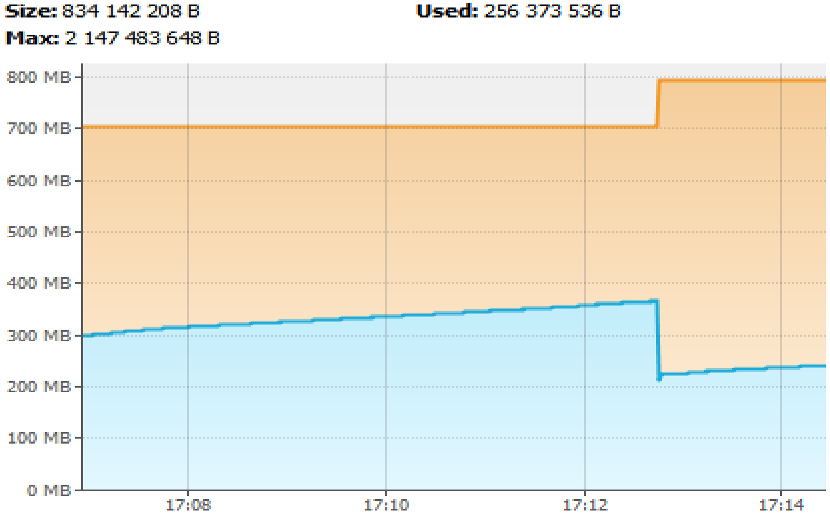
Size: 771 358 720 B **Used:** 278 090 512 B
Max: 2 147 483 648 B

✓ 6 races found



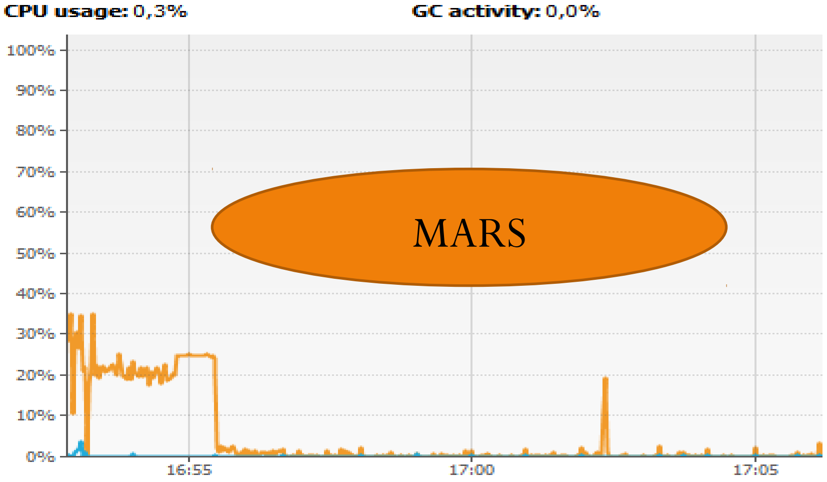
CPU usage: 67,0% **GC activity:** 0,0%

QD + DRD

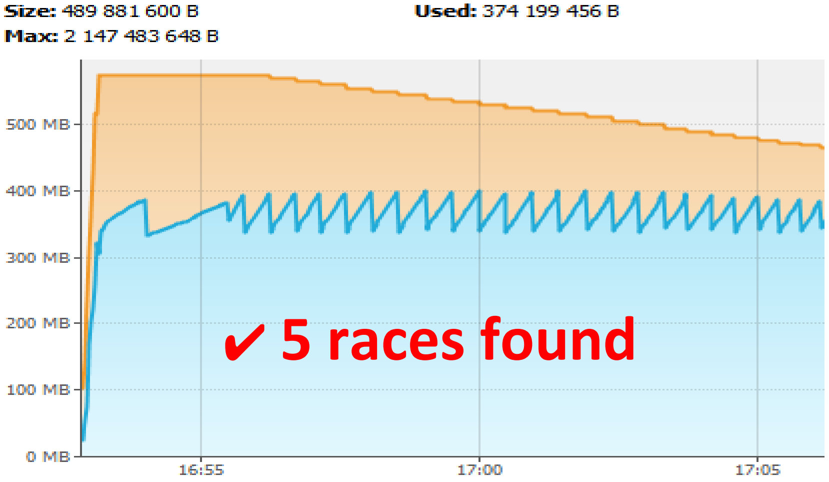


Size: 834 142 208 B **Used:** 256 373 536 B
Max: 2 147 483 648 B

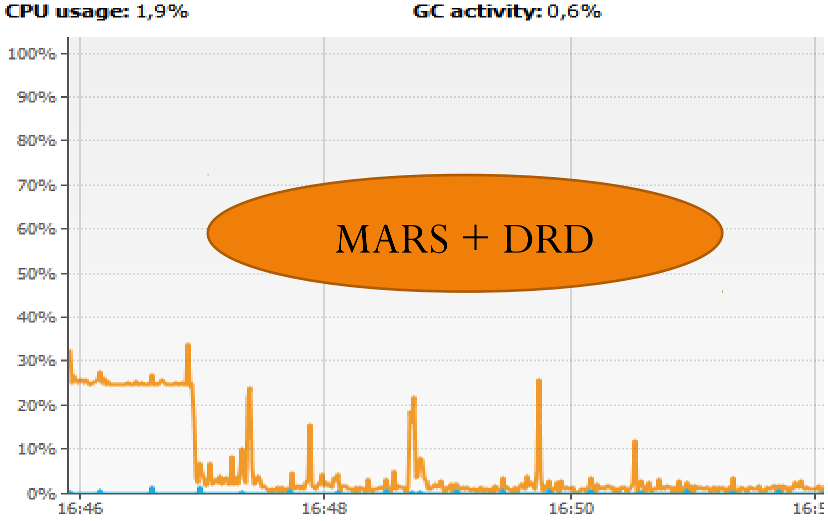
DRD in Real Life: MARS UI



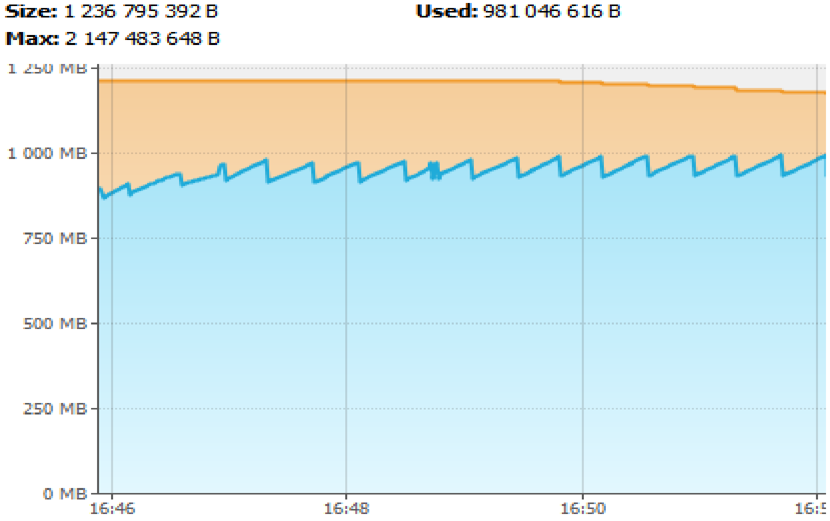
MARS



✔ 5 races found



MARS + DRD



DRD Race Report Example

```
WRITE_READ data race between current thread Thread-12(id = 33) and thread Thread-11(id =
    32)

Race target : field my/app/DataServiceImpl.stopped

Thread 32 accessed it in my/app/DataServiceImpl.access$400(line : 29)

-------------------------------Stack trace for racing thread (id = 32) is not available.------------

------------------------------------Current thread's stack trace (id = 33) : --------------------------
    at my.app.DataServiceImpl.stop(DataServiceImpl.java:155)
    at my.app.DataManager.close(DataManager.java:201)
    ...
```

DRD Advantages

- Doesn’t break serialization
- No memory leaks
- Few garbage
- No JVM modification
- Synchronization contracts
  - **very important:** Unsafe, AbstractQueuedSynchronizer

Links

- http://code.devexperts.com/display/DRD/: documentation, links, etc
- Contact us: drd-support@devexperts.com

- Useful links: see also on product page
  - IBM MSDK
  - ThreadSanitizer for Java
  - jChord
  - FindBugs

  - JLS «Threads and locks» chapter

Q & A