JFokus 2013 - Finding And Solving Java Deadlocks

Dr Heinz Kabutz

Javaspecialists.eu
java training

Heinz Kabutz

- **Brief Biography**
  - German from Cape Town, now lives in Chania
  - PhD Computer Science from University of Cape Town
  - The Java Specialists' Newsletter
  - Java programmer
  - Java Champion since 2005
- **Advanced Java Courses**
  - Concurrency Specialist Course
    - Offered in Stockholm 19-22 March 2013
  - Java Specialist Master Course
  - Design Patterns Course
  - http://www.javaspecialists.eu

1: Introduction

Javaspecialists.eu
java training

Structure Of Hands-On Lab

- **Three short lectures, each followed by a short lab**
  - **http://www.javaspecialists.eu/outgoing/jfokus2013.zip**
- **We only have three hours to cover a lot, so let's go!**

Questions

- **Please please please please ask questions!**
- **Interrupt us at any time**
  - **This lab is on deadlocks, we need to keep focused in available time**
- **The only stupid questions are those you do not ask**
  - **Once you've asked them, they are not stupid anymore**
- **The more you ask, the more we all learn**

JFokus 2013 - Finding And Solving Java Deadlocks

Avoiding Liveness Hazards

10: Avoiding Liveness Hazards

- **Fixing safety problems can cause liveness problems**
  - **Don't indiscriminately sprinkle "synchronized" into your code**
- **Liveness hazards can happen through**
  - **Lock-ordering deadlocks**
    - **Typically when you lock two locks in different orders**
    - **Requires global analysis to make sure your order is consistent**
      - **Lesson: only ever hold a single lock per thread!**
  - **Resource deadlocks**
    - **This can happen with bounded queues or similar mechanisms meant to bound resource consumption**
- **A thread deadlocked in BLOCKED state can never recover**

Lab 1: Deadlock Resolution By Global Ordering

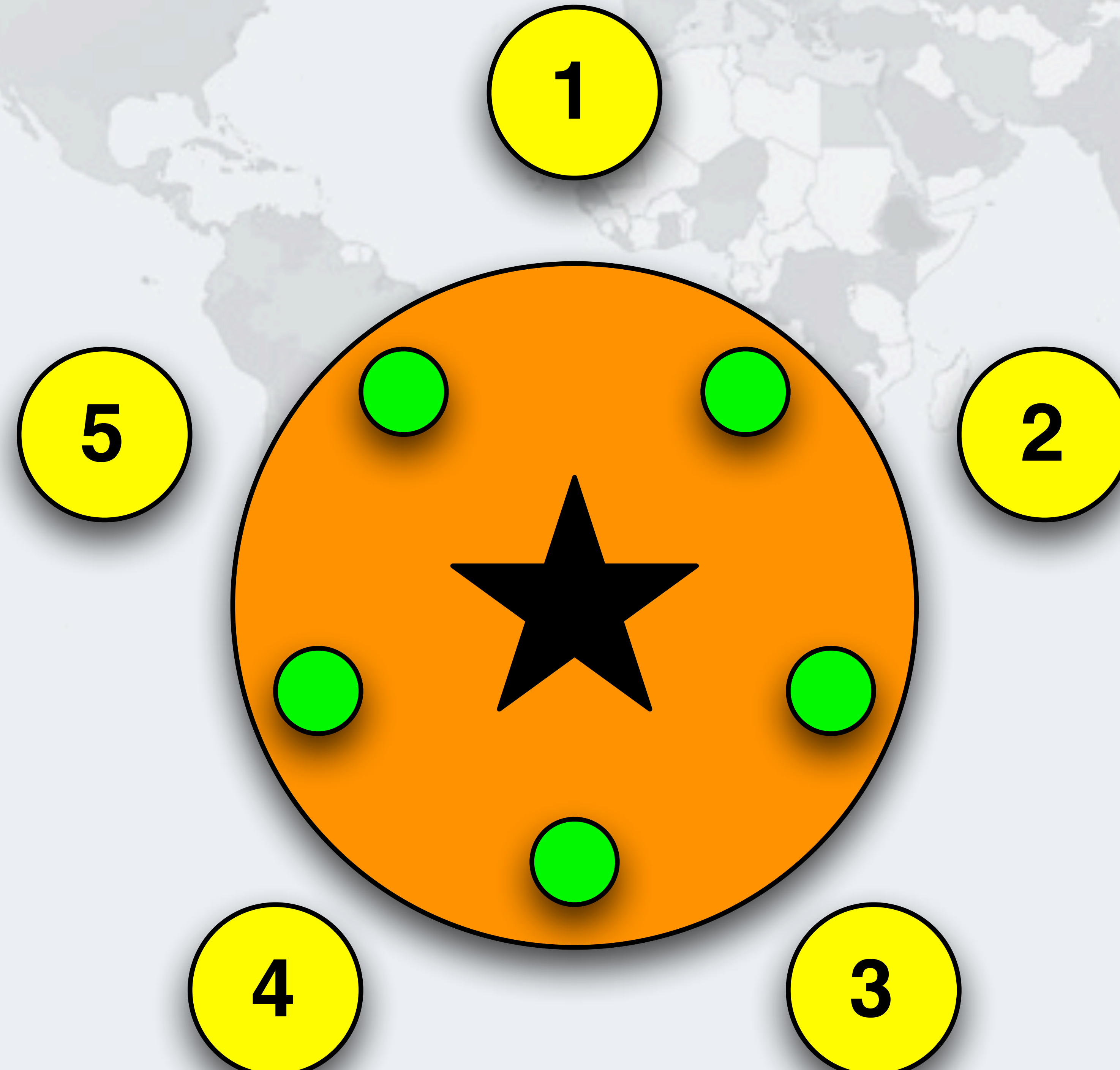
Avoiding Liveness Hazards

Javaspecialists.eu
java training

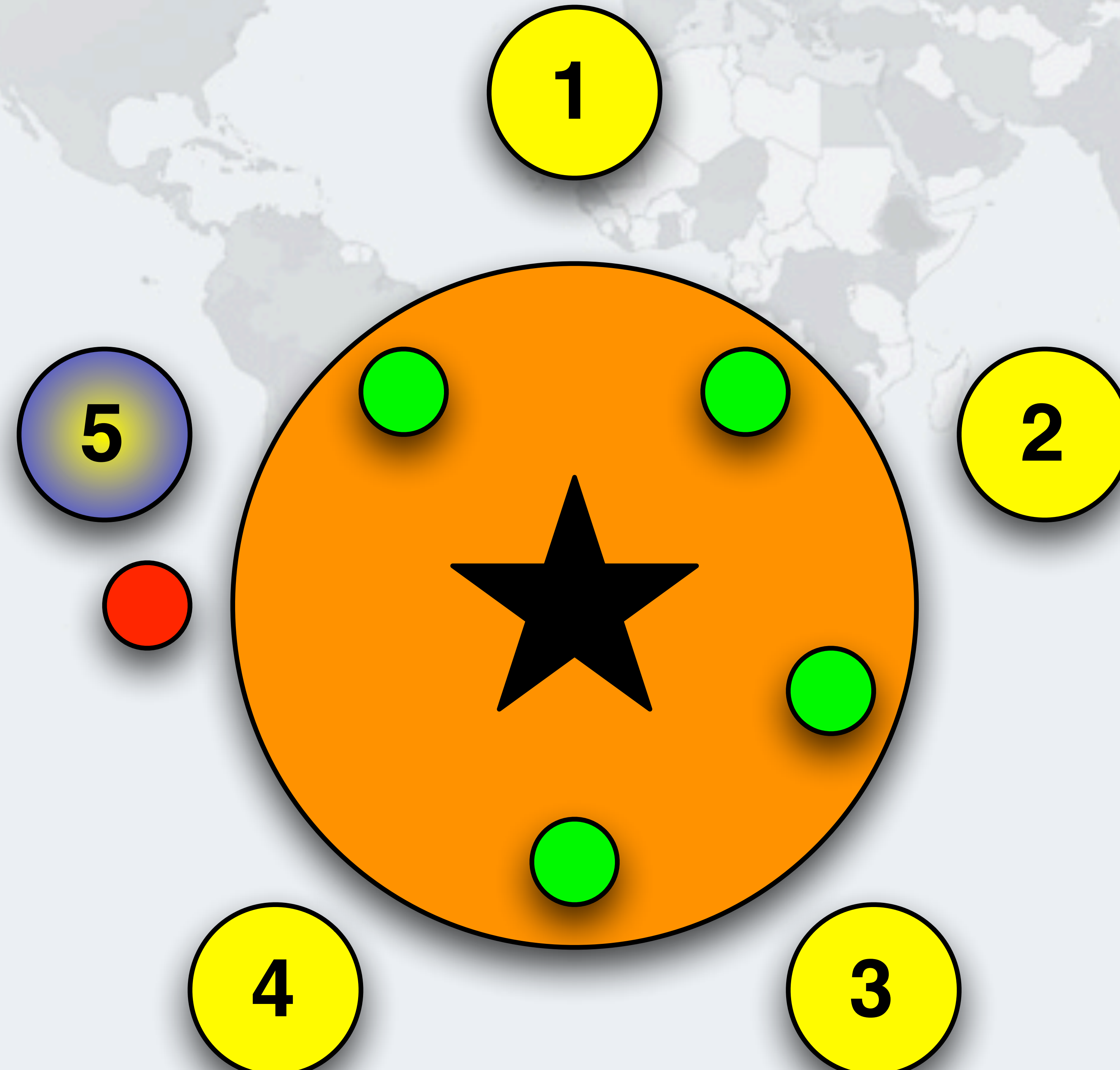
Lab 1: Deadlock Resolution By Global Ordering

- **Classic problem is that of the "dining philosophers"**
  - **We changed that to the "drinking philosophers"**
    - **That is where the word "symposium" comes from**
      - **sym - together, such as "symphony"**
      - **poto - drink**
    - **Ancient Greek philosophers used to get together to drink & think**
- **In our example, a philosopher needs two glasses to drink**
  - **First he takes the right one, then the left one**
  - **When he finishes drinking, he returns them and carries on thinking**

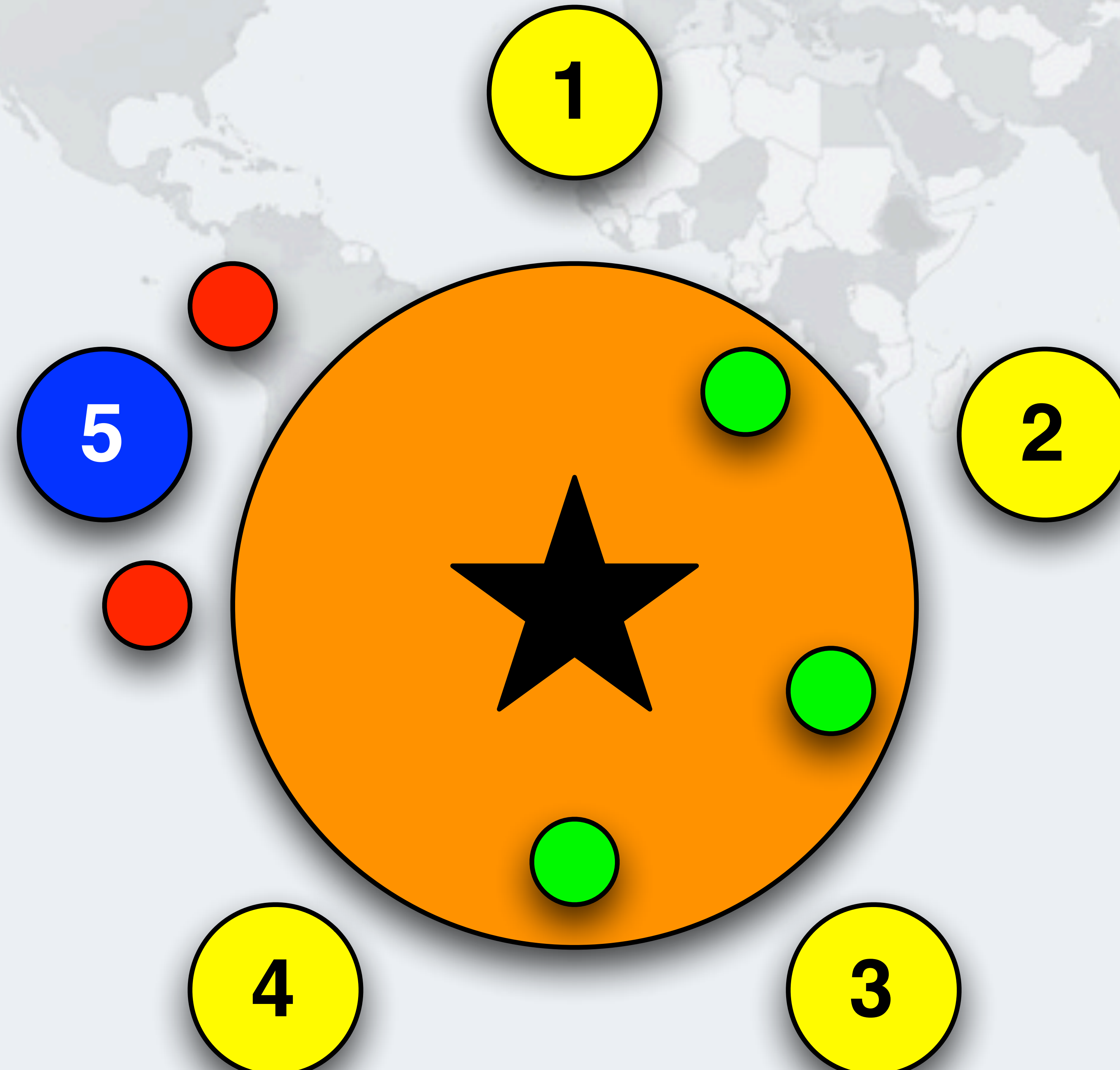
Table Is Ready, All Philosophers Are Thinking



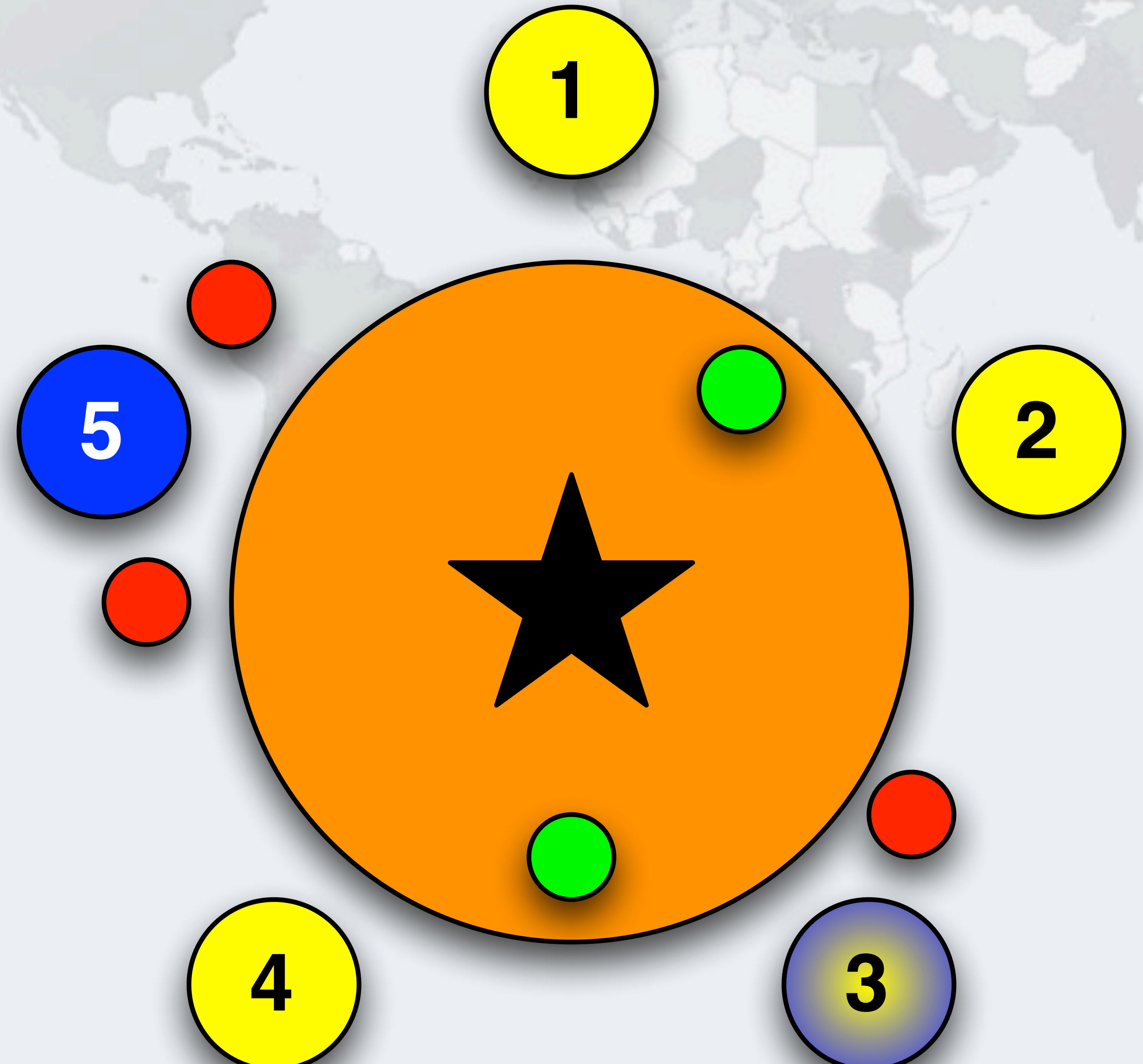
Philosophers 5 Wants To Drink, Takes Right Cup



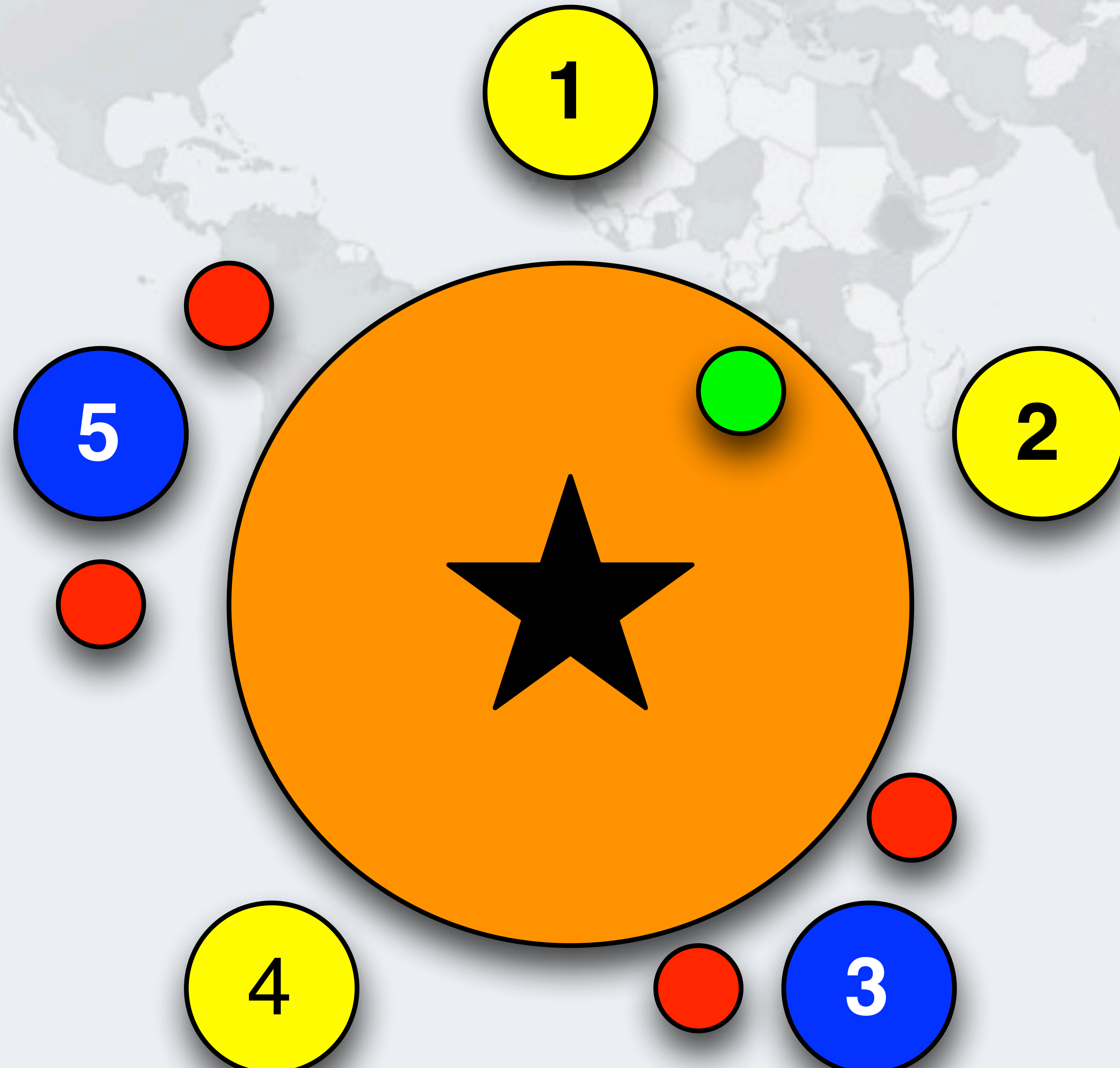
Philosopher 5 Is Now Drinking With Both Cups



Philosophers 3 Wants To Drink, Takes Right Cup

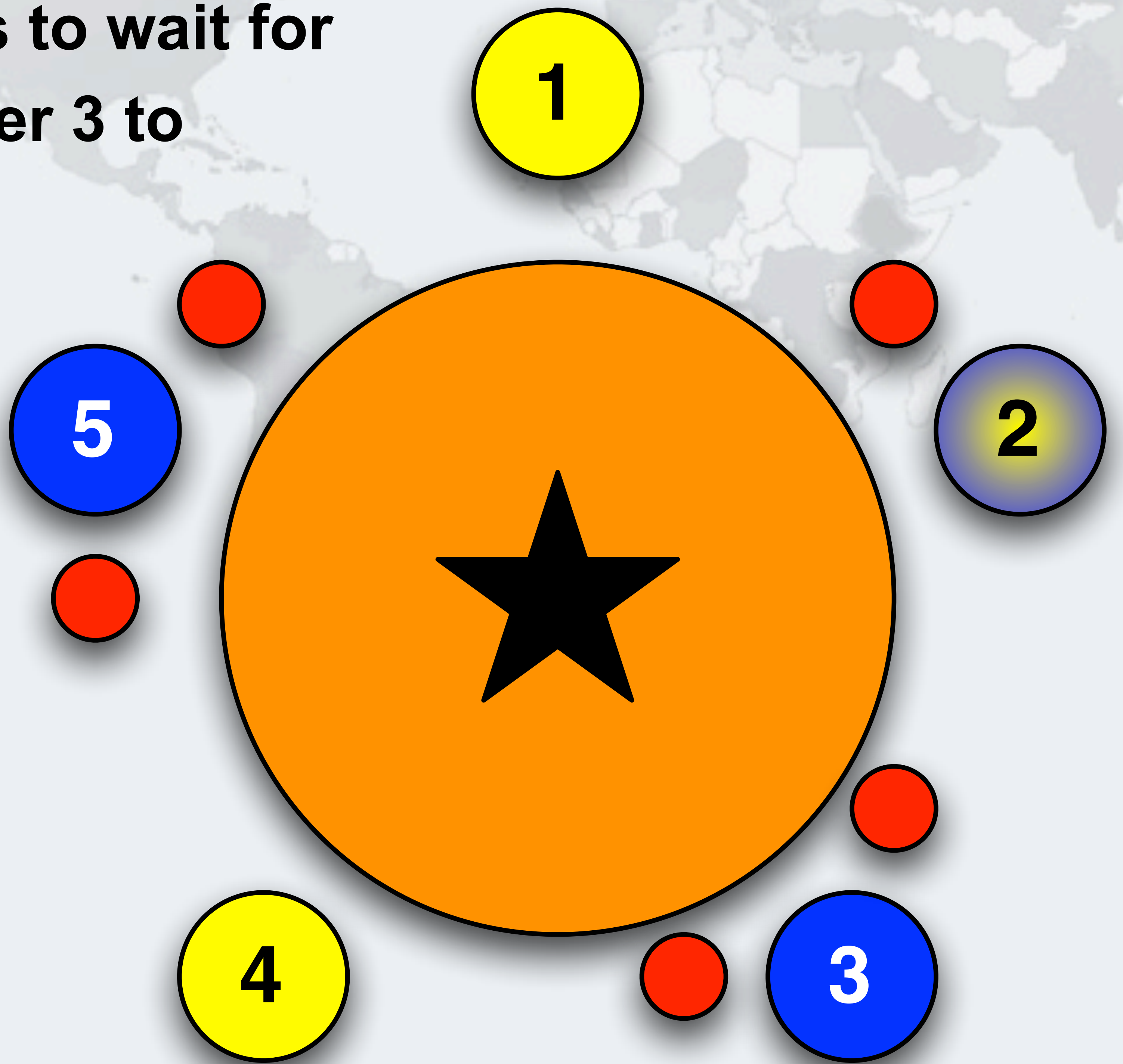


Philosopher 3 Is Now Drinking With Both Cups

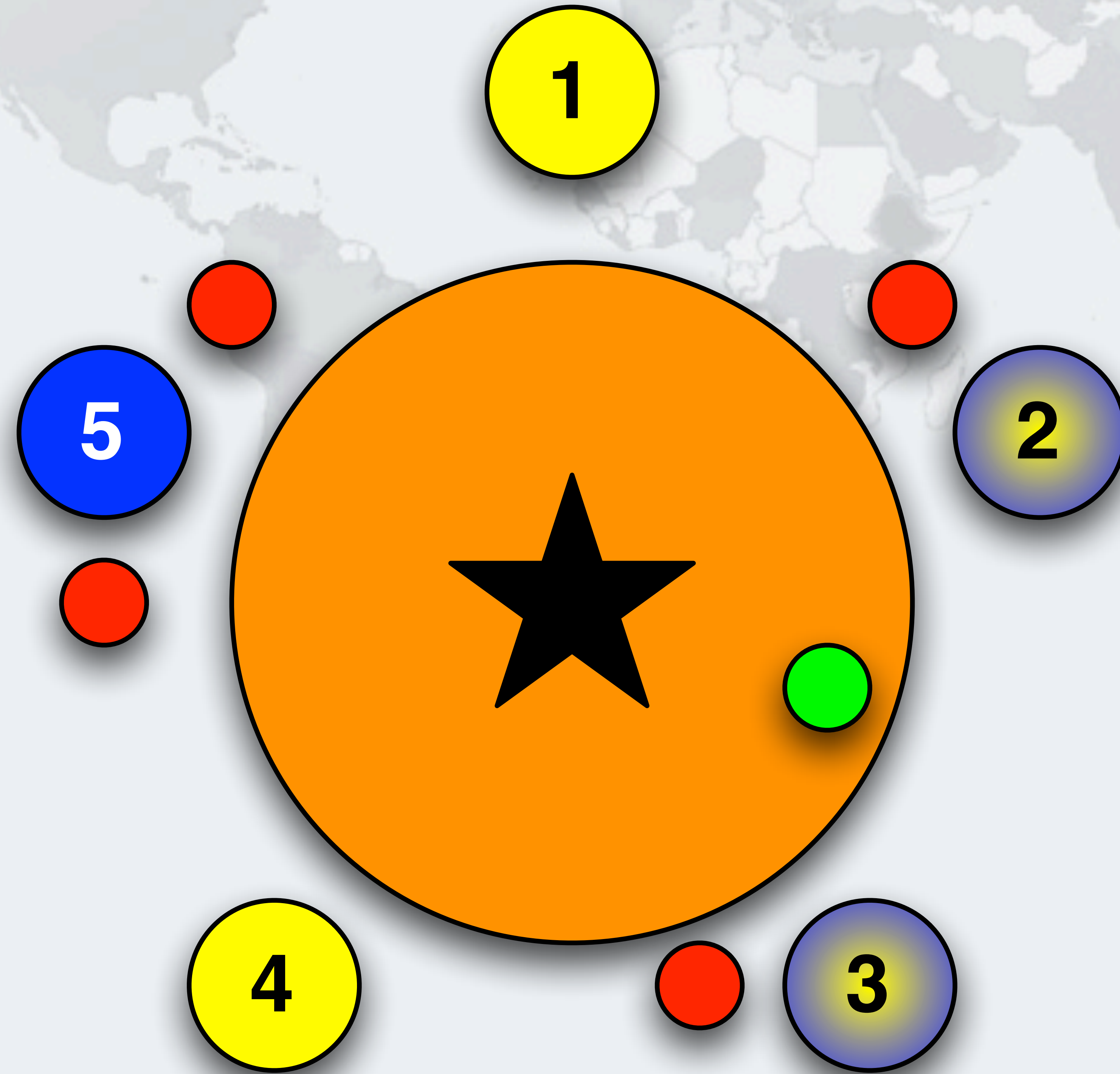


Philosophers 2 Wants To Drink, Takes Right Cup

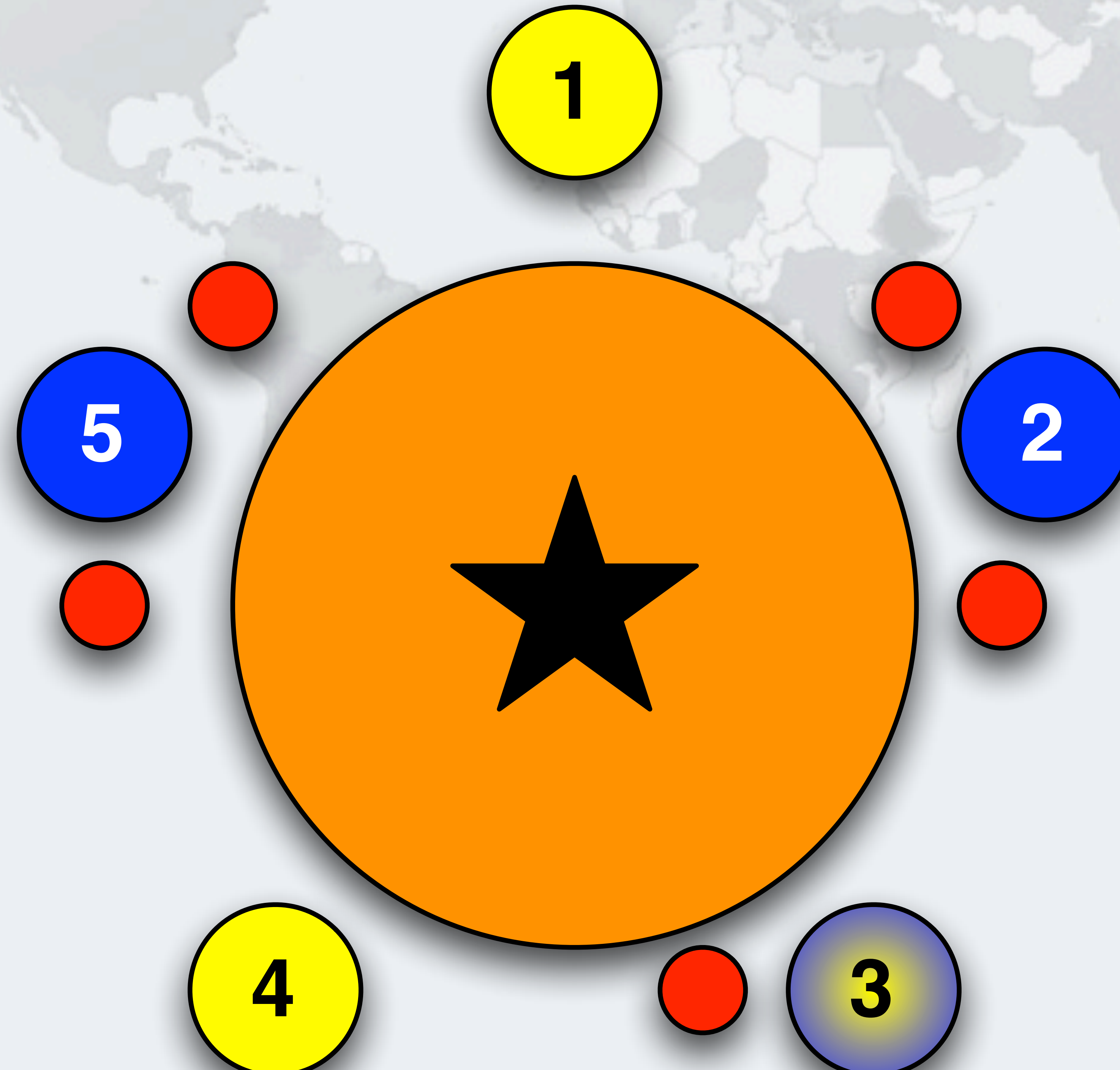
- But he has to wait for Philosopher 3 to finish his drinking session



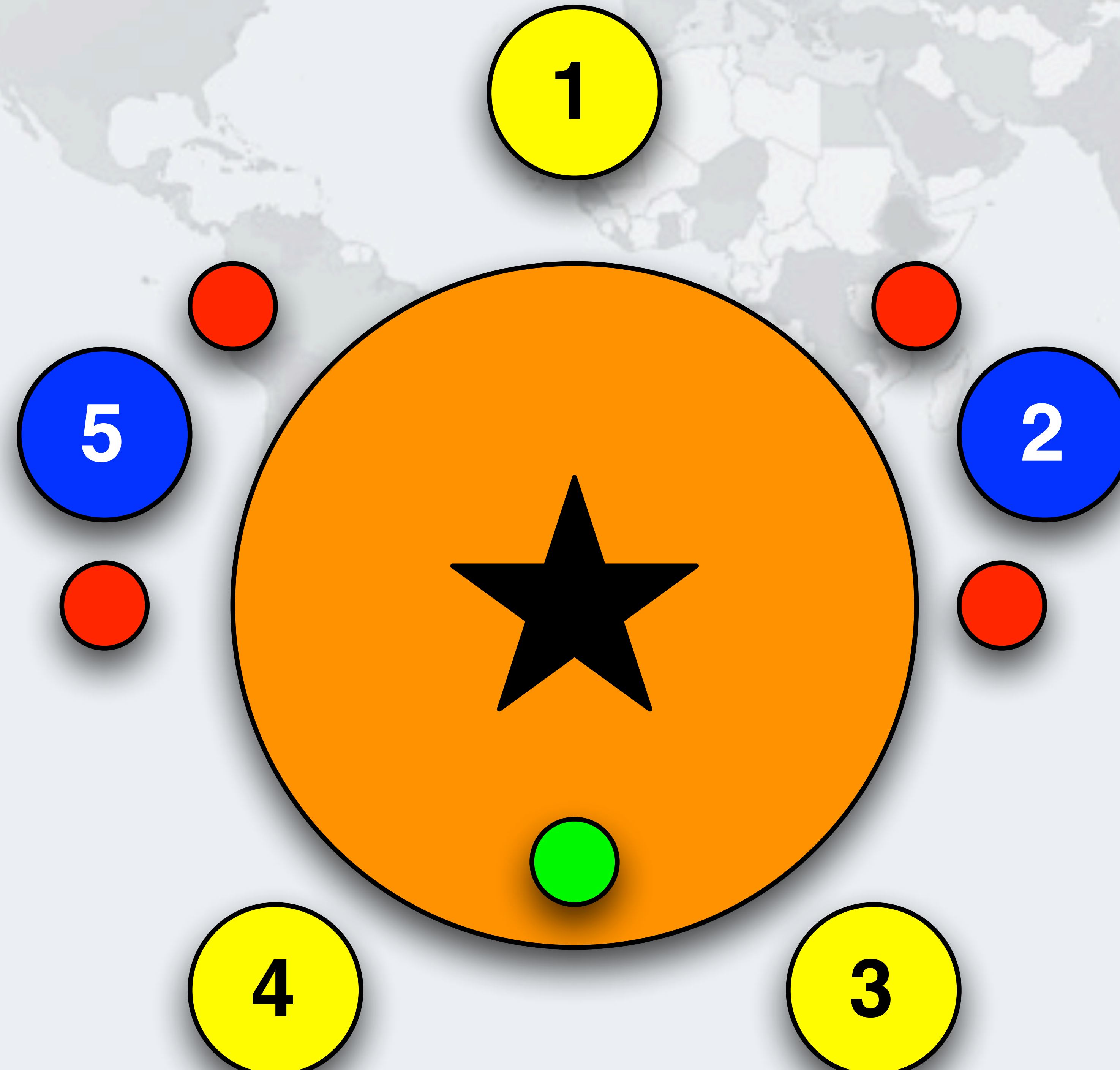
Philosopher 3 Finished Drinking, Returns Right Cup



Philosopher 2 Is Now Drinking With Both Cups



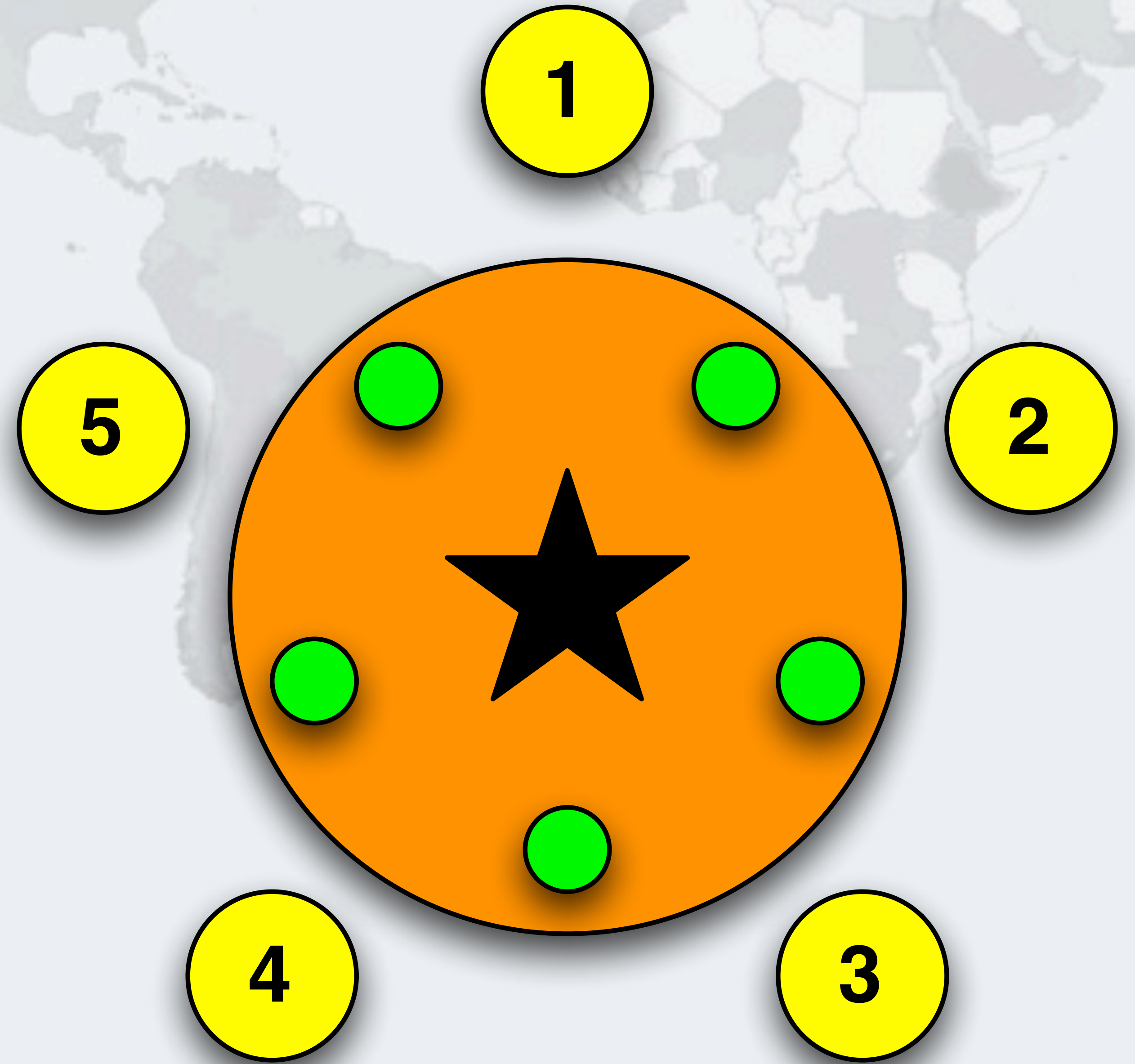
Philosopher 3 Returns Left Cup



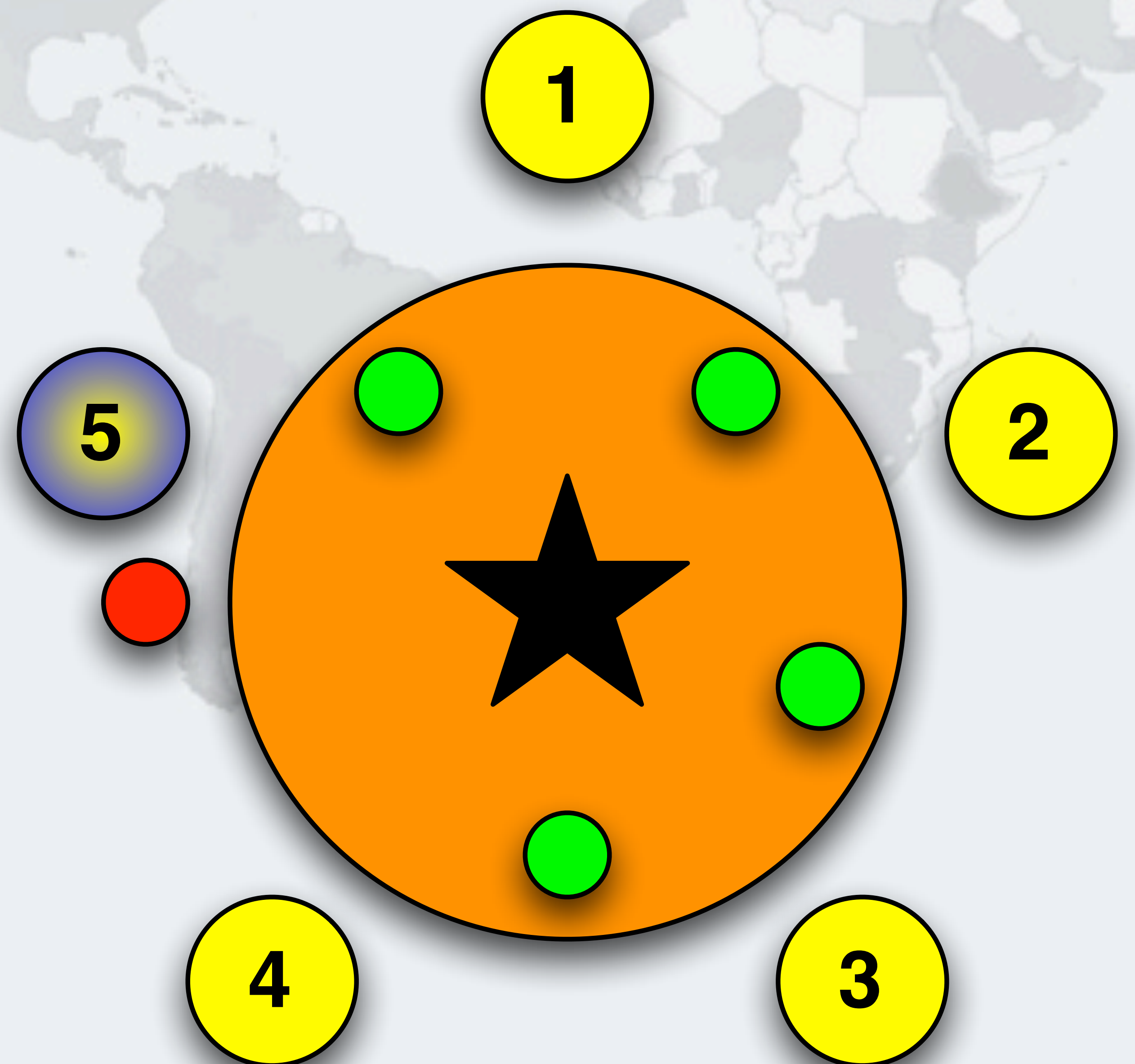
Drinking Philosophers In Limbo

- **The standard rule is that every philosopher first picks up the right cup, then the left**
  - **If all of the philosophers want to drink and they all pick up the right cup, then they all are holding one cup but cannot get the left cup**

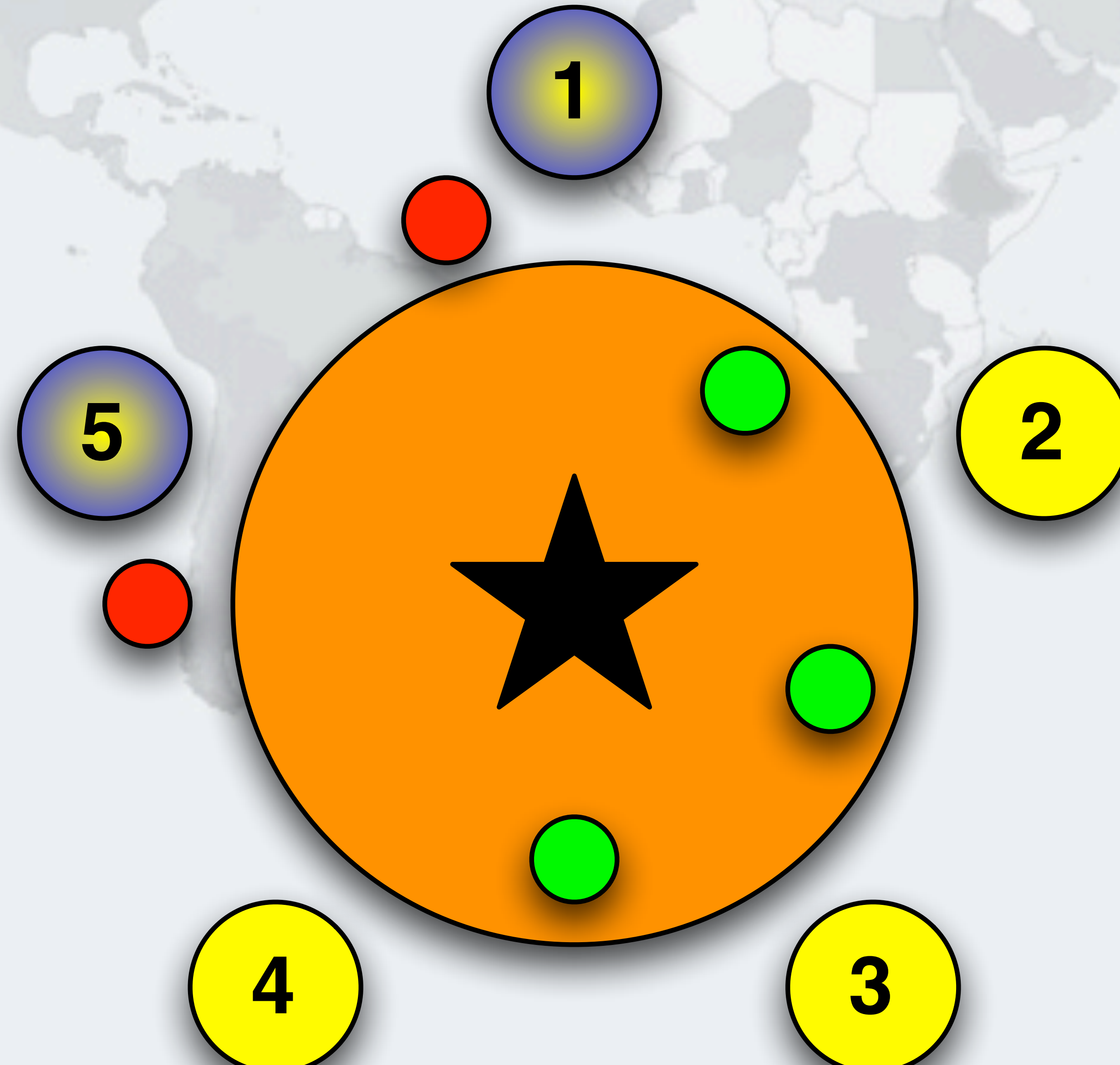
A Deadlock Can Easily Happen With This Design



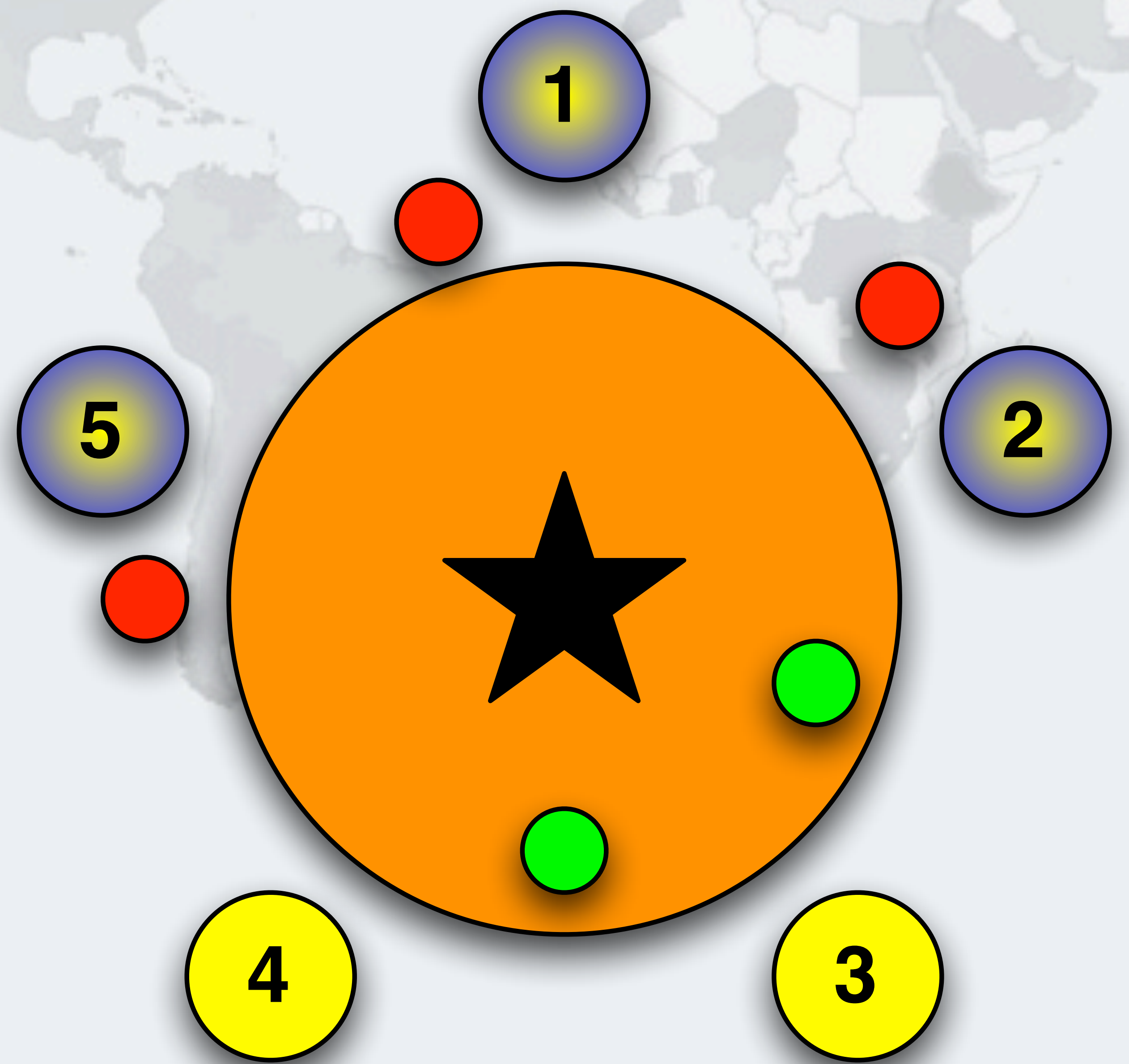
Philosopher 5 Wants To Drink, Takes Right Cup



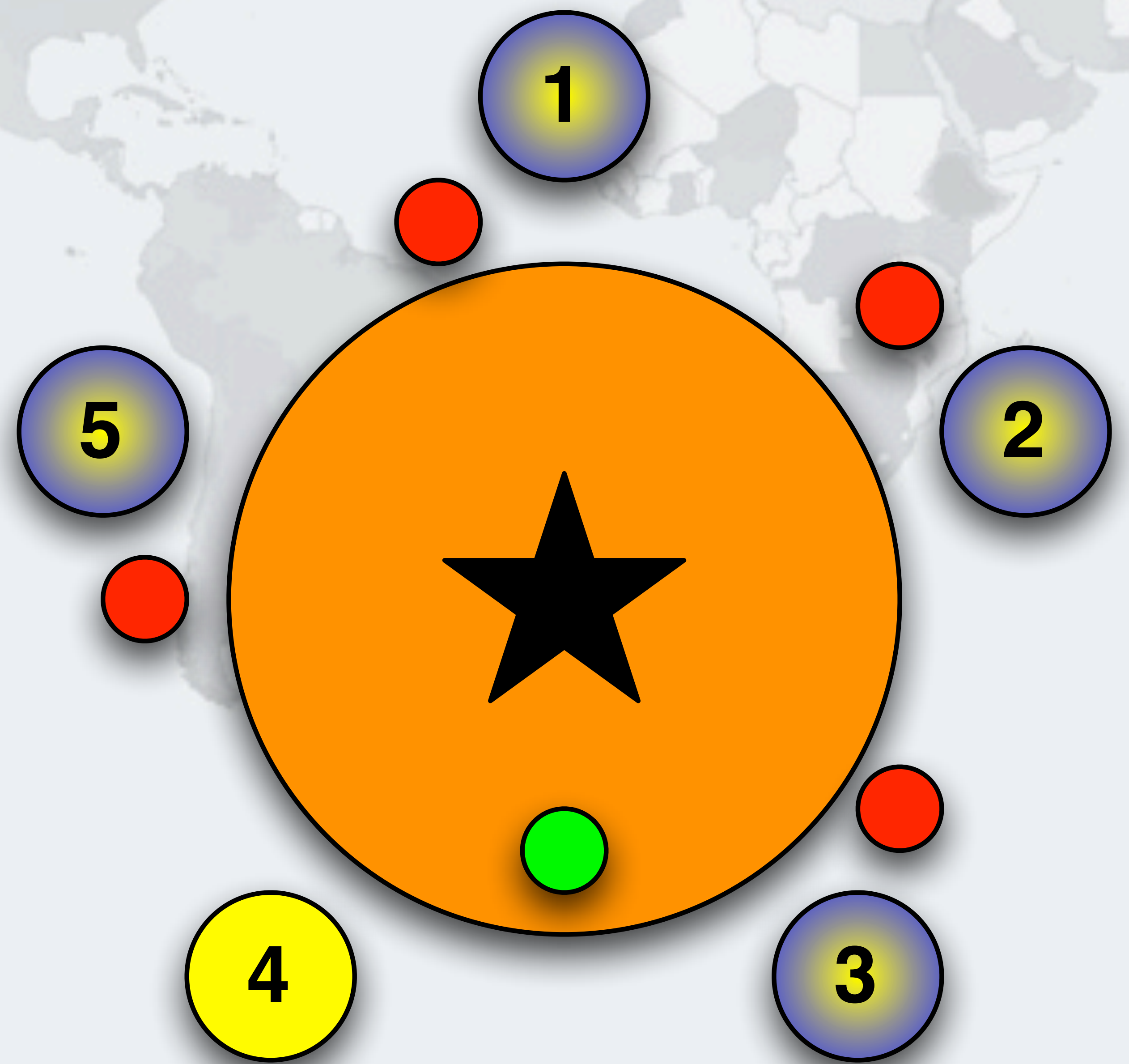
Philosopher 1 Wants To Drink, Takes Right Cup



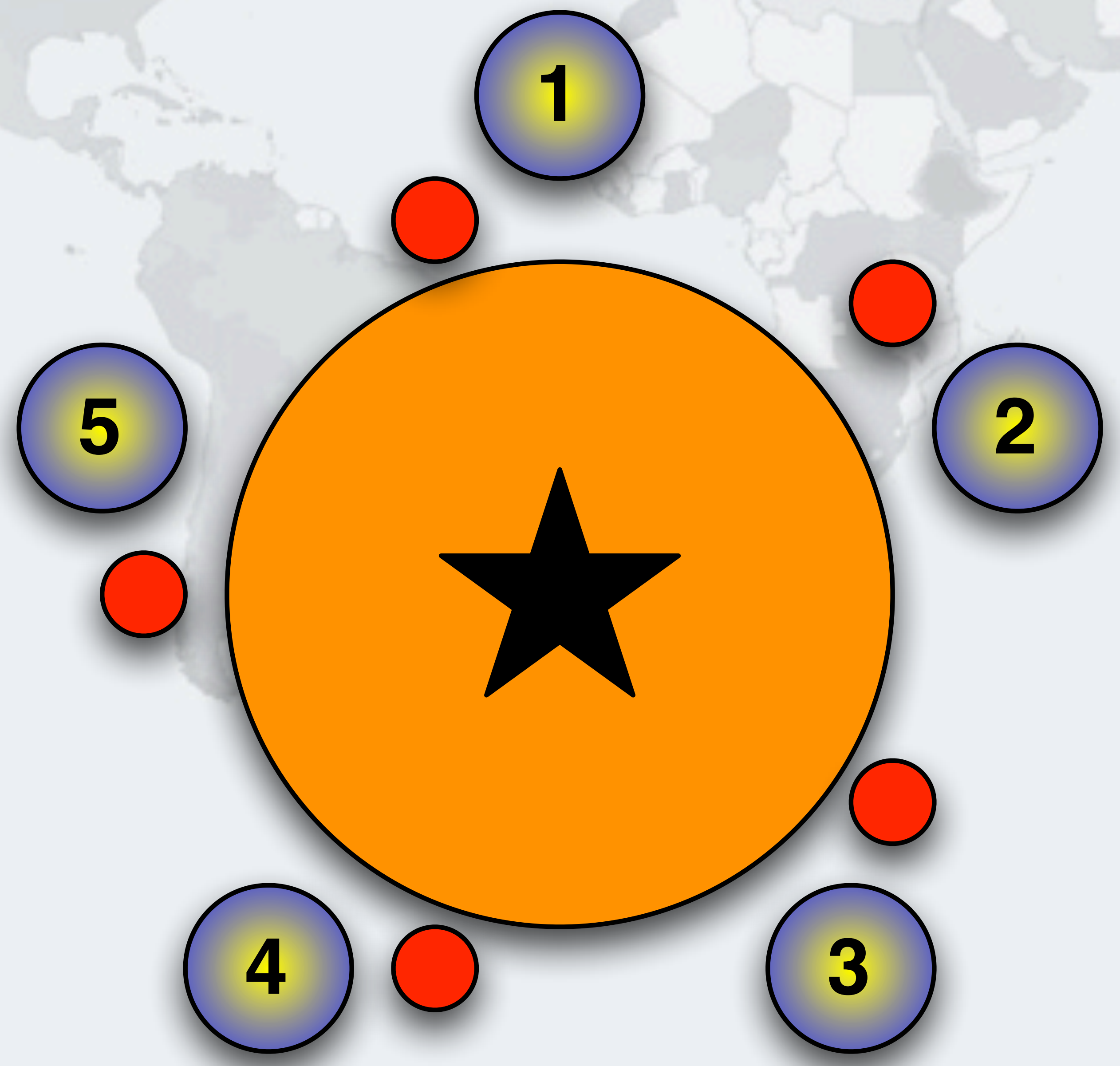
Philosopher 2 Wants To Drink, Takes Right Cup



Philosopher 3 Wants To Drink, Takes Right Cup

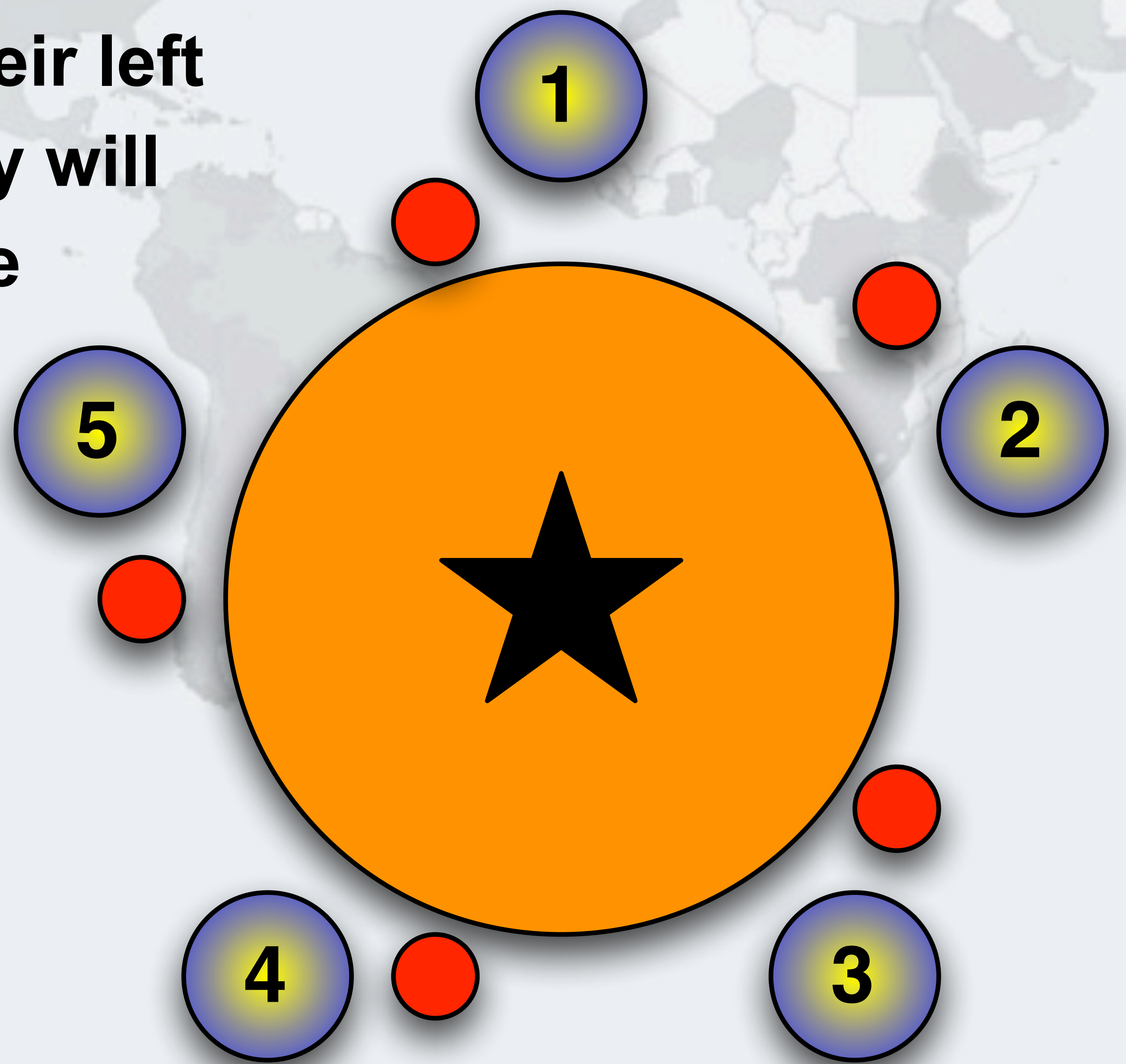


Philosopher 4 Wants To Drink, Takes Right Cup



Deadlock!

- All the philosophers are waiting for their left cups, but they will never become available



Resolving Deadlocks

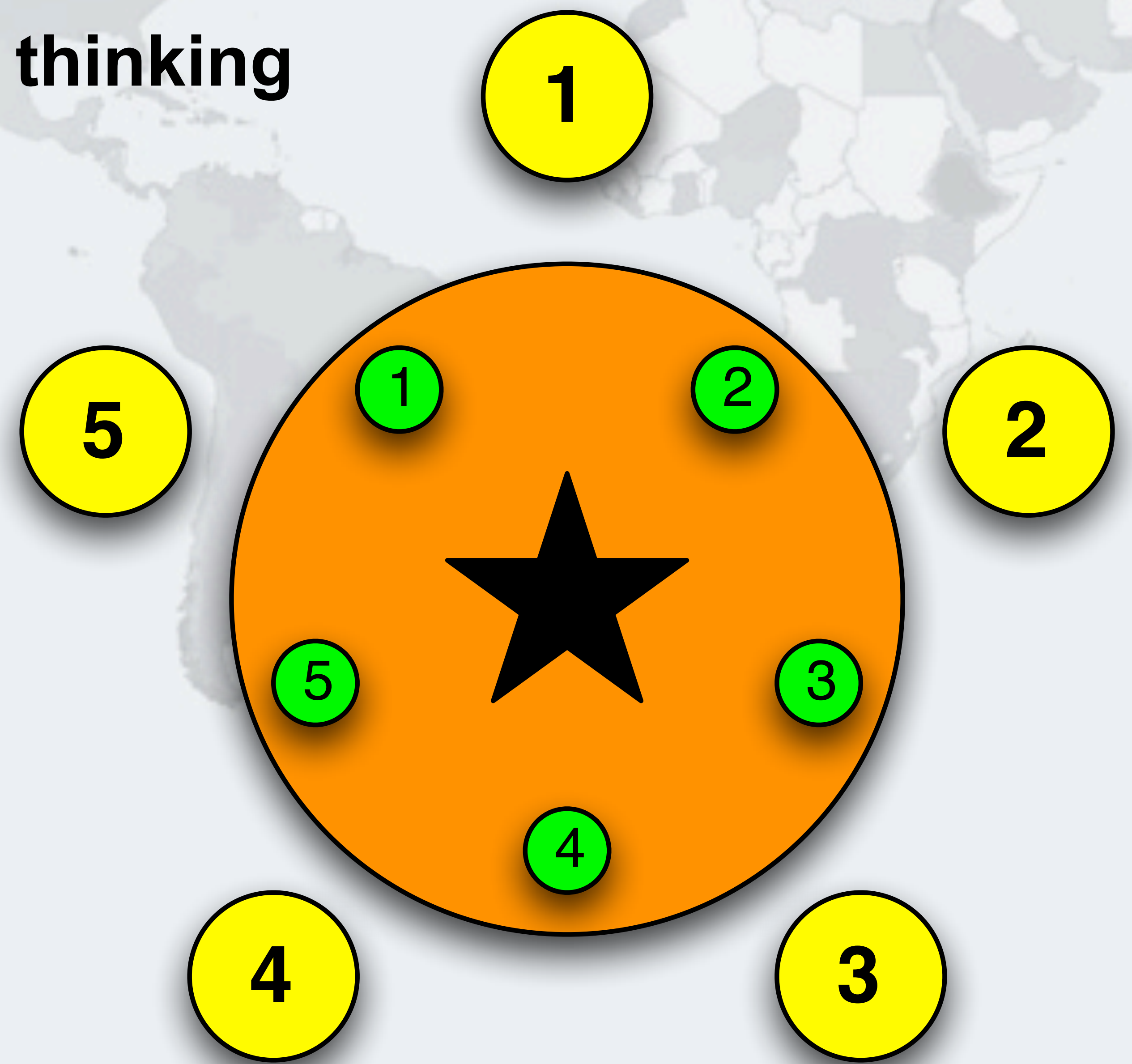
- **Deadlocks can be discovered automatically by searching the graph of call stacks, looking for circular dependencies**
  - ThreadMXBean can find deadlocks for us, but cannot fix them
- **In databases, the deadlock is resolved by one of the queries being aborted with an exception**
  - The query could then be retried
- **Java does not have this functionality**
  - When we get a deadlock, there is no clean way to recover from it
  - Prevention is better than the cure

Global Order With Boozing Philosophers

- **If all philosophers hold one cup, we deadlock**
  - Our solution must prevent all philosophers from holding one cup
- **We can solve the deadlock with the "dining philosophers" by requiring that locks are always acquired in a set order**
  - For example, we can make a rule that philosophers always first take the cup with the largest number
    - If it is not available, we block until it becomes available
  - And return the cup with the lowest number first

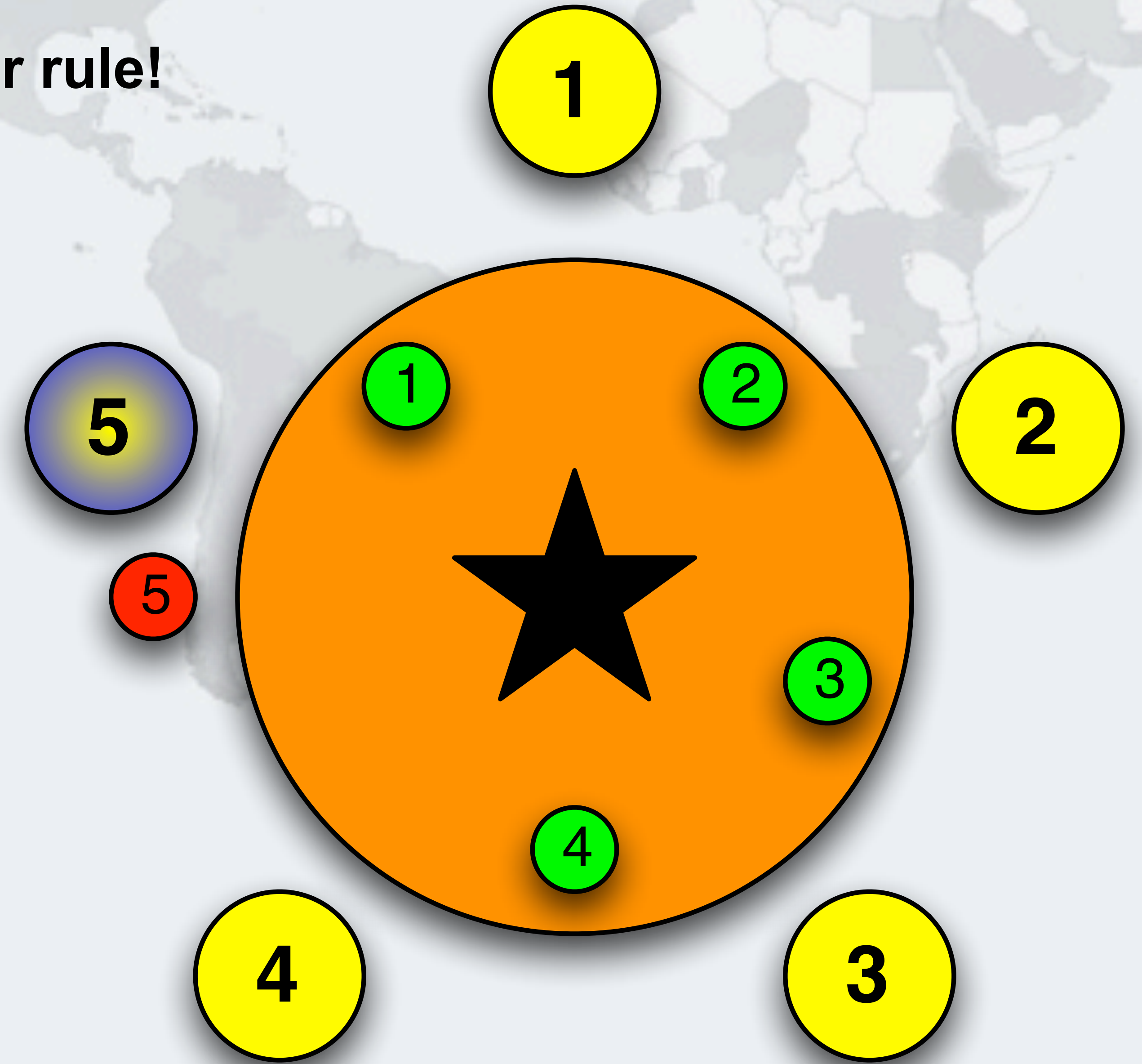
Global Lock Ordering

- We start with all the philosophers thinking



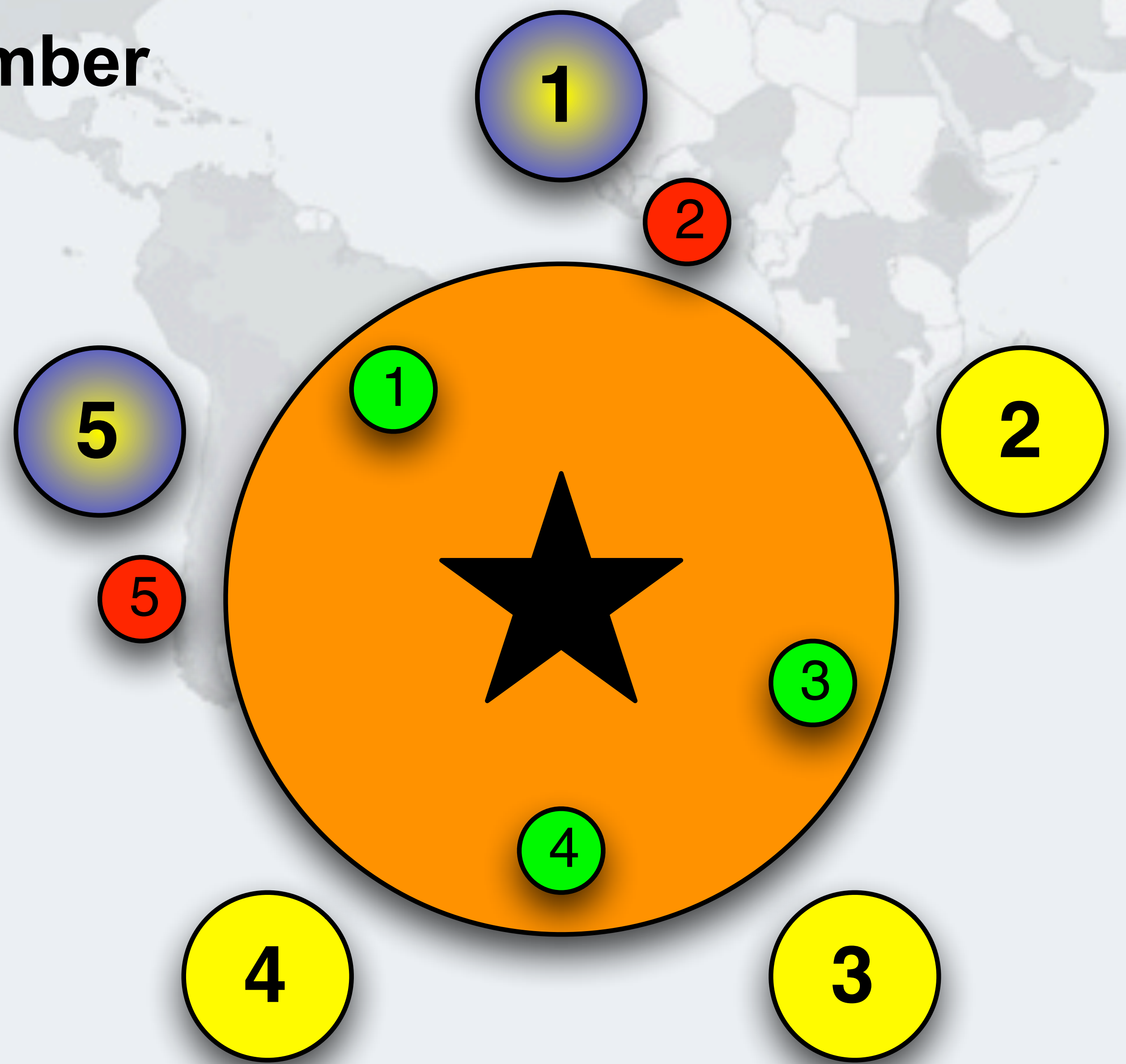
Philosopher 5 Takes Cup 5

- Cup 5 has higher number
  - Remember our rule!

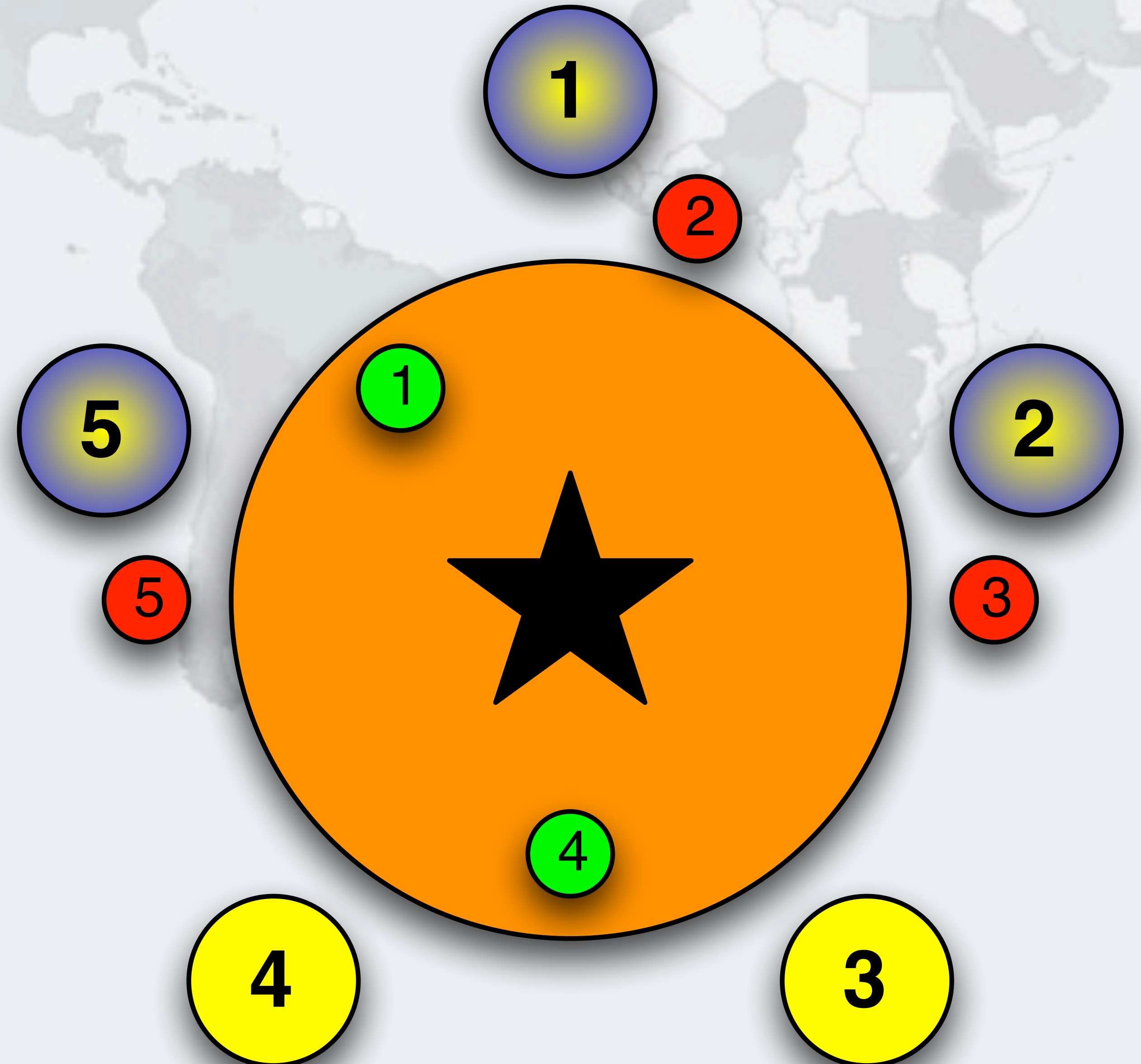


Philosopher 1 Takes Cup 2

- **Must take the cup with the higher number first**
  - **In this case cup 2**

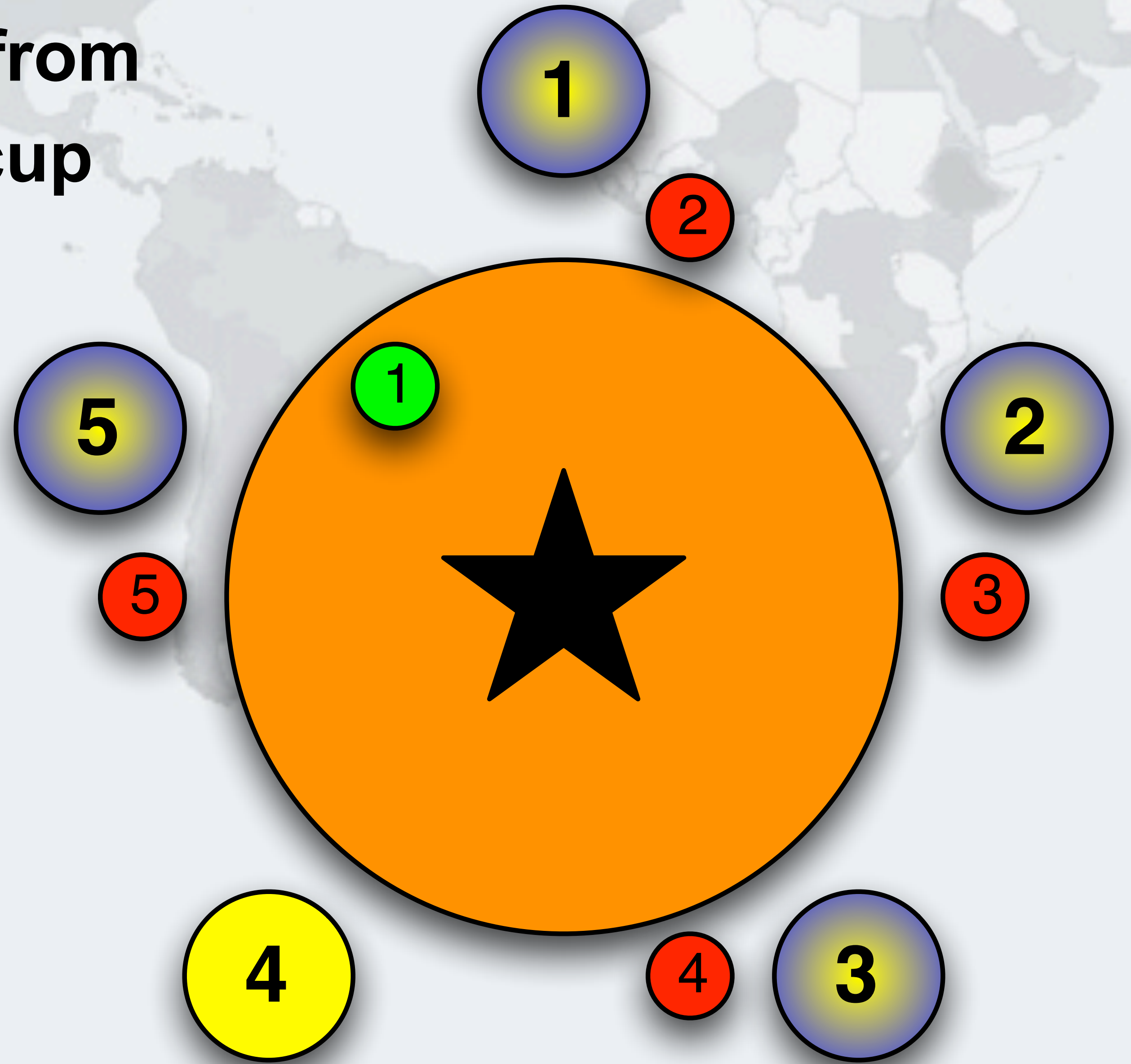


Philosopher 2 Takes Cup 3

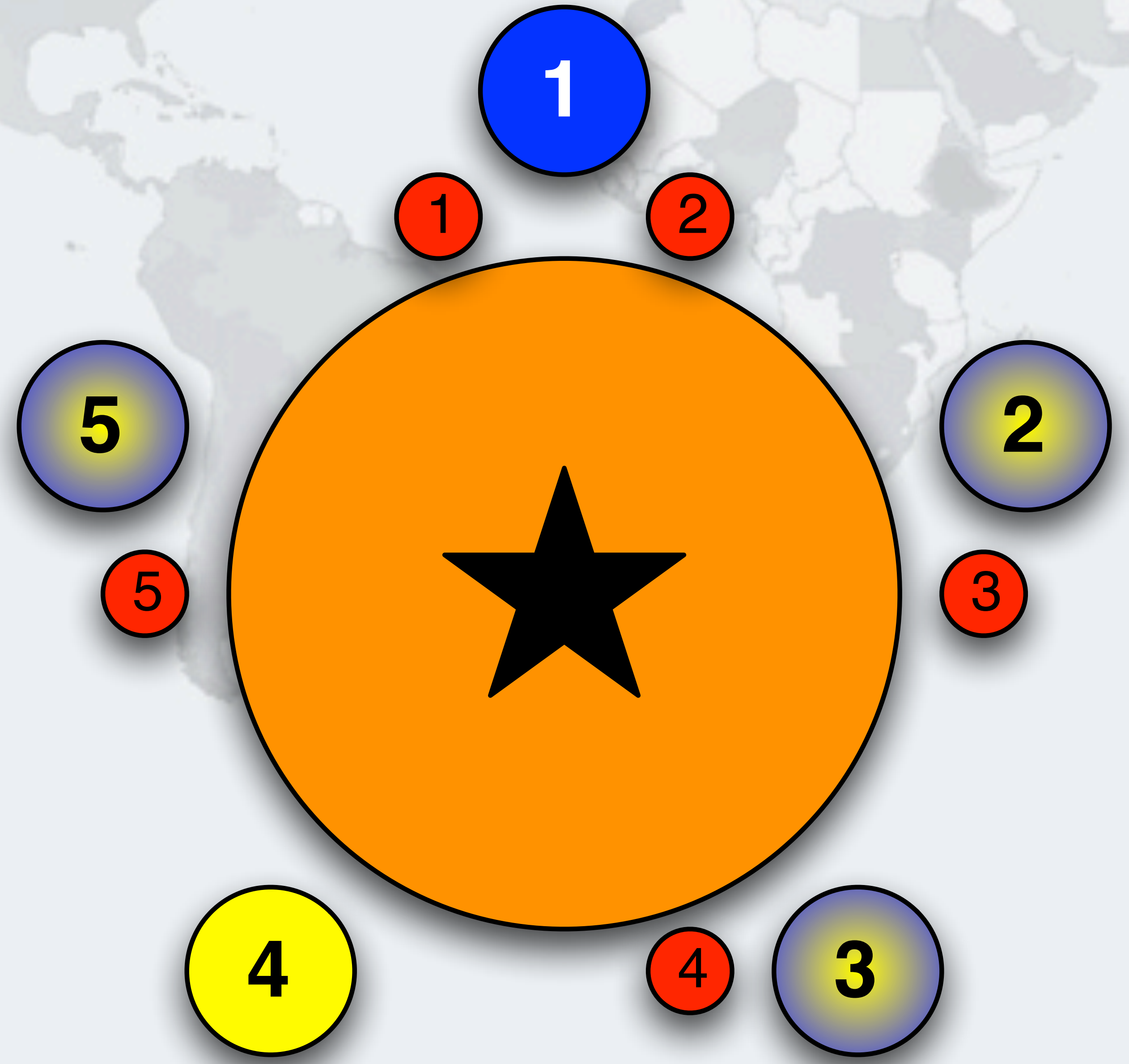


Philosopher 3 Takes Cup 4

- Note that philosopher 4 is prevented from holding one cup

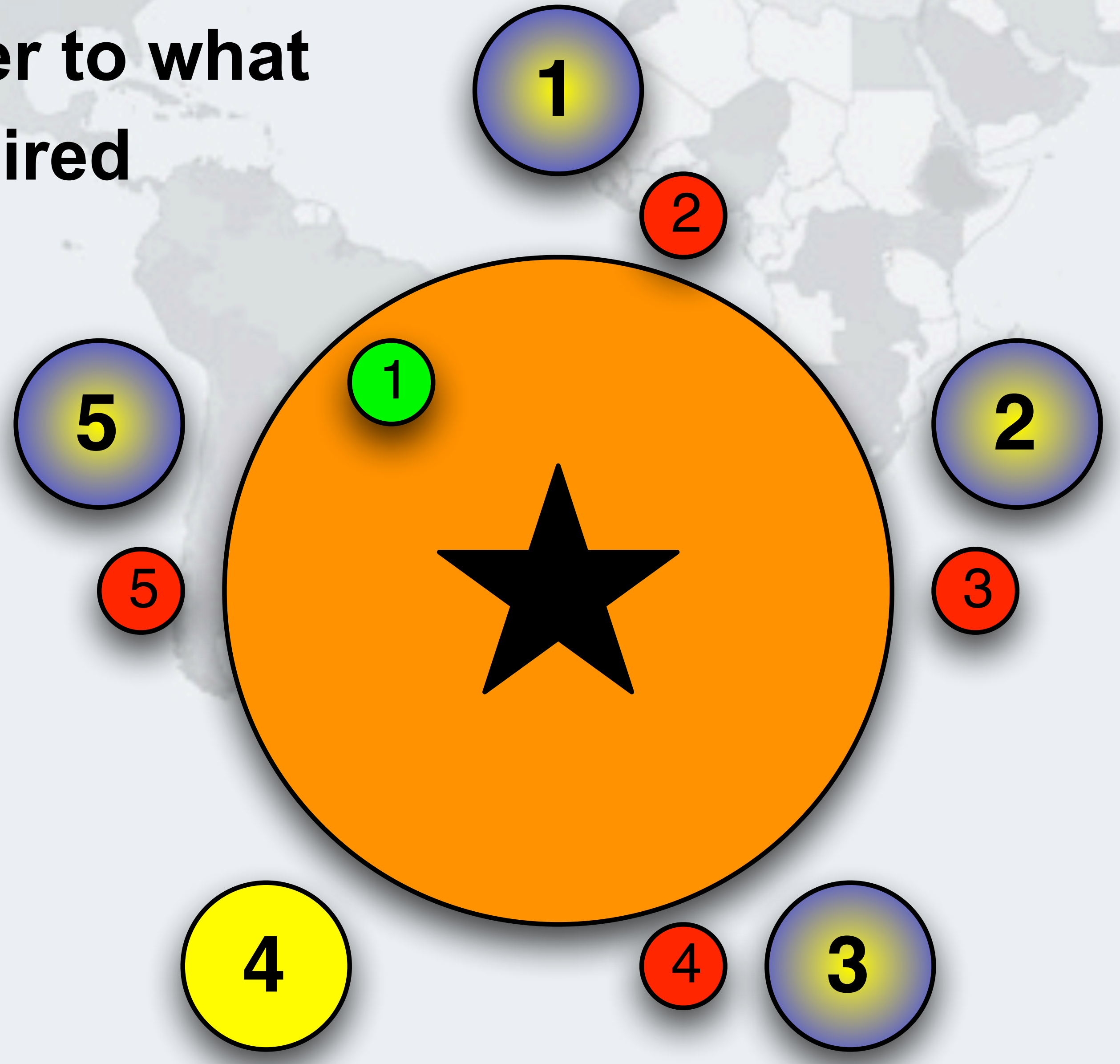


Philosopher 1 Takes Cup 1 - Drinking

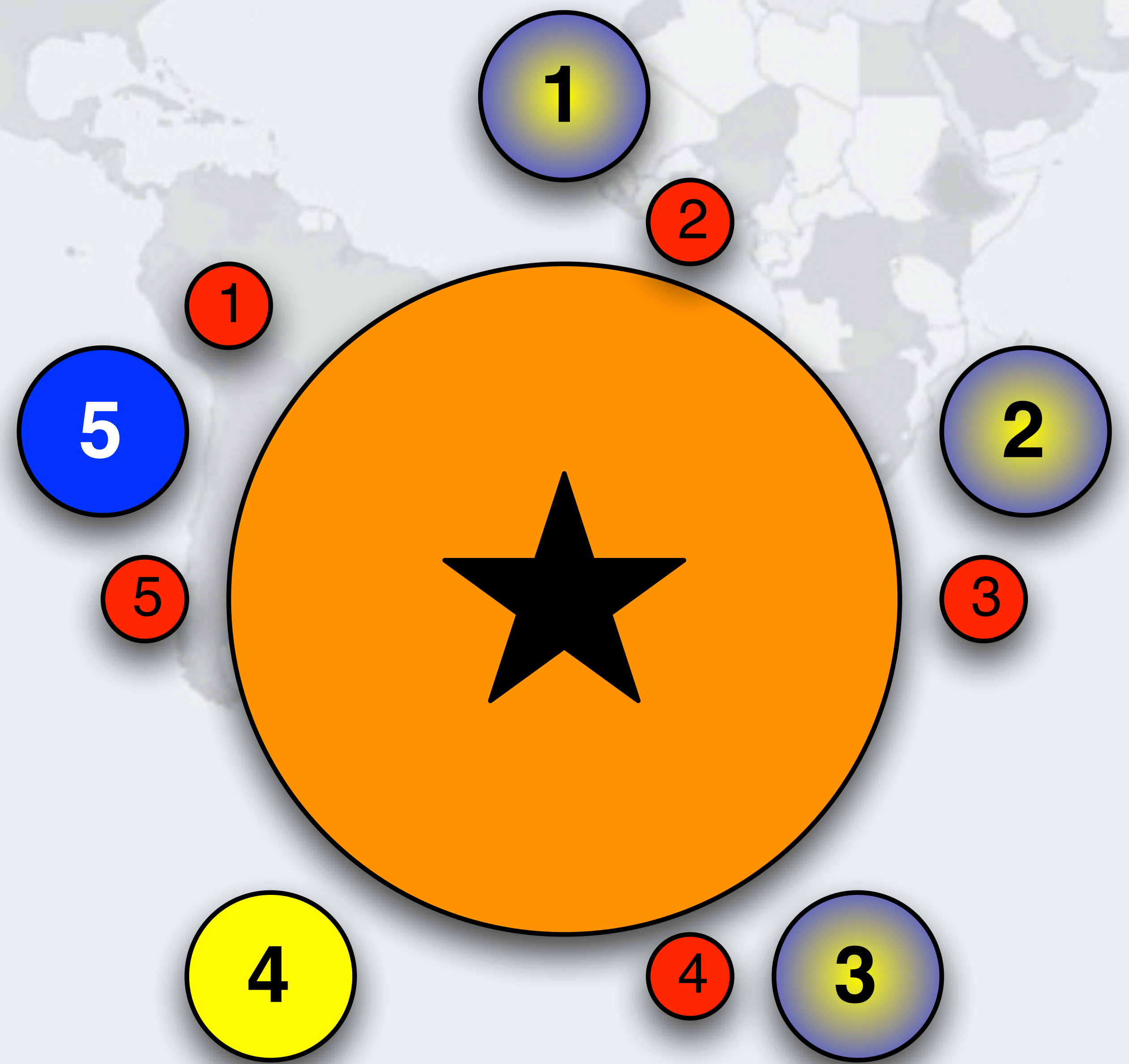


Philosopher 1 Returns Cup 1

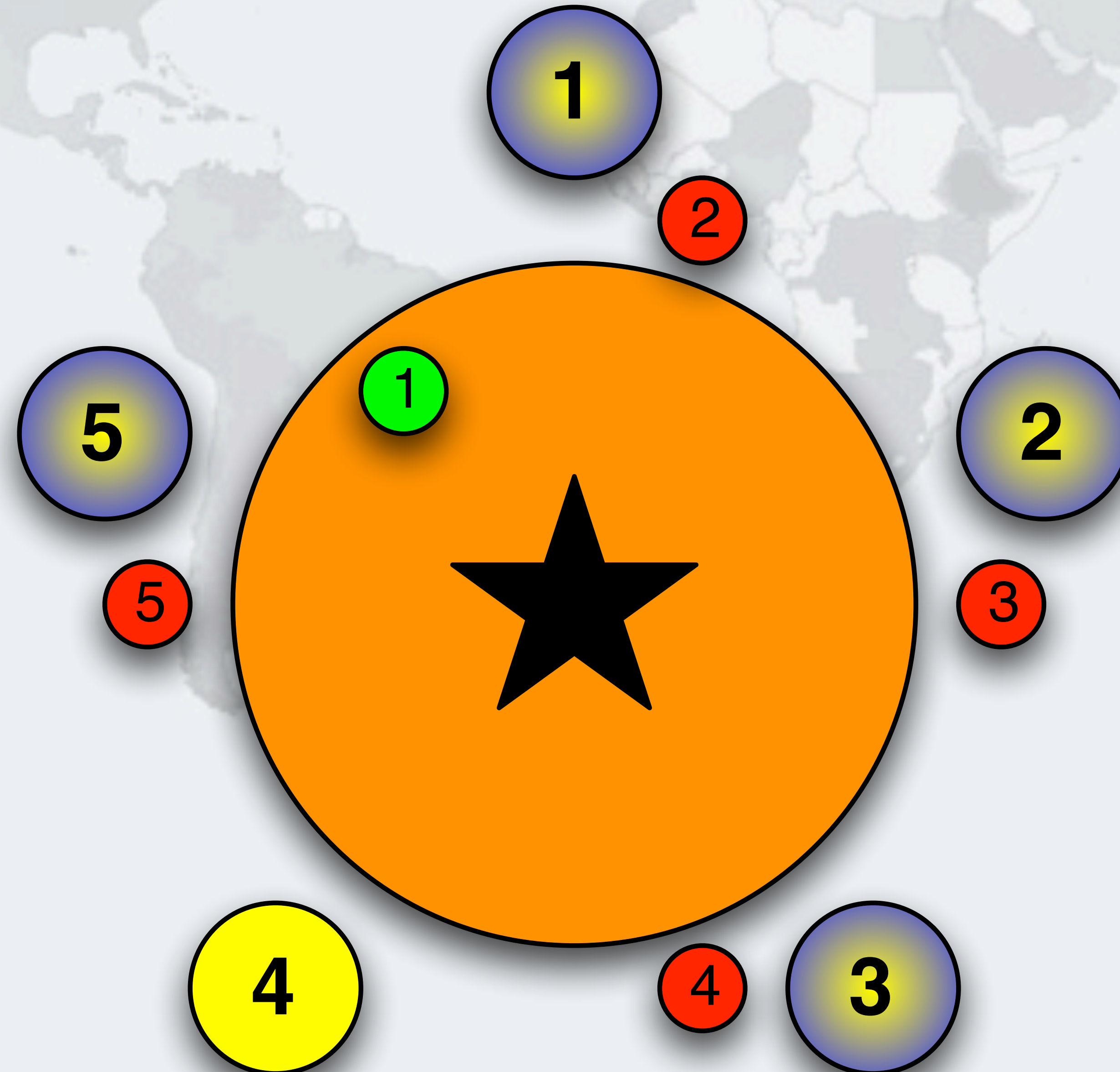
- Cups are returned in the opposite order to what they are acquired



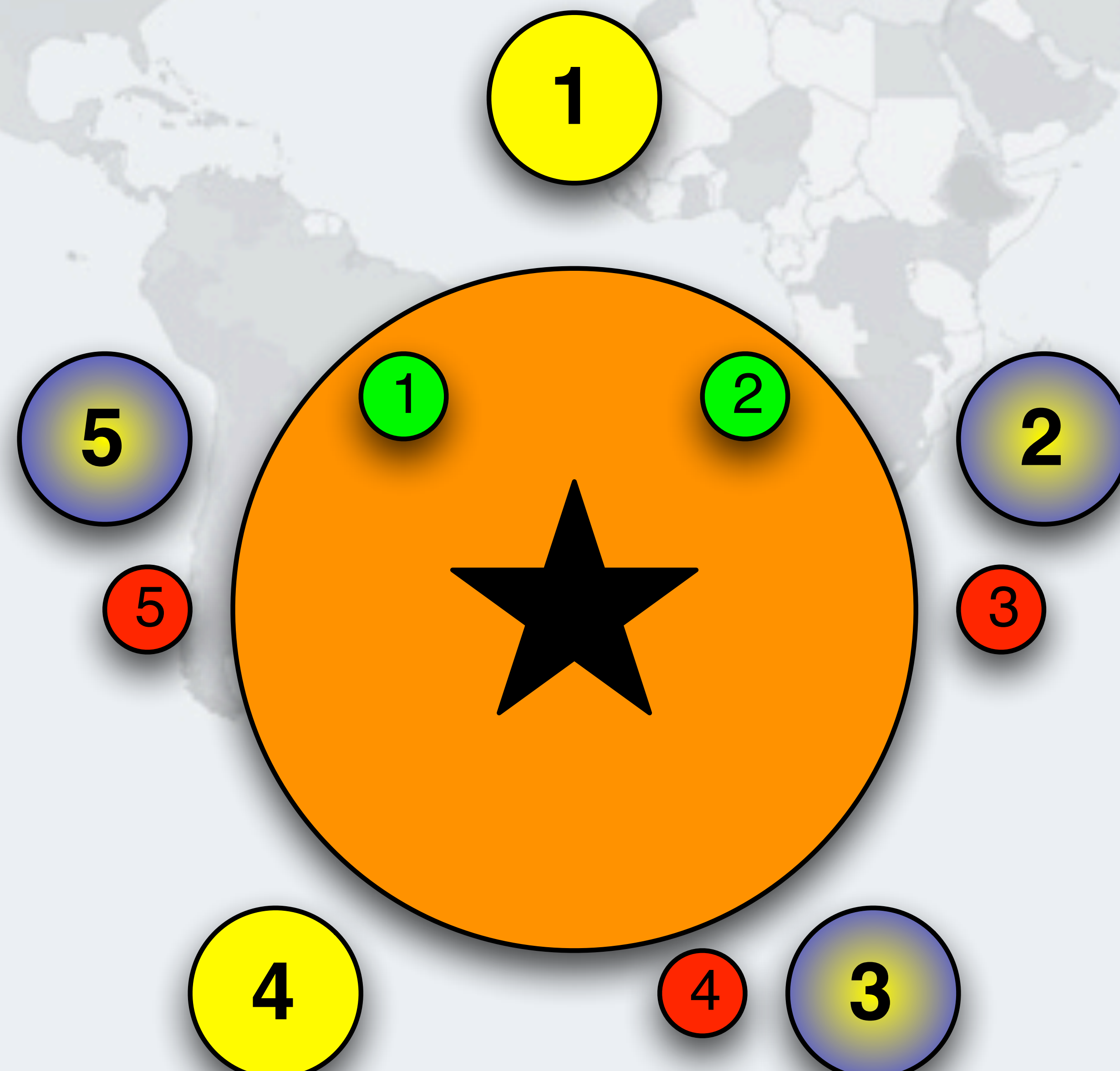
Philosopher 5 Takes Cup 1 - Drinking



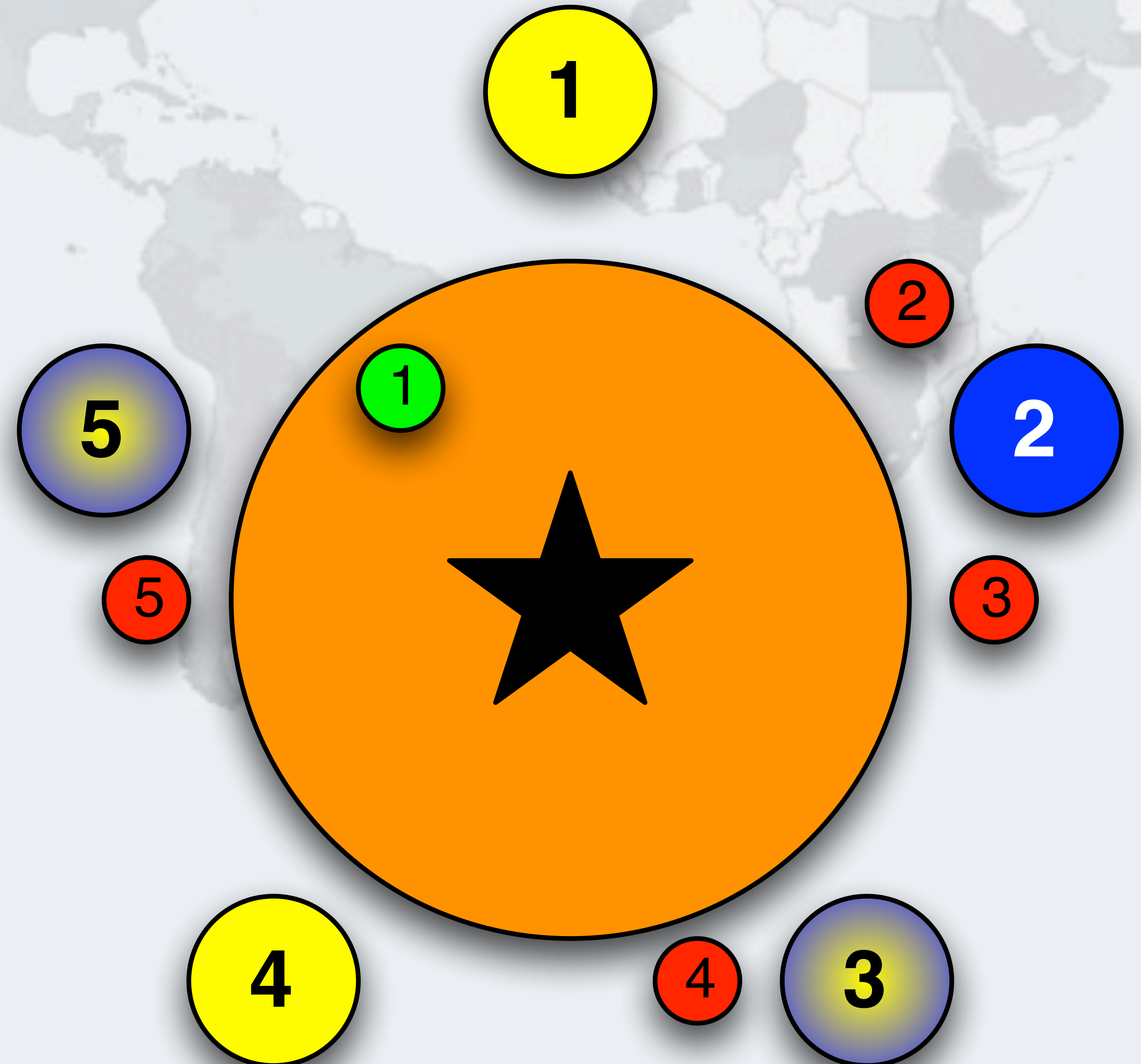
Philosopher 5 Returns Cup 1



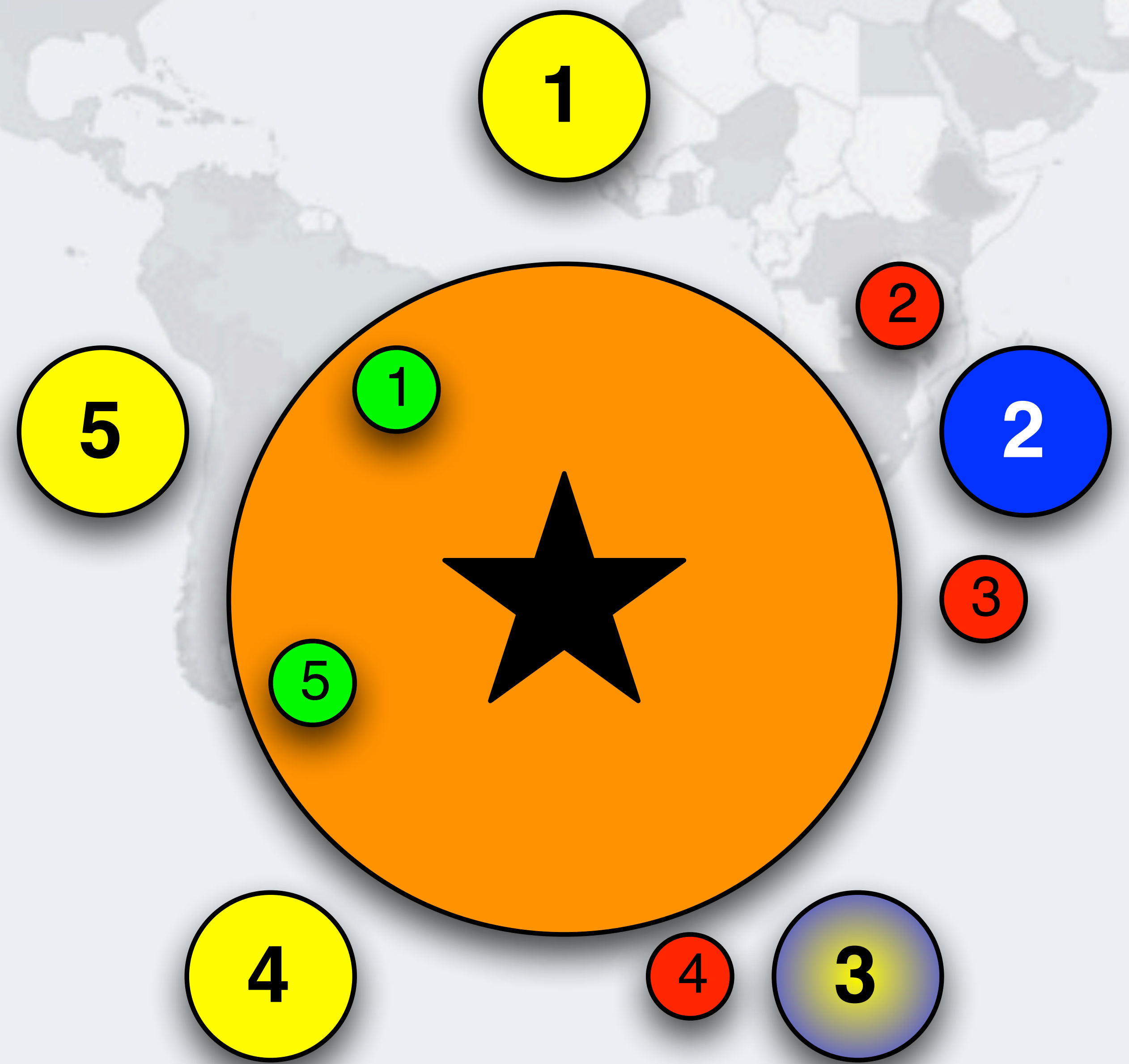
Philosopher 1 Returns Cup 2



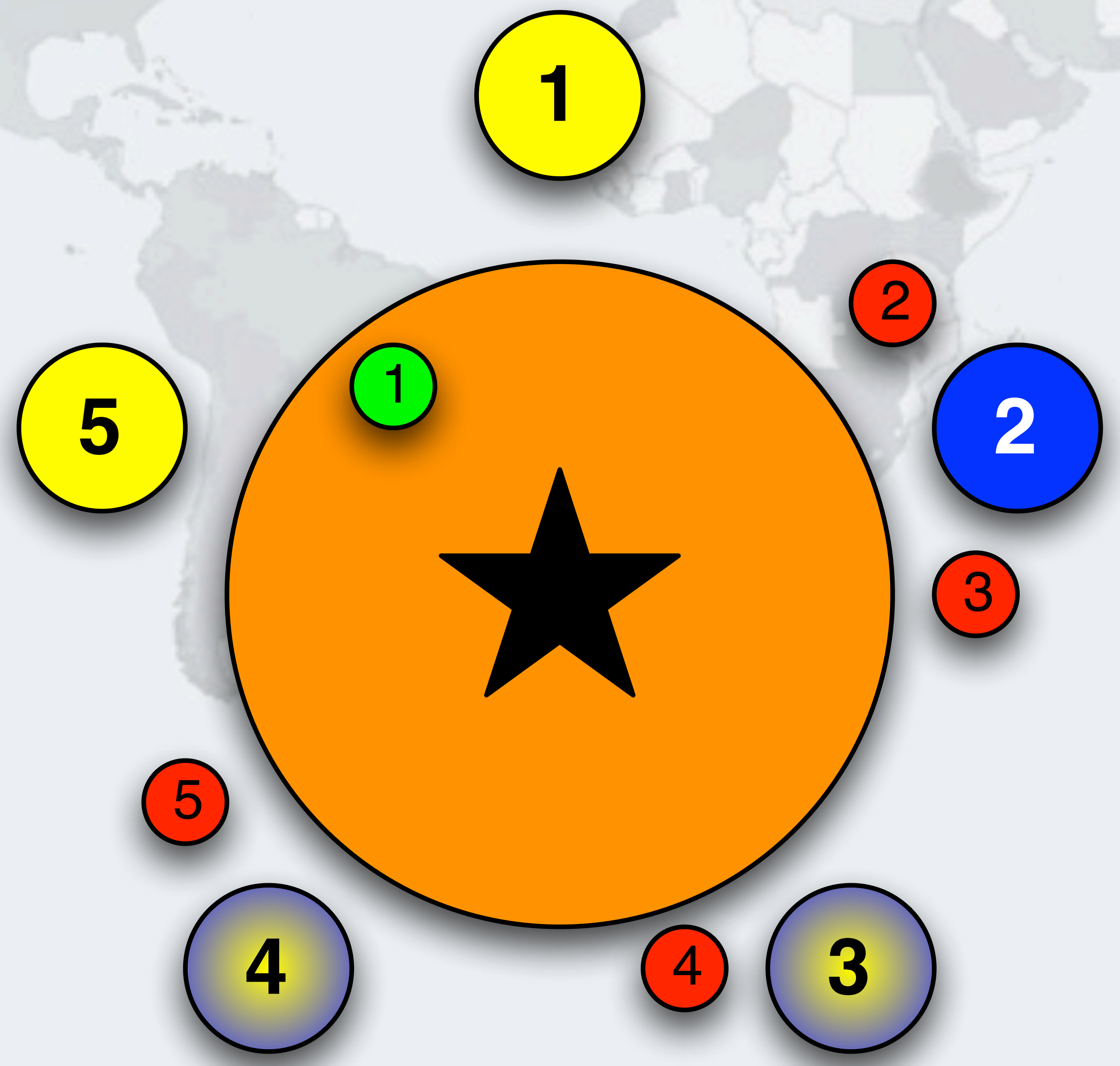
Philosopher 2 Takes Cup 2 - Drinking



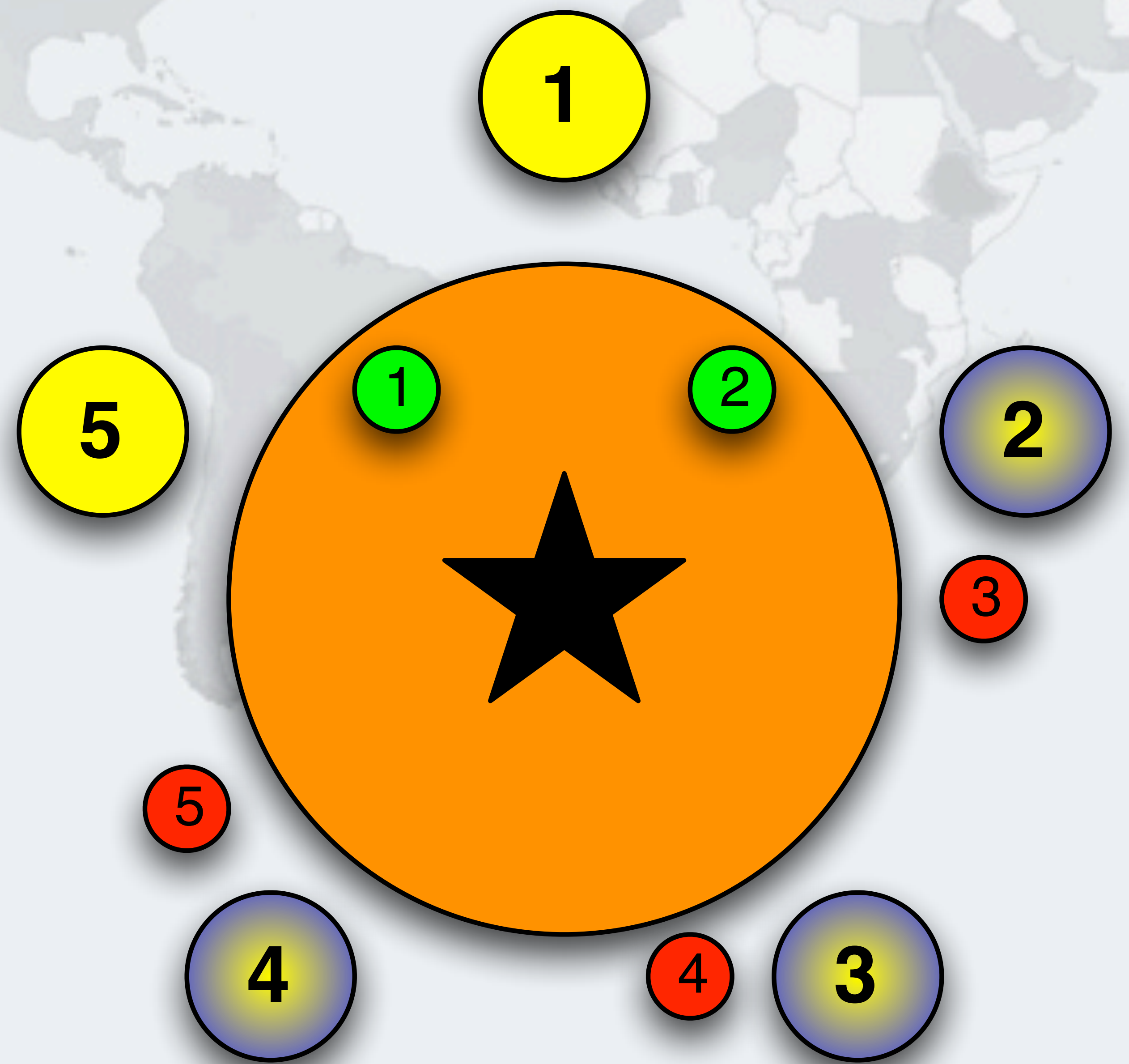
Philosopher 5 Returns Cup 5



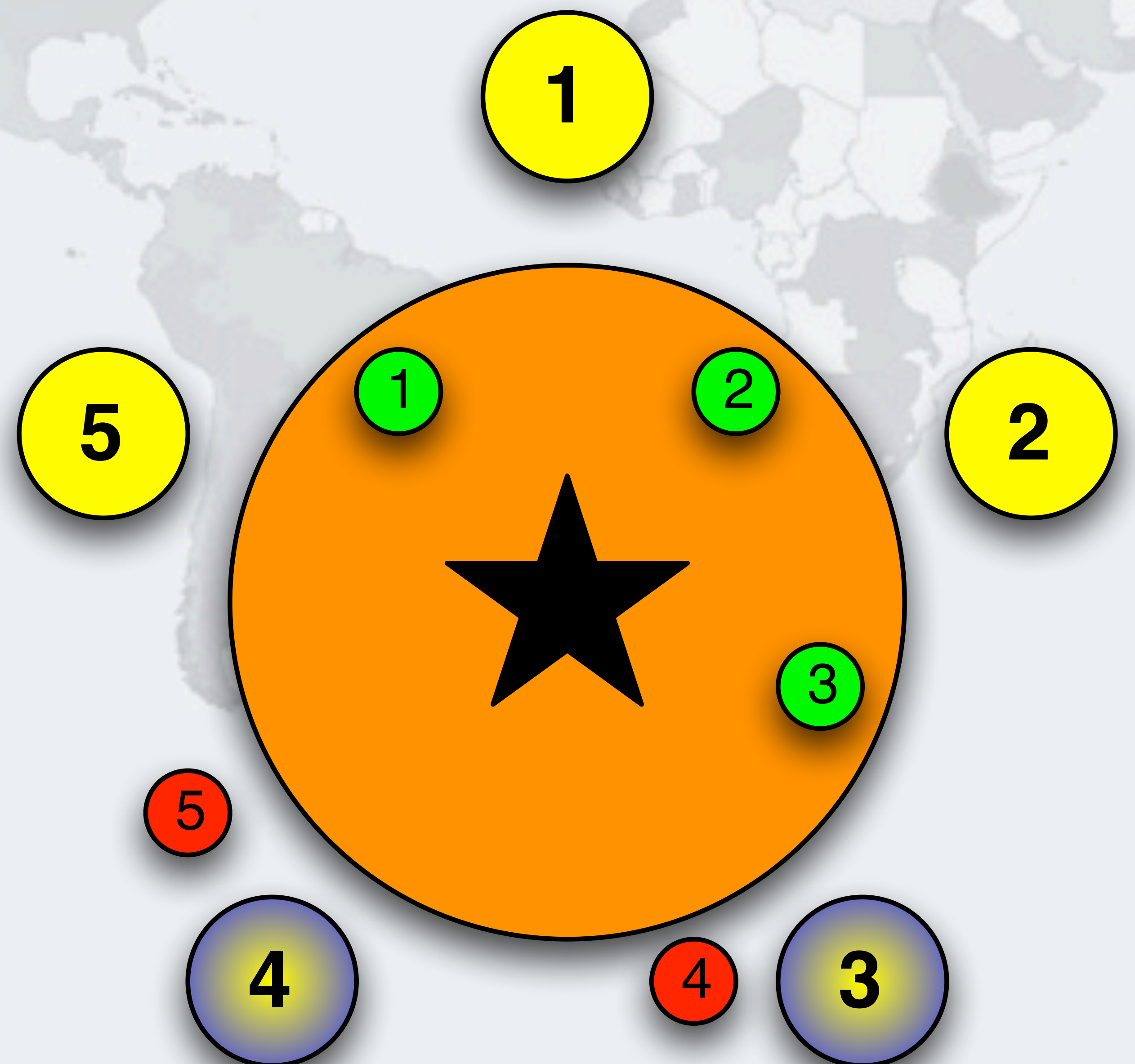
Philosopher 4 Takes Cup 5



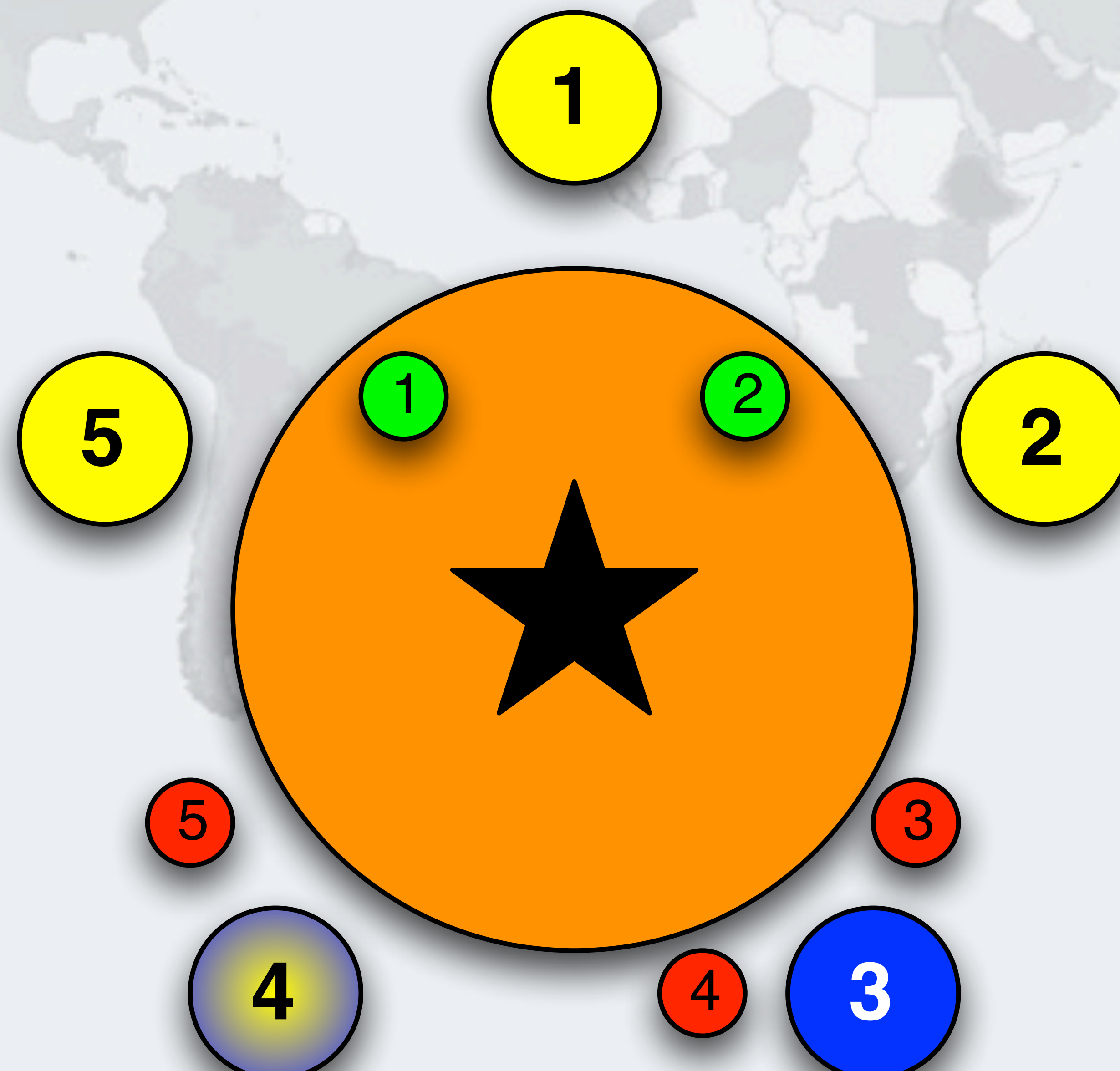
Philosopher 2 Returns Cup 2



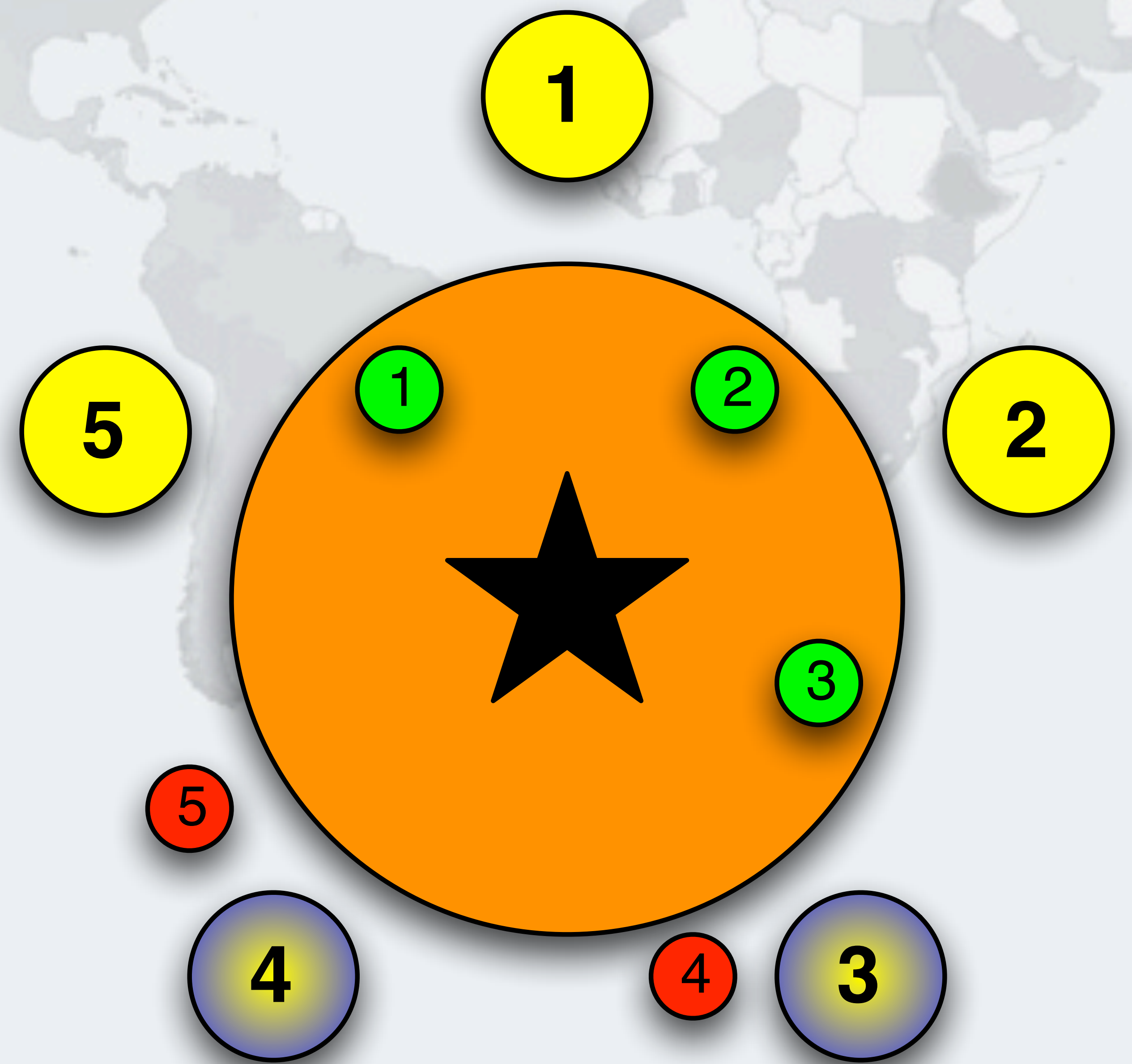
Philosopher 2 Returns Cup 3



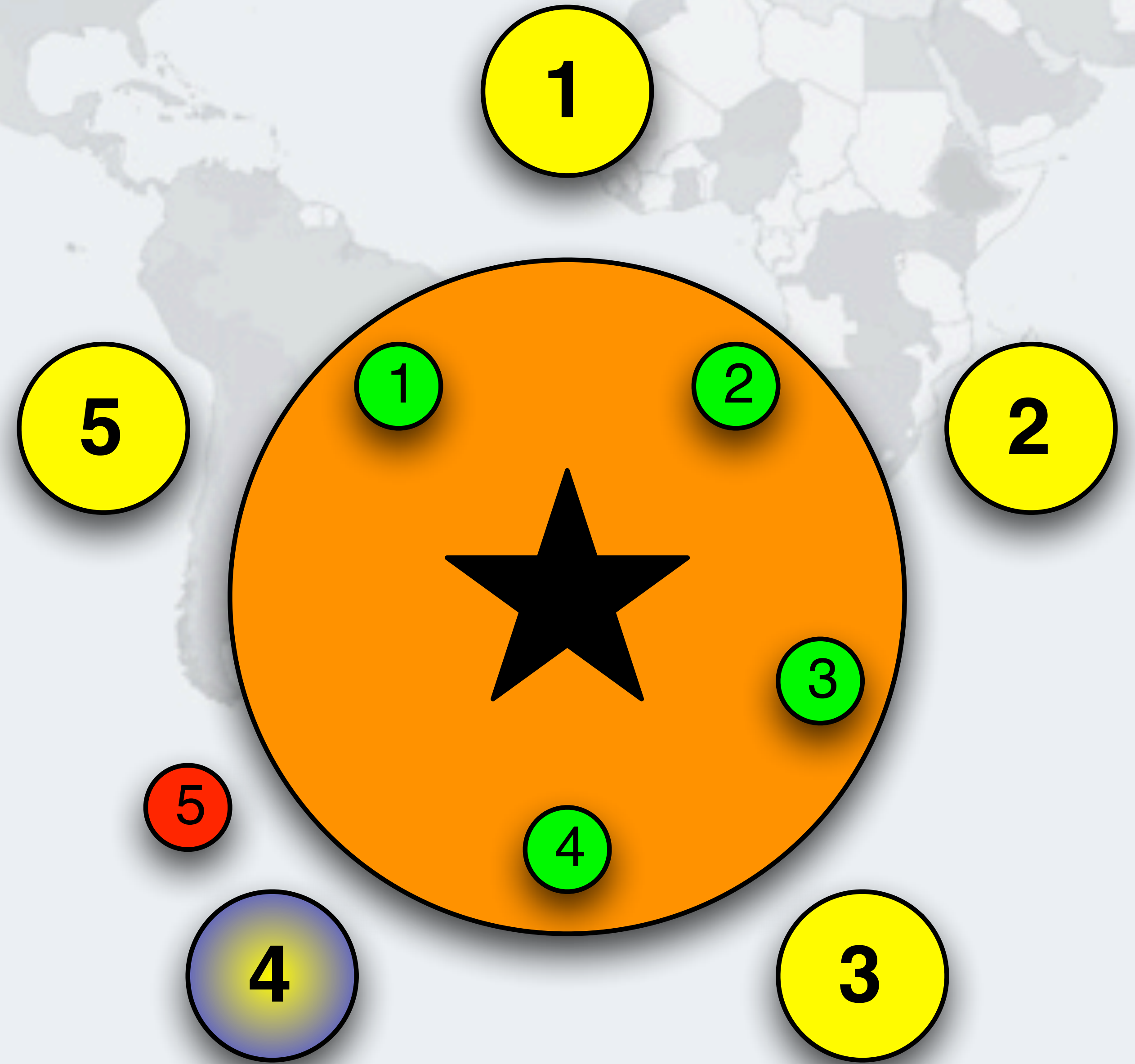
Philosopher 3 Takes Cup 3 - Drinking



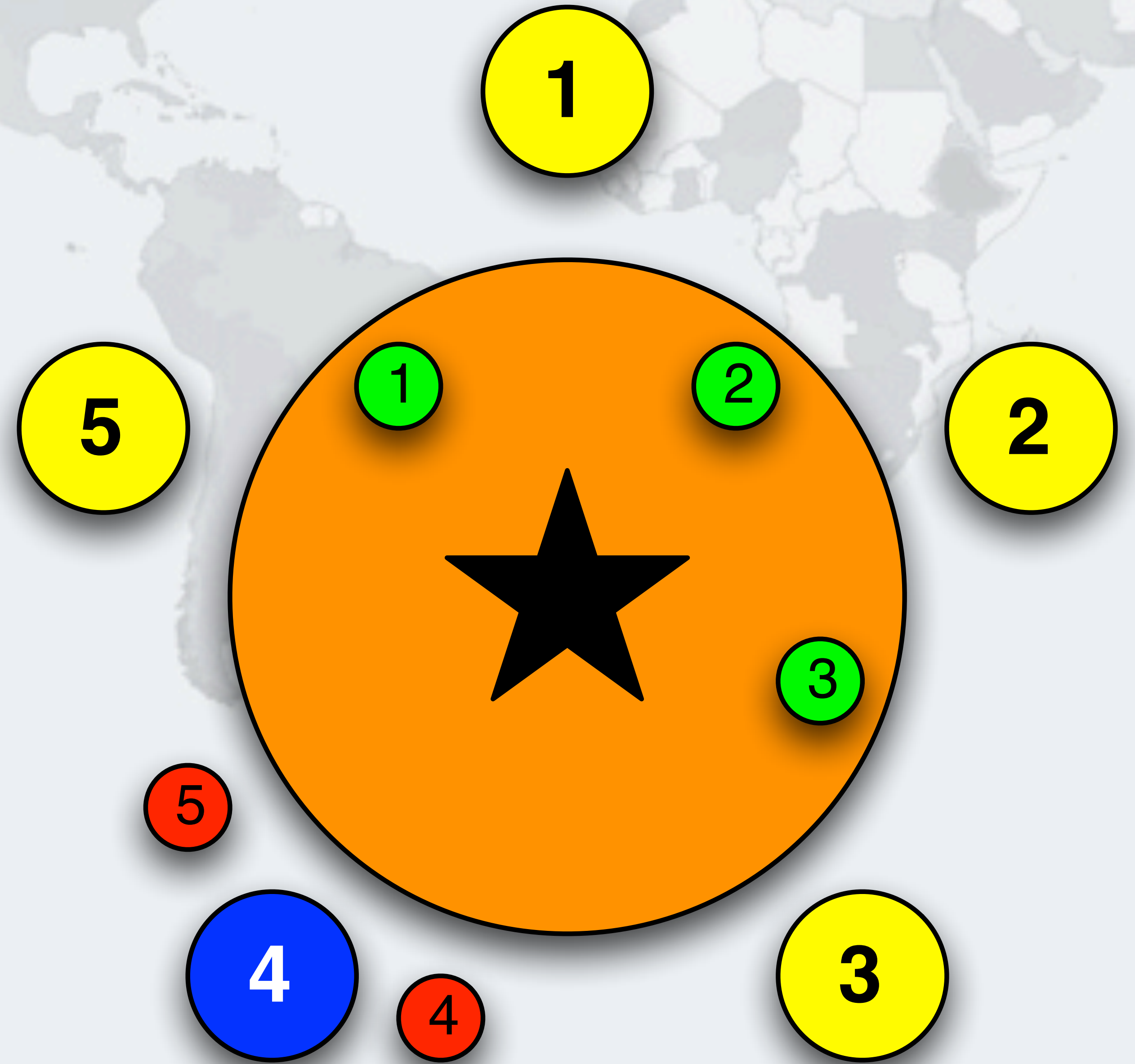
Philosopher 3 Returns Cup 3



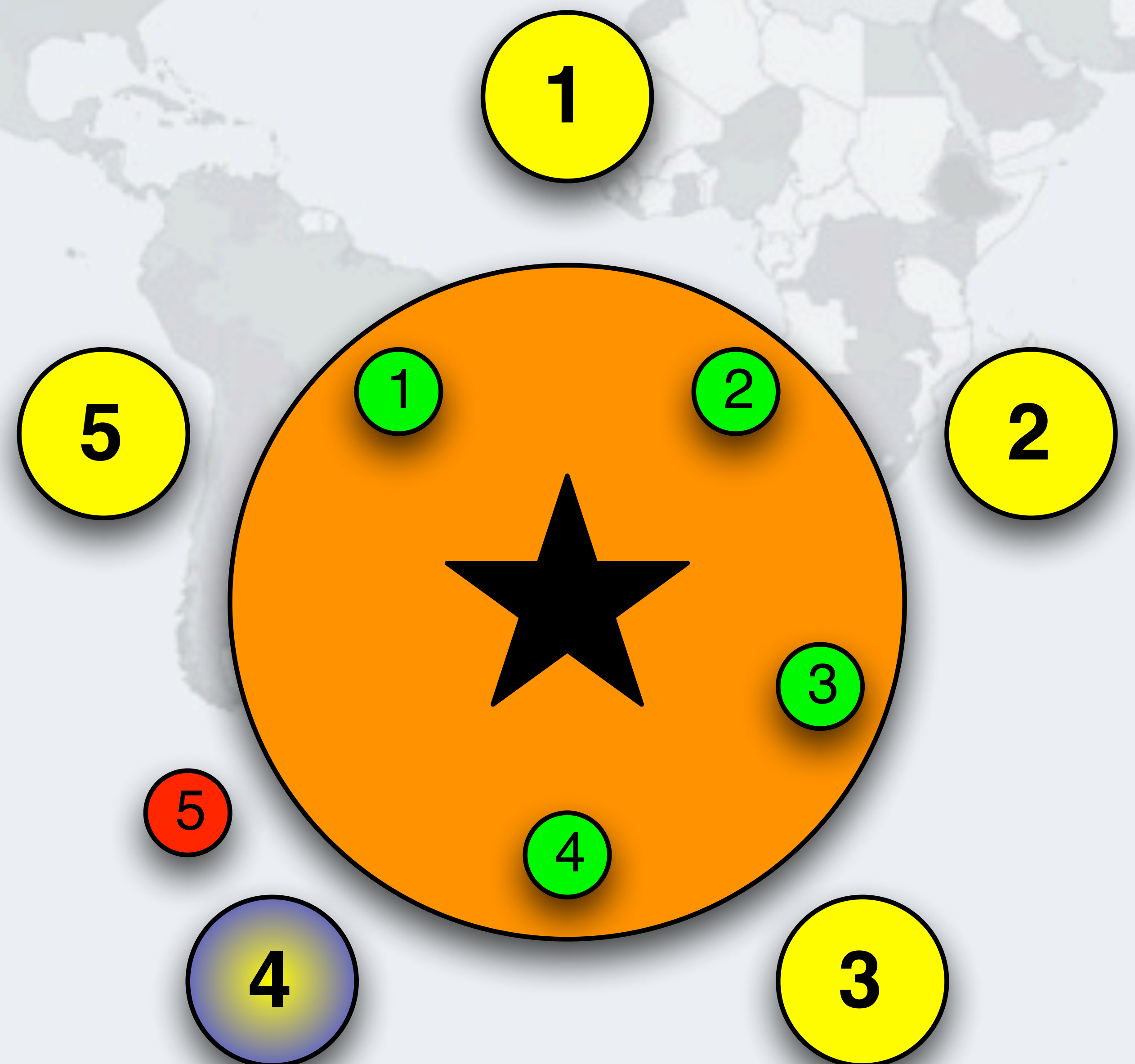
Philosopher 3 Returns Cup 4



Philosopher 4 Takes Cup 4 - Drinking

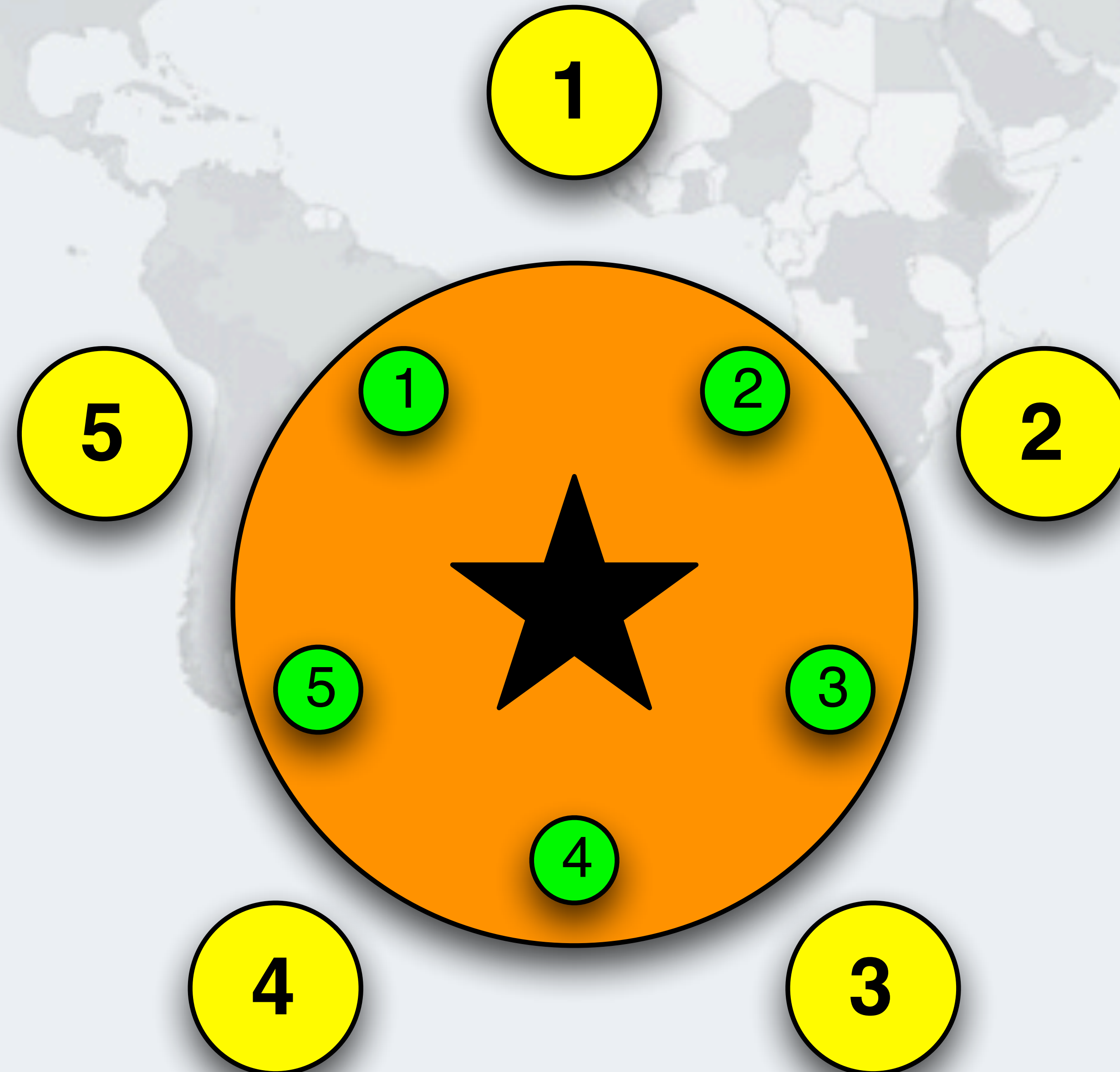


Philosopher 4 Returns Cup 4



Philosopher 4 Returns Cup 5

- **Deadlock free!**



Deadlock Is Avoided

- **Impossible for all philosophers to hold one cup**

Dynamic Lock Order Deadlocks

- **Often, it is not obvious what the lock instances are, e.g.**

```
public boolean transferMoney(
    Account from, Account to,
    DollarAmount amount) {
  synchronized (from) {
    synchronized (to) {
      return doActualTransfer(from, to, amount);
    }
  }
}
```

Causing The Deadlock With Transferring Money

- **Giorgos has accounts in Switzerland and in Greece**
  - **He keeps on transferring money between them**
    - **Whenever new taxes are announced, he brings money into Greece**
    - **Whenever he gets any money paid, he transfers it to Switzerland**
    - **Sometimes these transfers can coincide**
- **Thread 1 is moving money from UBS to Alpha Bank**

```
transferMoney(ubs, alpha, new DollarAmount(1000));
```

- **Thread 2 is moving money from Alpha Bank to UBS**

```
transferMoney(alpha, ubs, new DollarAmount(2000));
```

- **If this happens at the same time, it can deadlock**

Fixing Dynamic Lock-Ordering Deadlocks

- **The locks for transferMoney() are outside our control**
  - **They could be sent to us in any order**
- **We can *induce* an ordering on the locks**
  - **For example, we can use System.identityHashCode() to get a number representing this object**
    - **Since this is a 32-bit int, it is technically possible that two different objects have exactly the same identity hash code**
    - **In that case, we have a static lock to avoid a deadlock**

```
public boolean transferMoney(Account from, Account to,
                             DollarAmount amount) {
  int fromHash = System.identityHashCode(from);
  int toHash = System.identityHashCode(to);
  if (fromHash < toHash) {
    synchronized (from) {
      synchronized (to) {
        return doActualTransfer(from, to, amount);
      }
    }
  } else if (fromHash > toHash) {
    synchronized (to) {
      synchronized (from) {
        return doActualTransfer(from, to, amount);
      }
    }
  } else {
    synchronized (tieLock) {
      synchronized (from) {
        synchronized (to) {
          return doActualTransfer(from, to, amount);
        }
      }
    }
  }
}
```

Imposing Natural Order

- Instead of System.identityHashCode(), we define an order
  - Such as account number, employee number, etc.
  - Or an order defined for the locks used

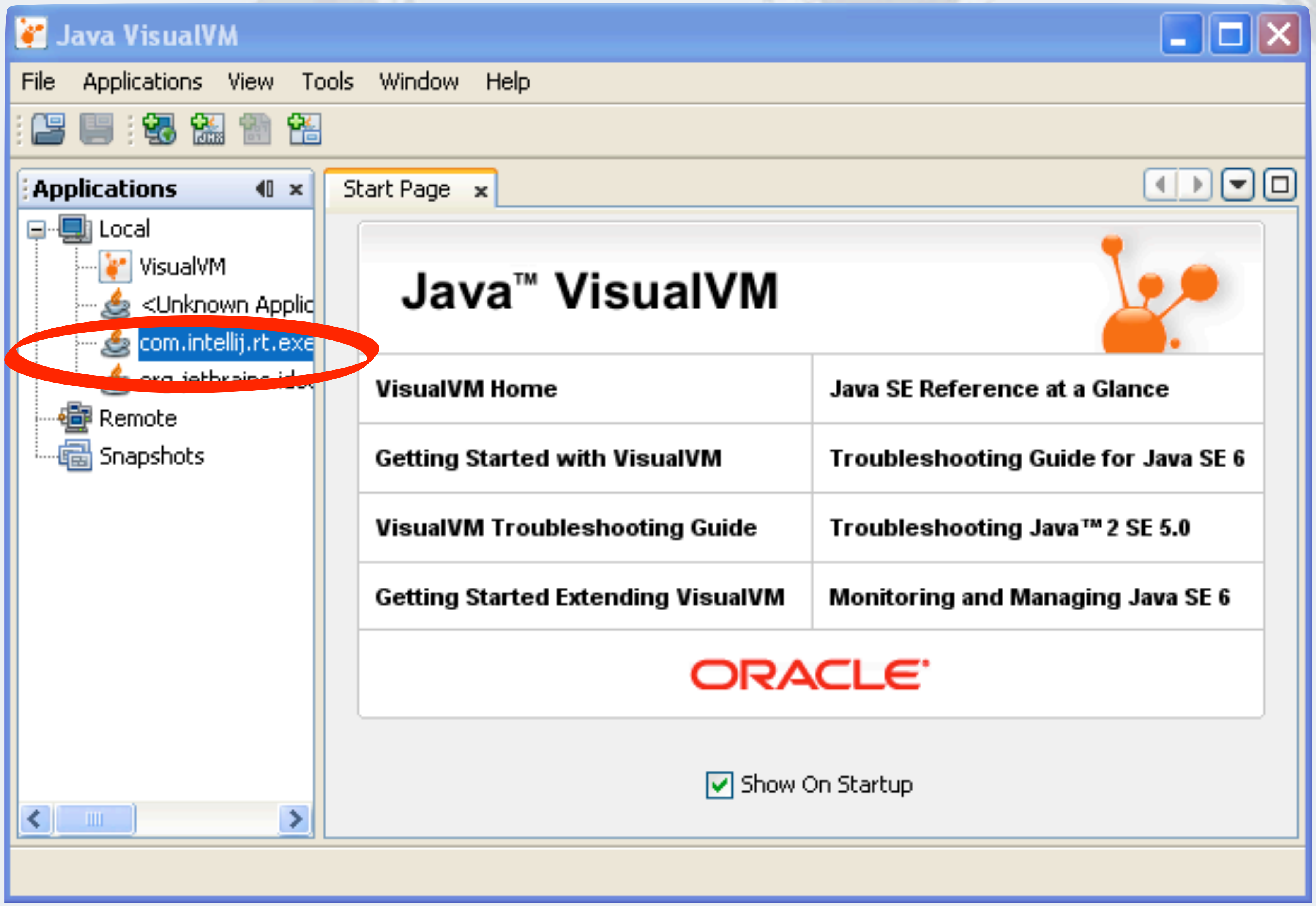
```java
public class MonitorLock implements Comparable<MonitorLock> {
  private static AtomicLong nextLockNumber = new AtomicLong();
  private final long lockNumber = nextLockNumber.getAndIncrement();

  public int compareTo(MonitorLock o) {
    if (lockNumber < o.lockNumber) return -1;
    if (lockNumber > o.lockNumber) return 1;
    return 0;
  }

  public static MonitorLock[] makeGlobalLockOrder(
      MonitorLock... locks) {
    MonitorLock[] result = locks.clone();
    Arrays.sort(result);
    return result;
  }
}
```

How To Find The Deadlocks

- **Deadlocks are almost always revealed in the thread dump**
- **They are not always shown as lock ordering deadlocks**
- **Often the deadlocks require some detective work**

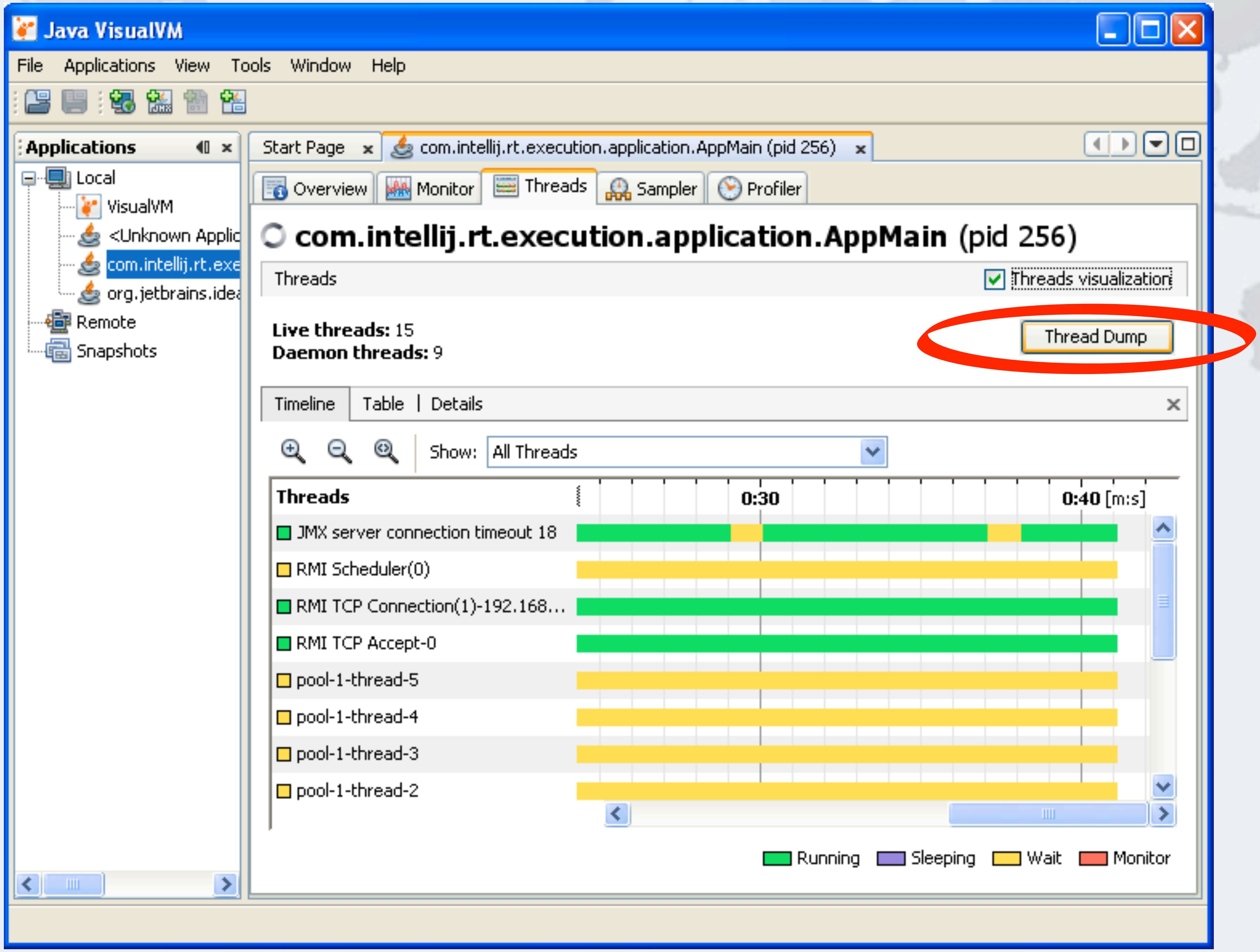
Capturing A Stack Trace

- **JVisualVM is a tool for monitoring what the JVM is doing**
  - **Found in the JDK/bin directory**
  - **Double-click on application**

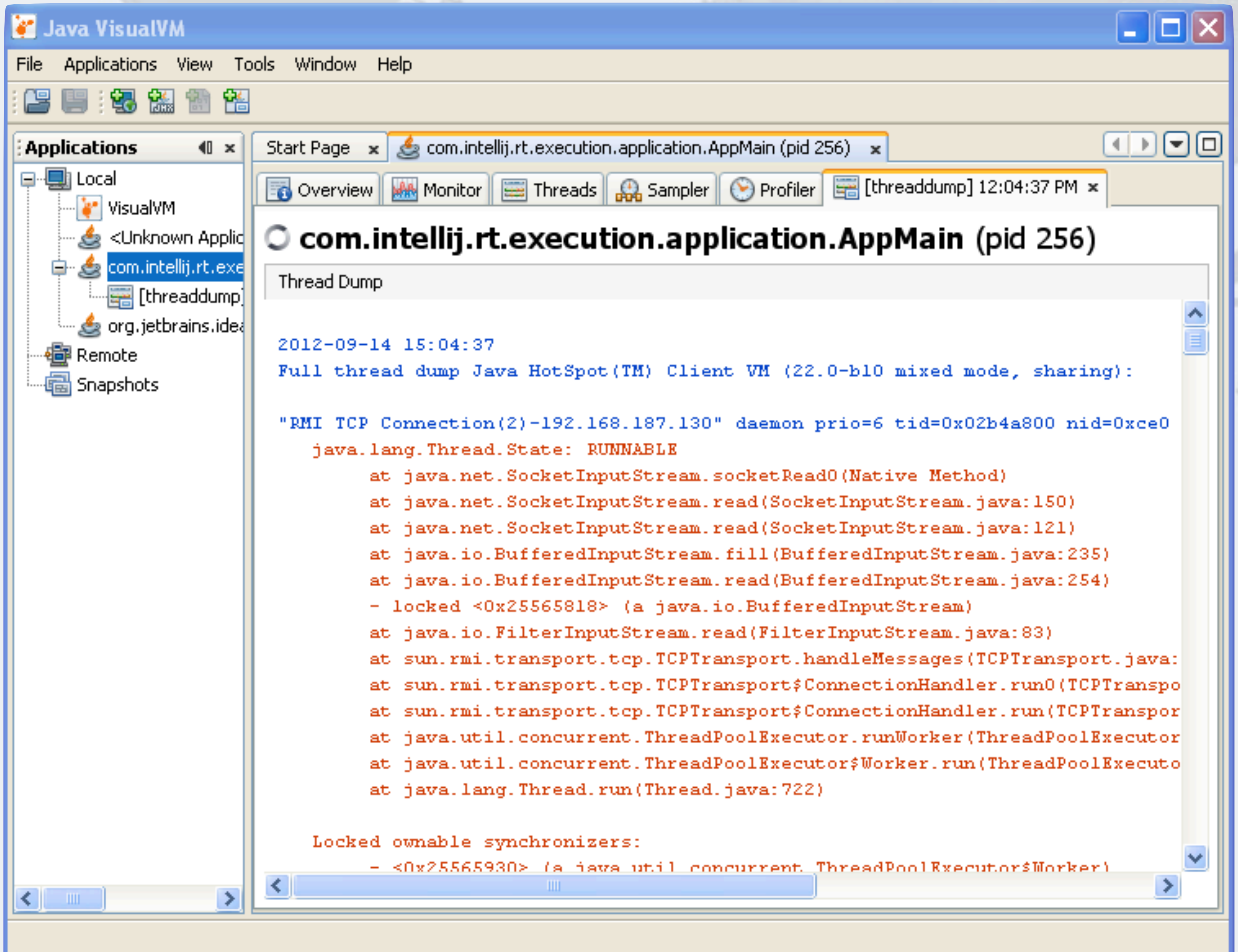


Click On "Threads" Tab

- **Click on "Thread Dump" button**

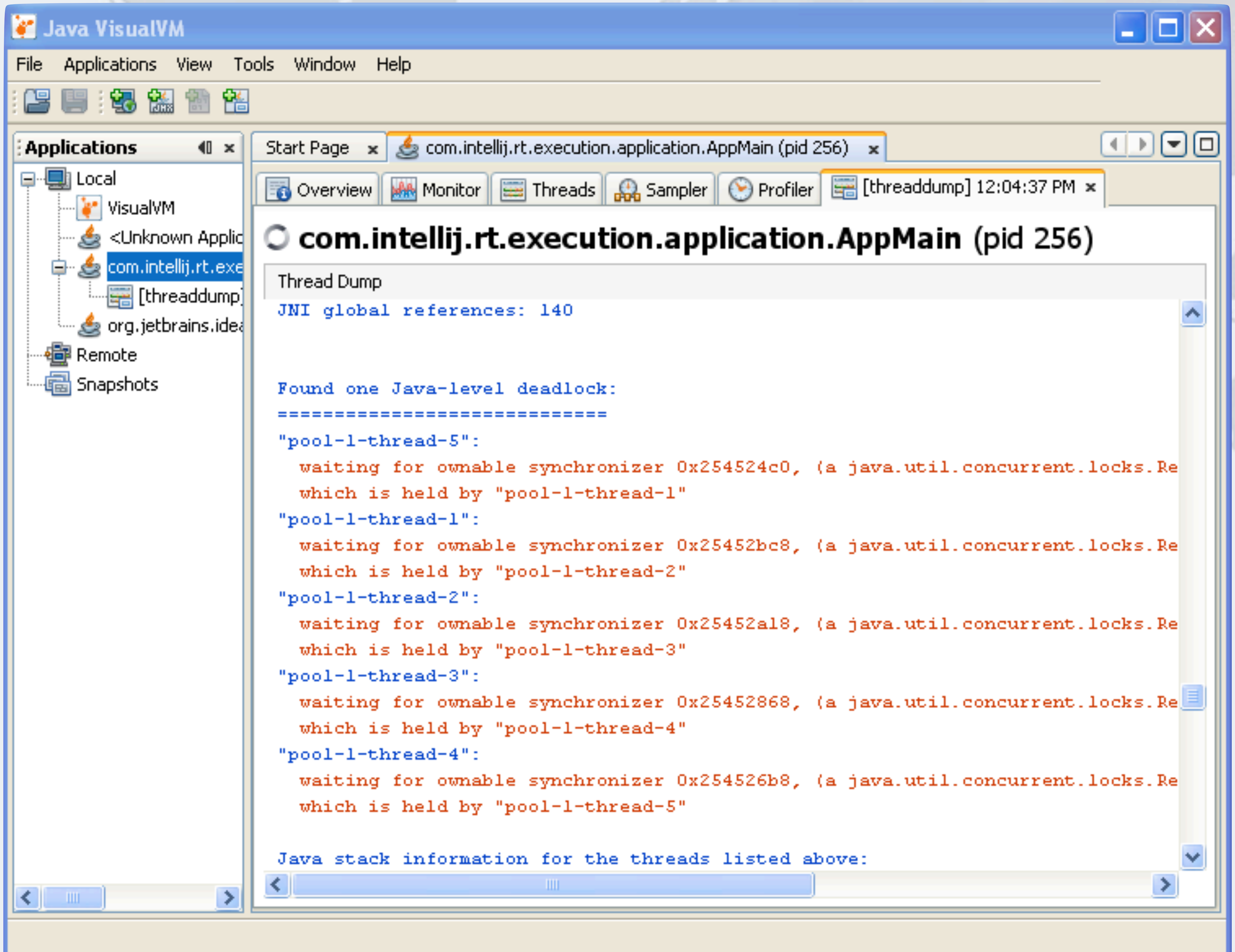


Stack Trace Shows What Threads Are Doing



Javaspecialists.eu

It Can Even Detect A Java-level Deadlock



Lab 1 Exercise

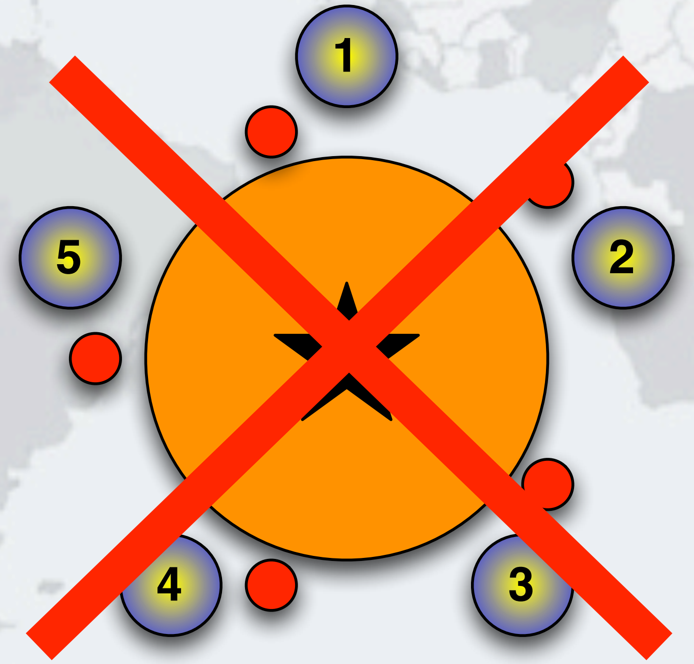
Deadlock resolution by global ordering

Lab1

- Run SymposiumTest class to trigger deadlock
  - You might need a few runs
- Define a global ordering for the locks that would prevent deadlock
  - We are synchronizing on the Krasi objects
  - Define a global ordering for Krasi objects by implementing Comparable and providing a unique number to sort on (Krasi.java)
  - Change the code to use the global ordering (Thinker.java)
  - Verify that the deadlock has now disappeared
- http://www.javaspecialists.eu/outgoing/jfokus2013.zip

Lab1 Exercise Solution Explanation

- **Goal: Prevent all philosophers from holding a single cup**



Lab1 Exercise Solution Explanation

- **Goal: Prevent all philosophers from holding a single cup**

Thinker	Cup 1 right	Cup 2 left
1	1	2
2	2	3
3	3	4
4	4	5
5	5	1

Thinker	Cup 1 big	Cup 2 small
1	2	1
2	3	2
3	4	3
4	5	4
5	5	1

- **The set of first cups is 2,3,4,5**
  - **This means that at most four philosophers can hold a single cup!**

Lab 2: Deadlock Resolution By TryLock

Avoiding Liveness Hazards

Javaspecialists.eu
java training

Lab 2: Deadlock Resolution By TryLock

- **Same problem as in Lab 1**
- **But our solution will be different**
- **Instead of a global order on the locks**
  - **We lock the first lock**
  - **We then try to lock the second lock**
    - **If we can lock it, we start drinking**
    - **If we cannot, we back out completely and try again**
      - **What about starvation or livelock?**

Lock And ReentrantLock

- **The Lock interface offers different ways of locking:**
  - **Unconditional, polled, timed and interruptible**

```java
public interface Lock {
  void lock();
  void lockInterruptibly() throws InterruptedException;
  boolean tryLock();
  boolean tryLock(long timeout, TimeUnit unit)
    throws InterruptedException;
  void unlock();
  Condition newCondition();
}
```

- **Lock implementations must have same memory-visibility semantics as intrinsic locks (synchronized)**

ReentrantLock Implementation

- **Like synchronized, it offers reentrant locking semantics**
- **Also, we can interrupt threads that are waiting for locks**
  - **Actually, the ReentrantLock never causes the thread to be BLOCKED, but always WAITING**
  - **If we try to acquire a lock unconditionally, interrupting the thread will simply go back into the WAITING state**
    - **Once the lock has been granted, the thread interrupts itself**

Using The Explicit Lock

- We have to call unlock() in a finally block
  - Every time, without exception
  - There are FindBugs detectors that will look for forgotten "unlocks"

```
private final Lock lock = new ReentrantLock();

public void update() {
  lock.lock(); // this should be before try
  try {
    // update object state
    // catch exceptions and restore
    // invariants if necessary
  } finally {
    lock.unlock();
  }
}
```

Synchronized Vs ReentrantLock

Explicit Locks

Synchronized vs ReentrantLock

- **ReentrantLock and intrinsic locks have the same memory semantics**
- **Reentrant locks can have polled locks, timed waits, interruptible waits and fairness**
  - **Performance of contended ReentrantLock was much better in Java 5**
- **However, intrinsic locks have significant advantages**
  - **Very few programmers structure the try-finally block correctly:**

```
lock.lock();
try {
  // do operation
} finally {
  lock.unlock();
}
```

```
synchronized(this)
  // do operation
}
```

Bad Try-Finally Blocks

- **Either no try-finally at all**

```
lock.lock();
// do operation
lock.unlock();
```

- **Or the lock is locked inside the try block**

```
try {
  lock.lock();
  // do operation
} finally {
  lock.unlock();
}
```

- **Or the unlock() call is forgotten in some places altogether!**

```
lock.lock();
// do operation
```

When To Use ReentrantLock

- **Use it when you need**
  - lock.tryLock()
  - lock.tryLock(timeout)
  - lock.lockInterruptibly()
  - fair locks
  - Multiple condition variables for one lock
- **Otherwise, prefer synchronized**

Deadlock Monitoring

- **Java 5 deadlock detection only works with synchronized**
- **In Java 6, it works with Lock and synchronized**
  - **– However, timed locks can be incorrectly detected as deadlocked**

Polled Lock Acquisition

- **Instead of unconditional lock, we can tryLock()**

```
if (lock.tryLock()) {
  try {
    balance = balance + amount;
  } finally {
    lock.unlock();
  }
} else {
  // alternative path
}
```

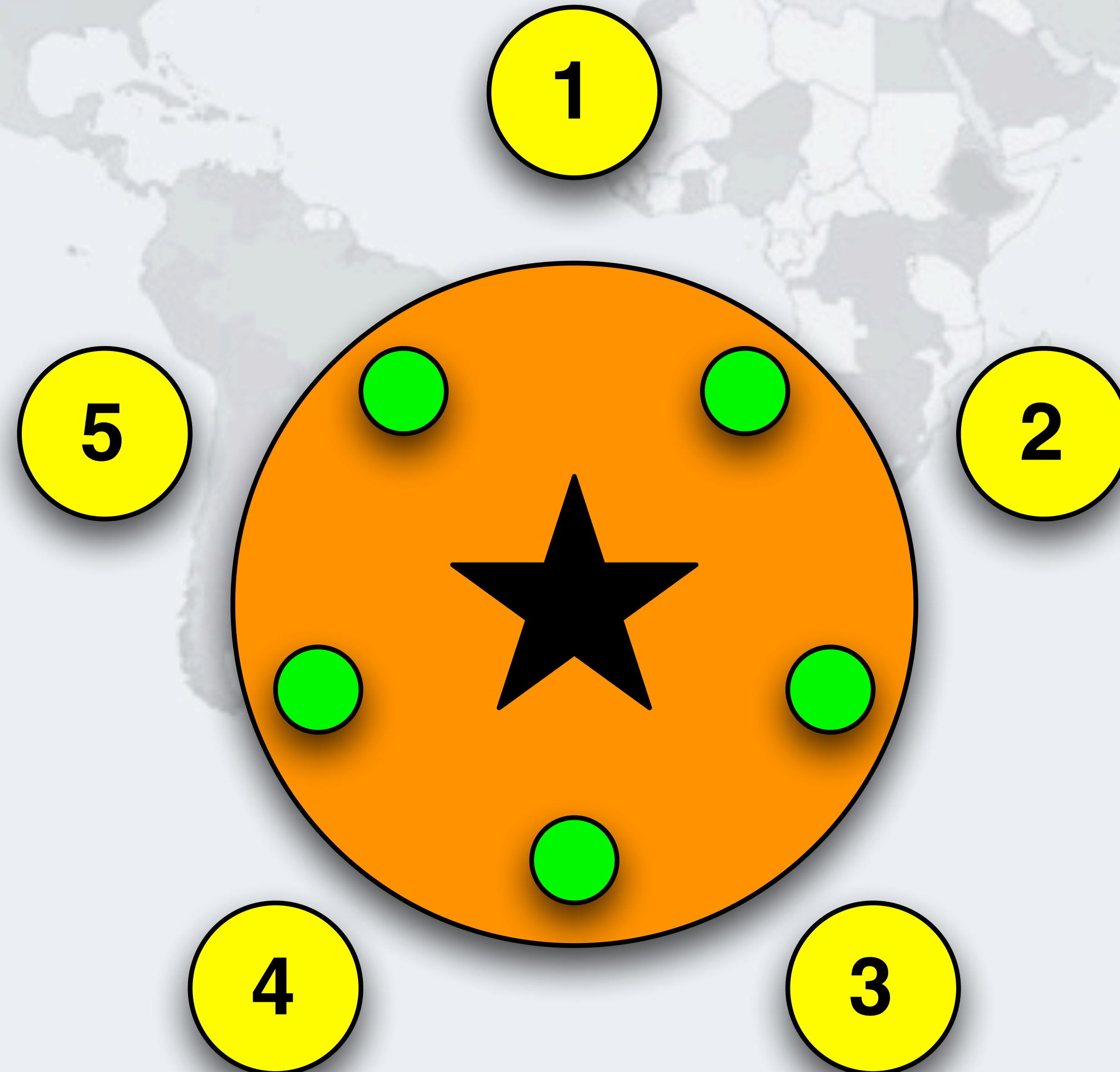
Using Try-Lock To Avoid Deadlocks

- **Deadlocks happen when we lock multiple locks in different orders**
- **We can avoid this by using tryLock()**
  - **If we do not get lock, sleep for a random time and then try again**
  - **Must release *all* held locks, or our deadlocks become livelocks**
- **This is possible with synchronized, see my newsletter**
  - **http://www.javaspecialists.eu/archive/Issue194.html**

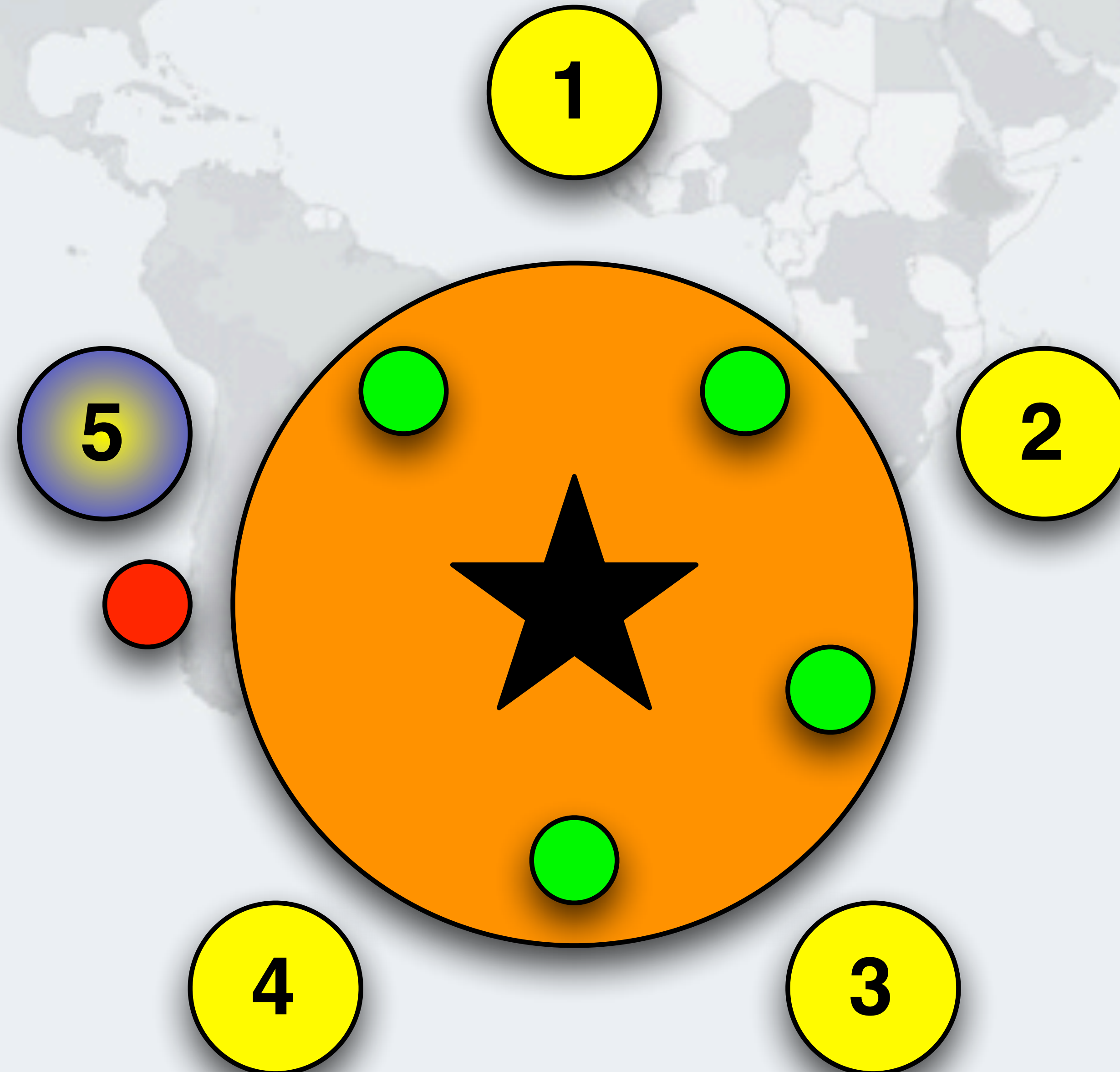
Using TryLock() To Avoid Deadlocks

```
public void drink() {
  while (true) {
    right.lock();
    try {
      if (left.tryLock()) {
        try {
          // now we can finally drink and then return
        } finally {
          left.unlock();
        }
      }
    } finally {
      right.unlock();
    }
    // sleep for a random time
  }
}
```

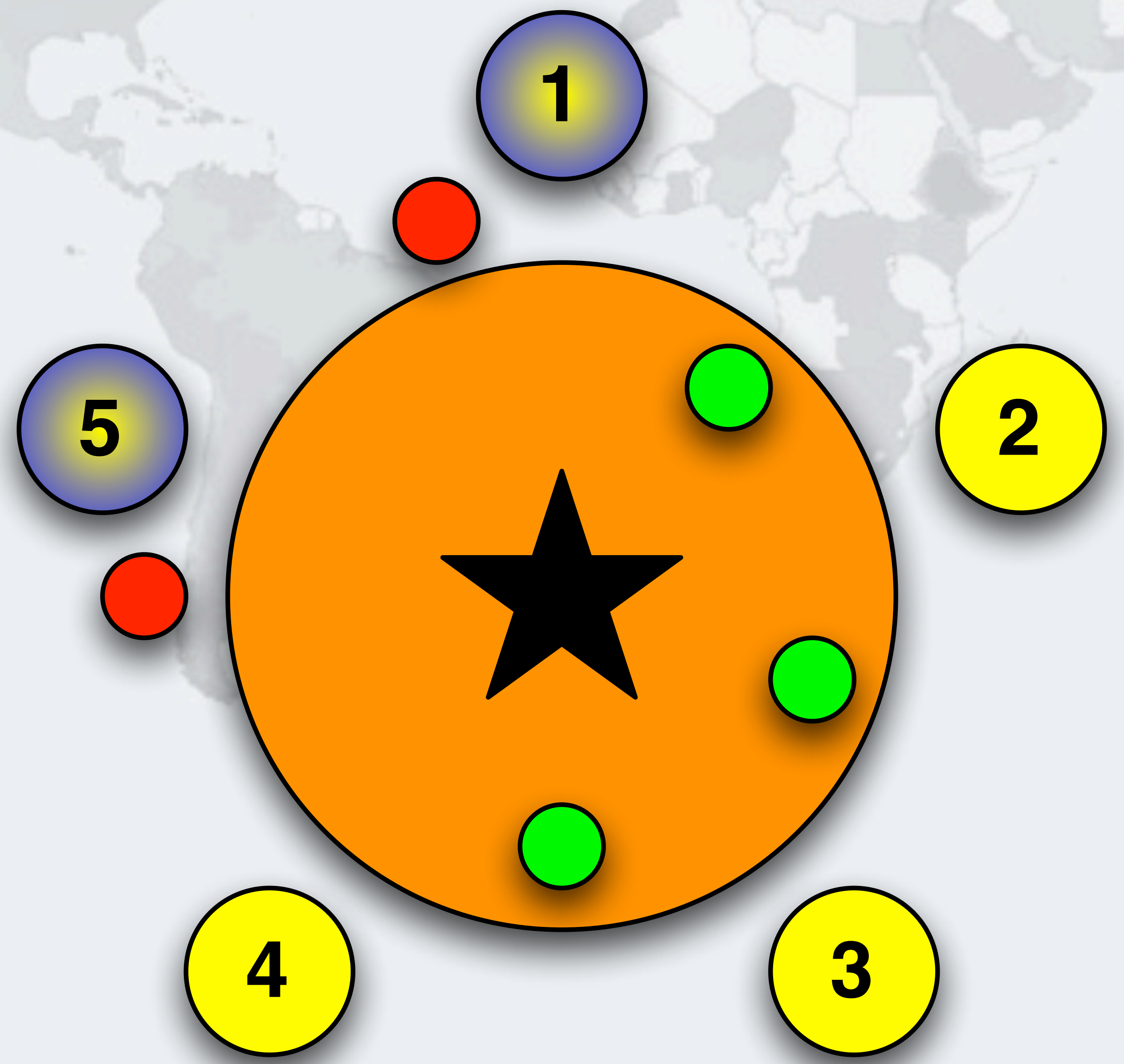
Deadlock Is Prevented In This Design



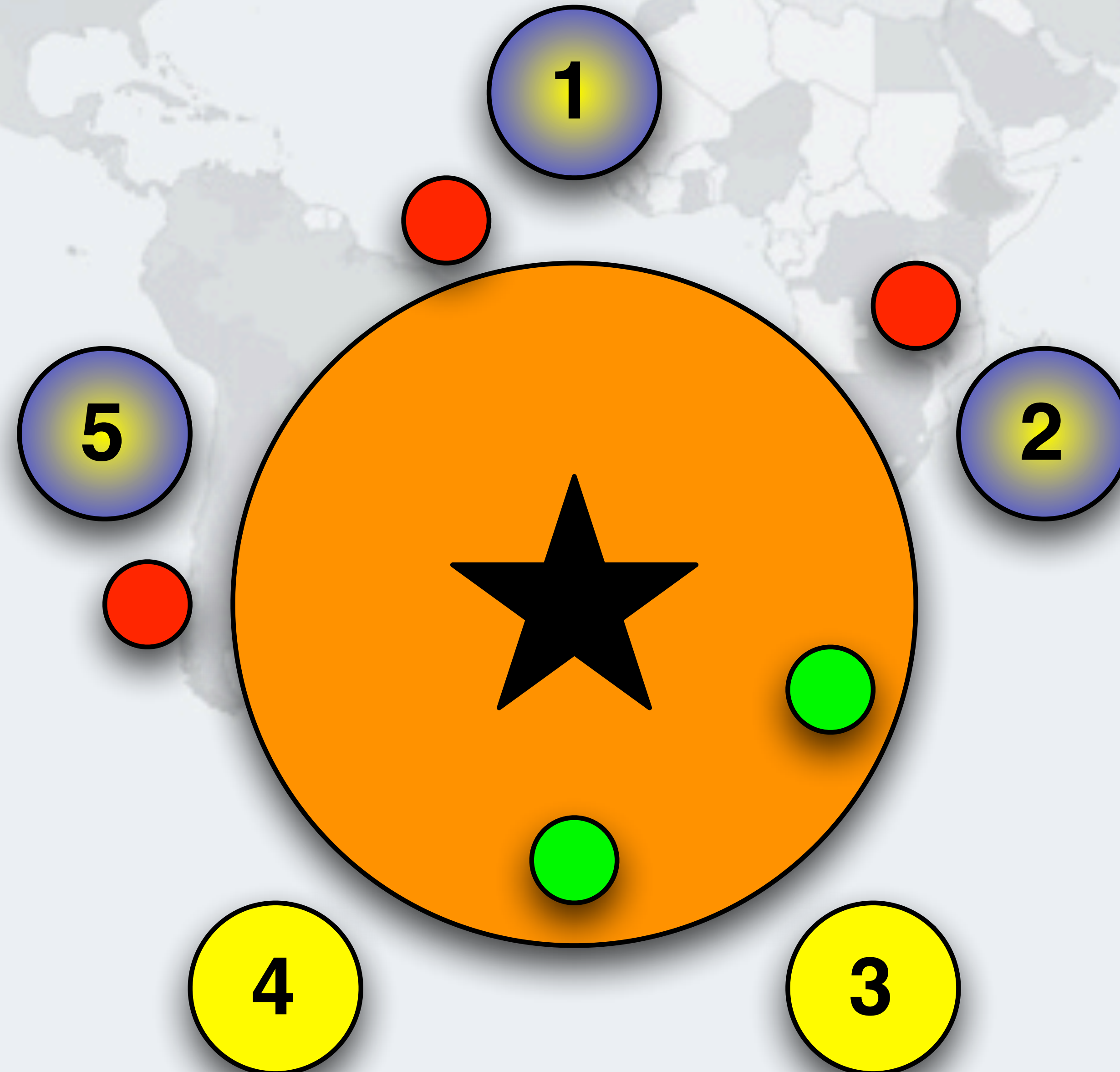
Philosopher 5 Wants To Drink, Takes Right Cup



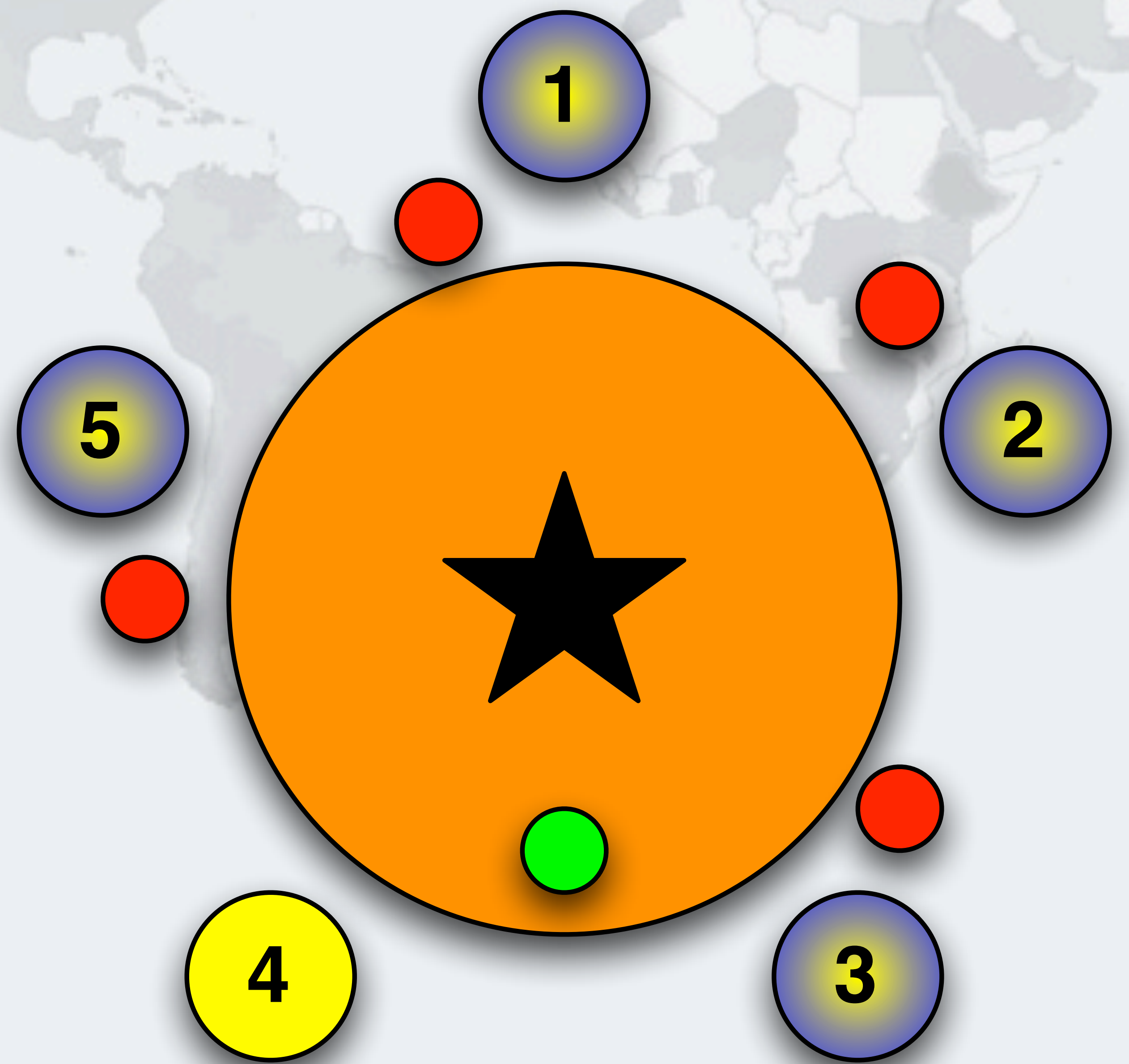
Philosopher 1 Wants To Drink, Takes Right Cup



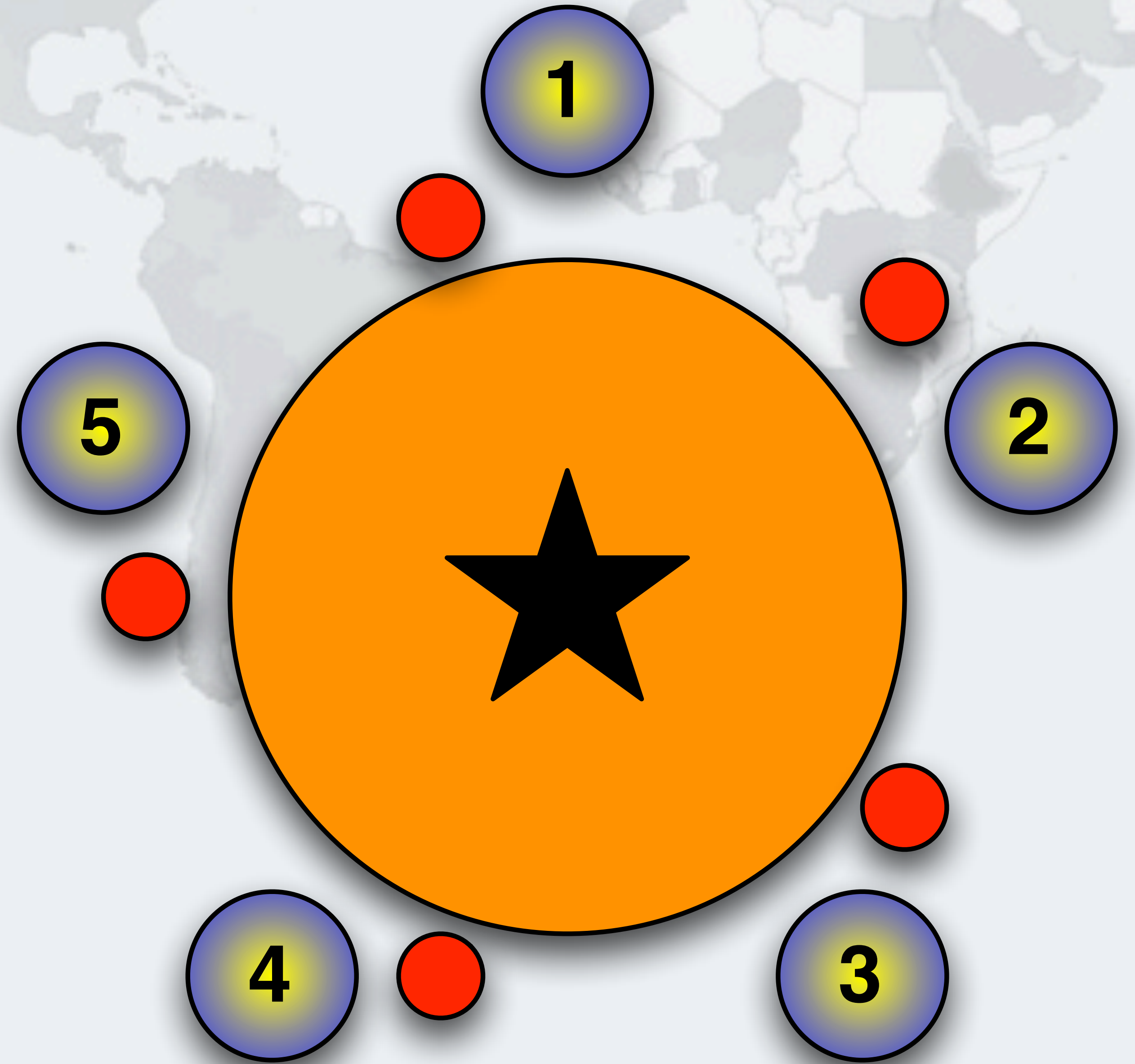
Philosopher 2 Wants To Drink, Takes Right Cup



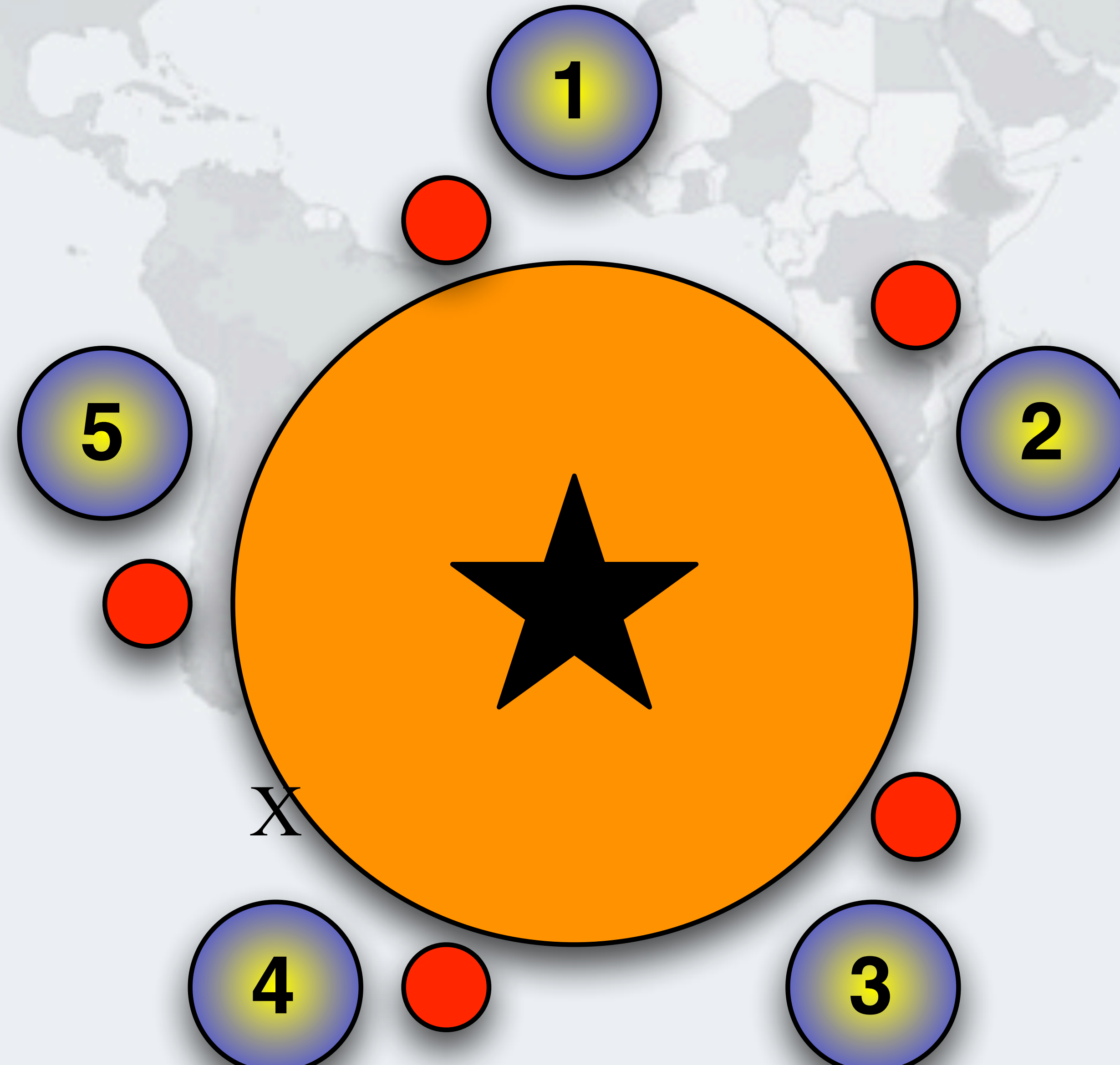
Philosopher 3 Wants To Drink, Takes Right Cup



Philosopher 4 Wants To Drink, Takes Right Cup

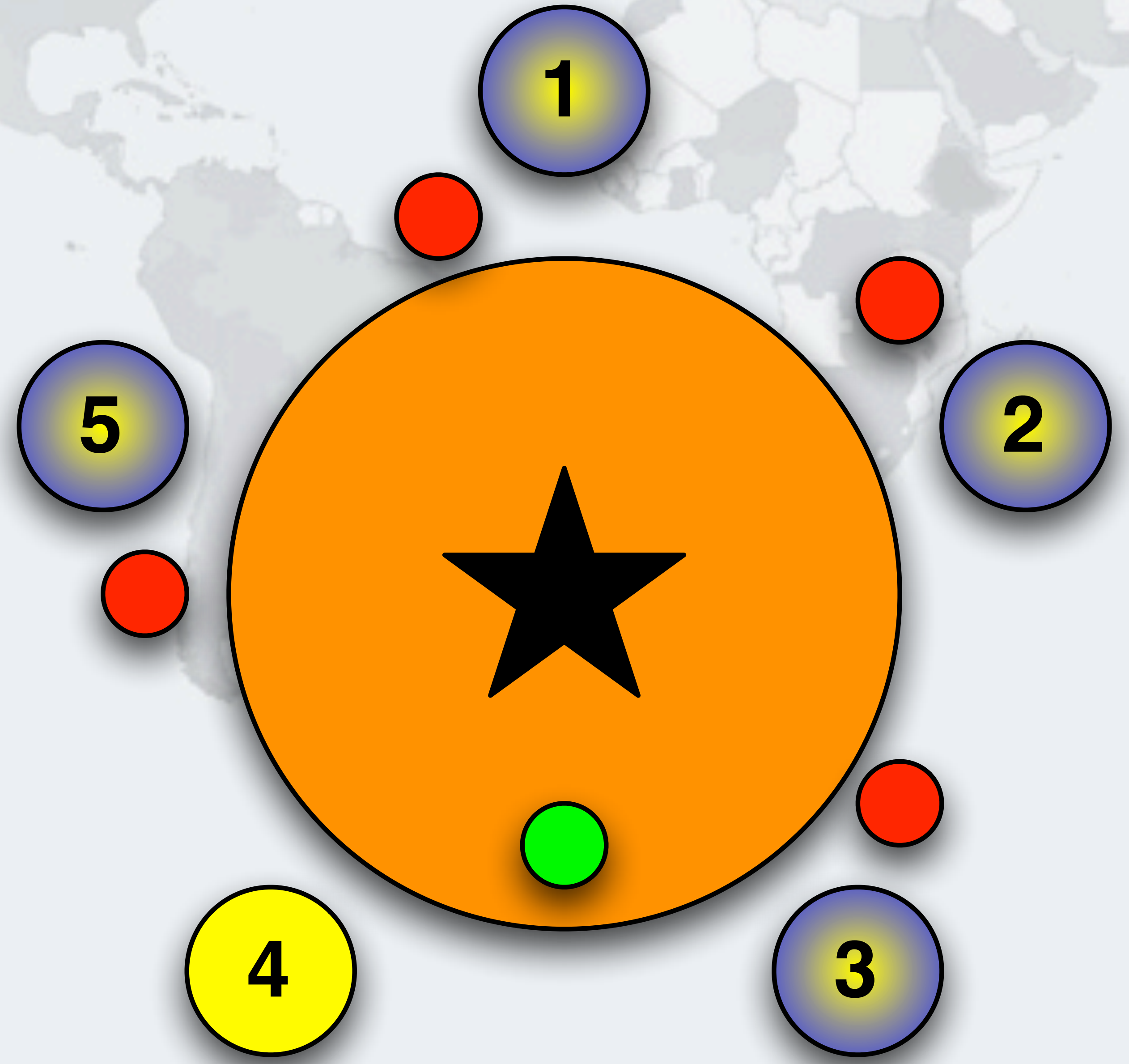


Philosopher 4 Tries To Lock Left, Not Available



Philosopher 4 Unlocks Right Again

- **Now Philosopher 3 can drink**



Lab 2 Exercise

Deadlock resolution by tryLock

Lab2

- **Run SymposiumTest class to trigger deadlock**
  - **You might need a few runs**
- **Use Lock.tryLock() to avoid blocking on the inner lock**
  - **lock the right**
  - **tryLock the left**
    - **if success, then drink and unlock both**
    - **otherwise, unlock right and retry**
  - **Change the Thinker.java file**
  - **Verify that the deadlock has now disappeared**
- **http://www.javaspecialists.eu/outgoing/jfokus2013.zip**

Lab2 Solution Explanation

- **Goal: Prevent all philosophers from forever blocking on the second cup**
  - **A philosopher should not die of thirst**
    - **We need to avoid livelocks**
    - **lock/tryLock vs. tryLock/tryLock**

Lab 3: Resource Deadlock

Avoiding Liveness Hazards

Lab 3: Resource Deadlock

- Problem: threads are blocked waiting for a finite resource that never becomes available
- Examples:
  - Resources not being released after use
    - Running out of threads
    - Java Semaphores not being released
  - JDBC transactions getting stuck
  - Bounded queues or thread pools getting jammed up
- Challenge:
  - Does not show up as a Java thread deadlock
  - Problem thread could be in any state: RUNNING, WAITING, BLOCKED

How To Solve Resource Deadlocks

- **Approach: If you can reproduce the resource deadlock**
  - **Take a thread snapshot shortly before the deadlock**
  - **Take another snapshot after the deadlock**
  - **Compare the two snapshots**
- **Approach: If you are already deadlocked**
  - **Take a few thread snapshots and look for threads that do not move**
- **It is useful to identify the resource that is being exhausted**
  - **A good trick is via phantom references (beyond scope of this lab)**

Resource Deadlocks

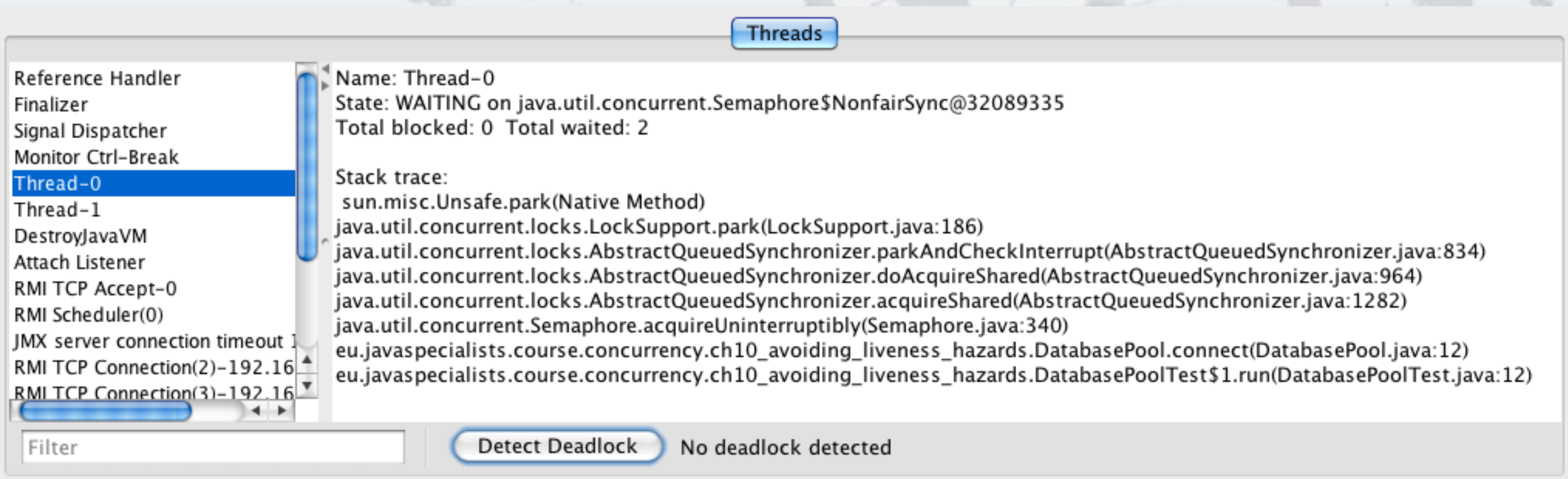
- We can also cause deadlocks waiting for resources
- For example, say you have two DB connection pools
  - Some tasks might require connections to both databases
  - Thus thread A might hold semaphore for D1 and wait for D2, whereas thread B might hold semaphore for D2 and be waiting for D1
- Thread dump and ThreadMXBean does not show this as a deadlock!

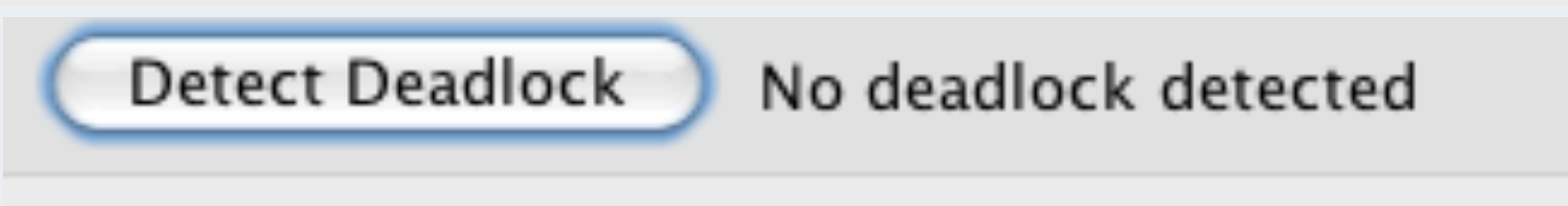
Our DatabasePool - Connect() And Disconnect()

```
public class DatabasePool {
  private final Semaphore connections;
  public DatabasePool(int connections) {
    this.connections = new Semaphore(connections);
  }

  public void connect() {
    connections.acquireUninterruptibly();
    System.out.println("DatabasePool.connect");
  }

  public void disconnect() {
    System.out.println("DatabasePool.disconnect");
    connections.release();
  }
}
```

ThreadMXBean Does Not Detect This Deadlock

```
DatabasePool.connect
DatabasePool.connect
```





Stack Trace Gives A Vector Into The Code

```
locks.AbstractQueuedSynchronizer.doAcquireShared(AbstractQueuedSynchronizer.java:964)
locks.AbstractQueuedSynchronizer.acquireShared(AbstractQueuedSynchronizer.java:1282)
Semaphore.acquireUninterruptibly(Semaphore.java:340)
ourse.concurrency.ch10_avoiding_liveness_hazards.DatabasePool.connect(DatabasePool.java:12)
```

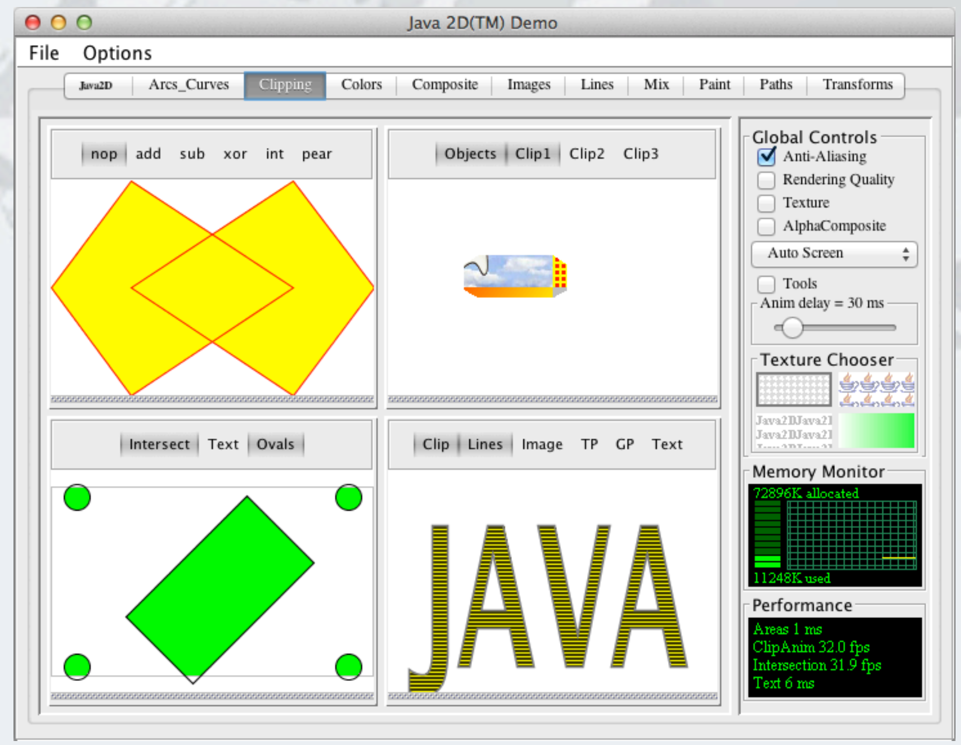
```java
public class DatabasePool {
  // ...

  public void connect() {
    connections.acquireUninterruptibly(); // line 12
    System.out.println("DatabasePool.connect");
  }
}
```

Lab 3 Exercise

Resource Deadlock

Lab3 Exercise Lab3/readme.txt

- **Start our modified Java2Demo**
  - **Connect JVisualVM and dump all threads**
  - **Use Java2Demo for a while until it deadlocks**
  - **Get another thread dump and compare to the first one**
    - **This should show you where the problem is inside your code**
  - **Fix the problem and verify that it has been solved**
    - **Hint: Your colleagues probably write code like this, but you shouldn't**



Lab3 Exercise Solution Explanation

- **Goal: Ensure that resources are released after use**
- **Diff between the two thread dumps using jps and jstack**

< at java.util.concurrent.locks.AbstractQueuedSynchronizer
$ConditionObject.await(AbstractQueuedSynchronizer.java:2043)
< at java.awt.EventQueue.getNextEvent(EventQueue.java:531)
< at java.awt.EventDispatchThread.pumpOneEventForFilters(EventDispatchThread.java:213)

> at
java.util.concurrent.locks.AbstractQueuedSynchronizer.parkAndCheckInterrupt(AbstractQueuedSynchronizer.java:
834)
> at
java.util.concurrent.locks.AbstractQueuedSynchronizer.doAcquireSharedInterruptibly(AbstractQueuedSynchronizer
.java:994)
> at – **Most likely the fault will be in one of our classes, rather than the JDK**
java.util.concurrent.locks.AbstractQueuedSynchronizer.acquireSharedInterruptibly(AbstractQueuedSynchronizer.ja
va:1303)
> at java.util.concurrent.Semaphore.acquire(Semaphore.java:317)
> at
eu.javaspecialists.deadlock.lab3.java2d.MemoryManager.gc(MemoryManager.j

What Is Wrong With This Code?

```
/**
 * Only allow a maximum of 30 threads to call System.gc() at a time.
 */
public class MemoryManager extends Semaphore {
  private static final int MAXIMUM_NUMBER_OF_CONCURRENT_GC_CALLS = 30;

  public MemoryManager() {
    super(MAXIMUM_NUMBER_OF_CONCURRENT_GC_CALLS);
  }

  public void gc() {
    try {
      acquire();
      try {
        System.gc();
      } finally {
        System.out.println("System.gc() called");
        release();
      }
    } catch (Exception ex) {
      // ignore the InterruptedException
    }
  }
}
```

Calling System.gc() is baddd (but not **the** problem)

Empty catch block hides problem

Javaspecialists.eu

Lab 4: Unsolvable Deadlocks

Avoiding Liveness Hazards

Javaspecialists.eu
java training

Lab 4: Unsolvable Deadlocks

- **Problem: Sometimes, things go wrong in your application that you cannot explain**
- **Challenge: You need to see if you can get the application to stop and then use the thread dumps to solve the problem**

Lab 4 Exercise

Resource Deadlock

Javaspecialists.eu
java training

Lab4

- **You are on your own**

Wrap Up

Avoiding Liveness Hazards

Conclusion On Deadlocks

- **Concurrency is difficult, but there are tools and techniques that we can use to solve problems**
- **These are just a few that we use**
- **For more information, have a look at**
  - **The Java Specialists' Newsletter - http://www.javaspecialists.eu**
- **We have helped a lot of companies by training their Java programmers**
  - **Java Concurrency**
  - **Java Performance Tuning**
  - **Java Design Patterns**
  - **Advanced Java Techniques (Java NIO, threading, data structs, etc.**

And One More Thing

- **We have prepared a fourth lab for you to do at home**
  - **Either take it along with a memory stick or get it from**
  - **http://www.javaspecialists.eu/outgoing/jfokus2013.zip**
  - **Send questions and comments to heinz@javaspecialists.eu**

Questions?

heinz@javaspecialists.eu

Javaspecialists.eu java training