一种多核环境中无锁的多进程负载均衡会话保持方案

吴和生*①② 王崇骏②③ 谢俊元②③
①(南京大学软件学院 南京 210093)
②(计算机软件新技术国家重点实验室 南京 210093)
③(南京大学计算机科学与技术系 南京 210093)

摘 要：负载均衡是云计算的基本问题之一，多核环境中多进程负载均衡会话保持问题得到广泛关注并发展成为研究热点。针对该问题，面向 Linux 内核，基于 Hash 化管理内核网络数据包传递的思想，该文提出并实现了一种无锁的多进程负载均衡会话保持方案。该方案避免了锁的使用，而且不需要对原有单进程负载均衡程序进行结构上的修改，能够快速地将现有单进程负载均衡程序转变为多进程架构。理论分析和实验表明，相较于传统的共享内存式锁机制会话保持解决方案，该方案性能更好、适用性更强，提高了多核环境中负载均衡系统的效率。

关键词：云计算；多核；多进程；负载均衡；会话保持；无锁
中图分类号：TP301 **文献标识码**： A **文章编号**：1009-5896(2013)04-0982-06
DOI: 10.3724/SP.J.1146.2012.01282

A Lock-free Multi-processing Session Persistence Mechanism for Load Balancing in Multi-core Environment

Wu He-sheng①② Wang Chong-jun②③ Xie Jun-yuan②③
①(*Software Institute*, *Nanjing University*, *Nanjing* 210093, *China*)
②(*National Key Laboratory for Novel Technology*, *Nanjing* 210093, *China*)
③(*Department of Computer Science and Technology*, *Nanjing University*, *Nanjing* 210093, *China*)

Abstract: Load balancing is a fundamental problem for cloud computing, multi-processing load balancing session persistence in multi-core environment have drawn more attention and have become a focus. For the issue, based on the idea of Hash Linux kernel network data packets passing, a lock-free multi-processing load balancing architecture is proposed, which avoids the use of locks, and can quickly change the existing single-processing load balancing procedure for multi-processing architecture without structural changes. The theory analysis and experimental results show that the proposed architecture is able to improve the overall performance of load balancing system in multi-core environment. Compared with the traditional shared memory architecture, the proposed is able to get better performance and has stronger applicability.

Key words: Cloud computing; Multi-core; Multi-processing; Load balancing; Session persistence; Lock-free

1 引言

负载均衡是云计算的重要组成部分，是服务器集群化中最为重要的环节[1]。与发展早期相比，现代负载均衡所面临的外部环境发生了许多变化，其中最为重要的变化之一是计算机处理器从单核变为多核。相较于传统的单进程负载均衡架构，在多核环境中多进程架构可以充分利用处理器的并行处理能力以提高系统的整体性能。因此，多核环境中多进程负载均衡成为现阶段云计算中的研究热点[2,3]。

在多核环境中如果采用多进程架构，让费时的包解析工作能够并发进行，就能显著提升系统的整体性能。但与现有单进程负载均衡架构相比，多进程负载均衡架构需要解决一些新的问题，其中一个尤为重要的问题是会话保持问题。

单核环境中负载均衡会话保持的研究已经积累了丰硕的成果，常用的方法有：简单会话保持(源地址会话保持)、基于 SSL Session ID 的会话保持、I-Rules 会话保持、HTTP Header 会话保持、HTTP Cookie 会话保持、基于 SIP ID 以及 Cache 设备的会话保持等。这些方法适用于单进程负载均衡架构，但应用在多核多进程负载均衡架构中会出现问题。以 HTTP 协议为例，应用这些方法造成多核多进程

2012-10-08 收到，2013-03-05 改回
国家自然科学基金(60503021, 60721002, 60875038)，江苏省高新技术研究发展计划(BE2009142, BE2010180)，教育部重点研究项目(108151)和南京大学研究生科研基金(2011CL07)资助课题
*通信作者：吴和生 weierson@smail.nju.edu.cn

负载均衡会话保持难以实现的原因是：第 1 个 HTTP 数据包到达进程 A 时新建的会话项被存储在进程 A 的会话表中，而第 2 个 HTTP 数据包到达进程 B 时进程 B 在自己的会话表中无法找到存储于进程 A 会话表中的对应会话项。解决这个问题的常见方法是使所有的负载均衡进程以共享内存的方式共享一个会话表。著名的负载均衡厂商 F5, Array 与 A10 均采用这一方式来实现多核多进程会话保持。这种方式将使会话表成为多进程并发访问的临界区，所以必须设计适当的锁机制来管理会话表的并发访问。锁机制的设计与使用的会话表数据结构密切相关，不管锁机制的设计如何精妙，都会或多或少地造成系统性能的降低；此外，使用共享内存的多进程负载均衡会话保持方案需要对原有单进程负载均衡程序进行大量结构上的修改。

本文针对多核环境中多进程负载均衡会话保持问题，面向 Linux 内核，基于 Hash 化管理内核网络数据包传递的思想，通过修改 Linux 内核实现了 Socket level hash 特性，在此基础上，提出并实现了一种无锁的多进程负载均衡会话保持方案。该方案避免了锁的使用，而且不需要对原有单进程负载均衡程序进行结构上的修改，能够快速地将现有单进程负载均衡程序转变为多进程架构。理论分析和实验表明，该方案提高了多核环境中负载均衡系统的效率。相较于传统的共享内存解决方案，本文提出的方法性能更好、适用性更强。

2　多核多进程负载均衡会话保持方案分析

2.1 多进程负载均衡在多核环境中的会话保持问题

为了实现会话保持，负载均衡程序需要维护一个会话表。在多核多进程架构中，由于多个进程监听在同一组 Socket 集上，当客户请求来到时，操作系统可能将该请求发往这些负载均衡进程中的任意一个，即属于同一个会话的多个请求很可能被发往不同的负载均衡进程进行处理。在这种情况下，不能像单进程架构那样维护局限于进程的会话表，而需要维护一个共享的全局会话表，每个进程都去访问该表，该表成为临界区。解决全局会话表的互斥访问问题的通用方案是用共享内存存储会话表，用锁来维护该表的一致性。

2.2 多核多进程负载均衡共享内存式锁机制分析

著名的负载均衡厂商 F5, Array 与 A10 普遍采用多进程共享内存式锁机制来解决多核环境中多进程会话保持问题。锁机制的设计与使用的会话表数据结构密切相关，在这里以开源软件 Haproxy 使用的会话表结构为例，对锁机制的设计作必要的分析。

Haproxy 中的会话表是一种散列表，用链表来解决 Hash 的冲突问题。每个会话项由会话 ID、后端服务器 ID、超时时间以及用来将会话项串入链表的 hash_list 结构组成。其中会话 ID 表示 HTTP 数据包中用来表示会话信息的内容，Haproxy 根据会话 ID 计算 Hash 值以决定将会话项放入哪个队列中；后端服务器 ID 表示该会话对应的后端服务器；超时时间表示该会话何时超时。

Haproxy 中对会话表的访问主要包括：

(1)当接收到 HTTP 数据包时，查询会话表，查找是否有对应的会话项，如果找到对应项，更新会话项的超时时间，否则生成新会话项并插入会话表。

(2)定期扫描会话表，删除超时会话项。

这些操作都需要用锁机制进行同步，针对 Haproxy 的会话表结构，锁设计方案有以下几种：

(a)只用一个锁来控制整个会话表的访问，进程在访问会话表前需要获得这个锁。毫无疑问这种设计的锁粒度太大，锁的争用将会非常严重，会造成系统性能的严重降低。

(b)每个 Hash 队列用一个锁控制访问。这种设计中使用同一个锁的范围减小到被 Hash 到同一队列的会话项的操作(查找、更新和插入)加上扫描并删除超时会话的操作(查找和删除)。锁的争用情况有所缓解，但当处理大量连接时，被 Hash 到同一队列的会话数目将会很大，系统性能依然可能因为等待锁而造成较为严重的损失。这种设计的锁粒度依然过大。

(c)每个会话项用一个锁来控制写操作，每个 Hash 队列用一个读写锁来控制。这种设计方案将锁的粒度进一步细化，并且改用读写锁来控制 Hash 队列的访问。这种设计方案较前两种方案更好地避免了锁的争用，但由于需要对每个会话项维护一个锁，对系统资源的消耗较大。

可见，不管锁机制如何设计，由于需要频繁查询、修改会话表，都会或多或少降低系统性能；何况使用共享内存的多进程负载均衡会话保持方案需要对原有单进程负载均衡程序进行大量结构上的修改。

3　无锁的多核多进程负载均衡会话保持方案

使用共享内存和锁机制的多进程负载均衡架构会造成系统性能的损失，而且现有单进程负载均衡程序需要大量结构上的修改才能适用这种架构。为了避免这些问题，本文通过修改 Linux socket 层的实现，使得 Linux 中处于 TCP_LISTEN 态的 Socket

能够提供一种称为 Socket level hash 的属性，该属性能保证当有多个用户进程尝试从该 Socket 获取新建 TCP 连接时，所有来自同一 IP 的连接请求都会发往同一个用户进程，进而保证所有属于同一用户会话的数据包都被发往同样的用户进程。

由于内核做了这样的保证，各个负载均衡进程只需要维护进程内部的会话表(正如单进程负载均衡程序所做的那样)而不需要将会话表放到共享内存中被所有进程共享，从而避免了锁的使用以及使用锁机制带来的性能损失。

3.1 Socket level hash特性

从 Linux 内核[4]Socket 层实现中可以看到，对一个处于 TCP_LISTEN 态的 Socket，每当一个 TCP 连接建立完成时，内核构造一个 request_sock 结构并将该结构放入该 Socket 对应 inet_connection_sock 结构的 icsk_accept_queue 队列中；当一个进程对指向该 Socket 的打开文件描述符调用 accept( )函数时，内核就从该 icsk_accept_queue 队列中取出(如果有)一个 request_sock 结构，并根据该结构生成新的 Socket，最终返回给用户一个指向新建 Socket 的打开文件描述符。内核对新建 TCP 连接并不区别对待，对尝试取 request_sock 结构的用户进程也不区别对待，任何进程都可能取得任何的 request_sock 结构。

本文通过给 TCP_LISTEN 态的 Socket 增两选项：SO_SOCKLEVELHASH 与 SO_SOCKADDLSPH 来实现 Socket level hash，其内核数据结构如图 1 所示。当一个处于 TCP_LISTEN 态的 Socket 设置了 SO_SOCKLEVELHASH 选项后，在内核中该 Socket 对应 inet_connection_sock 结构的 icsk_accept_queue 结构除了维护原有的 request_sock 队列外，还将额外维护 NR_CPUS 个高优先级队列(NR_CPUS 常量为系统中 CPU 的数目)。当某用户进程对一个处于 TCP_LISTEN 态并设置了 SO_SOCKLEVELHASH 选项的 Socket 设置 SO_SOCKADDLSPH 选项时，其进程 ID 将被注册到对应 icsk_accept_queue 结构的某个高优先级队列中，表示该用户进程将优先接收该队列中的 request_sock 结构。

打开 Socket level hash 选项的 TCP_LISTEN 态 Socket 处理新建 TCP 连接的方法与普通 Socket 不同，当一个 TCP 连接建立完成后，代表该 TCP 连接的 request_sock 结构被生成，内核首先调用 Hash 算法将该 request_sock 结构映射到 NR_CPUS 个高优先级队列中的一个中，如果该高优先级队列上已经有用户进程注册，则该 request_sock 结构被放入该队列中，否则，表示没有用户程序打算优先接收这个 request_sock 结构，内核将其放入默认队列中。

Linux IO 操作有 BLOCK_IO 和 NOBLOCK_IO 两类。对 NOBLOCK_IO，当用户进程调用 accept( )函数(最终调用 sys_accept( )内核函数)从打开 Socket level hash 选项的 Socket 中尝试获得新建 TCP 连接时，如果该用户进程已经将自己注册到某个高优先级队列中，则会首先从该队列取 request_sock 结构。对 BLOCK_IO, Linux 原来的逻辑是当用户进程调用 accept( )函数而不能取得 request_sock 结构时，进程将互斥地加入 sock 结构的 sk_sleep 等待队列并进入阻塞状态，当 TCP 连接建立，新的 request_sock 结构生成时，唤醒 sk_

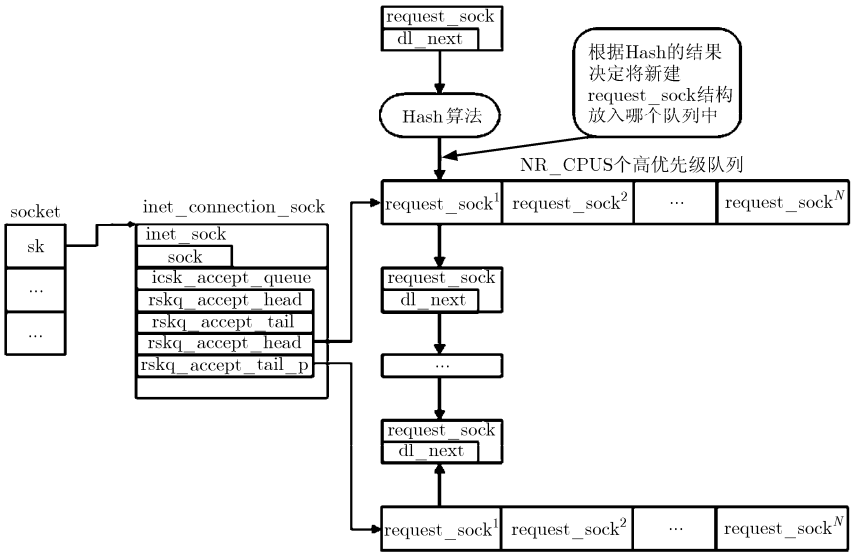

图 1 Socket level hash 内核数据结构

sleep 队列的一个进程，该进程将获得新建的 request_sock 结构。当 TCP_LISTEN 态 Socket 开启 Socket level hash 选项时，由于新建的 request_sock 不一定能映射给被唤醒的进程，原有处理 BLOCK_IO 的逻辑需要修改。修改方法有两种，一种方法是进程不再互斥地加入 sk_sleep 等待队列，每次 request_sock 生成时将 sk_sleep 上的所有进程都唤醒，这样该 request_sock 映射到的进程继续运行，其余进程继续阻塞在 sk_sleep 等待队列上。另一种方法是在 sock 结构中也相应地维护 NR_CPUS 个等待队列，用户进程根据自己所在 icsk_accept_queue 的哪个队列上注册来决定自己加入哪个等待队列，当一个 request_sock 结构生成时，也根据该结构被放入 icsk_accept_queue 的哪个队列来在对应等待队列上唤醒一个进程。

从以上讨论中可以看到，只要使用客户端 IP 地址作为 Hash 算法的输入，就可以保证所有来自同一客户端的连接请求都被发往同一个用户进程，从而保证所有属于同一用户会话的 HTTP 数据包都被发往同一个负载均衡进程；Hash 算法的分配均衡性和效率直接影响到 Socket level hash 的性能。

3.2 Socket level hash常用Hash算法

Hash 算法的种类很多，适应 Socket level hash 的 Hash 算法应该具有如下特征：

(1)对多种 Hash 输入类型都能产生较为均匀的 Hash 分布。这些 Hash 输入类型主要包括 TCP/IPV4, TCP/IPV6, IPV4, IPV6 协议数据包。

(2)在进程数目较少(如 2 个)或较多(如 64 个)的情况下都能将输入较为均匀地散列到各个进程中。

(3)Hash 函数的耗时要少。因为对每个新建连接都需要进行 Hash 运算，该 Hash 算法必须能计算得非常快，否则会成为系统性能的瓶颈。

根据上述特征分析表明，适用 Socket level hash 的 Hash 算法主要有 5 种(见算法 1～算法 5)。文献[5,6]讨论了前 4 种 Hash 算法的性质，文献[7]讨论了第 5 种 Hash 算法的性质。

算法 1　使用源 IP 地址进行 Hash

这是最简单的 Hash 方法，直接用源 IP 地址模进程数 N 即可。该 Hash 函数表示为：$H=\mathrm{SrcIP}\%N$。当 N=$2k$ 时，只需要取源 IP 地址的最后 k bit，就是 Hash 函数的结果。

算法 2　使用源 IP 的异或折叠(XOR Folding)

使用源 IP 地址的异或折叠 Hash 算法可表示成：H=$(D1\otimes D2\otimes D3\otimes D4)\%N$，其中 Di 表示源 IP 地址的第 i 个字节。

算法 3　使用源 IP 与目的 IP 的异或折叠

对算法 2 的简单改进是，将目的 IP 地址也作为 Hash 算法的参数，即将源 IP 地址与目的 IP 地址一起做异或折叠。该 Hash 算法表示为：H=$(S1\otimes S2\otimes S3\otimes S4\otimes D1\otimes D2\otimes D3\otimes D4)\ \%N$。

算法 4　CRC16 (16 bit 循环冗余检查)

CRC16 算法已经被证实可用于负载均衡中。本文用网络数据包中的 five-tuple(源 IP、目标 IP、源端口、目标端口、协议号)作为 CRC16 算法的输入，再对结果取模以构造 Hash 函数，算法描述为：$H=$ CRC16(five-tuple)$\%N$。

算法 5　Toeplitz hash

Toeplitz hash 是一种使用 Toeplitz 矩阵计算 Hash 值的 Hash 算法，Toeplitz 矩阵的特征是矩阵中处于同一对角线上的元素具有相同的值。更为精确地说，对 n 行 m 列的 Toeplitz 矩阵 $\boldsymbol{A}$，对任意 $1\le i, k\le n, i\le j, l\le m$，如果 $k-i=l-j$，则 $\boldsymbol{A}_{i,j}$=$\boldsymbol{A}_{k,l}$。

Free BSD 内核源码中提供了一个用于实现 Receive-Side Scaling 功能的 Toeplitz 矩阵[8]，本文使用该矩阵计算 request_sock 的 Hash 值。

4　理论分析及实验评估

4.1 理论分析

4.1.1 5 种 Hash 算法的比较分析　对于一个 Hash 算法，评价其优劣的重要标准应为分配均衡性和时间消耗。

Hash 算法的分配均衡性，即对任意一组样本，进入 Hash 表每一个单元之概率的平均程度。因为这个概率越平均，数据在表中的分布就越平均，表的空间利用率就越高。

现有研究表明，比特之间异或运算和位移运算能够提高哈希值的随机特性[9]。这 5 种 Hash 算法本质上都是位移和异或操作的组合，因此都具有较好的分配均衡性。

Hash 算法的时间消耗也直接影响着系统性能。不难计算，这 5 种 Hash 算法时间复杂度均为 $O(1)$。究竟哪种 Hash 算法计算最快，留待 4.2 节实验验证。

4.1.2 多核多进程负载均衡会话保持方案比较分析　多核环境中多进程负载均衡共享内存式锁机制会话保持解决方案(简称方案 1)需要精心设计锁来保护共享的会话表，对会话表的访问需要首先获得对应的锁，当锁被其他进程占用时，当前获取锁的进程必须等待锁的释放。当多核中同时有多个进程试图获得同一个锁时会产生争用，相关进程越多争用就越频繁，因而系统需要维护多个就绪队列和阻塞队列，开销将变得非常大。

不难计算，就绪队列和阻塞队列的时间复杂度和空间复杂度均为 $O(n)$，而锁争用过程中频繁地等待

资源，导致开销进一步加大。Etsion 等人[10]使用内核探测程序来测试内核锁的争用情况，统计结果显示，在 32 个进程时，内核锁竞争的开销多达 20%。袁清波等人[11]用 Sysbench 测试程序对 Linux 操作系统的文件系统部分进行了详细测试，结果显示，在 1024 个线程时，内核中对锁的等待时间已经超过对锁的占有时间，而且线程数越多这种现象越明显。显而易见，方案 1 锁的争用较内核锁争用的开销更大。

采用基于 Socket level hash 特性的无锁多核多进程负载均衡会话保持方案(简称方案 2)，需要额外维护多个优先级队列。基于“堆”(内嵌双向链表)实现的优先级队列的时间复杂度为 $O(\log_2 n)$，空间复杂度为 $O(1)$。与方案 1 的比较见表 1。

从表 1 可以看出，无论是在内存需求增量方面，还是在性能损耗方面，方案 2 都明显优于方案 1。

表 1 方案 1 与方案 2 的性能比较

关键比较项	方案 1	方案 2
内存的需求增量	需要维护多个就绪队列和阻塞队列，内存的需求增量为 $O(n)$	需要维护多个优先级队列，内存的需求增量为 $O(1)$；Hash 算法内存的需求增量为 $O(1)$
维护队列造成的性能损耗	需维护多个就绪队列和阻塞队列，性能损耗为 $O(n)$	需要维护多个优先级队列，性能损耗为 $O(\log_2 n)$
锁争用导致的性能损耗	大于维护队列造成的性能损耗，即 $\geq O(n)$	无
Hash 算法导致的性能损耗	无	$O(1)$

4.2 实验1 5种Hash算法的实验比较及评估

4.2.1 实验设置 本实验所使用数据集由美国麻省大学(Umass)[12]网络中心收集，该数据集记录了麻省大学从 2007 年 6 月 9 日到 2007 年 6 月 22 日期间每天上午 9:30~10:30 通过学校网关的数据包，共包括 654795 条记录。本实验使用 3.2 节提到的 5 种 Hash 算法计算这些包的 Hash 值，测试当存在 2, 4, 8, 16, 32, 64 个负载均衡进程的情况下 Hash 算法的分配均衡性，同时对这些过程计时，以考察 Hash 算法的时间消耗。

4.2.2 实验结果及评估 5 种 Hash 算法耗费时间分别为 0.031 s, 0.089 s, 0.162 s, 3.910 s, 0.153 s。

实验表明，算法 1 产生实验结果的不均衡性明显高于其他 4 种算法。

对其余 4 种算法产生的实验结果求标准差，结果如图 2 所示。

从图 2 中可以看到，当 Hash 目标数量较小时(如 2 或 4 个)，算法 2，算法 3 的均衡性严重降低；算法 4，算法 5 的均衡性几乎不受 Hash 目标数量多少的影响，具有很好的稳定性，而且无论 Hash 目标的多少，这两种算法的实验结果都具有最小的标准差，提供最均衡的 Hash 分配。但算法 4 的时间消耗是算法 5 的 25.6 倍，在 Socket level hash 的实现中，每个 request_sock 结构都需要进行一次 Hash 运算，该 Hash 算法的时间消耗直接影响着系统的整体性能，从这个角度来看，算法 5 优于算法 4。

综合考虑 Hash 算法的分配均衡性、时间消耗等各方面因素，可见 Toeplitz hash 最适合作为 Socket level hash 的 Hash 算法。

4.3 实验2 方案的实验比较及评估

4.3.1 实验设置 本实验使用以下工具：

(1)Haproxy Haproxy 是著名的开源 TCP/HTTP 负载均衡项目，提供了丰富的 7 层负载均衡功能[13]。Haproxy 使用单进程架构，正如本文中提到的问题，这款软件无法支持多进程负载均衡架构。

(2)Siege Siege 是一款网络性能测试工具，它以可配置的并发度在一段由用户配置的时间内不断地向目标服务器请求服务以测试目标服务器的性能，这个过程称为一次轰击(Hit)；在每次轰击之后 Siege 都会计算目标服务器的性能参数，包括目标服务器的连接率、吞吐量和并发连接数等[14]。

(3)VMware VMware 是一款虚拟化平台[15]，本实验使用 VMware 来生成虚拟机作为负载均衡器。本实验用到 2 台配置完全相同的虚拟机，虚拟机具有 2 个 CPU。其中虚拟机 1 运行原版 2.6.18 内核；虚拟机 2 运行修改过的支持 Socket level hash 的内核。两台虚拟机都运行 Haproxy 负载均衡程序，其中虚拟机 1 只有一个 Haproxy 进程，虚拟机 2 使用 Socket level hash 选项，运行着 2 个 Haproxy 进程。两台虚拟机都使用本机的 Redhat 自带 Http 服务器 Httpd 作为后端服务器。

4.3.2 实验结果及评估 实验步骤如下：

步骤 1 用 Siege 轰击虚拟机 1，记录性能参数。

步骤 2 选择 2 个具有会被 Socket level hash 分配给同一负载均衡进程的 IP 地址的机器，在这 2 台机器上分别用 Siege 轰击虚拟机 2，记录性能参数。

步骤 3 选择 2 个具有会被 Socket level hash 分配给不同负载均衡进程的 IP 地址的机器，在这 2 台上分别用 Siege 轰击虚拟机 2，记录性能参数。

以上每次轰击都持续 30 s，重复 10 次，最后得到的性能参数如图 3 所示。

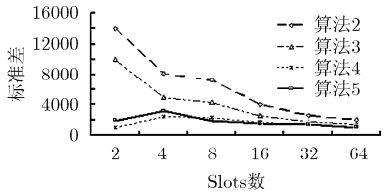

图 2 Hash 算法 2~算法 5 实验结果标准差

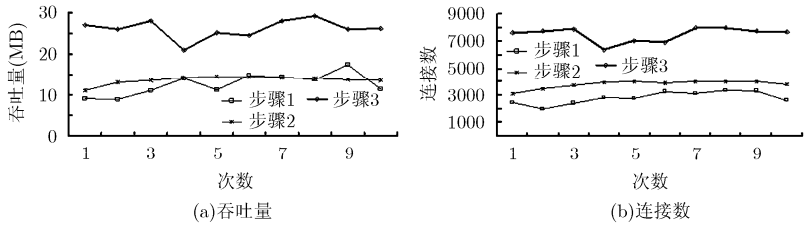

图 3 吞吐量和连接数

从图 3 中可以计算得到，步骤 3 相对于步骤 1 的吞吐量平均提高 107%，连接数平均提高 168%；步骤 2 相对于步骤 1 的吞吐量平均提高 8%，连接数平均提高 36%。

步骤 3 得到的服务器性能参数(无论是吞吐量还是连接率)远好于步骤 1，这说明相对于单进程架构而言，多核环境中多进程负载均衡架构能充分利用并行处理的好处，提高系统的整体性能。

值得注意的是，步骤 2 的性能参数远逊于步骤 3，与步骤 1 相比提高不多。步骤 2 模拟了一种极端的情况，即用在 Socket level hash 中的 Hash 算法产生极不均衡的分配时(步骤 2 中所有的连接都被发往同一个负载均衡进程)，系统的整体性能将急剧地降低；只有当 Socket level hash 产生较为均衡的分配时，使用 Socket level hash 的多进程负载均衡系统才能获得最大的性能提升。

5　结束语

多核环境中无锁的多进程负载均衡架构在 TeraScaler(作者参与研发的国家自然科学基金资助项目成果)负载均衡器中得以成功应用。TeraScaler 运行于充分定制的内核中，包括 Socket level hash 在内的定制内核功能极大地提高了产品的性能；弹性负载均衡资源管理功能的加入使得相较于普通负载均衡产品，TeraScaler 负载均衡器更适合应用于云计算环境中。当网络负载非常大时，负载均衡器本身可能成为整个系统的瓶颈，这时负载均衡的集群化就变得很有必要，这也是一个本文未能深入研究，但是未来值得研究的方向。

参 考 文 献

[1] Chiang M L, Yang C Y, and Lien S L. Kernel support for fine-grained load balancing in a web cluster providing streaming service[C]. Lecture Notes in Computer Science Volume 7439/2012: Algorithms and Architectures for Parallel Processing - 12th International Conference, Fukuoka, Japan, 2012: 458–472.

[2] Yang P J. Load balancing mechanism for QoS-aware cloud computing using eucalyptus platform[OL]. http://140.118.33.1/ETD-db/ETD-search/view_etd?URN=etd-0612111-175517, 2012.3.

[3] Hu J H, Gu J H, Sun G F, *et al*.. A scheduling strategy on load balancing of virtual machine resources in cloud computing environment[C]. 3rd International Symposium on Parallel Architectures, Algorithms and Programming (PAAP 2010), Dalian, China, 2010: 89–96.

[4] Torvalds L, *et al*.. The linux kernel archives[OL]. http://www.kernel.org/, 2012.4.

[5] Cao Z, Wang Z, and Zegura E. Performance of hashing-based schemes for Internet load balancing[C]. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies(Proceedings IEEE INFOCOM 2000), Tel Aviv, Israel, 2000: 332–341.

[6] Mansour Y, Nisan N, and Tiwari P. The computational complexity of universal hashing[C]. Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, Baltimore, USA, 1990: 235–243.

[7] Microsoft Corporation. Receive-side scaling enhancements in Windows Server 2008 [OL]. http://www.microsoft.com/whdc/device/network/ndis_rss.mspx, 2012.8.

[8] Ziehau S. FreeBSD/linux kernel cross reference: sys/net/toeplitz.c[OL].http://fxr.watson.org/fxr/source/net/toeplitz.c?v=DFBSD, 2012.7.

[9] 程光, 龚俭, 丁伟, 等. 面向 IP 流测量的哈希算法研究[J]. 软件学报, 2005, 16(5): 652–658.
Cheng Guang, Gong Jian, Ding Wei, *et al*.. A hash algorithm for IP flow measurement[J]. *Journal of Software*, 2005, 16(5): 652–658.

[10] Etsion Y, Tsafrir D, Kirkpatrick S,*et al*.. Fine grained kernel logging with KLogger: experience and insights[C]. Proceedings of the 2007 EuroSys Conference, Lisbon, Portugal, 2007: 259–272.

[11] 袁清波, 赵健博, 陈明宇, 等. 多核平台共享内存操作系统性能瓶颈分析及解决[J]. 计算机研究与发展, 2011, 48(12): 2268–2276.
Yuan Qing-bo, Zhao Jian-bo, Chen Ming-yu, *et al*.. Performance bottleneck analysis and solution of shared memory operating system on a multi-core platform[J]. *Journal of Computer Research and Development*, 2011, 48(12): 2268–2276.

[12] Umass. YouTube traces from the campus network[OL]. http://traces.cs.umass.edu/index.php/Network/Network, 2012.6.

[13] Tarreau W. Haproxy: the reliable, high performance TCP/HTTP load balancer[OL]. http://haproxy.1wt.eu/, 2012.9.

[14] Fulmer J. Siege home[OL]. http://www.joedog.org/ index/siege-home, 2012.9.

[15] VMware. Application platform[OL]. http://www.vmware.com/, 2012.9.

吴和生：　男，1973 年生，博士生，高级工程师，研究方向为应用软件工程、人工智能.
王崇骏：　男，1975 年生，教授，研究方向为人工智能、数据挖掘.
谢俊元：　男，1961 年生，教授，博士生导师，研究方向为人工智能、信息安全.