

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

November 1 - 2, Moscow



Software Engineering
Conference in Russia

Dynamic data race detection in concurrent Java programs



Vitaly Trifanov, Dmitry Tsitelov

Devexperts LLC

Agenda

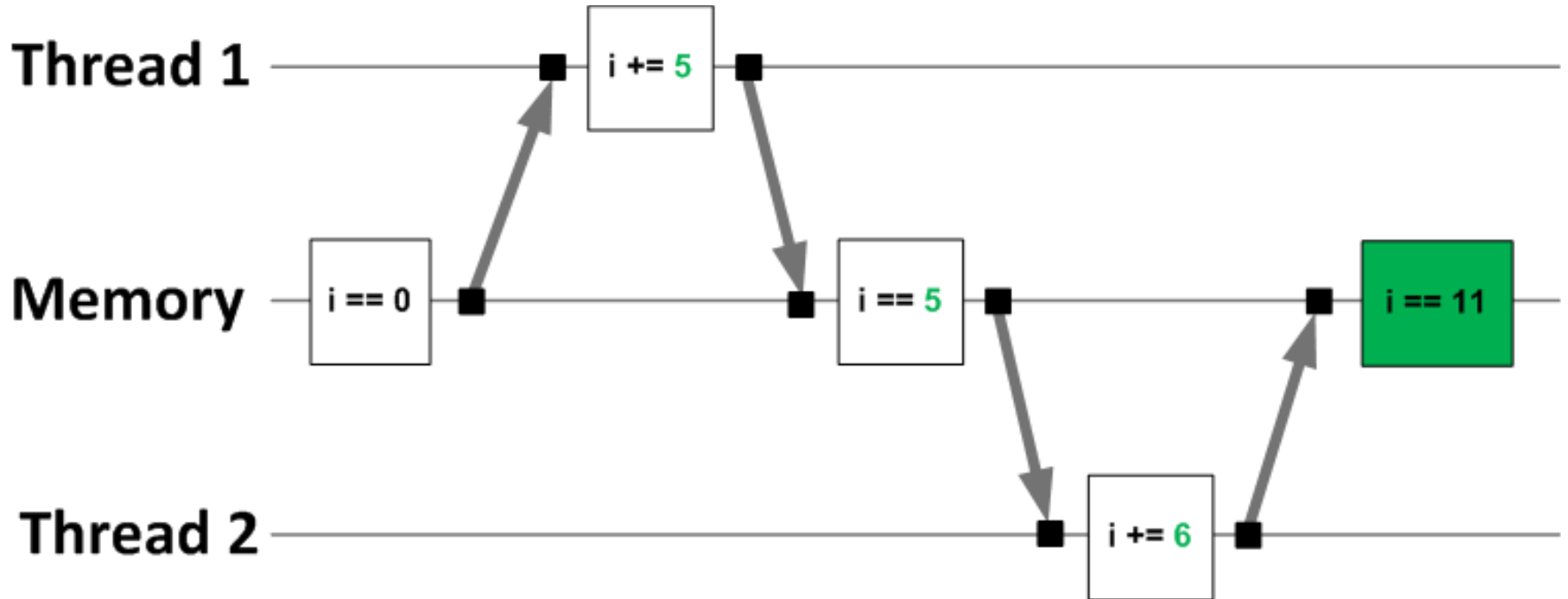
- What are data races and why they are dangerous
- Automatic races detection
 - approaches, pros & cons
- Happens-before race detection algorithm
 - Vector clocks
- Our dynamic race detector
 - implementation
 - solved problems

Data Race Example

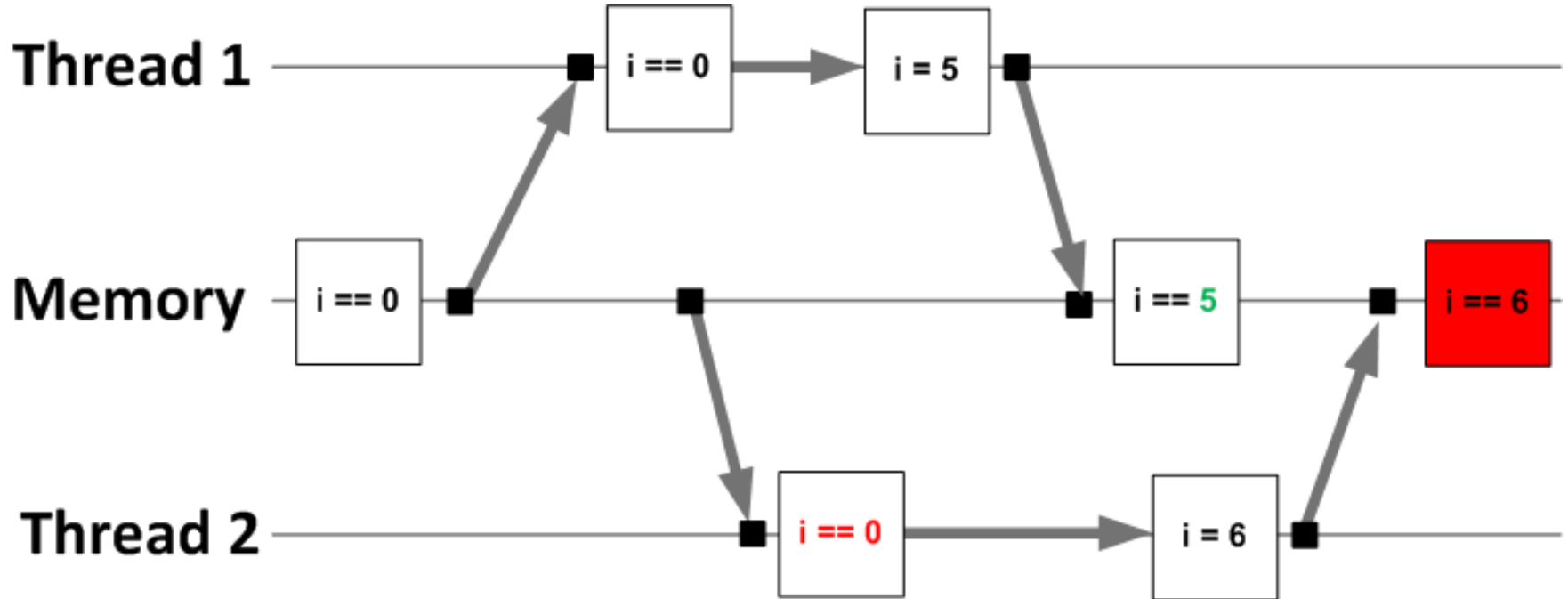
```
public class Account {
    private int amount = 0;
    public void deposit(int x) {amount += x;}
    public int getAmount() {return amount;}
}

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

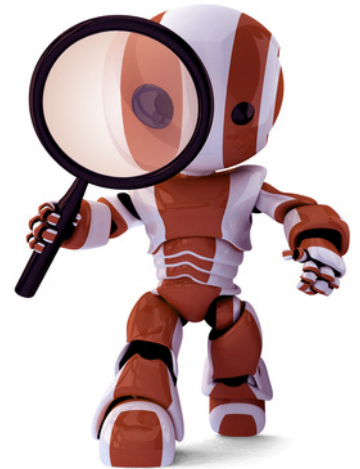


Data Races Are Dangerous

- 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

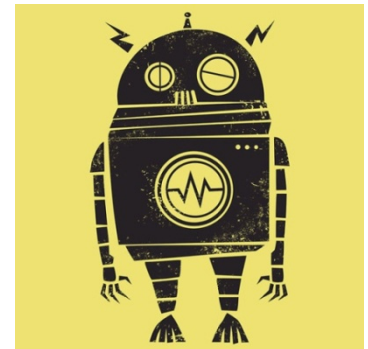


Dynamic Detectors vs Static



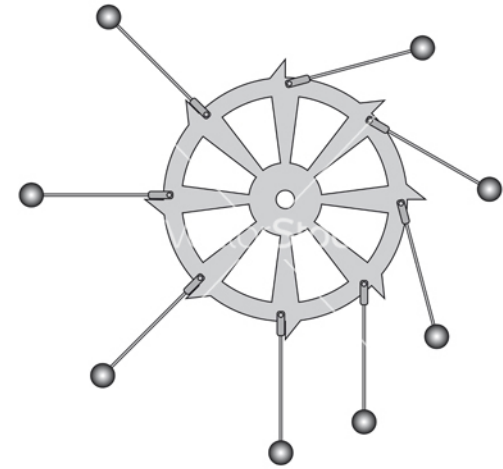
Static Approach

- 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



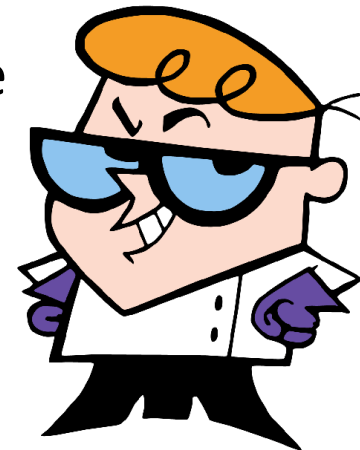
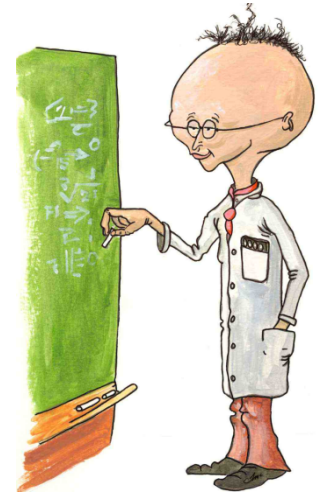
Dynamic Approach

- 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?

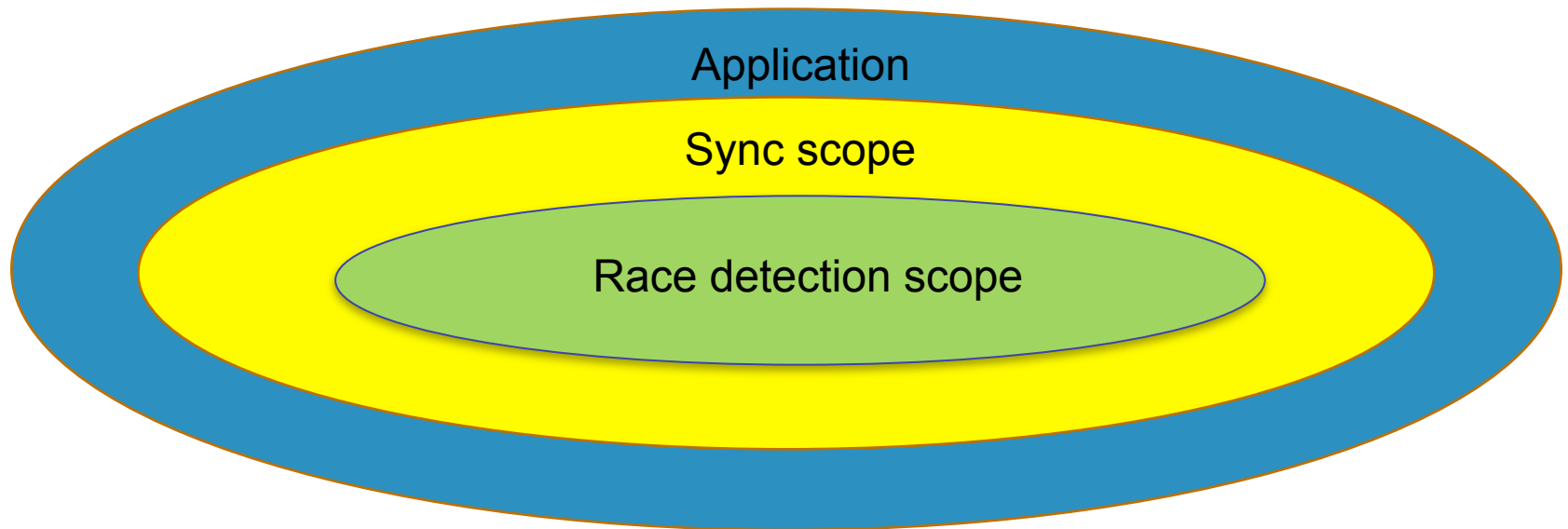
- Use both approaches 😊
- Static (FindBugs/Sonar, jChord, ...)
 - Eliminate provable synchronization inconsistencies on the early stage
- Dynamic
 - Try existing tools, but they are unstable
 - IBM MSDK, Thread Sanitizer for Java
 - That's why we've developed our own!



Data Race Detector Concept

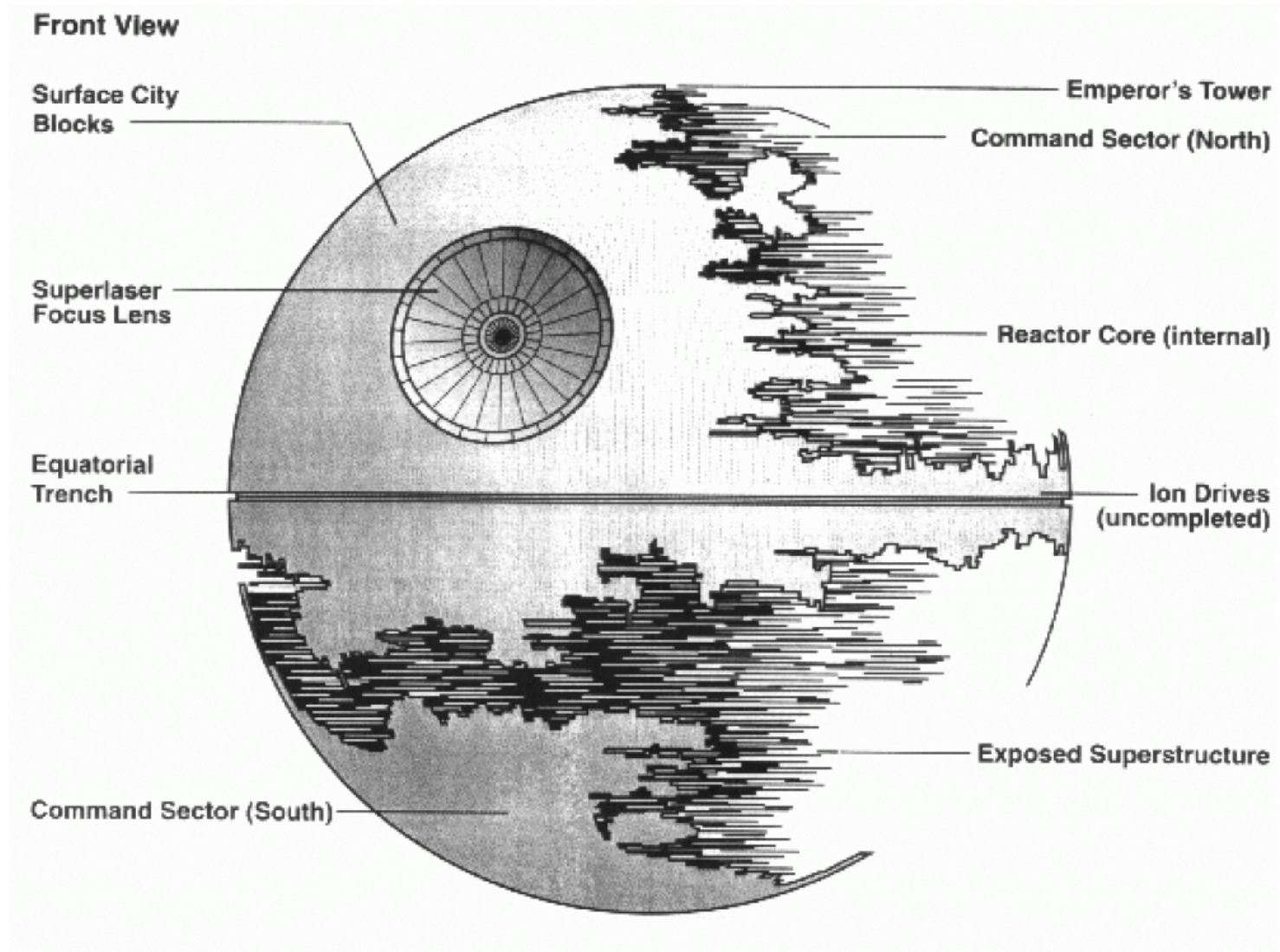
- Application uses libraries and frameworks via API
 - At least JRE
- API is well documented
 - “Class XXX is thread-safe”
 - “Class YYY is not thread-safe”
 - “XXX.get() is synchronized with preceding call of XXX.set()”
- Describe behavior of API and exclude library from analysis

Code Analysis Scope



- Application: whole app code with JRE, libraries, etc.
- Sync scope: intercept synchronization operations
- Race detection scope: detect races

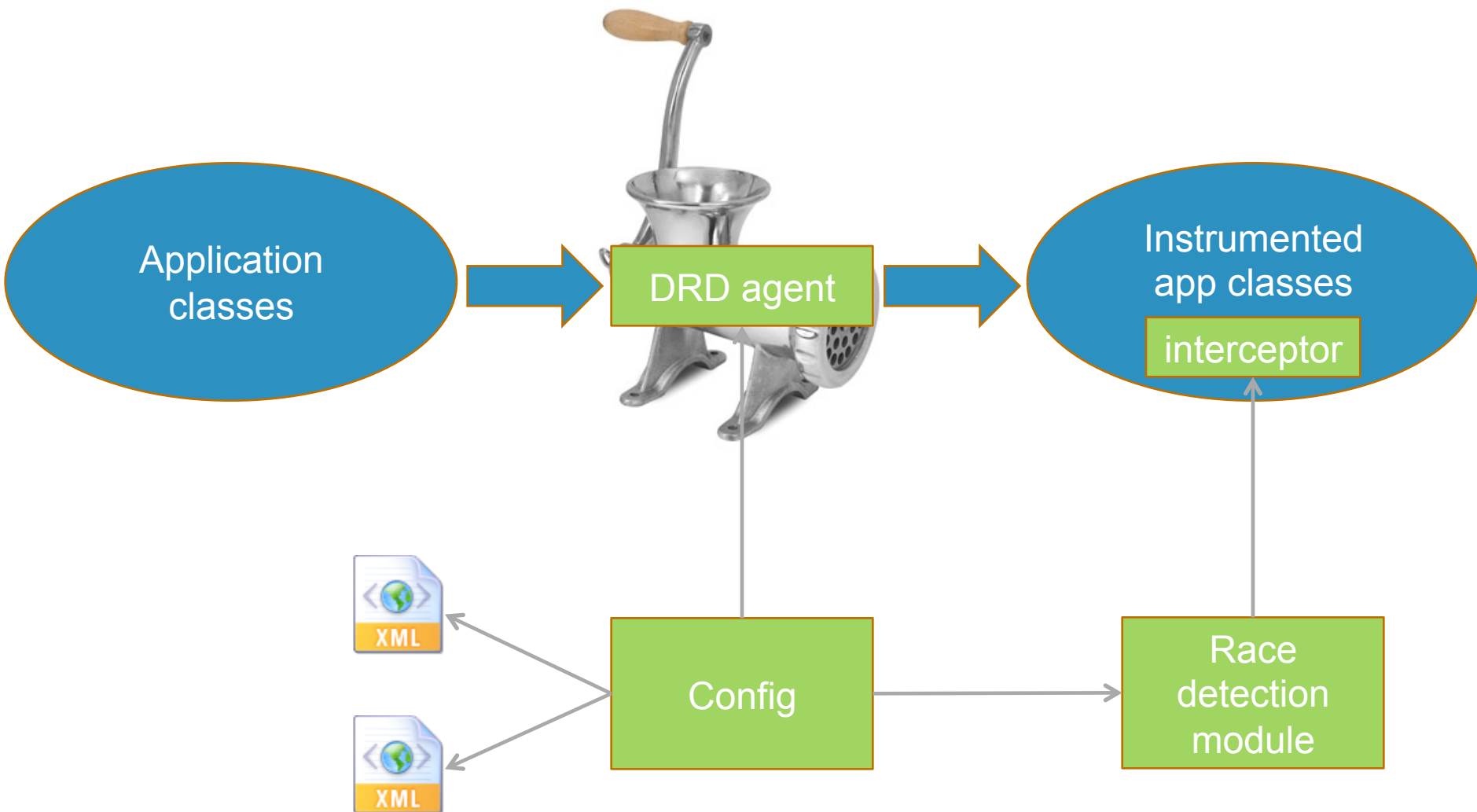
DRD: How It's Organized



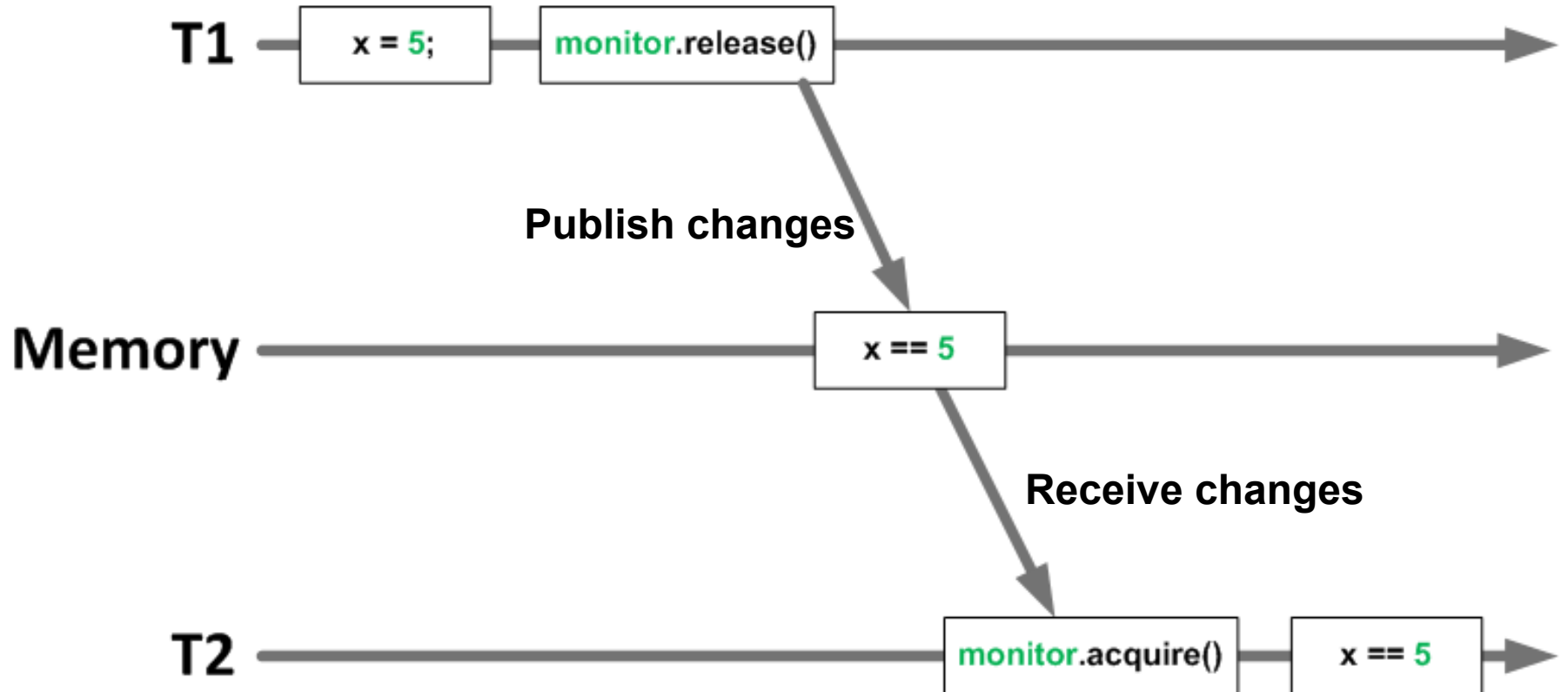
What Operations to Intercept?

- Synchronization operations
 - thread start/join/interrupt
 - synchronized
 - volatile read/write
 - `java.util.concurrent`
- Accesses to shared data
 - fields
 - objects

How It Works



JLS: Publishing Data



JLS: Synchronized-With Relation

- “Synchronized-with” relation
 - unlock monitor $M \mapsto$ all subsequent locks on M
 - volatile write \mapsto all subsequent volatile reads
 - ...
- Notation: **send** \mapsto **receive**

JLS: Happens-Before & Data Races

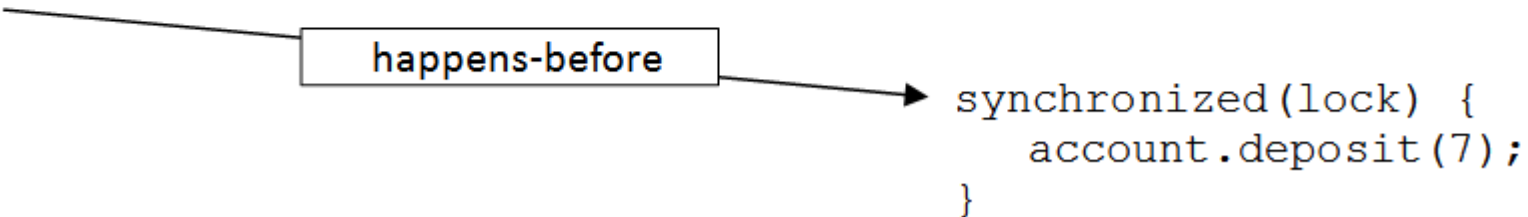
- X happens-before Y, when
 - X, Y - in same thread, X before Y in program order
 - X is synchronized-with Y
 - Transitivity: exists Z: $hb(X, Z) \ \&\& \ hb(Z, Y)$
- Data race: 2 conflicting accesses, not ordered by happens-before relation

Happens-Before Example

Thread 1

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

happens-before



Thread 2

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

- **No data race**

Happens-Before Data Race Detection

- Track all synchronization events
 - Store synchronized-with edges between them
 - At each shared field *read* R from T2 check if last *write* W from T1 happened-before it - i.e. there were events A in T1 and B in T2:
 - B happened-before R
 - W happened-before A
 - A synchronized-with B
- $W \rightarrow A \rightarrow B \rightarrow R$
- At each shared field *write* W check if
 - last write W_{last} happened-before W
 - All reads R_0, \dots, R_N , happened-after W_{last} , happened-before W

Vector Clock



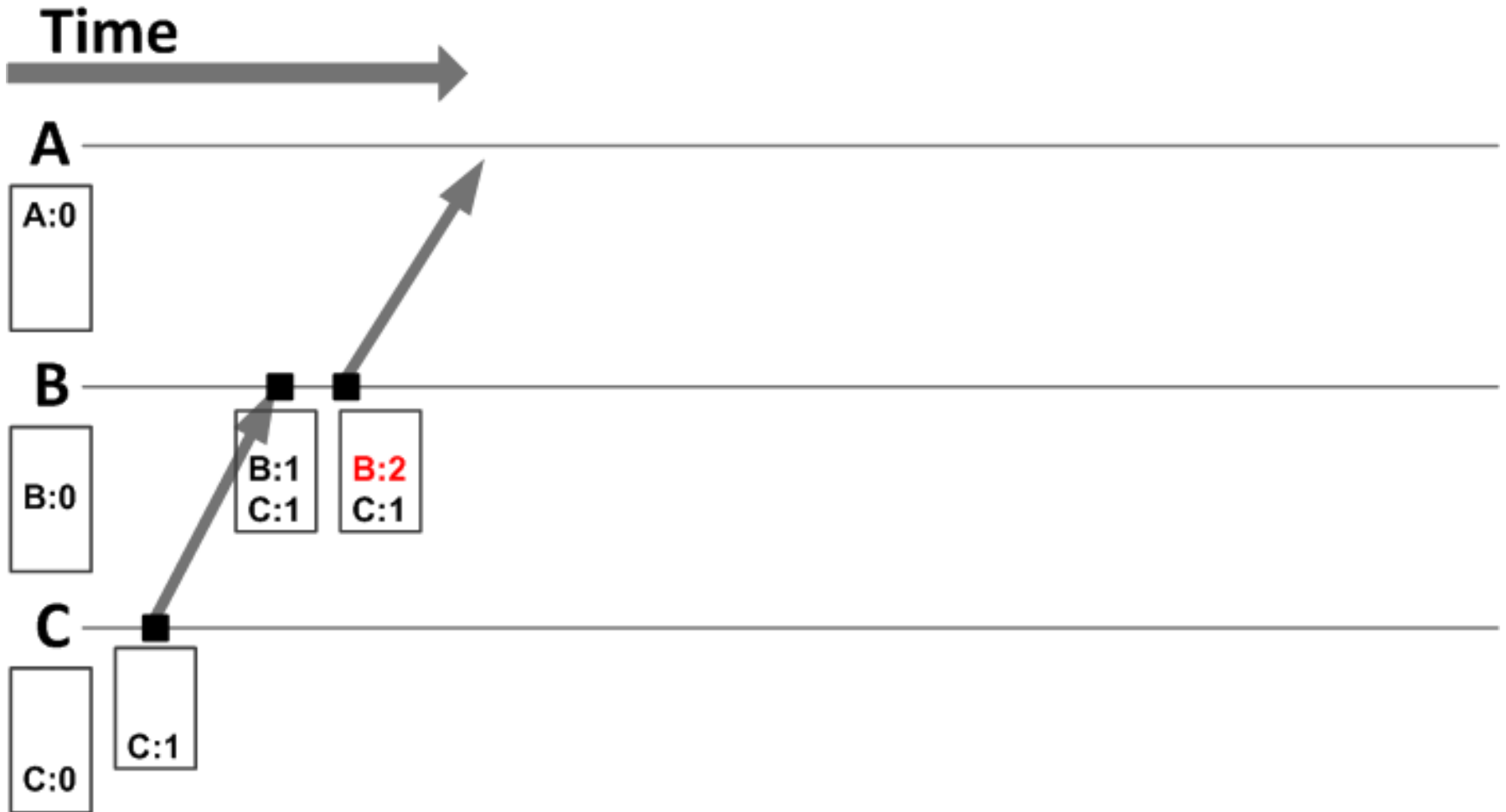
Vector Clock



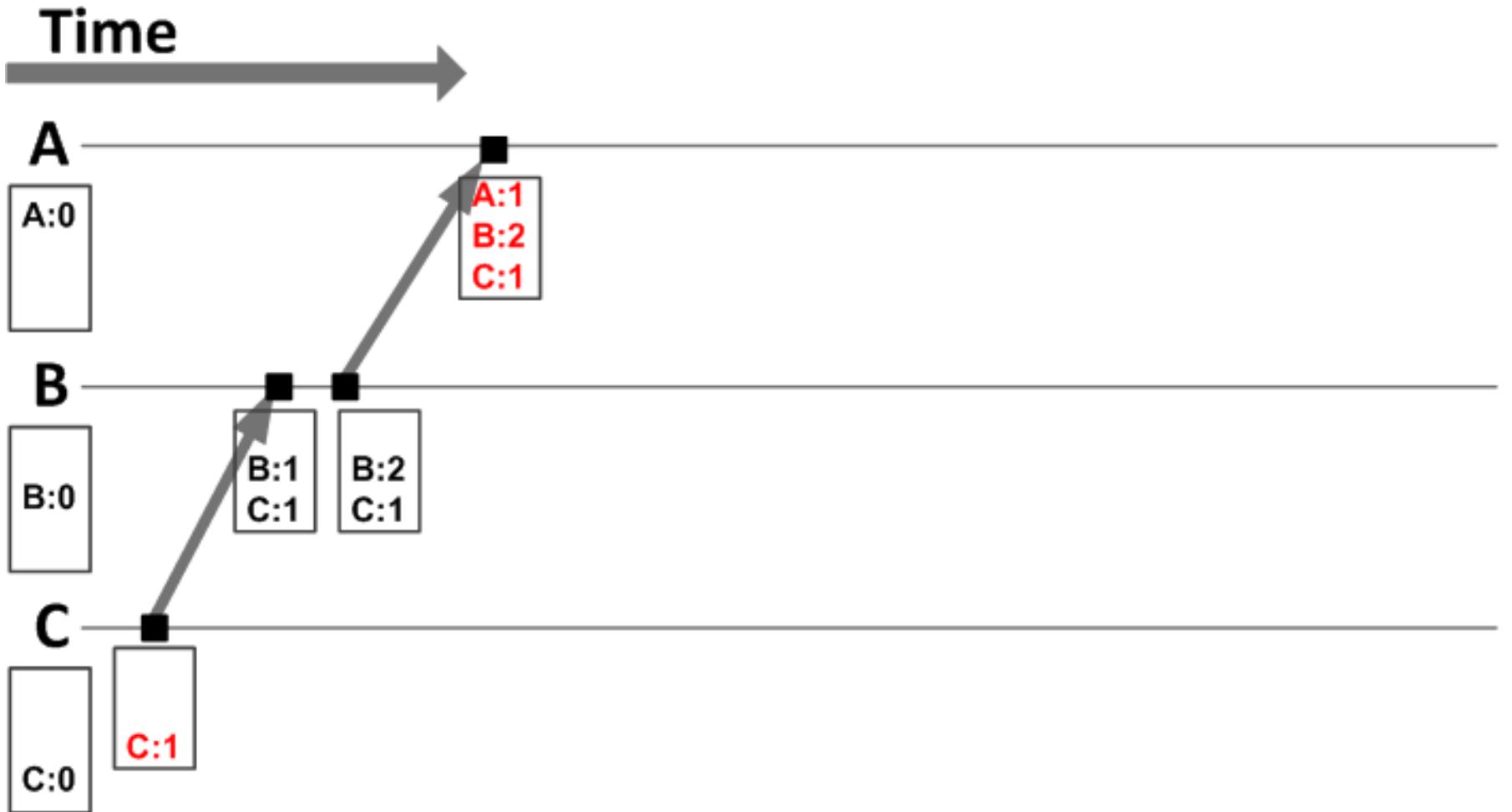
Vector Clock



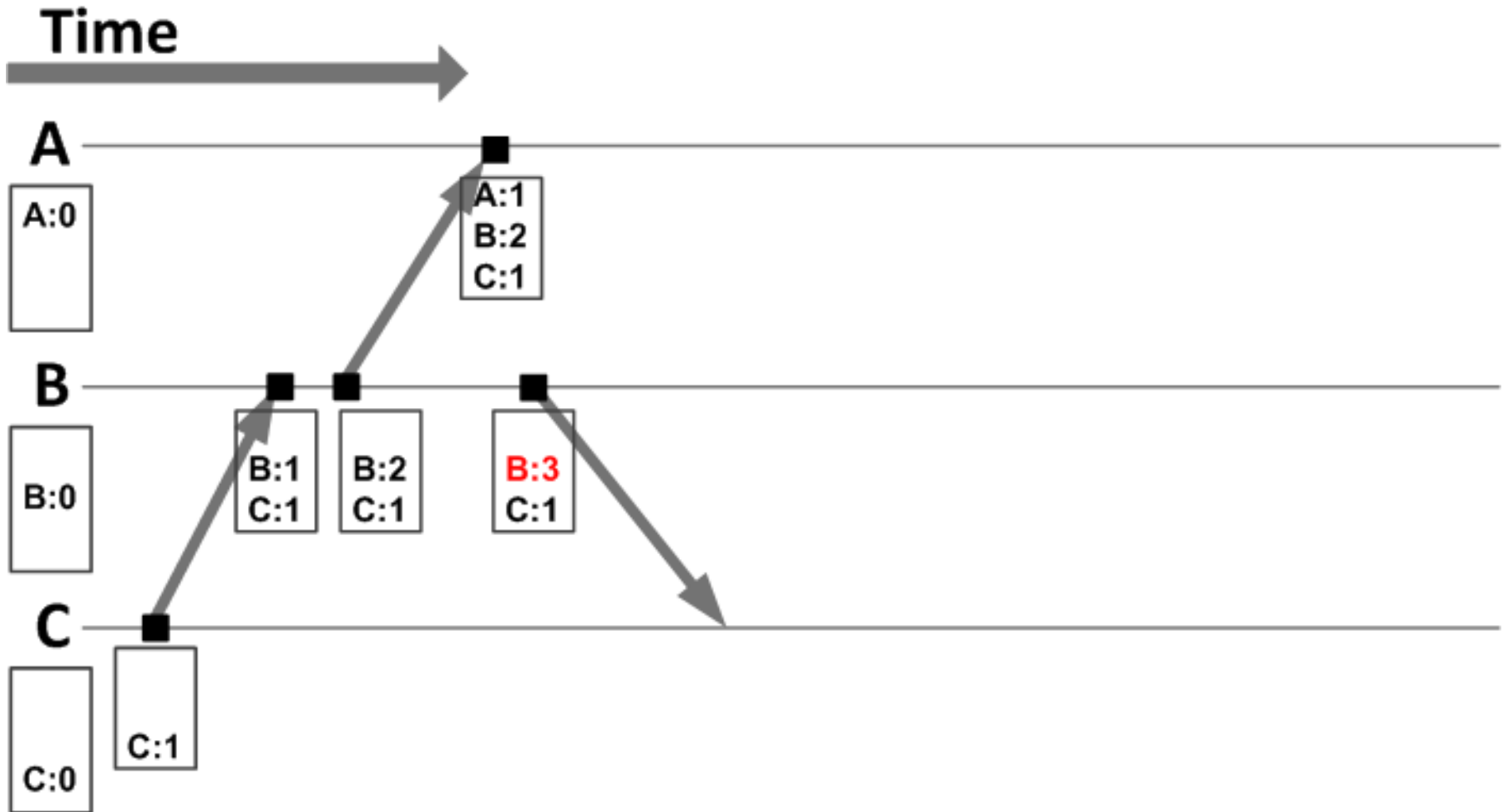
Vector Clock



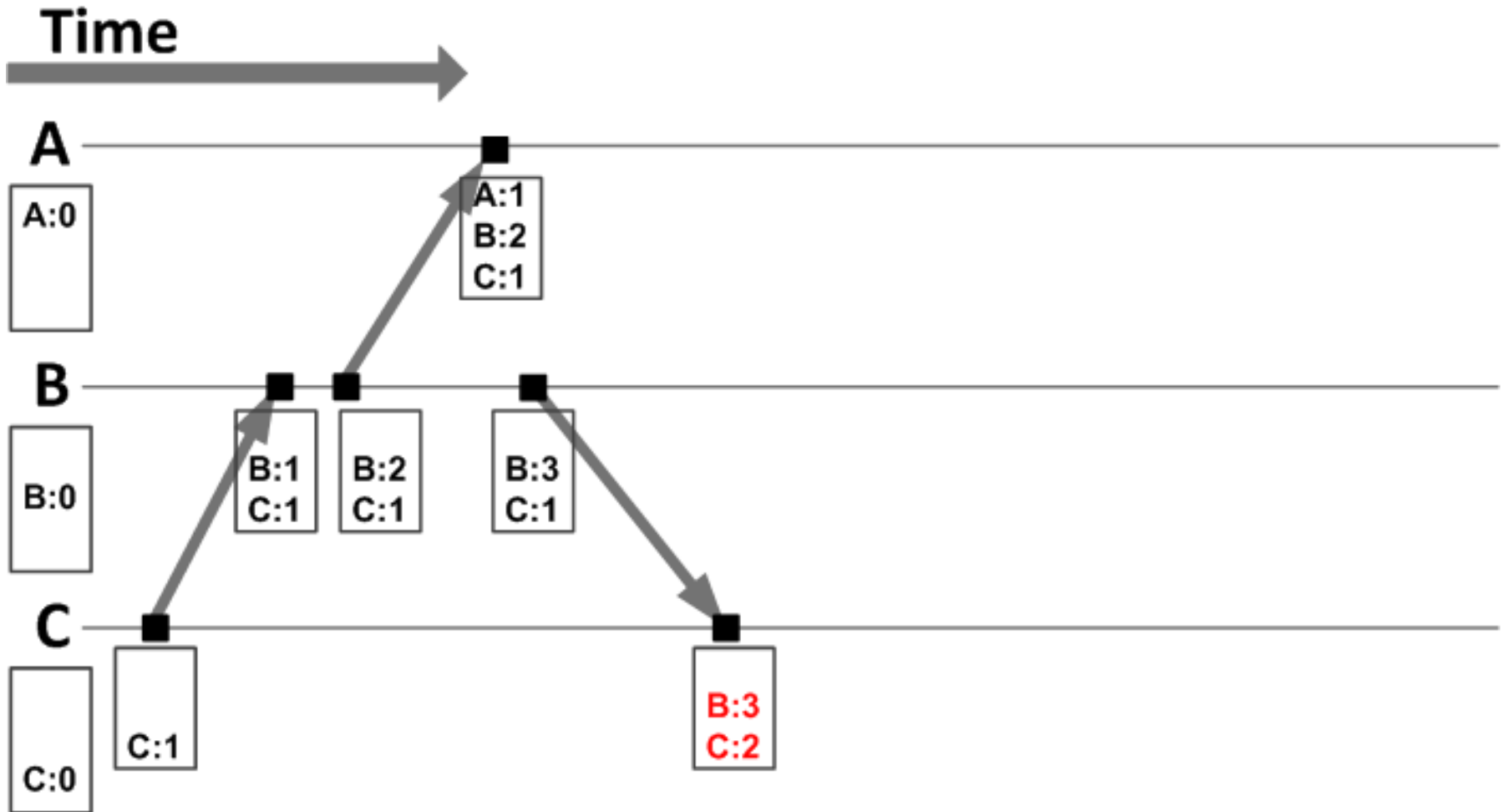
Vector Clock



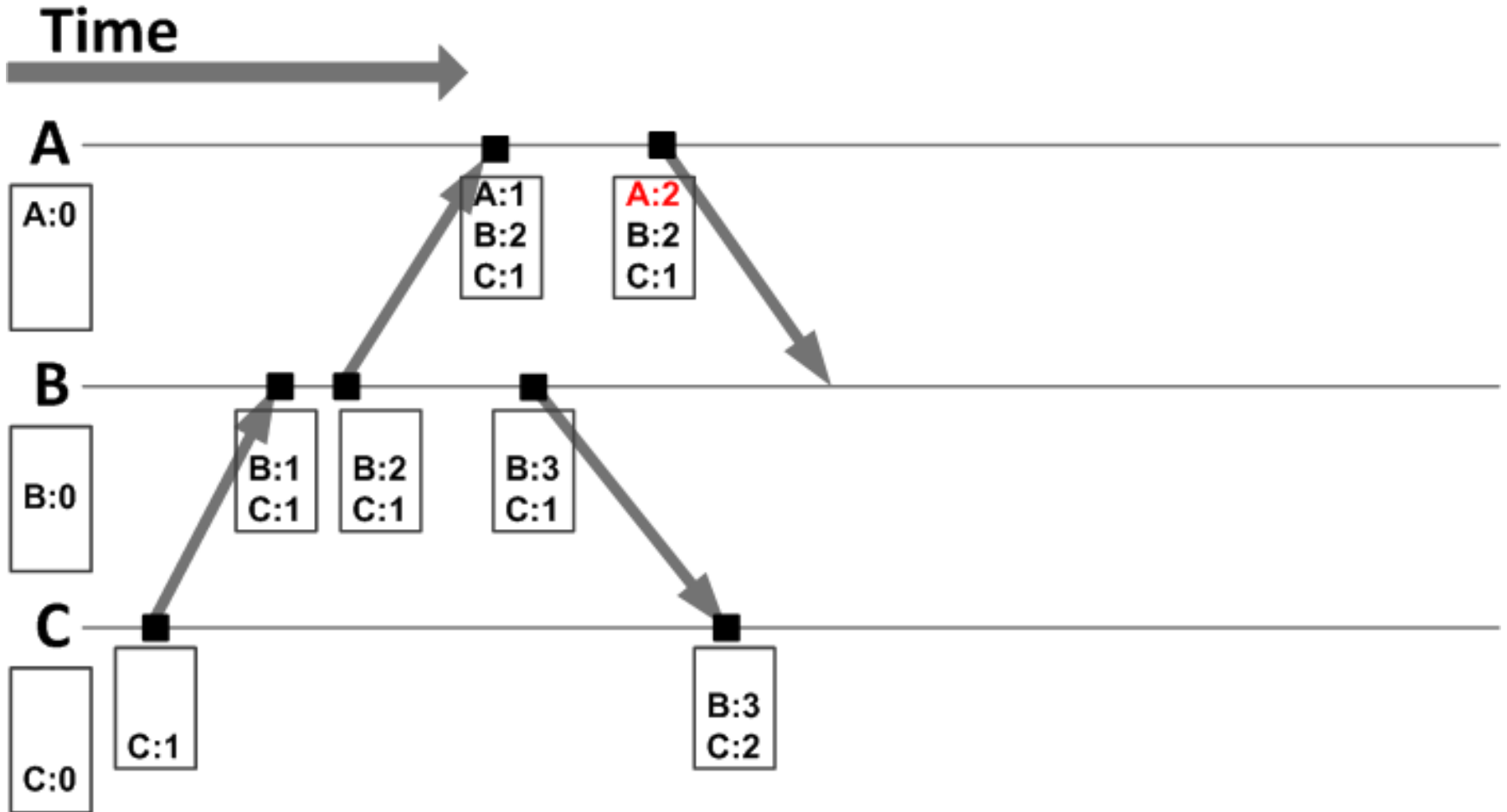
Vector Clock



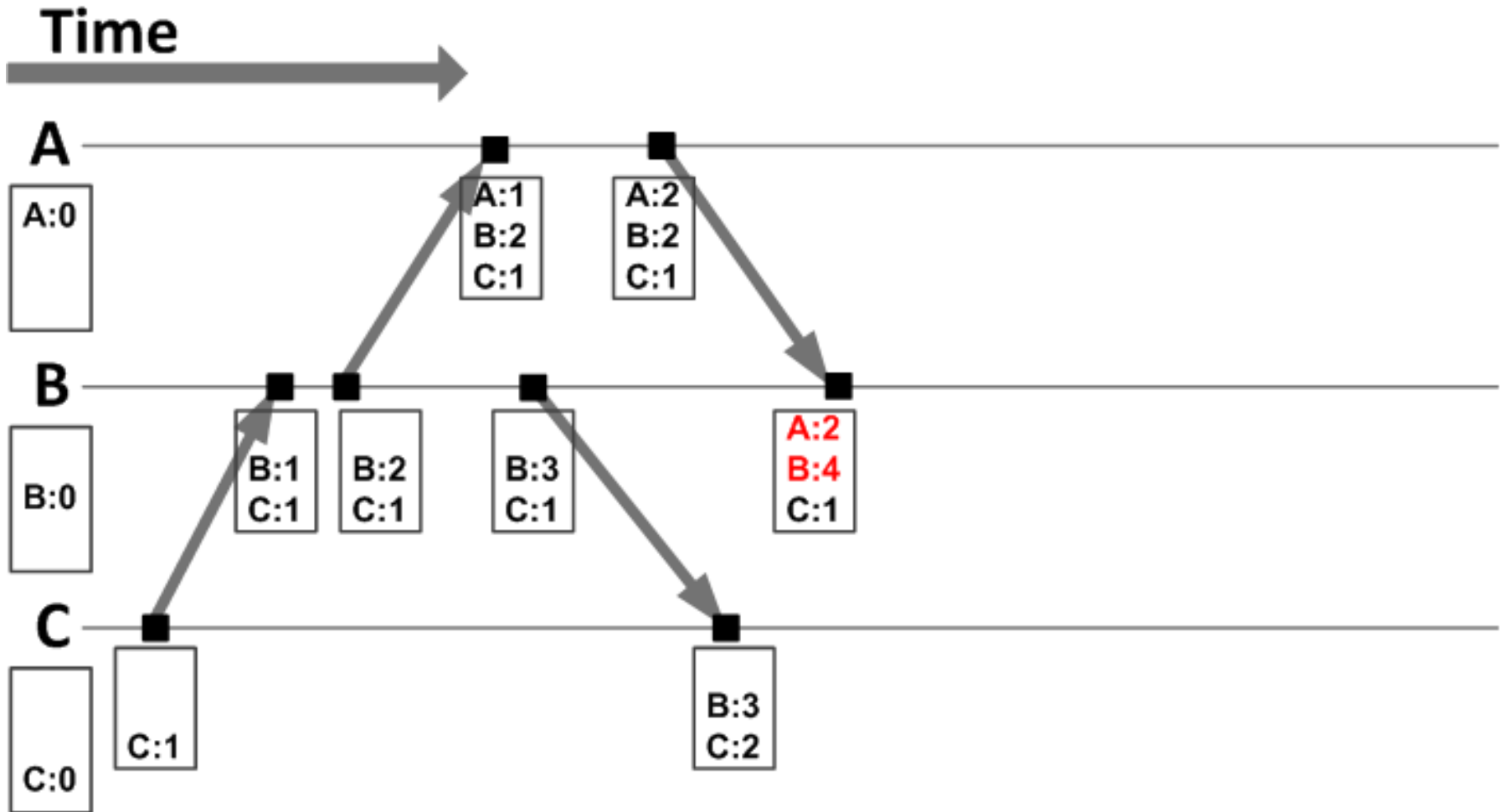
Vector Clock



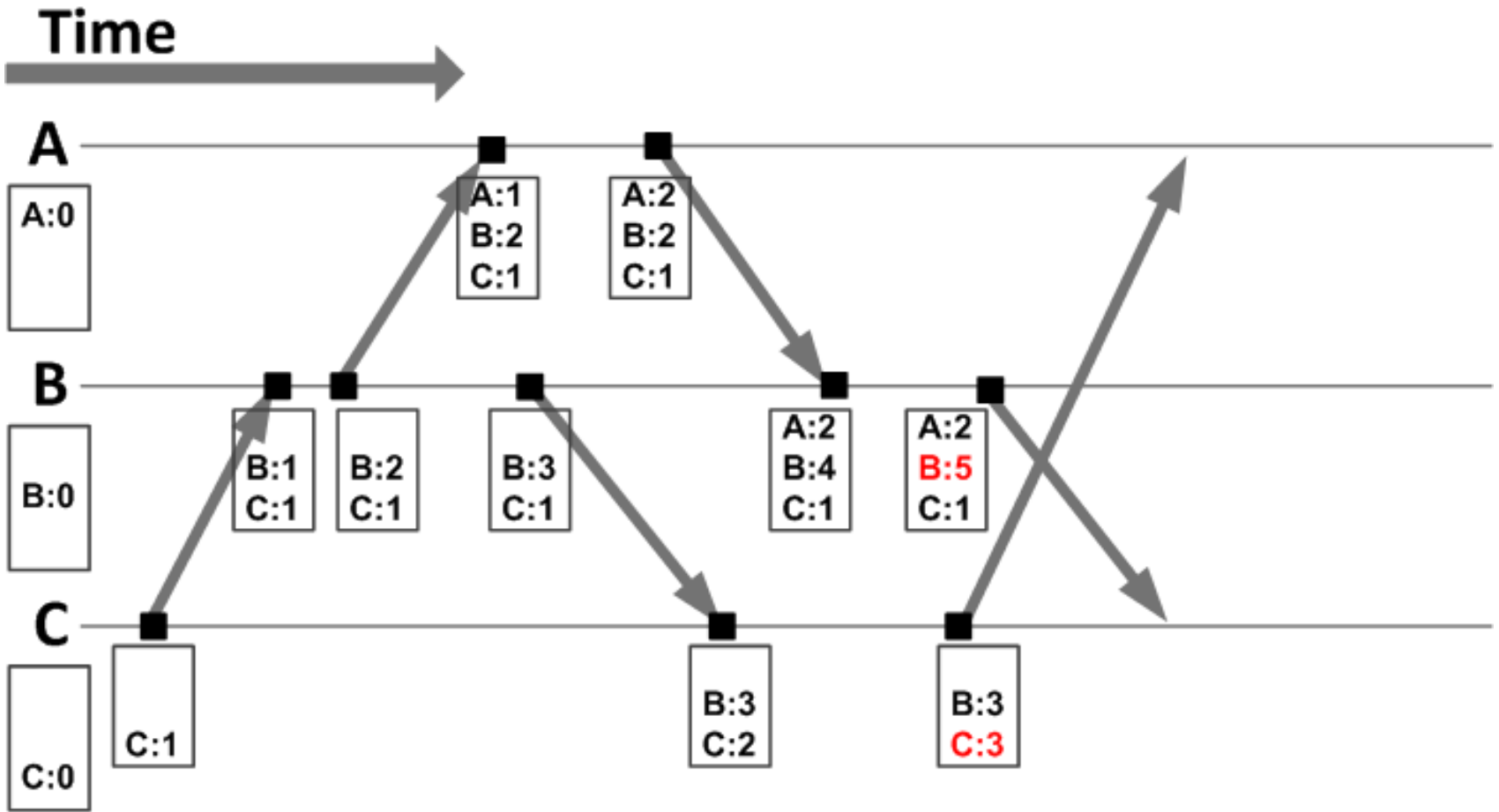
Vector Clock



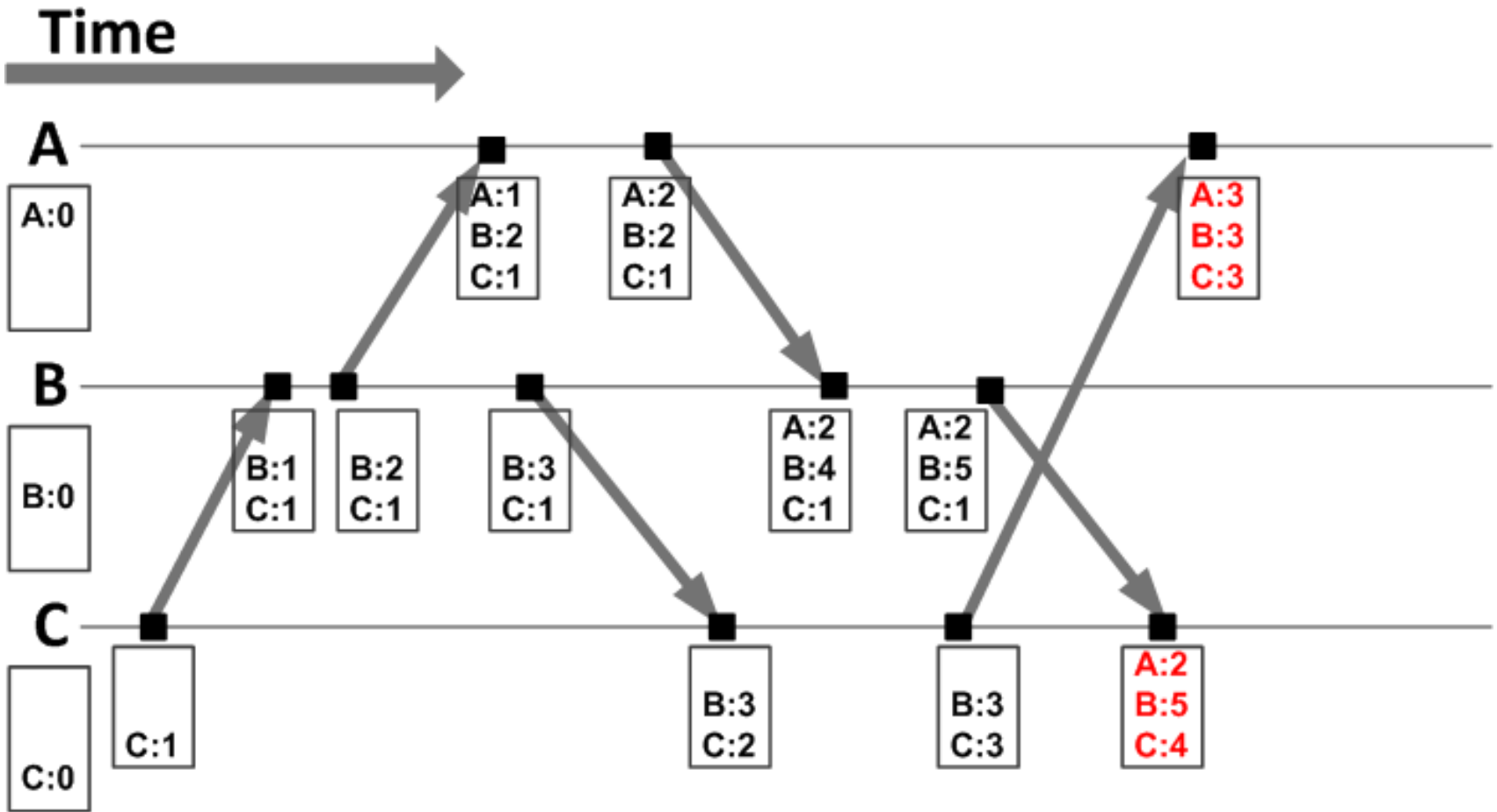
Vector Clock



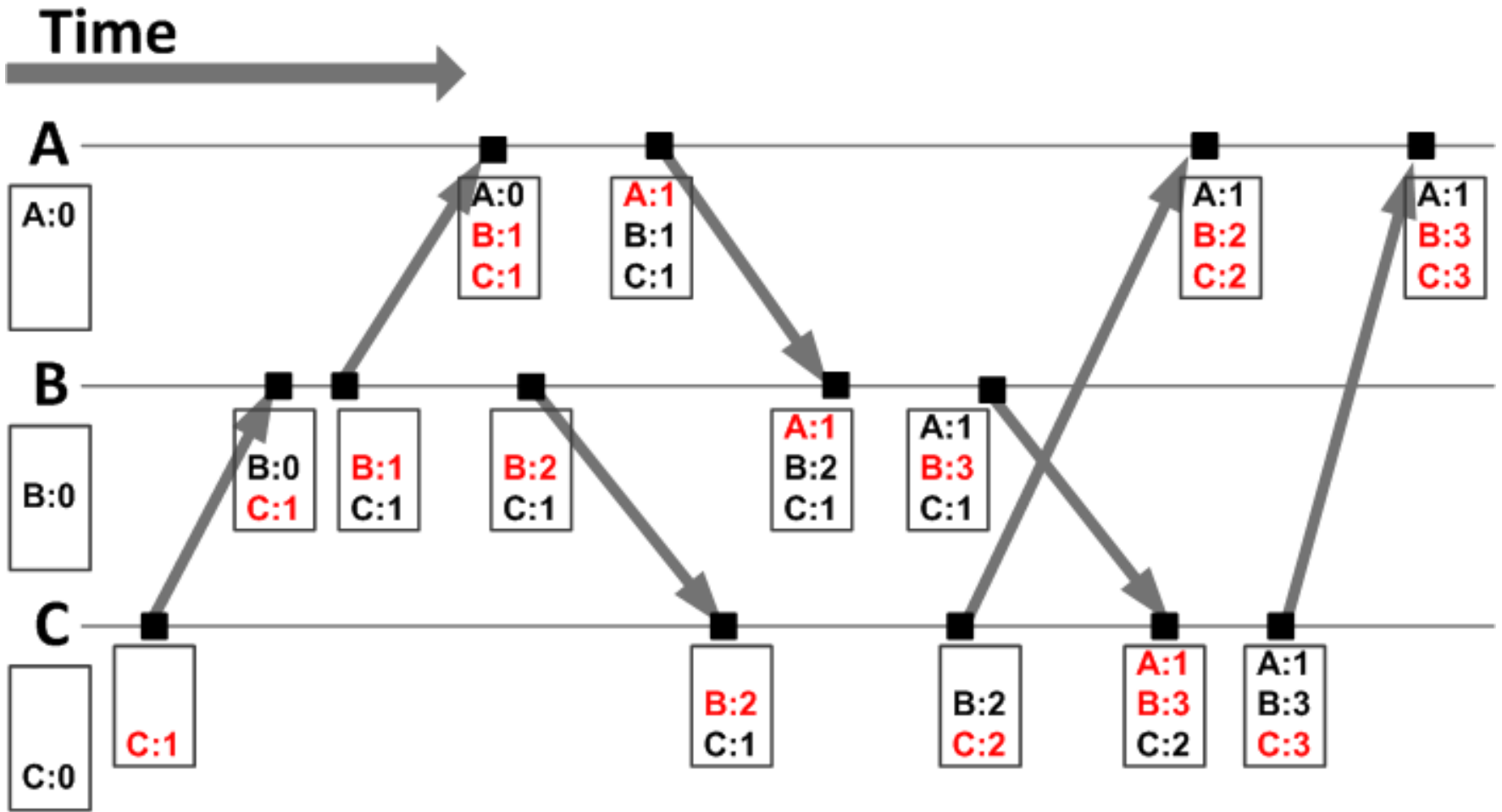
Vector Clock



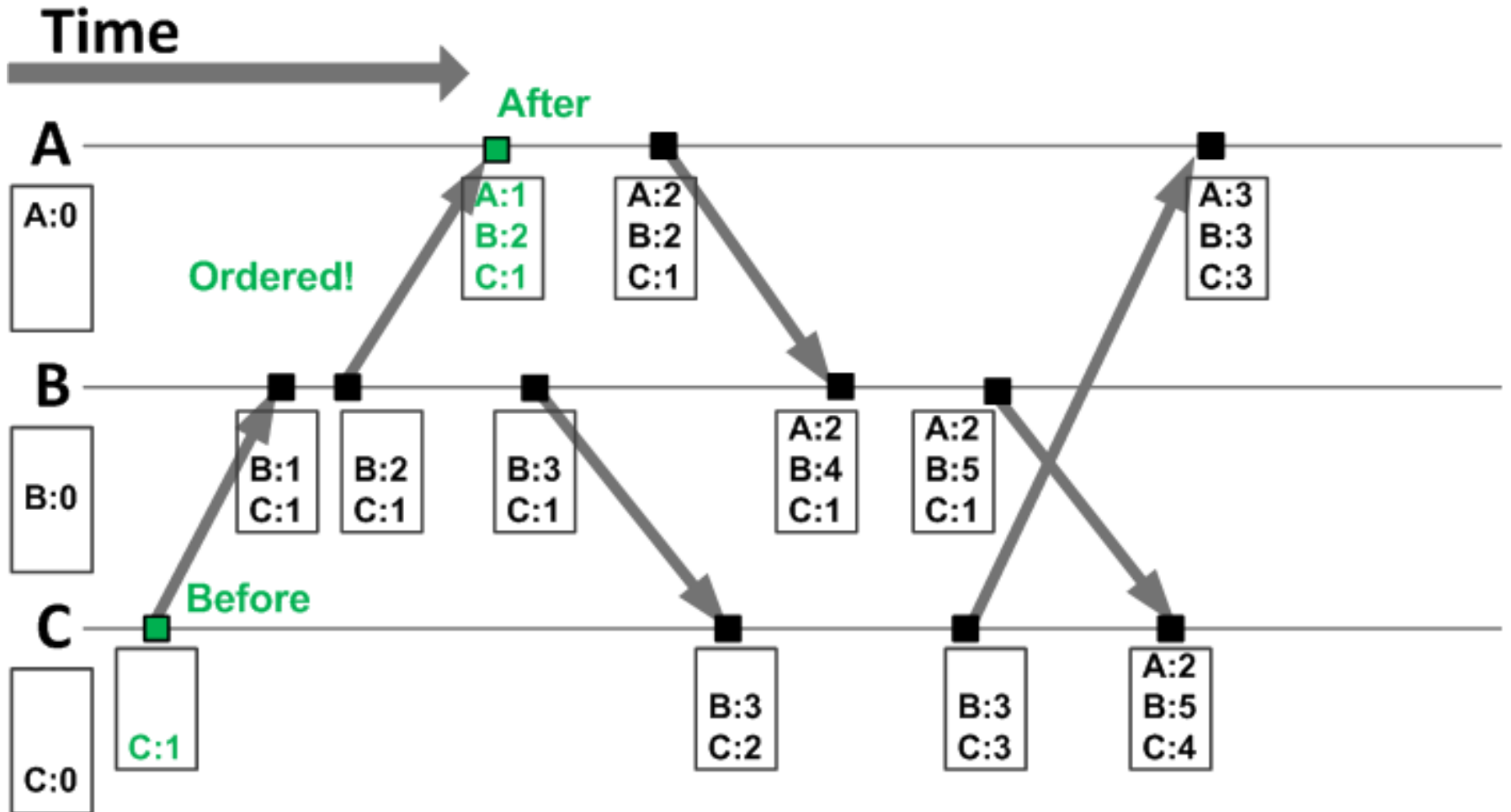
Vector Clock



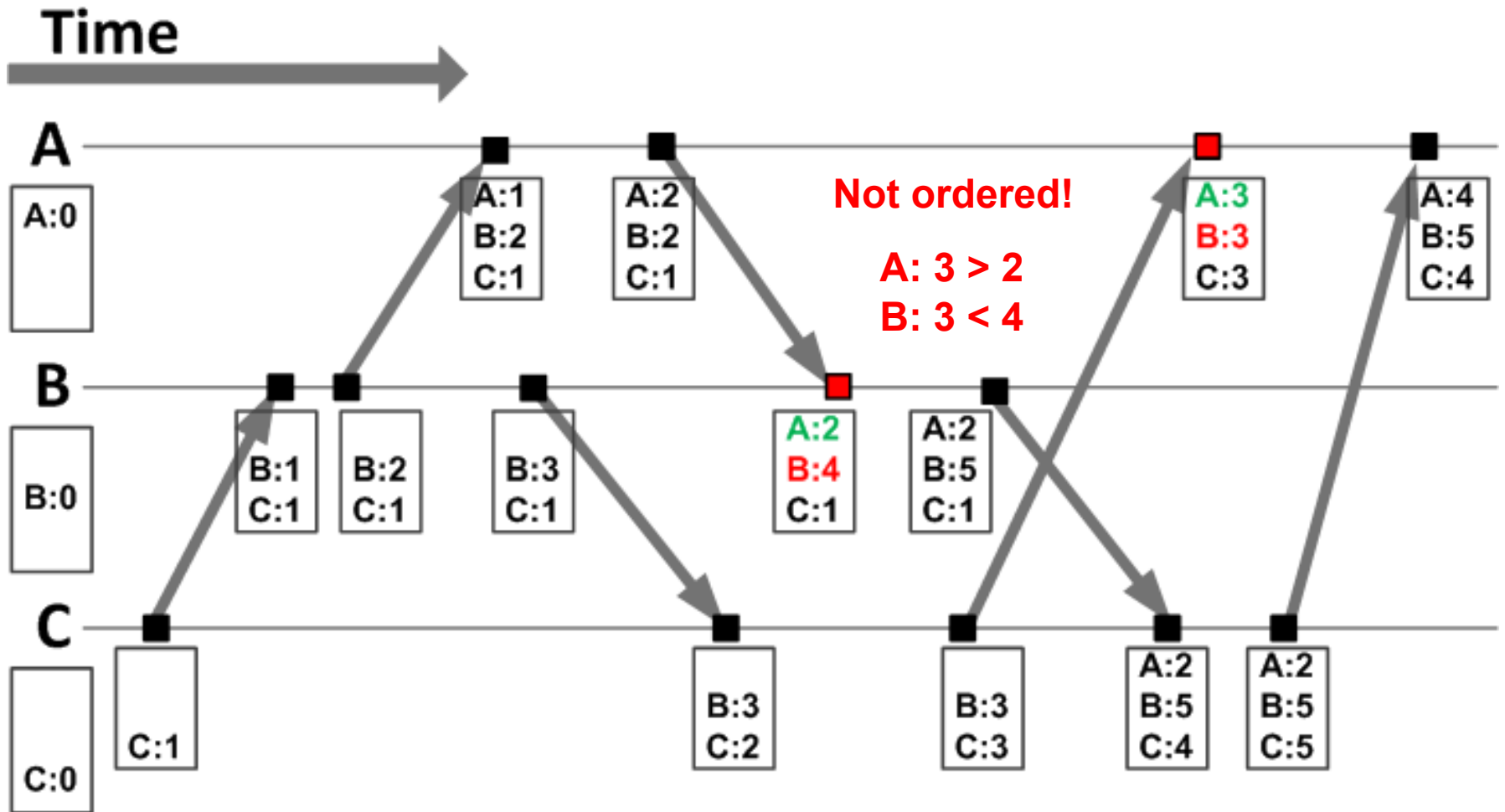
Vector Clock



Vector Clock



Vector Clock



Happens-Before Algorithm

- Vector clock for each thread
- Vector clock for each synchronization object
- Algorithm
 - Event A: T_1 accessed object X at time T_A
 - Event B: T_2 accessed object X at time T_B
 - B occurred after A in program order
 - **$T_1.VC[1] \text{ at } T_A \geq T_2.VC[1] \text{ at } T_B \Rightarrow !hb(A,B)$.**
- Write and read vector clocks for each shared object
- At each access to some shared object:
 - check thread clock against object clock
 - store thread into object clock

How It Works. No Data Race Example


Thread T₁

T₁.VC=[5,10]

Thread T₂

T₂.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 T₁

T₁.VC=[5,10]

Thread T₂

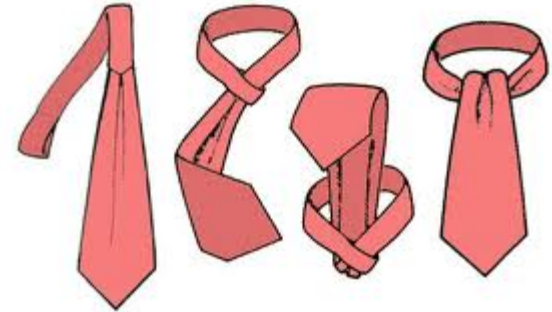
T₂.VC=[3,12]

```
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
```

How We Do It

- So, we store vector clock for each
 - shared object/variable
 - synchronization object
 - thread
- Update corresponding clocks before/after intercepted operations
- Huge impact, um?
 - Not huge-huge-huge: we have config and optimizations



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);
    }

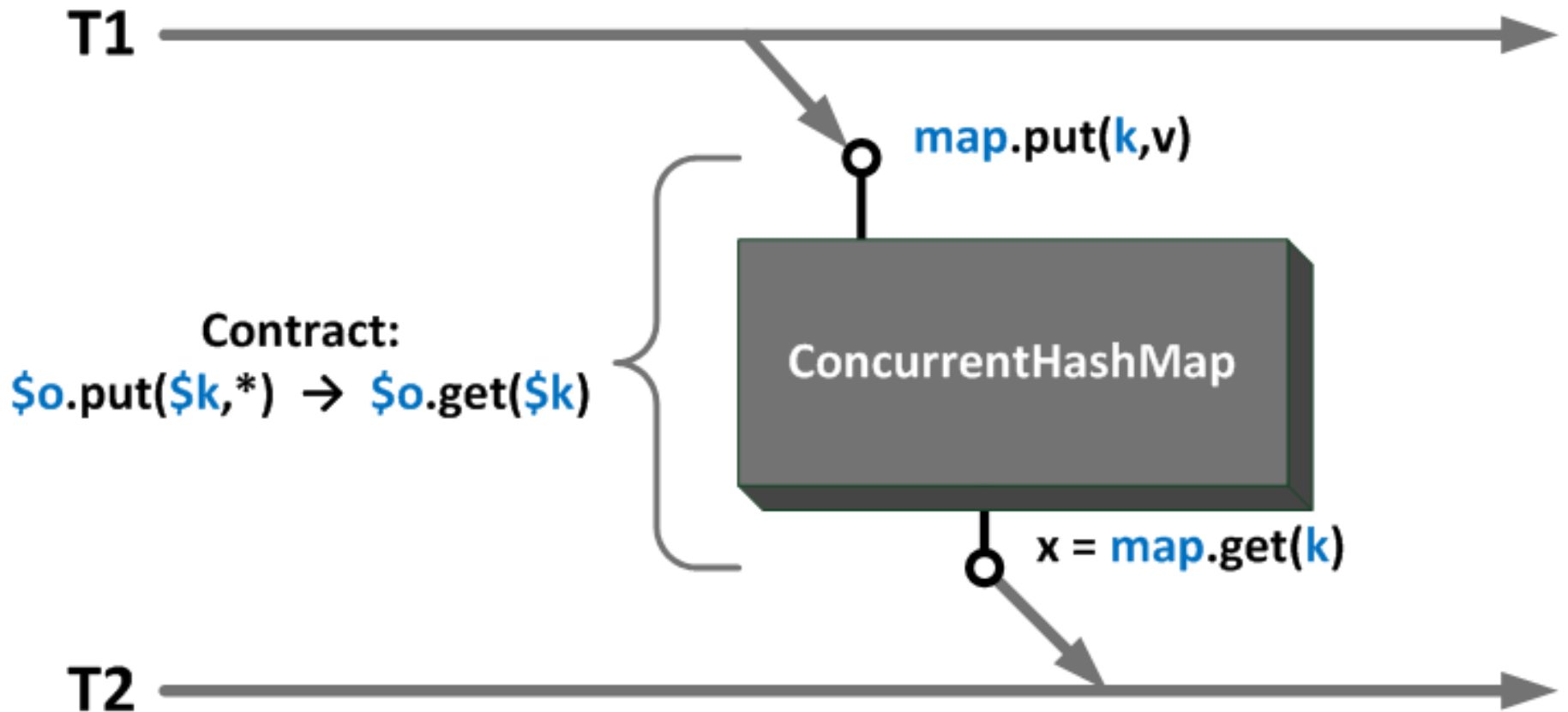
    public void reload() {
        items = deserealizeFromFile();
    }
}
```

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



Synchronization Contract Example

```
<Multiple-Sync owner="java.util.concurrent.atomic.AtomicLong">
  <Multiple-Links>
    <Multiple-Link type="owner"/>
  </Multiple-Links>
  <Call type="receive" name="get" descriptor="()J"/>
  <Call type="send" name="set" descriptor="(J)V"/>
  <Call type="full" name="getAndSet" descriptor="(J)J"/>
  <Call type="full" name="compareAndSet" descriptor="(JJ)Z"
    shouldReturnTrue="true"/>
  <Call type="full" name="getAndIncrement" descriptor="()J"/>
  <Call type="full" name="getAndDecrement" descriptor="()J"/>
  <Call type="full" name="getAndAdd" descriptor="(J)J"/>
  <Call type="full" name="incrementAndGet" descriptor="()J"/>
  <Call type="full" name="decrementAndGet" descriptor="()J"/>
  <Call type="full" name="addAndGet" descriptor="(J)J"/>
</Multiple-Sync>
```

Synchronization Contract Example

```
<Sync>
  <Links>
    <Link send="owner" receive="owner"/>
    <Link send="param" send-number="0" receive="param" receive-number="0"/>
  </Links>
  <Send>
    <MethodCall owner="java.util.concurrent.ConcurrentMap" name="put"
      descriptor="(Ljava/lang/Object;Ljava/lang/Object;)Ljava/lang/Object;"/>
  </Send>
  <Receive>
    <MethodCall owner="java.util.concurrent.ConcurrentMap" name="get"
      descriptor="(Ljava/lang/Object;)Ljava/lang/Object;"/>
  </Receive>
</Sync>
```

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

- Volatile: owner object + field name
- Synchronization contract:
 - owner?
 - parameters?
 - It's specified in configuration!
- We need composite keys
- Generate them on-the-fly

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

Composite Keys

- Always have instance field of type Object
 - make it WeakRef to avoid memory leaks
- Different for different cases (contracts)
 - parse synchronization config and generate them on-the-fly
 - instrument loadClass() method in all ClassLoader-s
- Used as keys in ConcurrentHashMap
 - calculate & store hashcode immediately after construction
 - override hashCode() and equals()
- Instances of CompositeKeyXXX are frequently created
 - maintain ThreadLocal<ReusableCompositeKeyXXX>
 - override equals() to make them comparable to each other

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

- Composite keys performance
 - Calculate hashcode after construction
 - Override hashcode() and equals()
- Garbage producing
 - ThreadLocal<MutableKey> for map keys
- Loading of classes, generated on-the-fly
 - Instrument ClassLoader.loadClass()

Solved Problems

- Don't break serialization
 - compute `serialVersionUID` 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

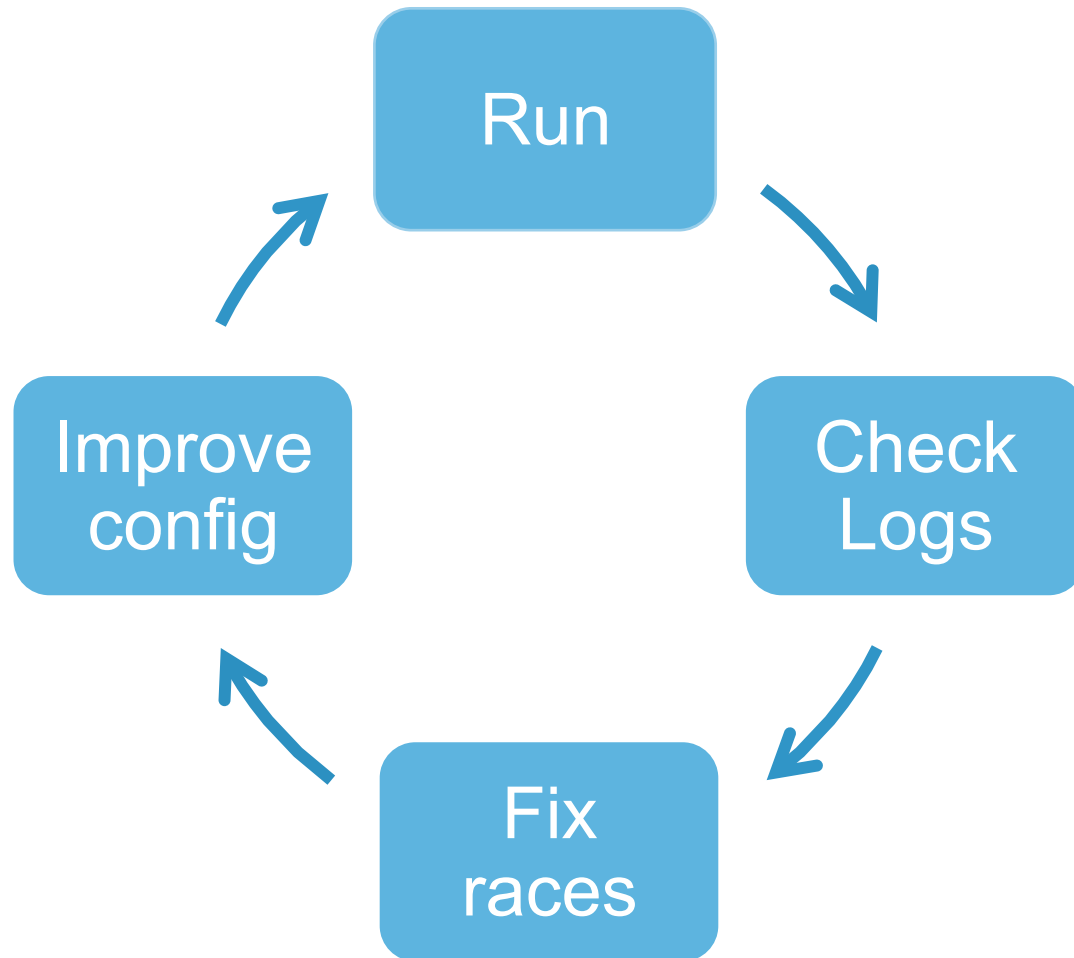
Class Loading Hell

- We generate classes on-the-fly: composite keys, etc.
- Bootstrap classloader should know how to load them
 - instrument loadClass() method
- Other classloaders should know too
 - not obliged to delegate to parent classloader
 - instrument loadClass() method 😊

Synchronization contracts: interfaces

- Usually interface contract is described (ConcurrentMap)
- Instrumentation stage:
 - specified interface might even not been loaded yet
 - get “potential contracts” from method name & signature
 - code path for each potential contract
- Runtime:
 - is potential contract real?
 - caching: {class, contract_id} → boolean

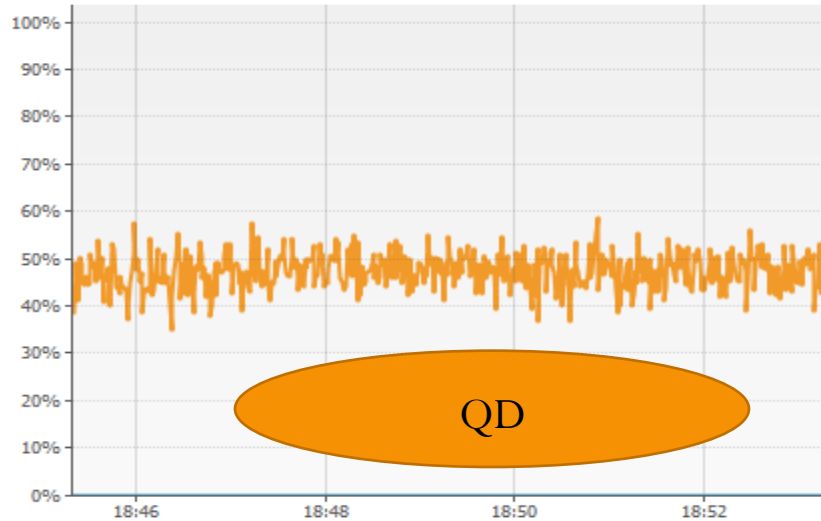
DRD in Real Life



DRD in Real Life: QD

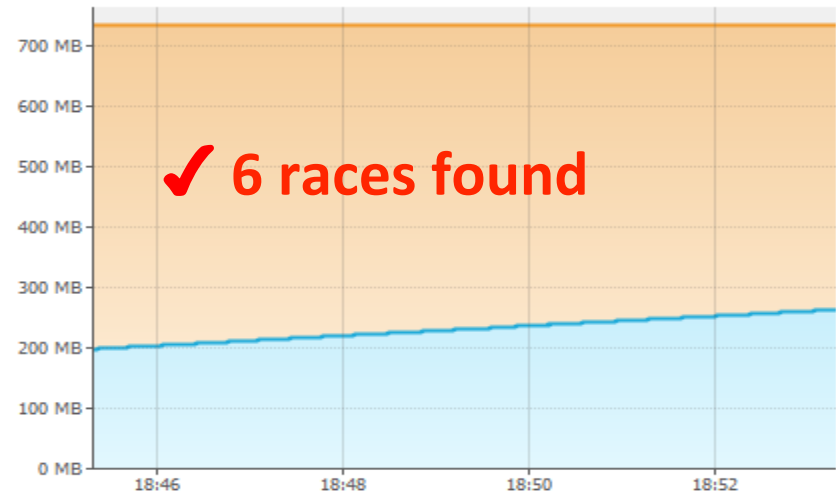
CPU usage: 42,5%

GC activity: 0,0%



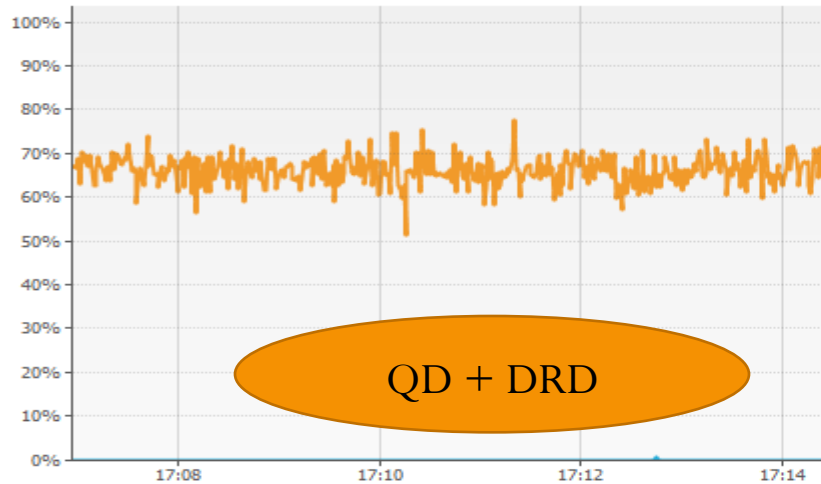
Size: 771 358 720 B
Max: 2 147 483 648 B

Used: 278 090 512 B



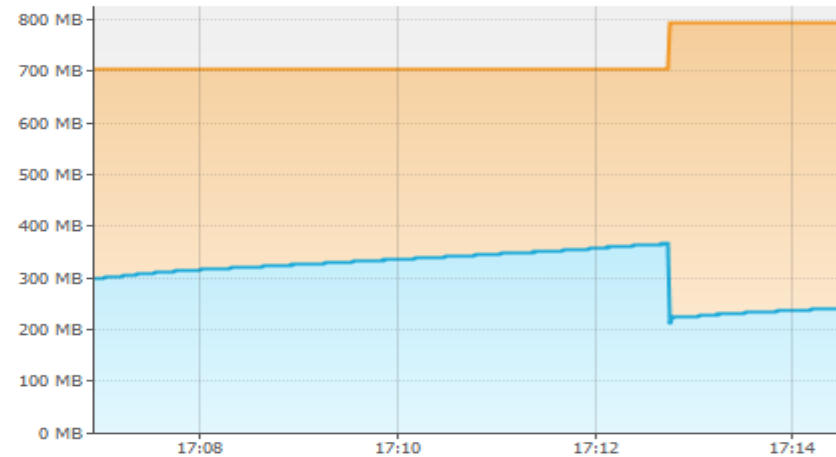
CPU usage: 67,0%

GC activity: 0,0%



Size: 834 142 208 B
Max: 2 147 483 648 B

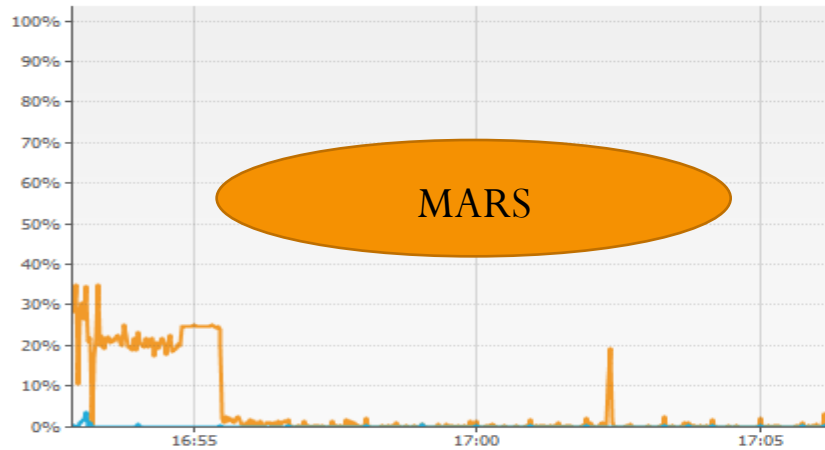
Used: 256 373 536 B



DRD in Real Life: MARS UI

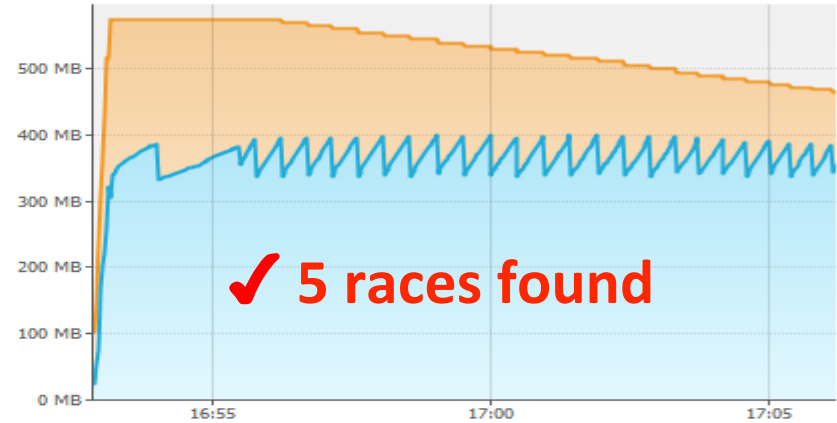
CPU usage: 0,3%

GC activity: 0,0%



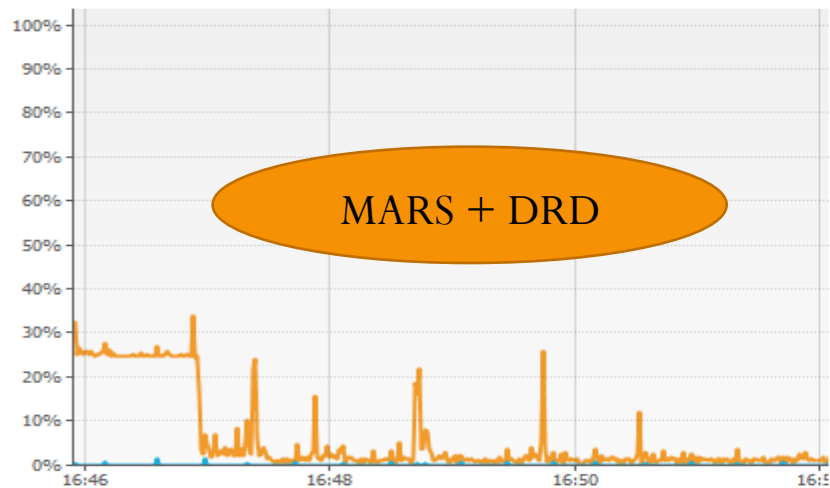
Size: 489 881 600 B
Max: 2 147 483 648 B

Used: 374 199 456 B



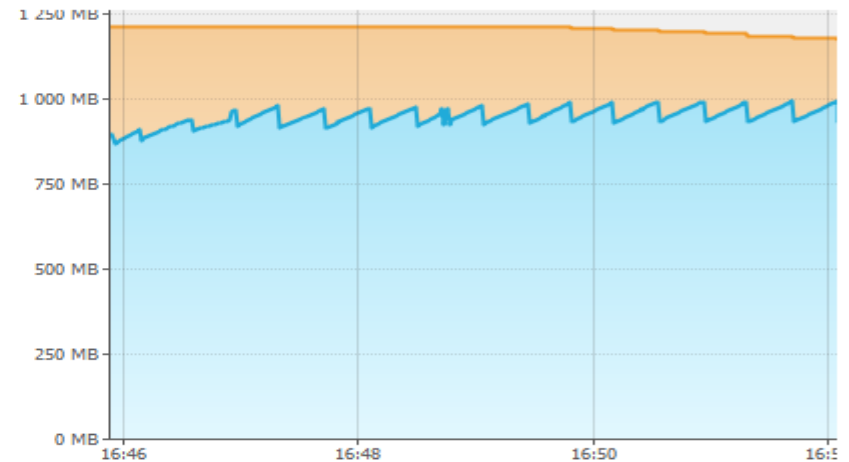
CPU usage: 1,9%

GC activity: 0,6%



Size: 1 236 795 392 B
Max: 2 147 483 648 B

Used: 981 046 616 B



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



Perspectives

- Verify declared intentions
 - “variable X is protected by lock L”
- Static analysis
- Integrate with tools for multithreaded unit-tests
- Annotations

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

