Finding And Solving Deadlocks In Multi-Threaded Java Code

In Cooperation With ExitCertified

Dr Heinz M. Kabutz

Last updated 2012-07-20

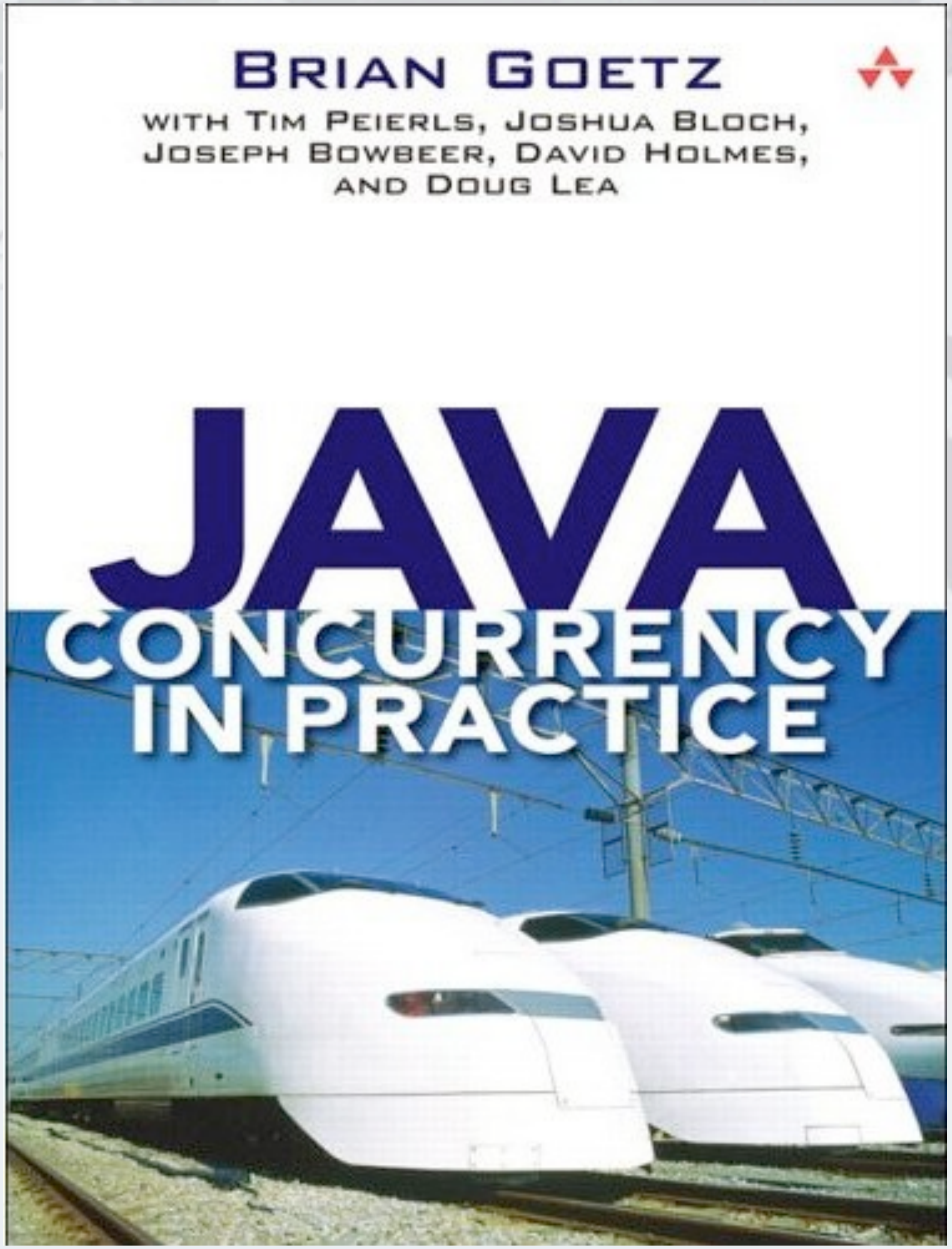
Copyright Notice

Short Introduction To Course Authors

- **Dr Heinz Kabutz**
  - Born in Cape Town, South Africa, now lives in Greece / Europe
  - Created The Java Specialists' Newsletter
    - http://www.javaspecialists.eu/archive/archive.html
  - One of the first Sun Java Champions
    - https://java-champions.dev.java.net
- **Victor Grazi**
  - Former salesman from New York
    - Realized early on that programming was more fun than selling!
  - Core Java Development at Credit Suisse Client Technology Services
  - One of the newest Oracle Java Champions
  - Creator of Java Concurrent Animated www.jconcurrency.com

Short Introduction To Brian Goetz

- **Brian Goetz wrote seminal masterpiece "Java Concurrency in Practice"**
  - **Our recommended book for Java concurrency**
  - **Course uses this as a basis**
- **Now is Oracle's "Java Language Architect"**
- **Most thorough text on how to deal with Java concurrency in everyday work**

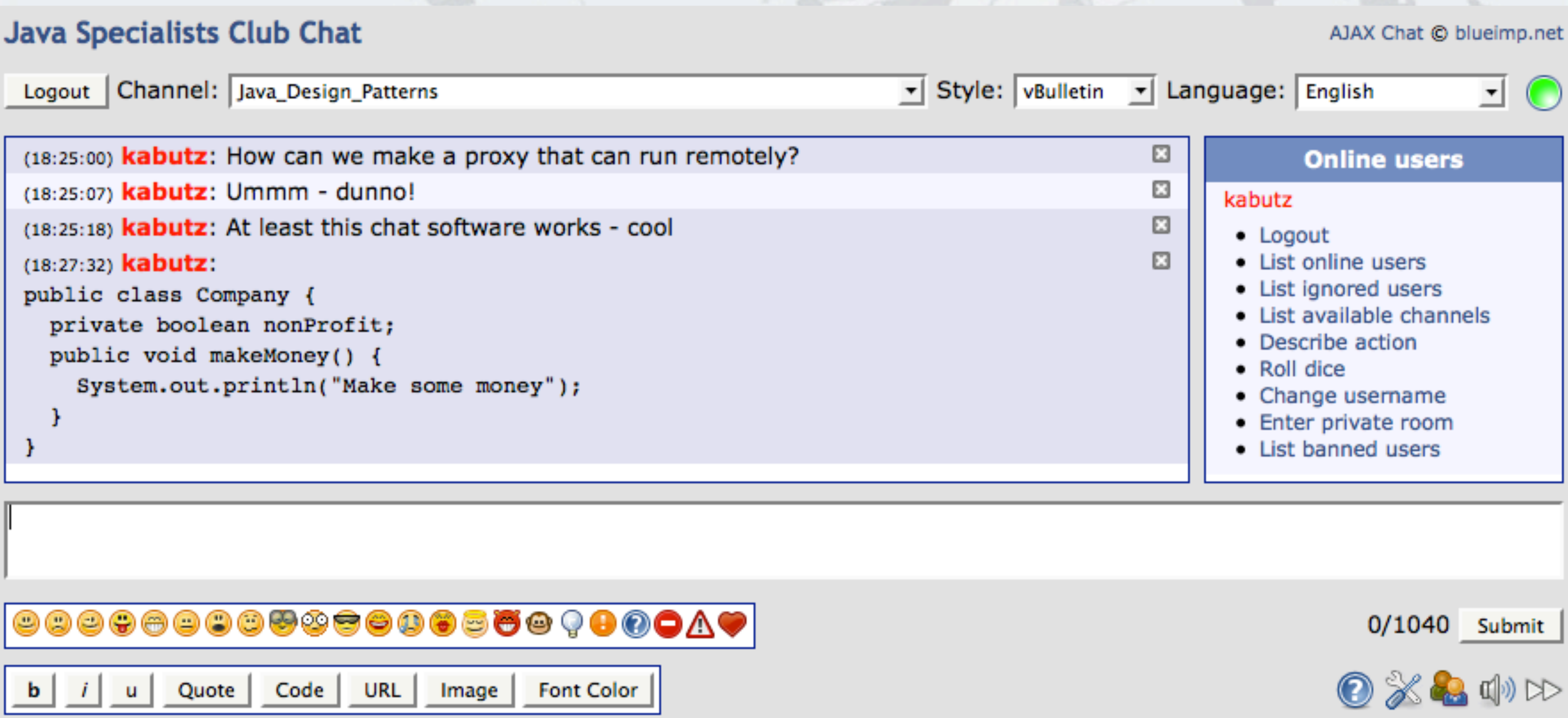


Workshop Structure

- 2 x 50 minute lectures, with break in between
- 1 x 50 minute lab, where you get to solve a liveness issue
  - Exact time depends on how quick you are
  - Download it from here: http://tinyurl.com/conc-zip
- Your workshop page:
  - http://javaspecialists.eu/courses/concurrency/exitcertified.jsp

Chat Room

- **http://www.javaspecialists.eu/forum/chat/**
  - **We will be in the "Public" channel**



Who Are The Participants

- **Skill level**
  - 31 either complete beginners or no practical experience
  - 124 intermediate
  - 71 advanced programmers
  - 3 super advanced
    - Two of which end their surname in "ev"
  - 38 unspecified
- **Our focus will be mainly on the intermediate and advanced programmers**
  - Will give an introduction to threading, what it is and why we need it

A Boat Called "Java"

- **In Greek, the Latin "J" is translated as "TZ" and "V" as "B"**
  - **So we get TZABA**

1: Introduction

Questions

- **Please please please please ask questions!**
- **Interrupt me at any time**
  - **Type it into chat: http://www.javaspecialists.eu/forum/chat/**
  - **Or put up your hand (little hand icon) and I will unmute you**
    - **Make sure your microphone volume is turned up**
- **There are some stupid questions**
  - **They are the ones you didn't ask**
  - **Once you've asked them, they are not stupid anymore**
- **The more you ask, the more we all learn**

The Concurrency Specialist Course

- **Course Contents**
  - Introduction
  - Thread Safety
  - Sharing Objects
  - Composing Objects
  - Building Blocks
  - Task Execution
  - Cancellation and Shutdown
  - Applying Thread Pools
  - SwingWorker and Fork/Join
  - Avoiding Liveness Hazards
  - Performance and Scalability
  - Testing Concurrent Programs
  - Building Custom Synchronizers
- **http://www.javaspecialists.eu/courses/concurrency.jsp**

Multiple Processes

- **Time slicing allows us to run many programs at once**
  - **Illusion; our O/S swaps between different processes very quickly**
- **Each process typically runs in its own memory space**
  - **Inter-process communication is expensive**

Why Use Threads?

- Threads are software abstractions to help us utilize the available hardware
- Threads are like lightweight processes, sharing the same memory space
- Quick for scheduler to swap between threads
- Performance can improve if we utilize all the cores
- Threading can also simplify coding
  - Our systems can be written with better OO principles
  - Independent workflows do not have to know about each other

Let's Go Fast Fast Fast

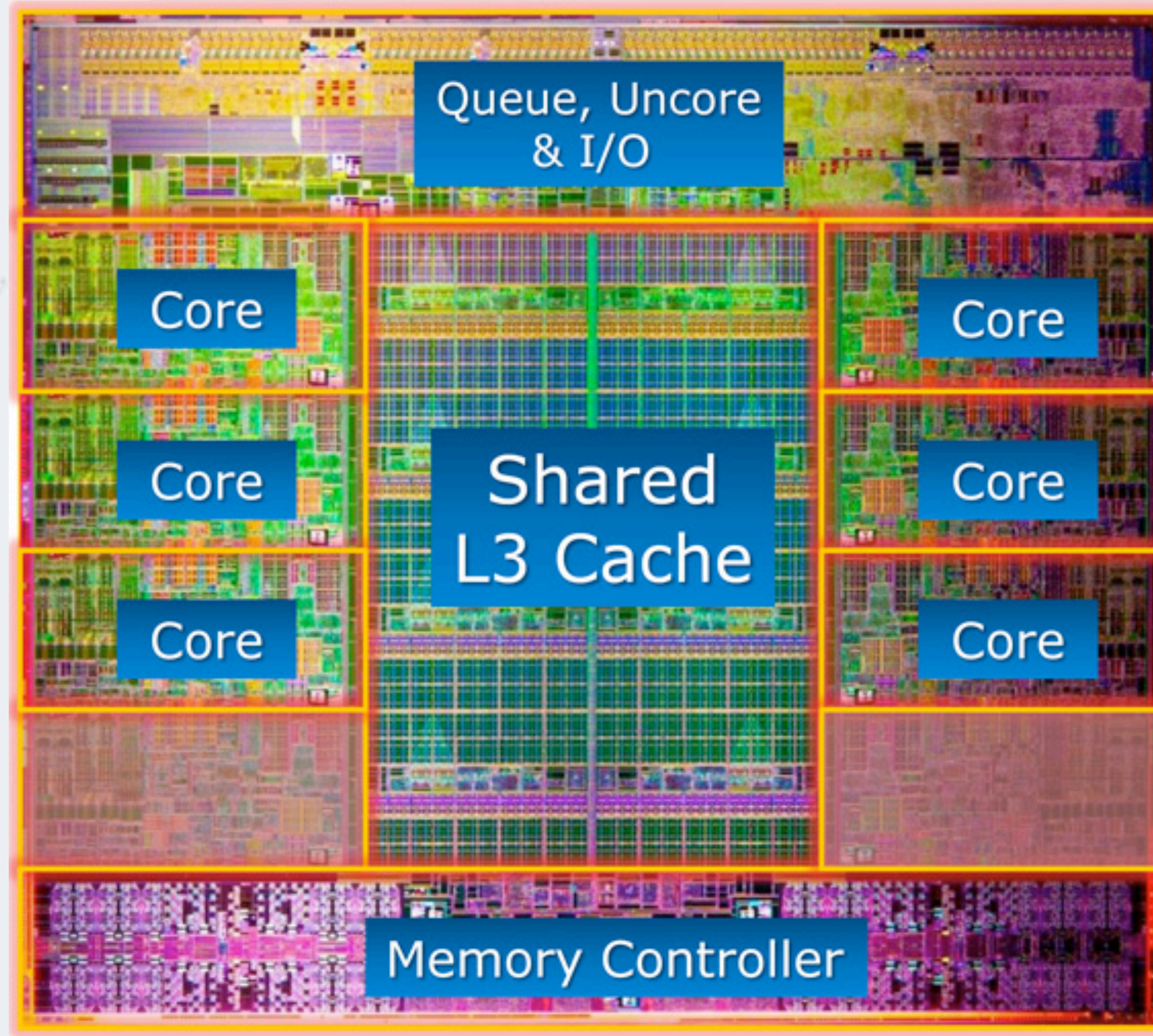
- **In 2000, Intel predicted 10GHz chips on desktop by 2011**
  - **http://www.zdnet.com/news/taking-chips-to-10ghz-and-beyond/96055**
- **Core i7 990x hit the market early 2011**
  - **3.46GHz clock stretching up to 3.73 GHz in turbo mode**
  - **6 processing cores**
  - **Running in parallel, we get 22GHz of processing power!**

Moore's Law

- **Stated in 1965 that for the next 10 years, the *number or transistors* would double every two years**
  - **The prediction was only made for 10 years, but it is still true today**
- **Clock speed has leveled off**
  - **Heat buildup means we struggle to go beyond 4GHz**
  - **Moore's Law has often been misunderstood as *clock speed* doubling every 2 years**
- **The way to scale is to have lots of cores working together**

CPU / Core / Hardware Thread

- The Intel i7-3960X
- One CPU socket
- Six activated cores
- Each core supports two hyperthreads
  - Each core can only execute a single instruction at a time, but the data is fetched in parallel
- Total of 12 threads
- Runtime.getRuntime().availableProcessors() = 12



Japanese 'K' Computer

- **In June 2011, could calculate 8.2 petaFLOPS**
  - **8 200 000 000 000 000 floating point operations per second**
  - **Intel 8087 was 30 000 FLOPS, 273 billion times slower**
  - **548,352 cores from 68,544 2GHz 8-Core SPARC64 VIIIfx processors**
- **By November 2011, it had surpassed 10 petaFLOPS**

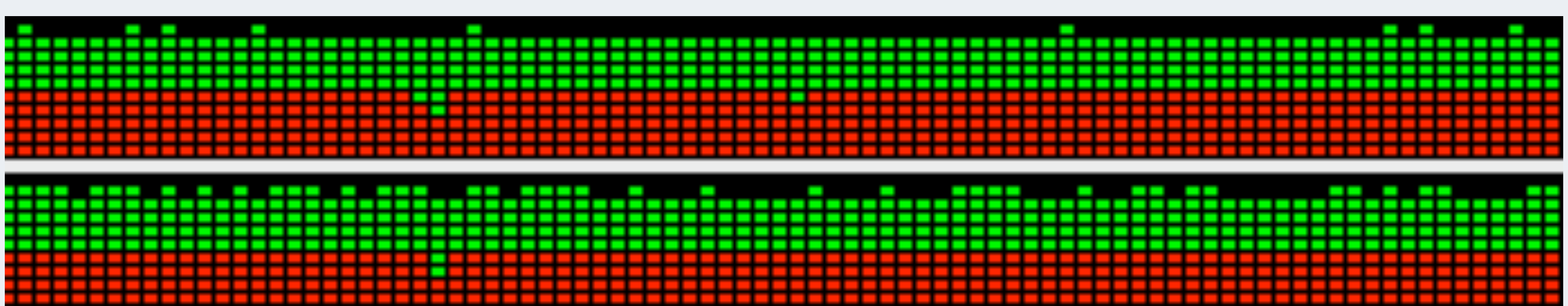
"Sequoia" At Lawrence Livermore National Lab

- **Used by USA's National Nuclear Security Administration to simulate nuclear bombs**
- **June 2012: Delivers 16 petaflops**
  - **1.6 million cores**
  - **1.6 petabytes of memory**

Utilization Of Hardware

- **Threading is software abstraction to keep hardware busy**
  - **Otherwise, why put up with safety and liveness issues?**
- **We want to utilize all our CPUs with application code**
  - **Having too many serial sections means that not all CPUs are working**

  - **Too much locking means we are busy with system code**



Threading Models

- Preemptive multithreading (Native Threads)
  - Operating system is responsible for forcing a context switch
  - Threads can be swapped in the middle of an operation
    - For example half-way through `balance = balance + 100`
- Cooperative multithreading (Green Threads)
  - Threads give up control at a stopping point
    - Yield, sleep, wait
  - Infinite loops could never give up control
- Which One?
  - Preemptive (native) is safer, but we get race conditions
  - In modern JDKs, preemptive is used

10: Avoiding Liveness Hazards

Safety first!

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

10.1 Deadlock

Avoiding Liveness Hazards

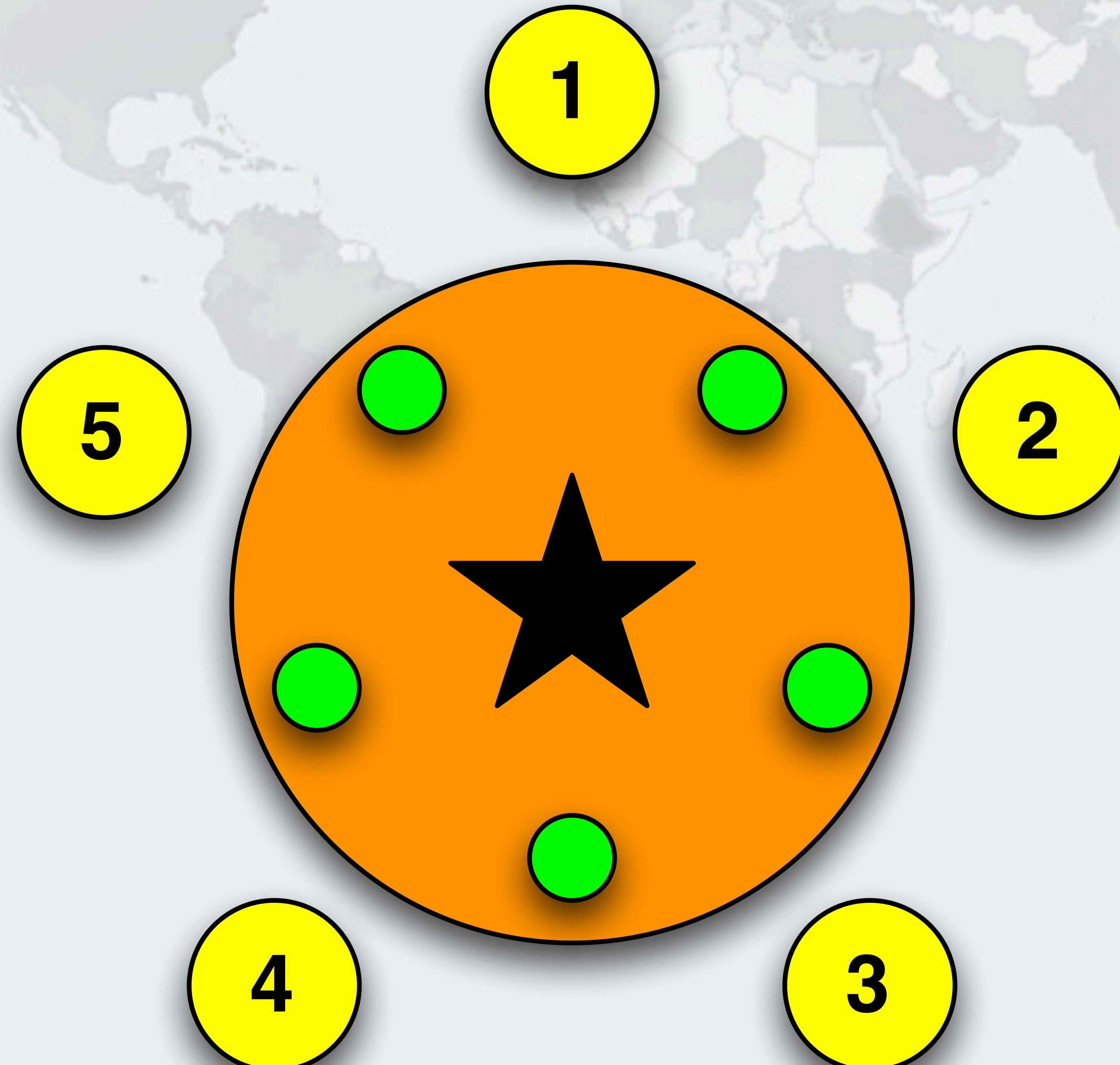
10.1 Deadlock

- **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**

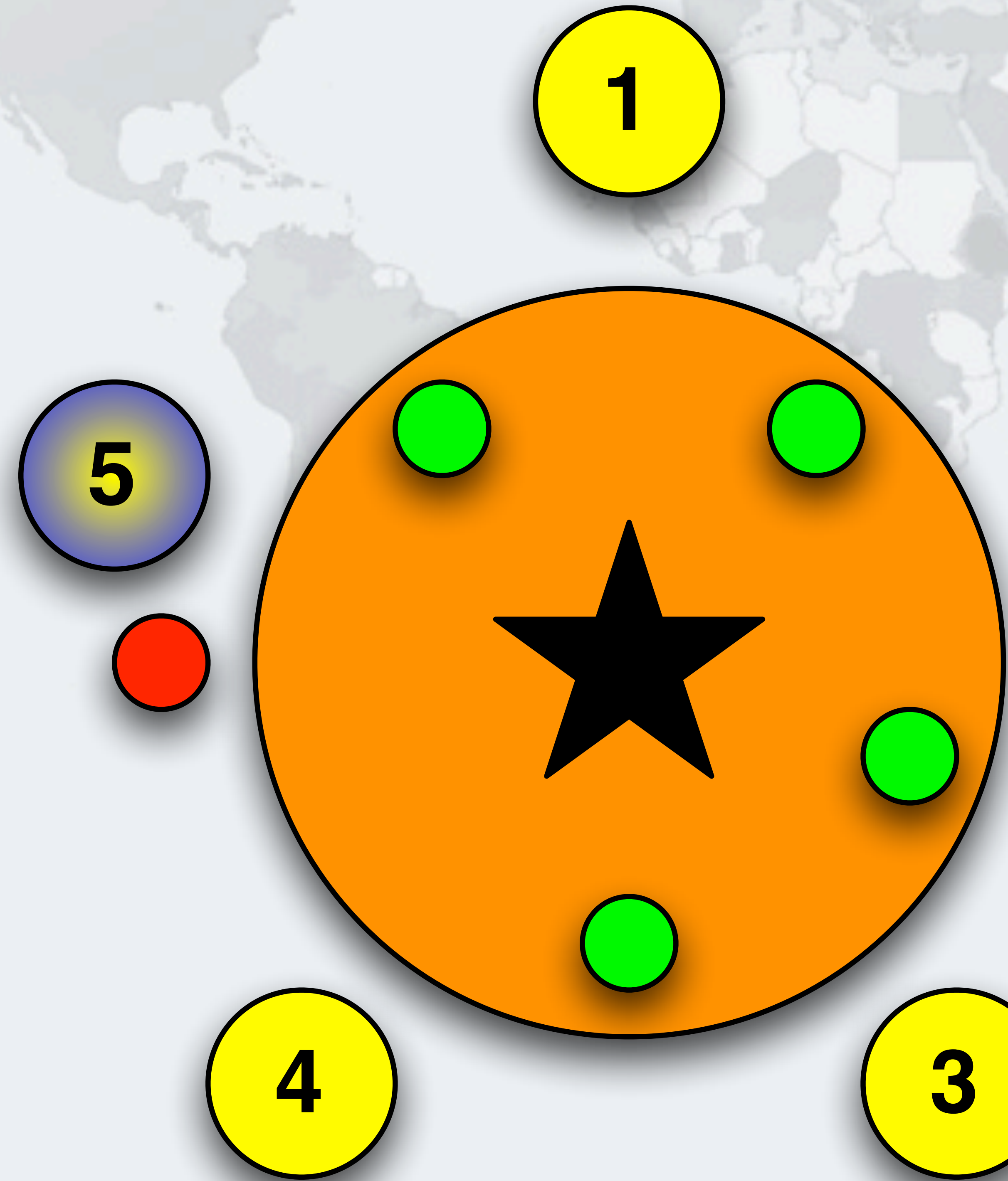
Legends For Example

- Thinking philosopher
- Drinking philosopher
- Changing state philosopher
- Available cup
- Taken cup

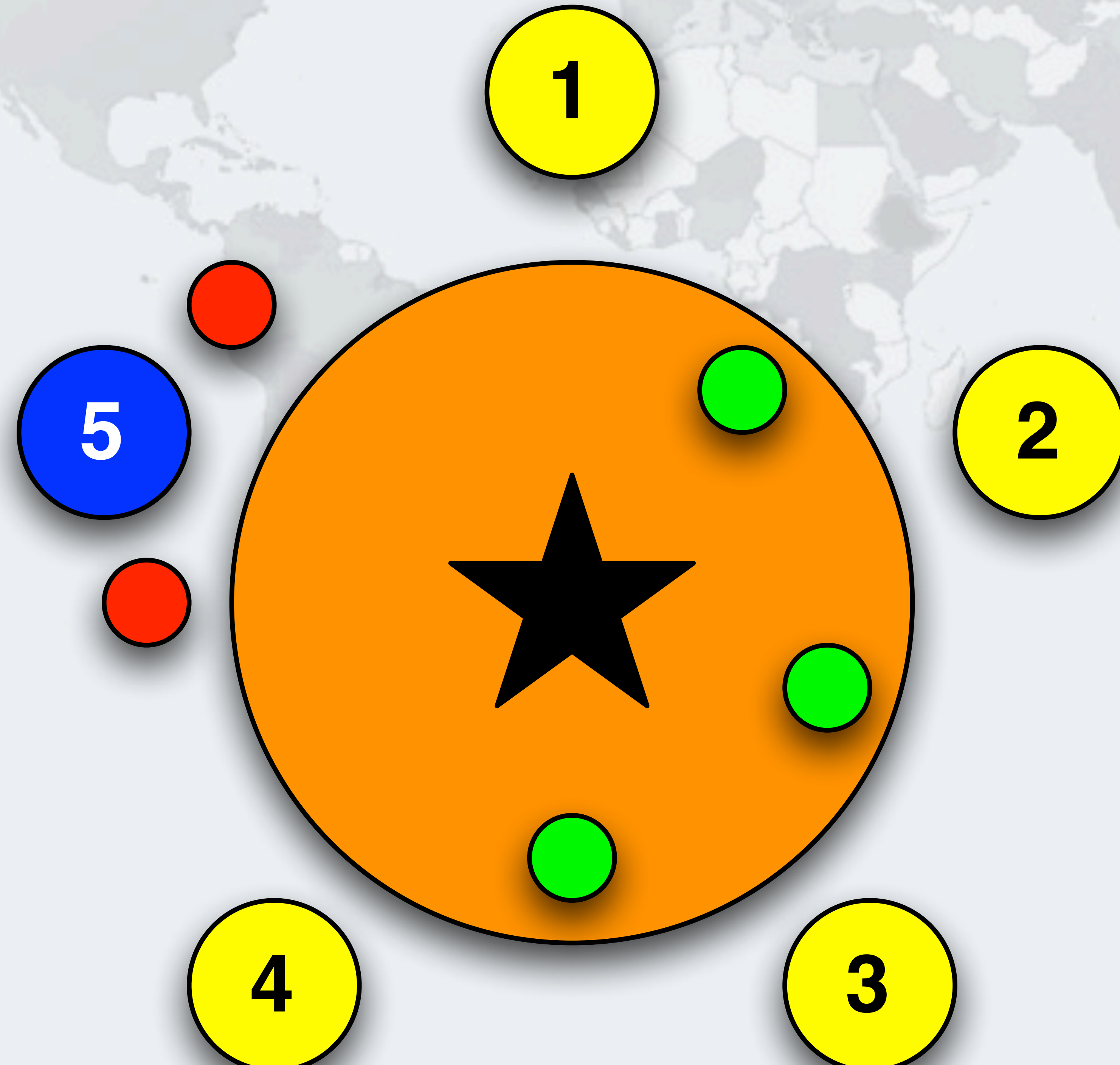
Table Is Ready, All Philosophers Are Thinking



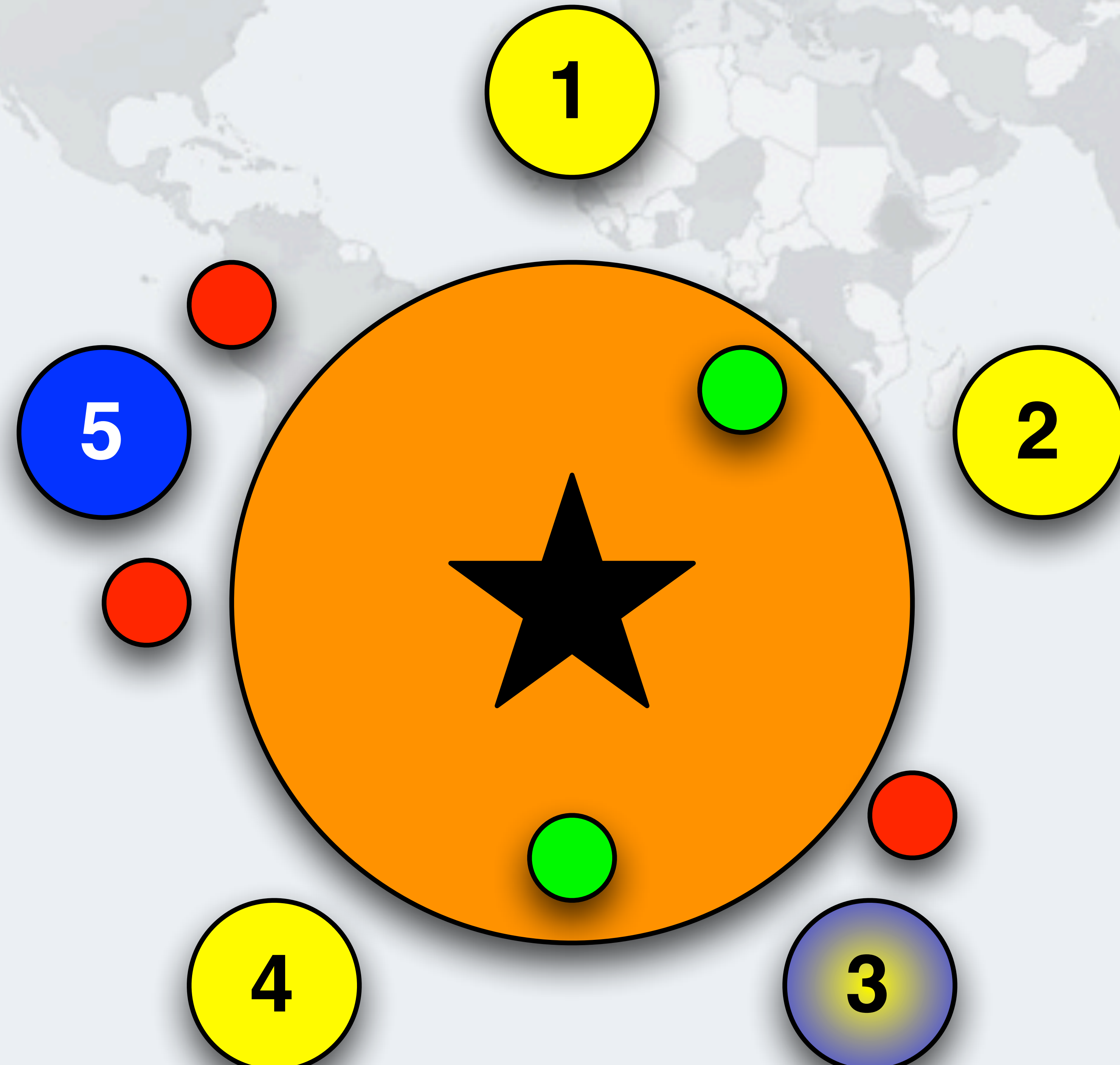
Philosophers 5 Wants To Drink, Takes Right Cup



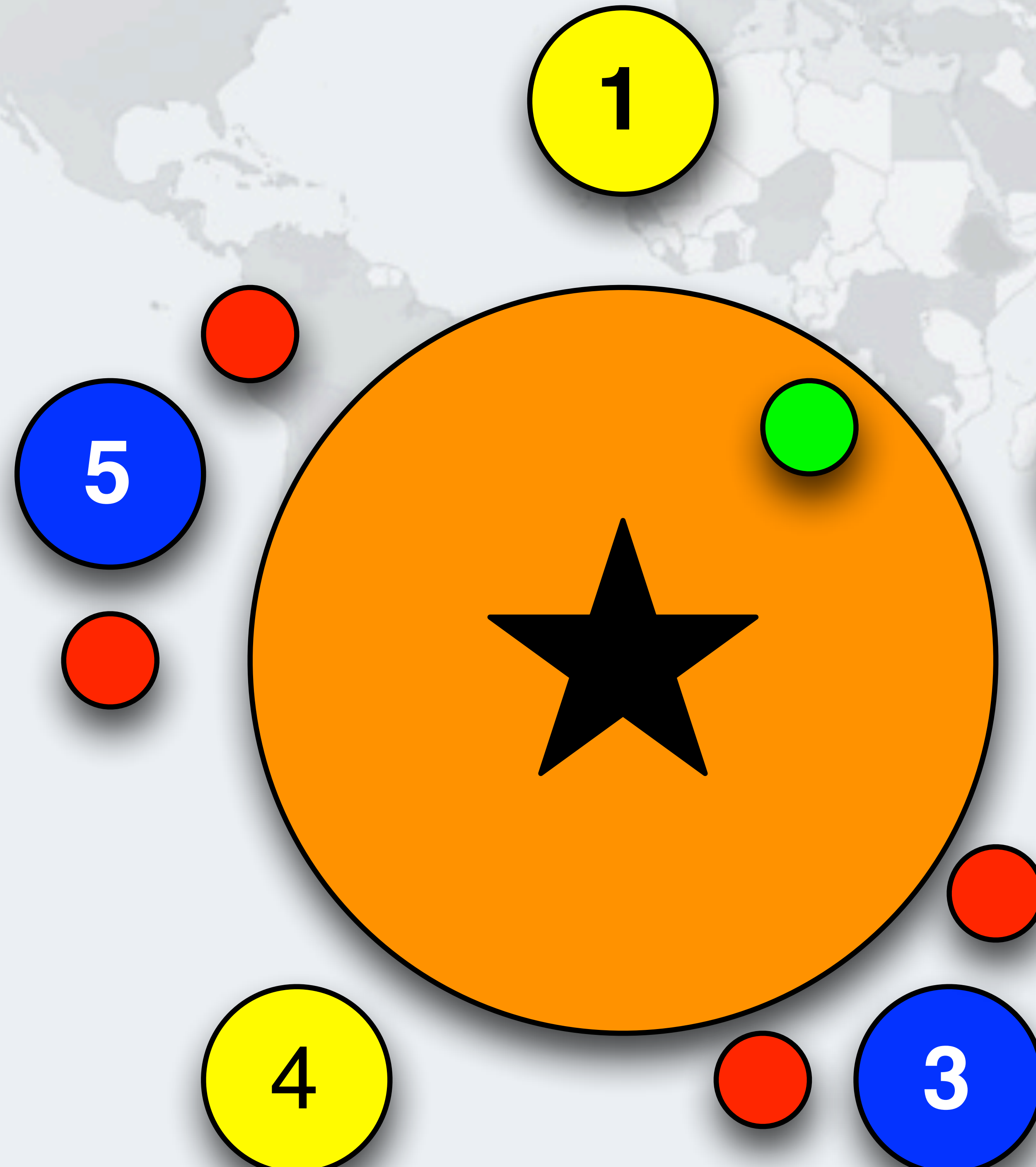
Philosopher 5 Is Now Drinking With Both Cups



Philosophers 3 Wants To Drink, Takes Right Cup

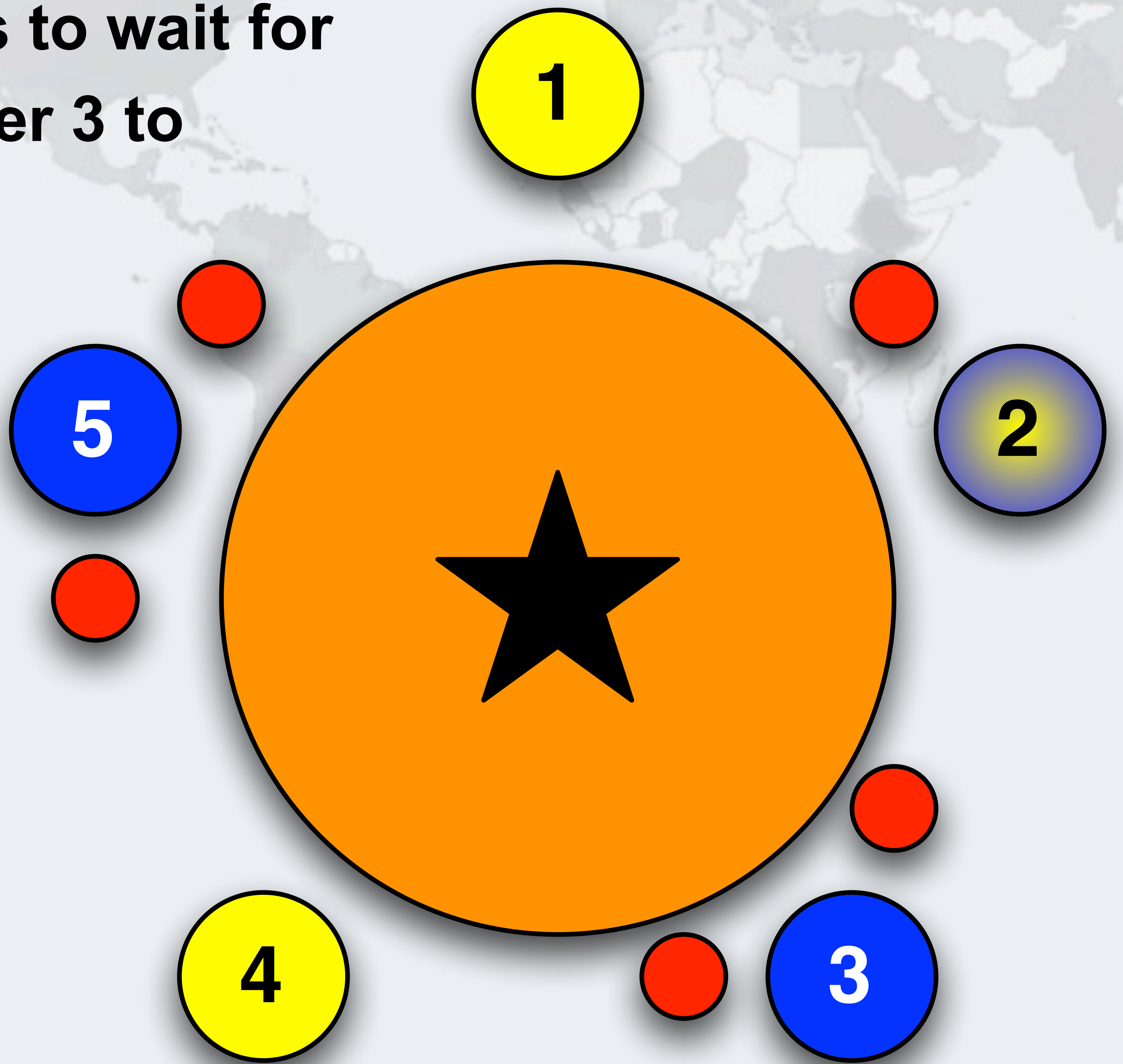


Philosopher 3 Is Now Drinking With Both Cups

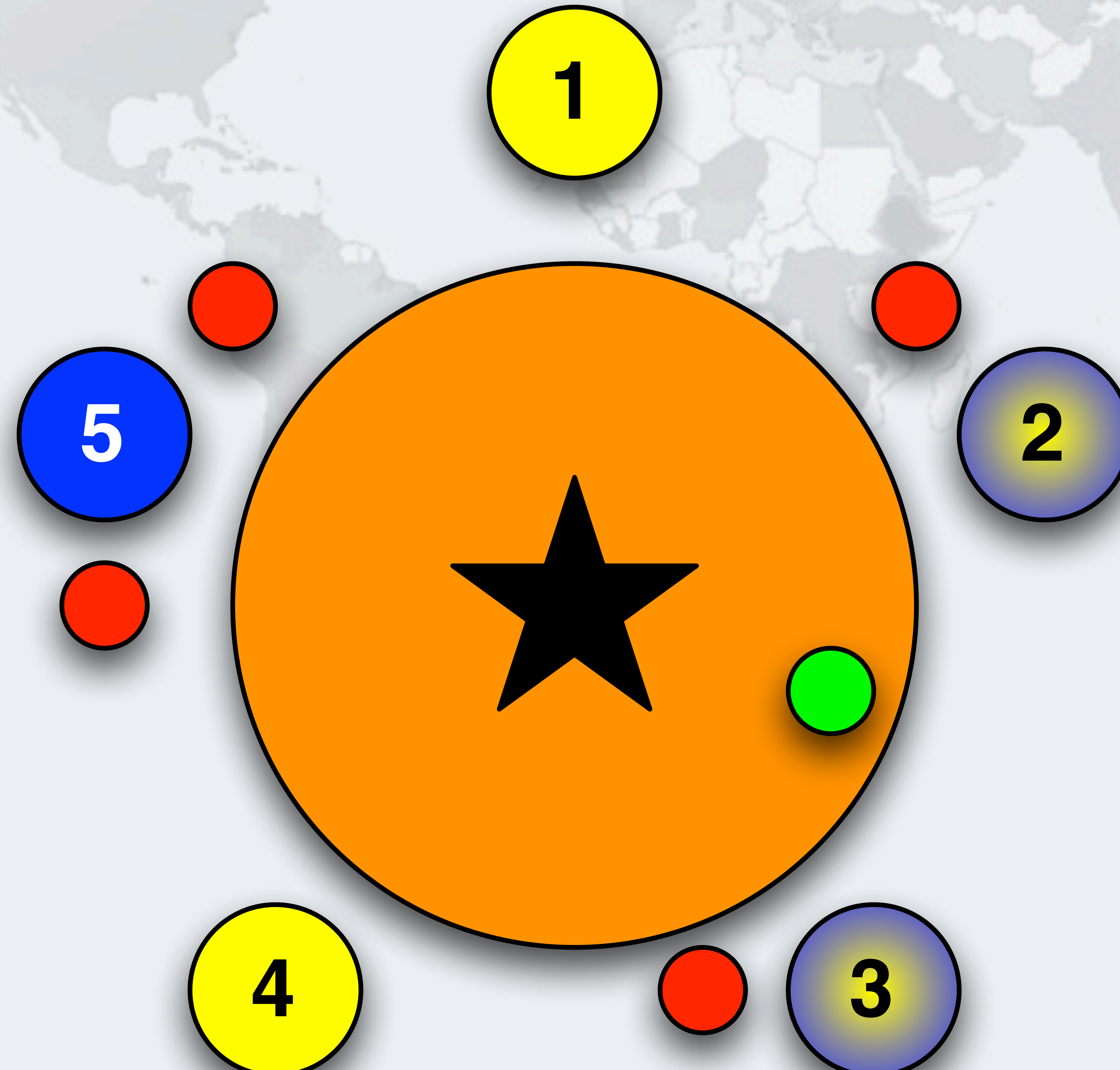


Philosophers 2 Wants To Drink, Takes Right Cup

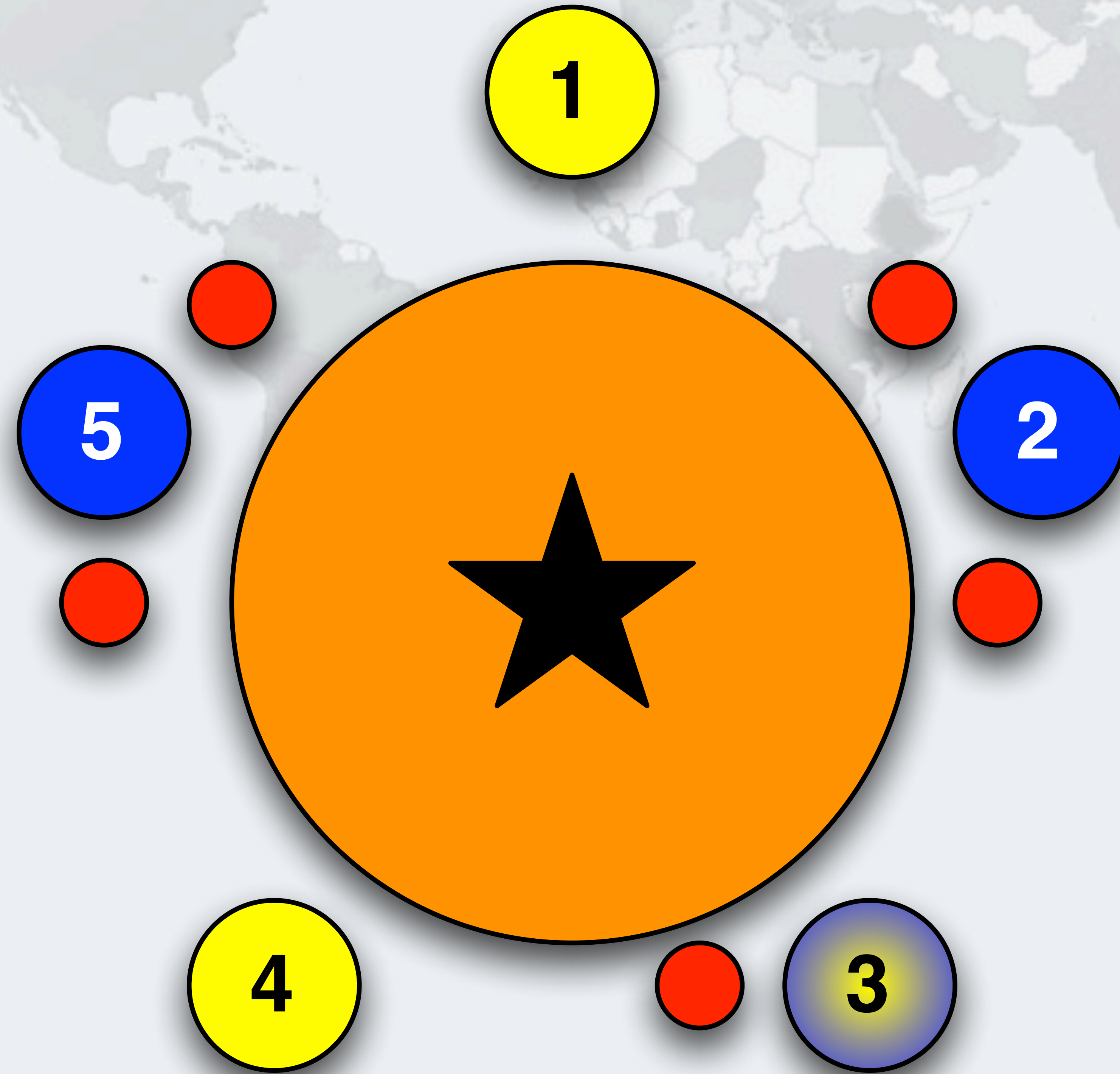
- But he has to wait for Philosopher 3 to finish his drinking session



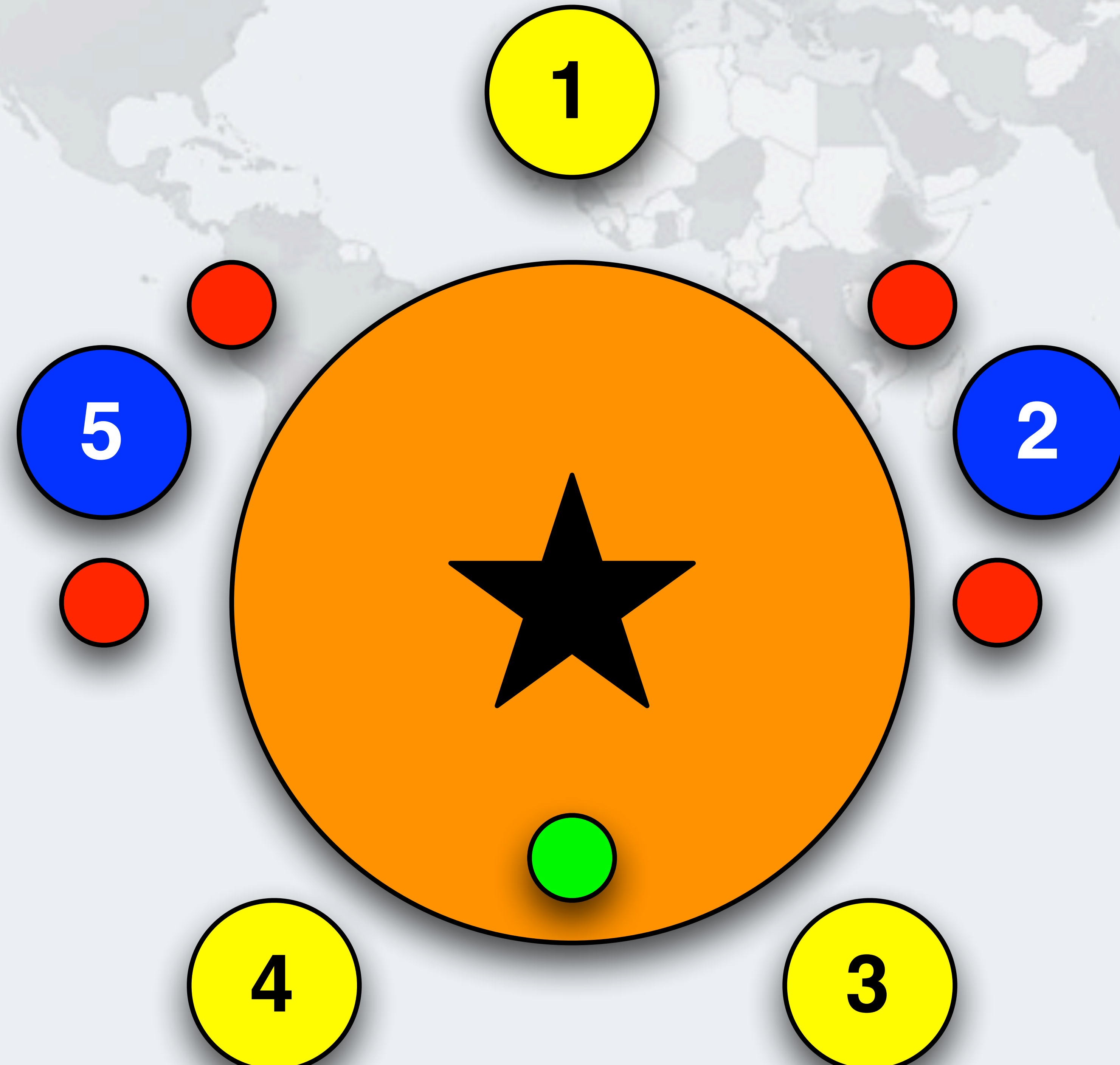
Philosopher 3 Finished Drinking, Returns Right Cup



Philosopher 2 Is Now Drinking With Both Cups



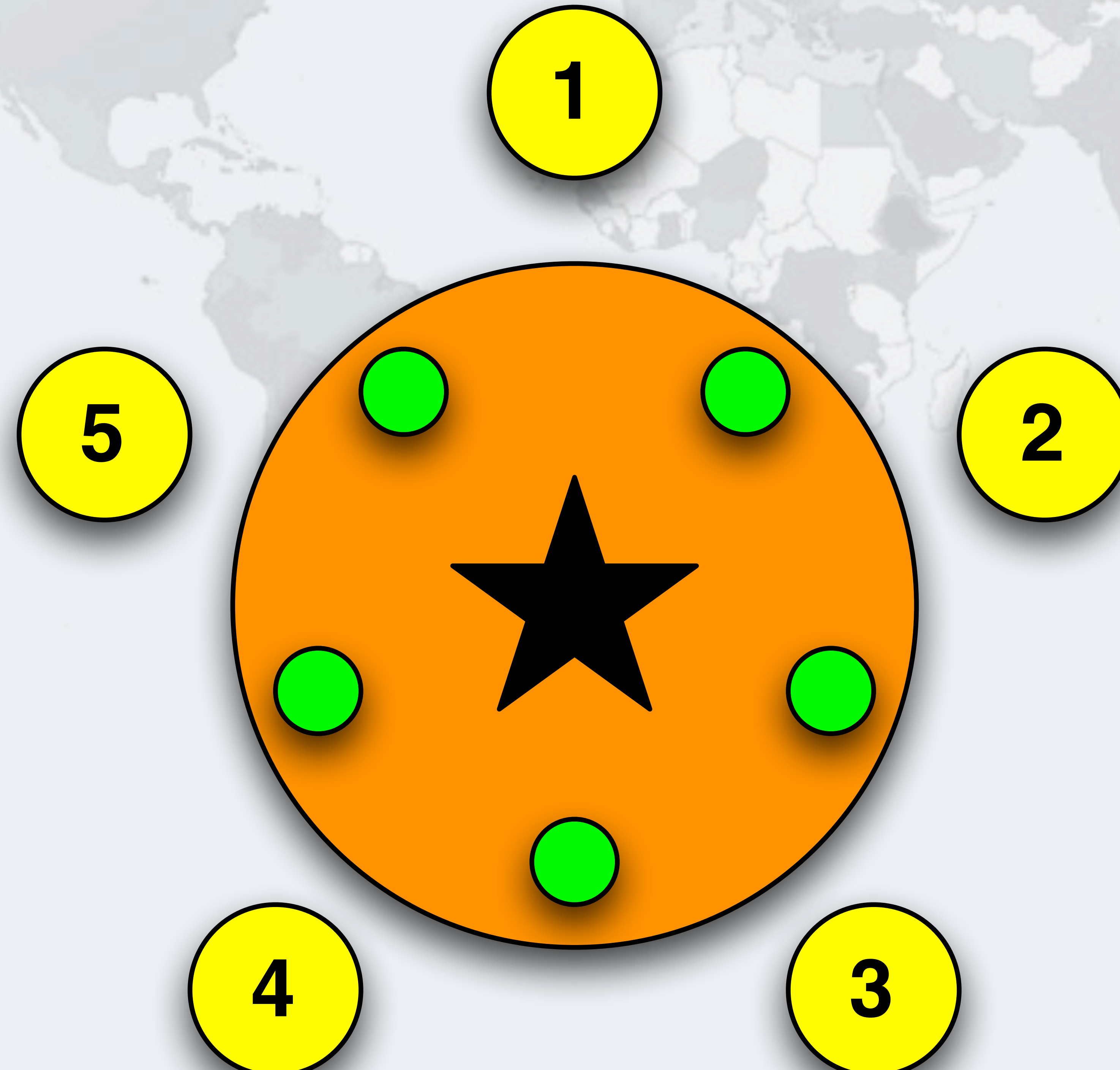
Philosopher 3 Returns Left Cup



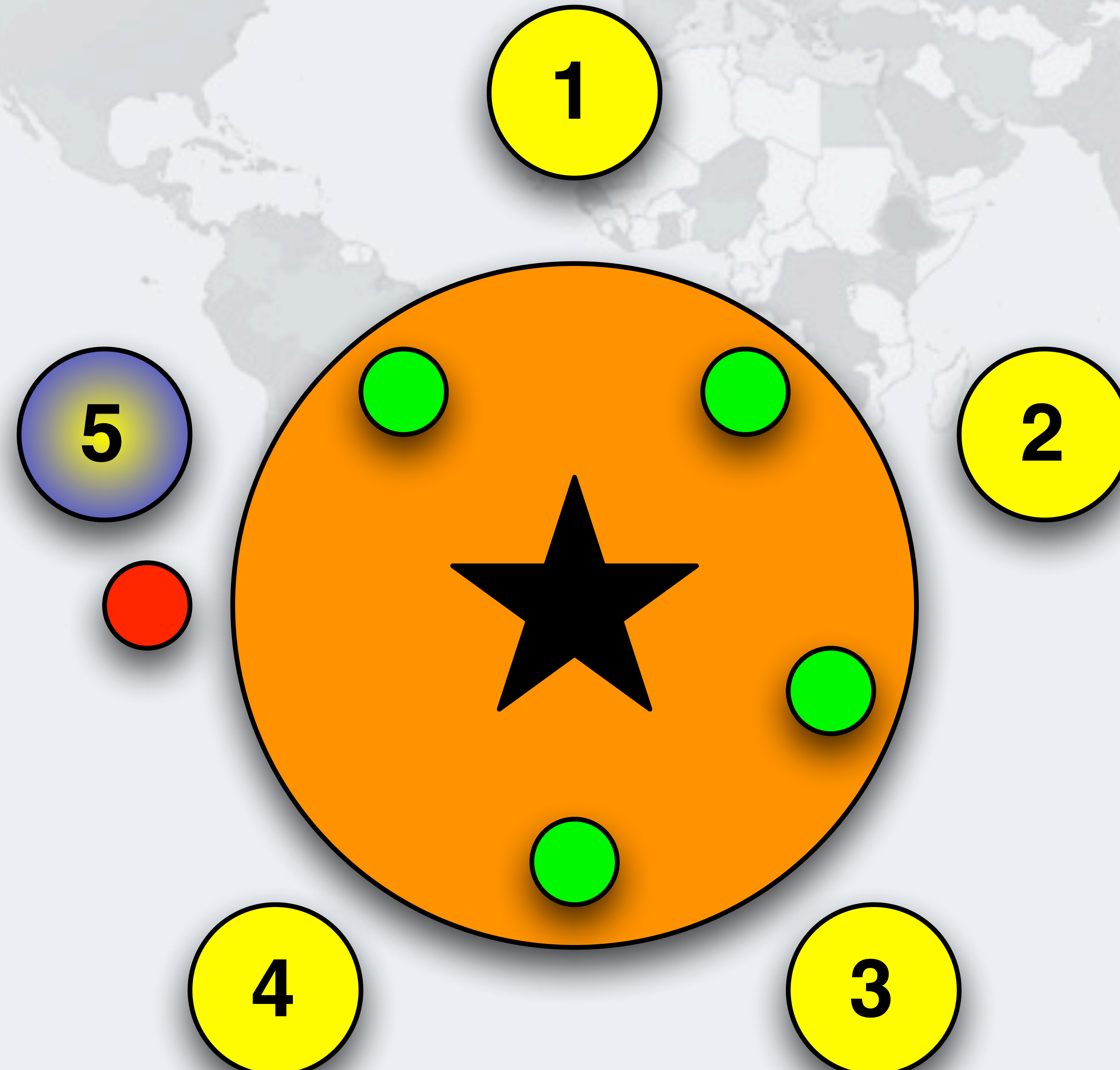
Drinking Philosophers In Limbo

- **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**

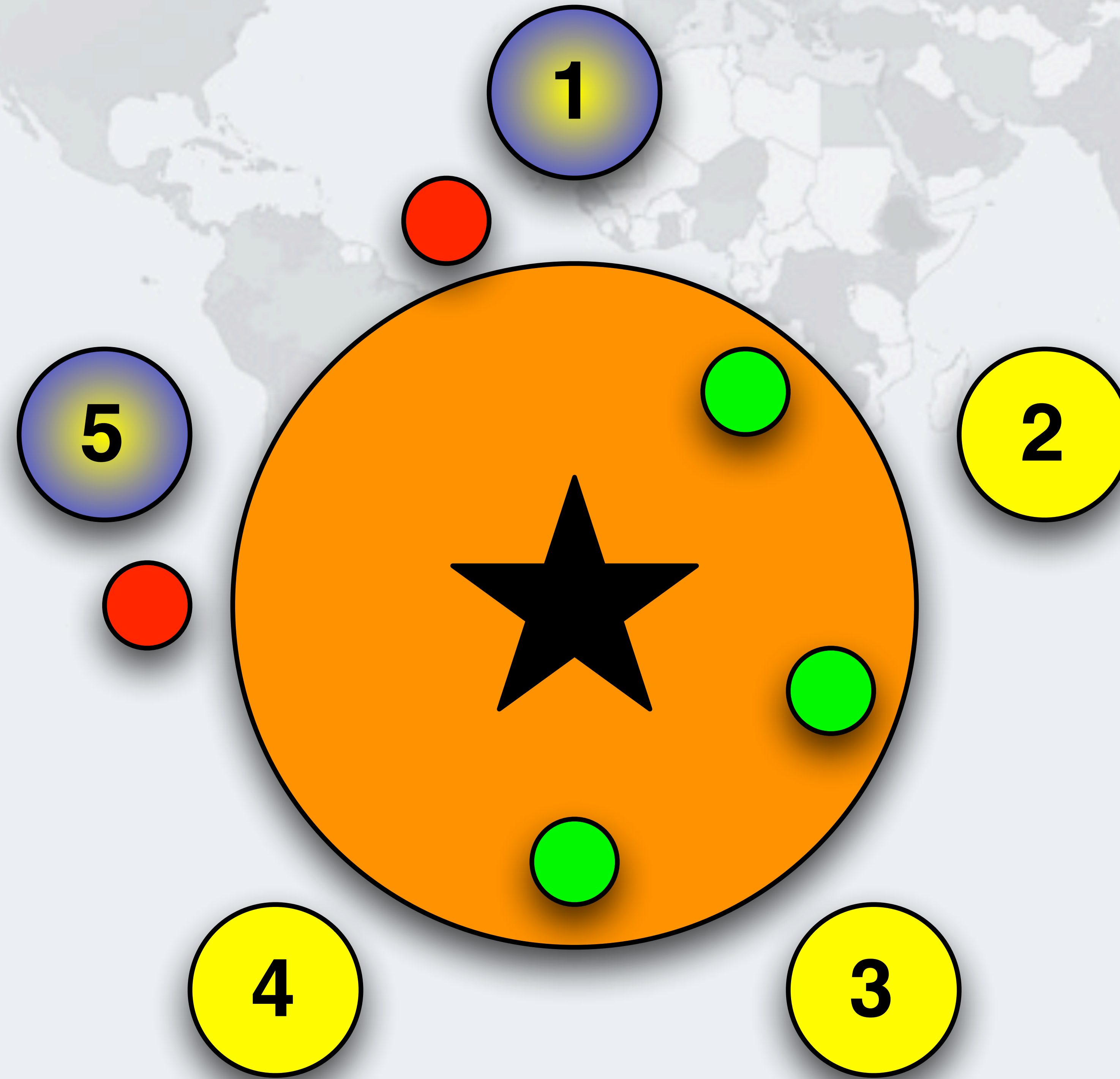
A Deadlock Can Easily Happen With This Design



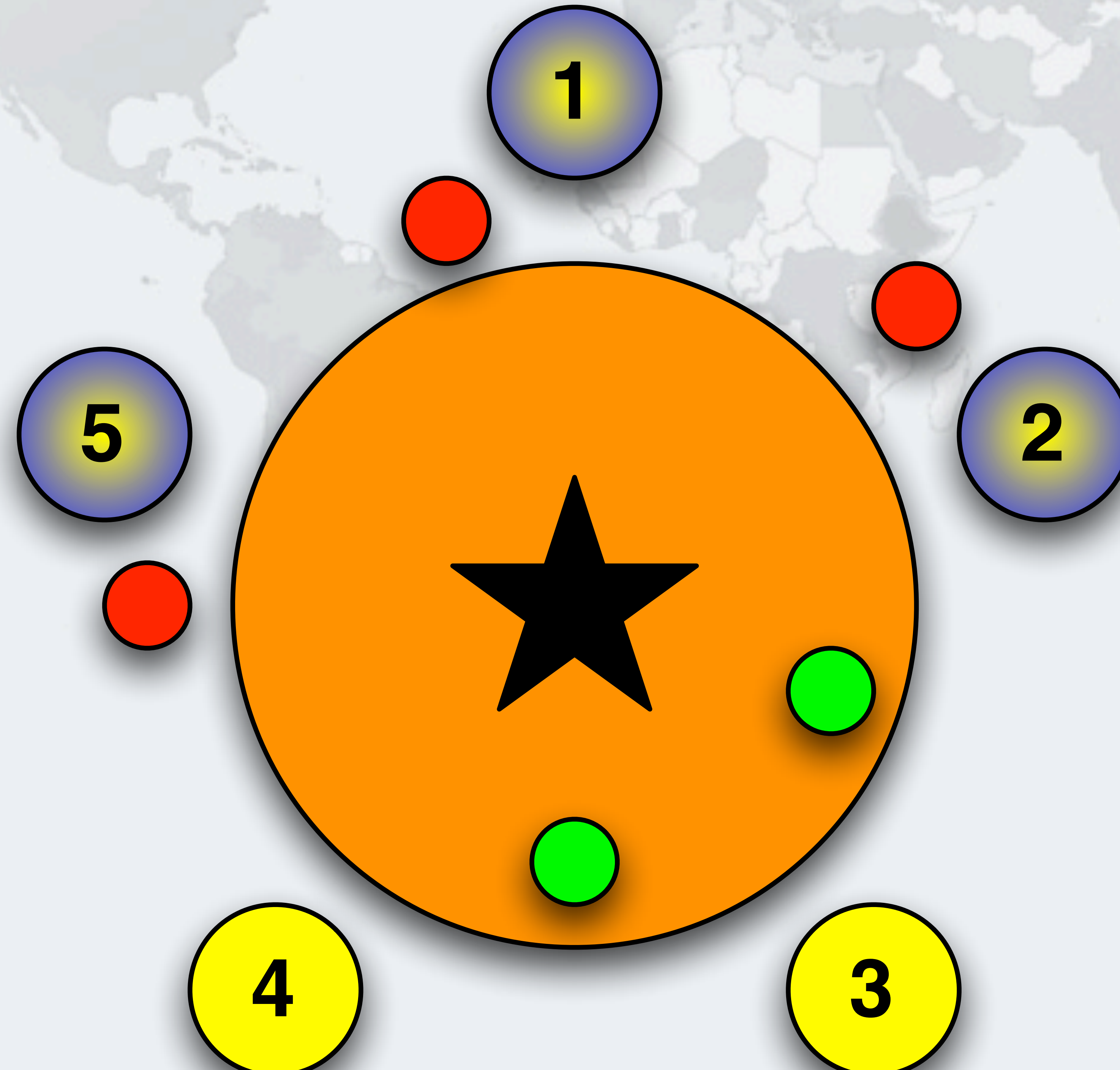
Philosopher 5 Wants To Drink, Takes Right Cup



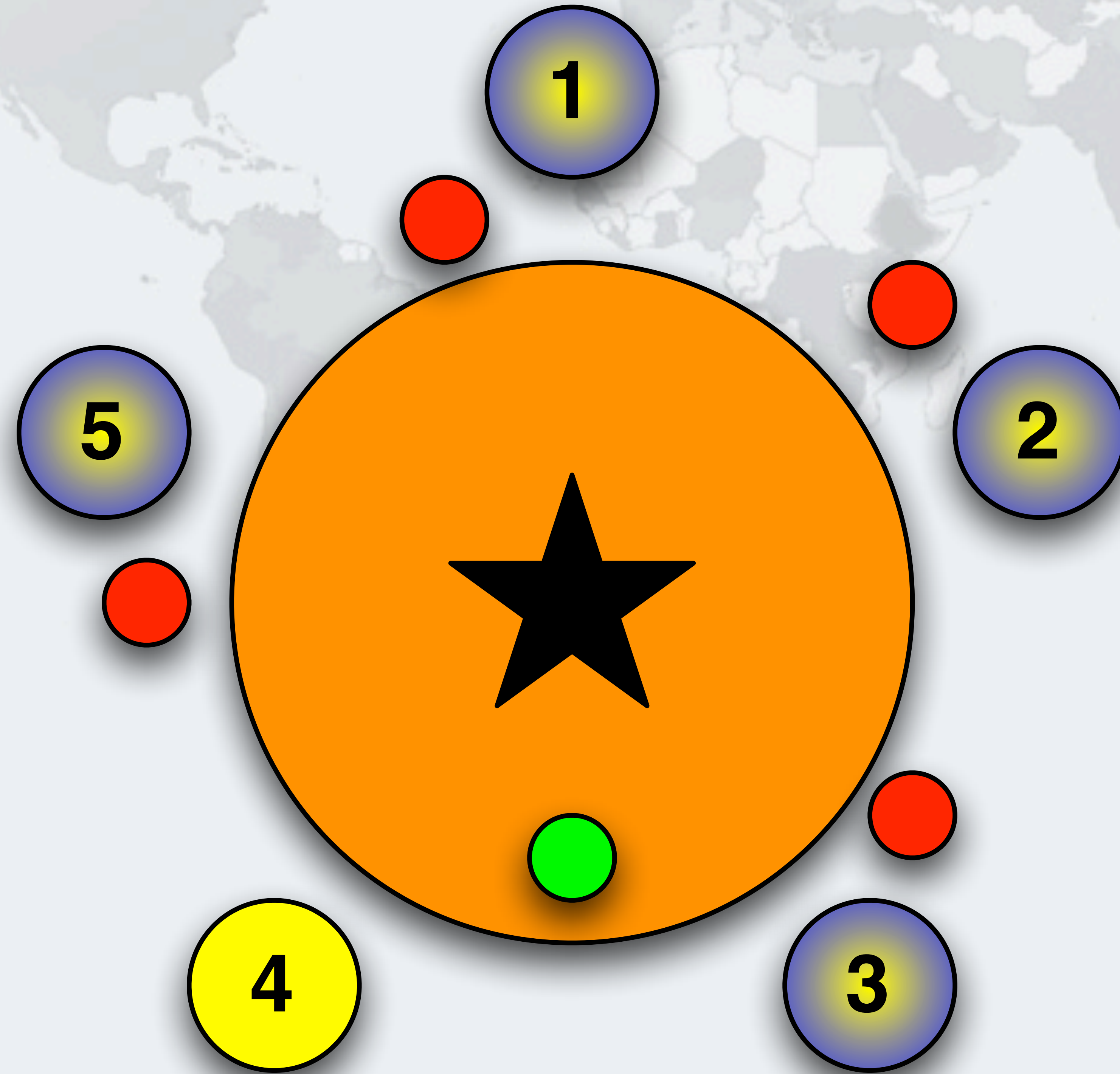
Philosopher 1 Wants To Drink, Takes Right Cup



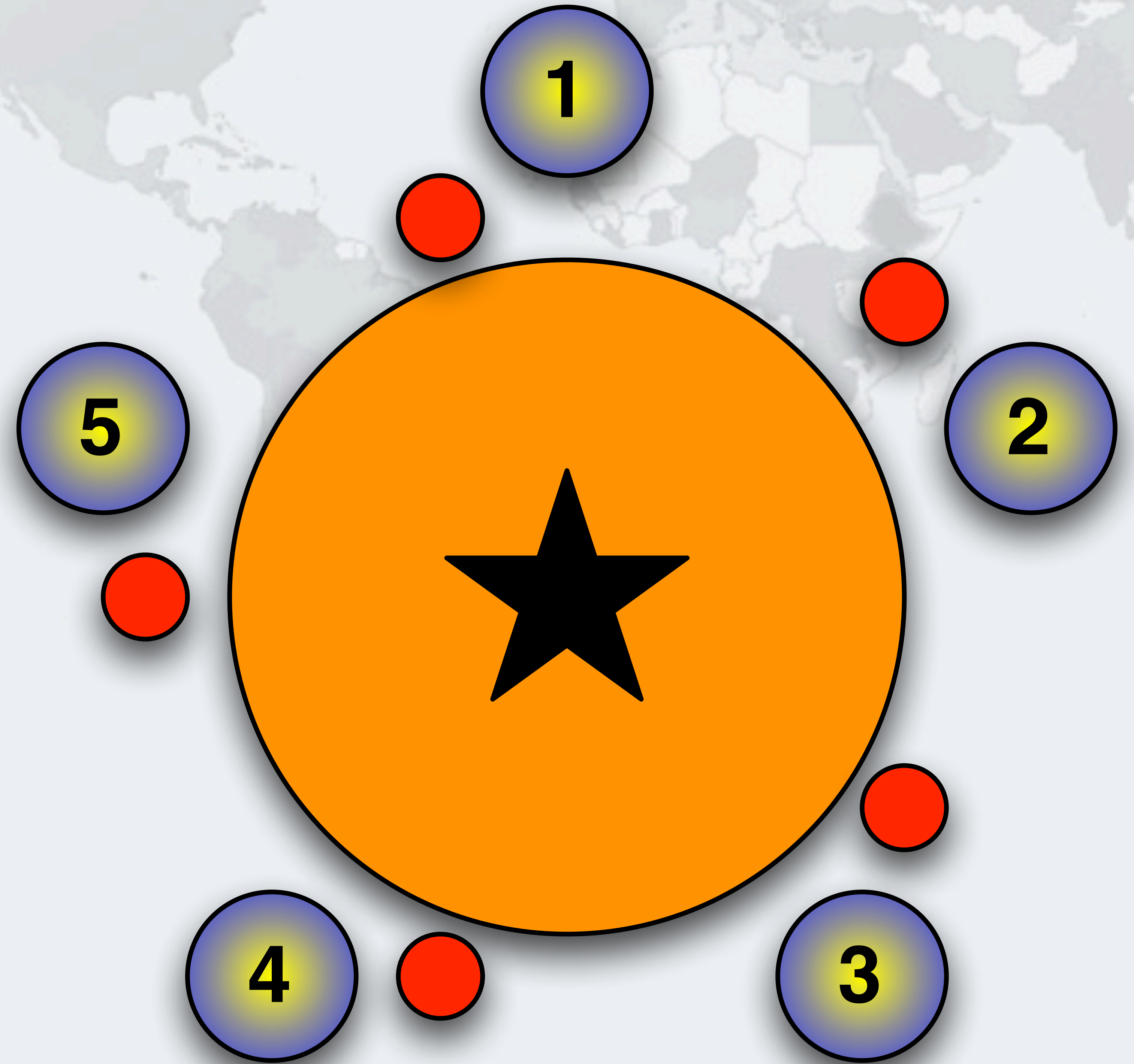
Philosopher 2 Wants To Drink, Takes Right Cup



Philosopher 3 Wants To Drink, Takes Right Cup

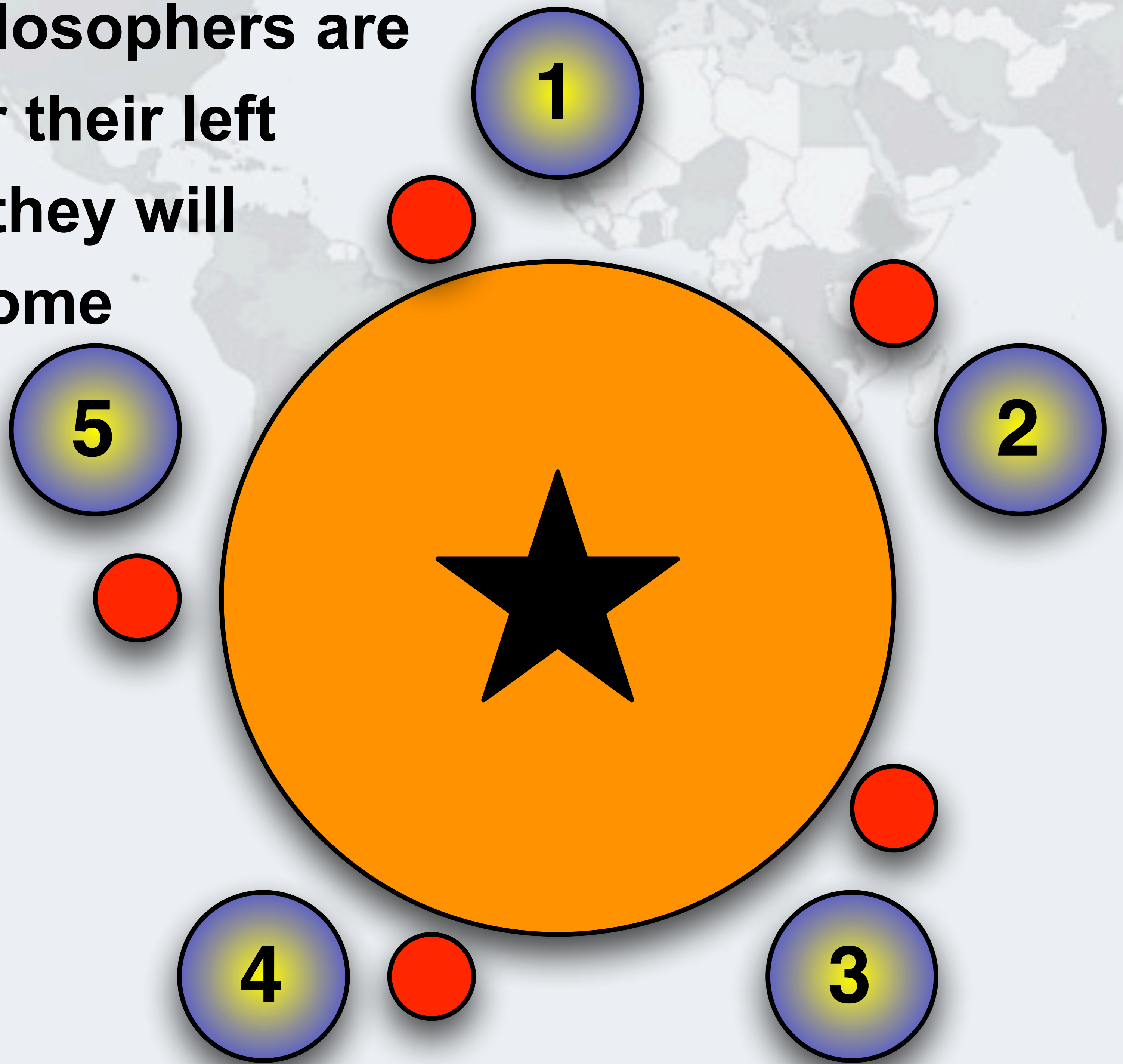


Philosopher 4 Wants To Drink, Takes Right Cup



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

How Do We Discover Deadlocks?

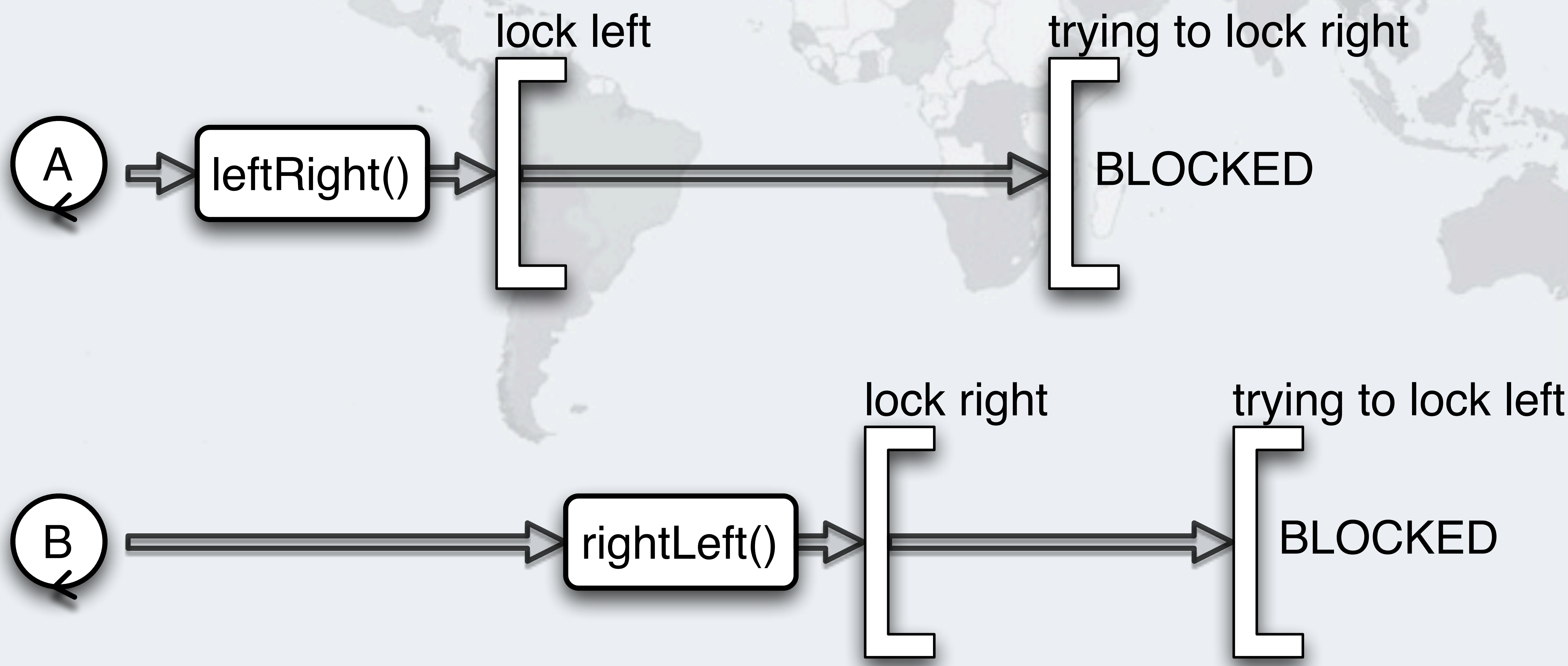
- **A lot of Java code contains subtle locking bugs**
  - **Calling methods in different orders could cause a *deadly embrace***
  - **Calling alien methods could cause a call-back**
  - **Limiting resources can cause deadlocks with dependent actions**
- **Most of the time, deadlocks do not manifest themselves**
  - **Usually never during testing**
  - **Seldom during production, only if the system is really busy**
    - **Often you will need to run the application for 5 days before it happens, usually on a Friday afternoon to ruin your weekend**

Lock-ordering Deadlocks

- **This code will cause deadlocks if called by two threads**

```
public class LeftRightDeadlock {
  private final Object left = new Object();
  private final Object right = new Object();
  public void leftRight() {
    synchronized (left) {
      synchronized (right) {
        doSomething();
      }
    }
  }
  public void rightLeft() {
    synchronized (right) {
      synchronized (left) {
        doSomethingElse();
      }
    }
  }
}
```

Interleaving Of Call Sequence Causes Deadlock



Global Order Of Locks

- **A program will be free of lock-ordering deadlocks if all threads acquire the locks they need in a *fixed global order***
  - **Thus we can solve the deadlock by changing rightLeft() to**

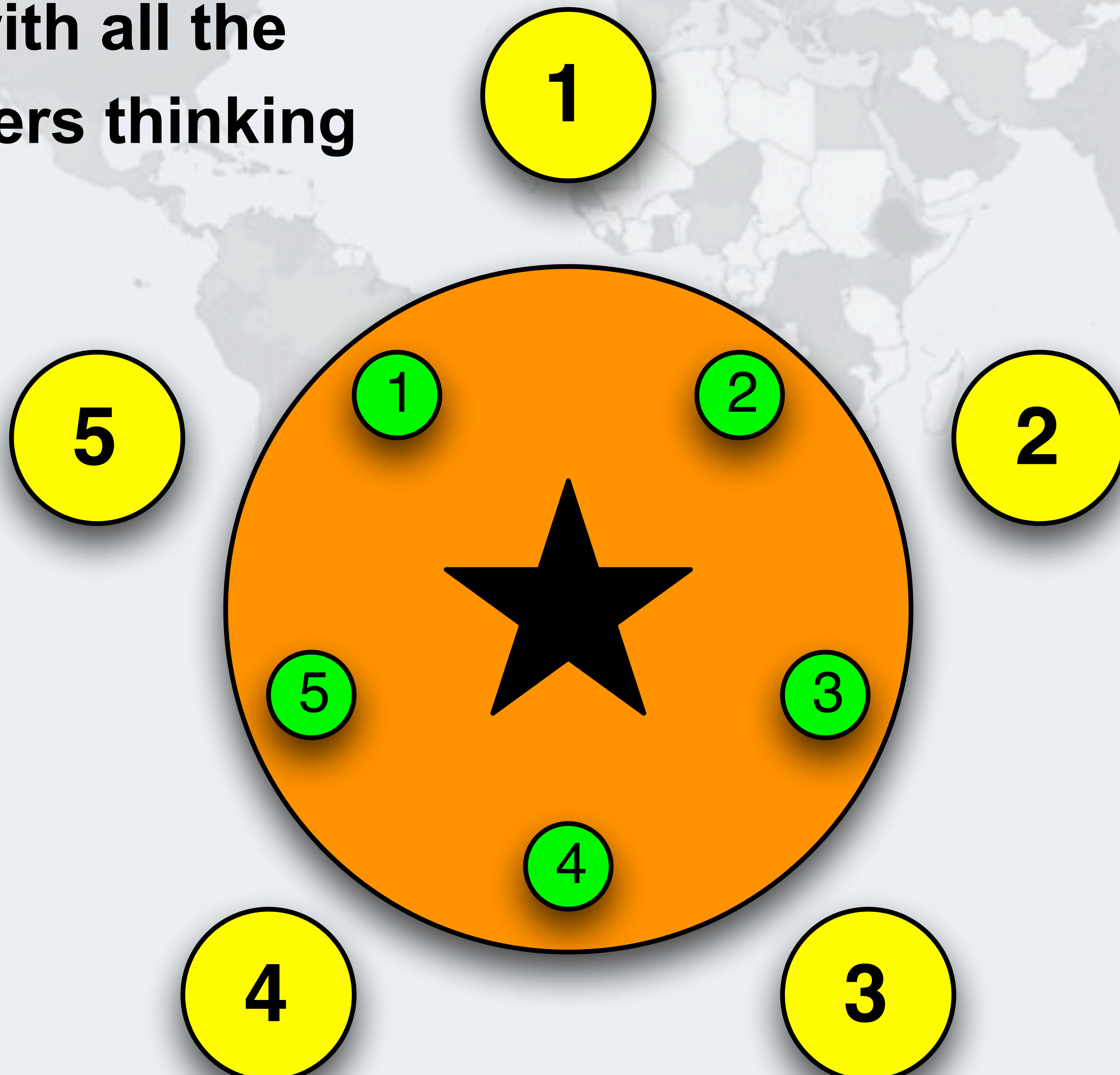
```
public void rightLeft() {
  synchronized (left) {
    synchronized (right) {
      doSomethingElse();
    }
  }
}
```

Global Order With Boozing Philosophers

- **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**
  - **And return the cup with the lowest number first**

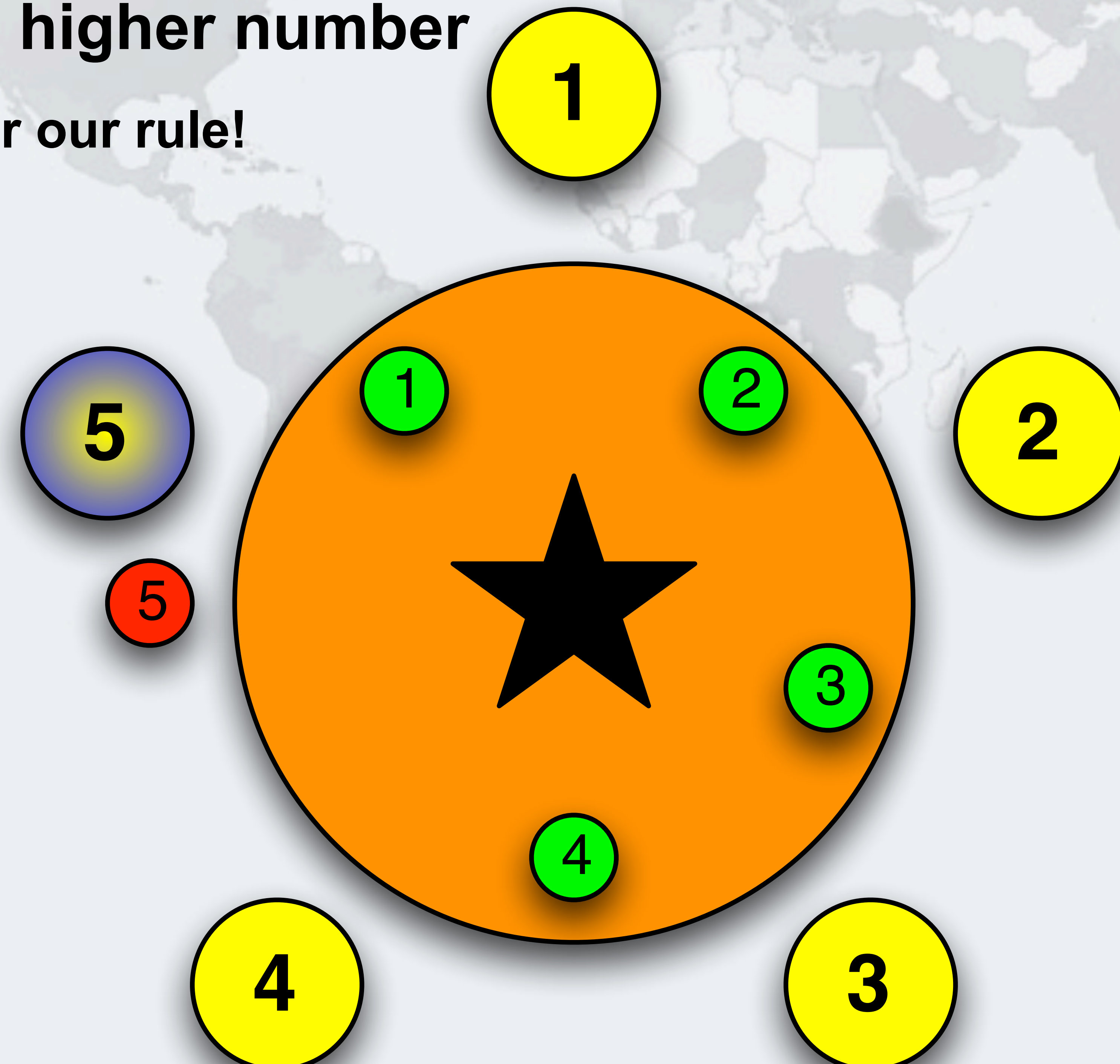
Global Lock Ordering

- **We start with all the philosophers thinking**



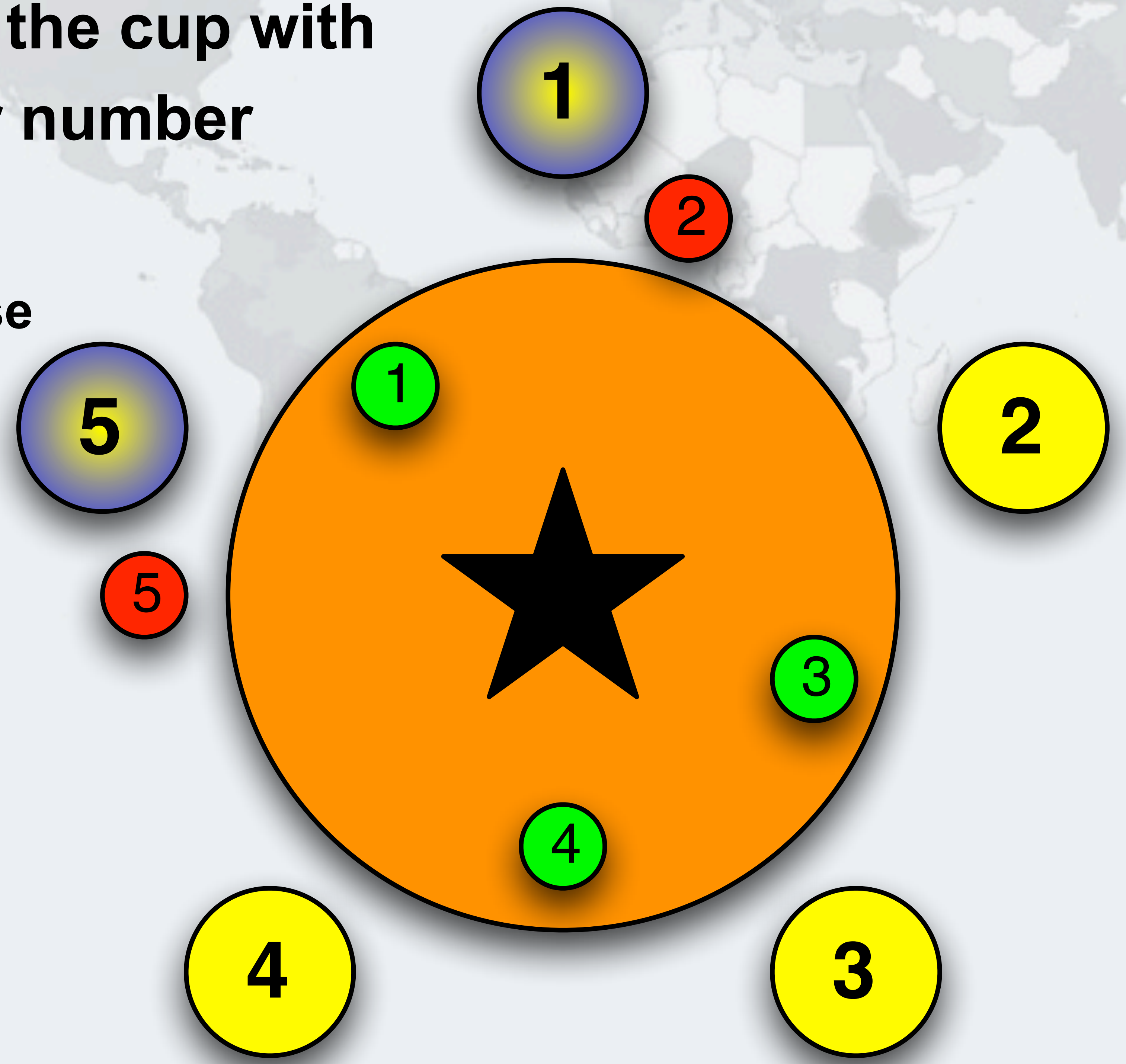
Philosopher 5 Takes Cup 5

- **Cup 5 has higher number**
  - **– Remember our rule!**

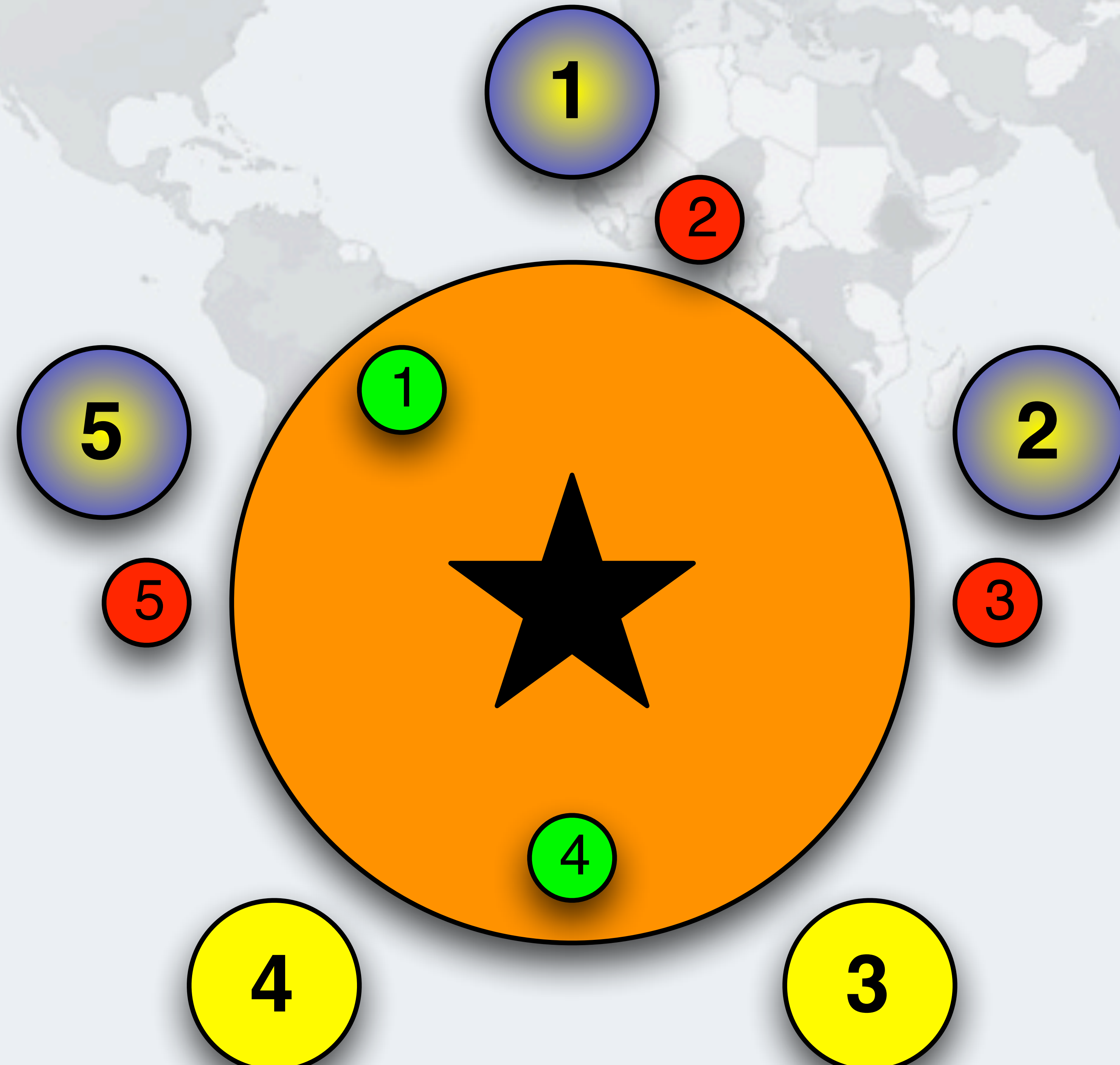


Philosopher 1 Takes Cup 2

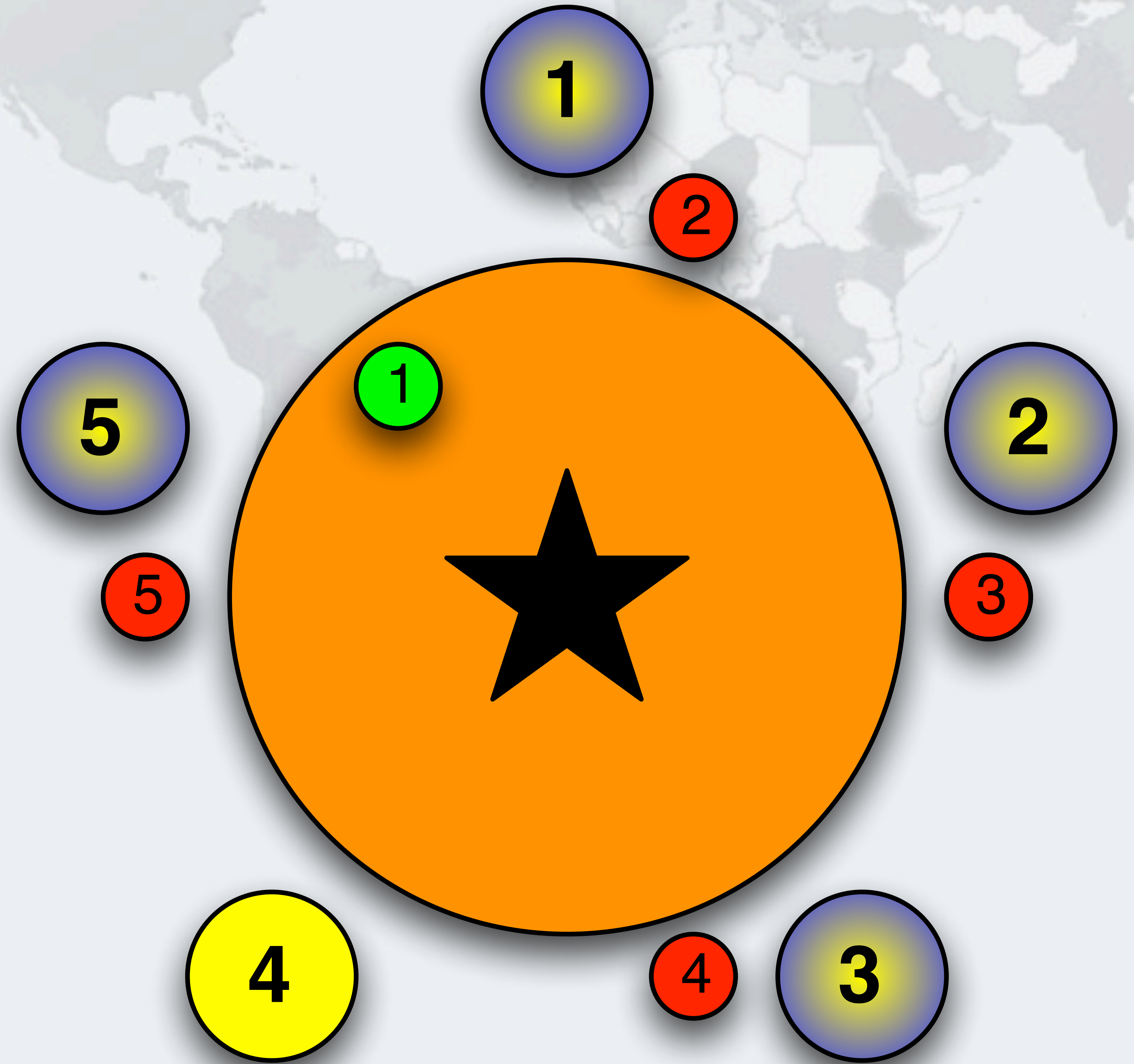
- Must take the cup with the higher number first
  - In this case cup 2



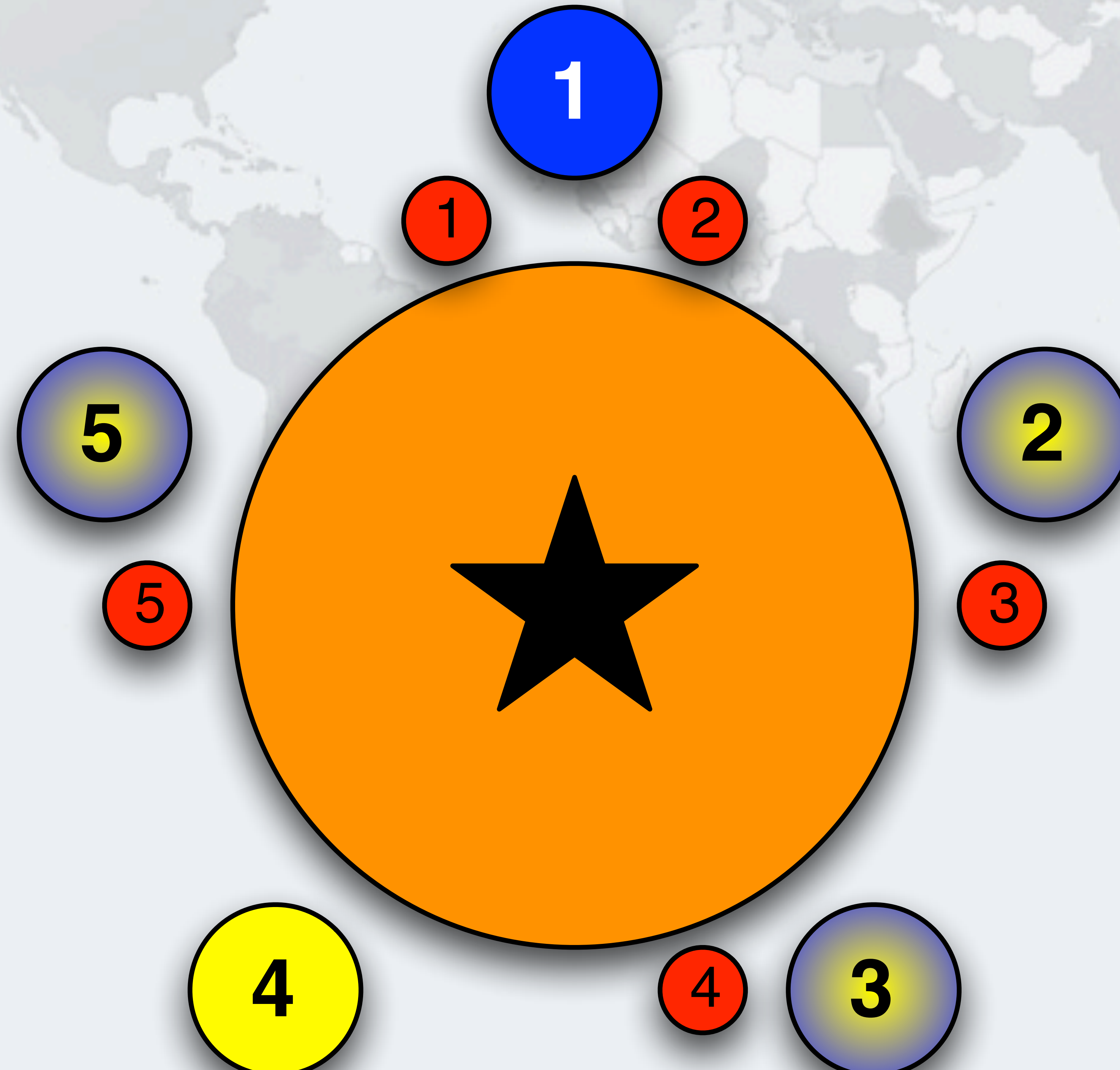
Philosopher 2 Takes Cup 3



Philosopher 3 Takes Cup 4

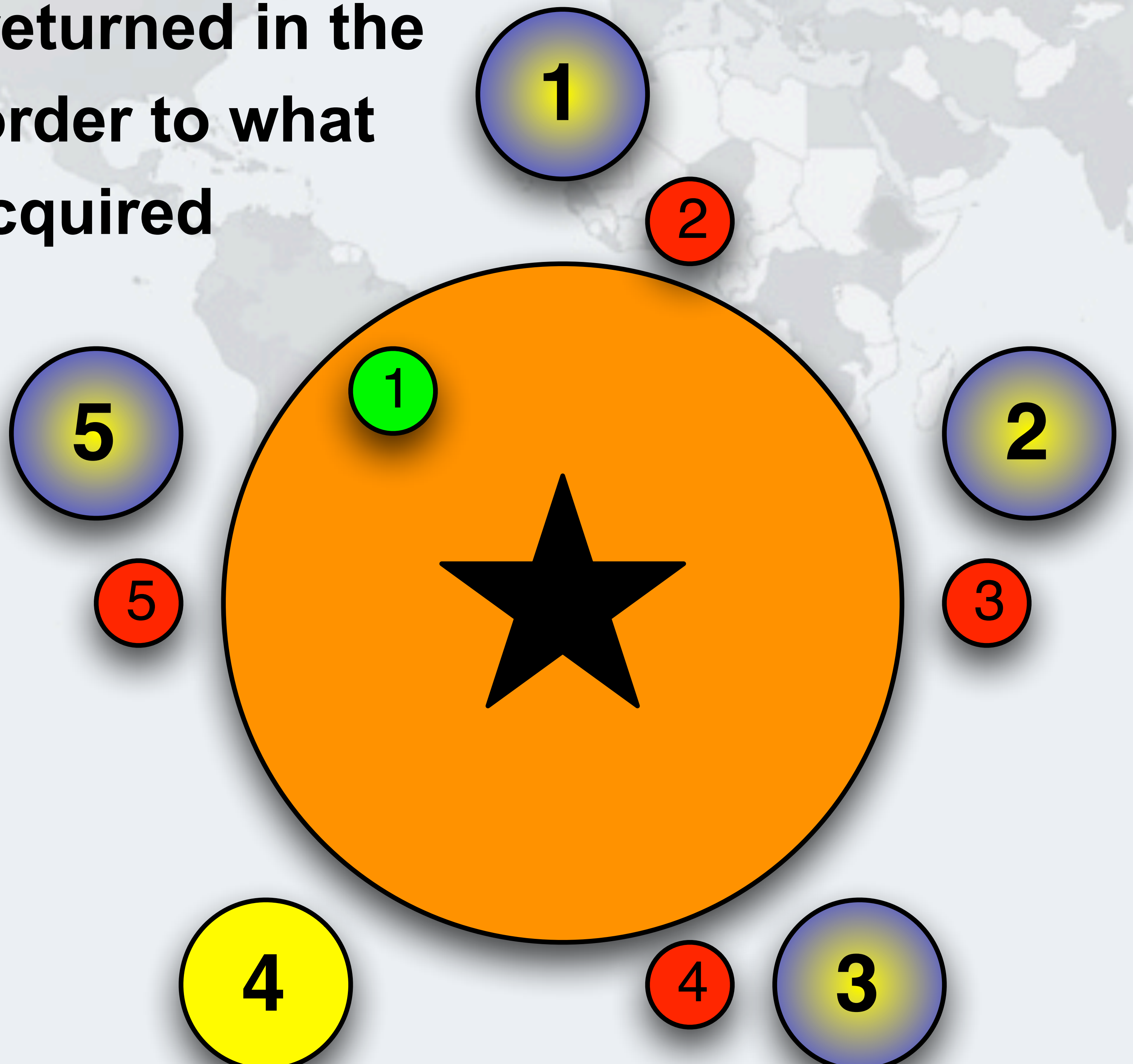


Philosopher 1 Takes Cup 1 - Drinking

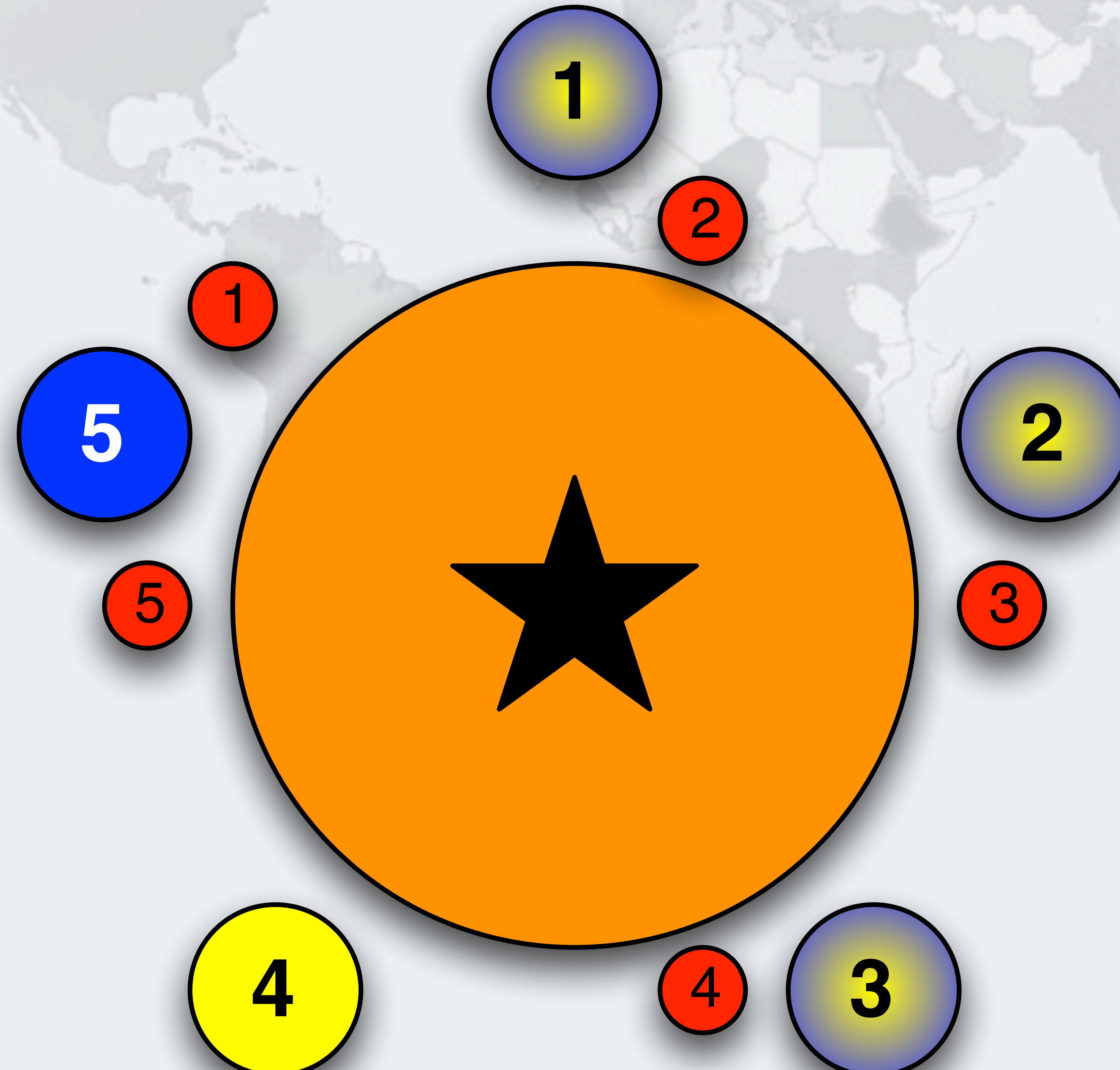


Philosopher 1 Returns Cup 1

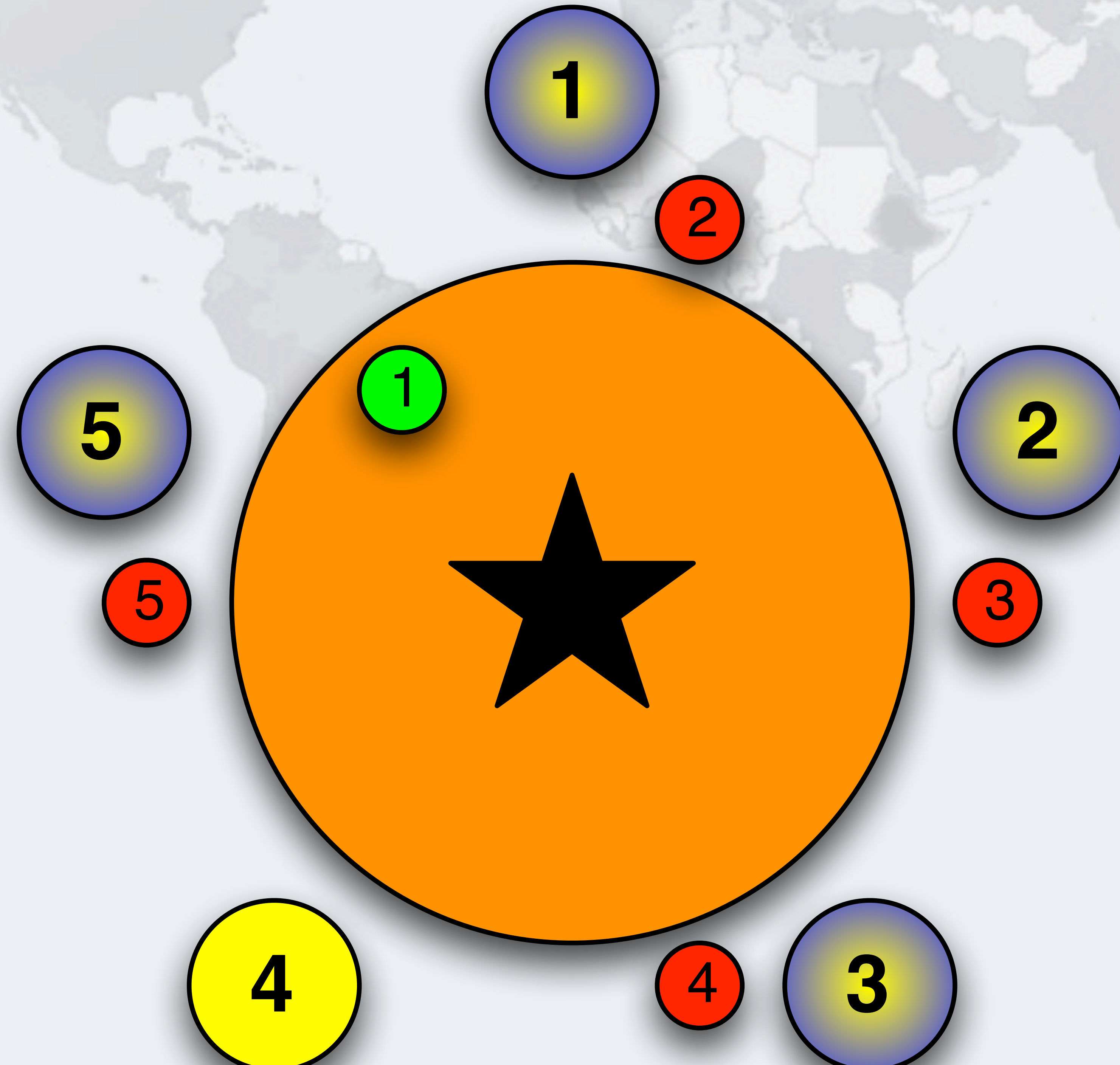
- **Cups are returned in the opposite order to what they are acquired**



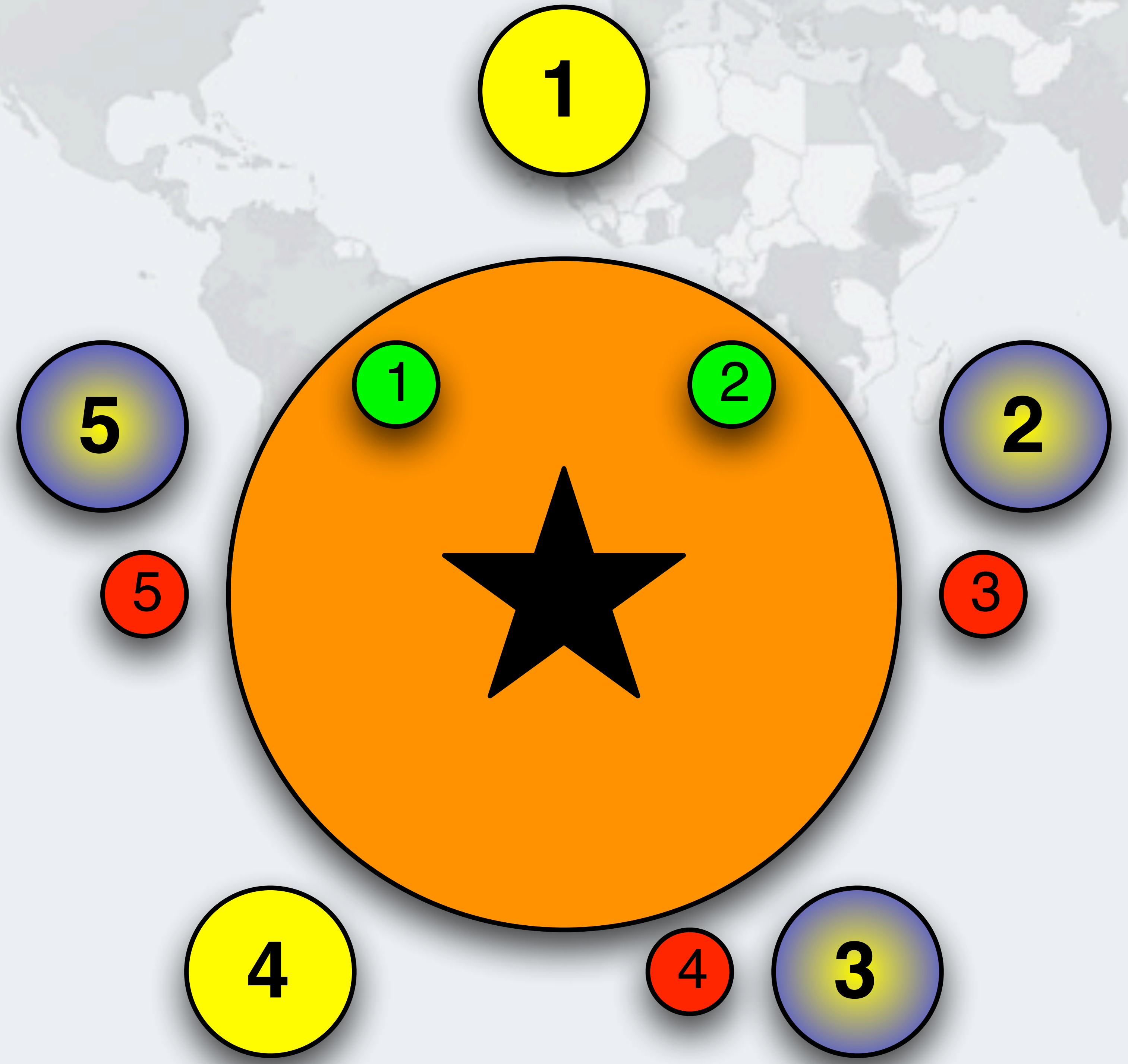
Philosopher 5 Takes Cup 1 - Drinking



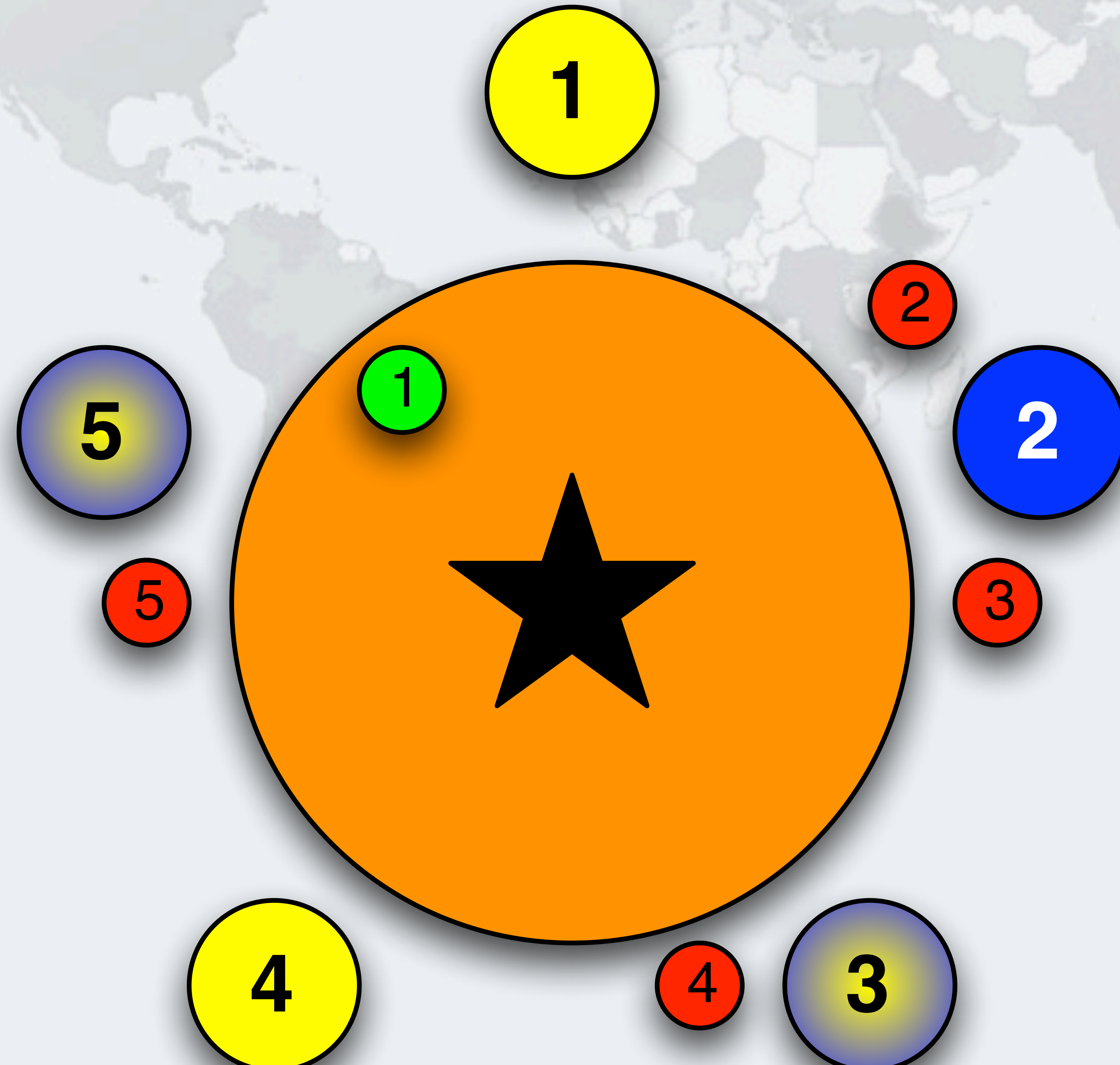
Philosopher 5 Returns Cup 1



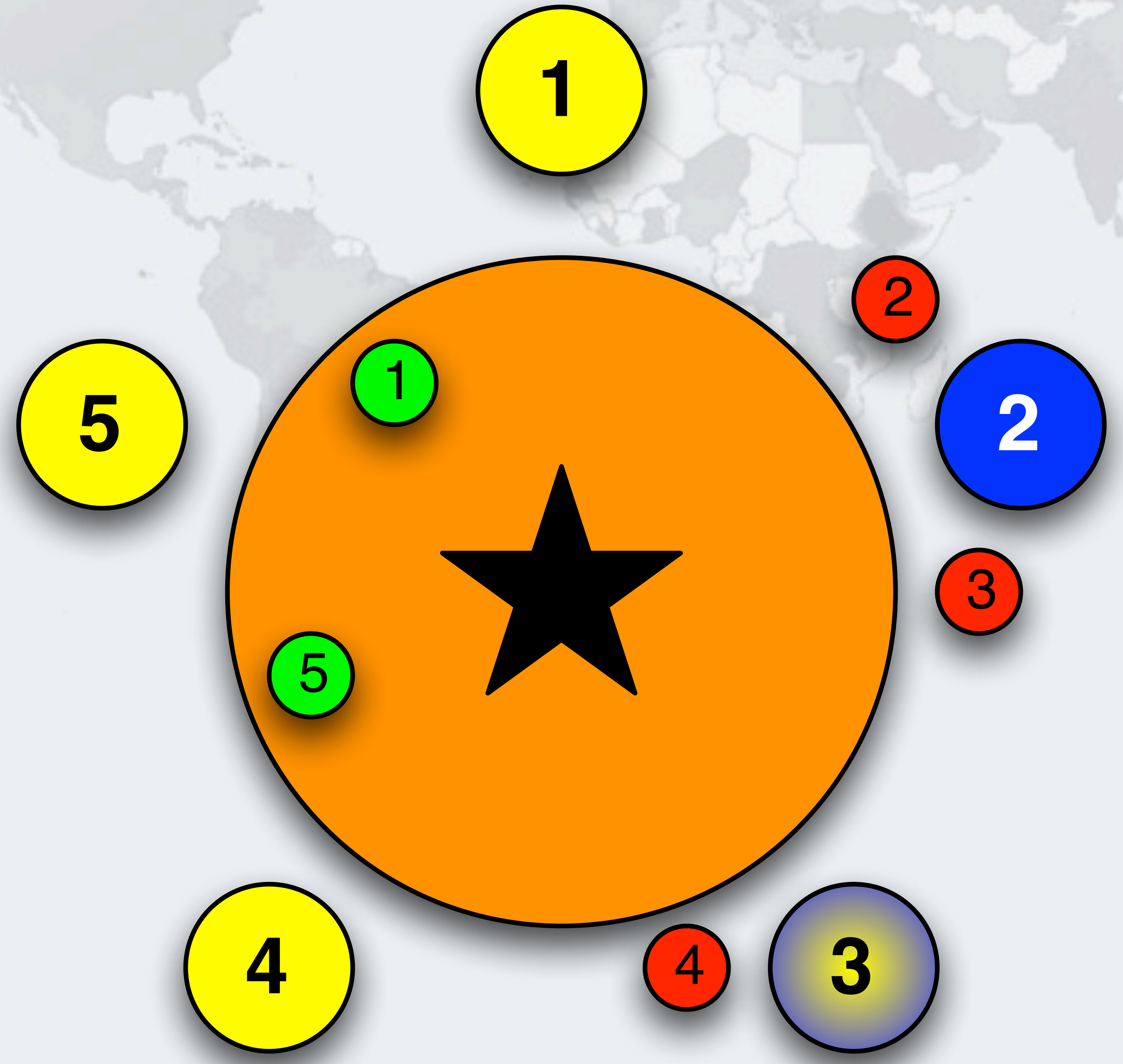
Philosopher 1 Returns Cup 2



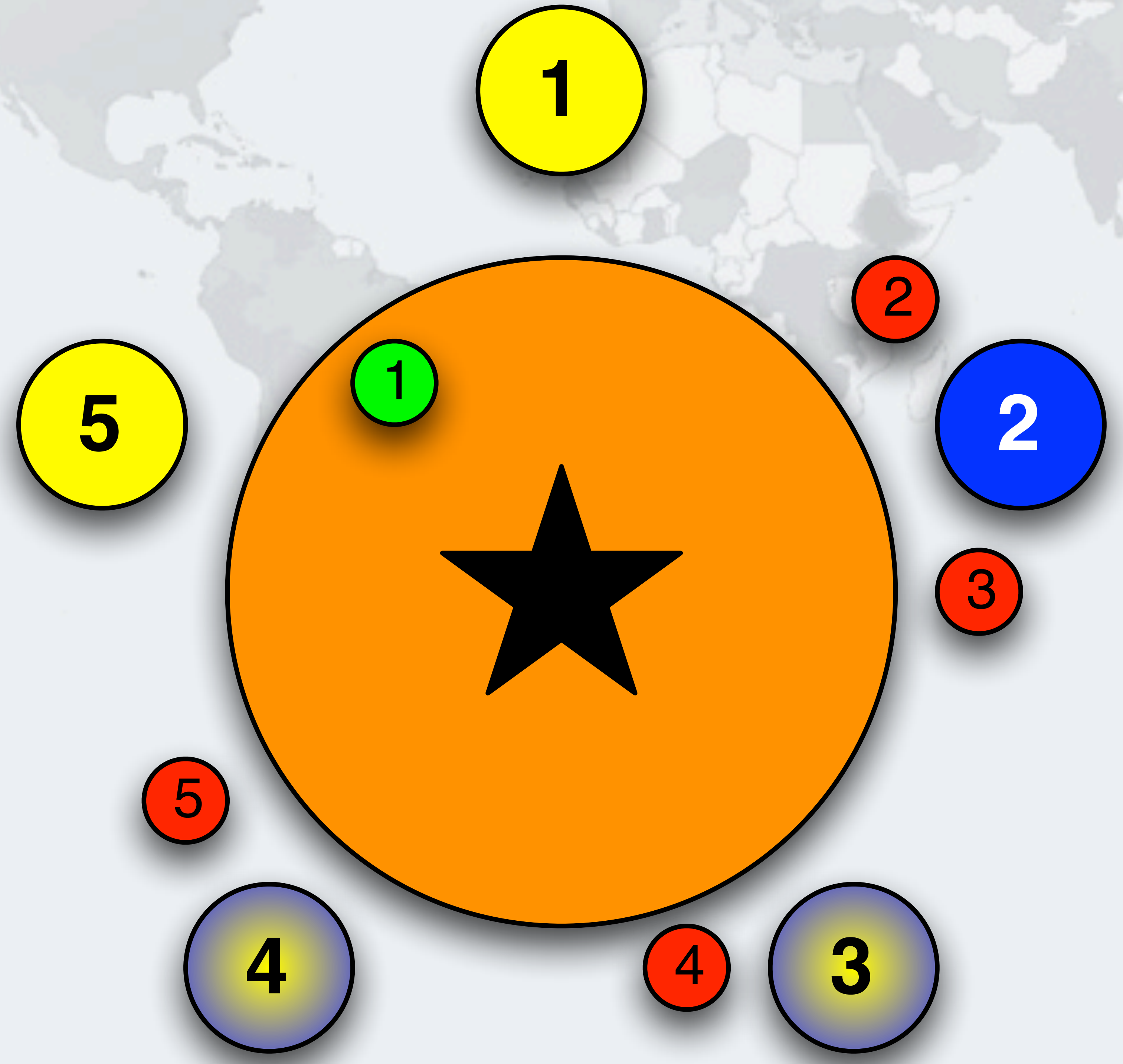
Philosopher 2 Takes Cup 2 - Drinking



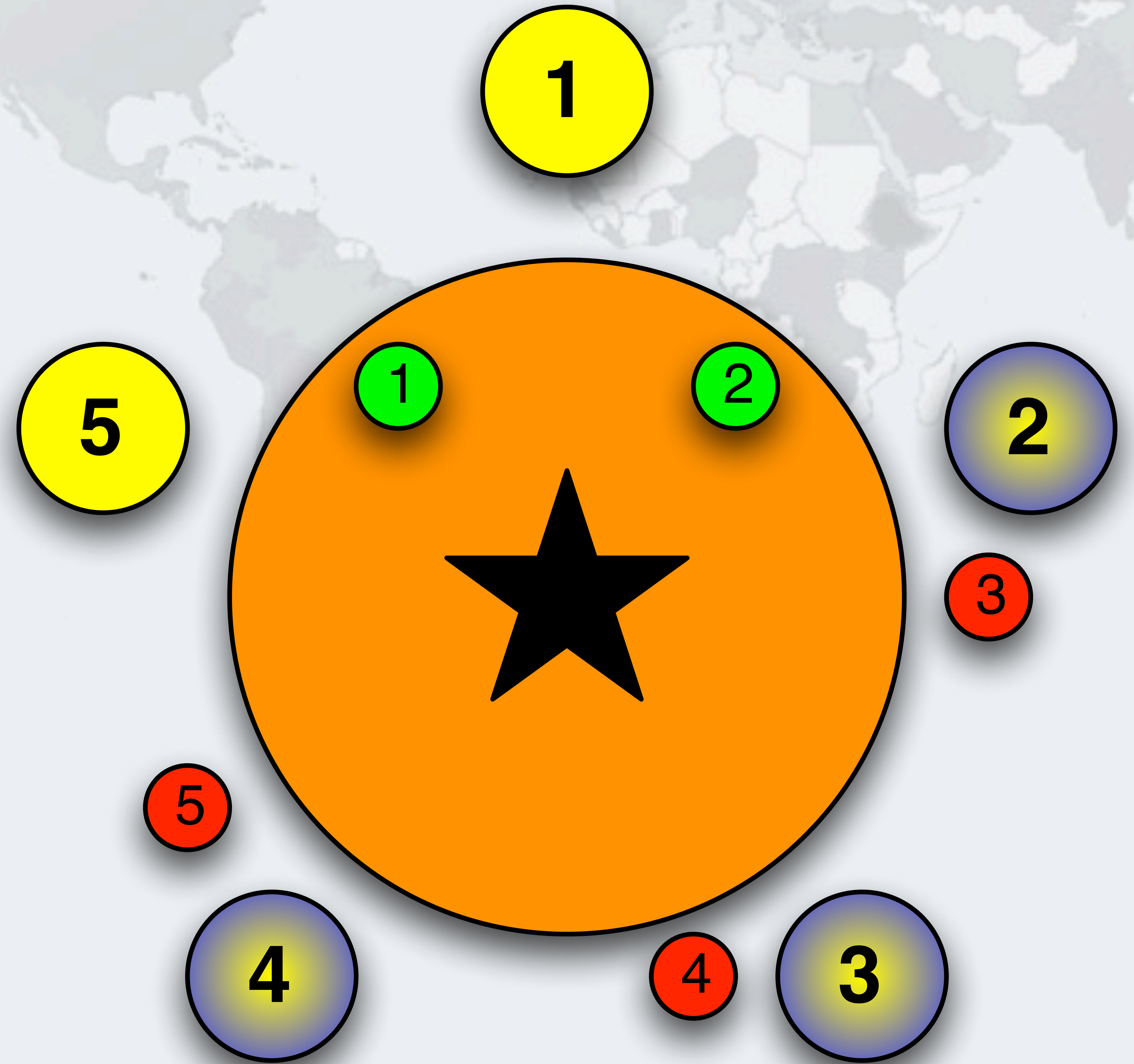
Philosopher 5 Returns Cup 5



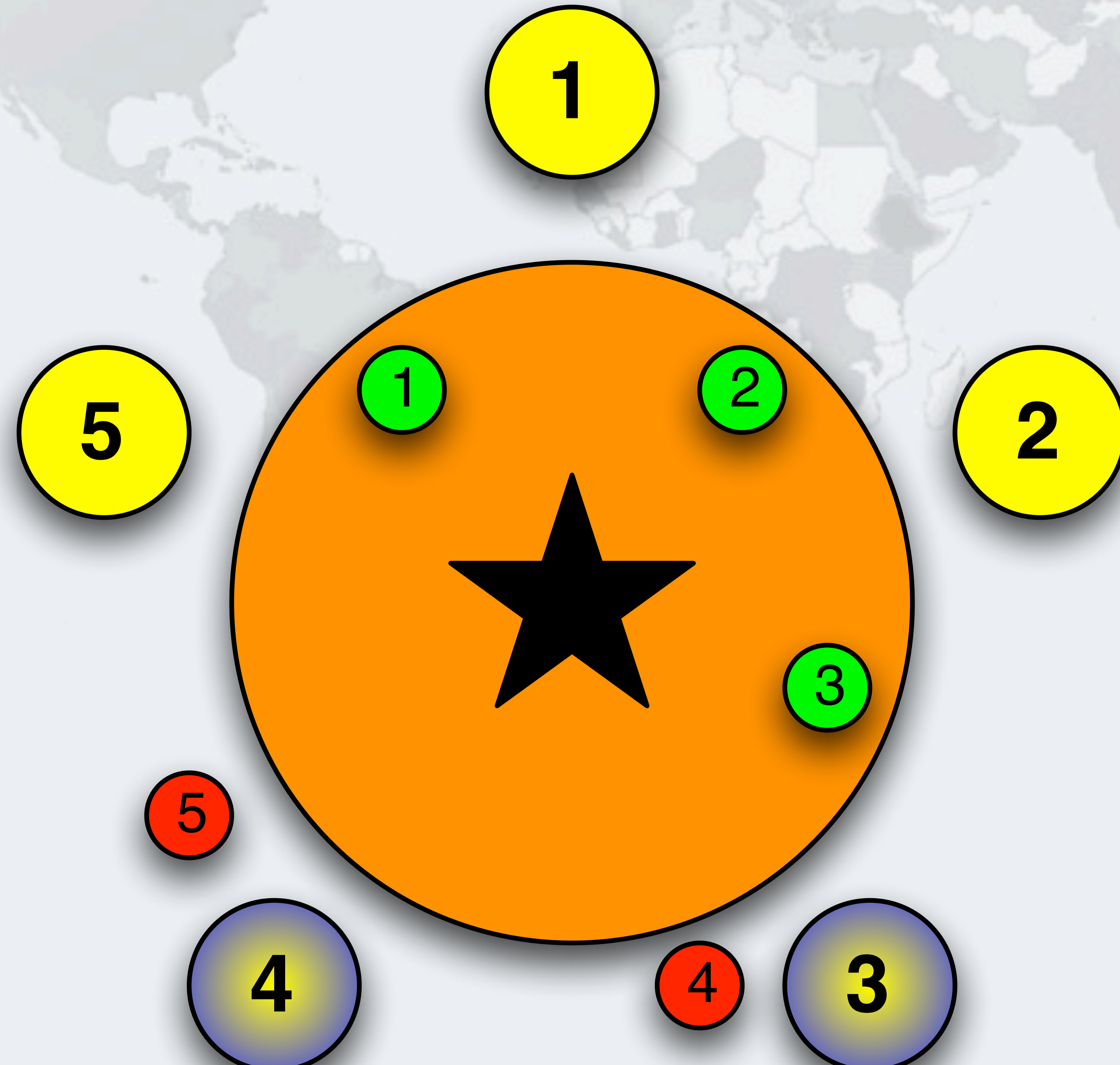
Philosopher 4 Takes Cup 5



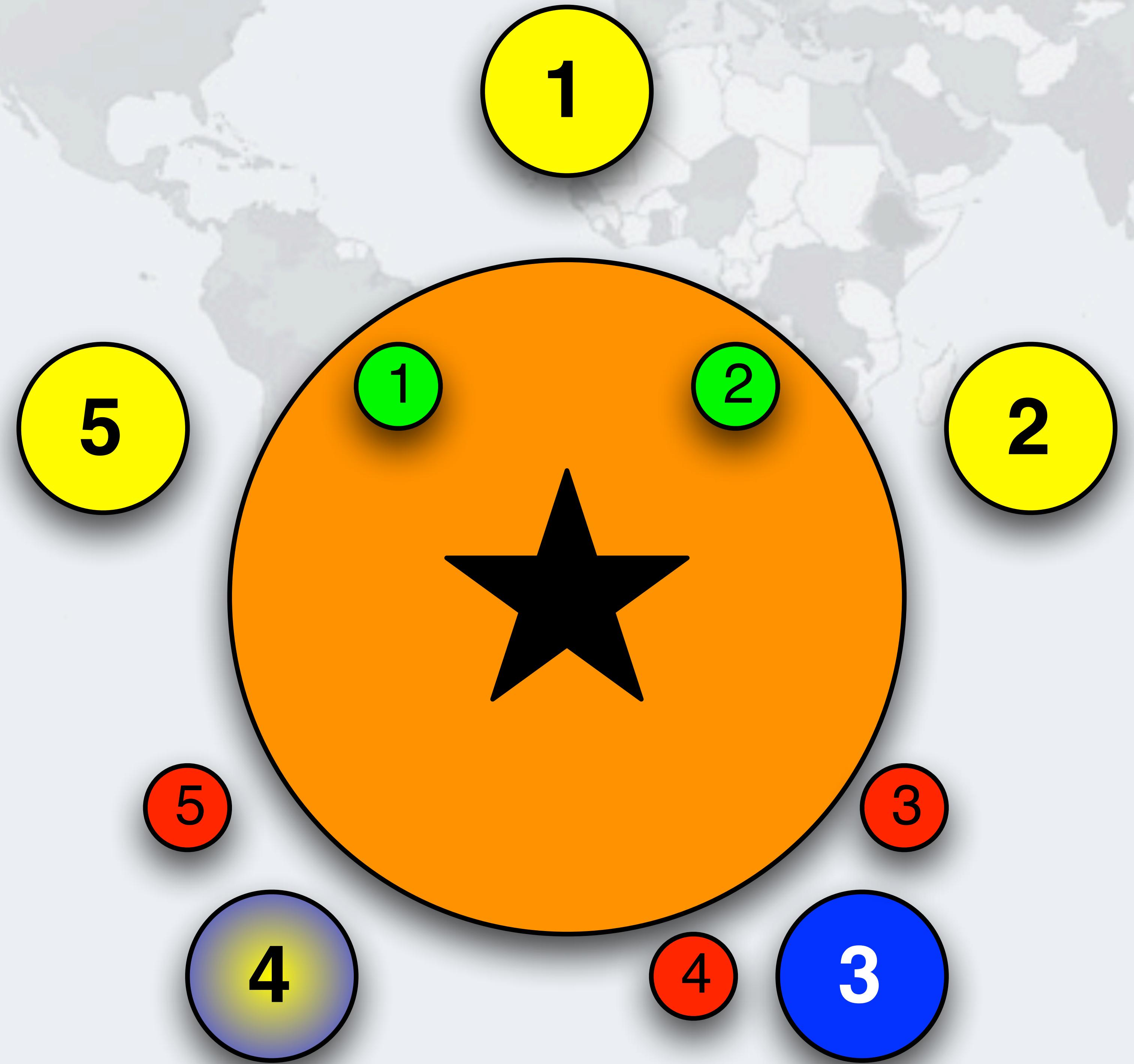
Philosopher 2 Returns Cup 2



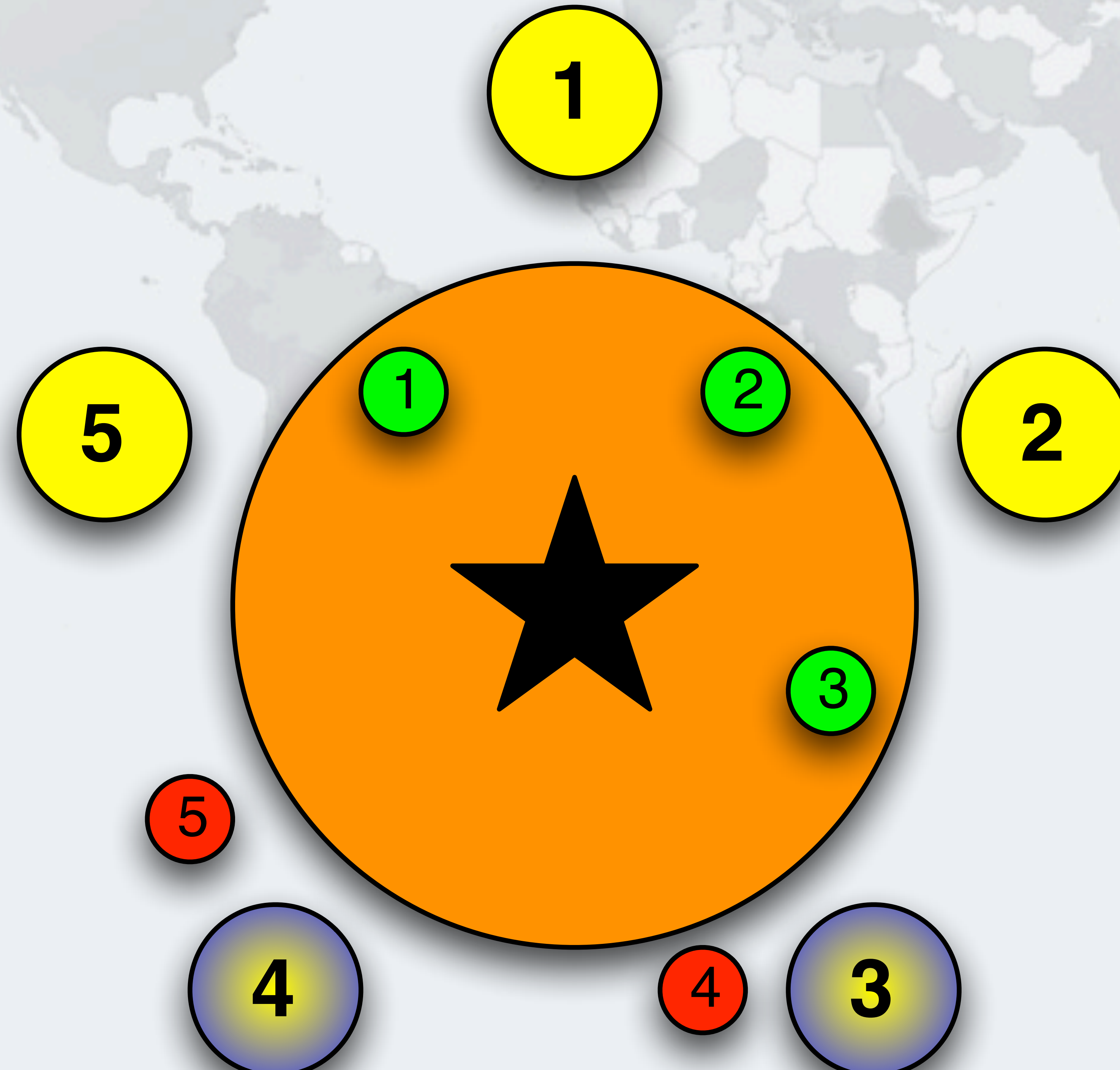
Philosopher 2 Returns Cup 3



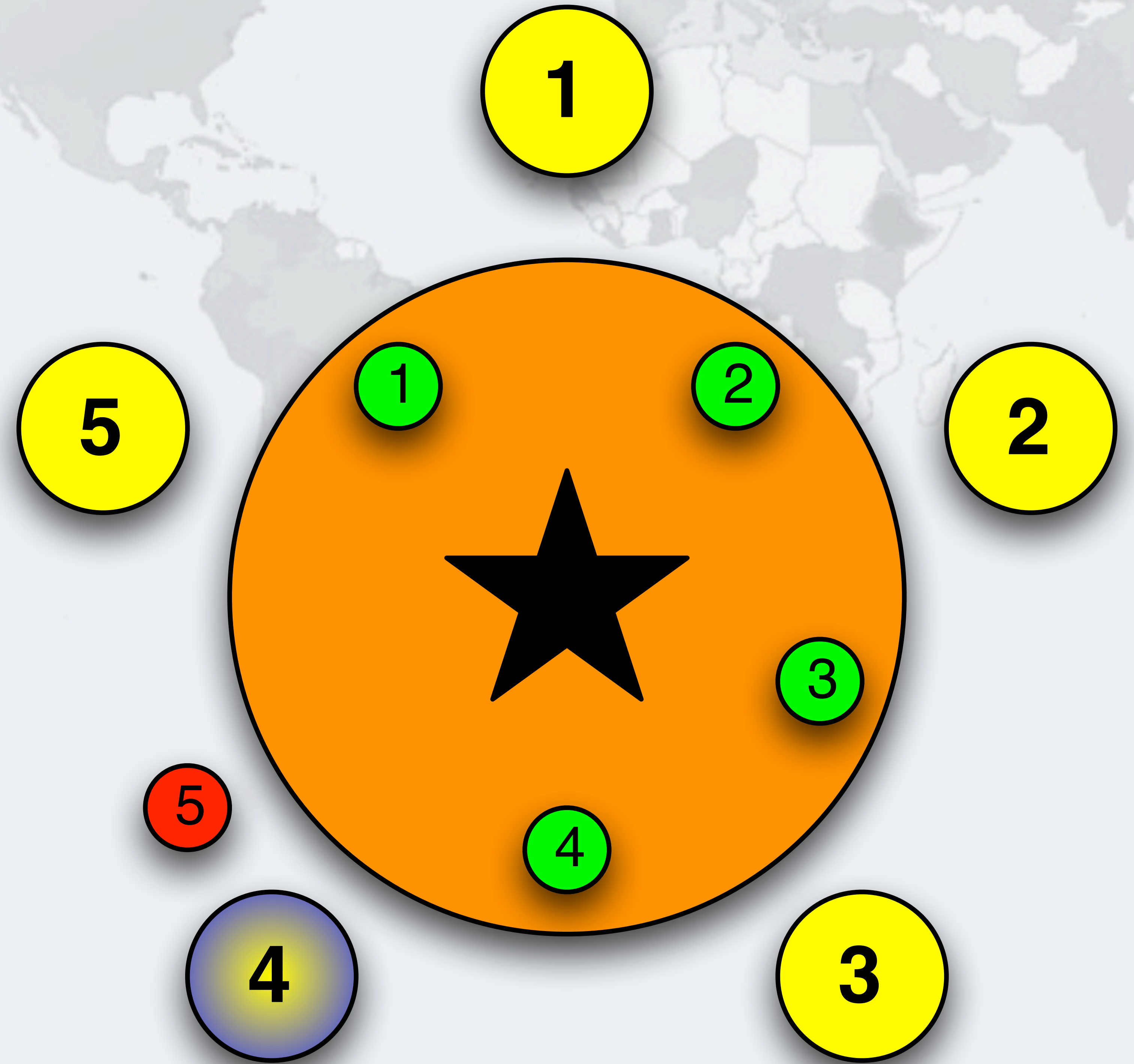
Philosopher 3 Takes Cup 3 - Drinking



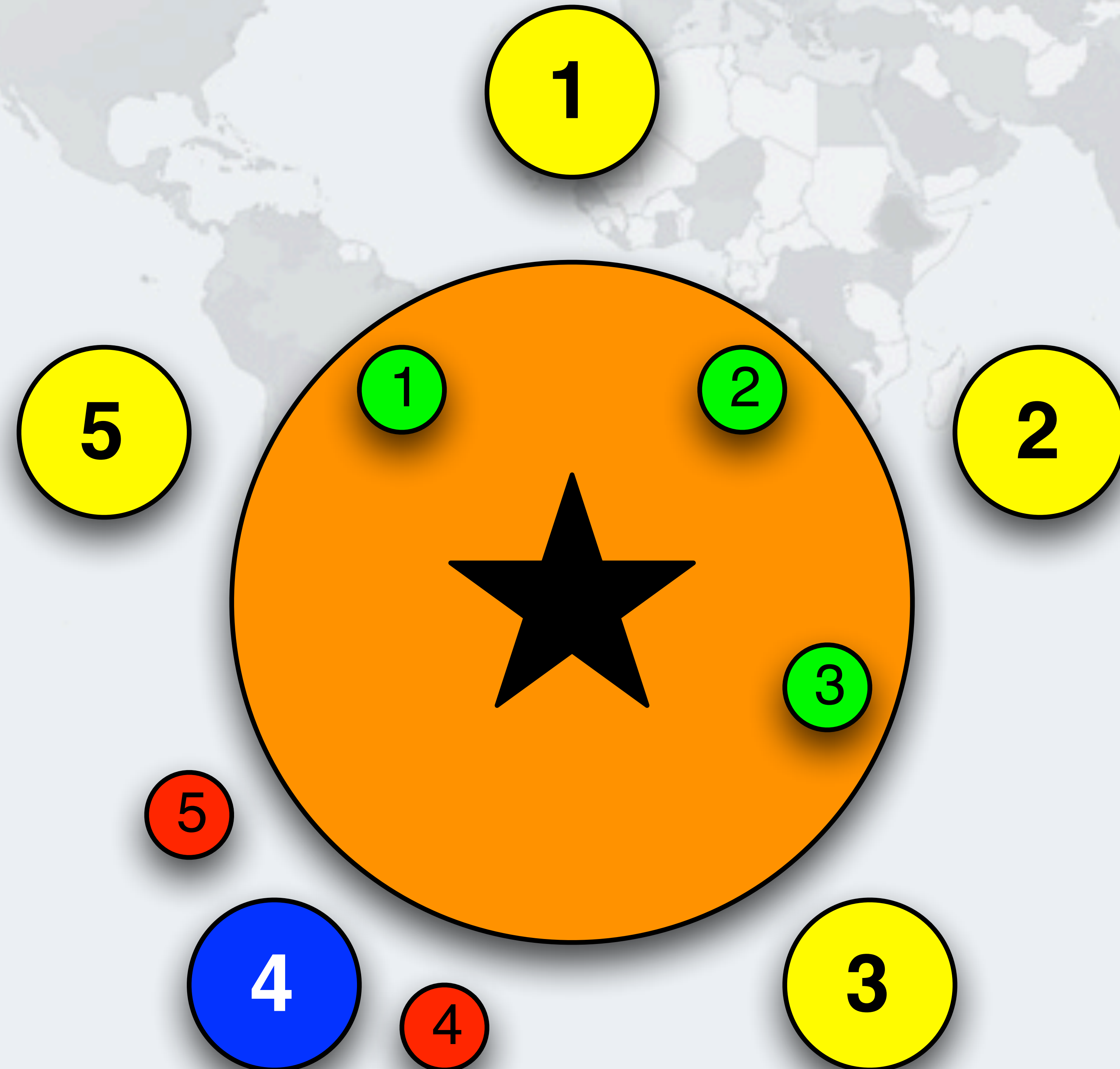
Philosopher 3 Returns Cup 3



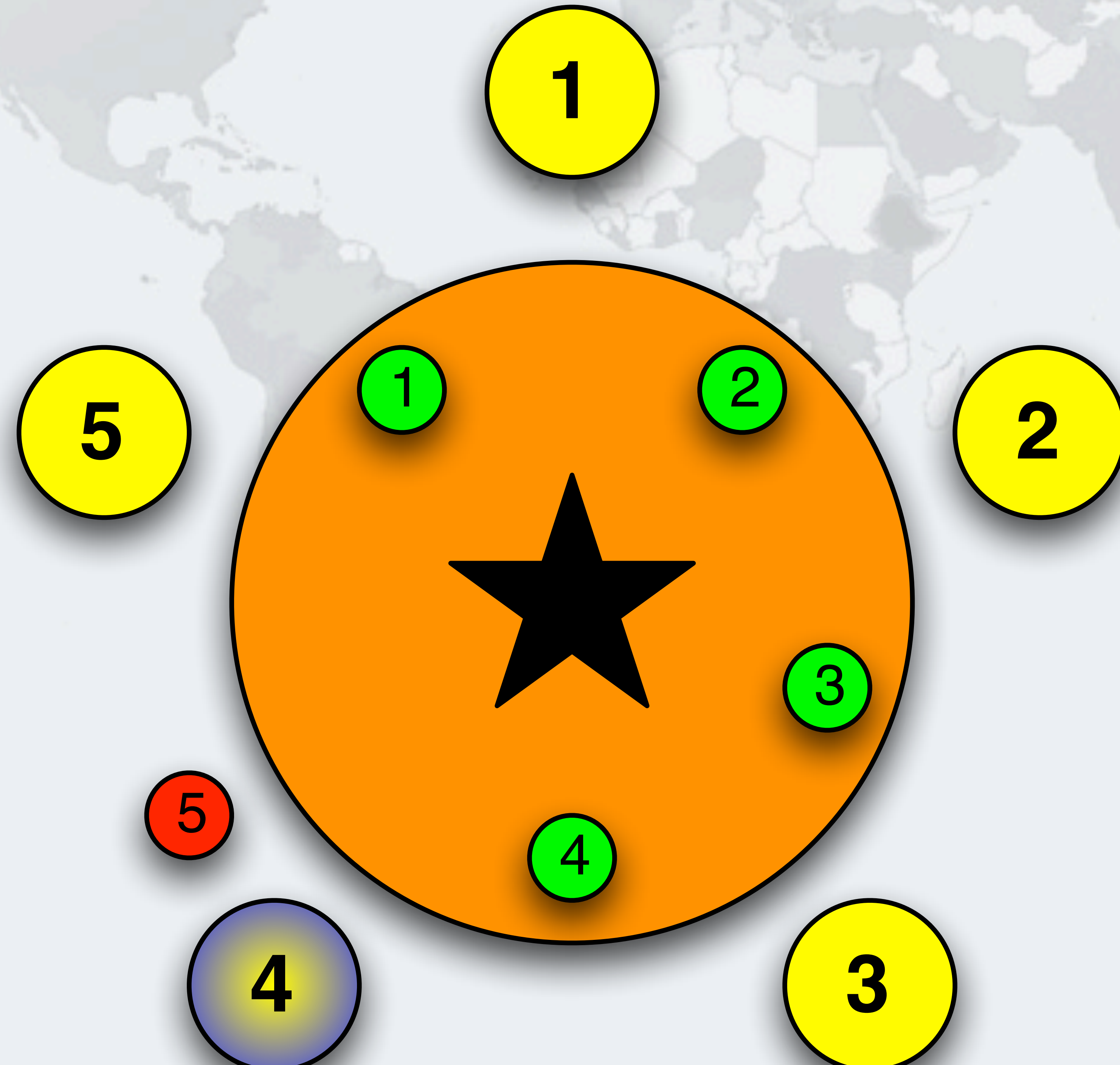
Philosopher 3 Returns Cup 4



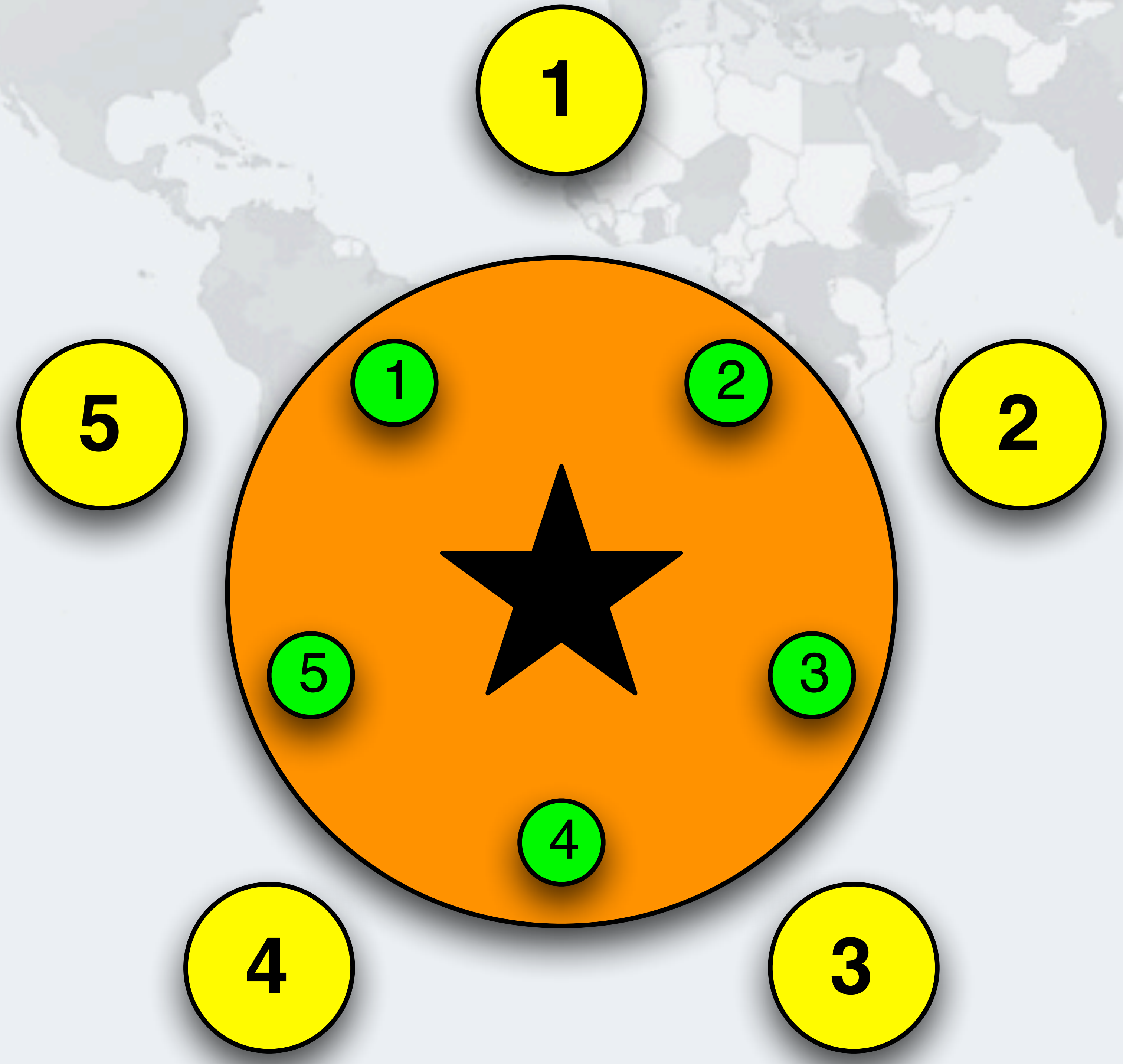
Philosopher 4 Takes Cup 4 - Drinking



Philosopher 4 Returns Cup 4



Philosopher 4 Returns Cup 5



Dynamic Lock Order Deadlocks

- **The LeftRightDeadlock example had an obvious deadlock**
- **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);
    }
  }
}
```

Checking Locks Are Held

- **In our doActualTransfer(), assert we hold both locks**

```
private boolean doActualTransfer(
    Account from, Account to, DollarAmount amount) {
  assert Thread.holdsLock(from);
  assert Thread.holdsLock(to);
  if (from.getBalance().compareTo(amount) >= 0) {
    from.debit(amount);
    to.credit(amount);
    return true;
  }
  return false;
}
```

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**

```
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;
  }
}
```

Deadlocks Between Cooperating Objects

- **In this example, the deadlock is more subtle**
  - **Taxi is an individual taxi with a location and a destination**
  - **Dispatcher represents a fleet of taxis**
- **Spot the deadlock**

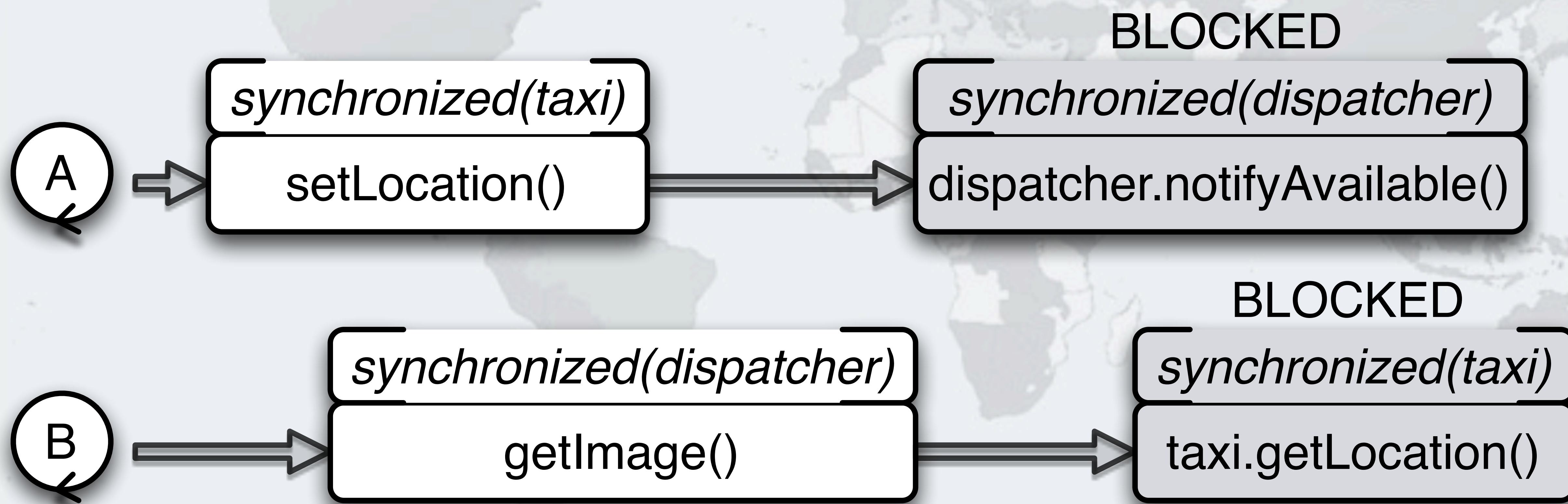
Taxi, Representing An Individual Vehicle

```
public class Taxi {
  @GuardedBy("this")
  private Point location, destination;
  private final Dispatcher dispatcher;

  public Taxi(Dispatcher dispatcher) {
    this.dispatcher = dispatcher;
  }

  public synchronized Point getLocation() {
    return location;
  }

  public synchronized void setLocation(
      Point location) {
    this.location = location;
    if (location.equals(destination))
      dispatcher.notifyAvailable(this);
  }
}
```

Dispatcher: Managing A Fleet Of Taxis

```
public class Dispatcher {
  @GuardedBy("this")
  private final Set<Taxi> taxis = new HashSet<>();
  @GuardedBy("this")
  private final Set<Taxi> availableTaxis = new HashSet<>();

  public synchronized void notifyAvailable(Taxi taxi) {
    availableTaxis.add(taxi);
  }

  public synchronized Image getImage() {
    Image image = new Image();
    for (Taxi taxi : taxis) {
      image.drawMarker(taxi.getLocation());
    }
    return image;
  }
}
```

How To Deadlock The Taxi Industry



- **Or in Greece you can simply announce that you will deregulate the taxi industry - that causes *real* deadlocks**
  - **In 2011, at height of tourist season, taxis went on strike for 3 weeks!**

Open Calls

- **Calling an *alien method* with a lock held is difficult to analyze and therefore risky**
- **Both Taxi and Dispatcher break this rule**
- **Calling a method with no locks held is called an *open call***
  - **Makes it much easier to reason about liveness**

Refactored Taxi.setLocation()

- **We should not call *alien methods* whilst holding locks**
- **Here we split the method up into parts that need the lock and those that call alien methods**

```
public void setLocation(Point location) {
  boolean reachedDestination;
  synchronized (this) {
    this.location = location;
    reachedDestination = location.equals(destination);
  }
  if (reachedDestination) {
    dispatcher.notifyAvailable(this);
  }
}
```

Refactored Dispatcher.getImage()

- **We make a copy of the set to prevent race conditions**

```
public Image getImage() {
  Set<Taxi> copy;
  synchronized (this) {
    copy = new HashSet<>(taxis);
  }
  Image image = new Image();
  for (Taxi taxi : copy) {
    image.drawMarker(taxi.getLocation());
  }
  return image;
}
```

Benefit Of Open Calls

- **Strive to use open calls throughout your program**
- **Programs that rely on open calls are far easier to analyze for deadlock-freedom than those that allow calls to alien methods with locks held**
- **Alien method calls with lock held are probably the biggest cause of deadlocks "in the field"**

Open Call In Vector

- **In Sun Java 6 Vector.writeObject() method is synchronized**
  - **This is to provide thread safety during writing**

```
private synchronized void writeObject(ObjectOutputStream s)
    throws IOException {
  s.defaultWriteObject();
}
```

  - **However, since it calls the alien "defaultWriteObject()" it can deadlock**
    - **http://www.javaspecialists.eu/archive/Issue184.html**

IBM Avoids This Problem With An Open Call

```
private void writeObject(ObjectOutputStream stream)
    throws IOException {
  Vector<E> cloned = null;
  // this specially fix is for a special dead-lock in customer
  // program: two vectors refer each other may meet dead-lock in
  // synchronized serialization. Refer CMVC-103316.1
  synchronized (this) {
    try {
      cloned = (Vector<E>) super.clone();
      cloned.elementData = elementData.clone();
    } catch (CloneNotSupportedException e) {
      // no deep clone, ignore the exception
    }
  }
  cloned.writeObjectImpl(stream);
}

private void writeObjectImpl(ObjectOutputStream stream)
   throws IOException {
 stream.defaultWriteObject();
}
```

OpenJDK 7 Also Uses An Open Call

```
private void writeObject(ObjectOutputStream s)
    throws IOException {
  final ObjectOutputStream.PutField fields = s.putFields();
  final Object[] data;
  synchronized (this) {
    fields.put("capacityIncrement", capacityIncrement);
    fields.put("elementCount", elementCount);
    data = elementData.clone();
  }
  fields.put("elementData", data);
  s.writeFields();
}
```

Resource Deadlocks

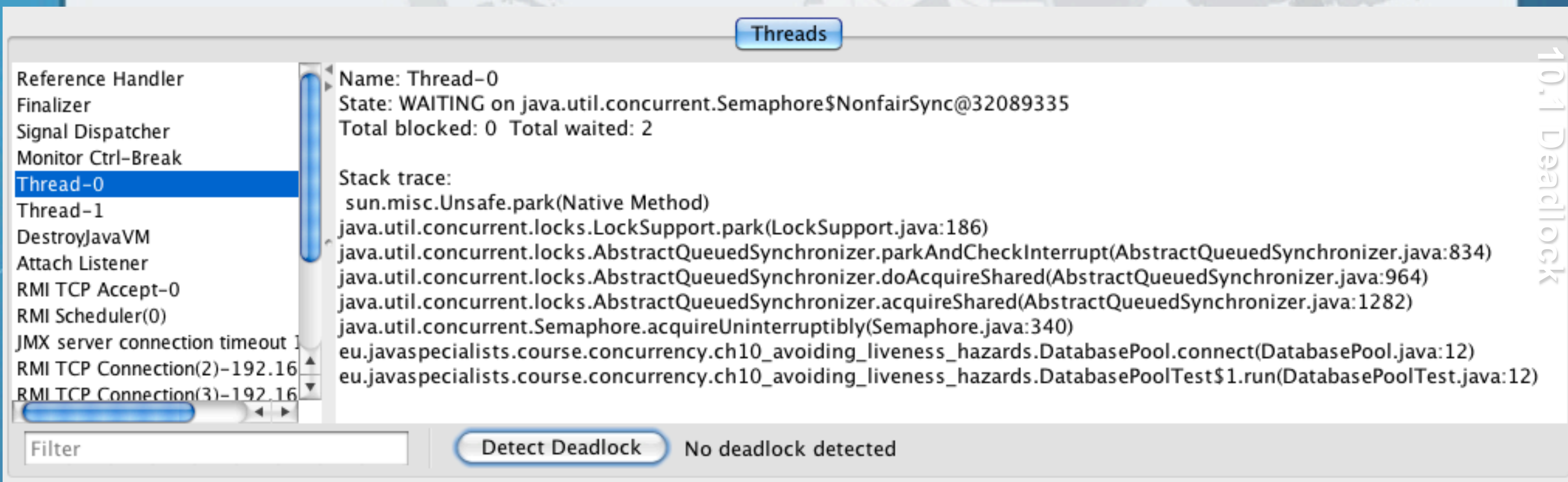
- **We can also cause deadlocks waiting for resources**
- **For example, say you have two DB connection pools**
  - **Some tasks might require connections to both databases**
  - **Thus thread A might hold semaphore for D1 and wait for D2, whereas thread B might hold semaphore for D2 and be waiting for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

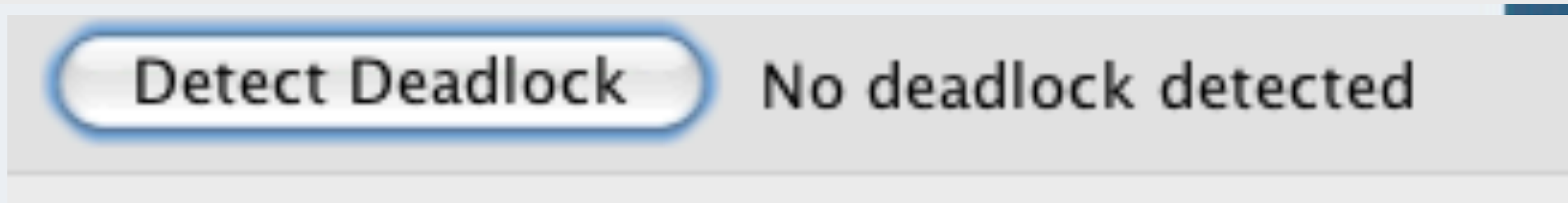
Our DatabasePool - Connect() And Disconnect()

```
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
```





10.1 Deadlock

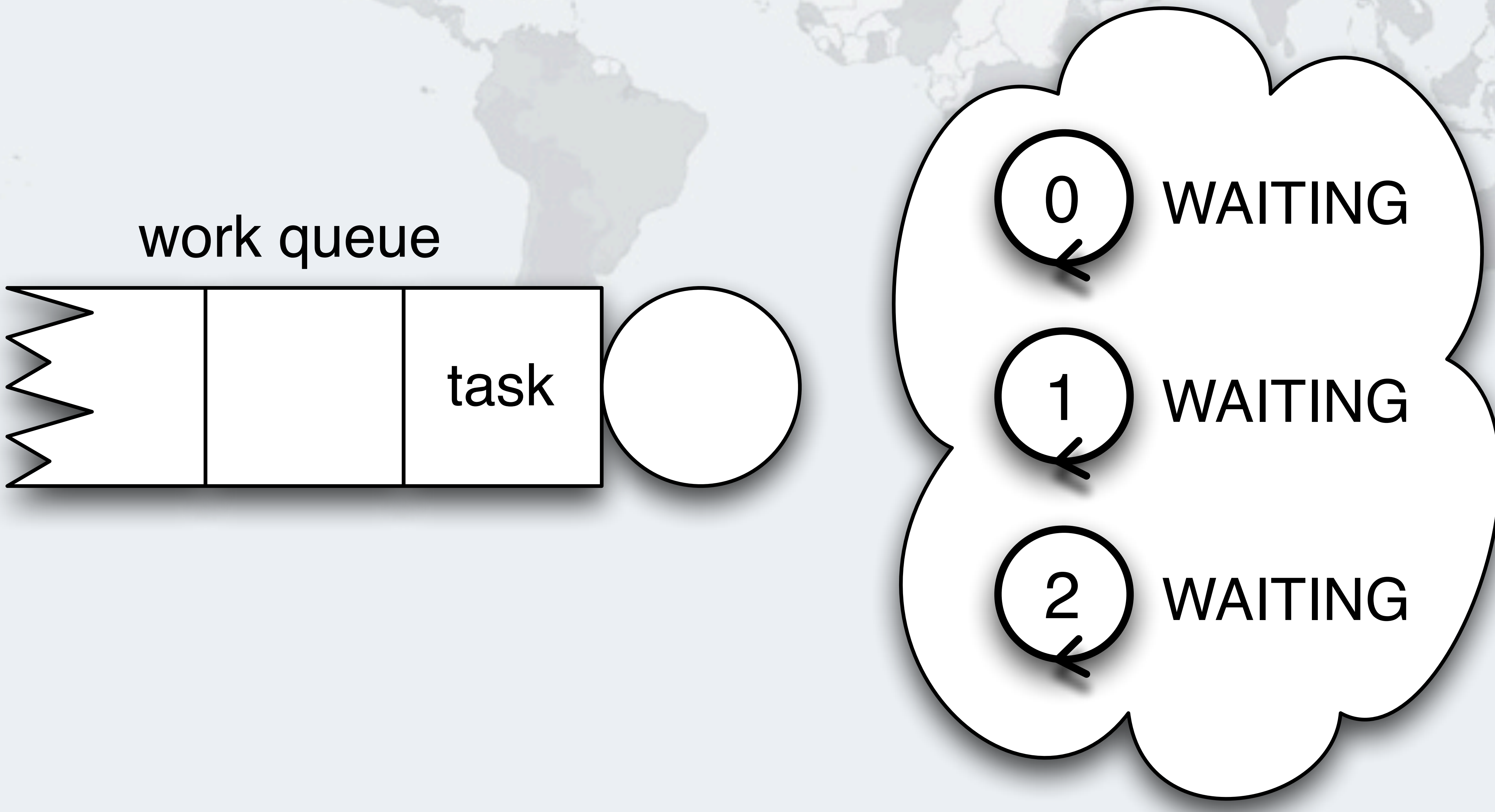
Dependent Tasks Causing Liveness Issues

- **Tasks that depend on others in pool can cause a thread-starvation deadlock**

```
ExecutorService pool = Executors.newFixedThreadPool(3);
final CountDownLatch latch = new CountDownLatch(4);
for (int i = 0; i < 4; i++) {
  pool.submit(new Runnable() {
    public void run() {
      System.out.println("countdown");
      latch.countDown();
      try {
        System.out.println("waiting");
        latch.await();
      } catch (InterruptedException e) {
        System.out.println("interrupting");
        Thread.currentThread().interrupt();
      }
      System.out.println("done");
    }
  });
}
```

Thread Pool Blocked Up

- **All the threads are waiting for "task" to be completed**
  - **Bounded thread pools and bounded queues can cause deadlocks**



10.2 Avoiding And Diagnosing Deadlocks

Avoiding Liveness Hazards

10.2 Avoiding And Diagnosing Deadlocks

- **If you only ever acquire one lock, you cannot get a lock-ordering deadlock**
  - **This is the easiest way to avoid deadlocks, but not always practical**
- **If you need to acquire multiple locks, include lock ordering in your design**
  - **Important to specify and document possible lock sequences**
  - **Identify where multiple locks could be acquired**
  - **Do a global analysis to ensure that lock ordering is consistent**
    - **This can be extremely difficult in large programs**
- **Use open calls whenever possible**
  - **Do not call alien methods whilst holding a lock**

Unit Testing For Lock Ordering Deadlocks

- Code typically has to be called many times before a deadlock happens
- How many times do you need to call it to prove that there is no deadlock?
  - Nondeterministic unit tests are bad - they should either always pass or always fail

Adding A Sleep To Cause Deadlocks

- **In the transferMoney() method, a deadlock occurs if after the first lock is granted, the first thread is swapped out and another thread requests the second lock**
- **We can force this to happen by sleeping a short while after requesting the first lock**

```
public class Bank {
  public boolean transferMoney(Account from, Account to,
                               DollarAmount amount) {
    synchronized (from) {
      sleepAWhileForTesting();
      synchronized (to) {
        return doActualTransfer(from, to, amount);
      }
    }
  }
  protected void sleepAWhileForTesting() {}
}
```

In Our Unit Test We Override The Class

- **We make the sleepAWhileForTesting() method sleep**
  - **In production, when we use only the normal Bank, the empty method will be optimized away by the HotSpot compiler**

```
public class SlowBank extends Bank {
  private final long timeout;
  private final TimeUnit unit;
  public SlowBank(long timeout, TimeUnit unit) {
    this.timeout = timeout;
    this.unit = unit;
  }
  protected void sleepAWhileForTesting() {
    try {
      unit.sleep(timeout);
    } catch (InterruptedException e) {
      Thread.currentThread().interrupt();
    }
  }
}
```

Verifying Thread Deadlocks

- **ThreadMXBean has two methods for finding deadlocks**
  - **findMonitorDeadlockedThreads()**
    - **Includes only "monitor" locks, i.e. synchronized**
    - **Only way to find deadlocks in Java 5**
  - **findDeadlockedThreads()**
    - **Includes "monitor" and "owned" (Java 5) locks**
    - **Preferred method to test for deadlocks**
    - **But, does *not* find deadlocks between semaphores**
  - **See http://www.javaspecialists.eu/archive/Issue130.html**

```
public class BankDeadlockTest {
  private final static ThreadMXBean tmb =
      ManagementFactory.getThreadMXBean();

  private void checkThatThreadTerminates(Thread thread)
      throws InterruptedException {
    for (int i = 0; i < 2000; i++) {
      thread.join(50);
      if (!thread.isAlive()) return;
      if (isThreadDeadlocked(thread.getId())) {
        fail("Deadlock detected!");
      }
    }
    fail(thread + " did not terminate in time");
  }

  private boolean isThreadDeadlocked(long tid) {
    long[] ids = tmb.findDeadlockedThreads();
    if (ids == null) return false;
    for (long id : ids) {
      if (id == tid) return true;
    }
    return false;
  }
```

```
  @Test
  public void testForTransferDeadlock()
      throws InterruptedException {
    final Account alpha = new Account(new DollarAmount(1000));
    final Account ubs = new Account(new DollarAmount(1000000));
    final Bank bank = new SlowBank(100, TimeUnit.MILLISECONDS);

    Thread alphaToUbs = new Thread("alphaToUbs") {
      public void run() {
        bank.transferMoney(alpha, ubs, new DollarAmount(100));
      }
    };
    Thread ubsToAlpha = new Thread("ubsToAlpha") {
      public void run() {
        bank.transferMoney(ubs, alpha, new DollarAmount(100));
      }
    };

    alphaToUbs.start();
    ubsToAlpha.start();

    checkThatThreadTerminates(alphaToUbs);
  }
}
```

Output With Broken TransferMoney() Method

- **We see the deadlock within about 100 milliseconds**

```
junit.framework.AssertionFailedError: Deadlock detected!
  at BankDeadlockTest.checkThatThreadTerminates(BankDeadlockTest.java:20)
  at BankDeadlockTest.testForTransferDeadlock(BankDeadlockTest.java:55)
```

- **If we fix the transferMoney() method, it also completes within about 100 milliseconds**
  - **This is the time that we are sleeping for testing purposes**
- **Remember that the empty sleepAWhileForTesting() method will be optimized away by HotSpot**

Timed Lock Attempts

- **Another technique for solving deadlocks is to use the timed tryLock() method of Java 5 locks (more in ch 13)**
- **Two things to consider**
  - **When a timed lock attempt fails, we do not necessarily know *why***
    - **Could be deadlock**
    - **Could be another thread holding the lock whilst in an infinite loop**
    - **Could be some thread just taking a lot longer than expected**
  - **ThreadMXBean will show the thread as *deadlocked* whilst it is waiting for the lock**

Deadlock Analysis With Thread Dumps

- **The ThreadMXBean can be invoked directly to find deadlocks between monitors or Java 5 locks**
- **However, we can also cause a thread dump in many ways:**
  - **Ctrl+Break on Windows or Ctrl-\ on Unix**
  - **Invoking "kill -3" on the process id**
  - **Calling jstack on the process id**
    - **Only shows deadlocks since Java 6**
- **Intrinsic locks typically show more information of where they were acquired than the explicit Java 5 locks**

Deadlock Analysis With Thread Dumps

- **Thread dump from a real system (names changed)**
- **It is useful to have unique threads names**
- **The stack trace confirms the deadlock**

```
Found one Java-level deadlock:
=============================
"ApplicationServerThread-0":
  waiting to lock monitor 0x080f0cdc
    (object 0x650f7f30, a MumbleDBConnection),
  which is held by "ApplicationServerThread-1"

"ApplicationServerThread-1":
  waiting to lock monitor 0x080f0ed4
    (object 0x6024ffb0, a MumbleDBCallableStatement),
  which is held by "ApplicationServerThread-0"
```

Stack Information Shows Where It Comes From

```
Java stack information for the threads listed above:
===================================================
"ApplicationServerThread-0":
    at MumbleDBConnection.remove_statement
    - waiting to lock <0x650f7f30> (a MumbleDBConnection)
    at MumbleDBStatement.close
    - locked <0x6024ffb0> (a MumbleDBCallableStatement)
    ...

"ApplicationServerThread-1":
    at MumbleDBCallableStatement.sendBatch
    - waiting to lock <0x6024ffb0>
                        (a MumbleDBCallableStatement)
    at MumbleDBConnection.commit
    - locked <0x650f7f30> (a MumbleDBConnection)
    ...

Found 1 deadlock.
```

What Caused The Deadlock?

- **Inside the JDBC driver, different calls acquired locks in different orders**
  - **JDBC vendor was trying to build a thread-safe driver**
    - **But then ended up writing a potential deadlock**
  - **This could be fixed in the JDBC driver by imposing a global order**
- **However, in the system the JDBC connection was shared by multiple threads**
  - **This caused the bug to appear**
- **Solution: single threaded access to each individual connection**

Stopping Deadlock Victims

- **In extreme situations threads that are deadlocked in the WAITING state can be stopped as deadlock victims**
- **This only works with "owned" Java 5 locks, not monitors**
  - **A thread in the BLOCKED state cannot be stopped**
- **We can throw a special exception with Thread.stop()**

```
public class DeadlockVictimError extends Error {
  private final Thread victim;
  public DeadlockVictimError(Thread victim) {
    super("Deadlock victim: " + victim);
    this.victim = victim;

  }
  public Thread getVictim() { return victim; }
}
```

```
public class DeadlockArbitrator {
  private static final ThreadMXBean tmb =
      ManagementFactory.getThreadMXBean();

  public boolean tryResolveDeadlock() throws InterruptedException {
    return tryResolveDeadlock(3, 1, TimeUnit.SECONDS);
  }

  public boolean tryResolveDeadlock(
      int attempts, long timeout, TimeUnit unit)
      throws InterruptedException {
    for (int i = 0; i < attempts; i++) {
      long[] ids = tmb.findDeadlockedThreads();
      if (ids == null) return true;
      Thread t = findThread(ids[i % ids.length]);
      if (t == null)
        throw new IllegalStateException("Could not find thread");
      t.stop(new DeadlockVictimError(t));
      unit.sleep(timeout);
    }
    return false;
  }

  private Thread findThread(long id) {
    for (Thread thread : Thread.getAllStackTraces().keySet()) {
      if (thread.getId() == id) return thread;
    }
    return null;
  }
}
```

Applicability Of DeadlockArbitrator

- **Only use in extreme circumstances**
  - **Code that is outside your control and that deadlocks**
  - **Where you cannot prevent the deadlock**
- **Remember, it only works with Java 5 locks (more later)**

10.3 Other Liveness Hazards

Avoiding Liveness Hazards

10.3 Other Liveness Hazards

- **Deadlock is the most common liveness hazard**
  - **Even though there is no way to cleanly recover, it is usually fairly easy to recognize with the thread dumps**
- **However, other liveness hazards can be more difficult to find, for example**
  - **Starvation**
  - **Missed signals (covered in Chapter 14)**
  - **Livelock**

Threading Problems – Starvation

- **In concurrent applications, a thread could perpetually be denied resources.**
- **Starvation can cause OutOfMemoryError or prevent a program from ever completing.**

Starvation In Java

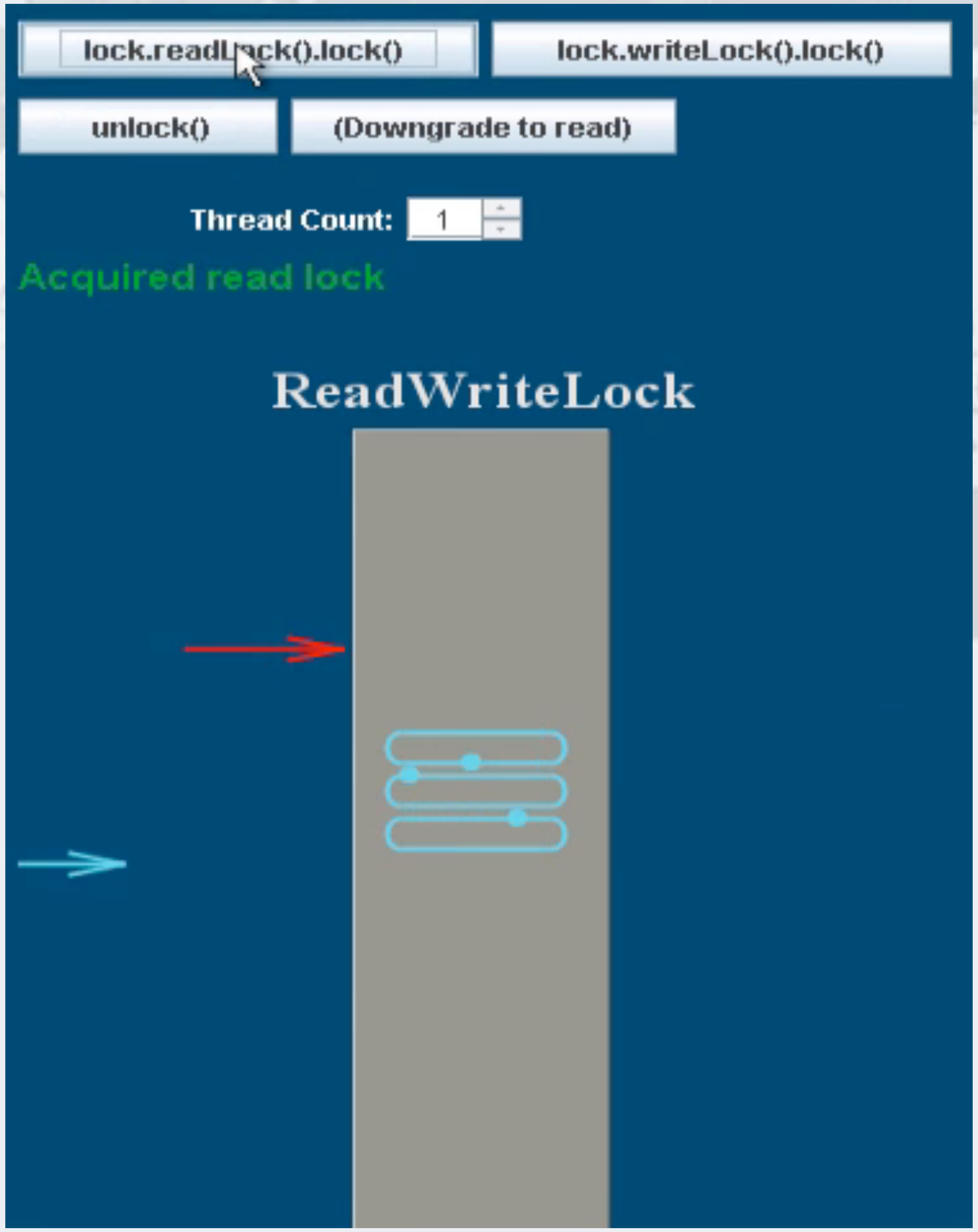
- **Most common situation is when some low priority thread is ignored for long periods of time, preventing it from ever finishing work**
- **In Java, thread priorities are just a hint for the operating system. The mapping to system priorities is system dependent**
- **Tweaking thread priorities might result in starvation**

ReadWriteLock Starvation

- **When readers are given priority, then writers might never be able to complete (Java 5)**
- **But when writers are given priority, readers might be starved (Java 6)**
- **Only use ReadWriteLock when you are sure that you will not continuously be acquiring locks**
- **See http://www.javaspecialists.eu/archive/Issue165.html**

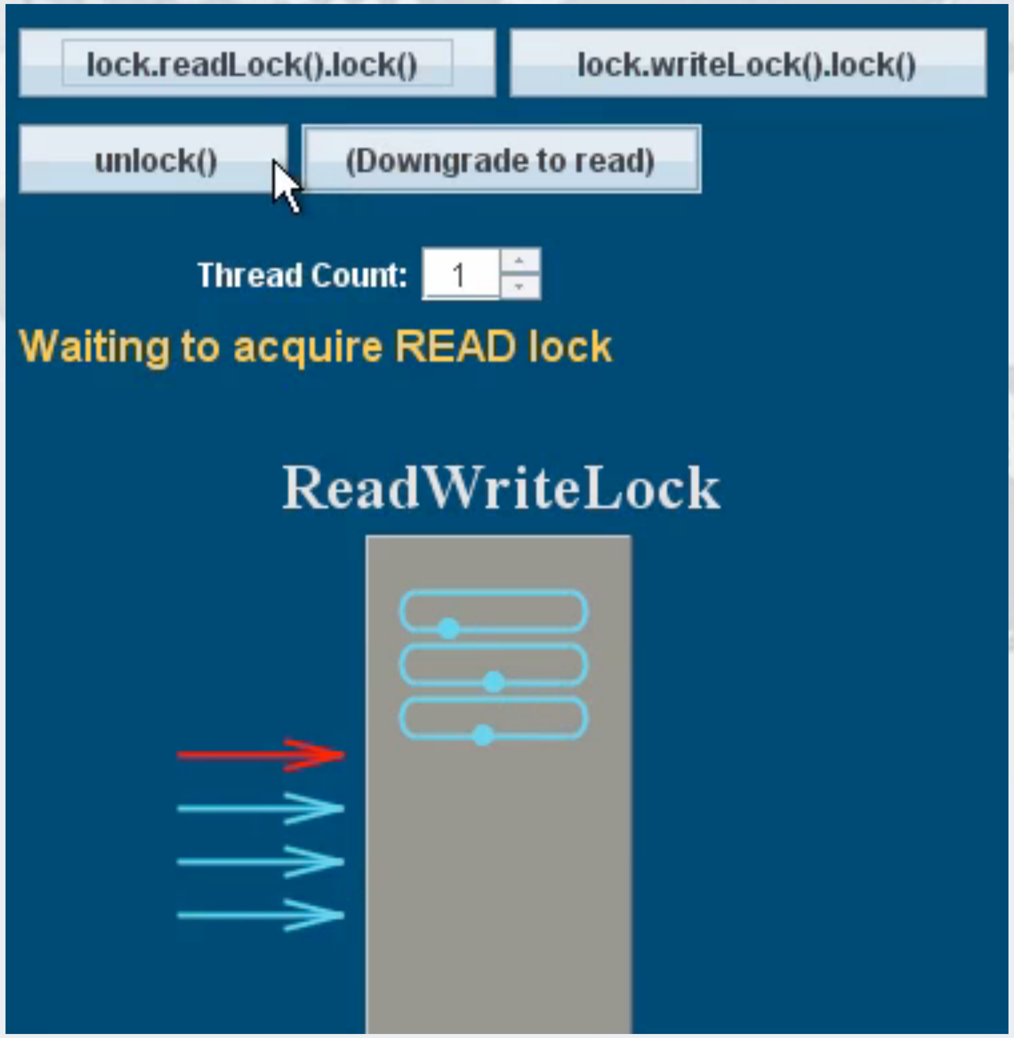
Java 5 ReadWriteLock Starvation

- We first acquire some read locks
- We then acquire one write lock
- Despite write lock waiting, read locks are still issued
- If enough read locks are issued, write lock will never get a chance and the thread will be starved!



ReadWriteLock In Java 6

- **Java 6 changed the policy and now read locks have to wait until the write lock has been issued**
- **However, now the readers can be starved if we have a lot of writers**



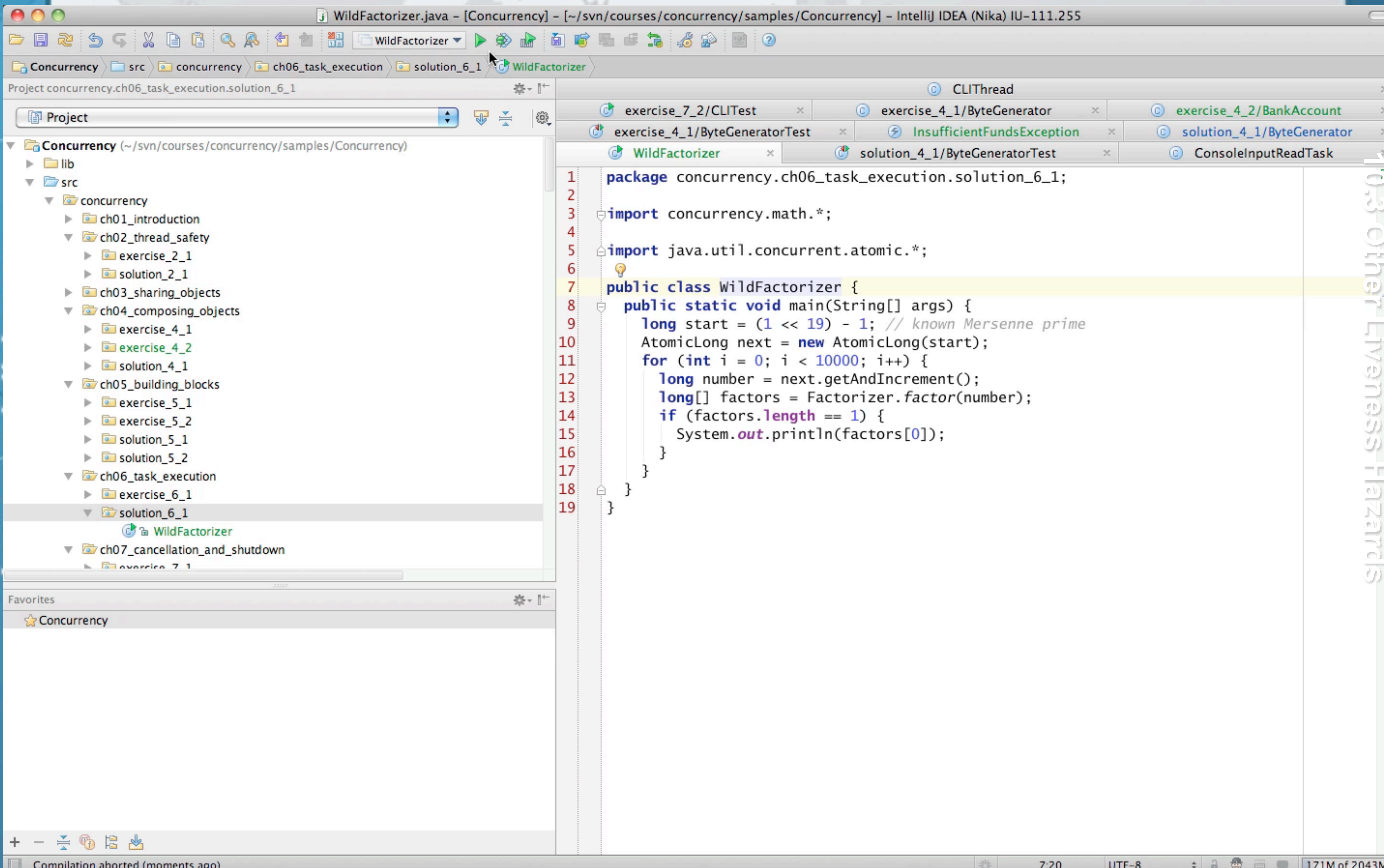
Livelock

- **Thread is running, but still not making progress**
- **Typically forever retrying a failed operation**
  - **Eventually you need to give up**
- **Often occurs in transactional messaging applications, where the messaging infrastructure rolls back a transaction if a message cannot be processed successfully, and puts it back at the head of the queue.**
  - **This form of livelock often comes from overeager error-recovery code that mistakenly treats an unrecoverable error as a recoverable one.**

Real-World Scenario

- **Two polite people meet in a narrow corridor. Each steps to the side to make room for the other. They keep on doing this at the same time, never getting past each other.**
  - **Fortunately people are not that stupid**
    - **But computers are!**
- **Can happen especially in code that tries to recover from a deadlock situation**
  - **Only possible with Java 5 locks, in a controlled fashion**

Livelock In IntelliJ IDEA



10.4: Where To From Here?

The Java Specialists' Newsletter

- **Written for programmers who use Java professionally**
- **Please subscribe for free here:**
  - **http://www.javaspecialists.eu/archive/subscribe.jsp**

Advanced Java Training With ExitCertified

- **Concurrency Specialist Course**
  - More of what we did today, also looking at safety, Fork/Join, etc.
- **Java Specialist Master Course**
  - Threading, Java NIO, reflection, data structures, performance
- **Design Patterns Course**
  - The most useful Gang-of-Four patterns, in Java
- **http://www.javaspecialists.eu**
- **http://www.exitcertified.com/sun-microsystems-training/java-concurrency-JAV-404.html**

10: Exercises

Avoiding Liveness Hazards

Exercise 10.1: Test Java2Demo For Liveness

- **Run the Java2Demo and check for liveness, such as**
  - Deadlock
    - You would notice that part of the program stops responding
  - Livelock
    - Typically your CPU is very high, without any real progress made
- **Please download the workshop exercises from:**
  - http://tinyurl.com/conc-zip
- **Workshop support information is available here:**
  - http://javaspecialists.eu/courses/concurrency/exitcertified.jsp

The End – Thank You!

http://www.javaspecialists.eu