第 29 卷 第 7 期
2006 年 7 月
计 算 机 学 报
CHINESE JOURNAL OF COMPUTERS
Vol. 29 No. 7
July 2006

一种应用敏感的 Web 服务请求调度策略

官荷卿[1),2),3)] 张文博[1),2),3)] 魏 峻[1)] 黄 涛[1),2)]

[1)](中国科学院软件研究所软件工程技术研发中心 北京 100080)

[2)](中国科学院软件研究所计算机科学重点实验室 北京 100080)

[3)](中国科学院研究生院 北京 100039)

摘 要 在当前企业级 Web 服务应用中,性能问题一直是人们关注的重点.然而作为支撑 Web 服务应用的主流平台,Web 应用服务器对请求的调度仍然是传统的先来先服务策略(FCFS).这种策略无法区分请求的重要性,降低了关键请求的性能.以往的研究较少从应用的性能需求出发考虑服务器的请求调度机制,影响了服务器性能保障的效果.对此,提出了应用敏感的 Web 服务请求调度策略(AWSRS),使用应用获益来评估服务器为应用提供的性能保障效果.服务器将请求按照应用的需求进行分类,并按照应用获益最大化的目标为不同类型的请求分配资源.实验表明 AWSRS 策略能够有效提高关键请求的性能.

关键词 Web 服务;请求调度;应用感知策略;Web 应用服务器;服务质量

中图法分类号 TP301

An Application-Aware Web Service Requests Scheduling Strategy

GUAN He-Qing[1),2),3)] ZHANG Wen-Bo[1),2),3)] WEI Jun[1)] HUANG Tao[1),2)]

[1)](*Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences, Beijing* 100080)

[2)](*Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing* 100080)

[3)](*Graduate University of Chinese Academy of Sciences, Beijing* 100039)

Abstract For commercial Web-services based applications (WSBAs), performance problem is a major concern. As the main platform for WSBAs, Web application servers (WASs) typically adopt the first-come first-served (FCFS) scheduling strategy. FCFS strategy does not distinguish critical requests from non-critical ones, and therefore the performance of critical ones might be reduced. Most of the existing research seldom takes into account the requirements from applications, which impairs the performance guarantee from WASs. This paper proposes a novel application-aware Web service requests scheduling (AWSRS) strategy for WSBAs, introduces the concept of "application benefit" to evaluate the effect of WAS in improving the performance for applications. Based on that, the strategy is able to classify requests and intelligently assign corresponding resources for them, in order to maximize the total application benefits. Later experimental results indicate that the strategy effectively improves the performance of critical requests.

Keywords Web service; request scheduling; application-aware strategy; Web application server; QoS

收稿日期:2006-02-17;**修改稿收到日期**:2006-05-17.**本课题得到国家自然科学基金**(60573126)、**国家"九七三"重点基础研究发展规划项目基金**(2002CB312005)**资助**.**官荷卿**,**男**,1980 **年生**,**博士研究生**,**主要研究方向为网络分布式计算**. E-mail: ghq@otcaix.iscas.ac.cn. **张文博**,**男**,1976 **年生**,**博士研究生**,**主要研究方向为网络分布式计算**. **魏 峻**,**男**,1970 **年生**,**博士**,**研究员**,CCF **高级会员**,**主要研究领域为面向服务计算、中间件、移动计算和软件工程**. **黄 涛**,**男**,1965 **年生**,**博士**,**研究员**,CCF **高级会员**,**主要研究领域为软件工程、网络分布式计算**.

1 引 言

Web 服务已经成为基于标准技术构造分布式计算平台的基础，它能够支持应用系统以松散耦合的方式实现集成与协作，并通过降低应用连接的复杂性来减少应用维护和更新的代价，是目前 Web 应用环境中一种最有前景的解决方案[1]. 与此同时，Web 应用服务器(WAS)已经成为支持 Web 服务的最重要的基础软件，是当前构建企业级 Web 服务应用系统的主流技术平台，各大服务器厂商纷纷推出各自的基于应用服务器的 Web 服务解决方案.

在当前企业级 Web 服务应用(Web-services based application)中，性能问题一直是人们关注的重点[2]. Web 服务应用的性能至少受到两方面因素的影响：(1)它所使用的 Web 服务；(2)应用服务器为之提供的运行环境. 所以，解决 Web 服务应用的性能问题也应该从这两方面因素来考虑. 当前研究大多集中在第一个方面，包括 Web 服务缓存技术、高效的 Web 服务消息处理技术、Web 服务选择技术等. 其中服务选择方面的研究更是引起了众多的关注，这方面研究的目的在于为某个 Web 服务应用选择合适的 Web 服务，从而使得该应用的整体性能表现能够满足用户的需求. 比如 Zeng 等人提出了 Web 服务的 QoS 模型，并在此基础上给出了使用全局规划(global planning)的方法来计算满足 QoS 需求的 Web 服务组合执行路径[3]；Menascé 也提出了 QoS 感知的 Web 服务软件组件模型(QoS Aware Software Component Model for Web Service)，并使用 WSFG(Web Service Flow Graph)来估算工作流的响应时间[4]. 然而这些研究工作都是研究 Web 服务应用的静态执行特性，而忽略了 Web 服务应用的运行支撑环境对其性能的影响. 事实上，如果 Web 服务运行支撑环境不具备任何 QoS 控制，那么这种通过静态方式获得的执行路径的执行结果也可能无法达到用户的要求. 例如，类似一般的 Web 站点，Web 服务应用也会面临过载的威胁[5]，在这种情况下需要为 Web 服务应用分配合适的资源，为其提供必要的性能保障.

然而，作为支撑 Web 服务应用的主流平台，Web 应用服务器对请求的调度仍然是传统的先来先服务策略(FCFS)，它无法区分请求的重要性. 当服务器处于重负载时，次要请求与关键请求争夺有限的资源，从而严重降低了关键请求的性能. 服务器在系统重负载的情况下，应当优先保证关键请求的处理，从而集中有限资源为关键请求提供更好质量的服务. 为此，本文提出了一种应用敏感的 Web 服务请求调度策略(Application-Aware Web Service Request Scheduling Strategy，AWSRS). 该调度策略提出了“应用获益”的概念，即 Web 服务应用完成一定的用户请求，则认为应用得到了一定获益. AWSRS 以应用获益最大化作为优化的目标，使用运行时探测到的环境信息，驱动服务器为不同类型的 Web 服务请求实现系统资源的分配，使调度结果能够更准确地满足用户的需求. 本文中的“应用”和“请求”，如未加特别说明，分别指的是“Web 服务应用”和“Web 服务请求”.

本文第 2 节简要介绍系统模型；第 3 节阐述 AWSRS 策略的基本思想、关键技术和算法；第 4 节是实验结果及分析，并与其它调度策略进行比较；第 5 节是对相关工作的介绍和比较；最后总结本文.

2 系统模型

Web 服务应用的性能可以通过不同的指标来衡量，但是时间和吞吐量是其中最为重要的两个指标，因此本文着重于衡量 Web 服务应用的响应时间和吞吐量. 对于一般的 Web 服务请求而言，其处理过程如下：(1)Web 服务应用接收来自用户的请求；(2)Web 服务应用把请求转发给后端的 Web 服务；(3)Web 服务处理完毕之后，把结果返回给 Web 服务应用；(4)Web 服务应用把最终结果返回给用户. 从这个流程可以看出，Web 服务应用的性能受到两方面因素的影响，一是后端的 Web 服务的性能，另一是应用服务器所使用的调度策略.

图 1 为本文使用的系统模型，在 Web 服务应用和 Web 服务之间，我们引入了服务实例池(Service Pool)[6]对它们进行解耦. 因此，Web 服务请求的调

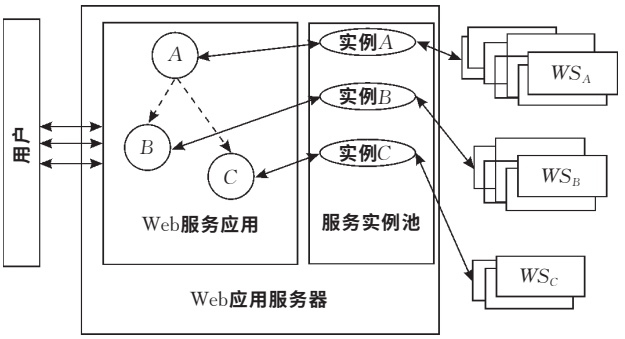


图 1 系统模型

度被分为两个层次，第 1 层（称为服务器调度），位于应用服务器内部，为 Web 服务应用进行系统资源的分配，而第 2 层（称为实例池调度）则位于单个服务实例池，为 Web 服务应用选择合适的后端 Web 服务，最终使请求的处理结果能够更准确地满足用户的需求. 本文我们重点研究服务器调度的问题，而对实例池调度可以借鉴 Web 服务选择的研究成果.

3　应用敏感的请求调度策略 AWSRS

目前应用服务器大多采用一种先到先服务（FCFS）的调度策略，这种方式公平性好，但是缺乏对重要请求的优先调度，而文献[7,8]提出了一种根据用户优先级进行请求分类（Priority Layered，PL）的调度策略，其主要特点是按照用户重要性级别的不同，将请求分成若干个请求队列，重要用户的请求将会得到优先的调度处理. 这种调度策略保障了不同用户请求之间的优先级，重要用户的请求总是能够得到优先的调度处理. 但是这种调度方式由于分类标准的单一，使得其不能应对更为灵活的性能需求，比如对同等重要用户发起的不同操作类型的请求则无法实现区别对待.

针对上述解决方案的不足，本文提出了一种应用敏感的 Web 服务请求调度策略 AWSRS. 该调度策略可以对请求进行更为灵活的分类，并使用应用获益作为优化目标来调整服务器的资源，从而保障服务器能够按照应用的意愿进行调度的优化.

3.1　应用获益的原理

3.1.1　性能与应用需求的分析

已有研究成果[9,10]表明，应用的用户对请求响应时间的满意程度并非一成不变的，在客观的响应时间和用户主观的感受之间存在某种关联：

（1）用户对请求响应时间的满意程度与其期望值有密切关系. 例如，一个重要的用户通常希望自己得到更快的响应时间，而一个匿名的用户则对响应时间的要求不会很苛刻.

（2）随着用户与 Web 应用系统之间的交互时间的增加，其对请求响应时间的容忍程度会逐渐变差. 在交互过程的后期，通常会进行一些关键的操作，如提交订单等，重点保障这些操作有助于使用户有耐心完成整个交互.

以上关联使我们对 Web 服务应用的性能需求有如下认识：

（1）用户的请求如果不能在一定的时间内完成，则不会再有意义.

（2）位于一个交互过程后期的请求通常需要得到更好的响应时间保障.

（3）服务器需要区别请求重要性，保证较重要的请求得到优先处理，请求的重要度由应用来指定.

因此，Web 服务请求调度策略应该充分考虑这些性能需求. 为此，应用需要对用户的请求进行分类，声明其相应的重要程度，服务器将根据这些声明在请求调度时做出合理的选择. 根据以上分析，可从两个维度对应用请求进行分类（图 2）.

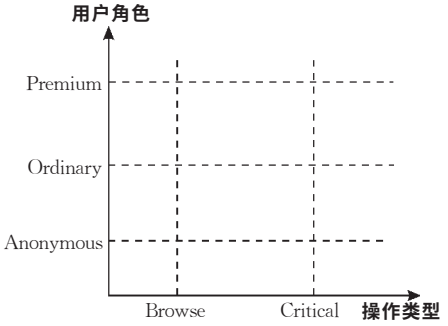


图 2　请求分类视图

（1）按照请求的发起用户的角色重要性分类：

重要用户（Premium）. 如电子商务网站的正式会员、VIP 用户等.

普通用户（Ordinary）. 相对重要性稍差的用户，如普通的注册用户等.

匿名用户（Anonymous）. 以浏览请求为主的非注册用户.

（2）按照位于业务流程不同阶段的请求操作类型分类：

关键（Critical）请求. 通常是业务中的关键部分，如提交订单等.

浏览（Browse）请求. 除关键请求外的其它类型请求，如浏览页面、查询信息等.

图 2 中的坐标方向反映了不同请求类型的相对重要性. 高等级用户的请求、关键性请求将得到更好的性能保障. 本文将研究如何根据应用对请求类型的声明来执行请求调度，以更好地满足用户的需求.

3.1.2　应用获益的概念

Web 服务应用的每个用户都期望自己能够得到最优的服务，但是由于服务器的资源有限，服务器必须权衡如何最大限度地满足不同用户的需求. 服务器每完成应用的一个请求，都可以看作用户从服务器得到了一定的好处，服务器应当使一个重要客户的请求比一个普通客户的请求得到更多的好处.

这种请求期望得到的好处由 Web 服务应用来指定，反映了应用对性能的需求. 我们使用“应用获益”的概念对其描述. 下面给出相关的定义.

用户向应用发出的请求由一组在服务器上执行的操作组成，这些操作的用户角色和操作关键性的组合代表了该请求的请求类型(Q_i). 用户请求的一次处理过程称为一个请求实例(q_j). $q_j \in Q_i$ 表示 q_j 属于类型 Q_i. 设 $\Omega=\{Q_1, Q_2, \cdots, Q_n\}$ 是应用服务器中所有请求类型组成的集合，$\varpi=\{q_1, q_2, \cdots, q_m\}$ 是所有请求实例组成的集合. 而部署在服务器上的应用 app 包含的所有请求类型组成的集合，记为 Ω_{app}，显然 $\Omega_{\text{app}} \subset \Omega$.

定义 1. 请求实例的获益. 指一个请求实例 $q_j \in Q_i$ 在服务器上处理完之后，用户对 q_j 的处理结果的评价，表示为 $b_j=f_i(q_j)$，其中，

b_j 称为 q_j 的获益，为一个实数. b_j 越大，说明用户对 q_j 的处理结果越满意. 若 b_j 为 0，则表示 q_j 的处理结果对用户来说已经没有意义.

f_i 称为请求的获益方案，是 $\varpi \rightarrow R$ 的映射(R 为实数集合)，表示请求实例获益的计算方式. 同一请求类型 Q_i 的请求实例使用相同的 f_i.

定义 2. 请求类型的获益. 在一个时间段 T 内，属于请求类型 Q_i 的所有请求实例的获益的总和 $B_i=\sum_j b_j(q_j \in Q_i)$，称之为请求类型 Q_i 在 T 内的获益.

定义 3. 应用获益. 在时间段 T 内，应用 app 的获益为该应用所包含的所有请求类型的获益的总和，即 $B_{\text{app}}=\sum_i B_i(Q_i \in \Omega_{\text{app}})$.

应用获益 B_{app} 衡量了用户可感知的、对请求处理结果的满意程度，反映了请求得到性能保障的重要程度. 对于一个 Web 服务应用而言，用户关心的性能指标(如吞吐量、平均响应时间等)都和请求的实际响应时间相关，因此，本文主要从请求实际响应时间(Real Response Time，RRT 或者 Tr)出发来考虑应用获益，简单地以请求实例 q_j 的 RRT_j 来衡量 q_j 的处理情况，从而获益方案可以简化为是 $R \rightarrow R$ 的映射，即 $b_j=f_i(RRT_j)$. 由 3.1.1 小节中的需求分析可知，用户对请求响应时间存在着阈值 $Timeout_j$ 的要求，使得

(1) 若 q_j 的响应时间 $RRT_j \leq Timeout_j$，则 $b_j=f_i(RRT_j)>0$；

(2) 若 $RRT_j > Timeout_j$，则 $b_j=0$.

3.2 AWSRS 调度策略

AWSRS 的原理如图 3 所示. 调度过程中的主要参与组件包括：

请求分发器(Classifier). 它负责接收来自用户的请求，对请求进行分类，并将其放入相应的请求队列. 每个请求队列存放一种类型的请求，每个请求队列内部都采取先入先出的方式，这是为了保证同类请求之间的调度是公平的.

调度器(Scheduler). 它是调度过程中的核心组件，并不负责调整队列中请求的执行顺序，而是通过调整各个请求队列的接受率来影响各类请求的执行情况. 接受率是请求队列所对应的系统资源在服务器的所有资源总量中所占的比例. 接受率越大，则说明服务器处理该类请求的能力越高. 调度器需参考获益评估器的结果，按照获益最大化的原则进行调度.

获益评估器(Benefit Evaluator). 获益评估器对请求到达情况、请求处理时间等进行监测，根据请求完成情况定期地计算应用获益，将应用获益评估值反馈给调度器.

请求队列监控器(Request Queue Monitor). 它

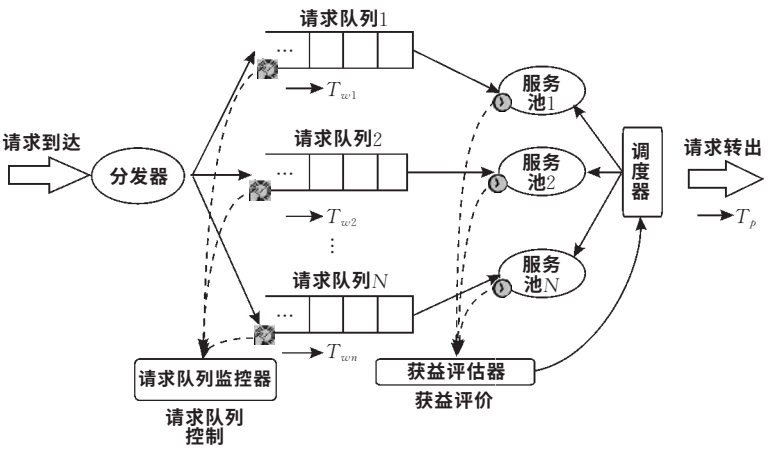


图 3 AWSRS 调度策略原理图

监测各个请求队列的状态，实现请求的准入控制，根据请求的到达和完成情况来限制请求队列的长度，这在服务器过载时可以防止已经或可能超时的请求继续等待.

与 FCFS 调度策略相比，PL 和 AWSRS 策略都可以解决重要客户的请求优先处理问题，但是 AWSRS 比PL 具有的优势体现在：AWSRS 的请求分类规则更灵活，应用不但可以指定用户的等级，还可以区分同一等级用户的不同类型请求；应用获益的动态评估考虑了运行时环境的状态，使请求调度更接近于实际需求，提高了调度的合理性和准确性. 以下对 AWSRS 的关键技术进行进一步阐述.

3.2.1　请求获益描述

获益驱动的调度策略要求应用开发者或部署者描述应用请求的类型，为它们设置各自期望的获益. 我们使用特定的应用部署描述符 BenefitSchema.xml 来表达每种请求类别和获益相关的描述，它的定义内容如图 4. 其中：

UserLevel 表示应用的用户重要性级别，可取值包括 *Premium*，*Ordinary* 和 *Anonymous*.

QType 表示请求的操作类型，可取值包括 *Critical*，*Browse*.

QBScale 表示该类请求对获益的期望级别，体现了不同类别请求之间相对的重要性，取值为非负整数. *QBScale* 为 0 时，表示该类请求不使用 AWSRS 策略，而只能得到 FCFS 策略的保障；*QBScale*＞0 时，取值越大表示请求越重要.

Timeout 声明请求的最大容忍响应时间，只有在该时间内完成的请求才能有获益，否则获益值为 0.

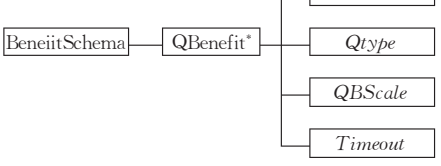


图 4　获益方案描述符

3.2.2　请求队列的预期获益

由于请求到达和处理的不确定性，如何为不同的请求队列分配系统资源才能获得最大的应用获益是调度策略进行性能优化的关键问题. 本文使用在一定的采样间隔内各个请求队列中请求的到达速率和处理速率等统计信息预测调度资源分配后的同一请求类型的可能获益. 由于一个队列只处理一种类型的请求，因此该获益也称为请求队列的预期获益.

在一个采样间隔 T 内，请求队列 i 的预期获益可以分为两部分：$B_i'=B_i'|_{\text{arrive}}+B_i'|_{\text{left}}$，$B_i'|_{\text{arrive}}$ 表示 T 内新到达的请求被处理后产生的获益，$B_i'|_{\text{left}}$ 表示 T 内请求队列中残留请求被处理后产生的获益.

假设请求队列 i 的请求到达速率为 λ_{i1}，处理速率为 λ_{i2}，采样初始时刻任务队列的长度为 L_i，则对任意的 $t\in[0,T]$，t 时刻到来的请求 q_j 预期的响应时间为 $RRT_j'=[L_i+(\lambda_{i1}-\lambda_{i2})\times t]/\lambda_{i2}+T_{pi}$，也可表示为 $RRT_j'=g(t)=(\lambda_{i1}/\lambda_{i2}-1)\times t+(L_i/\lambda_{i2}+T_{pi})$，根据获益的定义，我们可以得到

$$B_i'|_{\text{arrive}}=\lambda_{i1}\int_0^T f_i(g(t))\mathrm{d}t.$$

另一方面，请求队列中残留请求 q_j 的预期响应时间为 $RRT_j'=T_{pi}+position/\lambda_{i2}$，其中 $position$ 为 q_j 在请求队列中的位置. 于是可以得到

$$B_i'|_{\text{left}}=\sum_{position=1}^{\lfloor L_i/(\lambda_{i2}\times T_{pi})\rfloor}\lambda_{i2}\times T_{pi}\times f_i(T_{pi}+position/\lambda_{i2}).$$

上述公式中，参数 L_i 直接通过监测系统状态得到，而 λ_{i1}，T_{pi} 将使用上一次采样间隔中统计出的请求平均到达速率 $\bar{\lambda}_{i1}$ 和请求平均处理时间 $\bar{T}_{pi}$，λ_{i2} 则需要根据请求队列的接受率计算而得. 由于本文使用的可调系统资源为服务器的 Web 服务实例的个数，因此假设调度器分配给该请求队列的服务实例数量为 m，那么 $\lambda_{i2}=m/T_{pi}$.

在每个调度周期开始前，获益评估器计算出请求队列的预期获益 B_i'，同时计算出每个队列在获得 Δm 个服务实例后的预期获益 B_{i+}' 和失去 Δm 个服务实例后的预期获益 B_{i-}'. 我们把 $B_i'|_{\text{win}}=B_{i+}'-B_i'$ 称为请求队列的预期获益增益，而 $B_i'|_{\text{loss}}=B_i'-B_{i-}'$ 则称为请求队列的预期获益损失. 调度器将根据预期获益增益和损失将来分配系统资源，以达到应用获益最大化的目标.

3.2.3　获益驱动的调度优化算法

获益驱动的调度策略基于启发式爬山法来实现调度优化算法. 优化的目标是搜索应用最大的请求获益，即寻找一种最优的 Web 服务实例分配方案，使得处理应用的请求获得的获益最大，算法的启发式信息则是各个请求队列的预期获益.

AWSRS 策略周期性地对 Web 服务实例的分配进行优化. 调度周期 T 对分配的效果具有影响. T 过短可能造成统计数据失真，且使得调度器过于频繁地调度；而采样周期过长则难以保证系统始终位于较优的状态. 为此，本文同时采取如下两个措施：

(1) 为了防止 T 过短，事先指定算法的最小调度周期 $T_{\min}$，确保调度器不会过于频繁地调度，浪费

系统资源. 同时调度周期始终设置为一个相对较小的值,从而尽可能确保调度器能及时应对负载的变化. 因为本文的算法是启发式爬山法,所以如果系统的负载在一定时间内处于稳定状态,则通过若干次调整之后,系统将逐渐接近最优状态;而如果系统出现了临时的负载突变,那么为应对这种突变而进行的资源配置,将在接下来的几次调度中恢复原状.

(2) 指定算法的最大调度周期 $T_{\max}$,确保调度器能对负载的变化及时做出响应. 同时在每个调度周期结束后根据调度的实际情况小范围地调整 T 的大小(调整的步长为 $\Delta T \leq T_{\min}$).

假设 i 和 j 表示两个不同的请求队列,在一个调度周期 T 中发生了一次从 i 到 j 的分配,即减少队列 i 的服务实例数,同时增加队列 j 的服务实例数,用 $i \rightarrow j$ 表示. 算法在执行过程中将对这种分配进行记录. 如果调度周期内未发生分配,则称产生一条空分配. T 调整的原则如下:

(1) 若在连续的调度周期 T 中产生了两条空分配,则增大调度周期 T,

$$T := (T+\Delta T < T_{\max})\ ?(T+\Delta T) : T_{\max};$$

(2) 若在某个周期 T 中产生了一条分配,则缩短调度周期,

$$T := (T-\Delta T > T_{\min})\ ?(T-\Delta T) : T_{\min}.$$

同理,算法还指定最大调度资源 $\Delta m_{\max}$ 和最小的调度资源 $\Delta m_{\min}$ 来限制调度的精度,提高优化的效率. Δm 调整的原则如下:

(1) 若在连续的调度周期中产生了两条相同的分配,则增大 Δm,

$$\Delta m := (\Delta m+\Delta m_{\min} < \Delta m_{\max})\ ?(\Delta m+\Delta m_{\min}) : \Delta m_{\max};$$

(2) 若在连续的调度周期中产生了两条相互抵消的分配,如 $i \rightarrow j$, $j \rightarrow i$,则减小 Δm,

$$\Delta m := (\Delta m-\Delta m_{\min} > \Delta m_{\min})\ ?(\Delta m-\Delta m_{\min}) : \Delta m_{\min}.$$

算法使用向量 $(\boldsymbol{BV}_{\text{win}})$ 和 $(\boldsymbol{BV}_{\text{loss}})$ 分别存放应用 app 的每个请求队列的预期获益增益和预期获益损失,其中 $(\boldsymbol{BV}_{\text{win}})_i$ 和 $(\boldsymbol{BV}_{\text{loss}})_i$ 表示对应向量的第 i 个变量. 在每个调度周期开始时,算法执行的具体步骤如下:

```
1. for(each queue i){
2.     (BV_win)_i = ComputeBwin(i);
       (BV_loss)_i = ComputeBloss(i);
                    //计算每个请求队列的预期获益增益和损失
3. }
4. if(exists i,j, such that (i !=j && (BV_win)_i == max(BV_win) && ((BV_loss)_j) !=0 && (BV_loss)_j == min(BV_loss) && (BV_win)_i > (BV_loss)_j))
                    //如果存在可行的分配 j→i
5. then {
6. m_i += Δm; m_j -= Δm; //一次 j→i 的分配
7. record(j→i); //记录分配
8. T=(T-ΔT>T_min) ?(T-ΔT): T_min;
9. if(loop(sequential record)) then
   Δm=(Δm-Δm_min>Δm_min) ?(Δm-Δm_min): Δm_min;
                    //如果出现了相互抵消的分配
10. else if(iteration(sequential record)) then
   Δm=(Δm+Δm_min<Δm_max) ?(Δm+Δm_min): Δm_max;
                    //如果出现了两条相同的分配
11. }else{
12.   if(exist two empty records)
                    //如果存在两条空分配记录
13.   then {T=(T+ΔT<T_max) ?(T+ΔT): T_max;}
14. }
```

由算法的描述可知,算法的每次调度都会保证应用的预期获益非减,如果请求的到达速率和处理时间稳定,则系统会逐渐调整到最优的状态,并一直保持下去,但是由于实际运行环境不可能如此理想,所以算法会不断地根据系统的当前状态调整自己的调度行为,及时地改变资源的分配,使各类请求的综合处理情况始终保持在较优的状态,从而按照应用的意愿合理地实现资源的分配. 此外,该算法对调度周期 T 和调度资源 Δm 的动态调整,也进一步提高了算法的有效性.

我们用 n 表示全部请求的种类数,那么,从上述算法的描述中,可以看出 1～3 步为一重循环,其时间复杂度为 $O(n)$. 而第 4 步是在 n 种请求类型中寻找可行的资源分配,需要遍历所有的请求的获益收益和获益损失,因此其时间复杂度为 $O(n)$,而其它各步均不存在循环. 所以,总的来说,上述算法的时间复杂度为 $O(n)$.

3.2.4 请求队列约束

请求队列约束是在服务器过载时的一种请求准入措施. 请求队列监控器根据当前请求的到达速率和处理速率,判断在调度周期内已不能处理更多的请求时,即会启动该措施,以尽快地把无法满足的请求直接做失败处理.

对于不同类型的请求,请求队列监控器的行为有所不同. 对于 $QBScale$ 为 0 的请求队列,请求队列监控器根据上一采样周期 T 内的平均请求处理时间 $\overline{T}_{pi}$、当前的 Web 服务实例数 m_i 和当前请求队列长度 L_i 来估算队列中请求的预期响应时间 $T_{\text{expected}} = L_i \times \overline{T}_{pi}/m_i + \overline{T}_{pi}$,如果 T_{expected} 大于该类请求的

Timeout，则直接拒绝对该请求的处理. 对于*QBScale*大于 0 的请求，只有当其等待时间 Tw 实际超过其 *Timeout* 时才选择放弃对该请求的处理.

4　实验数据及分析

AWSRS 调度策略在中国科学院软件研究所的 Web 应用服务器产品 OnceAS[11] 中进行了原型实现，本小节将通过实验来对其进行验证.

4.1　实验设计

实验采取一个在线书店 Web 服务应用作为模拟场景. 用户可以进行两类操作，一类是浏览书目信息，另一类是选择中意的图书，并完成网上订购. 应用支持的用户类型有三种：正式会员、普通会员和游客，分别使用 P(Premium)，O(Ordinary)和 A(Anonymous)来表示. 浏览和定购两类操作分别属于浏览(Browse)请求和关键(Critical)请求类型，分别使用 B 和 C 来表示. 由于游客不能进行定购的操作，所以应用的请求类型共有 5 种，分别使用 PC，PB，OC，OB 和 AB 来表示，如 PC 表示正式会员发出的订购图书的请求. 应用在部署描述符中声明关键的请求，而用户的重要级别则可以通过请求的登录信息获得，因此应用的每个请求都可以对应到上述的 5 种请求类别之一，以下给出了获益描述符的配置例子. 为简单起见，我们把各类请求的获益属性归纳为表 1 所示的情形.

表 1　各类请求的获益属性

	Timeout(s)	*QBScale*
PC	1	4
PB	2	2
OC	2	2
OB	4	1
AB	4	0

实验环境包括 5 台 PC 机. 其中一台 PC 机专门作为应用服务器，其配置为 Dell PowerEdge 2600，Dual CPU 3.04GHz，2GB 内存. 一台 PC 机为数据库服务器，其配置为 IBM Xseries 235，Single CPU 2.66GHz，2GB 内存，Oracle8.1.6 数据库. 另外三台 PC 机为用户测试机，其配置为 Dell OPTIPLEX GX260，CPU 2.66GHz，512MB 内存，分别对应三种不同的用户(P，O 和 G)，按照相同的负载向服务器发起 Web 服务请求，并记录请求的完成情况. 每台用户测试机发出订购请求 C 和浏览请求 B 的比例为1∶2，使用 λ 表示请求的到达速率，则各类请求的到达速率(PC，PB，OC，OB 和 AB)分别为 λ，2λ，λ，2λ 和 2λ.

实验的初始配置：服务器端全部 Web 服务实例数为 500，$\Delta m_{\min}=2$，$T_{\min}=5$s，$T_{\max}=30$s. 实验阶段分为初始期 $T_{\text{warm}}=60$s，测试期 $T_{\text{steady}}=3000$s 和结束期 $T_{\text{down}}=30$s. 在 T_{warm} 时间段内测试机只发送请求不记录测试结果，只在 T_{steady} 内统计请求处理结果.

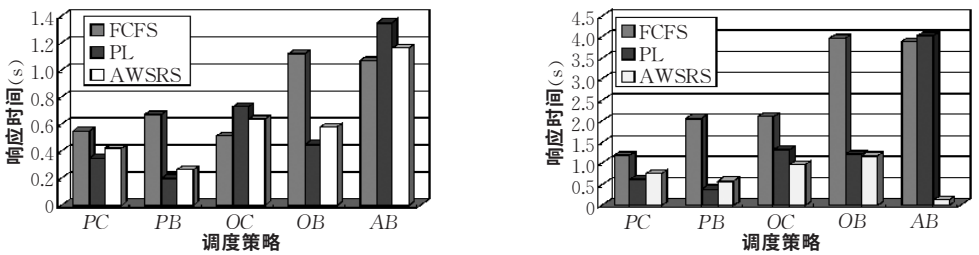
为了比较 AWSRS 策略的调度效果，我们还实现了 FCFS 和 PL 策略，我们在实验中还对请求进行了超时的处理，即当一个请求到达其 *Timeout* 时，服务器将拒绝对该请求的处理.

4.2　测试结果与分析

我们使用两组负载来进行测试，分别代表服务器正常服务和过载两种情况，其中 λ 分别为 300reqs/s 和 600reqs/s，图 5 的(a)和(b)分别表示了这两种负载下，各种请求在不同的调度策略下的平均响应时间.

当服务器正常服务时，服务器都可以正常完成所有请求，响应时间都不超过各自的 *Timeout*. FCFS 调度下 PC 和 PB 请求的响应时间明显比其它两种调度要高，主要原因就是在因为其没有区分重要请求的能力，同样的原因，AB 请求的响应时间则比其它两种调度要低. 在服务器过载时，三种调度策略对不同类型请求的响应时间影响较大，具体分析如下：

(1) FCFS 调度下的各种请求响应时间大幅上



(a) 正常情况(λ=300reqs/s)　　(b) 过载情况(λ=600reqs/s)

图 5　不同调度策略下各类请求的响应时间

升，都接近或超过请求的 *Timeout*，大量的请求因为超时而被服务器拒绝处理.

(2) PL 和 AWSRS 两种调度策略都可以保障除 *AB* 请求外的其它类型请求，区别在于：PL 调度策略下 *PC* 和 *PB* 两类请求的响应时间较 AWSRS 低，而 *OC* 和 *OB* 两类请求的响应时间则比 AWSRS 调度要高，尤其是 *OC* 请求更为明显. 此现象说明 PL 调度对请求的优先级分类使得重要用户的请求得到了更好的调度，但是完全缺乏对同等重要级别用户的不同类型请求进行服务区分的能力. 而 AWSRS 调度则同时兼顾了用户的重要性等级以及同类用户中的关键请求.

(3) PL 调度下 *AB* 请求响应时间达到了 *Timeout*，由于服务器过载时，重要级别最低的 *AB* 请求几乎无法得到处理，因而基本都被服务器进行了超时处理，而 AWSRS 由于对 *AB* 的响应时间进行了预测并提前采取拒绝措施，所以响应时间明显降低.

图 6 给出了三种调度策略在服务器过载时处理请求数量的统计.

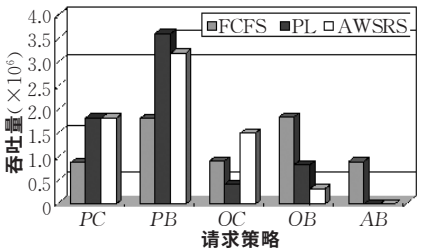


图 6 过载时不同调度策略下各类请求的吞吐量

由于服务器过载，所以三种调度策略都不能完成全部各类请求. 它们完成的各类请求数分别在用户发起的相同种类的请求总数中所占的比例见图 7.

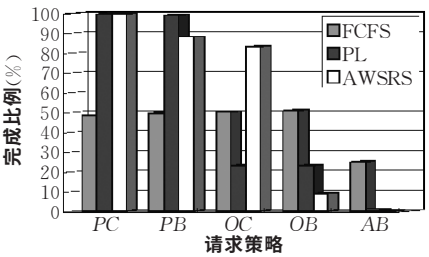


图 7 过载时不同调度策略下各类请求的完成百分比

从完成请求的比例来看，FCFS 完成请求的比例最平均，它只完成了最关键请求的 48.2%，但是同时完成了游客请求的 24.9%，因而这种调度方式是最不理想的；PL 调度按照优先级的原则处理了几乎全部的正式会员的请求，但是对于普通会员的请求处理则明显缺乏区分，导致其定购请求的处理率同浏览的处理率接近，游客的请求由于优先级较低，所以几乎没有得到处理；AWSRS 策略处理完了几乎全部的 *PC* 请求，而对游客的请求由于其请求的获益值为 0，则几乎没有得到处理. AWSRS 与 PL 差别最大的是对 *PB*，*OC* 和 *OB* 请求的处理，由于 *PB* 和 *OC* 的获益值相同，所以在调度的过程中只完成了 *PB* 请求的 88.3%，但是同时完成了 83.1% 的 *OC* 请求，而对获益值更低的 *OB*，则只完成了 9%.

为了验证请求获益对应用需求的反映，我们考察在不同获益比例设置下的 AWSRS 调度结果，对各种请求类型的获益比例值做如下修改(见表 2)，即提高用户级别的重要性，将正式会员请求的获益比例提高一倍；并且提高请求操作的重要性，将关键请求的获益比例提高一倍. 我们分别使用 AWSRS1 和 AWSRS2 表示在这两种不同的配置下的 AWSRS 策略测试情况，主要与 PL 策略进行比较，此时各种调度策略完成各类请求的比例如图 8 所示.

表 2 不同的获益比例

	AWSRS1	AWSRS2
PC	8	8
PB	4	2
OC	2	4
OB	1	1
AB	0	0

从图 8 可以看出，AWSRS1 更强调用户的优先级，所有的 *PC* 和 *PB* 请求都处理完成；AWSRS2 更强调请求操作的优先级，所以 *PC* 和 *OC* 都顺利完成，同时，在两种 AWSRS 调度中，*OC* 由于具有更高的获益值，所以处理情况要明显好于 *OB*. 因此与 PL 调度相比，AWSRS 能更灵活地满足用户需求的变化，并且通过适当的请求分类和获益值的定义即可达到 PL 调度的效果.

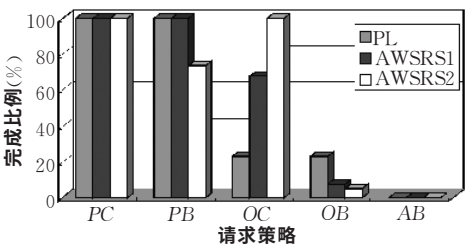


图 8 不同调度策略下各类请求的完成比例

4.3 实验小结

综上所述，FCFS 调度策略缺乏对重要请求的优先调度，而基于优先级的 PL 调度策略和本文讲

述的 AWSRS 都可以有效地使得重要的请求优先得到处理. 但是，AWSRS 具有更好的效果，它可以通过对不同用户不同请求类型指定其获益比例来影响服务器的请求调度情况，从而更灵活地满足应用的性能需求.

5　相关工作比较

Web 服务应用的性能问题已经引起了众多的关注. Web 服务应用的性能一方面由它所使用的 Web 服务决定，另一方面由它所在的运行支撑环境决定. 如何选择性能更优的 Web 服务是一个热门的研究领域[1,3～5]，然而本文的工作侧重于研究 Web 服务应用运行支撑环境的调度策略.

作为支撑 Web 服务应用的主流中间件平台，应用服务器目前最流行的调度策略仍然是 FCFS. 这种调度策略常常使得重要请求的性能无法满足用户的需求(尤其是在服务器重负载时). 为此，研究者提出了各种不同的改进方案. 文献[12,13]通过对请求的准入控制来降低过载对服务器造成的影响，而文献[7,8,14]则通过对请求进行分类的优先级调度应对服务器的过载. 这些研究成果的主要不足之处在于调度的原则都是出于服务器的设计考虑，而没有充分考虑应用的性能需求. 文献[9,10]研究了网络或服务器提供的客观的服务质量与用户对服务质量感受之间的关联，并给出了如何在服务器设计中考虑用户或应用对服务质量的需求. 本文吸收了这些工作的成果，使用应用获益来反映应用对服务器的性能需求，并以之作为调度的优化目标，从而最大限度地保证了服务器按照应用的意愿来提供服务质量.

服务器的性能自优化也是目前研究的一个热点问题. 它研究服务器如何通过自动调整自己的配置和调度策略来为应用提供服务质量保障. 文献[15,16]引入了反馈控制理论对 Web 服务器的静态页面调度进行自优化，但是该方法在复杂的 Web 应用中的应用还比较困难；文献[17]给出了一种 Web 系统中通过排队论对服务质量进行自优化的方法. 与之相比，本文通过计算应用的预期获益来进行有针对性的资源调整，这种方式比其通过对大量参数进行试探性的调整更有效.

6　结束语

Web 服务请求调度问题是企业级 Web 服务应用中亟待解决的问题. 本文从 Web 服务应用支撑环境出发，提出了一种应用敏感的 Web 服务请求调度策略 AWSRS. 该策略采用应用获益作为服务器的性能优化目标，利用运行时环境探测和获益预测结果，进行服务器资源的优化配置，能够优先保障更为重要的请求，从而有效反映了应用的性能需求. 文章后面的模拟实验验证了这一策略的有效性.

在本文中，我们并没有考虑各个请求类别之间的关联关系，然而挖掘不同类型请求之间的关联关系，可使得本文的方法更接近实际应用，在进一步的工作中，我们将深入考虑这一问题，进一步增强本文方法的实用性. 此外，本文的工作可以和 Web 服务选择的工作互补，我们将结合这两者实现更为灵活、更为完整的 Web 服务请求调度策略.

参　考　文　献

1　Yue Kun, Wang Xiao-Ling, Zhou Ao-Ying. Underlying techniques for Web services: A survey. Journal of Software, 2004, 15(3): 428～442(in Chinese)
(岳　昆，王晓玲，周傲英. Web 服务核心支撑技术：研究综述. 软件学报，2004，15(3)：428～442)

2　Menascé D. A.. QoS issues in Web services. IEEE Internet Computing, 2002, 6(6): 72～75

3　Zeng Liang-Zhao, Benatallah Boualem *et al*. QoS-aware middleware for Web services composition. IEEE Transactions on Software Engineering, 2004, 30(5): 311～327

4　Menascé D. A.. QoS-aware software components. IEEE Internet Computing, 2004, 8(2): 91～93

5　Erdogan Dogdu, Venkata Mamidenna. Efficient scheduling strategies for Web services-based E-Business transactions. In: Processdings of the 6th International Workshop of Technologies for E-Services, Trondheim, Norway, 2005, 113～125

6　Patel Chintan, Supekar Kaustubh, Lee Yugyung. A QoS oriented framework for adaptive management of Web service based workflows. In: Proceedings of the 14th International Conference of Database and Expert Systems Applications, Prague, Czech Republic, 2003, 826～835

7　Urgaonkar B., Shenoy P.. Cataclysm: Policing extreme overloads in Internet applications. In: Proceedings of the 14th International World Wide Web Conference, Chiba, Japan, 2005, 740～749

8　Bhatti N., Friedrich R.. Web server support for tiered services. IEEE Network, 1999, 13(5): 65～71

9　Bouch A., Kuchinsky A., Bhatti N.. Quality is in the eye of the beholder: Meeting users' requirements for Internet quality of service. In: Proceedings of the CHI 2000 Conference on Human Factors in Computing Systems, The Hague, The Netherlands, 2000, 297～304

10 Bhatti N., Bouch A., Kuchinsky A.. Integrating user-perceived quality into Web server design. In: Proceedings of the 9th International World Wide Web Conference, Amsterdam, Netherlands, 2000, 1～6

11 Huang Tao, Chen Ning-Jiang, Wei Jun, Zhang Wen-Bo, Zhang Yong. OnceAS/Q: A QoS-enabled Web application server. Journal of Software, 2004, 15(12): 1787～1799(in Chinese)
(黄 涛,陈宁江,魏 峻,张文博,张 勇. OnceAS/Q:一个面向 QoS 的 Web 应用服务器. 软件学报, 2004, 15(12): 1787～1799)

12 Elnikety S., Nahum E., Tracey J., Zwaenepoel W.. A method for transparent admission control and request scheduling in E-commerce Web sites. In: Proceedings of the 13th International Conference on World Wide Web, New York, USA, 2004, 276～286

13 Cherkasova L., Phaal P.. Session based admission control: A mechanism for peak load management of commercial Web sites. IEEE Transactions on Computers, 2002, 51(6): 669～685

14 Schroeder B., Harchol-Balter M.. Web servers under overload: How scheduling can help. ACM Transactions on Internet Technology, 2006, 6(1): 20～52

15 Abdelzaher T., Shin K. G., Bhatti N.. Performance guarantees for Web server end-systems: A control-theoretical approach. IEEE Transactions on Parallel and Distributed Systems, 2002, 13(1): 80～96

16 Abeni L., Palopoli L., Lipari G., Walpole J.. Analysis of a reservation-based feedback scheduler. In: Proceedings of the 23rd IEEE Real-Time Systems Symposium, Texas, USA, 2002, 71～80

17 Menascé D. A., Dodge R., Barbará D.. Preserving QoS of E-commerce sites through self-tuning: A performance model approach. In: Proceedings of the 3rd ACM Conference on Electronic Commerce, Florida, USA, 2001, 224～234

GUAN He-Qing, born in 1980, Ph. D. candidate. His research interests include distributed computing, software engineering and service oriented computing.

ZHANG Wen-Bo, born in 1976, Ph. D. candidate. His research interests include distributed computing and software engineering.

WEI Jun, born in 1970, Ph. D., professor. His research interests include service oriented computing, middleware, mobile computing, software engineering.

HUANG Tao, born in 1965, Ph. D., professor, Ph. D. supervisor. His research interests include distributed computing and software engineering.

Background

This research is supported by the National Natural Science Foundation of China (No. 60573126) and the National Basic Research Program (973 Program) of China (No. 2002CB312005). Both the two projects put an important role on the QoS guarantee of the middleware platform, especially the latter focus on the QoS mechanism of the internetware middleware to adaptive to the highly dynamic Internet environment.

The Internet-oriented applications constructed by the Web service technologies often suffer from the unexpected overloads, which require the QoS guarantee from the middleware platform emergently. This paper presents an application-aware Web service requests scheduling (AWSRS) strategy to deal with such overloads considering the QoS requirements, which proved to be effective to protect the benefit of the applications. This work would be solid progress to improve the QoS guarantee capability of the middleware platform.