基于分类的自适应失效检测系统①

杨立苑[1,2], 宋云奎[2], 张文博[2], 钟 华[2]

[1](中国科学院大学, 北京 100049)
[2](中国科学院软件研究所 软件工程技术研究开发中心, 北京 100190)

摘 要: 虚拟化环境中的实例失效往往会造成巨大的经济损失. 描述了一种面向虚拟化环境的失效检测系统设计与实现. 考虑了虚拟化环境的层次依赖性, 采用一种层次失效检测模型对不同层次的实例失效进行了分类, 提高了实例失效恢复的准确性; 通过自适应性动态周期失效检测机制均衡了虚拟机检测器的 CPU 资源消耗和时效性. 该系统在 OnceCloud[1]平台中进行了实现与实例验证.

关键词: 虚拟化环境; 失效检测; 层次结构; 失效分类

Self-Adaptive Failure Detection System Based on Classification

YANG Li-Yuan[1,2], SONG Yun-Kui[2], ZHANG Wen-Bo[2], ZHONG Hua[2]

[1](University of Chinese Academy of Sciences, Beijing 100049, China)
[2](Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)

Abstract: The failure of instances in virtual environment is always causing huge losses. This paper designs and implements a virtual-environment-oriented failure detection system. In consideration of hierarchical dependence in virtual environment, a hierarchical model is used to classify the failure of instances in different levels in order to improve the accuracy of instance failure recovery, and the mechanism of self-adaptive dynamic failure detection cycle is applied to the trade-off between the CPU resource cost and the timeliness of the virtual machine detector. This system has been implemented and verified in the OnceCloud platform.

Key words: virtual environment; failure detection; hierarchical model; failure classification

1 背景

虚拟化环境下的宕机会带来巨大的损失, 所以虚拟化环境的可靠性是研究与产业界关注的焦点. 其中, 可靠性的一个关键要素是失效检测. 根据 Tellme Networks 的技术报告, 检测失效的时间占用了失效恢复时间的 75%[2], 另有研究指出早期的失效检测能够缓解 65%的失效发生[3]. 所以在虚拟化环境中需要失效检测系统来检测实例的失效.

失效实例的最终行为可以分为两类: 一类是实例崩溃停止; 另一类是实例还在运行, 但内部繁忙或者其他问题无法响应检测信息的处理. 在传统失效检测的过程中, 两种基本检测方法(Push 和 Pull)都使用端到端的超时机制[4]. 这种超时机制很难区分实例的这两种不同行为. 在虚拟化环境下, 失效检测器如果不对以上两类最终行为进行区分, 则会给失效恢复的花费收益(Cost-Benefit)带来不利影响. 例如, 如果失效虚拟机的最终行为是崩溃停止(虚拟硬件丢失), 那么会采用快照克隆的恢复方式, 如果失效虚拟机的最终行为是无响应(虚拟机蓝屏), 那么可以采用简单的重启, 这些措施的收益和花费各不相同.

为了获取失效实例精确的信息, 本文提出了一种面向虚拟化环境的失效检测系统(OnceFD).

虚拟化环境通常分为物理机层、虚拟机层、Web 应用层, 层次之间的实例是一对多关系, 每一层的实例都会发生失效, 失效检测器的检测目标数量也成指数级增加. 所以仅仅依靠一个检测节点或者几个检测

① 基金项目:国家自然科学基金(61173004, 61363003);国家科技支撑计划(2015BAH55F02)
收稿时间:2014-12-17;收到修改稿时间:2015-01-29

节点的有限检测节点模型已经无法满足虚拟化环境的需求.

另一方面, 为了避免在物理机 CPU 资源紧张的情况下, 失效检测器与物理机抢占 CPU 资源, 失效检测器需要考虑均衡(Trade-Off)CPU 资源消耗和检测周期. 失效检测器服务质量的关键指针之一是失效检测器检测真实失效的时效性[5]. 失效检测时效性越好, 周期越短, 资源消耗的越多, 但是为了减少资源的消耗, 增加周期的长度, 又会影响失效检测的时效性.

面对上述问题和挑战, OnceFD 着重考虑了三个方面的问题: 1)虚拟化环境下失效检测模型; 2)虚拟化环境下失效实例最终行为分类问题; 3)虚拟化环境下虚拟机失效检测器的 CPU 资源消耗和时效性的 Trade-Off 问题.

2 相关工作

陈宁江等人针对 Web 应用服务器的失效情况, 首先定义了一种失效检测模型并且为此失效检测模型设计了一种基于 QoS 的适应性失效检测算法[6]. 该文只是针对 Web 应用服务器的失效进行了讨论, 但是虚拟化环境还包括了物理机、虚拟机等资源. 该失效检测模型并不能很好的适用于虚拟化环境中.

Joshua 等人针对应用如何获知失效的问题实现了一个失效报告服务: Pigeon. Pigeon 展现了一个新的概念: 失效通知者(Failure Informer) [7]. 这个失效通知者能够让应用获得通知、采用面向应用的恢复措施、在处于失效怀疑的时候进行安全处理. 但是它设计的感知器(Sensors)包括了 Process Sensor、Embedded sensor、Router Sensor 和 OSPF Sensor, 并没有设计针对虚拟机失效的 Sensor.

Chandra 等人引入了不可靠失效检测器(Unreliable Failure Detector, 下文简称 UFD)的概念并且研究了如何应用不可靠失效检测器解决异步系统由于崩溃失效造成不一致的问题[8]. 但是 Chandra 等人总结的不可靠检测器只能通过超时机制来判断检测目标是否失效, 但是并没有对失效实例的最终行为进行区分.

Chen 等人研究了关于失效检测器的服务质量(QoS)问题[5]. 该文在已定义的七个指标基础上, Chen 等人设计了一个新的失效检测算法并且分析了算法的服务质量. 为了满足失效检测器的服务质量要求, 该文在服务质量与网络带宽之间进行了均衡.

朱晨杰设计并且实现了一种面向虚拟化集群的故障检测系统 Smart ZK[9]. 该原型系统基于 Xen 和 Zookeeper 对物理机和虚拟机进行监控和检测. 但是该系统并没有检测 Web 应用层, 并且该系统没有考虑到时效性与资源消耗的关系, 导致系统不能随着外部资源消耗改变自身的服务质量.

3 OnceFD的基本模型

虚拟化环境中的 Web 应用层、虚拟机层、物理机层都可能会发生失效, 本文采用一种面向虚拟化的层次失效检测模型. 采用这种层次模型的主要原因在于: 第一, 下一层检测器可以检测到上一层的检测器是否失效, 并且某一层次检测器的失效并不会影响同一层次其他检测器的使用; 第二, 失效检测局限于某一个特定的范围内, 一个层次的失效检测器只检测一种实例, 降低了该检测器的逻辑实现复杂度; 第三, 一个层次的失效检测器只占用了该层次的少量资源, 节约了资源.

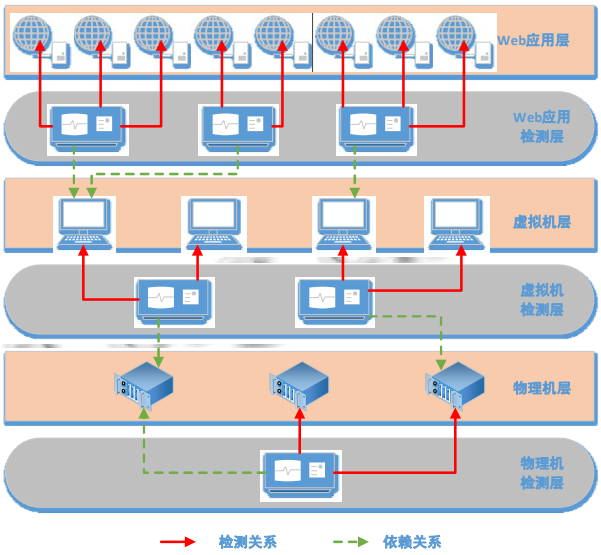

图 1 OnceFD 层次结构图

OnceFD 的层次结构如图 1 所示. OnceFD 包括 3 个层次: 物理机检测层、虚拟机检测层、Web 应用检测层. 其中物理机检测器运行在虚拟化环境中的某一台物理机上, 检测器对其余的物理机进行检测. 虚拟机检测器运行在虚拟化环境中的每一个物理机上, 它只对该物理机上运行的虚拟机进行检测. Web 应用检测器运行在虚拟化环境中的虚拟机上, 它只对该虚拟

机上运行的 Web 应用进行检测. 公式(1)定义了各层次检测器以及该检测器上一层检测实例的关系和算法. 其中 FD_i 代表第 i 层的检测器, $Uuid$ 代表该检测器的唯一标识符, $InstanceSet$ 代表检测器所检测的上一层实例, $Module_i$ 代表第 i 层检测器使用的检测模块.

$$FD_i = \{Uuid, InstanceSet, Module_i\} \quad (1)$$

由于虚拟机层或者物理机层中的某一实例发生失效, 运行在该实例上的检测器也会发生失效. 例如: 虚拟机层的某一台虚拟机突然停止运行, 该虚拟机上的 Web 应用检测器也会随之停止失效. 这也就是说上一层的检测器对下一层的实例有一定的失效依赖关系, 公式(2)定义了检测器与实例之间的失效依赖关系. 其中 $f_{i-fd_{i-1}}$ 是检测器与实例之间序偶的集, FD_{i-1} 是第 i-1 层的检测器, 是第 i 层的实例.

$$f_{i-fd_{i-1}} = \{< FD_{i-1}, Instance_i >\} \quad (2)$$

4 基于最终行为的失效分类检测

虚拟化环境下造成实例失效的原因很多, 表 1 是虚拟化环境下各层失效的列表. 在虚拟化环境中, 由于虚拟化备份、迁移所带来的便利性, 失效恢复过程中往往不去关注失效的具体原因, 只需要关注某些原因带来的实例失效行为. 例如: 虚拟机在正常运行时由于虚拟化资源的突然丢失造成了虚拟机的最终行为是虚拟机关闭, 这时恢复者只需要知道虚拟机关闭便能够去尝试启动该虚拟机. 所以 OnceFD 不用去关心失效的具体原因或者细节, 通过失效后的实例行为, 本文将失效分为以下两类: 崩溃失效(Stop Failure)和无响应失效(No Response Failure). 下文分别简称 SF 和 NRF. 崩溃失效是指实例停止执行, 不会再接收或发送任何消息. 无响应失效是指实例没有停止执行, 但也不会接收或发送消息. OnceFD 的失效分类如表 2 所示.

表 1 层次失效表

失效层	原因举例
Web 应用	内存错误, 断言失效, 达到退出条件, 程序死锁, 并发量过大
VM 层	OS 内核挂起, OS 内核崩溃, VM 资源丢失, VM 硬盘坏道
物理机层	虚拟机监视器停止运行, OS 崩溃, 内存错误, 断言失效

表 2 失效分类表

种类	层次	原因举例
SF	Web 应用	达到退出条件
	VM	VM 资源丢失
	物理机层	物理机关机
NRF	Web 应用	程序死锁, 并发量过大
	VM	OS 内核挂起
	物理机层	OS 崩溃

4.1 Web 应用检测器的关键技术

Web 应用检测器分布在 VM 层中. 当一个需要被检测的应用在虚拟机上启动时, 检测器会启动一个新的线程对应用进行检测. 当该应用将要关闭时, 检测器会停止对该应用检测的线程. 公式(3)形式化定义了 Web 应用失效检测逻辑.

$$\begin{cases} appPID \notin ProcessTable \rightarrow SF \\ (appPID \in ProcessTable) \wedge RequestTimeout \rightarrow NRF \end{cases} \quad (3)$$

SF: 当 VM 还在正常运行, 如果目标应用进程不在操作系统的进程表中, 这说明该应用已经不在运行的状态当中. 导致应用停止失效的原因有很多,例如应用的 Bug、触发了应用的退出条件、内存溢出.

NRF: 当 VM 还在正常运行, 如果目标应用的进程在操作系统的进程表中, 但却不响应进程检测器, 这说明进程已经无响应. 导致应用无响应失效的原因有很多, 例如应用死锁、链接数遇到了瓶颈.

4.2 虚拟机检测器的关键技术

虚拟机检测器分布在物理机层中. 当一台虚拟机启动时, 该虚拟机会向检测自身的虚拟机检测器进行注册. 当一台虚拟机关闭时, 该虚拟机会取消该注册. 公式(4)形式化定义了虚拟机失效检测逻辑.

SF: 为了检测这些状态的变化, 虚拟机检测器需要在内存中将每个虚拟机对应的状态记录下来, 动态周期性 T_{VMFD} 去对比每个虚拟机现在的生命周期状态和上一个时间点的生命周期状态. 但是仅仅这样对比是无法分辨出虚拟机生命周期的基本变化过程是由于失效引起的还是正常的用户操作. 所以另外还需要维护和用户操作有关的记录, 通过这个记录可以获取最近一次正常操作所导致的虚拟机生命周期状态, 虚拟机检测器检测到的虚拟机生命周期状态和记录中的状态是不一致的, 这就说明状态的变化是由于失效引起的.

$$\begin{cases} VMStateChange \wedge \neg ManualOpration \rightarrow SF \\ \neg(VMState = Down) \wedge HeartBeatTimeOut \rightarrow NRF \end{cases} \quad (4)$$

NRF: 针对虚拟机无响应的情况，检测器使用心跳的机制进行判断. 虚拟机在向虚拟机检测器注册之后，会周期性的向检测器发送心跳信息. 当检测器在一定的时间内没有收到虚拟机发送的心跳消息，并且该虚拟机还处于开机状态，检测器则认为该虚拟机发生了无响应失效.

4.3 物理机检测器的关键技术

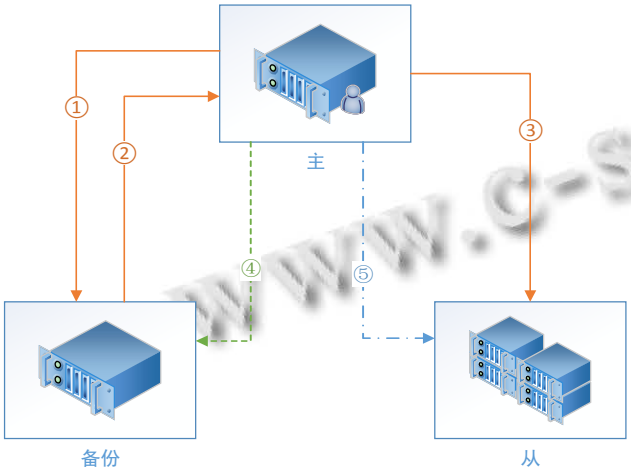

图 2　高可用检测体系结构

物理机检测器和物理机 Xen 内核中的 VMM 处于同一进程当中. 物理机检测器采用一种双机热备[2]的高可用检测体系结构，如图 3 所示，主要包括三个模块: 主节点模块、备份节点模块、从节点模块. 图 2 中实线 1、实线 2、实线 3 表示检测数据流，虚线 4 表示同步数据流，画点线 5 表示控制流. 检测数据流主要包括节点之间互相监听发送的请求数据. 同步数据流主要包括主节点到从节点的体系结构信息数据. 控制流主要包括控制从节点与备份节点之间的角色转换.

- 主节点(Master)模块: 相应备份节点的检测，维护双机热备的结构信息. 负责检测备份节点和从节点是否失效. 并且在备份节点失效的情况下，从从节点当中选取一个新的备份节点.
- 备份节点(Backup)模块: 响应主节点检测，同步主节点中的结构信息. 负责检测主节点是否失效. 并且当主节点失效的情况下，自己成为主节点，然后从从节点当中选取一个新的备份节点.
- 从节点(Slave)模块: 响应主节点检测.

4.3.1 节点之间的检测机制

公式(5)形式化定义了物理机失效检测逻辑.

$$\begin{cases} NoICMP \wedge IPMIPowerOff \rightarrow SF \\ NoICMP \wedge IPMIPowerOn \rightarrow NRF \end{cases} \quad (5)$$

SF: 本文使用智能平台管理接口(IPMI)来判断操作系统是否是正在运行的状态. IPMI 是一种开放标准的硬件管理接口规格，定义了嵌入式管理子系统进行通信的特定方法. 使用 IPMI 的优点在于可以只通过与主板的通信来快速确定操作系统的开关状态，这种判断方法和操作系统本身的内部运行状态、类型无关.

NRF: 主节点使用 ICMP 协议去 Ping 虚拟化环境中的其余节点. 如果其余节点在一定时间内无法响应主机点的请求，则判断该节点是无响应失效.

算法 1　监听算法

```
Module Master-Backup;
Input: Master, Backup, Slaves;
Output: Null;
while self is Master: //检测自身是否是 Master
    if Backup is On: //Backup 是否存活
        sleep a while
        continue
    clear old Backup
    //选举一个新的 Backup
    for each Host in Slaves:
        if Host is on:
            choose Host as Backup
            break
    //Master 同步结构数据到 Backup
    copy structure data to Backup
    finish set new Backup
Module Backup-Master;
Input: Master, Backup;
Output: Null;
while self is Backup: //检测自身是否是 Backup
    if Master is On: //Master 是否存活
        //从 Master 获取结构数据
        get structure data from Master
        sleep a while
        continue
    make self is Master
```

4.3.2 单点失效检测

检测器的高可用实现的基本方法就是节点之间互相监听. 当节点中的任何一台物理机发生失效，监听它的物理机都会发现并做出处理. 检测器的各个模块之间主要分为两个监听通道，监听算法如算法 1 所示.

- 主节点监听备份节点，当备份节点失效时产生一个新的备份节点.

- 备份节点监听主节点，当备份节点失效时产生一个新的主节点.

5 虚拟机检测器的Trade-Off策略

失效检测器检测真实失效的速度是失效检测器服务质量的关键指标之一[5]. 在虚拟化环境下，一方面，失效检测器需要提高服务质量，例如更加快速的检测出失效，但这样失效检测器会消耗更多的资源. 另一方面，失效检测器必须要尽可能少的消耗资源，不能影响正常服务的使用，但这样需要降低失效检测器的服务质量. 所以，这就需要失效检测器均衡(Trade-Off)服务质量和资源消耗.

本文的虚拟机检测器考虑到服务质量和 CPU 资源消耗之间的均衡，当服务器 CPU 资源使用紧张时，在可容忍的范围内增加虚拟机检测器的检测周期，降低虚拟机检测器的CPU资源消耗. 当服务器CPU资源宽松时，减少虚拟机检测器的检测周期，提高检测器的服务质量. 所以，虚拟机检测器的服务质量能够根据实际环境进行自适应的变化. 本文将预测 CPU 利用率，建立相应的约束模型，使用 CPU 利用率对虚拟机检测器的检测周期进行约束.

5.1 CPU 利用率的预测模型

本文使用卡尔曼滤波器对资源的 CPU 占用率进行预测. 卡尔曼滤波只要获知上一时刻状态的估计值以及当前状态的观测值就可以计算出当前状态的估计值. 假设: $cpu_{k|k-1}$ 表示 k 时刻 CPU 利用率的先验状态估计值. $cpu_{k|k}$ 表示 k 时刻CPU利用率的后验状态估计值. $P_{k|k-1}$ 表示 k 时刻的后验估计协方差. $P_{k|k}$ 表示 k 时刻的先验估计协方差. F 表示状态转移矩阵. H 表示量测矩阵. Q 表示过程激励噪声协方差. R 表示测量噪声协方差. K_k 表示滤波增益矩阵. B 表示输入控制模型. u_k 表示控制器向量. z_k 表示 k 时刻 CPU 利用率的测量值.

卡尔曼滤波器的操作包括两个阶段: 预测与更新. 在预测阶段，如公式(6)所示，滤波器使用上一状态的估计，做出对当前状态的估计.

$$\begin{cases} cpu_{k|k-1} = Fcpu_{k-1|k-1} + Bu_k \\ P_{k|k-1} = FP_{k-1|k-1}F^T + Q \end{cases} \tag{6}$$

在更新阶段，如公式(7)所示，滤波器利用对当前状态的观测值优化在预测阶段获得的预测值，以获得一个更精确的新估计值.

5.2 预测模型的参数选取

由于CPU利用率是一个标量值，所以本文将使用标量形式的卡尔曼滤波. F 取值标量 1, H 取值标量 1, I 取值标量 1. 初始状态变量 $cpu_{0|0}$ 和初始协方差 $P_{0|0}$ 影响比较小，并且会随着计算过程很快收敛，所以 $cpu_{0|0}$ 直接取首个测量值，$P_{0|0}$ 不能为 0，这里取 1. Q 的值越小越好，但不能为 0，这里取 e^{-6}. R 的取值不能过大，也不能过小，否则会造成滤波效果差，这里取 e^{-1}. 在本文中不存在控制增益，所以 B 取 0.

$$\begin{cases} K_k = P_{k|k-1}H^T(HP_{k|k-1}H^T + R)^{-1} \\ cpu_{k|k} = cpu_{k|k-1} + K_k(z_k - Hcpu_{k|k-1}) \\ P_{k|k} = (I - K_kH)P_{k|k-1} \end{cases} \tag{7}$$

5.3 虚拟机检测器的检测周期约束

本文使用 $cpu_{k|k-1}$ 对虚拟机检测器的进行质量约束. 虚拟机检测器的检测周期 T_{VMFD} 是关于物理机 $cpu_{k|k-1}$ 的分段函数，该函数如公式(8)所示:

$$T_{VMFD} = \begin{cases} 0.5s, cpu_{k|k-1} < 50\% \\ 2s, 50\% \le cpu_{k|k-1} < 80\% \\ 5s, 80\% \le cpu_{k|k-1} \end{cases} \tag{8}$$

6 实验验证

6.1 实验环境

本文使用三台物理机作为虚拟化环境的物理资源环境. 三台物理机配置都是24核CPU, 16G内存. 三台物理机互相通过千兆路由器连接. 每台物理机上部署CentOS6.4 操作系统，内核替换成 OnceXen[1], OnceXen 内核被分配 2 核 CPU. OnceXen 是由中科院软件研究所基于 Xen 研发的虚拟化内核. 在每台物理上部署虚拟机检测器和物理机检测器，在每个虚拟机中部署 Web 应用检测器. 每台物理机上运行 16 台 1 核、0.5G 内存的虚拟机.

6.2 OnceFD 失效检测分类功能验证

OnceFD 的功能测试分别针对 Web 应用检测器、虚拟机检测器、物理机检测器的 SF 和 NRF 进行测试. 用例和结果如表 3 所示.

6.3 虚拟化检测器 Trade-off 有效性测试

为验证虚拟机检测器中 Trade-Off 检测周期与 CPU 资源消耗的有效性，本文将对比含有自适应策略(Self-Adaptive, SA)模块、定期 3 秒策略(Regular 3s, R3s)模块和定期 0.5 秒策略(Regular 0.5s, R0.5s)模块的虚拟机检测器检测周期和 CPU 利用率.

本文分别在物理机 CPU 利用率是 7%左右、55%左右和 80%左右的情况下，记录使用上文三种策略的虚拟机检测器进的 CPU 利用率和检测周期. 测试结果

如图 3 和图 4 所示.

表 3 功能测试用例

检测器	注入失效	结果
Web 应用	使用 Shell 命令 kill 强制杀死 Web 应用进程.	SF
	使用 LoadRunner 模拟过量用户访问 Web 应用.	NRF
虚拟机	使用 Xen 命令强制关闭虚拟机.	SF
	使用 CPU 密集型程序消耗 VM 的所有 CPU 资源.	NRF
物理机	通过物理机开关强制关闭物理机.	SF
	使用 CPU 密集型程序消耗物理机的所有 CPU 资源.	NRF

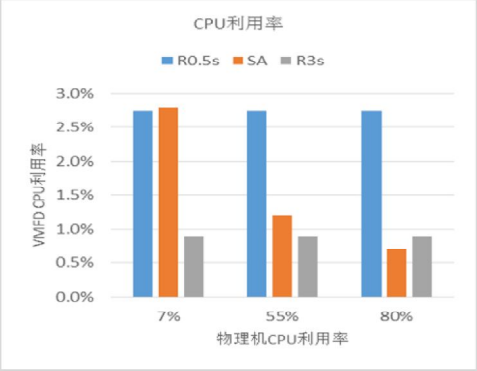

图 3 CPU 利用率测试结果

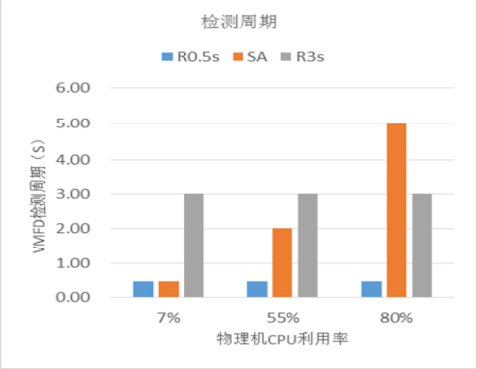

图 4 检测周期测试结果

由测试可以得出: 1)含有自适应策略模块的虚拟机检测器 CPU 利用率能够随着物理机 CPU 利用率的增加而减少, 减少了 CPU 资源的消耗; 而不含有自适应策略模块的虚拟机检测器的 CPU 利用率不会随着物理机 CPU 利用率的变化而改变, 其中 R0.5s 检测器在 CPU 资源紧张时仍然占用较多的 CPU 资源. 2)含有自适应策略模块的虚拟机检测器检测周期能随着物理机 CPU 利用率的减少而缩小, 加快了检测失效的速度. 而不含有自适应策略模块的虚拟机检测器检测周期不会随着物理机 CPU 利用率减少而改变, 其中 R3s 检测器在物理机 CPU 利用率较低时, 仍然保持较高的检测周期.

7 总结

虚拟化环境运行的过程中, 失效检测对于提升系统的可用性和避免巨大的损失发挥着重要的作用. 考虑到虚拟化环境的特点, 本文设计的失效检测系统采用了层次结构模型, 根据实例的失效行为对失效进行了分类, 使用动态周期检测机制均衡了虚拟机检测器的服务质量与资源消耗. 通过在 OnceCloud 平台下的功能验证和有效性测试, 本文设计的失效检测系统能够在虚拟化环境中实现满足需求的失效检测.

参考文献

1 http://www.once.com.cn/OncePortal/oncecloud.

2 Chen M, Accardi A, Kiciman E, ed. Path-based failure and evolution management. Proc. of the 1st Conference on Symposium on Network Systems Design and Implementation. San Jose. 2004. 23–36.

3 Oppenheimer D, Ganapathi A, Patterson DA. Why do Internet services fail, and what can be done about it? Proc. of 4th Usenix Symposium on Internet Technologies and Systems. Seattle. 2003. 1–16.

4 董辉,雷大军.分布式系统中失效检测器研究.计算机系统应用,2009,18(5):38–41.

5 Chen W, Toueg S, Aguilera MK. On the quality of service of failure detectors. IEEE Trans. on Computers, 2002, 51(1): 561–580.

6 陈宁江,魏俊,杨波,黄涛.Web 应用服务器的适应性失效检测.软件学报,2005,16(11),1929–1938.

7 Leners JB, Gupta T, Aguilera MK, ed. Improving availability in distributed systems with failure informers. USENIX Symposium on Networked Systems Design and Implementation. Lombard. 2013. 427–441.

8 Chandra TD, Toueg S. Unreliable failure detectors for reliable distributed system. Journal of the ACM, 1996, 43(2): 225–267.

9 朱晨杰.SmartZK:面相虚拟化集群的高可用服务系统[硕士学位论文].杭州:浙江大学,2012.