

Testing Concurrent Software

Shmuel Ur

Why is Concurrent Testing Hard?

- ◇ Concurrency introduces **non-determinism**
 - ◇ Multiple executions of the same test may have different interleavings and different results
 - ◇ Very hard to reproduce and debug
- ◇ No useful coverage measures for the interleaving space
- ◇ Typically appear only in specific configurations
 - ◇ Therefore commonly found by users
 - ◇ Require large configurations to test
- ◇ Represent only ~10% of the bugs but an unproportional number are found late or by the customer -> **Very expensive**

The costly effort of testing concurrency at system level is **seemingly** unavoidable

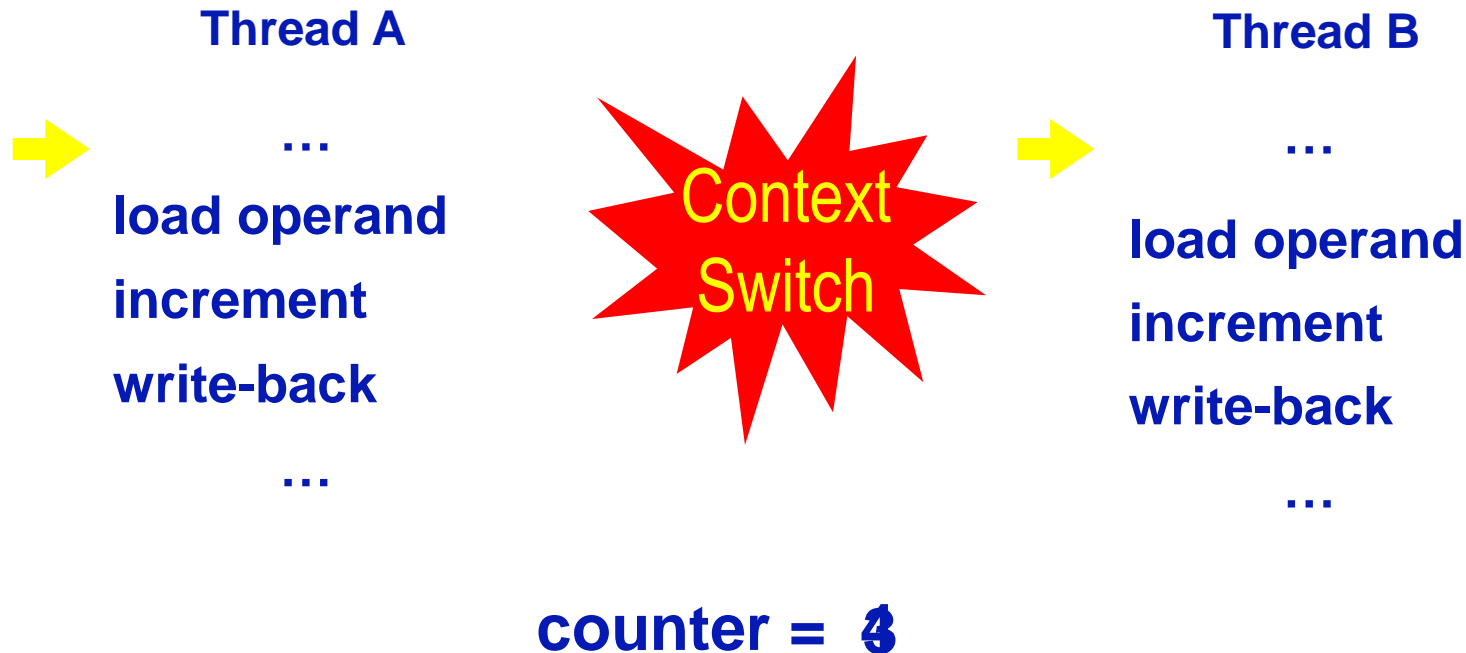
The Classic Java Example – A Counter

Let's consider the function `increase()`, which is a part of a class that acts as a counter

```
public void
increase ()
{
    counter++;
}
```

Although written as a single “`increase`” operation, the “`++`” operator is actually mapped into three JVM instructions [`load operand`, `increment`, `write-back`]

Counter Example – Continued



“Race conditions represent one of your biggest enemies when it comes to programming concurrent applications”

High Performance Java Platform Computing, Christopher &
Thiruvathukal

Sun Java Series, 2000

Riddle I: How many bugs do you see?

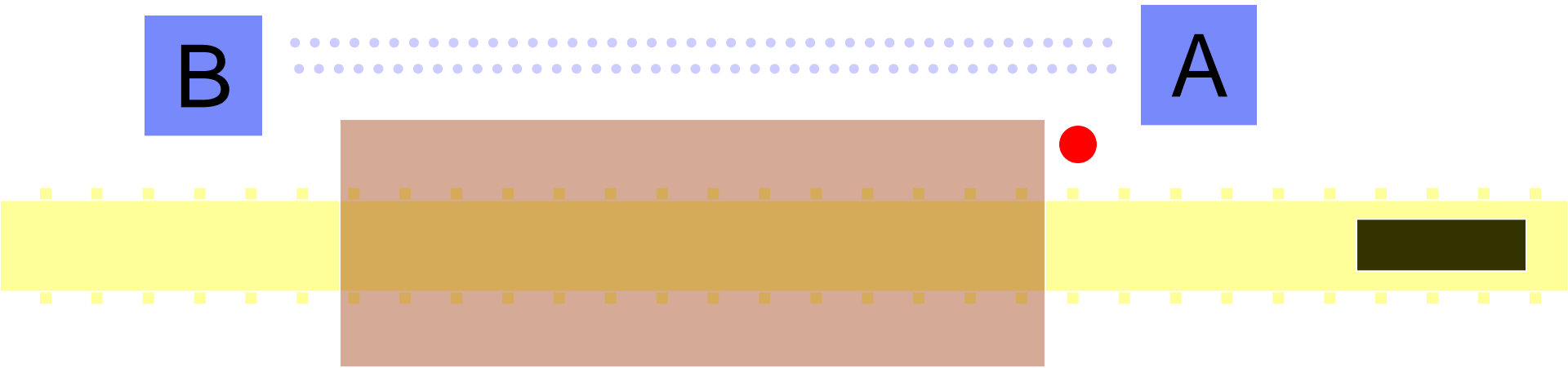
```
public class Helloworld{
    public static void main(String[]
        argv) {
        Hello helloThread = new
            Hello();
        World worldThread = new
            World();
        helloThread.start();
        worldThread.start();

        try{Thread.sleep(1000);}
        catch(Exception exc){};
        System.out.print("\n");
    }
}
```

```
class Hello extends Thread{
    public void run(){
        System.out.print("hello ");
    }
}

class World extends Thread{
    public void run(){
        System.out.print("world");
    }
}
```

Simple Protocol



The train enters the tunnel.

The semaphore automatically turns the light to red.

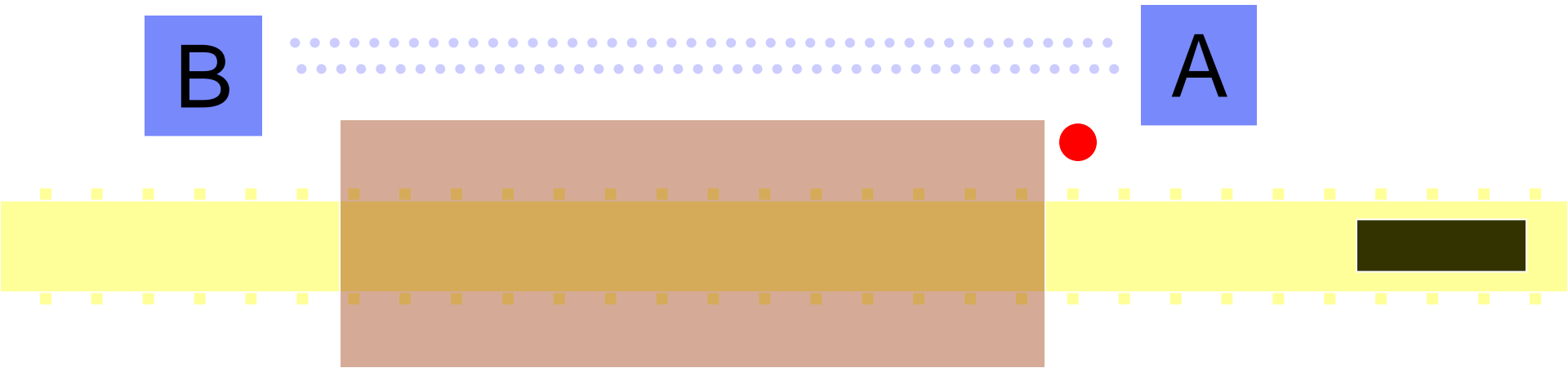
Signalman A sends a message to signalman B that a train is in the tunnel.

The train exits the tunnel.

Signalman B sends a message to signalman A, that the train has exited the tunnel.

Signalman A sets the light to green.

Train Stopping on Red



The first train enters the tunnel.

The semaphore automatically turns the light to red.

Signalman A sends a message to signalman B, that a train is in the tunnel.

The second train approaches the tunnel and stops at the red light.

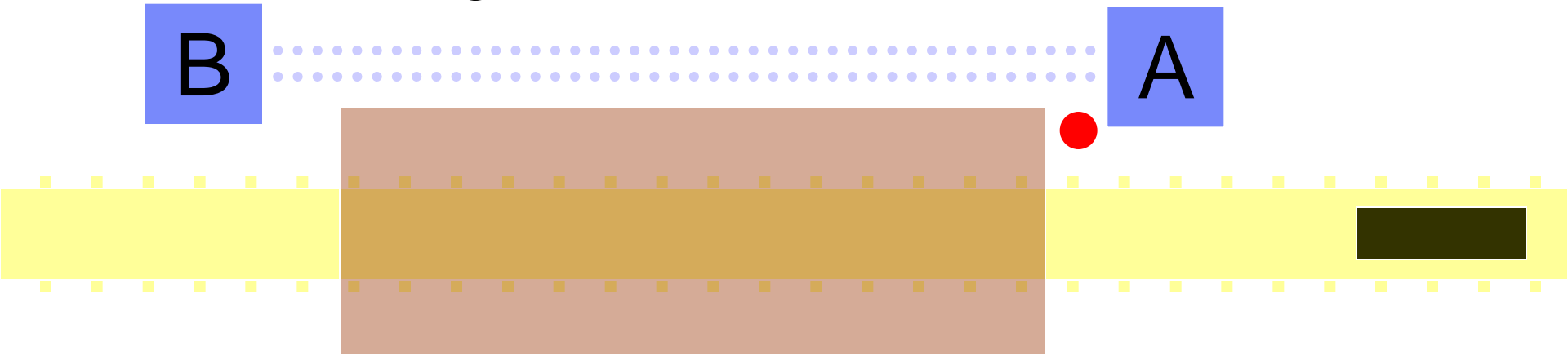
The first train exits the tunnel.

B sends a message to A, that the first train has exited the tunnel.

A sets the light to green.

The second train enters the tunnel.

Malfunctioning Semaphore



The first train enters the tunnel – the semaphore fails to turn the light red!

Signalman A sends a message to signalman B, that a train is in the tunnel.

The second train approaches the tunnel.

Signalman A runs to the track and signals to the train to stop and sets the light to red, manually.

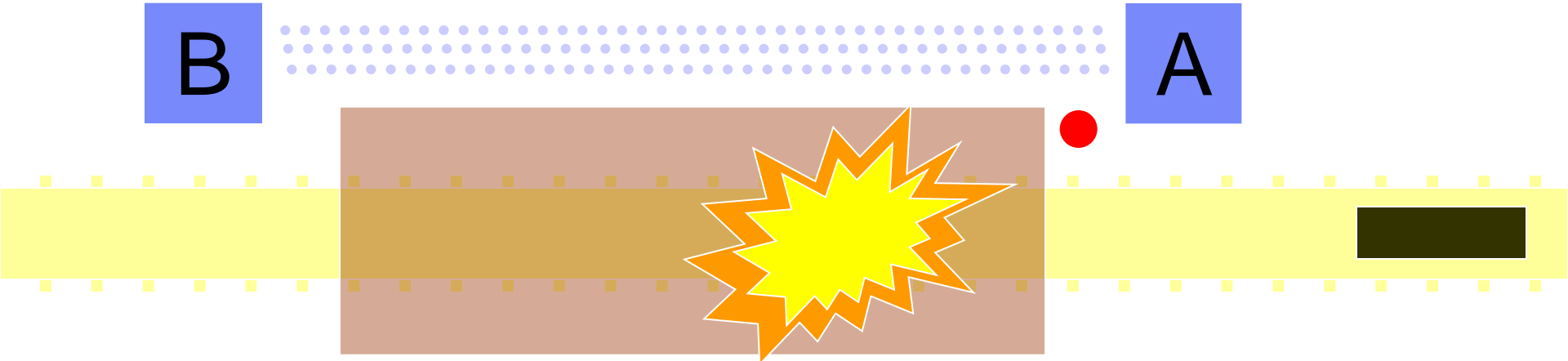
The first train exits the tunnel.

B sends a message to A, that the first train has exited the tunnel.

A sets the signal to green and lets the second train pass.

The second train enters and leaves the tunnel.

Clayton Tunnel Accident - 25 August 1861



The first train enters the tunnel - the semaphore fails to turn the light red!

Signalman A sends a message to signalman B, that a train is in the tunnel.

B sends a message to A, that the train has exited the tunnel.

Signalman A runs to the track to get the light to red and signal other trains to stop, and he sets the light to green.

But a second train arrives and enters the tunnel before he has time to change the light to red. The driver sees something but doesn't have time to stop.

The second train driver decides to back out of the tunnel.

Signalman A sends another message to signalman B, that a train is in the tunnel.

The third train enters the tunnel.

The third train approaches the tunnel and stops.

The second and third trains collide.

The driver of the second train is suspicious that something is wrong and stops in the middle of the tunnel.

Riddle II: Understanding Synchronization Primitives

- ◆ Locks protect the code segment and not the shared data
 - ◆ Obtaining a lock when accessing the shared resources
- ◆ On an error path (e.g., an exception) does the system release the lock?
- ◆ Consider the following class: `class Conflict {`
 - ◆ `Conflict(...){ synchronized(Conflict.class){...}; }`
 - ◆ `void h(...){ synchronized(this){....};}`
 - ◆ `synchronized void g(...){....};`
 - ◆ `void r(...){...}; }`
 - ◆ `synchronized static void f(...){....};`
- ◆ Can `f || g`, `f || h`, `f || r`, `g || h`, `g || r`, `h || r` cause a conflict?

A Bug Published by the ConTest Project Team

A race condition is a possible source for a defect, since the value of the variable at the time of reading depends on the scheduling. However, not all race conditions are defects. For example, the following code swaps two integers. There is a race condition, but no defect, as the swapping occurs regardless of the interleaving.

```
class Change{
    static int x = 4, y = 5;
    //Used to implement a busy wait.
    static int z1 = -1, z2 = -1;
    //Swap the value of x and y concurrently
    public static void main(String args[ ]){
        (new Thread(new ChangeA( ))).start( );
        (new Thread(new ChangeB( ))).start( );
    }
    class ChangeA implements Runnable{
        public void run( ){
            Change.z1 = Change.x;
            while(Change.z2 == -1)
                System.out.println("A is waiting");
            Change.x = Change.z2;}}
    class ChangeB implements Runnable{
        public void run( ){
            Change.z2 = Change.y;
            while(Change.z1 == -1)
                System.out.println("B is waiting");
            Change.y = Change.z1;}}
```

It should be noted that race conditions are execution-dependent: a program might be in a race condition in one execution and not in another. Therefore, tools that detect races at run time (or by analyzing the trace of a given run) are likely to miss some potential data races.

Bug Found by ConTest in Websphere Site Analyzer

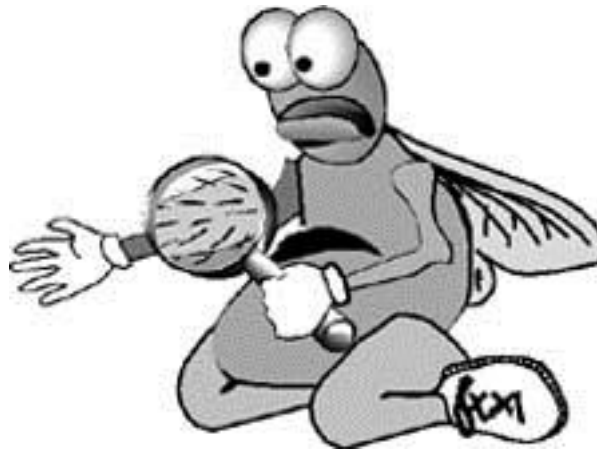
- **Crawler, a ~1000 line Java component, is part of the Websphere Site Analyzer used to perform content analysis of web sites**
- **Bug description:**
 - `if (connection != null) connection.setStopFlag();`
 - Connection is checked to be !null
 - CPU is lost
 - Connection is set to null before CPU is regained
 - If this happens before `connection.setStopFlag();` is executed, an exception is taken
- **This bug was found while we were still testing ConTest**
- **This bug should (also) have been found in unit testing...**

Typical Concurrent Bug Patterns



Non-Atomic

- ◇ An operation is assumed to be atomic but is actually not
 - ◇ Source code operations often seem to the inexperienced programmer to be atomic when they are not
 - ◇ Example: `x++`



Atomicity is Never Ensured

```
static void transfer(Transfer t) {  
    balances[t.fundFrom] -= t.amount;  
    balances[t.fundTo] += t.amount;  
}
```

Expected Behavior:

Money should pass from one account to another

Observed Behavior:

Sometimes the amount taken is not equal to the amount received

Possible bug:

Thread switch in the middle of money transfers

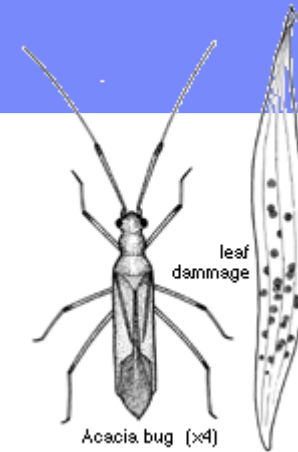
Atomicity is Never Ensured II

- Assume Atomic transfer (can't be implemented locally in Java)
 - `balances[t.fundFrom] -= t.amount;`
 - `balances[t.fundTo] += t.amount;`
- Assume a counting loop that checks if total remains the same
- Does it work now?

Two-Stage-Access

Two stage access:

- ◇ We are given two tables
- ◇ To change a record in the second table, the first table is queried and then the second
- ◇ Each table is protected by a separate lock



lock [First query key1 -> key2]

window -> the tables can be changed here

lock [Second query key2 -> record to be changed]

Wrong/No-Lock

Wrong lock or no lock

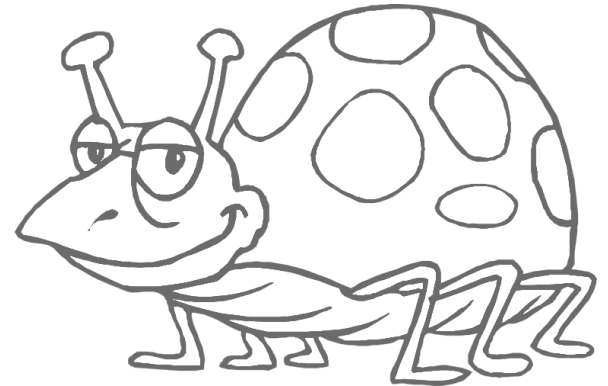
- ◇ Protection of thread one does not apply to thread two
- ◇ There is an access protocol that is not followed due to:
 - ◇ A new team member
 - ◇ An attempt to improve performance

Thread 1

```
Synchronized (o){  
    x++;  
}
```

Thread 2

```
x++;
```



Initialization-Sleep

- ◆ One example is adding `sleep()` statements to ensure that only the correct interleavings occur
 - ◆ Partial, non-consistent results are used by the thread that assumes that initialization is done



Lost-Notify

- ◆ Losing notify: the notify is “lost” because it occurs before the thread executes the wait() primitive
 - ◆ The gap was created because the programmer didn't think the notify would occur before the wait

Thread 1

```
Synchronized (o){  
    o.wait();  
}
```

Thread 2


```
synchronized (o){  
    o.notifyAll();  
}
```

Orphaned-Thread

- ◆ The tale of the orphaned thread
 - ◆ A single master thread drives actions of other threads
 - ◆ Messages are put on the queue by the master thread and processed by the worker's threads
 - ◆ Abnormal termination of the master thread results in the remaining threads being orphaned
 - ◆ The system often blocks

Unintentional-Different-Thread

- ◇ A call to an API (typically a GUI API) is assumed to be in the same thread
- ◇ Is actually in a different thread
- ◇ Can result in concurrency bugs



Nasty to have a hanging single thread program...

Condition-For-Wait

- ◆ Missing condition enclosing the wait
 - ◆ When returning from a wait the programmer forgets to check, or checks incorrectly if the reason for which he waited still holds
 - ◆ In Java 1.5 a wait can decide to terminate. Is your code ready?
- ◆ From [http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Object.html#wait\(long\)](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Object.html#wait(long))
 - ◆ A thread can also wake up without being notified, interrupted, or timing out, a so-called *spurious wakeup*. While this will rarely occur in practice, applications must guard against it by testing for the condition that should have caused the thread to be awakened, and continuing to wait if the condition is not satisfied. In other words, waits should always occur in loops, like this one:
- ◆

```
synchronized (obj) {  
    while (<condition does not hold>)  
        ◆ obj.wait(timeout); ... // Perform action appropriate to condition  
    ◆ }
```

Visibility

- ◆ When a heap variable is updated
 - ◆ It is updated in the registers or “thread local memory”
 - ◆ It reaches the heap on specific language dependant events
 - ◆ For example, a lock release
 - ◆ The programmer assumes that a condition updated by one thread is visible to another thread—when it is not

Some Interesting Observations

- ◆ In practice thread switches are few and far between
 - ◆ The probability that the previous bug will be found is low
 - ◆ Synchronization usually operate as a no-op
 - ◆ Removing all synchronizations usually will not impact testing results!
 - ◆ Not knowing the synchronization primitives exact definition does not impact testing but the program is incorrect
 - ◆ Exception in synchronization: do you still have the lock?
 - ◆ What is the synchronization on?
- ◆ Thread scheduling for small applications is almost deterministic in simple environment
 - ◆ Each environment has its own interleaving
 - ◆ Customer on the first day find bugs in well tested applications!

Noise Makers



With ConTest each Test Goes a Long Way

How Does ConTest Find Bugs?

- ◆ ConTest instruments every concurrent event
 - ◆ Concurrent events are the events whose order determines the result of the program, such as accesses to shared variables, calls to synchronization primitives
- ◆ At every concurrent event, a random-based decision is made whether to cause a context switch by injecting “noise”
 - ◆ E.g., using a sleep statement
- ◆ Philosophy
 - ◆ Modify the program so it is more likely to **exhibit** bugs (without introducing new bugs – no false alarms)
 - ◆ Minimize impact on the testing process (**under-the-hood** technology)
 - ◆ Reuse existing tests

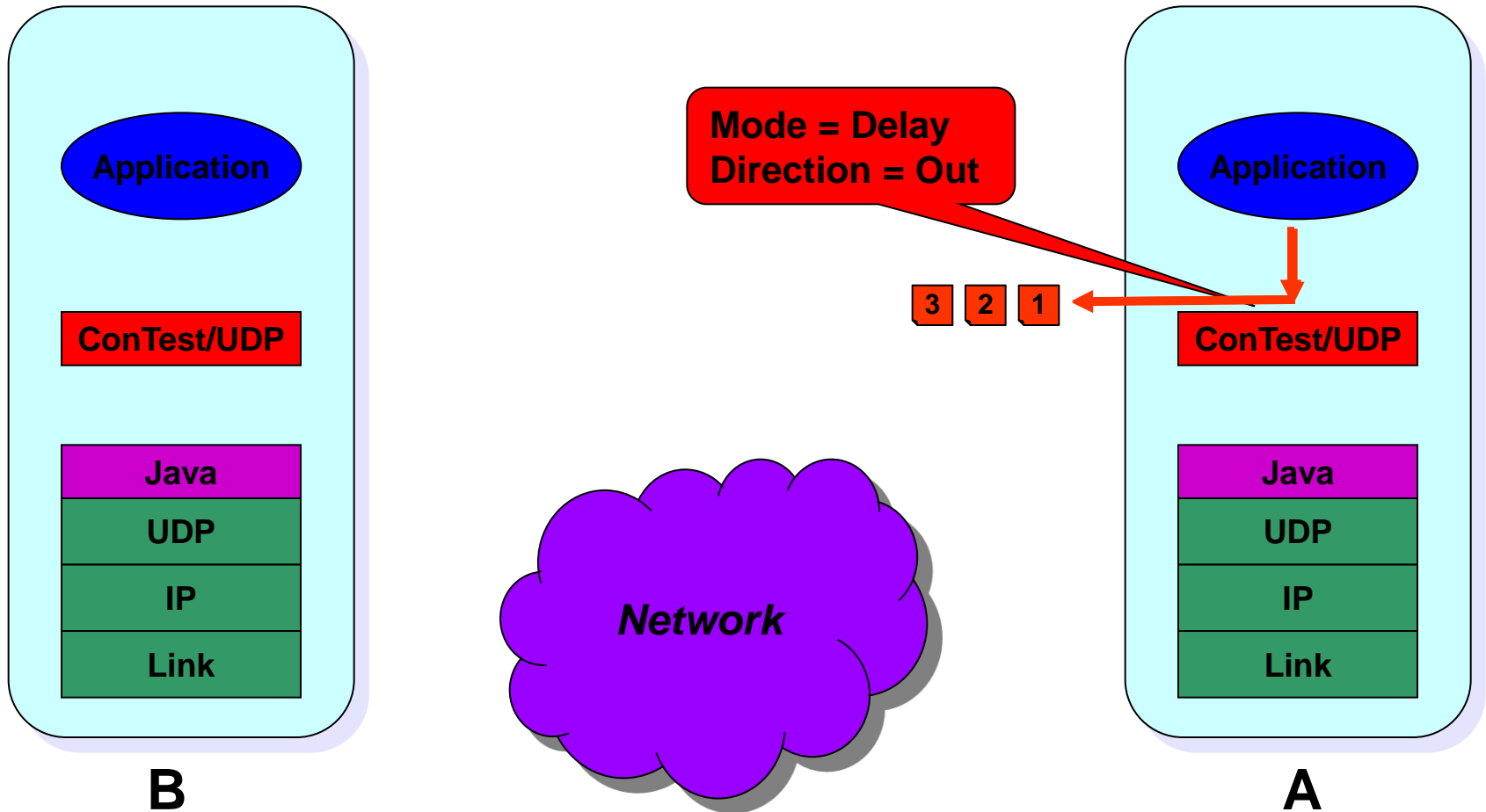
Why ConTest is needed

- ◆ Most unit tests which test concurrency, don't *really* test concurrency.
- ◆ As an exercise, remove the synchronization protocol from the code, and run the test.
 - ◆ From our experience, most probably the test will run just as well...
- ◆ Something is needed in order to make contention actually happen.
- ◆ ConTest's noise injection is that something.
- ◆ Download from: <http://www.alphaworks.ibm.com/tech/contest>

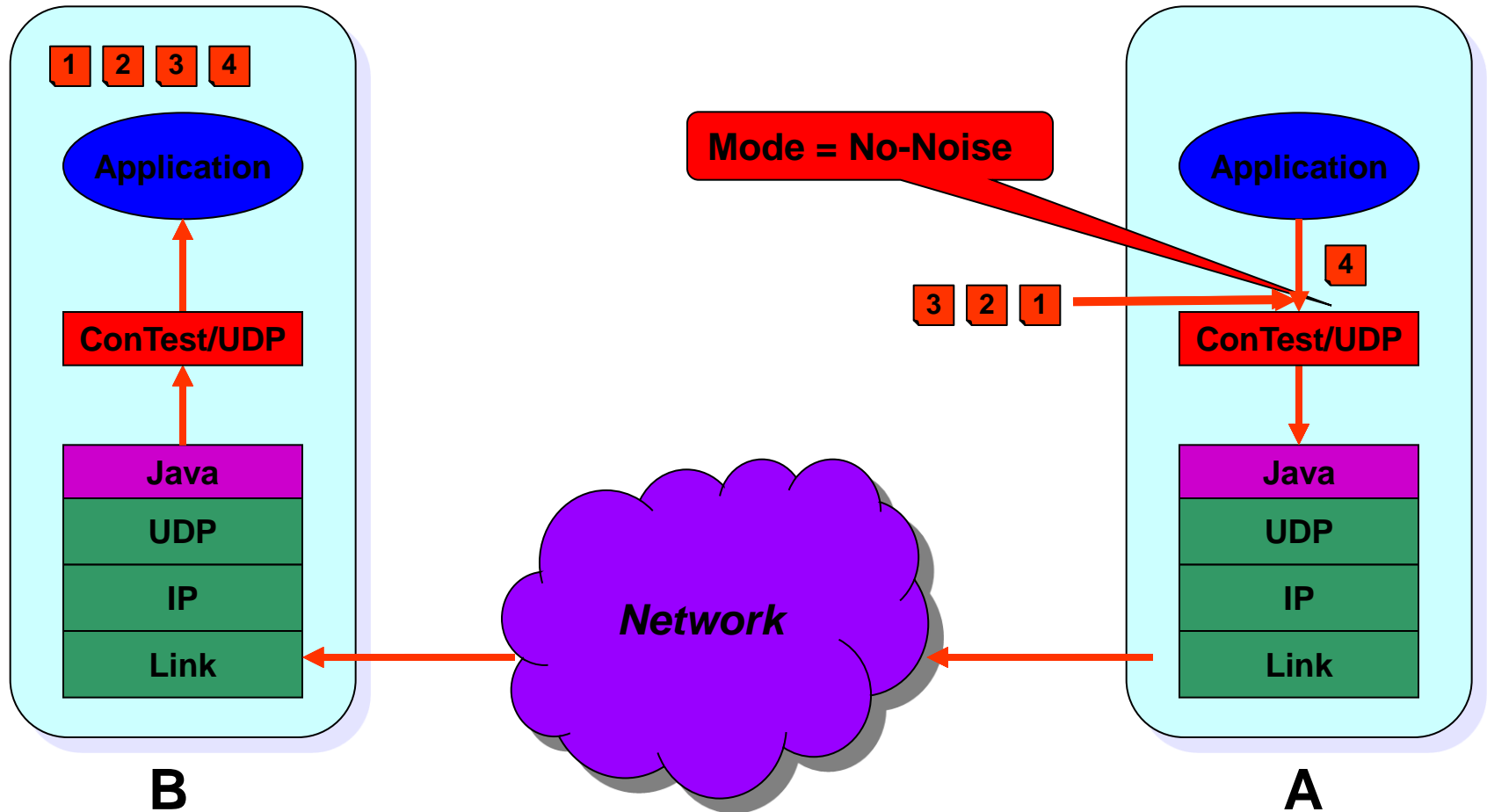
ConTest Noise Heuristics

- ◆ Noise Type
 - ◆ yields, sleeps, synchYields, random
- ◆ Halt One Thread
 - ◆ Stop a thread until no other is willing to run
- ◆ Tamper with timeouts
- ◆ Shared Variables Noise
 - ◆ Concentrate on all or one variable
 - ◆ Collect shared variables dynamically
- ◆ Option to start late
 - ◆ At certain class, method
- ◆ UDP Noise

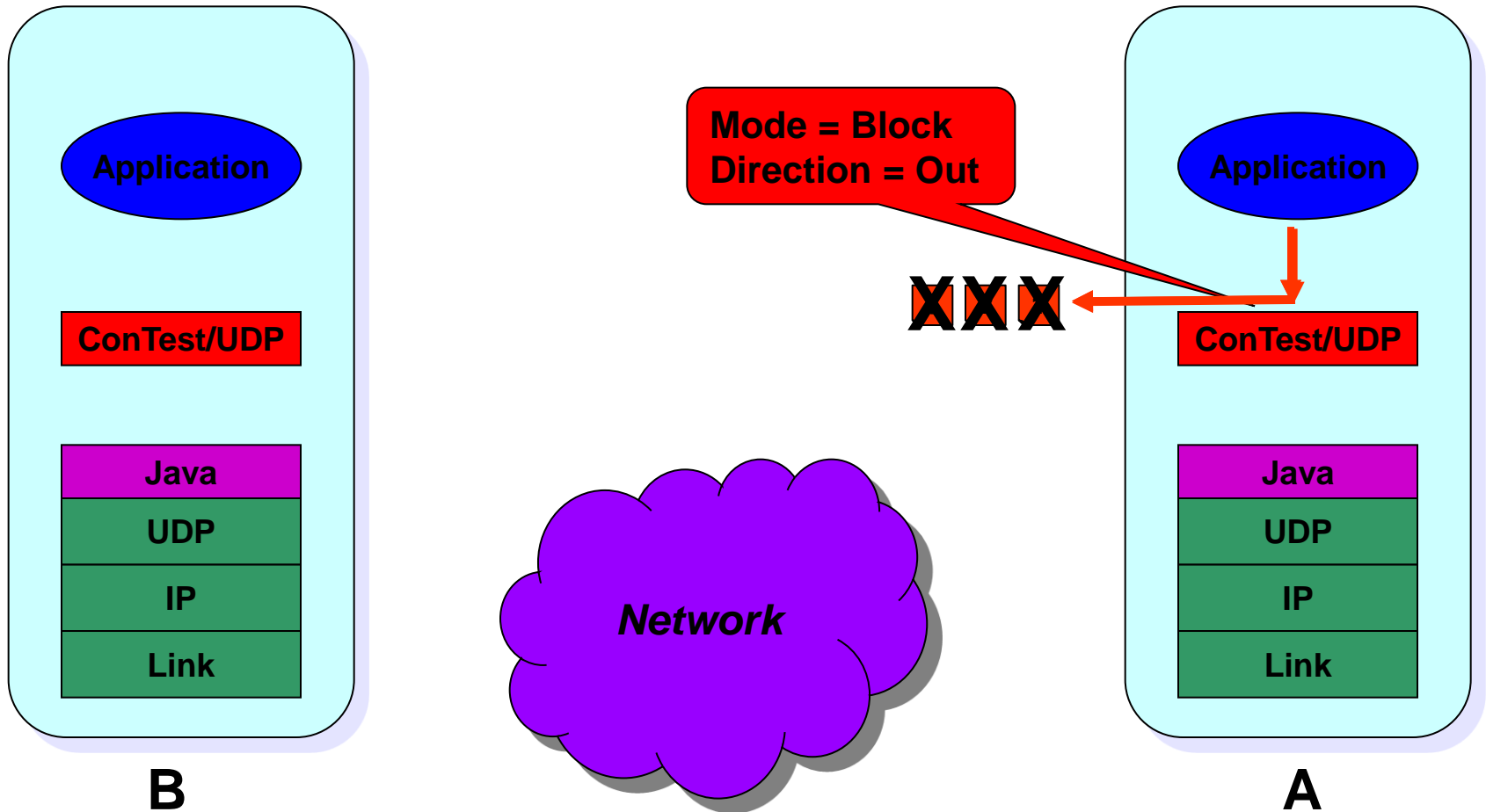
Delaying messages with ConTest/UDP



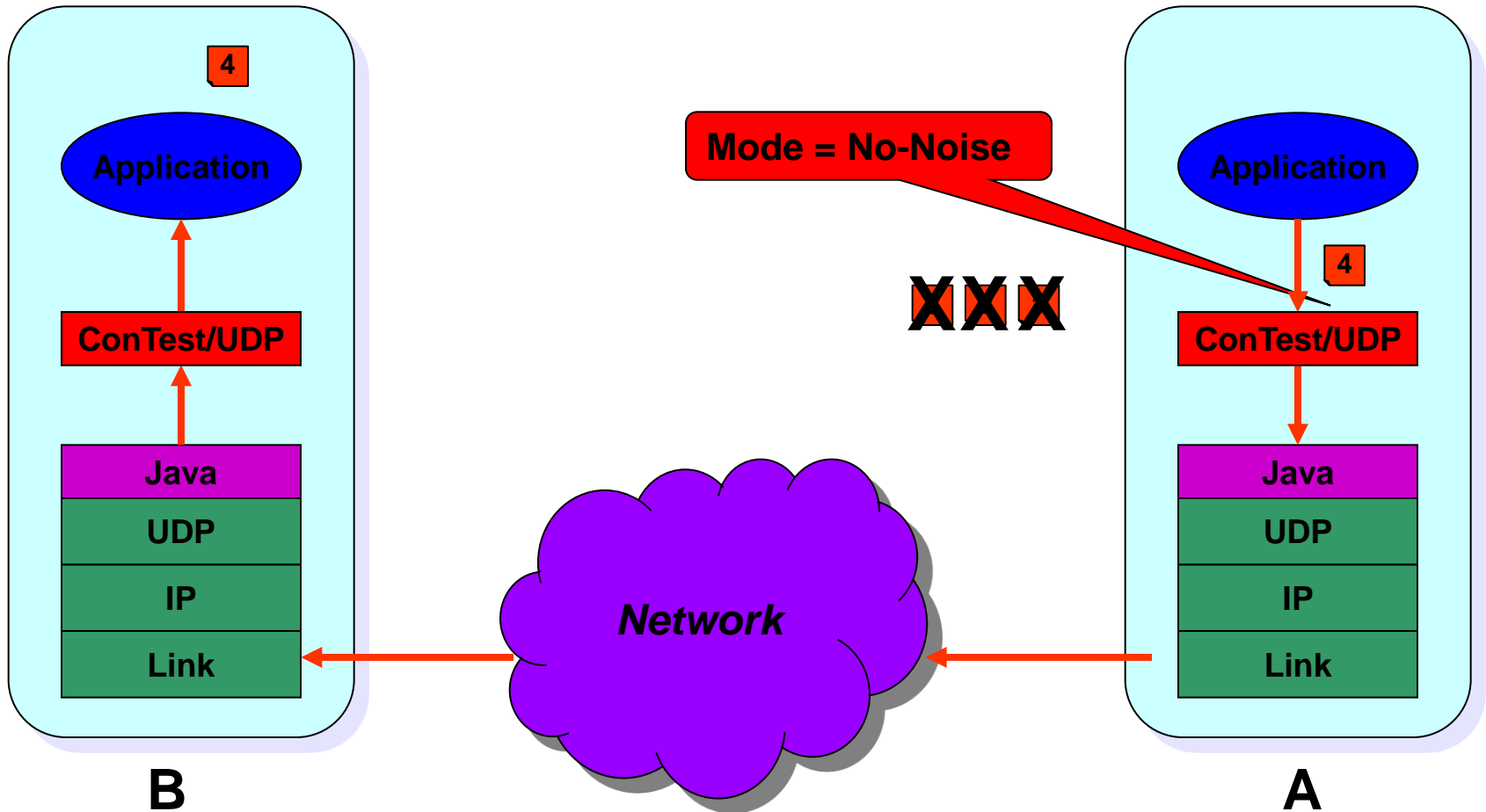
Delaying messages with ConTest/UDP



Losing messages with ConTest/UDP



Losing messages with ConTest/UDP



ConTest Debug Options

- ◆ Deadlocks
 - ◆ Location of each thread
 - ◆ Cycle of waiting on locks
 - ◆ Lock taking trace
- ◆ Lock discipline violation detection
 - ◆ Advanced options: exhibiting and healing
- ◆ orange_box
 - ◆ Remember {last_values_size} of values and locations for each variable
 - ◆ Created for null pointer exception
- ◆ Replay
 - ◆ Record and reuse replay information on noise injection
- ◆ Talk with ConTest (socket/keyboard) while the application runs

Measuring Concurrent Coverage

- ◆ ConTest also measures code coverage
 - ◆ Standard coverage models such as basic block coverage, method coverage
- ◆ How do you know if the tests are any good from concurrency view?
- ◆ Synchronization coverage
 - ◆ Make sure that every synchronization primitive was fully exercised
- ◆ Shared variable coverage
 - ◆ Make sure shared variables were accessed by more than one thread
- ◆ More systematic metrics exist but they do not scale
- ◆ We will later see applications of synchronization coverage

Instrumentations Issues

- ◆ In Java, we had three choices: Source, bytecode, JVM
 - ◆ We chose bytecode
 - ◆ Can be performed offline as a preliminary stage or dynamically at classload (by adding a flag to the run command)
- ◆ In C/C++
 - ◆ Where to instrument: source, object, libraries
 - ◆ Issues to consider: ease of implementation, portability, capabilities
 - ◆ Implement for every concurrency library

Other Porting Issues

- ◇ Understanding the primitives
 - ◇ Vital not to introduce new errors
 - ◇ Example: `sleep()` in Java
- ◇ Lock-based vs. interrupt-based
 - ◇ So far, worked on lock-based
- ◇ Type of appropriate noise

Unit test goals and outline

- ◆ Do a thorough test of **each** synchronization protocol.
- ◆ Use extraction and mock objects to **isolate** the protocols.
- ◆ Use **interleaving review** to guide writing tests.
- ◆ Test all corners, cover the **scenario space**.
 - ◆ 100% basic block coverage
 - ◆ 100% synchronization coverage
- ◆ Automate, and run tests all night.
- ◆ Verify lock discipline.

Unit Testing Concurrent Code with ConTest

1. Remove non-relevant code (optional where applicable)
2. Create a number of tests
3. Instrument the code with ConTest
4. Run each test multiple time and measure synchronization coverage
 1. If a bug is found, use ConTest to debug, fix, go to 3
 2. If deadlock violation found, fix, go to 3
5. If coverage not sufficient, analyze
 1. Need to rerun tests, go to 4
 2. Need to write new tests, go to 2
 3. Genuinely uncoverable, adjust coverage goal, go to 5
6. Done

System Test Goals and Outline

- ◆ Program is tested in a “natural” setting (or an approximation to natural setting, e.g. simulation).
- ◆ Synchronization protocols are tested indirectly.
- ◆ Test as much as possible, according to available resources, with no claim to completeness.
- ◆ Code coverage is used to find big “holes”: areas of the code that received little attention in the test.

System Test Goals and Outline (Continued)

- ◆ The big question: did the test stress all the synchronization protocols?
 - ◆ Not obvious, since the tester usually does not know the protocols.
 - ◆ Synchronization coverage helps finding the answer.

Function and System Test with ConTest

- ◆ Instrument only the classes containing the protocols.
 - ◆ The more classes are instrumented, the more noise ConTest makes.
 - ◆ Noise costs runtime: more noise, less tests.
 - ◆ Instrumenting only concurrent code will focus the noise where it is most useful.
- ◆ After running the tests:
 1. Verify that no bug was found
 2. Check lock discipline
 3. Check coverage