DEVEXPERTS

Upscale Financial IT

All-Round-the-Globe

Dynamic data race detection in concurrent Java programs

DEVEXPERTS

Vitaly Trifanov

trifanov@devexperts.com

Dmitry Tsitelov

cit@devexpert.com

Life story

The app is developed …

tested …

load tested …

delivered

Life story

Everything works fine for a couple of weeks …

and then …

strange exception, impossible data,

lightning from the skies

(add your favorite)

Life story

For two weeks everyone seeks for a problem …

customer in a rage …

then some hero finally finds the offender …

the missing volatile on a field

Got Races

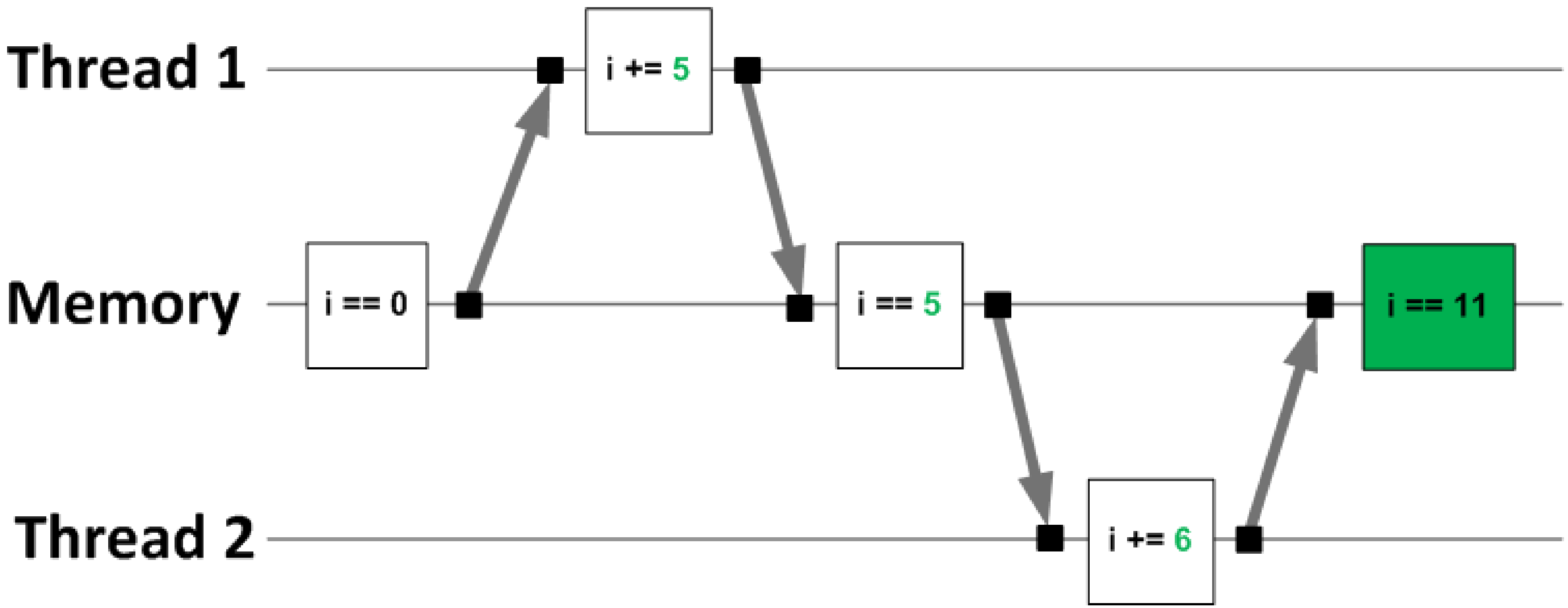
Data Race Example

Data Race Example

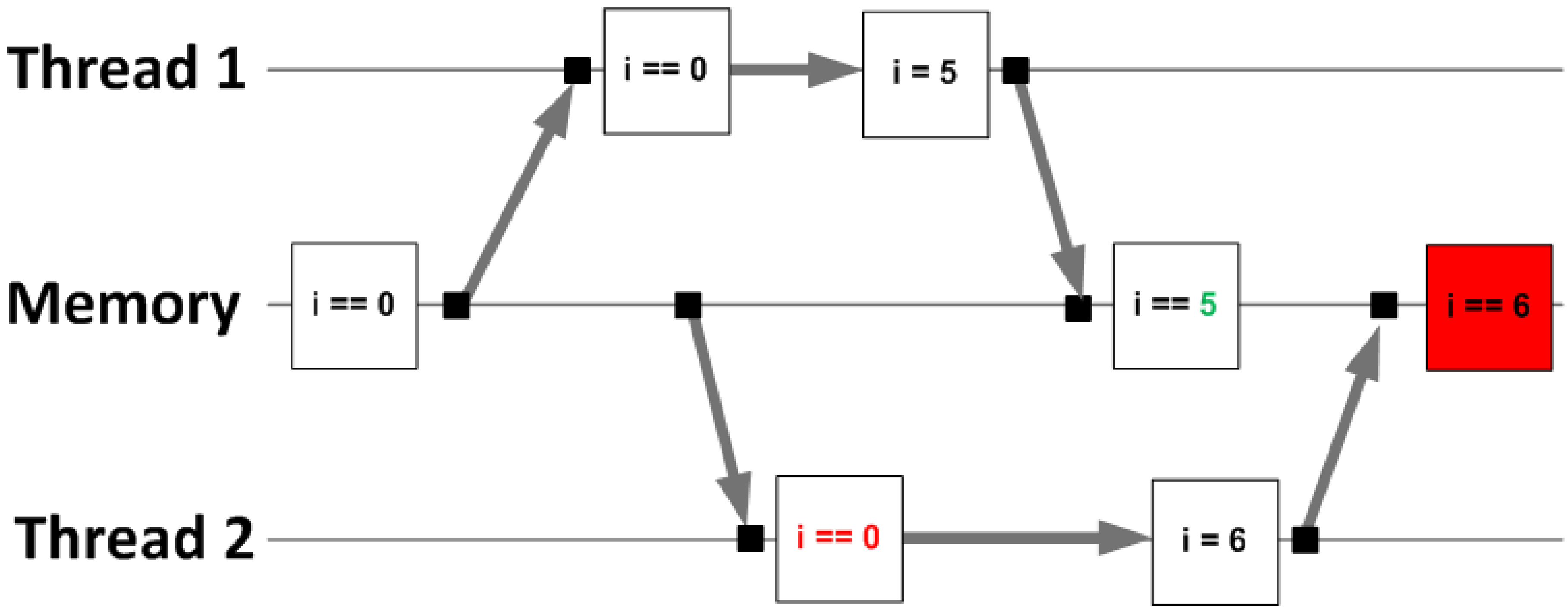
```
public class Account {
         private int amount = 0;
         public void deposit(int x) {amount += x;}
         public int getAmount() {return amount;}
}

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

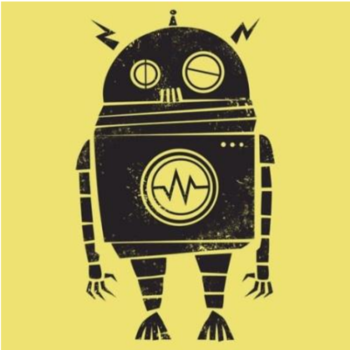
Automatic Race Detection

- **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

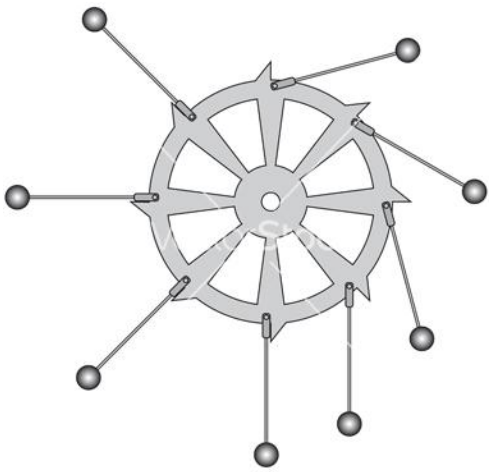
Static Approach

- **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

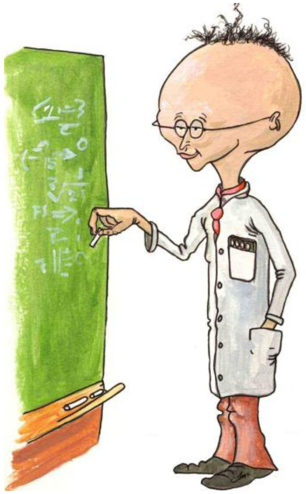
- **Pros**
  - Complete information about program flow
  - Lower level of false alarms
- **Cons**
  - Analyzes only current execution
  - Very large overhead
- **No existing stable dynamic detectors for Java**



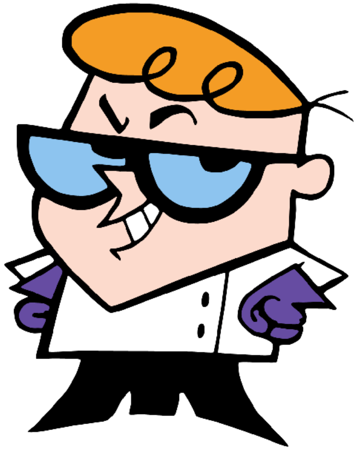
Static vs Dynamic: What To Do?

- 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!



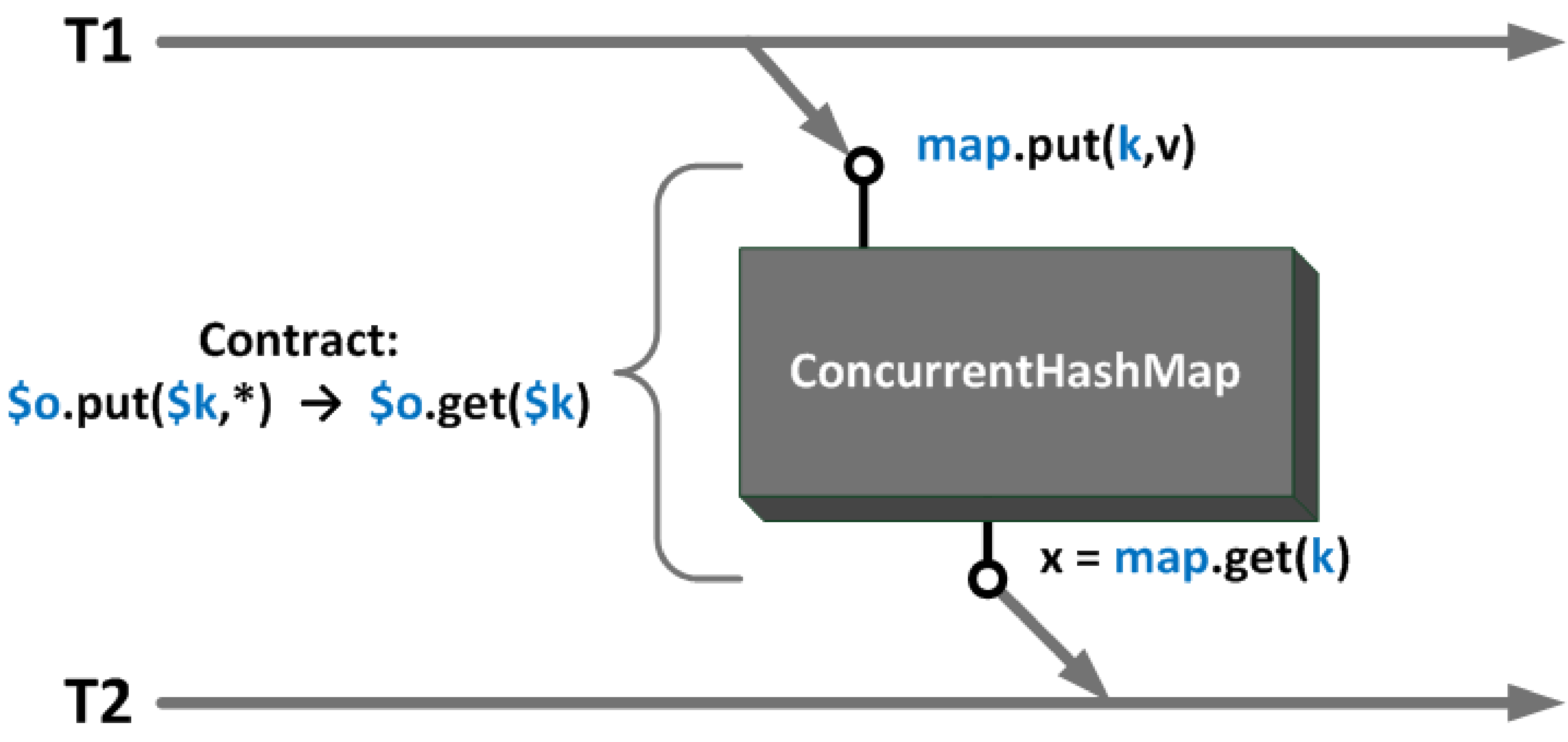
Requirements for Perfect Detector

- Dynamic
- Fast
- Precise
- Scalable

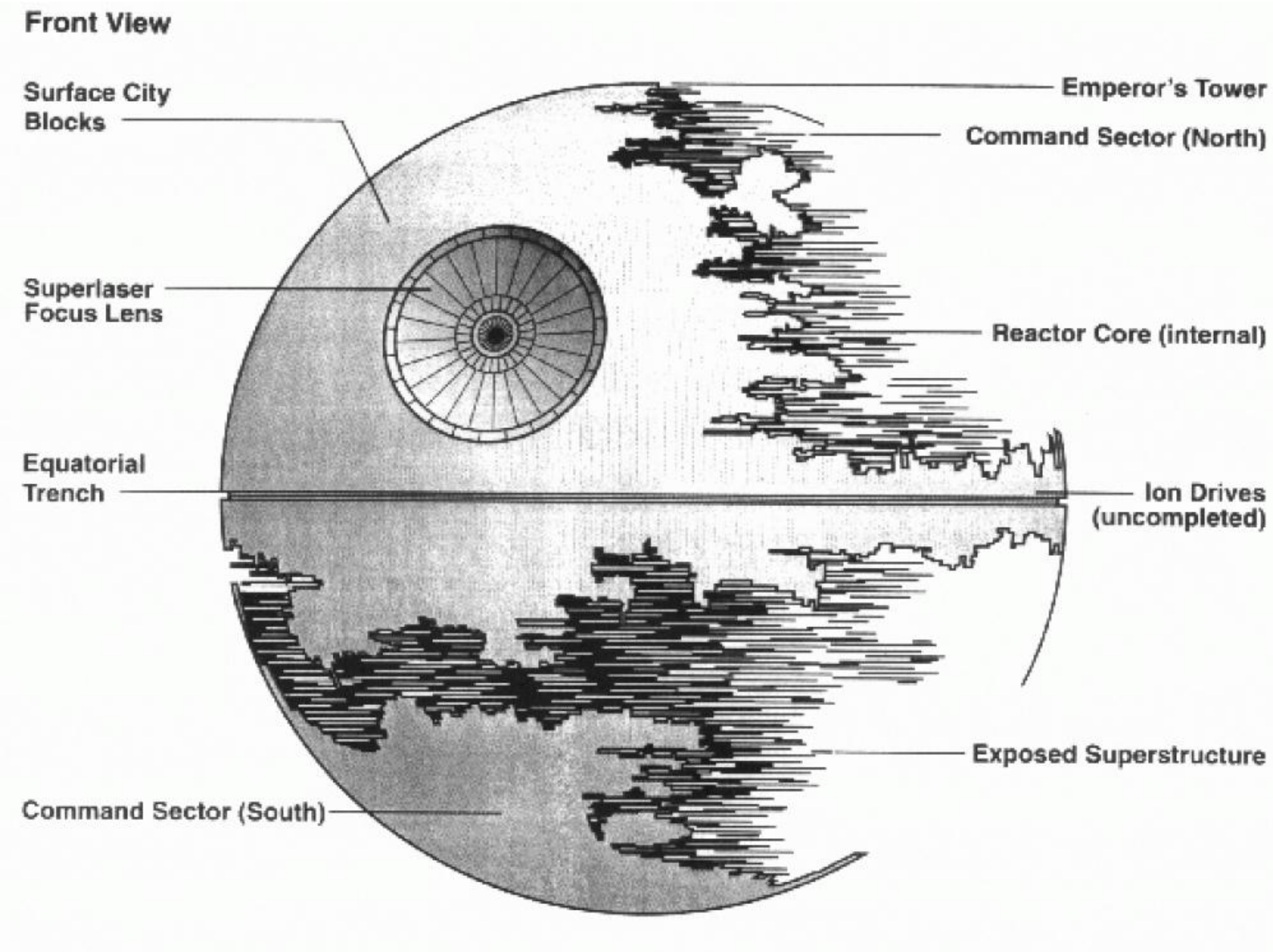
Scalability Concept

- **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**

Synchronization Contract Example



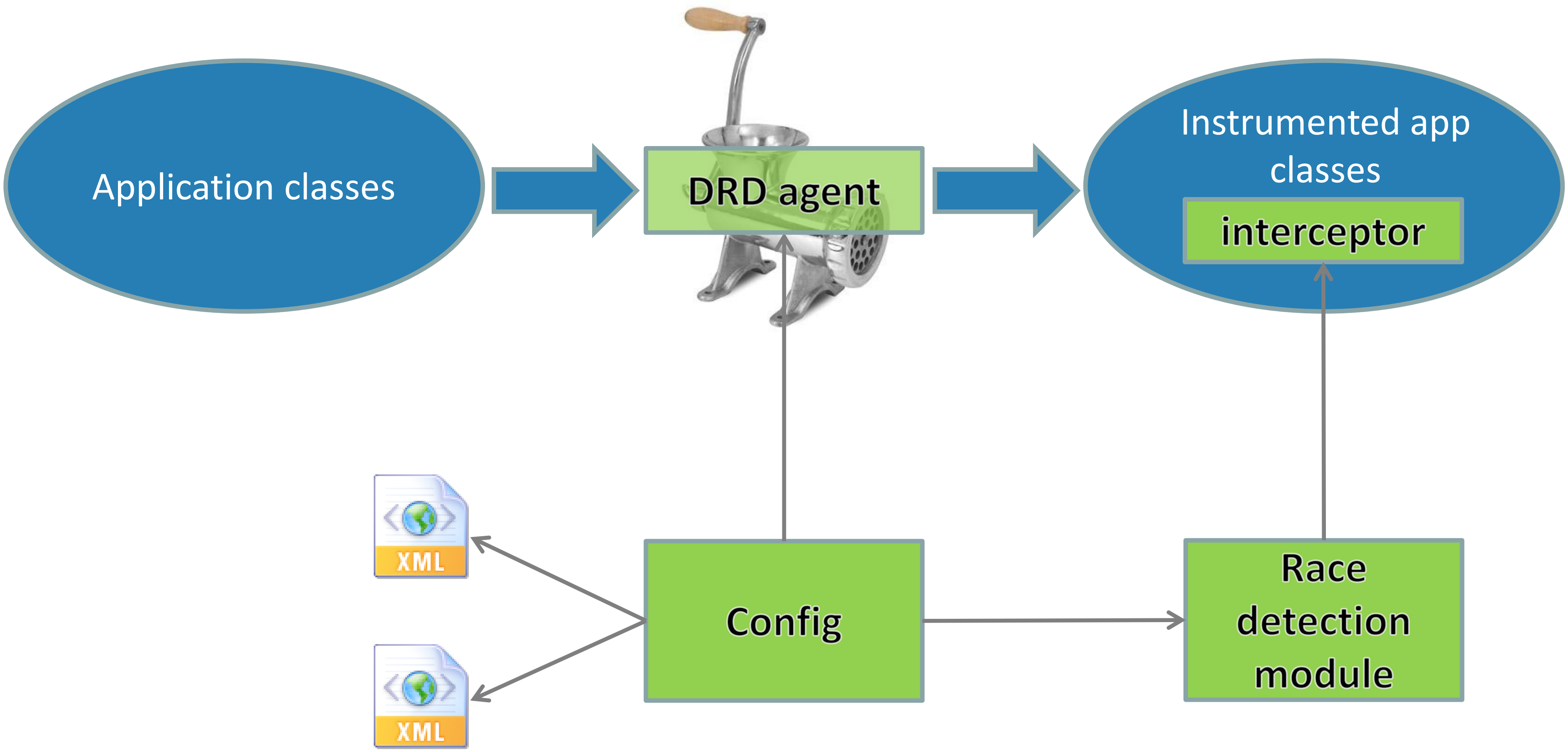
DRD: How It’s Organized



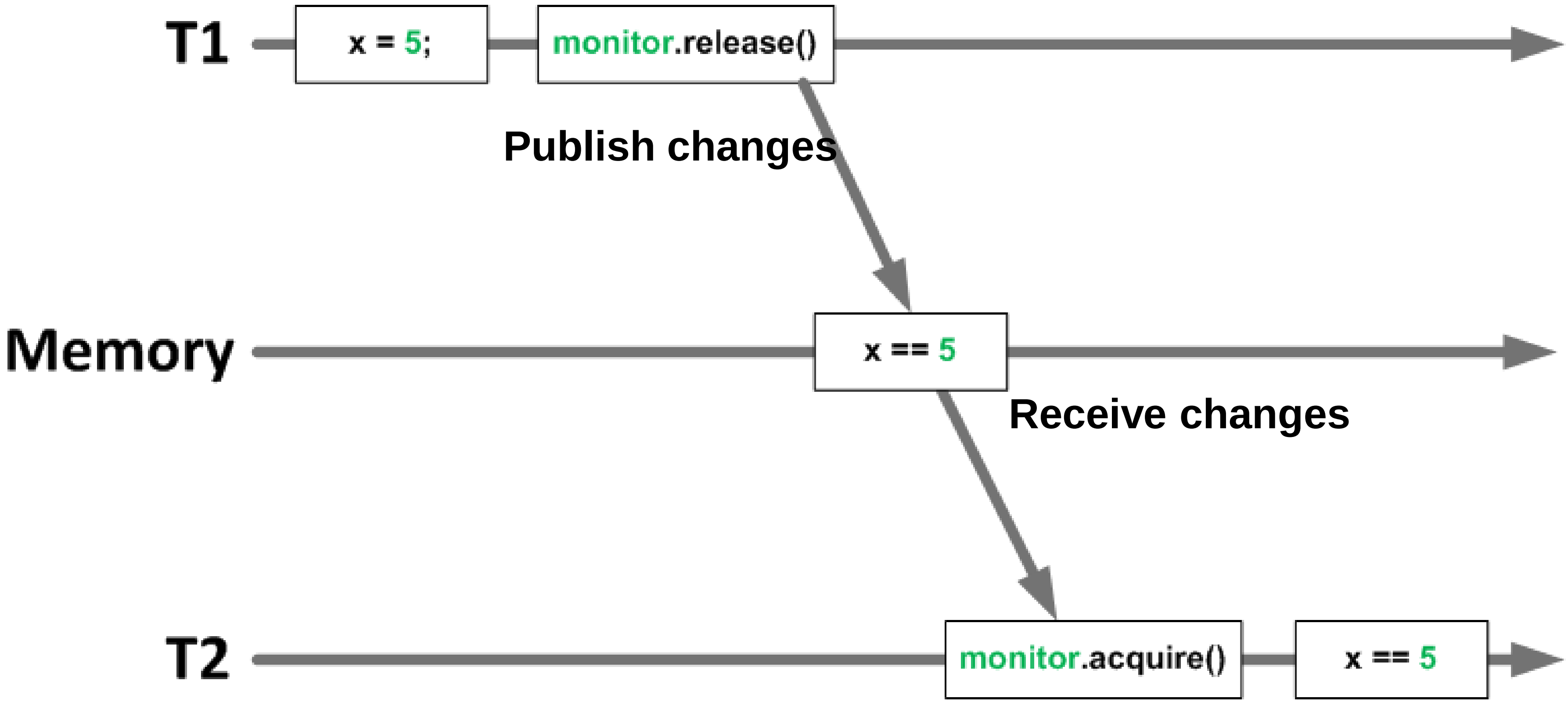
What Operations to Intercept?

- **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
XML
XML
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);
}
```

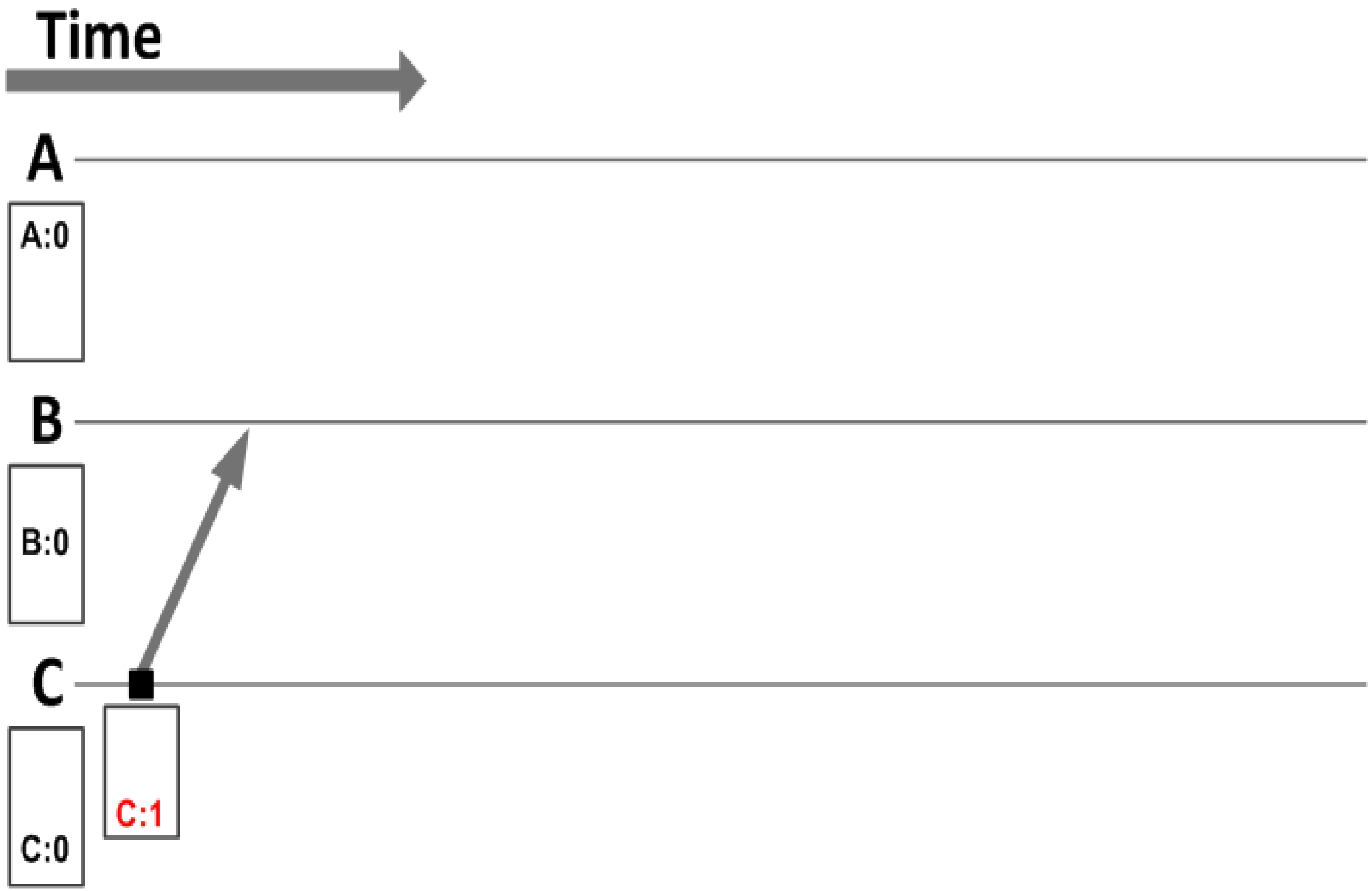
happens-before

Thread 2

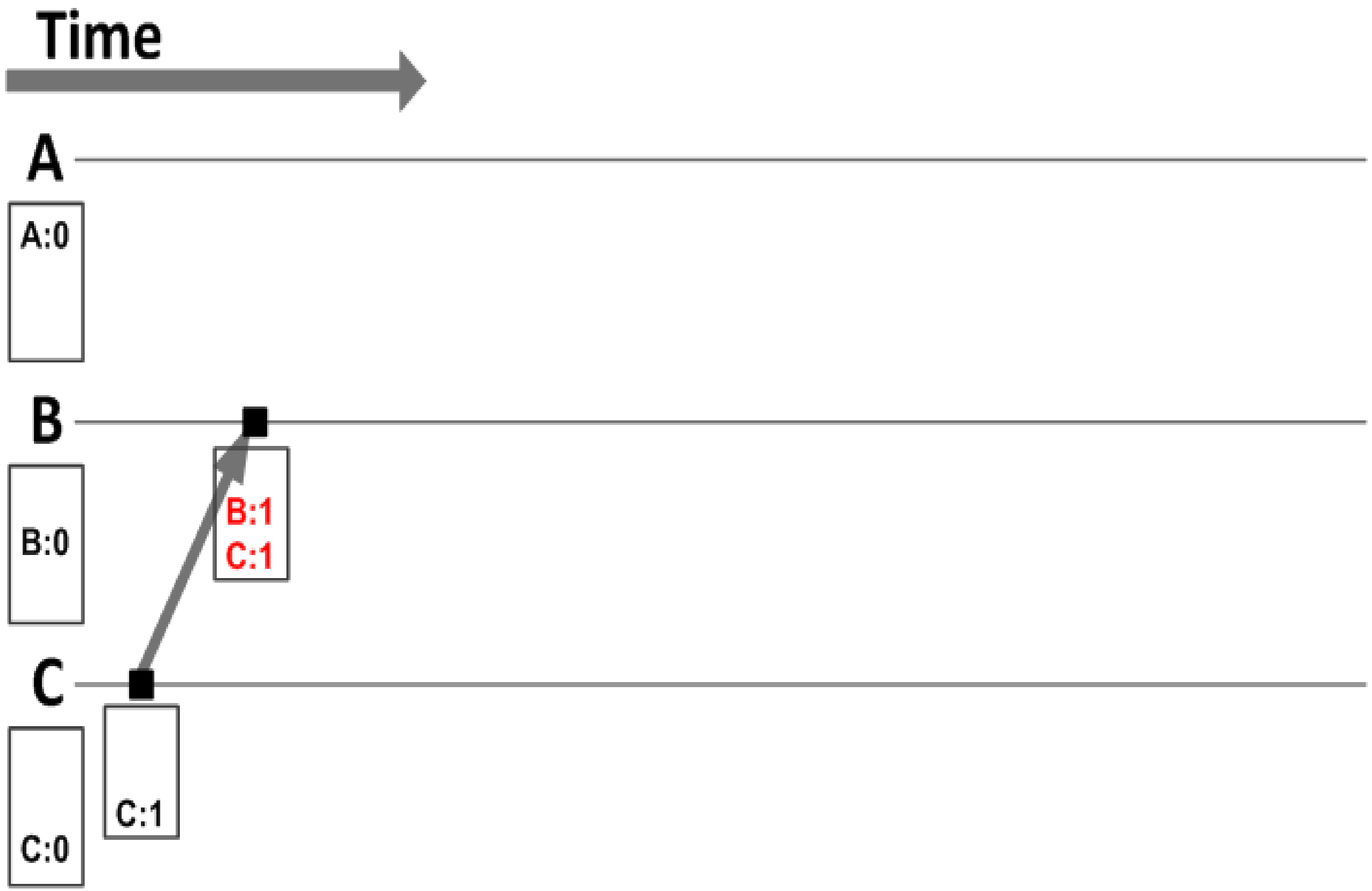
```
synchronized(lock) {
    account.deposit(7);
}
```

Vector Clock

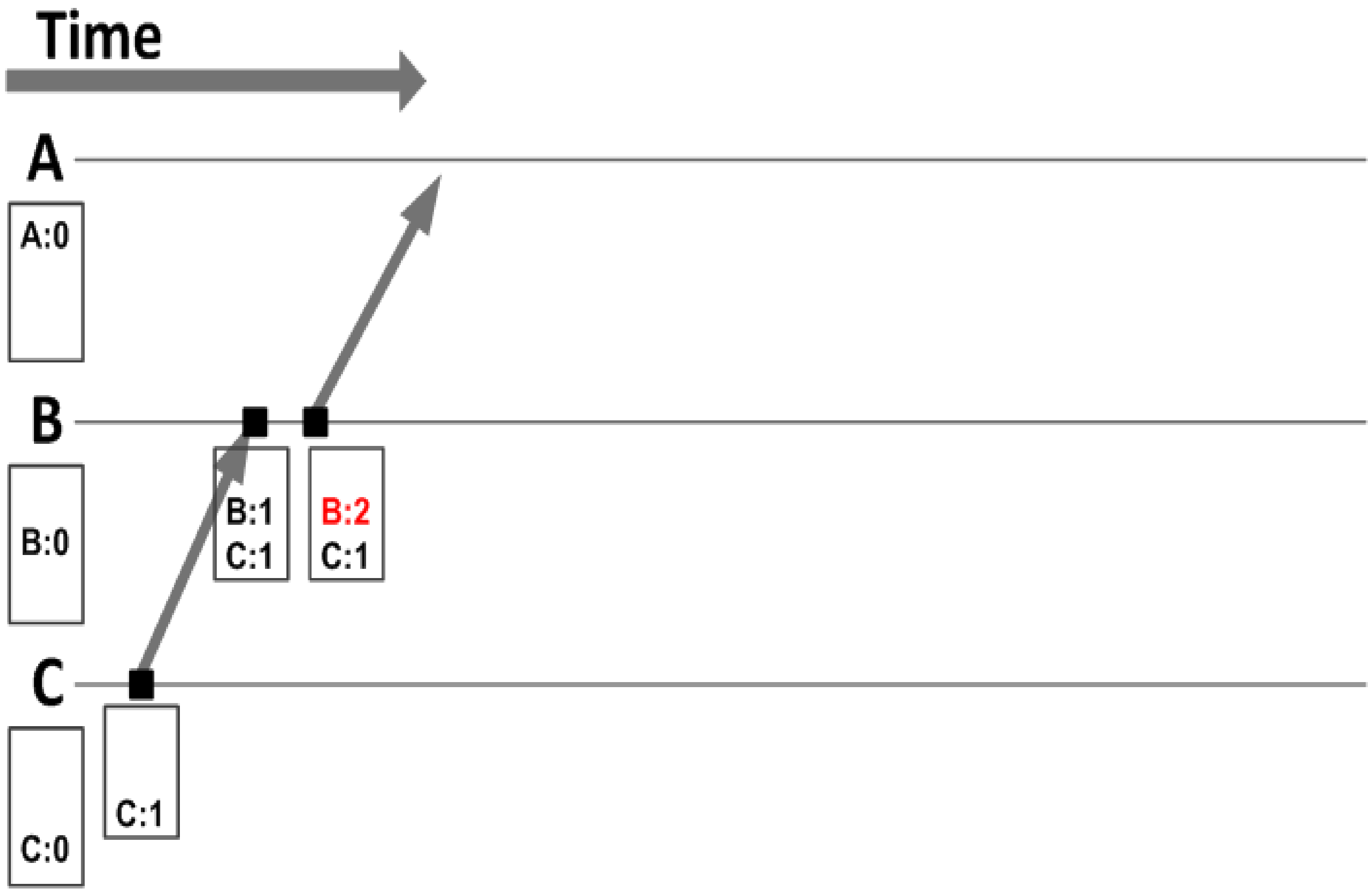
Vector Clock



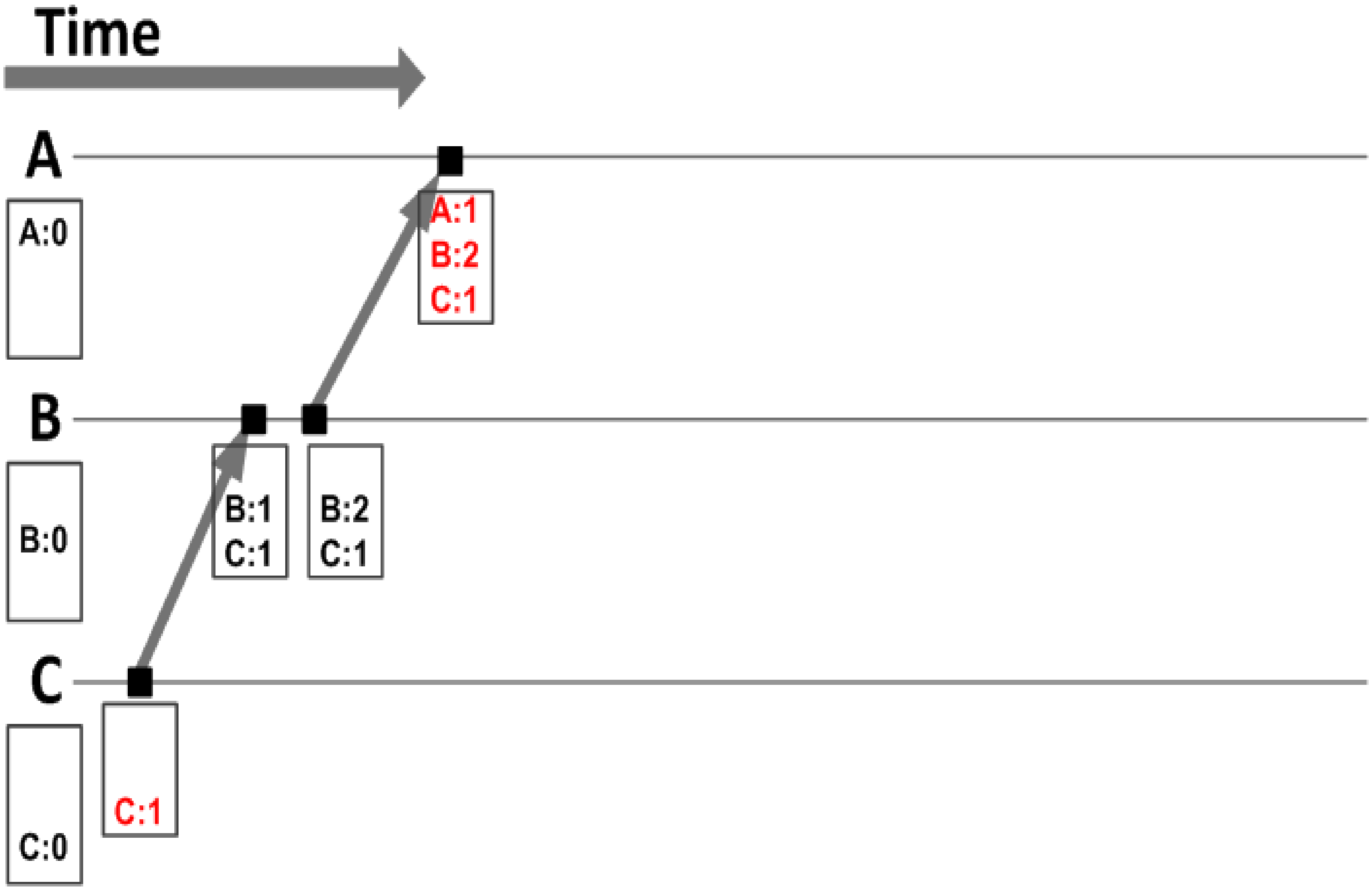
Vector Clock



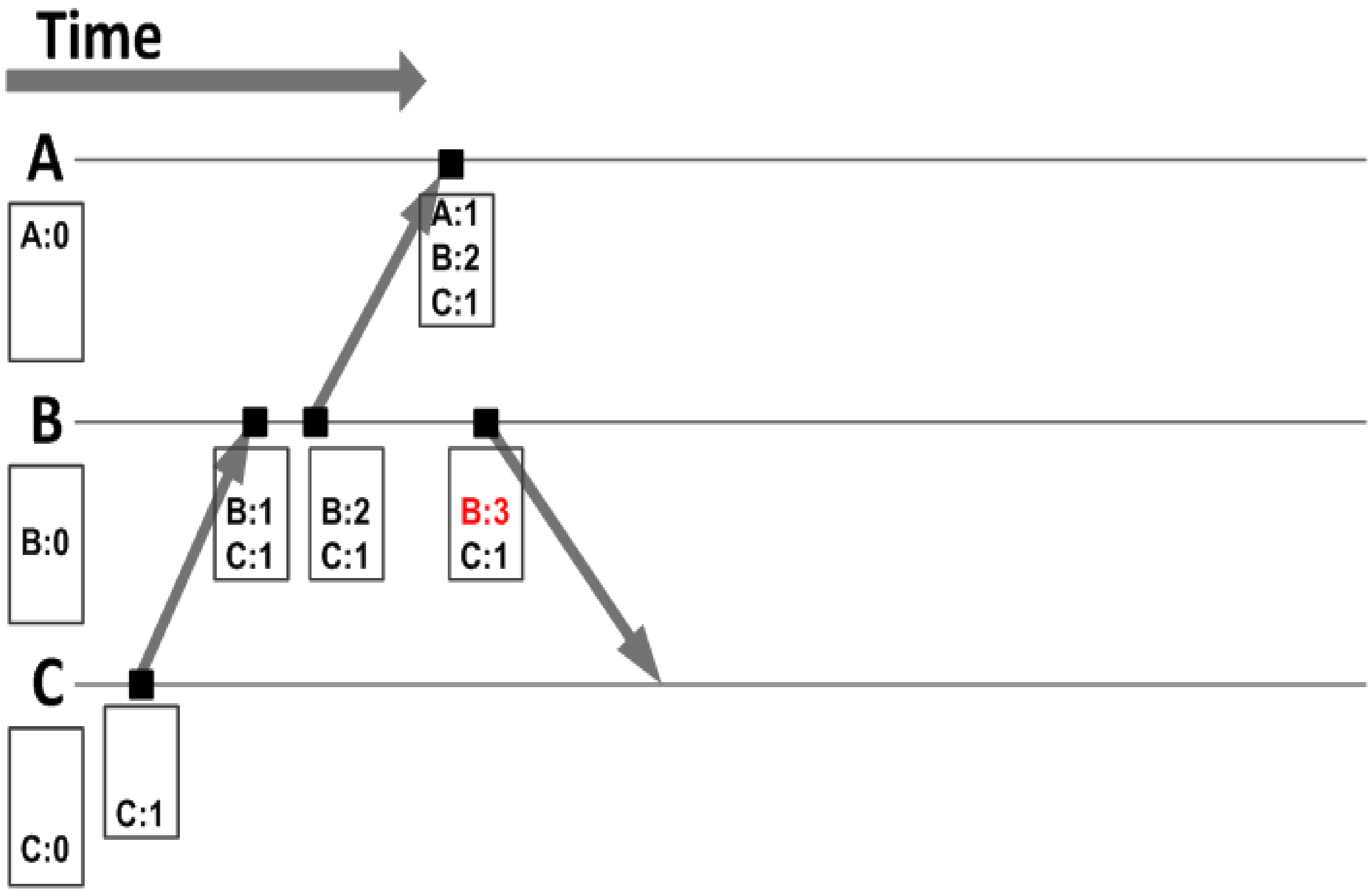
Vector Clock



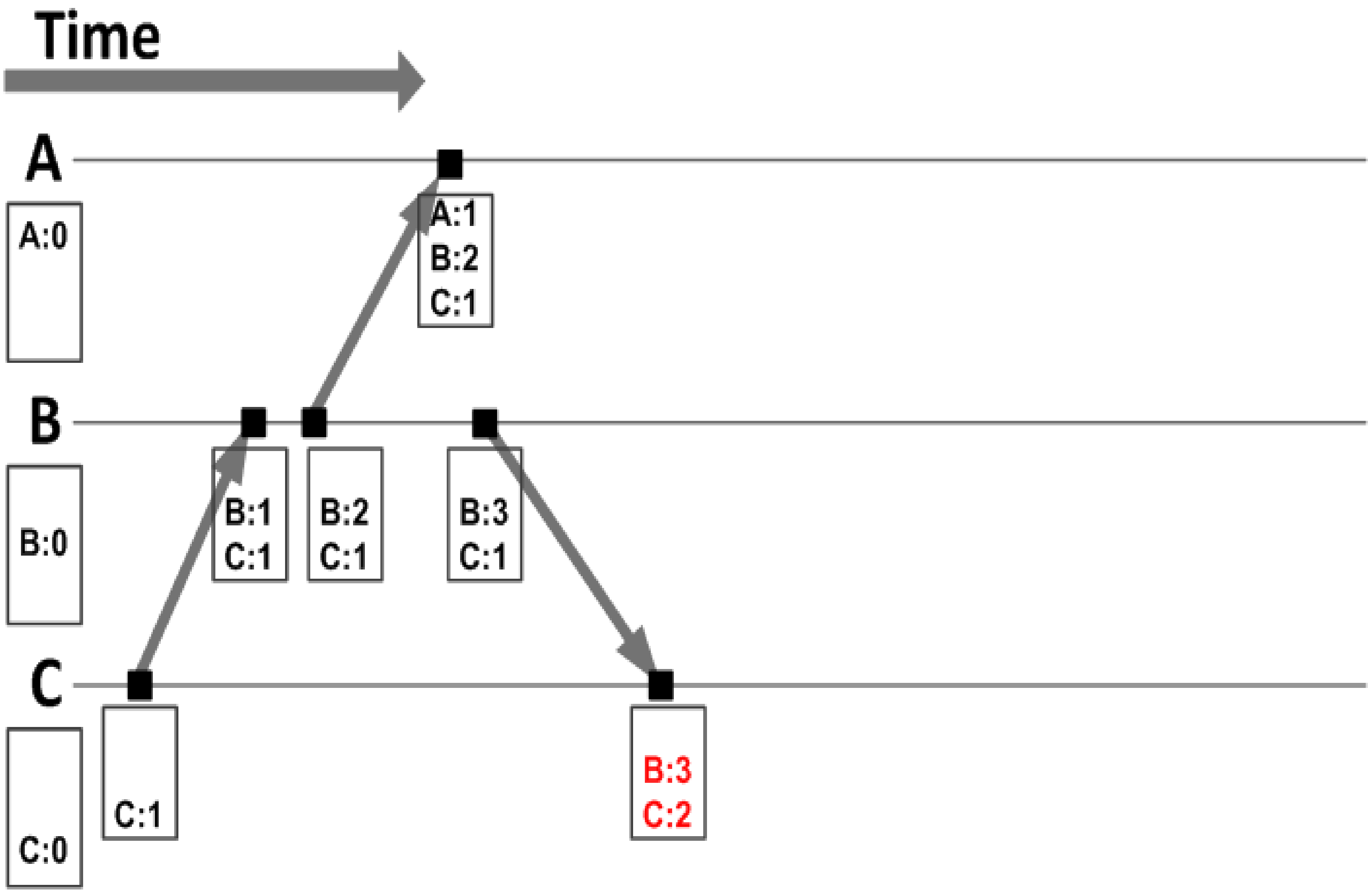
Vector Clock



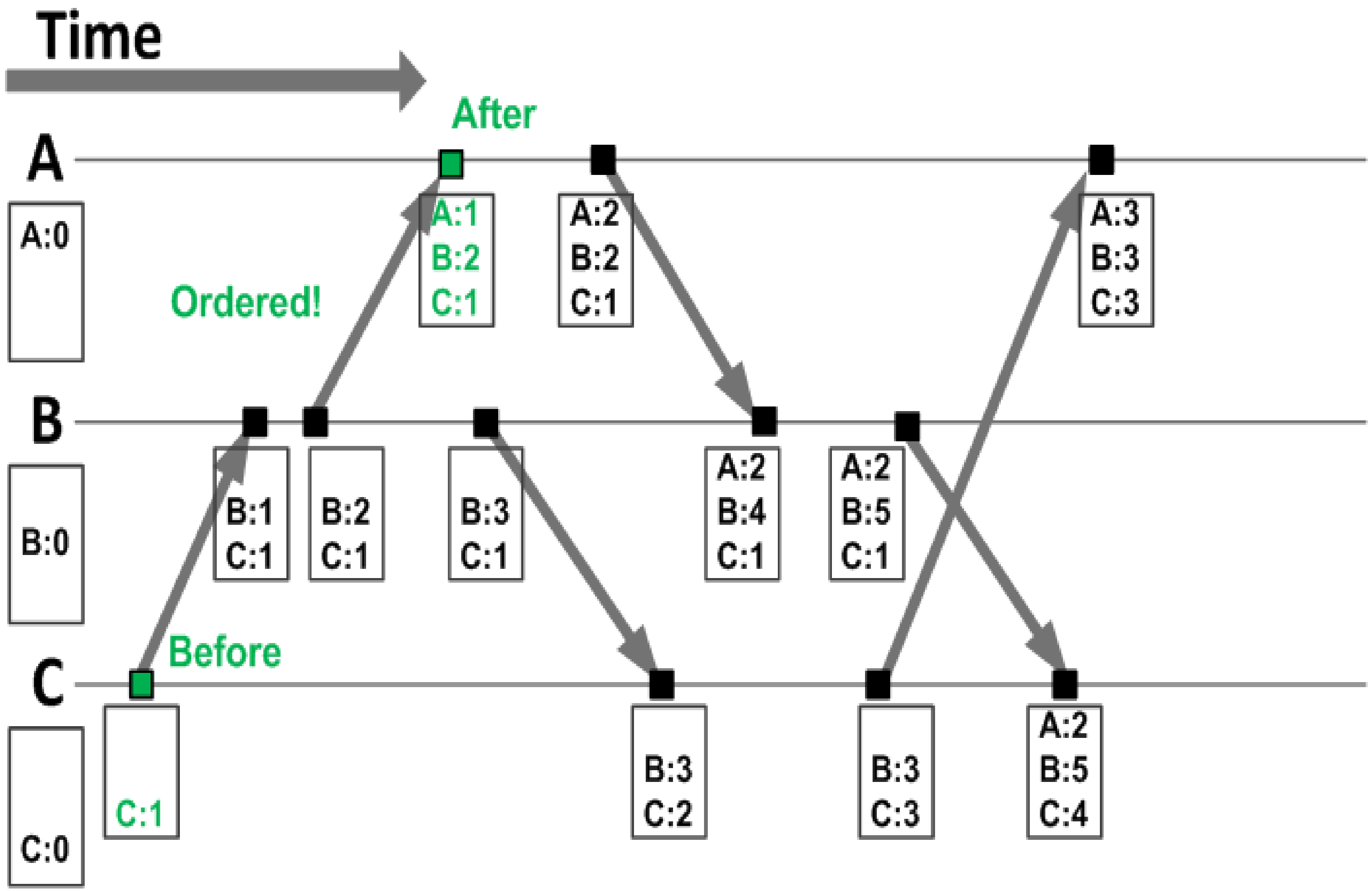
Vector Clock



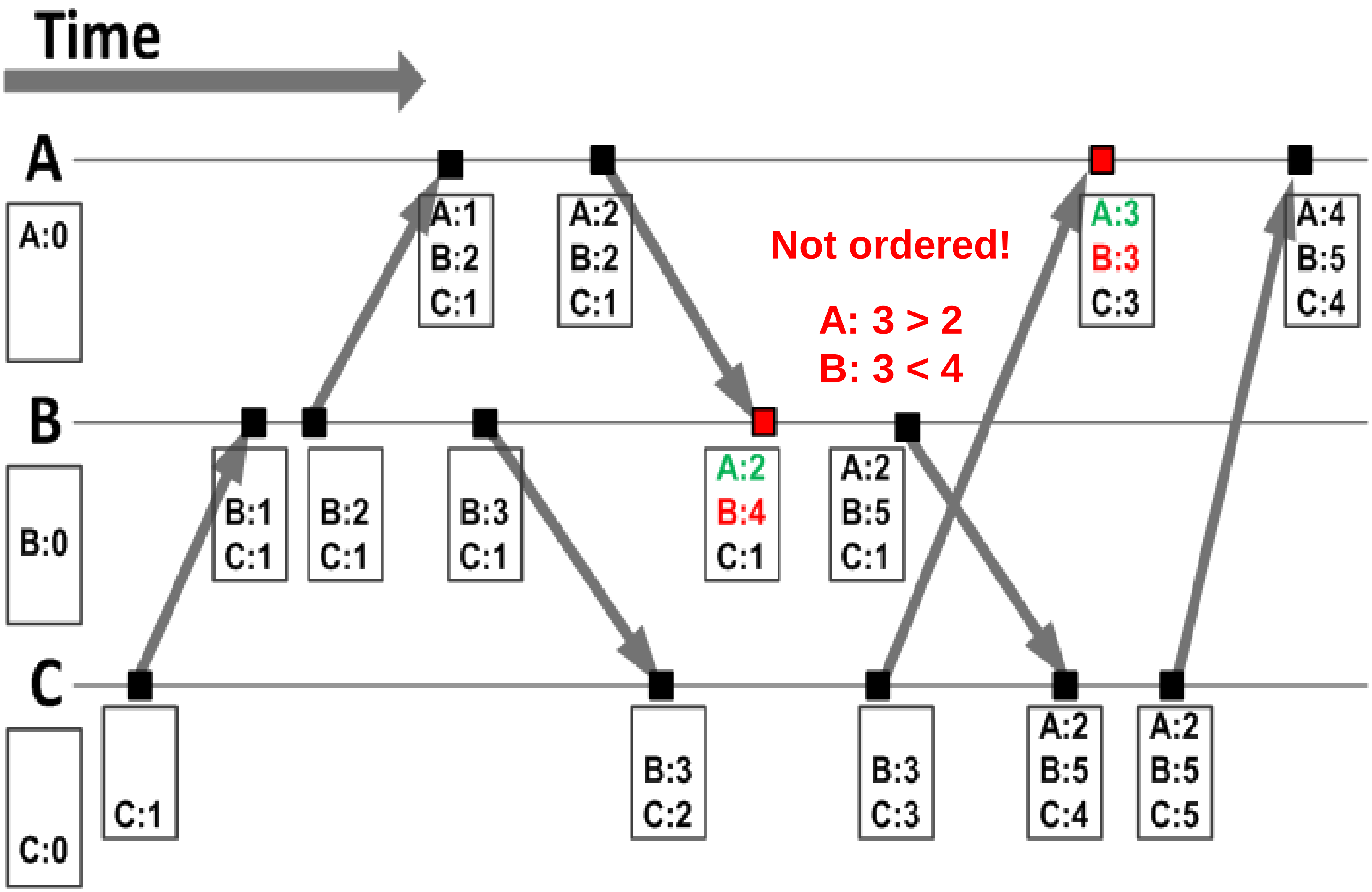
Vector Clock



Vector Clock



Vector Clock



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]

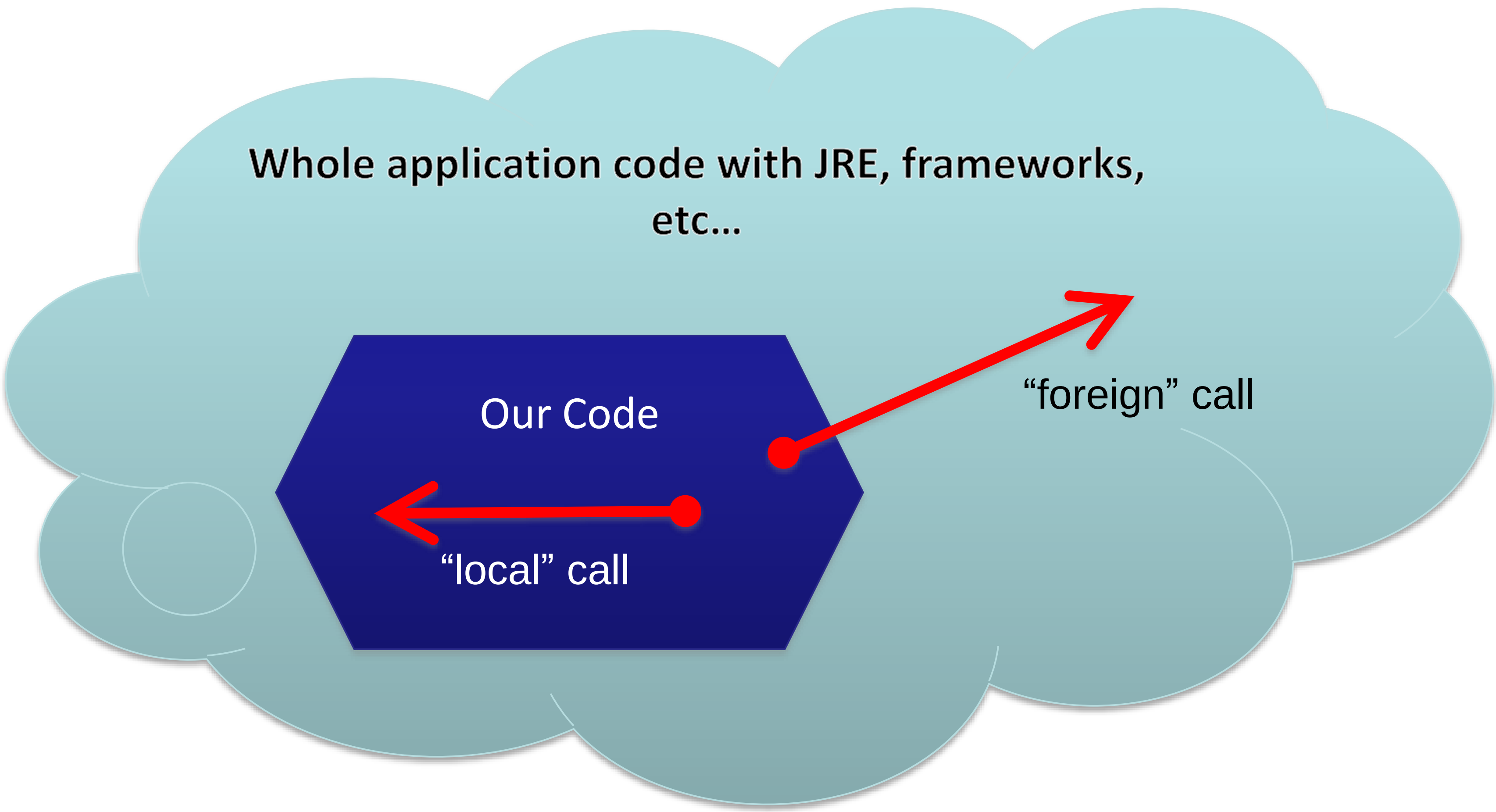
```
synchronized(lock) {
  X=1; //X.VC.load(T₁.VC): [5,10]
  //T₁.VC.tick(): [6,10]
  //lock.VC.load(T₁.VC): [6,10]
}
```

```
//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

Detection Scope



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

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

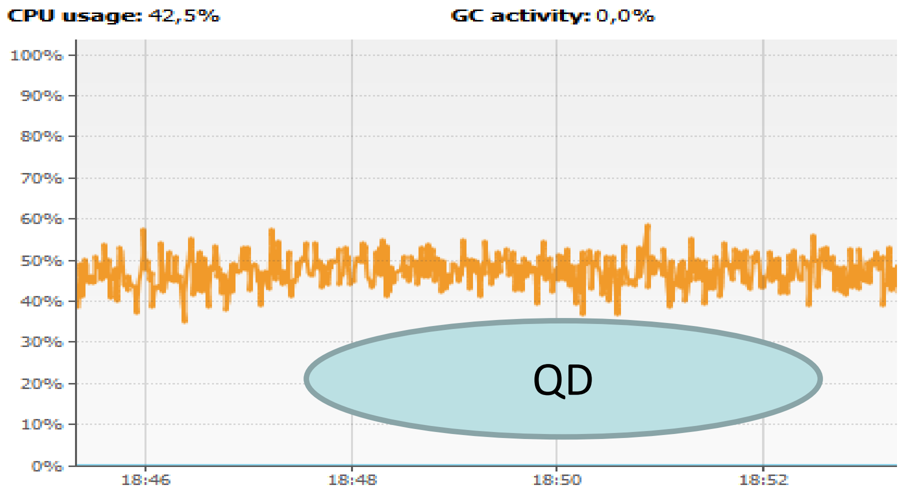
Solved Problems

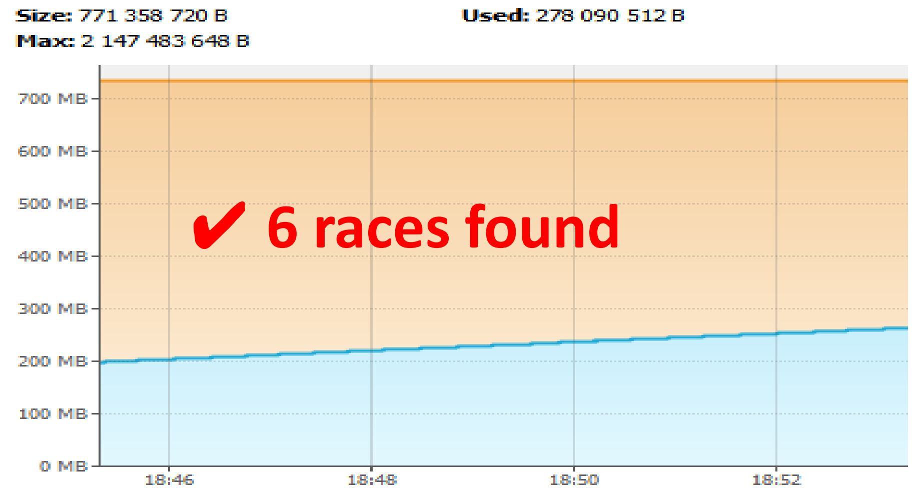
- **Composite keys for contracts and volatiles**
  - Generate them on-the-fly
- **Avoid unnecessary keys creation**
  - ThreadLocal<MutableKeyXXX> for each CompositeKeyXXX
- **Loading of classes, generated on-the-fly**
  - Instrument ClassLoader.loadClass()

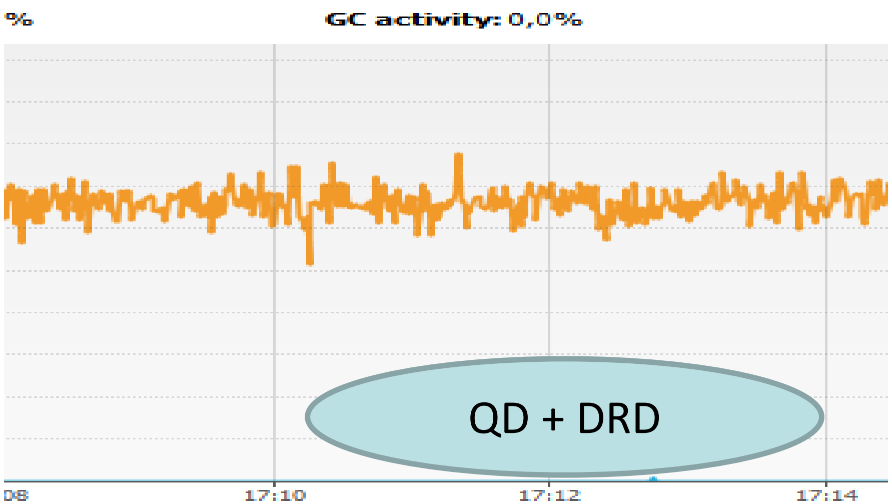
Solved Problems

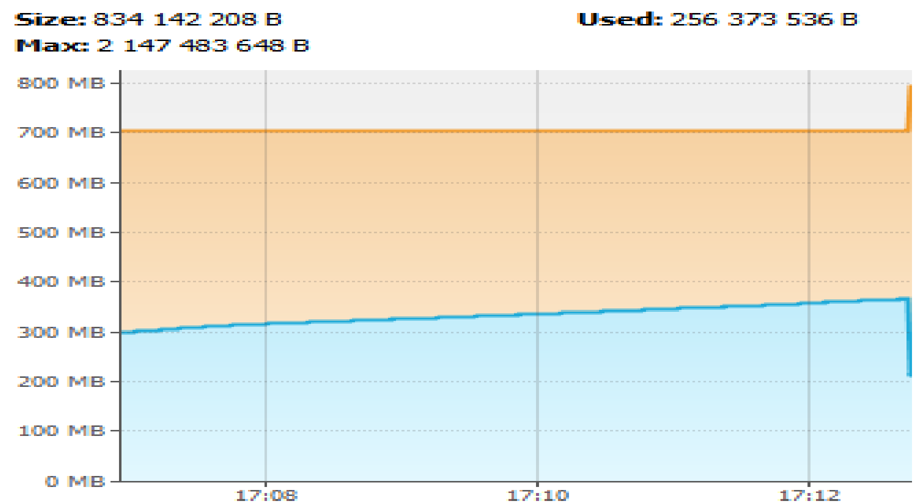
- **Doesn’t break serialization**
  - compute serialVersiodUid before instrumentation
- **Caching components of dead clocks**
  - when thread dies, its time frames doesn’t grow anymore
  - cache frames of dead threads to avoid memory leaks
  - local last-known generation & global generation

DRD in Real Life: QD

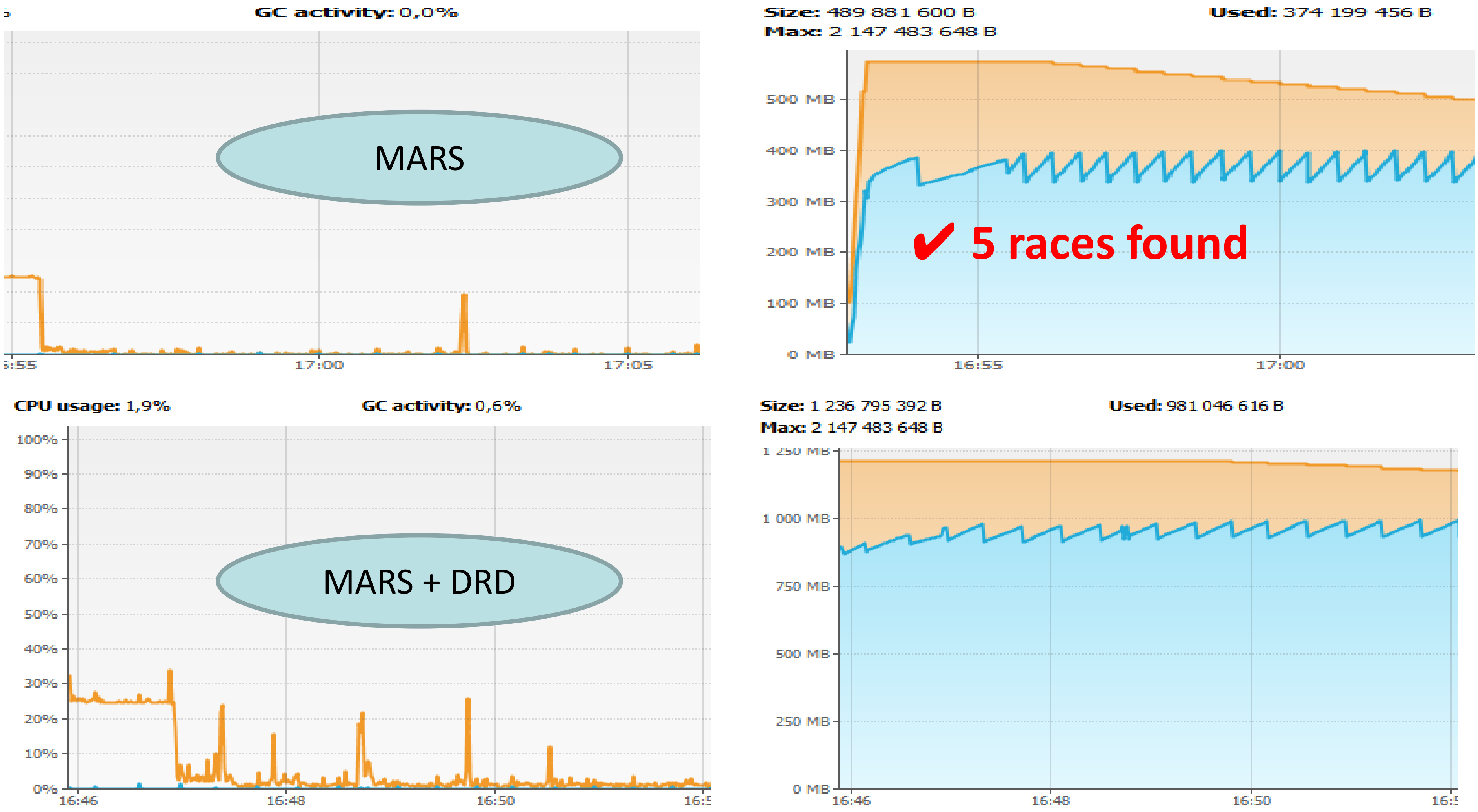








DRD in Real Life: MARS UI



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

Limitations: synchronization contracts

- **We support only simple explicit links and their combinations**
  - owner – owner
  - param – param
  - owner – param (partially)
- **We do not check return values of contract methods**
  - only true/false for CAS-like operations
- **We do not support implicit contracts**
  - Future<T> ExecutorService.submit(Callable<T> callable)
  - ConcurrentMap.entrySet().iterator()….

Future works

- **Research**
  - Synchronization contracts
  - Verify declared intentions ("X is protected by lock L")
  - Module testing
- **Development**
  - Post-mortem mode
  - Integrate with tools for multithreaded unit-tests
  - Annotations
  - Optimizations
- **Evaluation**

Links

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

- IBM MSDK
- ThreadSanitizer for Java
- jChord
- FindBugs

- JLS «Threads and locks» chapter

Try it!

- **It's free as in beer**
- **Any troubles, bugs, questions?**
  - feel free to contact us at drd-support@devexperts.com
- **Success story? Epic fail?**
  - Let us know. Any feedback will be appreciated and will help us to make DRD better for the common good.

Q & A

Thank you!