第十章 基于程序动态分析的服务缺陷检测

王旭
北京航空航天大学

程序的动态行为?

```
#include <stdio.h>main(t,_,a)char *a;{return!0<t?t<3?main(-79,-
13,a+main(-87,1-_,main(-86,0,a+1)+a)):1,t<_?main(t+1,_,a):3,main(-94,-
27+t,a)&&t==2?_<13?main(2,_+1,"%s %d %d¥n"):9:16:t<0?t<-
72?main(_,t,"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l,+,/n
{n+,/+#n+,/#¥;#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';dq#'l
¥q#'+d'K#!/+k#;q#'r}eKK#}w'r}eKK{nl]'/#;#q#n')()#}w'){){nl]'/+#n';d}rw' i;#
¥){nl]!/n{n#'; r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#n'wk nw'
¥iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c ¥;;{nl'-
{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;#'rdq#w! nr'/ ') }+}{rl#'{n' ')#
¥}'+}##(!!/"):t<-50?_==*a?putchar(31[a]):main(-
65,_,a+1):main((*a=='/')+t,_,a+1)  :0<t?main(2,2,"%s"):*a=='/'||main(0,main(-
61,*a,"!ek;dc i@bK'(q)-[w]*%n+r3#l,{}:¥nuwloca-O;m
.vpbks,fxntdCeghiry"),a+1);}
```

Thomas Ball, The Concept of Dynamic Analysis, FSE'99

程序动态分析

- 动态分析：运行期的程序状态分析
  - 需要实际运行代码
  - Workloads或Testcases非常重要
  - 一般只能覆盖程序的部分代码或者行为
  - 有助于理解程序的动态行为
- 功能
  - 状态追踪(Tracing)，记录日志(Logging)
  - 发现程序的运行状态，包括时间戳、运行时间、事件、线程、消息流、资源消耗等
  - 发现潜在的程序缺陷：并发缺陷、性能瓶颈等

测试与动态分析

- 工业界
  - 单元测试
  - 集成测试
  - 常规workload下直观观察程序行为
- 自动化的动态分析
  - 收集运行时状态数据，根据数据进行进一步的逻辑分析和推理，发现更深层次的缺陷
  - 存在问题
    - 受限于workload，只能覆盖一部分程序行为，不可能发现所有的缺陷(false negative)
    - 可基于自动化的Trigger，确认发现的缺陷是否是真的(false positive)

动态分析

- **切面和追踪分析(Profiling/Tracing Analysis)**
  - **针对具体研究的问题，选取合适的workload，监控相关实体的状态，获取日志**
    - 关注的进程、线程、方法、语句、变量和锁对象等
    - 关注的内存访问、磁盘访问和API调用等
    - 关注资源消耗，比如内存、CPU、线程池等
    - 关注执行时间和访问次数等
- **因果分析(Causality Analysis)**
  - **跨线程、跨节点的事件之间的依赖关系**
    - Profiling进程、线程、事件以及消息传递，构建逻辑因果关系
    - 没有依赖的事件可并行，数据事件间存在data race

切面分析

- 源代码
  - 相对简单， 直接修改源代码
  - 但是，如果关注的切面很多，人工成本较高
- 中间代码，比如Java字节码
  - 借助解释器提供的Instrumentation接口
  - 边解释执行，边插装Profiling代码片段
- 二进制代码
  - 运行时机制，替换链接接口，或者插装重编译Profiling代码片段
  - 与操作系统和CPU相关

源代码切面分析

- 直接修改源代码
  - 定位需要修改的切面
  - 嵌入Instrumentation代码片段
  - 获取上下文变量值

```
For any Method such as void *(int, double)

void insertAfterFunEenter(void *this, void *callsite)
{
        int a=$0, double b=$1 then log /* called on function
entry */
}

void insertBeforeFunExit(void *this, void *callsite)
{
        int a=$0, double b=$1 then log /* called just
before returning from function */
}
```

实现工具

- **AspectJ: Java语言的AOP框架**
  - 方法扩展，添加方法

```
aspect VisitAspect {
  void Point.acceptVisitor(Visitor v) {
    v.visit(this);
  }
}
```

  - 定义切面

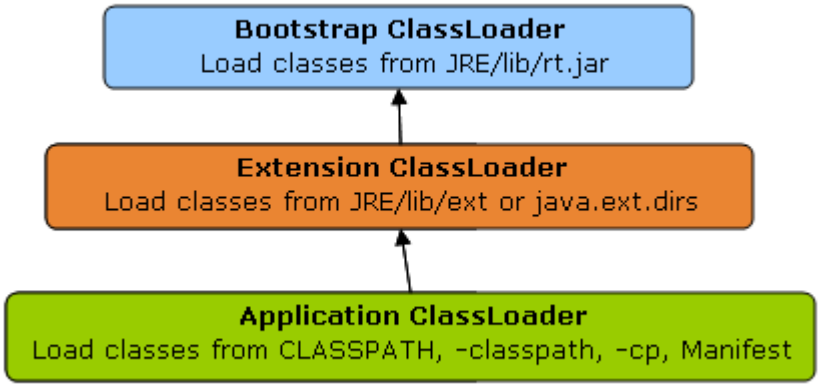
```
pointcut set() : execution(* set*(..) ) && this(Point);
```

  - After切面set()执行Display.update()

```
after () : set() {
  Display.update();
}
```

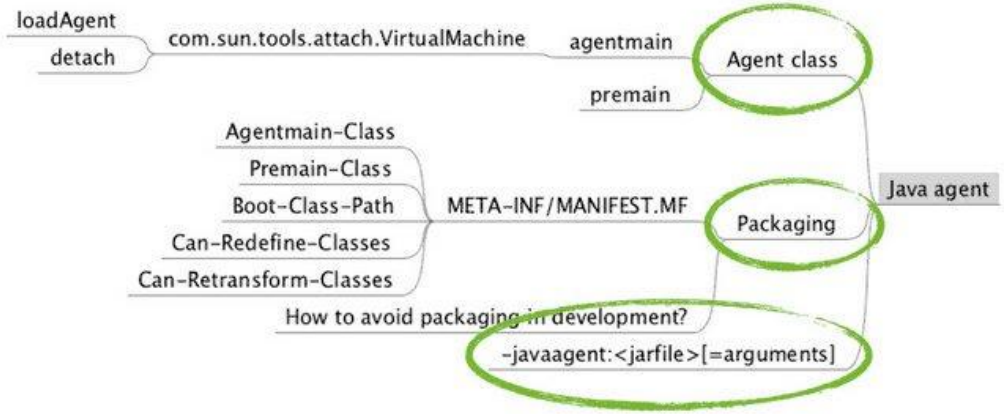
字节码切面分析

- 与源代码类似，在JVM解释执行时起作用
  - 源代码本身不发生变化
  - Class文件Load到JVM后进行插装和修改
- JVM的ClassLoader过程
  - 系统Class load
  - 应用Class load
- 字节码切面分析
  - 动态加载Class
  - 同时修改Class



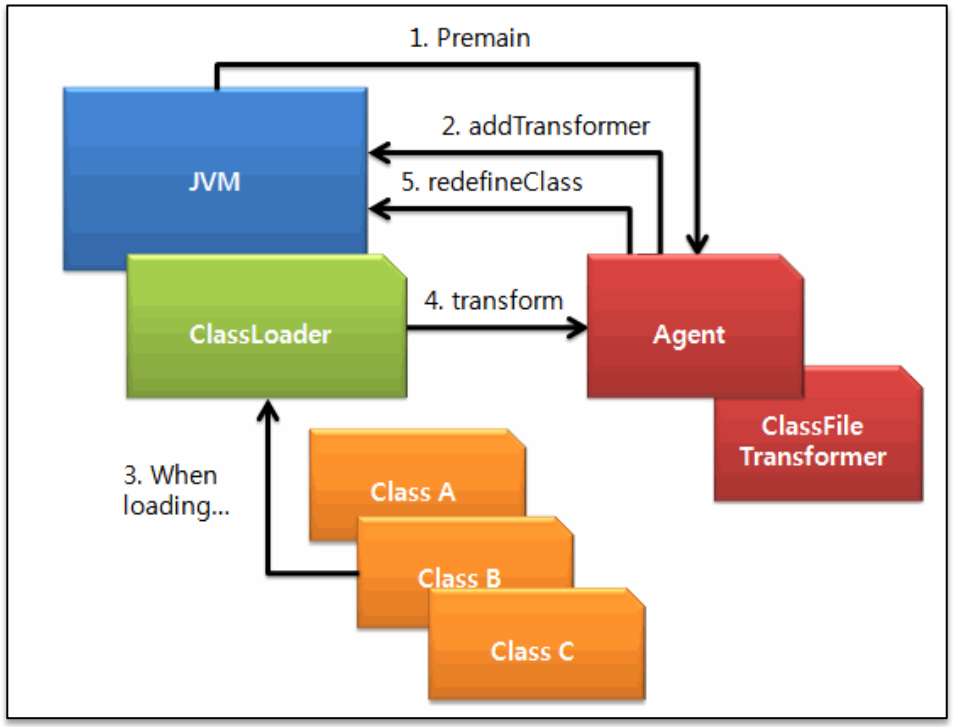
实现机制(1)

- Java agent机制
  - Java参数中添加agent类的Jar包
  - Agent类中实现premain方法
  - 每当JVM load一个新的Class时，都会调用premain方法



实现机制(2)

- **Class Transformer**
  - 继承实现Transformer接口，定义自己的Class Transformer类



实现机制(3)

- **Class操作：Javassist库**
  - **Java Bytecode的操作库**
  - **构建类对象和方法对象**

```
ClassPool pool = ClassPool.getDefault();
CtClass cc = pool.get("Point");
CtMethod m = cc.getDeclaredMethod("move");
m.insertBefore("{ System.out.println($1); System.out.println($2); }");
cc.writeFile();
```

  - **常见的操作**
    - 方法的insertBefore,insertAfter
    - 修改方法体
    - 增加新的类方法和属性

实现机制(4)

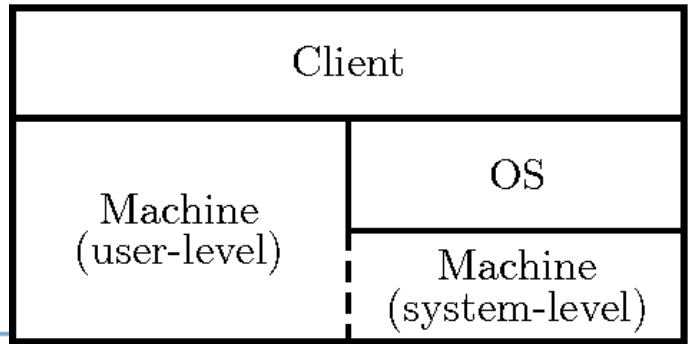
- **Class操作：Javassist库**
  - **Bytecode level Operation，是对Class文件字节级的操作，一般用于细粒度的Tracing**
    - Heap Read/Write
  - **增加int常量，并作为返回值**

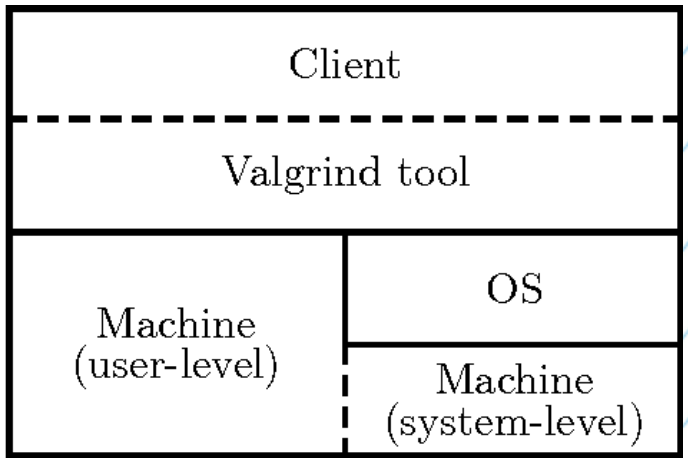
```
ConstPool cp = ...;    // constant pool table
Bytecode b = new Bytecode(cp, 1, 0);
b.addIconst(3);
b.addReturn(CtClass.intType);
CodeAttribute ca = b.toCodeAttribute();
```

  - **具体参考JVM Specification文档**

二进制切面分析

- 与操作系统有关，一般X86-Win/Linux
- 分析工具嫁接到待分析的二进制程序
  - 反汇编，自定义IR
  - 在IR中插入Instrumentation代码片段
  - JIT编译，边插装边执行





实现机制(1)

- LD_PRELOAD环境变量
  - 在调用库文件之间，预先加载自定义的C/C++库文件
  - 函数同名情况下优先调用自定义的函数
  - 可以替换和重写已有的库函数，从而插入Instrumentation代码片段，在执行时获取程序信息
- 例子：获取程序malloc信息
  - LD_PRELOAD=/path/to/my/malloc.so /bin/ls
- 简单方便，适用于动态库的情形

实现机制(2)

- Valgrind开源工具
  - 支持x86/Linux、AMD64/Linux、ARM64/Linux、MIPS32/Linux, MIPS64/Linux等
  - IR Instrumentation和JIT编译
- 自带常见tools
  - 比如Memchecker(内存越界和泄露)，Massif(heap profiler)等等
- 用户可以开发自定义的profiling工具

因果分析(Causality Analysis)

- 核心是获取事件间的先后执行顺序
  - Lamport Happened-before关系的分布式推广
- Tracing的信息包括
  - 进程、线程的创建和执行
  - 事件的创建和处理
  - 消息的发送和接收
- 构建因果依赖图
  - 点：事件
  - 边：同一进程和线程按照时间戳排序，不同线程、进程、事件和消息按照创建（发送）和开始（接收）事件排序

一个工具

- **DCatch: java语言**
  - **基于Javassist实现**
  - **支持大规模系统的因果分析**

App	Inter-Node			Intra-Node	
	Sync. RPC	Async. Socket	Custom Protocol	Sync. Threads	Async. Events
Cassandra	-	✓	-	✓	✓
HBase	✓	-	✓	✓	✓
MapReduce	✓	-	✓	✓	✓
ZooKeeper	-	✓	-	✓	✓

基于程序动态分析的服务缺陷检测

- 常见动态行为的缺陷检测
  - 内存缺陷
  - Heap溢出缺陷（Out-of-Memory）
  - API相关的性能缺陷
- 复杂缺陷检测
  - 并发缺陷(Concurrency Bug)
  - 失效时序缺陷(Timing-of-Fault Bug)
  - 低效循环/同步性能缺陷及其传播缺陷 (Loop/Synchronization Performance Bugs and their Cascading Bug)

内存缺陷检测

- 常见内存缺陷(Valgrind Memcheck)
  - 内存溢出
  - 内存泄露

```
#include <stdlib.h>

void f(void)
{
   int* x = malloc(10 * si
   x[10] = 0;          // problem 1: heap block overrun
}                      // problem 2: memory leak -- x not freed

int main(void)
{
   f();
   return 0;
}
```

```
==19182== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==19182==    at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
==19182==    by 0x8048385: f (a.c:5)
==19182==    by 0x80483AB: main (a.c:11)
```

```
==19182== Invalid write of size 4
==19182==    at 0x804838F: f (example.c:6)
==19182==    by 0x80483AB: main (example.c:11)
==19182==  Address 0x1BA45050 is 0 bytes after a block of size 40 alloc'd
==19182==    at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
==19182==    by 0x8048385: f (example.c:5)
==19182==    by 0x80483AB: main (example.c:11)
```

Heap溢出缺陷检测

- Heap Profiling
  - 监控Heap操作和内存大小
  - 寻找影响大的操作
- 字节码分析：Javassist Bytecode 操作
  - GetField
  - PutField
- 二进制分析
  - 统计内存操作的字节数

API相关的性能缺陷

- 系统或第三方API调用
  - 文件读写访问: file-related
  - 网络传输API: socket-related/http-related
  - 大数据结构：array-related/hashmap-related
- 字节码分析：Javassist操作
  - Method invocation
  - Parameters capture

并发缺陷(1)

- 两个操作间没有因果依赖的先后顺序，执行顺序可能发生改变
  - A与B并发：以为A在B之前发生，但实际可能B在A之前发生，从而导致错误
  - 单机：多线程

```
1   Thread 1::
2   if (thd->proc_info) {
3     ...
4     fputs(thd->proc_info, ...);
5     ...
6   }
7
8   Thread 2::
9   thd->proc_info = NULL;
```

```
1   Thread 1::
2   void init() {
3      ...
4      mThread = PR_CreateThread(mMain, ...);
5      ...
6   }
7
8   Thread 2::
9   void mMain(...) {
10      ...
11      mState = mThread->State;
12      ...
13  }
```

并发缺陷(2)

- 两个操作间没有因果依赖的先后顺序，执行顺序可能发生改变
  - A与B并发：以为A在B之前发生，但实际可能B在A之前发生，从而导致错误
  - 分布式系统：跨节点、消息传递

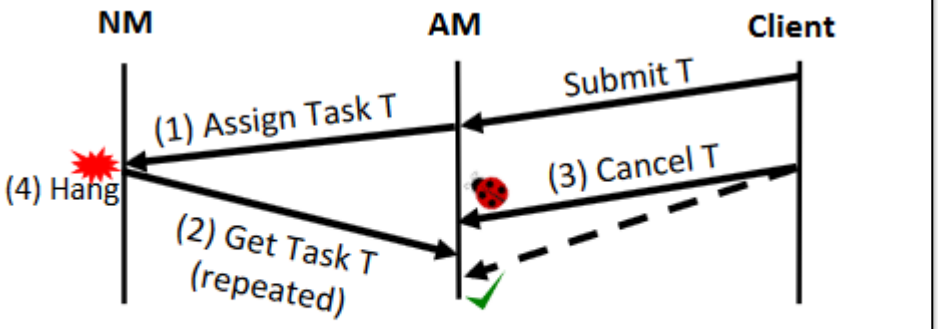


Figure 1. A Hadoop DCbug: Hang (buggy) if #3 happens before #2, or no failure (√) if the otherwise.

并发缺陷(3)

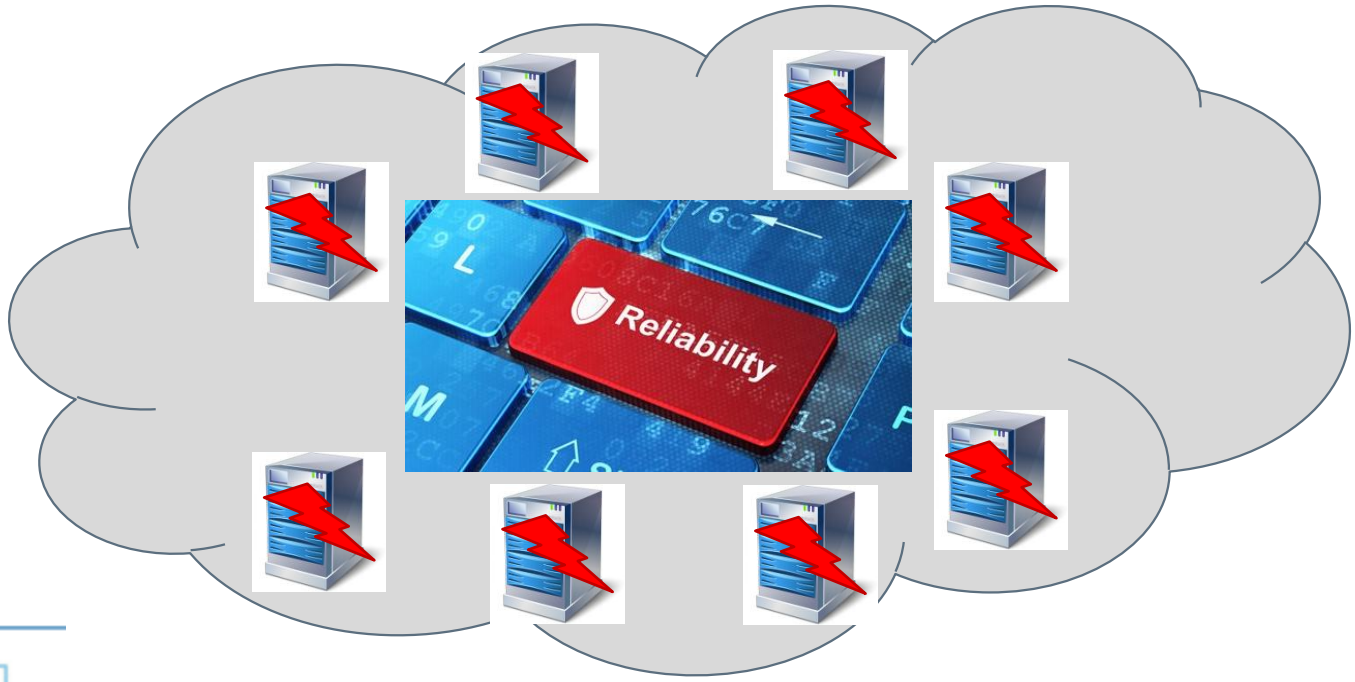
- 因果分析
  - 寻找不存在因果的操作对A和B
- Data Race
  - 操作A和B是对同一个数据对象的读或写
  - A与B至少有一个是写操作
  - A与B不存在因果（即顺序，可并行）
  - 数据对象的种类
    - Heap对象
    - 本地文件
    - 分布式文件
    - 分布式锁对象

并发缺陷(4)

- 数据对象Profiling
  - Heap对象
    - Javassist bytecode分析
  - 本地文件
    - Javassist系统API分析
  - 分布式文件
    - Javassist第三方API分析
  - 分布式锁对象
    - Javassist第三方API分析

失效时序缺陷(1)

- 分布式系统的组件失效(Faults)
  - Node crash
  - Message drop



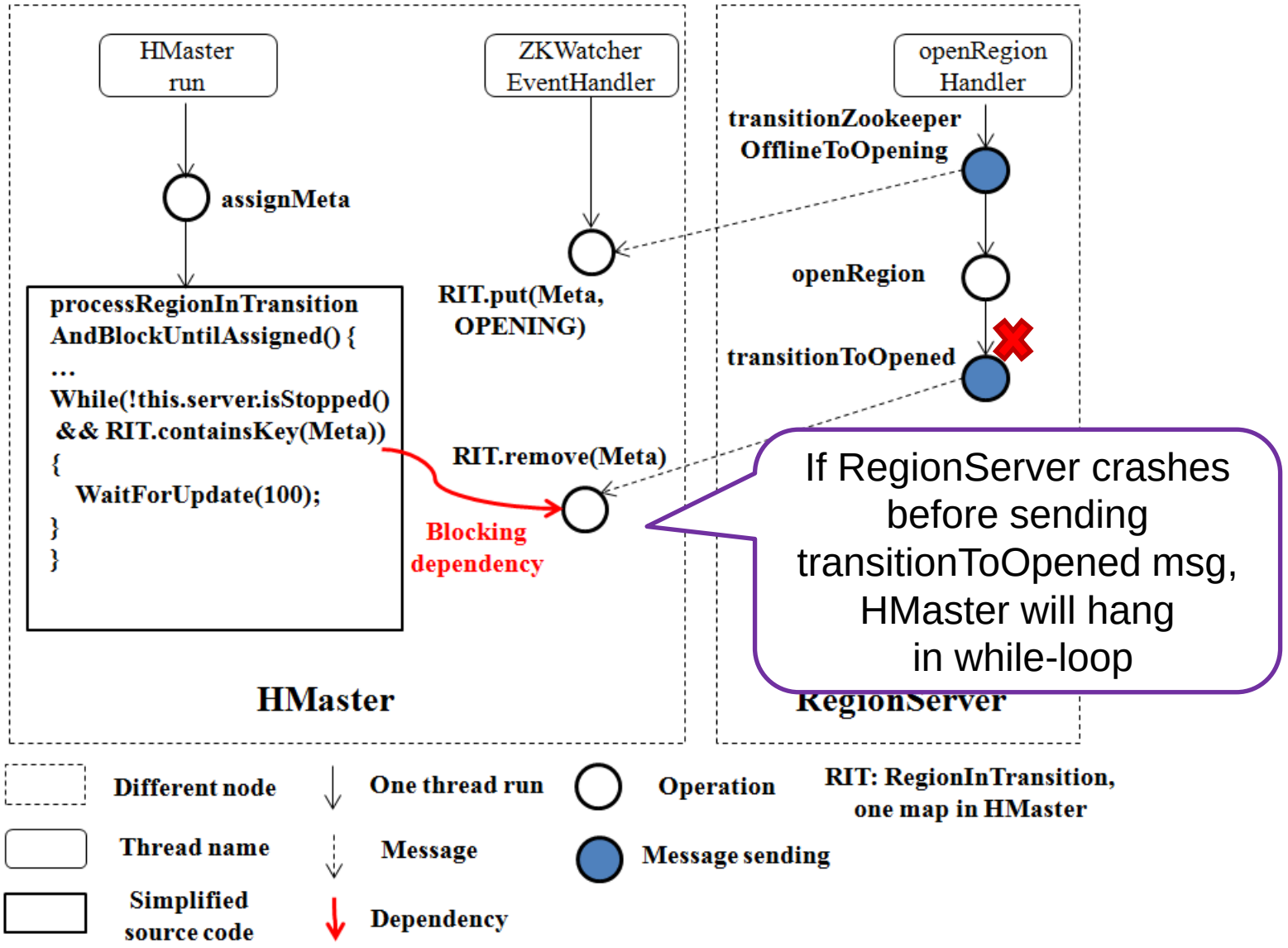
失效时序缺陷(2)

- **Time-of-Faults(TOF) bugs**
  - **Trigger: a component failure (fault) at a special time**
  - **Error: unexpected system state left by the faulty node**
  - **Failure: system hang/failure due to improper handling of the unexpected state**
- **失效时序缺陷：特殊时间发生的失效导致的系统错误**

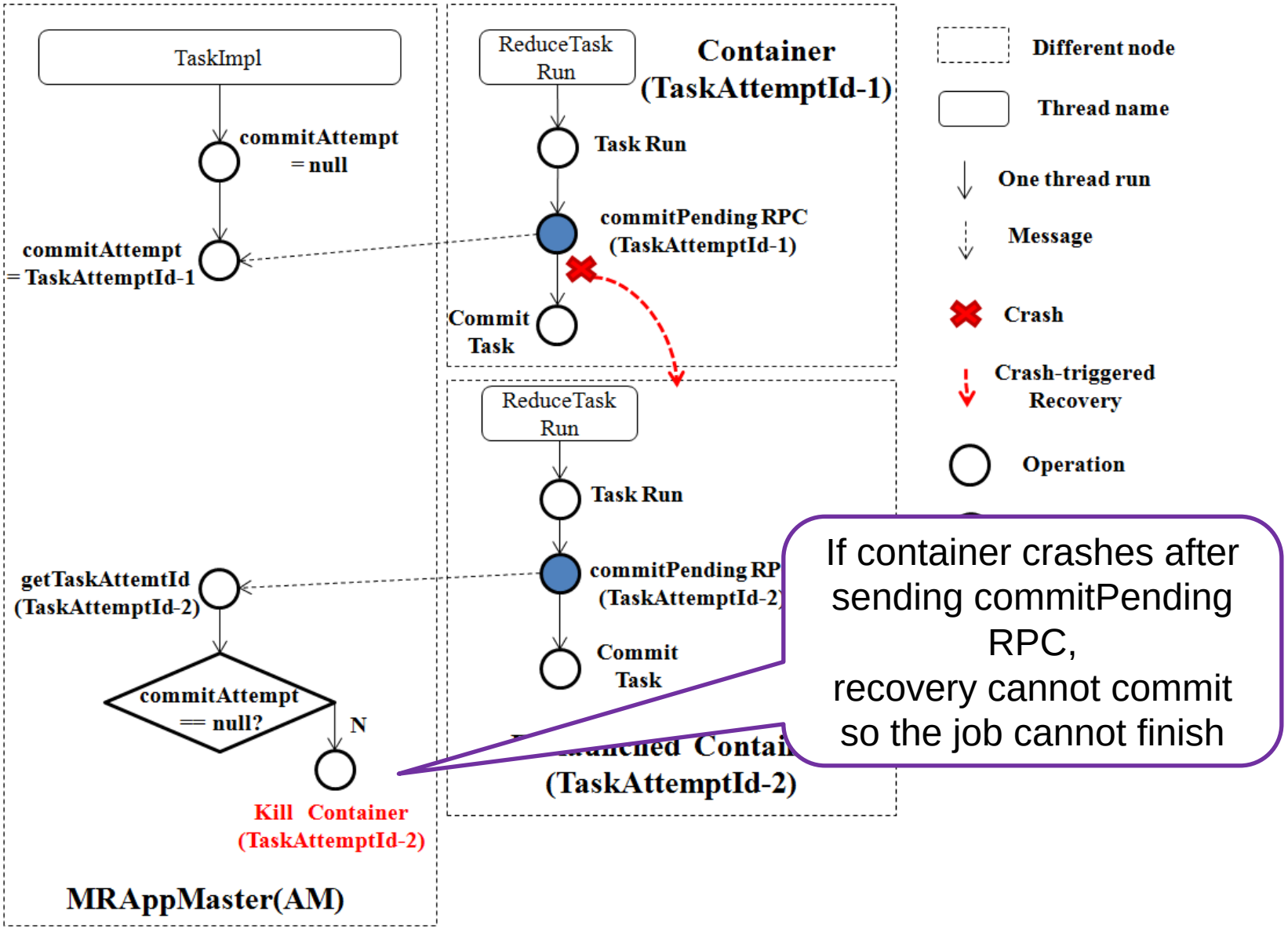
失效时序缺陷(3)

- 分布式系统中很常见
  - 商用的低成本计算机和网络设备
  - 数据中心规模大
  - Google Report: servers will crash at least twice yearly (2-4% failure rate)
- 检测的难度
  - 常规的测试很难发现
  - 动态行为不确定
  - 特殊的时序

失效时序缺陷(4): 例子HB10090

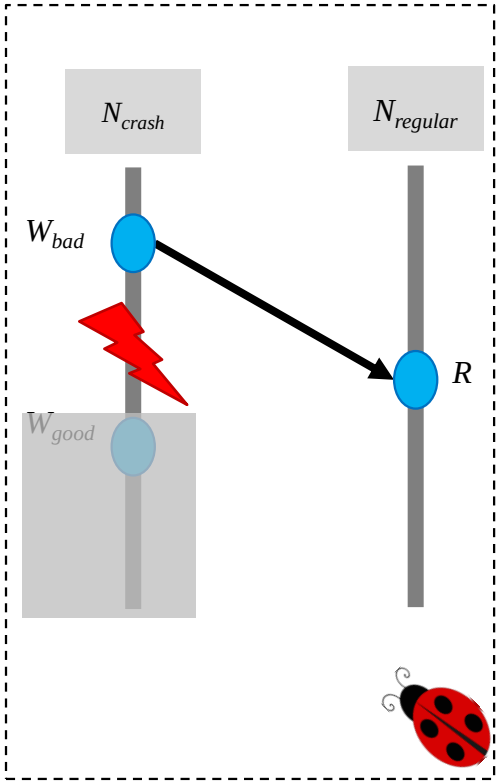


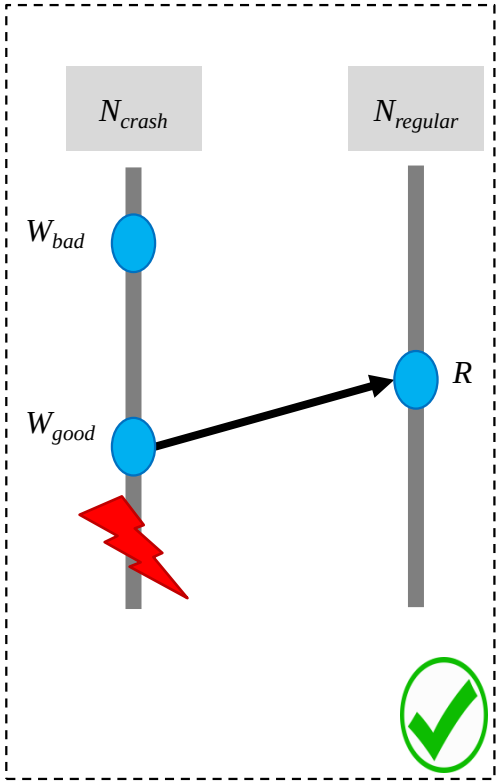
失效时序缺陷(5): 例子MR3858



失效时序缺陷(6)

Crash-Regular TOF缺陷

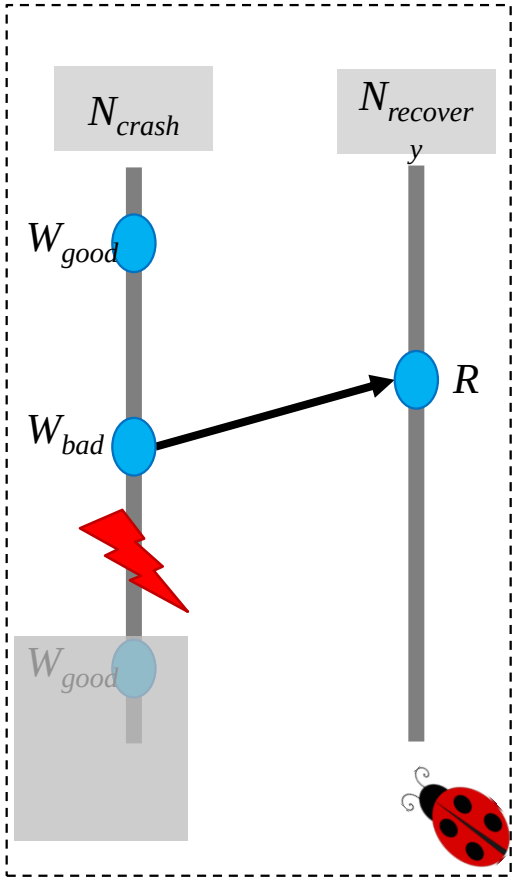


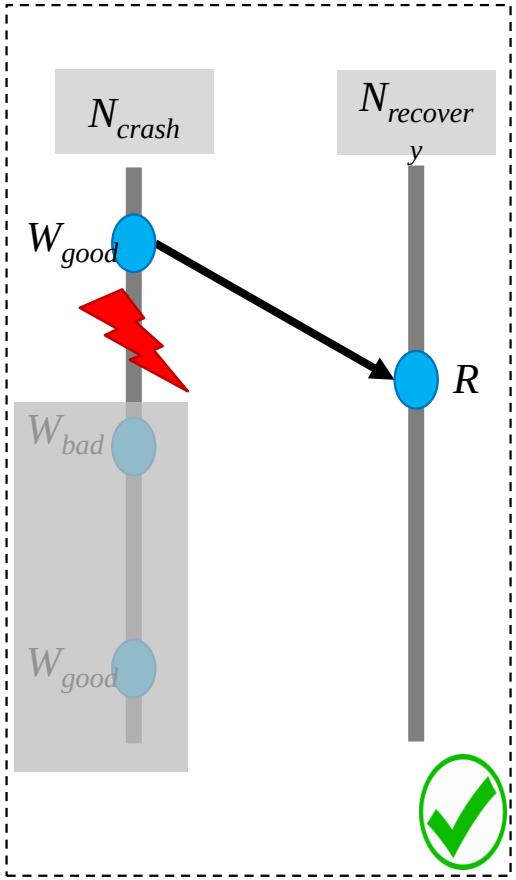


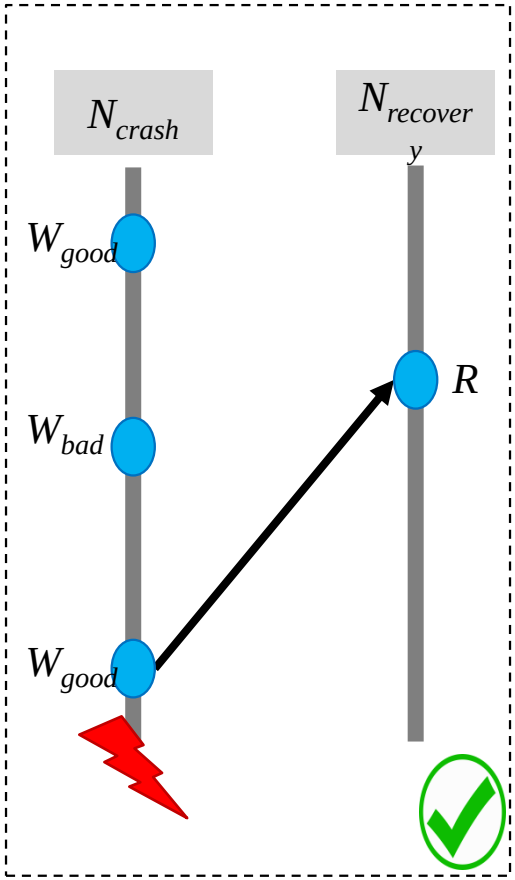
W_{good} and R should have **blocking** happens-before relationship[1] ($W \xrightarrow{b} R$)!

失效时序缺陷(7)

Crash-Recovery TOF缺陷

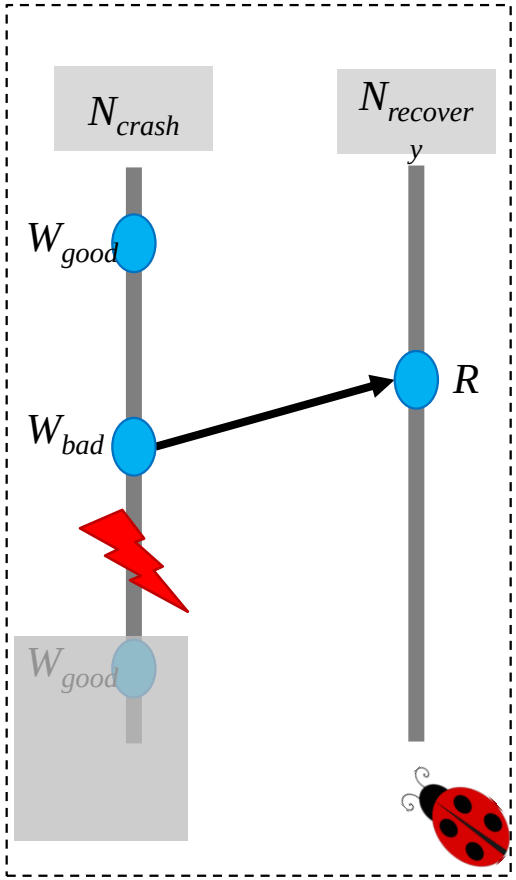


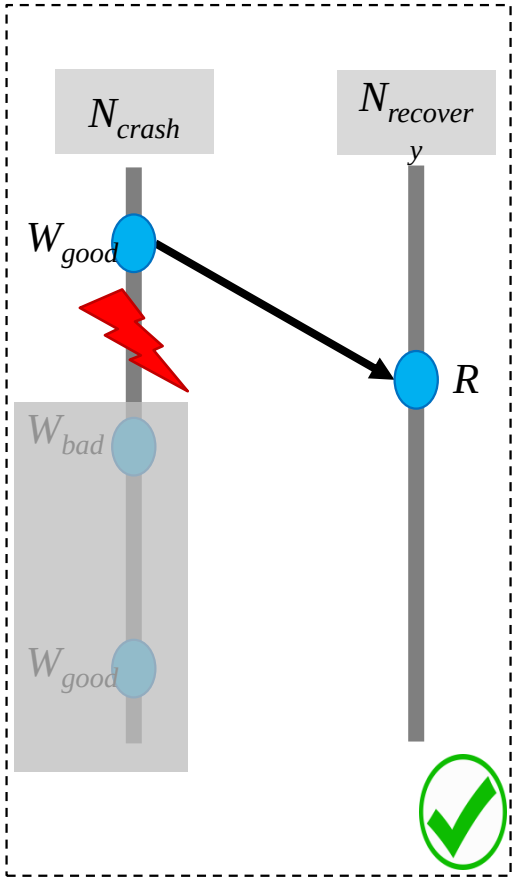


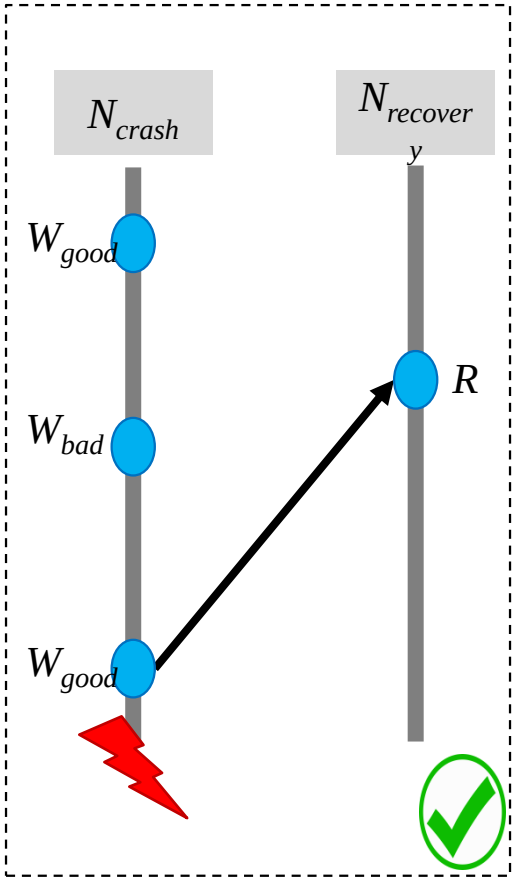


失效时序缺陷(7)

Crash-Recovery TOF缺陷







失效时序缺陷(8)

- Crash-Regular TOF缺陷检测
  - 因果分析，寻找并行操作A和B
  - 操作A等待操作B的消息
- Crash-Recovery TOF缺陷检测
  - One crash injected tracing
  - 操作B属于recovery中的读操作
  - 操作A属于normal run中的写操作
  - A与B操作同一个数据对象

失效时序缺陷(9)

- **Failure site静态分析：slicing分析**
  - **Crash-Regular TOF缺陷：**
    - 操作A是一个Blocking操作：Singal/wait或者while(condition)
  - **Crash-Recovery TOF缺陷：**
    - 操作B属于recovery中的读操作，并且某个严重的fatal error依赖该读操作
- **Failure tolerance静态分析**
  - **Crash-Regular TOF缺陷：**
    - Singal/wait或者while(condition)是否存在timeout机制
  - **Crash-Recovery TOF缺陷：**
    - If-checker: fatal error之前是否有状态判断

低效循环/同步性能缺陷

- 性能Profiling
  - 定位overhead较大的代码区域
  - 该代码区域与循环或者同步区域相关
  - 代码区域的访问频率：循环轮数、同步lock的访问次数和等待次数
- 循环
  - 分析循环条件依赖的变量
- 同步
  - 分析lock对象的粒度

性能传播缺陷(Cascading Bugs)

- 性能影响分析
  - 跨节点：因果分析
  - 单节点：需要考虑线程间的同步依赖

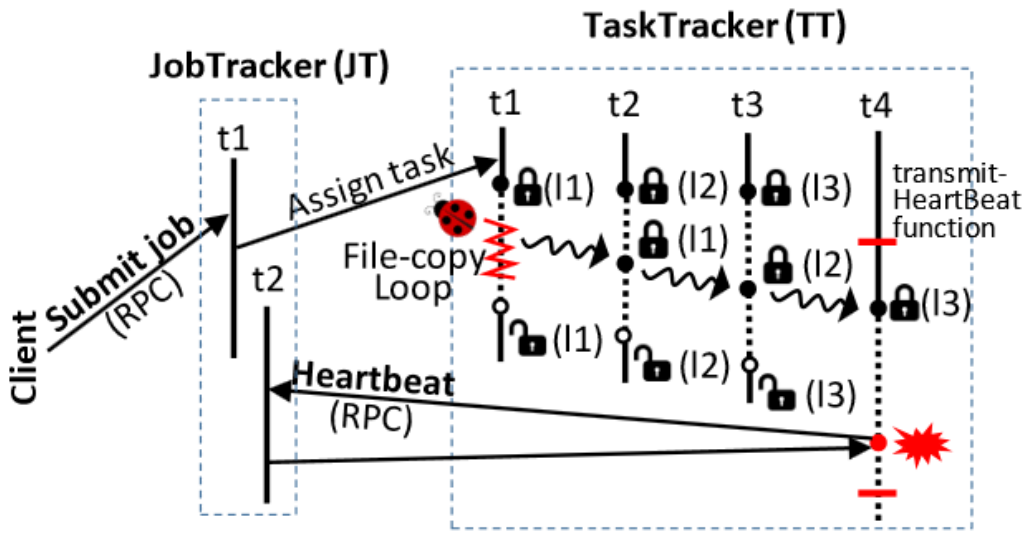


Figure 1: A PCbug in MapReduce: the client's job delays heartbeats through lock contentions and causes the whole Tasktracker node to fail.

动态分析小结

- 动态分析擅长
  - 判断对象是否相等
  - 判断操作间是否存在Job-driven的因果依赖
  - 记录执行频率和时间
- 一般用于缺陷的初步定位
- 缺陷的Root Cause或者Failure Site分析一般需要借助静态分析进一步定位

Q&A