多租户应用的性能管理关键问题研究

林海略 [1,2]　韩燕波 [1]

（1 中国科学院计算技术研究所 网络重点实验室 北京 100190）

（2 中国科学院研究生院 北京 100190）

摘　要　SaaS 软件交付模式将应用软件以服务的形式提供给客户，可缩减硬件采购、系统管理上的开销。从 SaaS 服务提供商的角度，如何在维持较高的资源利用率的同时为各个租户提供一定的性能指标保障是一个挑战性问题。本文定义了一个特定的多租户架构 — MDSA，并从业务逻辑层和数据处理层两方面探索其性能管理问题，提出了基于延迟的应用级请求调度算法 ADRS 以及惰性副本管理算法 LRM。在业务逻辑层，ADRS 通过逐步降低服务需求较大的请求的优先级来避免其对整体性能造成影响。在数据处理层，LRM 通过动态调整负载在各个副本之间的分配以及副本在节点间的放置来适应负载的动态变化。本文将典型的 Web 应用 TPC-W 转换成多租户应用，并以此为基础进行了实验分析，结果表明了上述算法的可行性和有效性。

关键词　软件即服务；多租户；性能管理

中图法分类号　TP311

Performance Management for Multi-Tenant Web Applications

LIN Hailue[1,2]　HAN Yanbo[1]

[1](*Key Laboratory of Network Science and Technology*, *Institute of Computing Technology*, *Chinese Academy of Sciences*, *Beijing* 100190, *China*)

[2] (*Graduate University of Chinese Academy of Sciences*, *Beijing* 100190, *China*)

Abstract　Software-as-a-Service (SaaS) applications are becoming commonly known as the more flexible and easier-to-manage alternative to traditional in-house software. Multi-tenant architectures are becoming more and more prominently used among SaaS providers for its manageability and cost-effective scalability. For a multi-tenant Web application, the capability to ensure certain level of quality of service (QoS) guarantee is essential to achieve high user satisfaction. In this paper, a specific multi-tenant architecture—MDSA is proposed. In the context of MDSA, the performance management problem is discussed, and two key algorithms ADRS and LRM are proposed. By dynamically decreasing the priority of requests with a high service demand, ADRS is able to avoid resource hijacking. By dynamically adjusting replica placement and load dispatching among the database servers, LRM is able to adapt the variation of workload. Experiment results show that the reported works are feasible and effective.

Key words　Software as a Service; Multi-tenancy; Performance Management

作者简历：　林海略 (1982-)，男，博士研究生，linhailue@software.ict.ac.cn，电话:13811610642 主要研究领域为面向服务的计算；韩燕波 (1962-), 男，博士生导师, 研究员，yhan@ict.ac.cn，主要研究领域为面向服务的计算，分布式系统，工作流技术。

1. 引言

软件即服务(SaaS)模式通过将IT系统的开发及运营外包给 SaaS 服务提供商，减轻了客户构造、使用和维护软件应用的成本，因而获得了广泛的认可和使用。在SaaS 模式下，我们将具有共性需求的最终用户群体称作租户。最终用户以租户为单位租用软件。一个租户对应一个只为所用不为所有的虚拟系统。对于其租户，SaaS 提供商往往采用基于租赁的销售模式以及基于使用量和性能指标的计费模式。因而，SaaS 提供商需要其租户提供明确的性能保障，以提高应用的可信度和用户体验。

一般来说，SaaS 提供商会同时为多个租户提供软件服务，以利用规模经济效应，通过在租户间共享人力、设备等资源来降低成本、提高收益。本文将基于同一基础设施，同时为多个租户提供具有共性的软件服务的应用称为多租户应用。

多租户应用需要在保障各租户应用性能指标的同时，提高其整体资源利用率。如何实现该目标需要依据应用类型的不同而进行区分考量。本文所关注的是数据驱动的 Web 应用。这是一类常见的 Web 应用，其特点在于应用的功能和使用模式围绕着数据的增删改查来展开。常见的数据驱动的 Web 应用包括：客户关系管理(CRM)，网络商店等等。

与用于静态信息发布的 Web 应用不同，数据驱动的 Web 应用往往需要处理关键的业务数据、执行复杂的业务逻辑，因而需要保障较高的数据一致性级别，支持复杂的数据查询。另外，由于系统的负载可能随时间动态变化，因而多租户应用适合于建立在硬件云计算平台之上(如 Amazon EC2)，以便通过动态改变服务器的数量来适应负载的波动。具体的，表 1 总结了多租户应用的特点以及多租户应用与在线事务处理应用(OLTP)以及以 Google App Engine (GAE)等托管环境所支持的应用之间的不同之处。

多租户应用的上述特点使得针对 OLTP 应用的架构难以适应其较大的数据量以及负载的动态变化，而 GAE 的架构难以满足其数据一致性以及查询表达能力方面的需求，因而需要重新考虑多租户应用的架构设计问题。具体的，多租户应用的架构设计依据租户间共享资源的方式的不同可以分为两种类型[1]：独享应用实例模式以及共享应用实例模式。前者为每个租户提供独立的物理节点或者虚拟机以及运行在其上的独立应用实例，后者在多个节点上运行单一应用实例来支撑所用的租户。

在本文中，我们关注共享应用实例模式。一般来说，采用共享应用实例的架构意味着能够支持较高的租户密度以及较低的软件管理和维护成本，然而实现该架构也面临着应用定制、按需扩展、性能管理等方面的挑战，本文主要关注于该架构下的性能管理问题。

表 1：多租户应用的特点

	OLTP	GAE 之上的应用	多租户应用
数据量	较小	可能很大	每个租户的数据量不大，但需要支持大量的租户
一致性	要求严格的一致性	不要求严格的一致性	要求严格的一致性
查询的表达能力	支持复杂的查询	只支持简单的查询	支持复杂的查询
可用资源量	静态的	动态的	动态的，每个租户按照其资源使用量付费。

本文第二节对共享实例型多租户架构做了进一步介绍，并给出了一个具体的架构-MDSA，第三节阐述 MDSA 中的业务逻辑层的性能管理，第四节阐述 MDSA 中的数据管理层的性能管理，第五节对 MDSA 及其性能管理技术进行试验分析，第六节介绍了相关工作，最后是结论。

2. 共享应用实例的多租户架构

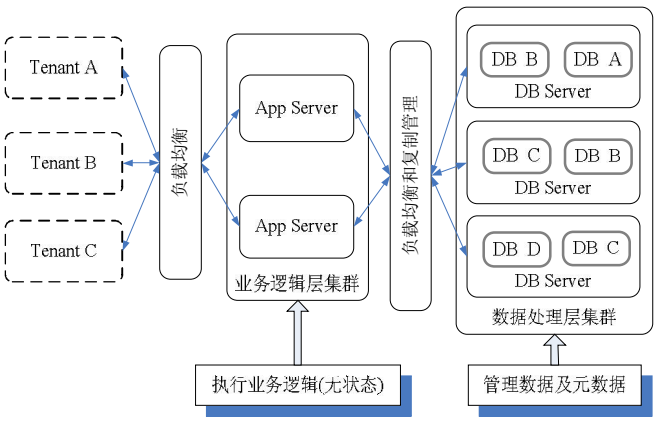


图 1：MDSA 多租户架构

本文讨论的多租户架构如图 1 所示，该架构采用元数据驱动模式来保障其可定制性，通过将状态保存在数据库层以及基于租户的数据库划分来保障其可扩展性，因此被命名为 MDSA (Metadata-Driven Scalable Architecture)。MDSA 由业务逻辑层和数据处理层构成。一个完整的请求处理过程从业务逻辑层开始，中间可能需要多次访问数据处理层，直到生成完整的应答并且发送回客户端为止。由于前文提到的应用特点，MDSA 采

用关系型数据库来管理各个租户的数据。进一步，为了保障数据的高可用性，MDSA 为每个租户数据库维护了多个分布在不同节点上的数据副本。在 MDSA 中，每一个层次都可以通过使用多个节点来提高其处理能力。在这种情况下，每一个层次都需要增加一个请求分发器用来在节点之间进行负载均衡。

MDSA 架构继承了经典的多层架构灵活、模块化等好处，然而，多租户应用的特定需求还在可定制性、可扩展性、可管理性等方面为 MDSA 的设计提出了新的挑战。另外，为了避免每个采用 MDSA 架构的多租户应用都需要重复考虑这些问题，本文提出了一个支持该架构的基础平台。具体的，多租户平台总结和实现了应对上述问题的通用方案，以缩减构造每个特定的多租户应用的开销。下面我们分别从上述几方面来进一步介绍 MDSA 的设计及多租户平台在其中的作用。

2.1 可定制性

企业的业务需求千差万别，每个企业都期望拥有适合自己的租户应用，这使得可定制性成为了多租户应用的一个重要方面。具体的，支持应用定制需要考虑如下问题：如何支持一个租户扩展默认的数据对象或者引入全新的数据对象？如何支持不同的租户定义的不同数据对象的共存？如何在数据对象可能变动的环境下编写业务逻辑？如何支持每个租户添加自己的业务逻辑？当业务逻辑和数据对象有所变动时，如何维护它们之间的一致性？

在 MDSA 中，对可定制性的支持是通过采用元数据驱动的架构[2]来实现的，即将应用中易变或不确定的部分从软件中剥离出来，用元数据（Meta-data）来描述它们。具体的，适合用元数据表示的可变的部分包括：软件界面的呈现逻辑；软件中涉及到的业务规则；软件中涉及到的流程；各种报表的扩展字段等等。

表 2：支持可定制化的通用组件

	可以由应用平台提供的通用组件	
界面呈现	创建和修改呈现元数据的编辑工具；	基于界面元数据生成客户界面的页面模板引擎；
业务规则	创建和修改业务规则元数据的编辑工具；	基于规则元数据判定业务逻辑走向的规则引擎；
业务流程	创建和修改业务流程元数据的建模工具；	基于流程元数据执行流程步骤的流程引擎
报表	创建和修改业务报表元数据的编辑工具；	基于报表元数据生成和展现业务报表的报表引擎

由于应用的定制往往与应用本身的业务功能有着紧密的联系，因此对可定制性的支持需要在特定应用的设计开发阶段就有所考虑。然而，多租户平台可以通过提供常用组件和推广最佳实践的方式提高其开发效率。具体的，表 2 列举了一些常见的元数据处理相关的组件。

元数据驱动的架构将每个租户应用可能的不同之处都提取成为元数据。这样，定制一个租户应用就表现为修改该租户应用的元数据，进而避免了应用定制对软件整体逻辑的影响，为不同的租户定制的数据对象和业务逻辑的共存提供了支撑。

2.2 可扩展性

对于多租户应用，可扩展性指的是当租户的数量及其负载增加或减少时，系统能够通过添加或删除服务器来应对这一变化。下面我们分别从业务逻辑层和数据处理层两方面来探讨 MDSA 的可扩展性。

对于业务逻辑层，我们采用无状态模式来保障其可扩展性，即应用服务器本身不保存状态，因而，对于一个 HTTP 请求，该请求可以分发到任意一个应用服务器来处理。这样，我们可以通过添加或减少应用服务器的数量来动态伸缩业务逻辑层的处理能力。

对于数据处理层，高效的处理跨节点、大数据量的查询是非常困难的，因而影响可扩展性的关键因素在于如何避免跨节点的查询。MDSA 通过数据划分来避免跨节点的查询。具体的，MDSA 将所有租户的数据划分为多个租户数据库，其中每个数据库包括一个或多个租户，而每个租户只属于一个租户数据库。由于单个租户预期的数据量不大，通过选定合适划分模式可以使得 1) 单个数据库节点能够容纳多个租户数据库; 2) 单个数据库节点所容纳的租户数据库数量不会太多，以避免管理元数据的开销[3]。上述划分使得每个数据库查询都可以对应到一个特定的租户数据库，进而可以由一个节点处理，避免了跨节点的查询，因而保障了数据处理层的可扩展性。

2.3 可管理性

对于多租户应用，可管理性指的是如何管理系统中的资源总量以及资源在租户间的分配来应对负载变化，避免租户间性能相互影响，以期在维持较高的资源利用率的同时为各个租户提供一定的性能指标保障。

Web 应用请求的处理过程可能会涉及到多种不同的资源，其中资源利用率较高的资源被称作资源瓶颈。对于不同类型的应用其资源瓶颈可能不同，但是对于特定的应用来说，资源瓶颈往往只有一个。[4]等工作都表明，动态 Web 应用的资源瓶颈在于 CPU 资源，因此，

本文主要考虑对 CPU 资源的管理。

已有工作提出了很多自动化的性能和资源管理手段。其中常用的资源管理手段包括请求调度、负载分配、准入控制、资源提供、资源划分等等。

借鉴已有工作，本文从业务逻辑层及数据处理层两方面出发，探讨 MDSA 架构下的性能管理遇到的新问题及已有工作的适用性。

前面提到，MDSA 的业务逻辑层节点是无状态的，即每一个请求可以在任意一个节点进行处理。因此，租户之间的竞争主要表现为来自于不同租户的请求之间的竞争。与业务逻辑层相关的性能管理手段主要有请求调度，负载分配，准入控制，资源提供等等。而无状态模式下的负载分配、准入控制以及资源提供都有着较成熟工作，因此在业务逻辑层，本文的讨论主要围绕请求调度展开，具体内容在第三节介绍。

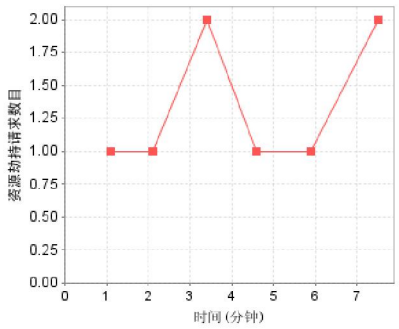
MDSA 的数据处理层是有状态的服务，即按照管理的租户数据库的不同，各个节点有着不同的状态，处理来自于不同租户的负载。因此，租户之间的竞争主要表现为放置在同一个节点中的不同租户数据库之间的资源竞争。性能管理的关键问题在于租户数据库的副本在节点间的放置以及负载在各个副本之间的分配。因此在数据处理层，本文的讨论主要围绕负载分配和副本放置两方面展开，具体内容在第四节介绍。

3. 业务逻辑层的性能管理

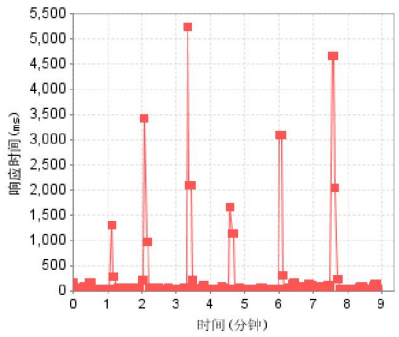
业务逻辑层的性能管理面临的挑战在于请求服务需求的不可预计性。这种不可预计性是由 Web 应用本身的复杂性引起的。以业务流程管理应用中的流程启动请求为例，依据该请求的参数，应用逻辑会查找相应的流程定义并启动一个新的流程实例。这个过程中消耗的资源量与流程定义的结构、流程的启动参数等因素密切相关，因此处理不同的流程启动请求所消耗的资源量可能相差很大。

请求服务需求的不可预计性会使得当某些请求的服务需求远远大于其他的请求时，即使其数量很少，也会导致其他请求的性能急剧下降。我们将这一现象称为资源劫持，将导致资源劫持的请求称为资源劫持请求。图 2 给出了一个资源劫持现象的实例，图 2(a)是资源劫持请求到来过程，图 2(b)展示了系统平均响应时间的变化过程，可以看出资源劫持请求的到来会严重影响请求的正常处理。

资源劫持现象的发生意味着一个租户的请求可能会影响其他租户请求，因此多租户应用应当避免这类现象。解决该问题一个直接方法是为请求的处理时间设置一个严格的阈值，当一个请求的处理时间超出该阈值则丢弃该请求。然而，这种方式给应用的开发带来较大的限制，使得开发人员难以实现一些复杂的请求处理逻辑。因此，本节我们考虑如何在避免资源劫持的同时支持较为宽松的请求处理时间阈值（例如 Google App Engine 对于一个请求执行的最长限额为 30 秒，而相对的，一个简单的 HTTP 请求的处理时间只需要几毫秒）。



(a)



(b)

图 2：资源劫持现象

多租户应用对各个租户提供的性能保障是和特定的指标联系在一起的。由于 Web 应用交互式的访问特征，请求的响应时间是提高客户满意度的关键性指标。具体的，本节采用延长因子作为主要的性能指标。延长因子是指请求的响应时间和服务需求的比值。其优势在于不依赖于具体的业务处理逻辑，能够用来单独分析应用运营商的服务水平。

从解决手段的角度，请求调度策略可以如下分类：如果一个策略的执行需要预知所有已经到达的请求的服务需求，则称该策略是透视的(Clairvoyant)。如果一个策略不需要预知请求的服务需求，则称该策略是非透视的(Non-Clairvoyant 或者 Blind)。由于请求服务需求的不可预计性，在 MDSA 中难以使用透视的请求调度策略。

基于上述分析，本节的研究目标可以定义为：设计一个非透视的请求准入策略，使得存在资源劫持请求

时，业务逻辑层请求的延长因子 $SF= R/D$ 的数学期望不应剧烈变化，其中 R 为请求的响应时间，D 为请求的服务需求。

3.1 延迟调度算法 ADRS

已有工作表明，以最小化平均响应时间为目标，最短剩余处理时间优先调度策略(SRPT)是最优的调度策略。图 3 展示了存在资源劫持请求的情况下轮转调度策略 RR 和 SRPT 的性能比较。可以看出，SRPT 能够明显缩减由于少量资源劫持请求带来的性能下降。然而 SPRT 是透视的算法，其使用需要知道每个请求的服务需求。因此，我们的设计思路是设计一个能够模拟 SPRT 行为的非透视的算法。

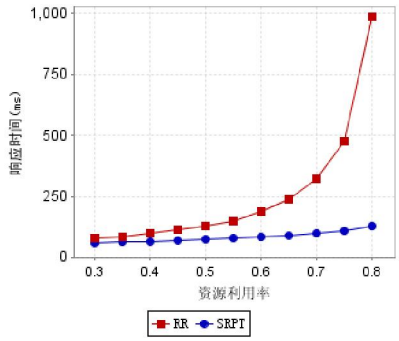


图 3：存在资源劫持请求时不同调度策略的表现

参照 SPRT 调度策略，本节提出 ADRS(Application-level Delay-based Request Scheduling)调度算法。ADRS 对每个请求处理过程中的资源消耗情况进行监控，并动态降低资源消耗较多的请求的优先级，以避免资源劫持现象。具体的，我们引入如下概念：

定义 1：请求的挂起和继续：给定一个 HTTP 请求，应用层请求调度器主动暂停其处理过程的事件称为请求的挂起；相对的，重新将一个挂起的请求恢复为可执行状态的事件称为请求的继续。

定义 2：请求的连续处理时间：给定一个 HTTP 请求，允许该请求不挂起连续执行的最长时间，记作 T_c。

定义 3：请求的挂起延迟时间：给定一个 HTTP 请求，该请求挂起到下一次尝试继续的时间间隔称为挂起的延迟时间，记作 T_d。

基于上述定义，ADRS 的描述如表 3 所示。可以看出，在 ADRS 中，每个请求被挂起的次数(delayCount)被用作该请求的相对优先级。当 delayCount=0 时，请求的优先级最高。对于服务需求较大的请求，其 delayCount 会随着请求的处理过程逐渐增加，相应的其优先级也逐渐下降。这样的性质与 SPRT 调度策略的思路是一致的。该算法与操作系统中的多级反馈队列调度(MLFQ)算法也有相似之处，其区别在于 MLFQ 是以线程为调度对象，而 ADRS 的调度对象是 HTTP 请求。

表 3：延迟调度算法(ADRS)

```
//每一个请求执行线程 T 执行如下算法，其中：
//变量 delayCount，某一请求被挂起的次数。
//函数 okToContinue，判定当前的请求是否应当跳出挂起状态。
while(true){
    delayCount = 0;
    Request r = retriveRequestFromQueue();
    while(true){
        executeRequest(r, Tc);
        if(r.isFinished()){
            break;
        }
        delayCount++;
        while(true){
            sleep(Td);
            if(okToContinue()){
                break;
            }
        }
    }
}
```

```
boolean okToContinue(){
    Set<Thread> threadPool = getAllThreads();
    int minDelay = MAX_INT;
    for(Thread t : threadPool){
        if(t.isRunning()){
            if(t.delayCount < minDelay){
                minDelay = t.delayCount;
            }
        }
    }
    if(delayCount == minDelay){
        return true;
    }
    return false;
}
```

T_c 和 T_d 是 ADRS 中的关键参数，其设置对调度算法的性能有着重要影响。直观上考虑，T_c 过小会导致很多请求都需要经历延迟，会大幅降低系统的整体性能；另一方面，T_c 过大会降低系统对资源劫持请求的防御能力。因此，T_c 应当取较为适中的值。对于 Web 应用，已有的研究表明[5]，与网页交互时人的反映时间为 200ms 左右，即当请求响应时间小于 200 ms 时，用户能够流畅的使用 Web 应用。因此，考虑到网络传输等因素，一个请求在服务器端所消耗的时间应当小于 200 ms。进一步，考虑到由于资源竞争带来的延迟，软件开发商应当将每个请求的服务需求限制在几十毫秒以内。因此，T_c =100ms 是一个较为合适的选择。

对于延迟时间 T_d 来说，其值设置过大会导致被延迟的请求迟迟得不到再次运行的机会，降低被延迟的请求的性能，造成潜在的资源浪费。另一方面，其值设置过

小则可能导致被延迟的请求过于频繁的尝试再次运行，增加调度算法的开销。这里，一次判定 okToContinue()所需的开销包括两次线程上下文切换以及一次okToContinue()的运行。基于上述分析，设置 T_d 时应当在避免较大开销的前提下选择较小的值。后续的章节通过一系列实验表明，将 T_d 设置为 5ms 能够取得较优的性能和较低的开销。

3.2 ADRS 的实现原理

为了实现 ADRS 算法，需要对每个请求的 CPU 资源使用情况进行实时监控。由于通过修改操作系统内核来进行请求调度有着影响的范围较大、修改的难度较高、不便于移植等不足，因此本节介绍了一种基于代码转换的 CPU 资源监控机制，以实现 ADRS。本节描述的方法是特定于 Java 语言平台的，然而我们认为其思路具有一定的通用性，能够扩展到类似的语言运行环境中，例如 .NET。

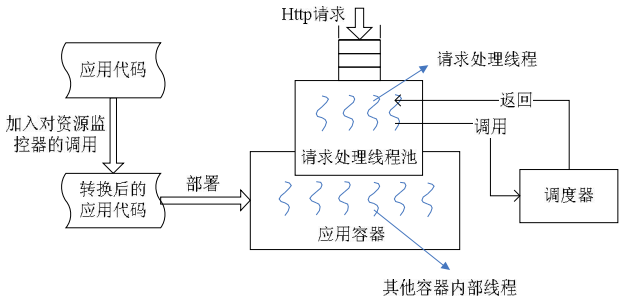


图 4：ADRS 的实现原理

图 4 展示了 ADRS 的实现原理。其基本思路是：1)在应用代码部署之前对其进行分析，找出一组适当的资源检查点，并插入资源检查代码，将转换后的代码部署到多租户平台。2) 每个 HTTP 请求的到来都会触发一个请求处理线程的执行，执行过程中会不断的遇到第一步插入的资源检查点。这时请求处理线程执行资源检查代码，并且在请求使用完当前资源限额时执行延迟调度算法。

在上述思路中，如何选定一组适当的资源检查点是提高上述算法性能的关键。选取较多的检查点可能会带来不必要的性能开销，选取较少的检查点则存在损害资源使用监控精度的风险。我们使用一种渐进式的方式来得到合适的检查点集合。具体的，我们首先采用保守的策略找到一组能够保证精度要求的检查点，并随后在不损害精度的前提下逐渐缩减检查点的个数。

保守的检查点设置策略如下：

1) 在每一个方法的开始加入对 checkConsumption()的调用。
2) 在每一个循环的开始加入对 checkConsumption()的调用。
3) 在每一个方法中不包含循环的、不包含函数调用并且长度超出 *MAXPATH* 的代码块的结束加入对checkConsumption()的调用，其中 *MAXPATH* 是系统定义的一个常量。

上述检查点设置模式使得当不考虑递归时，每执行 *MAXPATH* 长度的代码，至少会调用一次资源检查方法。在考虑递归时，假设系统为线程分配的栈的最大深度为 *MAXSTACK*，则每执行 *MAXPATH*MAXSTACK* 长度的代码，至少会调用一次资源检查方法。通过设定 *MAXPATH* 以及 *MAXSTACK* 的值，系统能够确保一定的资源监控精度。

为了进一步减轻资源监控开销，下面我们介绍几种减少资源检查点的方法。

首先，我们将不包含其他方法调用的方法称为原子方法。显然，省略掉原子方法中的第一类检查点对于监控的精确度的影响不大。由于 Java 应用中常常存在大量的原子方法(getters and setters)，所以省略对原子方法中的检查点能够大幅缩减监控的开销。

其次，在每个循环的每次执行中都调用一次资源检查代码往往是不必要的，我们可以在每个方法中加入一个局部变量来记录循环的执行次数。并且只在循环执行的次数超出特定阈值 *MAXLOOP* 时，才真正调用一次资源检查方法。通过使用局部变量操作来代替部分方法调用能够缩减一定的监控开销。

4. 业务逻辑层的性能管理

上一节提到，由于业务逻辑的多样性，HTTP 请求的服务需求是难以预计的。而对于数据库层，其访问接口是更为规范的结构化查询语言(SQL)，已有工作能够在执行一个 SQL 语句之前预计其资源使用量，这使得我们可以假定每个查询的服务需求是可以预知的。这一特点使得在数据库层，可以通过禁止应用执行服务需求过大的查询来避免资源劫持现象，因此每个数据库节点的查询调度策略不再是需要解决的核心问题。

在数据处理层，性能管理的难点来自于节点的有状态性，即对于一个租户数据库的查询，只有拥有该租户数据库副本的数据库节点才能处理。节点的有状态性使得数据处理层的性能管理面临着如下挑战。

1) 负载分配策略的制定：在副本放置固定的前提下，如何将负载分配到各个节点中是性能管理问题的一

个关键子问题，其难点在于在有状态性的制约下，如何尽可能的在节点间均衡负载，以提高系统的整体性能。

2) 副本放置策略的制定：在某些情形下，有状态性的限制会使得单纯的负载分配无法实现负载的均衡，需要考虑副本放置的调整。副本放置调整的内容包括节点总量的调整，副本到节点的映射的调整等等。除均衡负载之外，副本放置调整还需要考虑最小化资源使用量和最小化副本迁移次数，这些要求进一步增加了制定副本放置策略的难度。

3) 副本调整的执行：副本调整的执行关注的指标包括调整的速度、调整过程对一致性的影响、调整过程对性能的影响等等。

本章的内容主要关注于上述问题中的前两个。副本调整的执行可以较为独立的分析和求解，这方面的工作可以参考[6]。

多租户数据管理系统主要由数据库节点、租户数据库副本以及外部负载等几方面构成，其参数主要包括用于描述节点、租户以及负载的基本参数、用于描述系统中可控部分的控制变量以及用于描述系统预期运行状态的阈值变量。为了便于问题讨论，下面首先引入本节使用的一些符号：

表 4：多租户数据管理系统的基本参数

符号	含义
T	租户集合，其中第 i 个租户记作 t_i
N	租户的数量
S	数据库节点集合，其中第 j 个节点记作 s_j
M	数据库节点的数量
Λ_i	来自于租户 t_i 的请求到来速率，其中 Λ_i^{R} 表示来自于租户 t_i 的读请求到来速率，Λ_i^{W} 表示来自于租户 t_i 的写请求到来速率。
$\boldsymbol{\Lambda}$	系统的总体负载向量，即：$\boldsymbol{\Lambda}=<\Lambda_1,\Lambda_2,...,\Lambda_N>$。
$SF_{i,j}$	租户 t_i 的请求在 s_j 上执行时的延长因子(Stretch Factor)指标
R_i^{T}	租户 t_i 的副本数量。进一步，定义 $R_{\max}^{\mathrm{T}}$ 和 $R_{\min}^{\mathrm{T}}$ 分别为 R_i^{T} 的最大值和最小值。定义 $R_{\mathrm{default}}^{\mathrm{T}}$ 为 R_i^{T} 的默认值。

表 5：多租户数据库管理系统的控制变量

符号	含义
$\mathbf{K}$	服务器使用向量，k_j=1 表示服务器 s_j 处于运行状态，k_j=0 表示服务器 s_j 处于关机状态。
$\mathbf{A}$	数据库副本的放置矩阵，即：$\mathbf{A}=\begin{pmatrix} A_{1,1} & \cdots & A_{1,M} \\ \vdots & \ddots & \vdots \\ A_{N,1} & \cdots & A_{N,M} \end{pmatrix}$。其中如果服务器 s_j 维护着租户 t_i 的一个数据库副本，则 $A_{i,j}$=1；否则 $A_{i,j}$=0
$\mathbf{A}_0$	上一次副本调整过程中生成的副本放置矩阵
$\boldsymbol{\lambda}$	负载分配矩阵。即：$\boldsymbol{\lambda}=\begin{pmatrix} \lambda_{1,1} & \cdots & \lambda_{1,M} \\ \vdots & \ddots & \vdots \\ \lambda_{N,1} & \cdots & \lambda_{N,M} \end{pmatrix}$。其中 $\lambda_{i,j}$ 表示来自于租户 t_i 的负载分配到服务器节点 s_j 的请求速率。进一步，$\lambda_{i,j}^{R}$ 表示来自于租户 t_i 的读负载分配到服务器节点 s_j 的请求速率；$\lambda_{i,j}^{W}$ 表示来自于租户 t_i 的写负载分配到服务器节点 s_j 的请求速率。

表 6：多租户数据库管理系统的阈值变量

符号	含义
$SF_{\max}$	数据查询请求的延长因子阈值。
$U_{\max}$	每个节点的最大资源利用率阈值。
U_{target}	系统总体的目标资源利用率。
$U_{upperThreshold}$	系统总体资源利用率的上限浮动范围。
$U_{lowerThreshold}$	系统总体资源利用率的下限浮动范围。

4.1 负载分配

为了准确的分析负载分配问题，需要明确负载均衡程度的度量指标，并且建立数据库节点的性能模型，以用来在真正调整资源管理策略之前预计调整的效果。

基于已有工作，我们采用一个具有多个请求类型的 M/G/1/PS 队列作为每个数据库节点的性能模型。按照排队论结果，在节点 s_j 上执行的请求的延长因子为：

$$SF_{i,j}=\frac{1}{1-U_j}=\frac{1}{1-\sum_{i=1}^{N}\left(\lambda_{i,j}^{W}D_i^{W}+\lambda_{i,j}^{R}D_i^{R}\right)}$$

为了衡量一个负载分配策略达到的负载均衡程度，我们定义如下概念：

定义 4：负载倾斜度：$Skew=\sum_{j=1}^{M}\frac{1}{1-U_j}$

其中 U_j 是指节点 s_j 的资源利用率，并且 $0\le U_j<1$。基于该定义，我们可以得到负载倾斜度有如下的性质：

1) 当某一节点的资源利用率较高时，该节点对于负载倾斜度的贡献会大大提高。当某一节点的资源利用

率趋近于 100%时，系统的负载倾斜度趋近于无穷大。

2) 由于负载 $\boldsymbol{\Lambda}$ 是固定的，因此所有节点的资源利用率的和 $\sum_{j=1}^{M} U_j$ 也是固定的。在上述前提下，可以证明当每个节点的资源利用率相同时，负载倾斜度最小。

上述两个性质说明，负载倾斜度能够体现集群中节点负载的不均衡程度。

基于以上得到的性能模型和度量指标，我们可以将负载均衡问题进一步形式化为：

$$\min\ Skew(\boldsymbol{\lambda}) = \sum_{j=1}^{M} \frac{1}{1-\sum_{i=1}^{N}\left(\lambda_{i,j}^{W} D_i^{W} + \lambda_{i,j}^{R} D_i^{R}\right)}$$

其中的限定条件为：

$$\forall i \in [1,N] \quad \sum_{j=1}^{M} \lambda_{i,j}^{R} = \Lambda_i^{R} \tag{1}$$

$$\forall i \in [1,N], j \in [1,M] \quad \lambda_{i,j}^{\mathrm{R}} = 0 \text{ if } A_{i,j} = 0 \tag{2}$$

$$\forall i \in [1,N], j \in [1,M] \quad 0 \le \lambda_{i,j}^{R} \le \Lambda_i^{R} \tag{3}$$

$$\forall i \in [1,N], j \in [1,M] \quad \lambda_{i,j}^{W} = \begin{cases} \Lambda_i^{w} & \text{if } A_{i,j} = 1 \\ 0 & \text{if } A_{i,j} = 0 \end{cases} \tag{4}$$

$$\sum_{i=1}^{N}\left(\lambda_{i,j}^{W} D_i^{W} + \lambda_{i,j}^{R} D_i^{R}\right) < 1 \tag{5}$$

其中约束（1~3）限定了读请求分配的合理取值范围。约束（4）限定了写请求分配的合理取值范围。约束（5）要求对每个节点，为其分配的负载总和应该小于其处理能力。

上述问题属于一类特殊的优化问题，即资源分配问题。[7]给出了该问题的一个基于图的高效算法。我们将该算法得到的负载分配矩阵称为最优负载均衡方案(Well Balanced Load，WBL)，并将上述算法称为 **WBL 生成算法**。

WBL 只确定了整体上请求速率的分配，负载分配算法的实现还需要考虑每个请求的分配决策。为此，我们引入如下概念：如果在某一个负载分配算法 A 的控制之下得到的负载倾斜度与 WBL 之下的负载倾斜度相同，则称 A 为一个 **WBL 实现算法**。下面给出了两种 WBL 实现算法。

首先，我们将 WBL 之下每个租户 t_i 分配到节点 s_j 的负载强度 $\lambda_{i,j}$ 用作加权轮转调度算法的权重，并且将这一调度方式称作最优加权轮转调度(Optimal Weighted Round-Robin，OWRR)。由其定义可知，OWRR 是一种 WBL 实现算法。

其次，我们提出一个称为 SSQF (Stateful Shortest Queue First)的算法，对于租户 t 的请求，SSQF 将其分发到存有 t 的数据副本的节点集合中队列长度最小的节点。通过图 5 所示的模拟实验结果可知(其设计细节参见第 5.2 小节)，SSQF 能够达到与 OWRR 相近的负载均衡程度，因此 SSQF 也是一种 WBL 实现算法。这两种算法的性能比较和适用性分析将在后续章节中详细介绍。

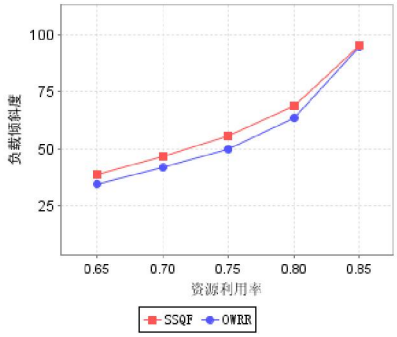


图 5：负载分配算法的负载倾斜度比较

4.2 副本放置

当负载的波动较为剧烈时，单纯的改变负载的分配难以实现本节的整体目标，即在保障各个租户性能指标的前提下，最小化资源的消耗。进一步实现该目标需要将副本放置的动态调整纳入考虑范围。具体的，同时考虑负载分配和副本放置两种性能管理手段，数据处理层的性能管理的目标可以形式的定义为：

$$\begin{aligned} &\min f(\boldsymbol{\lambda}, \mathbf{A}) = \sum_{j=1}^{M} k_j \\ &\min g(\boldsymbol{\lambda}, \mathbf{A}) = \left|\mathbf{A} - \mathbf{A}_0\right| \end{aligned} \tag{1}$$

其中的限定条件为：

$$A_{i,j} \in \{0,1\} \tag{2}$$

$$\forall i \in [1,N] \quad \sum_{j=1}^{M} A_{i,j} \in [R_{\min}^{\mathrm{T}}, R_{\max}^{\mathrm{T}}] \tag{3}$$

$$\forall j \in [1,M] \quad k_j = \begin{cases} 1 & \text{if } \sum_{i=1}^{N} A_{i,j} > 0 \\ 0 & \text{if } \sum_{i=1}^{N} A_{i,j} = 0 \end{cases} \tag{4}$$

$$\forall i \in [1,N], j \in [1,M] \quad \lambda_{i,j}^{\mathrm{R}} = 0 \quad \text{if } A_{i,j} = 0 \tag{5}$$

$$\forall i \in [1,N] \quad \sum_{j=1}^{M} \lambda_{i,j}^{R} = \Lambda_i^R \qquad (6)$$

$$\forall i \in [1,N], j \in [1,M] \quad 0 \le \lambda_{i,j}^{R} \le \Lambda_j^R \qquad (7)$$

$$\forall i \in [1,N], j \in [1,M] \quad \lambda_{i,j}^{W} = \begin{cases} \Lambda_i^w & \text{if } A_{i,j} = 1 \\ 0 & \text{if } A_{i,j} = 0 \end{cases} \qquad (8)$$

$$\forall j \in [1,M] \quad \frac{1}{1 - \sum_{i=1}^{N} \left(\lambda_{i,j}^{W} D_i^W + \lambda_{i,j}^{R} D_i^R \right)} \le SF_{\max} \qquad (9)$$

约束(3)限定了每个租户数据库副本个数的取值范围。约束(4)限定了只有处于运行状态的节点才能放置数据副本。约束(5~7) 限定了读请求分配的合理取值范围。约束(8) 限定了写请求分配的合理取值范围。最后一个约束(9)限定了每个租户的性能指标要求。

通过将上述问题规约为一个 K-划分问题可以得出该问题是 NP 难的，因此难以得到其最优解。本文采用一种贪心策略来试图找到一个较优解。我们称其为惰性副本管理算法 LRM(Lazy Replica Management)。LRM 由如下几个部分构成：表 7 所示的节点提供算法 DB-NP(Node Provisioning)以及表 8 所示的副本放置算法 DB-RP(Replica Placement)。LRM 在多处利用 WBL 生成算法作为启发，为方便起见，我们将某一节点 s_j 在 WBL 之下的资源利用率记作 U_j^{WBL}。

DB-NP 通过动态调整系统的节点数量来适应总体负载强度的变化。具体的，在缩减节点数量时，DB-NP 采用的策略是优先考虑删除 WBL 之下负载最轻的节点。进一步，删除某些节点可能会使得某些租户数据库的副本数量小于 $R_{\min}^{\mathrm{T}}$，这时不能简单的抛弃这些数据副本，而是要将其迁移到某个保留的节点中。DB-NP 同样利用 WBL 来解决这一问题，即，将这些副本迁移到 WBL 之下负载最轻的节点。

DB-RP 算法用来在负载变动时调整副本放置来适应系统负载中各租户比例的变化。首先，DB-RP 调用 DB-NP 来调整节点总量。随后，DB-RP 开始关注节点之间的负载均衡。从本质上讲，DB-RP 是一种贪心的算法。在每一次选择需要调整的副本时，DB-RP 首先选择能够最大化地缩减负载不平衡程度的副本。

总的来看，LRM 的惰性体现在如下几个方面。首先，通过利用 WBL 生成算法作为启发，LRM 优先考虑使用负载分配来均衡各节点的负载。其次，LRM 为每个租户数据库提供的副本数量具有一定的弹性，在新建副本时，如果相应的租户数据库的副本数量没有超出最大值，则并不立即删除某一原有副本；在删除副本时，如果相应的租户数据库的副本数量没有低于最小值，则并不立即在其他节点上创建一个新的副本。上述惰性特点使得 LRM 在保障数据库节点资源利用率的同时，能够减少副本添加或者缩减的次数，提高系统的稳定性。

表 7：节点提供算法 DB-NP

```
计算周期 T 内的服务需求总量 D。
if ( D/T/M > U_target + U_upperThreshold ){
    int K = 0;
    while( D/T/(M+K) > U_target + U_upperThreshold ){
        K = K + 1
    }
    将 K 个节点加入到节点集 S 中;
}
if ( D/T/(M-1) < U_target - U_lowerThreshold ){
    int K = 0;
    while( D/T/(M-1-K) < U_target - U_lowerThreshold ){
        K = K + 1;
    }
    基于当前负载 Λ 以及放置策略 A，计算 WBL;
    删除节点集 S 中 U_j^WBL 最低的 K 个节点，
                                将该集合记作 SK;
    for(节点 From : SK 中的所有节点){
        for(副本 R : From 上的所有副本){
            设 R 属于租户数据库 X;
            if (X 的当前副本数量 ≤ R_min^T ){
                重新计算 WBL;
                for(节点 To : WBL 之下负载从低到高){
                    if(To 不包含 X 的副本){
                        将 R 迁移到节点 To 中;
                        break;
                    }
                }
            }
        }
    }
}
```

表 8：副本放置算法 DB-RP

```
调用 DB-NP 添加或删除节点;
基于当前负载 Λ 以及放置策略 A，计算 WBL;
while( max_{j∈M}(U_j^WBL) > U_max ){
    找到一组节点 A、B，使得
                deltaUnbalance(A, B, WBL)最大;
    找到需要调整的副本 R_A 和 R_B。
    if(R_A!=null){
        在 B 节点上创建一个副本 R_A';
        if(R_A 所属的租户数据库的副本数量 > R_max^T ){
            从 A 节点上删除副本 R_A;
        }
    }
    if(R_B!=null){
        在 A 节点上创建一个副本 R_B';
        if(R_B 所属的租户数据库的副本数量 > R_max^T ){
```

```
            从 B 节点上删除副本 R_B;
        }
    }
    重新计算 WBL;
}
执行副本的添加、删除或者移动;
```

```
int deltaUnbalance(A, B, WBL){
    int oldUnbalance = abs(U_A^WBL − U_B^WBL);
    int delta = 0;
    for(副本 R_A : A 中的所有副本){
        for(副本 R_B : B 中的所有副本){
            int a = abs([U_A^WBL − U_R_A^WBL] − [U_B^WBL + U_R_A^WBL]);
            int b = abs([U_A^WBL + U_R_B^WBL] − [U_B^WBL − U_R_B^WBL]);
            int c = abs([U_A^WBL + U_R_B^WBL − U_R_A^WBL]
                        − [U_B^WBL + U_R_A^WBL − U_R_B^WBL]);
            int d = oldUnbalance − min(a,b,c);
            if(delta < d){
                delta = d;
            }
        }
    }
    return delta;
}
```

5. 分析及评价

本节分别从业务逻辑层和数据处理层两个方面讨论多租户性能管理机制的效果。实验的目的有如下几个：1) 对 ADRS 带来的开销进行定量分析。2) 验证 ADRS 能否有效地避免资源劫持请求带来的性能下降。3)验证 LRM 能否通过负载分配和副本调整适应负载的动态变化。其中前两项是基于实验验证的，最后一项是基于模拟分析的。

5.1 ADRS 的分析与评价

本文通过将一个针对单租户的基准测试应用转化到多租户模式来构造多租户基准测试应用。我们采用的是 TCP-W 基准测试应用。该应用是一个和 Amazon 类似的网上商店系统。多租户版本的 TPC-W 可以看做一个可以供多个客户按需使用的网上店铺。

TPC-W 提供了一个负载生成器来驱动应用的运行。通过配置该生成器的模拟浏览器(Emulated Browser)的个数可以改变其生成的负载强度。每个模拟浏览器按照预定的规则生成各种 HTTP 请求。在本文的多租户背景下，我们为每个租户运行一个负载生成器，同时驱动多个租户应用的运行。

所有的机器都采用相同的硬件配置，包括 1.7 GHz 的 CPU，512 MB 的内存，40 GB 的硬盘。服务器安装的操作系统都是 Linux 2.6.17。机器间采用 100Mbps 的以太网连接。应用服务器使用的是 Tomcat 6.0。数据库节点使用的是 MySQL 5.0.67。

在实验中，我们采用操作系统默认的调度策略(RR)作为 ADRS 的比较对象。图 6 给出了在 RR 以及 ADRS 调度模式下 TPC-W 的性能随负载变化的趋势，其中负载强度以所有租户的模拟浏览器个数之和作为衡量标准。可以看出，在不存在资源劫持请求的情况下，ADRS 与 RR 与性能较为接近。上述实验表明，ADRS 带来的性能开销是可以接受的。

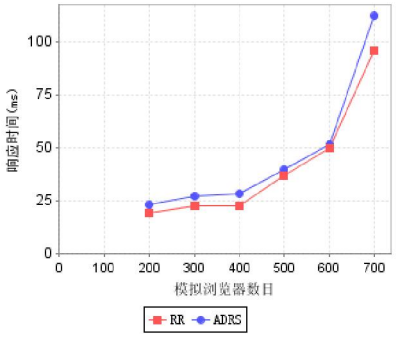
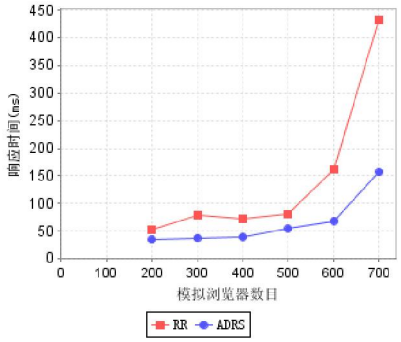
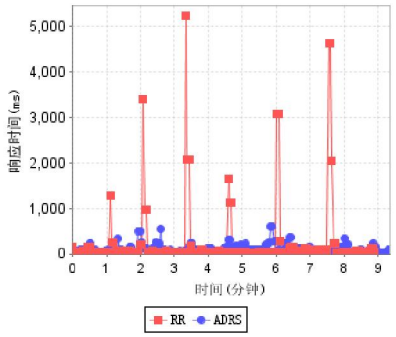


图 6：ADRS 的开销分析



(a)



(b)

图 7：存在资源劫持请求的情况下 ADRS 的表现

图 7 (a)给出了存在资源劫持请求条件下 TPC-W 的性能随负载变化的趋势。其中，资源劫持请求占总请求的 0.1%，每个资源劫持请求的平均服务需求为 10s。可以看出，在存在资源劫持请求的情况下，ADRS 的性能要明显好于 RR。当模拟浏览器数目为 600 时，ADRS 的平均响应时间比 RR 缩短了 40%。图 7 (b)进一步深入

查看了当一个资源劫持请求到来时，ADRS 和 RR 的不同反应，可以看出，在 RR 调度策略之下，每个资源劫持请求的到来都会使得整体响应时间迅速升高，而 ADRS 调度策略能够较好的避免这一现象。上述实验结果说明，ADRS 能够较好的避免资源劫持现象。

5.2 LRM 的分析与评价

我们采用模拟分析的方法来验证 LRM 能否通过负载分配和副本调整适应负载的动态变化。使用模拟器的原因在于 1) 能够较为容易的模拟大规模的集群，绕过实际实验环境的限制。2) 剥离了副本调整执行的复杂性。

具体的，本节中使用的模拟环境的特征如下：

1) 使用离散事件仿真的模式构建，其最小时间粒度设定为 1ms。
2) 模拟器提供了两种负载生成方式，合成负载和真实负载。对于合成负载，本文假定其请求的到来过程为泊松过程；请求的服务需求符合指数分布。对于真实负载，本文采用 WorldCup98 中采集到的真实记录[8]。默认情况下，本文假定请求服务需求的平均值为 50ms。
3) 对于负载分配策略，模拟器实现了 SRR、OWRR 以及 SSQF 三种分配策略。此处 SRR 指的是有状态的轮转负载分配策略，该策略将针对每个租户数据库的查询在该租户的所有副本之间平均分配。

由第四节的讨论的数据处理层性能管理的目标可知，评价 LRM 的指标主要有三个：系统的总体资源利用率，每个租户数据库的性能，以及副本调整的次数。我们首先来单独分析本文提出的负载分配策略的效果，即在总体资源利用率相同，副本放置固定的前提下，比较各个负载分配策略能够达到的性能指标。

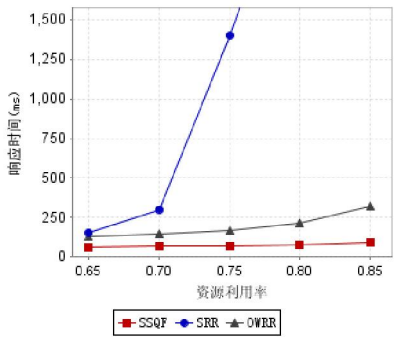


图 8：数据库层各种负载分配算法的性能比较

图 8 给出了模拟实验的结果，其中模拟器的参数配置如下：租户数量为 50，每个租户的副本个数为 3，节点数量为 30。在模拟过程中逐步调整了整体负载强度以反映节点平均资源利用率的变化。从图 8 可以看出，SSQF 和 OWRR 的性能远远好于 SRR。其原因在于 SRR 没有从整体上协调各个租户的负载，导致某些节点负载过重，某些节点负载过轻。而负载过重的节点上处理的请求的性能会大大降低。上述结果表明，与非 WBL 算法相比，实现了 WBL 的负载分配算法具有较好的性能。

对比 SSQF 和 OWRR 可知，SSQF 能够达到比 OWRR 算法更好的性能。其原因在于，除了在宏观上实现负载均衡，SSQF 能够利用系统的实时负载信息实现微观上的较优的分配决策。而 OWRR 只是基于在一定时间周期内得到的负载信息来更新各个节点的权重，没能实现微观上的优化调度。基于上述结果，我们总结出这两种算法各自的适用情形：

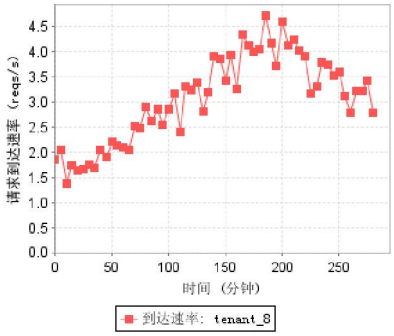
1) 如果系统在分配每个请求时都能够及时的得到各个节点的负载信息，使用 SSQF 算法。
2) 如果系统只能周期性的得到各个节点的负载信息，使用 OWRR 算法。

前面提到，单独使用负载分配难以应对整体负载强度以及负载组成的大幅波动，下面将副本放置的动态调整纳入考虑范围，验证 LRM 能否使得系统在资源利用率较高的前提下保障每个租户数据库的性能指标。

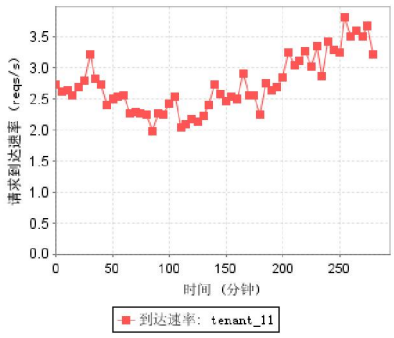
由于副本调整过程会影响系统的性能，因此我们期望副本调整的次数越少越好。副本调整可以细化为新建副本和删除副本，新建副本的代价远远高于删除副本，因而，在后续的试验中，我们使用新建副本次数作为另一个主要评价指标。

我们使用 WorldCup98 的负载来驱动各个租户。具体的，WorldCup98 中收集了该网站 88 天运行期间的负载。我们从中选取第 10 天至第 34 天共 25 天的负载来作为 LRM 的输入。其中每一天的负载对应一个租户，即系统共包含 25 个独立的租户。

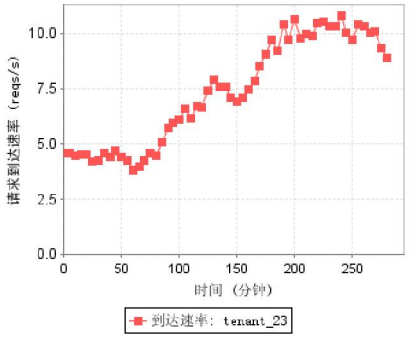
使用上述方式来生成多租户负载的合理性可以从以下几个方面体现：1）WorldCup98 网站的负载随着世界杯举行日期的临近而逐渐增大，因而每天负载的负载强度都有所不同。这个规律能够对应到多租户应用中各个租户负载强度大小不均的情形。2）图 9(a~c)给出了其中 3 天的负载变化过程，可以看出 WorldCup98 网站每天的负载变化规律不尽相同，能够对应各个租户的负载有着独立的变化规律的情形。3）图 9(d)给出了总体负载的图示，可以看出总体的负载也呈现出一定的变化规律，能够对应多租户系统整体负载变化的情形。基于如上几方面原因，我们将 WorldCup98 的负载转化为多租户数据管理服务的负载，并用来驱动本小节的模拟实验。



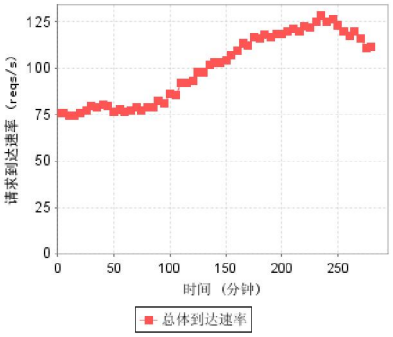
(a)



(b)



(c)

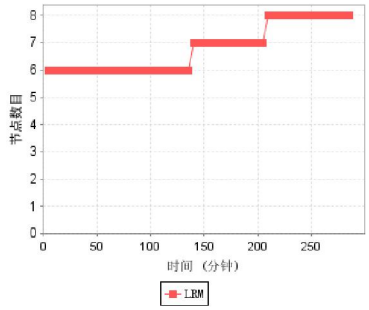


(a)

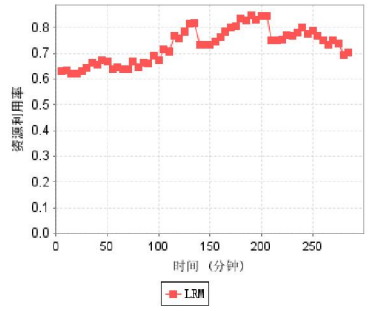
图 9：WorldCup98 负载图示

图 10 给出了 LRM 算法在上述负载下的性能表现，其中负载分配算法使用的是 SSQF。图 10 (a)给出了服务器的总量随时间的变化情况。图 10 (b)给出了节点的平均资源利用率随时间的变化情况。图 10 (c)给出了副本移动的次数随时间的变化情况。图 10(d)给出了所有租户的平均响应时间随时间的变化情况。从图中可以看出，对于上述负载，LRM 能够较好的保障平台整体的资源利用率。动态资源提供和副本调整动作发生的并不频繁，因而能够较好的避免副本调整和节点调整过程对性能的影响。

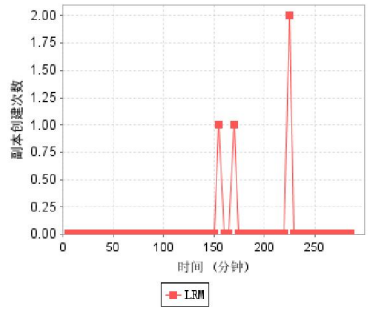
图 10 (e)给出了系统中各个节点的请求到来速率随时间的变化。可以看出，LRM 算法并没有实现各个节点的绝对负载均衡。该行为模式是由 LRM 算法需要在优化负载均衡度的同时最小化副本调整的次数的目标决定的。为此，LRM 算法在均衡负载时惰性策略，只有负载不均衡程度非常严重时才会进行副本调整。例如，考虑由图 10 (c)中第 170 分钟时的一次副本调整。由图 10 (e)可知，此次调整之前分配给节点 9 的请求远远小于其他节点，此次调整在节点 9 上创建了一个新的副本，并将一部分负载分配给该节点。在此次调整完成之后，节点 9 处理的请求量有了显著增加。尽管此时节点 9 的负载仍然小于其他节点，但由于每个节点的资源利用率以及平台整体的资源利用率都处于预先定义的范围内，LRM 算法并没有进一步增加节点 9 维护的副本数量。



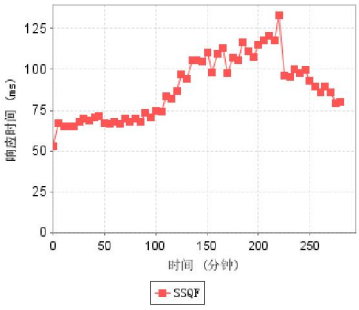
(a)



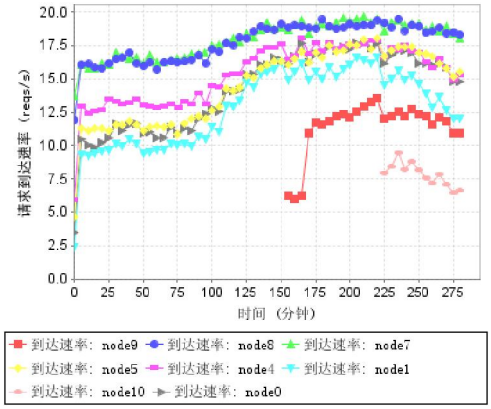
(b)



(c)



(d)



(e)

图 10：LRM 算法在真实负载驱动之下的性能

6. 相关工作

针对 Web 应用服务器端的性能管理问题，前人已经做了大量的工作。鉴于本文主要讨论的性能管理手段包括请求调度、负载均衡以及副本管理，本节主要上述几个方面来分析相关工作，并讨论已有工作在多租户应用中的适用性以及和本文提出的技术之间的异同。更多的相关工作分析请参见[9]。

6.1 请求调度

单服务器请求调度问题的形式化描述和理论分析已经有了较多的工作。以平均响应时间为目标，最短剩余处理时间优先(SRPT)调度策略已经被证明是最优的调度策略。但是 SRPT 要求预先知道每一个请求的服务需求，这在实际的系统中往往是不可行的。

从实际系统设计的角度，对性能影响最大因素在于如何处理并发的请求。事件驱动的模型[10]和多线程模型[11]是两种主流的并发处理模型。

事件驱动模型使用一个接收器线程通过一个 I/O 选择器(selector)监听来自于所有请求的 I/O 事件。一旦某个请求到来或者其处理过程中的 I/O 操作完成时，接收器线程将其分配到某一个工作线程继续处理该请求。当该请求的处理过程中遇到新的 I/O 时，工作线程再次将该请求注册到 selector 之上等待其 I/O 处理完成。事件驱动模式只需要维护少量的工作线程(每个 CPU 一个工作线程)，因而缩减了线程上下文切换的开销。然而请求处理过程中只有遇到 I/O 操作才会导致请求的切换，进一步的控制需要程序员的显式参与，因此增加了编程负担。由于这样的限制，事件驱动的编程模型往往只用在静态 Web 服务器等特定的应用类型的设计之中，不适用于实现通用的应用服务器。

多线程模型为每个请求分配一个线程进行处理。由于线程之间的调度是由操作系统完成的，因而多线程模式下程序员不需要显式考虑并发请求的调度。简单的多线程模型会导致线程频繁的创建和结束，进而对系统的性能造成影响。另外线程数量过大也会增加了线程上下文切换的开销。因此，多线程模型往往与一个线程池搭配使用。当并发的请求数量超出线程池的大小时，后到达的请求会被放置在一个外部请求队列中。相应的，多线程模式下请求的调度又可以细分为外部队列调度和线程池调度。

外部队列调度考虑如何对请求进入线程池的顺序进行调整来优化系统的性能。然而，外部调度往往假定每个 Web 请求的服务需求是已知的，基于先验知识来设计调度策略，但在实际系统中，准确的预测请求的服务需求往往较为困难。线程池内部调度关注的是如何调度正在处理不同请求的多个线程以优化系统性能。然而，针对于 Web 应用，由于线程池的广泛使用，使得每个线程与请求之间不存在一一对应的关系，进而导致线程调度无法直接为请求调度带来好处。

由于线程调度的复杂性，通过线程池内部请求调度来优化 Web 应用性能的工作相对较少。[12]通过 Linux 系统的内核进行修改，实现了一个支持多级处理器共享调度模式的请求调度器 SRQ。由于涉及到系统内核的修改，应用上述两种技术都需要对操作系统、语言运行环境一直到应用中间件等各个层次进行修改，因此实现的难度较大。本文提出的 ADRS 调度算法能够以应用层代码转换的方式实现，与相关工作相比，简化了实现难度，提高了可移植性。

6.2 负载均衡及副本管理

6.2.1 问题空间的工作

从问题空间出发考虑，已有的数据处理层性能管理工作大多是针对单一数据库的。在负载分配方面，[13]提出了冲突感知的查询分配策略，并通过实验表明对于数据库系统，该策略比传统的基于负载指标的分配策略

更加有效。

在存在多个数据库的情况下，如何将各个数据库的多个副本分配到数据库节点集合中，如何在各个副本之间进行负载分配都会对整个系统的性能产生较大的影响。然而对多个数据库的性能管理进行联合考虑还处于刚刚起步的阶段。[14]向这一方向做出了探索，但主要讨论都集中在只有两个数据库的情况。

6.2.2 解决空间的工作

从解决空间出发考虑，由于多个租户数据库共享同一组服务器节点，因此多租户数据库层的性能管理与多个应用的联合资源管理有一定的相似之处。

当同一个托管平台中同时被多个 Web 应用所共享时，平台运营商不但要考虑每个应用的性能管理，还要考虑应用之间的资源竞争问题。特别的，在负载持续增加时往往难以完全满足所有应用的服务需求，需要使用一定的准入控制机制来避免性能的恶化。在支持按需硬件提供的云计算环境中，可以按需扩张或缩减整体资源量，我们称这一背景下的多应用资源管理问题为联合资源提供问题。

[15]考虑了将多应用的联合提供问题。其中不同的应用可以共享同一个节点。每个节点上的资源通过一个按比例共享的调度算法(Proportional Share Scheduler)进行隔离。该工作将问题形式化为一个非凸优化问题，优化的目标是系统的整体收益，并通过一个局部搜索算法给出了该问题的较优解。

上述资源划分和资源联合提供工作为解决多租户数据处理层性能管理问题提供了有价值的借鉴。但数据库负载的如下特点使得已有工作难以直接应用：1）负载无法完全划分，对于多个业务逻辑层副本，负载能够在各个副本之间任意的划分。但对于多个数据库副本，只有只读负载能够在副本之间进行划分，而写负载需要在每个数据库副本上执行。2）资源重新划分的开销较大。针对多租户数据库管理系统，资源的重新划分意味着系统需要在数据库节点之间移动数据副本。数据副本的移动会对系统的整体性能造成一定的影响，因此在调整放置策略时还要考虑如何最小化副本移动的次数。LRM 将上述特点纳入了考虑范围，其惰性特点使得 LRM 在保障数据库节点资源利用率的同时，能够减少副本添加或者缩减的次数，提高系统的稳定性。

7. 结束语

本文分析和总结了多租户应用的新需求和新特点，进而提出了一个特定的多租户应用架构 MDSA。文中探讨了 MDSA 在可定制性、可扩展性、可管理性等方面所遇到的挑战，并着重从运作机制、优化方法等方面研究当中的性能管理问题。

针对多租户业务逻辑层的请求处理过程，文中总结出其中影响性能的关键因素在于服务需求的不确定性以及由此导致的资源劫持现象。文中提出了一种基于延迟的应用级请求调度算法 ADRS 来解决该问题。实验分析表明，通过逐步降低服务需求量较大的请求的优先级，ADRS 能够有效地避免资源劫持。具体的，即使资源劫持请求较少的情况下(0.1%)，与默认的调度方式相比，ADRS 也能够有效缩短系统的平均响应时间，当资源利用率为 80%左右时，缩减的幅度约为 40%。另外，ADRS 算法能够以应用层代码转换的方式实现，与相关工作相比，简化了实现难度，提高了可移植性。

针对多租户数据处理过程，文中总结出其中影响性能的关键因素在于负载强度及组成的动态变化，并有针对性的从负载分配和副本放置两方面展开研究，提出了惰性副本管理算法 LRM。LRM 将固定副本放置前提下的负载均衡问题形式化为一个最优化问题，进而提出了能够实现最优负载均衡的负载分配子算法。当负载的变化使得单独优化负载分配无法保障租户的性能指标时，LRM 通过动态调整系统中的节点数量以及副本在节点间的放置来适应负载变化。模拟实验表明，LRM 能够通过较少的副本调整次数适应负载的动态变化，在维持较高的节点资源利用率的同时保障各租户数据库的性能指标。

参 考 文 献

[1] C. J. Guo, W. Sun, Y. Huang, Z. H. Wang and B. Gao, "A Framework for Native Multi-Tenancy Application Development and Management", *In Proceedings of The 9th IEEE International Conference on E-Commerce Technology and The 4th IEEE International Conference on Enterprise Computing, E-Commerce and E-Services (CEC-EEE'07)*, pp. 551-558, 2007

[2] C. D. Weissman and S. Bobrowski, "The Design of the Force.com Multitenant Iinternet Application Development Platform", *In Proceedings of the 35th SIGMOD International Conference on Management of Data (SIGMOD'09)*, pp. 889-896, 2009

[3] S. Aulbach, T. Grust, D. Jacobs, A. Kemper and J. Rittinger, "Multi-Tenant Databases for Software as a Service: Schema-Mapping Techniques", *In Proceedings of the 34th SIGMOD International Conference on Management of Data (SIGMOD'08)*, pp. 1195-1206, 2008

[4] C. Amza, A. Ch, A. L. Cox, S. Elnikety, R. Gil, K. Rajamani and W. Zwaenepoel, "Specification and Implementation of Dynamic Web Site Benchmarks", *In Proceedings of the 5th IEEE Workshop on Workload Characterization (WWC'02)*, pp. 147-156, 2002

[5] http://www.humanbenchmark.com/tests/reactiontime/index.php, link retrieved on 2010-06-24

[6] G. Soundararajan and C. Amza, "Online Data Migration for Autonomic Provisioning of Databases in Dynamic Content Web Servers", *In Proceedings of the 15th Annual International Conference on Computer Science and Software Engineering (CASCON'05)*, pp. 268-282, 2005

[7] T. Ibaraki and N. Katoh, "Resource Allocation Problems: Algorithmic Approaches", *MIT Press*, 1988

[8] M. Arlitt and T. Jin, "Workload Characterization of the 1998 World Cup Web Site", *HP Laboratories Palo Alto*, 1999
[9] H.L. Lin, Performance Management for Multi-Tenant Web Applications[Ph. D. dissertation]. Beijing: Graduate University of Chinese Academy of Sciences, 2010(in Chinese)
(林海略. 多租户 Web 应用的性能管理关键技术研究[博士学位论文]. 北京：中国科学院研究生院, 2010)
[10] J. R. von Behren, J. Condit and E. A. Brewer, "Why Events Are a Bad Idea (for High-Concurrency Servers)", *In Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS'03)*, pp. 19-24, 2003
[11] J. K. Ousterhout, "Why Threads Are a Bad Idea (for Most Purposes)", *In Keynotes of the USENIX WinterTechnicalConference*, 1996
[12] J. Zhou, C. Zhang, T. Yang and L. Chu, "Request-Aware Scheduling for Busy Internet Services", *In Proceedings of the 26th Conference on Computer Communications (INFOCOM'06)*, *IEEE*, pp. 107-118, 2006
[13] C. Amza, A. L. Cox and W. Zwaenepoel, "Conflict-Aware Scheduling for Dynamic Content Applications", *In Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS'03)*, pp. 328-347, 2003
[14] J. Chen, G. Soundararajan and C. Amza, "Autonomic Provisioning of Backend Databases in Dynamic Content Web Servers", *In Proceedings of the 3rd IEEE International Conference on Autonomic Computing (ICAC'06)*, *IEEE*, pp. 231-242, 2006
[15] D. Ardagna, M. Trubian and L. Zhang, "SLA Based Resource Allocation Policies in Autonomic Environments", *Journal of Parallel and Distributed Computing*, vol. 67, pp. 259-270, 2007

LIN Hailue, born in 1982, Ph.D. candidate. His current research interests include service oriented computing.

Han Yanbo, born in 1962, Ph.D. supervisor, professor. His current research interests include service oriented computing, distributed systems and workflow technologies.

Background

The main research interests of the author include service oriented computing, distributed computing and resource management etc. This paper is primarily supported by the 863 project “High Performance Computer and Grid Service Environment”, which explores new ways to develop, deliver and management of Web-based cooperative applications.

The paper proposes and discusses a specific multi-tenant architecture — MDSA. At the application logic layer of MDSA, due to the variation of request service demands, some requests with a large service demand may affect the performance of the other concurrent requests — a phenomenon which is called resource hijacking. ADRS is proposed to avoid this phenomenon. At the data processing tier of MDSA, to adapt the variation of overall arrival rate and the composition of the workload, LRM is employed to handle replica load balancing and dynamic replica placement. Experiments indicate that the proposed performance management techniques can help to ensure certain level of quality of service (QoS) guarantee for the tenants and to improve overall resource utilization.