国防科学技术大学

博士学位论文

可伸缩网络服务的研究与实现

姓名：章文嵩

申请学位级别：博士

专业：计算机科学与技术

指导教师：金士尧

2000.10.1

摘 要

人类社会正在进入以网络为中心的信息时代，人们需要更快捷、更可靠、功能更丰富的网络服务。万维网的流行促进互联网使用的指数级增长，现在很多站点收到前所未有的访问负载，经常担心系统如何被扩展来满足不断增长的性能需求，同时系统如何保持 24x7 的可用性。未来的应用将需要更高的吞吐率、更好的交互性、更高的安全性，这要求服务平台具有更强的处理能力和更高的可用性。所以，如何给出合理的框架和有效的设计方法，来建立高性能、高可伸缩、高可用的网络服务，这是摆在研究者面前极富挑战性的任务。本文研究的可伸缩网络服务便是围绕这一任务展开的。

首先，在分析现在和将来网络服务需求的基础上，本文提出基于集群的可伸缩网络服务体系结构 Linux Virtual Server，分为负载调度器、服务器池和后端存储三层结构。负载调度器采用 IP 负载均衡技术和基于内容请求分发技术。LVS 集群提供了负载平衡、可伸缩性和高可用性，可以应用于建立很多可伸缩网络服务。进而，提出地理分布的 LVS 集群系统，可节约网络带宽，改善网络服务质量，具有良好的抗灾害性。

在 IP 负载均衡技术上，针对网络服务非对称性的特点，为克服 VS/NAT 伸缩能力差的缺点，本文提出通过 IP 隧道和直接路由实现虚拟服务器的两种方法 VS/TUN 和 VS/DR，极大地提高了系统的伸缩性。在负载调度研究中，针对请求的服务时间变化大，本文提出一个动态反馈负载均衡算法，结合内核中的加权连接调度算法，根据动态反馈回来的负载信息来调整服务器的权值，从而有效地解决服务器间的负载不平衡。我们在 Linux 内核中高效地实现了 IP 虚拟服务器软件，支持三种 IP 负载均衡技术和五种连接调度算法。

在对已有基于内容请求分发（Layer-7 交换）的方法进行分析基础上，本文提出在操作系统中利用内核线程高效地实现基于内容请求分发的解决方法，避免多次内核与用户空间的切换和内存复制开销。并提出基于局部性最小连接（LBLC）调度算法，在服务器间负载基本平衡情况下，提高单个服务器的访问局部性。该算法可提高后端服务器的主存 Cache 命中率，从而提高了整个集群系统的性能。针对高并发度的需求，我们引入多线程事件驱动的服务器结构，在 Linux 内核中高效地实现了内核 Layer-7 交换机 KTCPVS。实际的性能测试表明内核 Layer-7 交换方法有更高的吞吐率。

结合 IPVS 的 IP 负载均衡技术和 KTCPVS 的基于内容请求分发技术，本文设计和实现了大规模基于内容请求分发的系统。一个 VS/DR 或 VS/TUN 调度器作为系统的单一入口点，将请求分发给一组 KTCPVS 分发器，KTCPVS 分发器通过使用中心的内容位置服务，确定处理该请求的后端服务器，再将该请求分发到后端服务器；并提出基于局部性的最小负载（LBLL）调度算法。进而，将该体系结构应用大规模 Cache 集群系统中，提出面向 Cache 应用的 LBLL 算法。算法通过基于内

容请求分发可提高单个 Cache 服务器上请求的局部性；KTCPVS 分发器将不可 Cache 的对象请求跳过 Cache 服务器，直接发给源服务器，这样可以缩短处理时间，同时可以提高 Cache 服务器的使用效率；算法还保证 Cache 服务器在正常负载下工作。

在以上工作基础上，我们提供了一个 LVS 框架（Framework），包括含有三种 IP 负载均衡技术的 IP 虚拟服务器软件、基于内容请求分发的 KTCPVS 软件和相应的集群管理软件。利用 LVS 框架可以高效地实现高可伸缩的、高可用的 Web、Cache、Mail 和 Media 等网络服务，进而开发支持庞大用户数的、高可靠的电子商务应用。LVS 框架已在世界各地得到很好的实际应用，如 Red Hat 集群服务器软件、VA Linux 集群解决方案、英国国家 JANET Cache 网、Linux 门户站点、real.com 网站等。

关键词 服务器集群，可伸缩网络服务，可伸缩性，高可用性，
负载平衡，调度算法

ABSTRACT

The human society has been entering a network-centered information era, in which users need access faster, more reliable and functional network services. The increasing popularity of the World Wide Web leads to an exponential growth in the Internet usage. More and more network sites have attracted traffic at an unprecedented scale, they often worry about how to scale up their systems over time to meet this increasing demand, and how their systems remain 24x7 available. Future applications will need higher throughput, interaction and security, which requires service platform provide more processing power and higher availability. Therefore, the Internet research community faces a number of challenges at providing a feasible framework and design methodologies for constructing high-performance, highly scalable and highly available network services. The scalable network service addressed in this thesis represents a step towards ameliorating this situation.

On the analysis of present and future network service requirements, we present the architecture of cluster-based scalable network services called Linux Virtual Server, which has three tiers of load balancer, server pool and backend storage. The load balancer supports IP load balancing and content-based request distribution techniques. The LVS cluster provides load balancing, scalability and availability, which can be deployed to build many scalable network services. Furthermore, we give a geographically distributed LVS clusters, which can save bandwidth, improve quality of network services and be disaster-tolerant.

With analyzing the shortcoming of IP load balancing technique VS/NAT and the non-symmetric property of Internet services, we propose two techniques VS/TUN and VS/DR for building a virtual server via IP tunneling and direct routing respectively, which can greatly improve scalability of the system. In order to smooth extremely high variable traffic, we present a dynamic-feedback load balancing algorithm. Combining the weighted connection scheduling algorithms, it can adapt the server weights according to dynamic-feedback load information, therefore it can effectively solve the load imbalance problem among the servers. Finally, we implement IP Virtual Server software inside the Linux kernel, which supports three IP load balancing techniques and five connection scheduling algorithms.

On the analysis of the existing content-based request redistribution (Layer-7 switching) methods, we propose the method of using kernel thread to implement content-based request redistribution inside the operating system, which can avoid the overhead of context switching and memory copy between the kernel and user space. Furthermore, we present a locality-based least connection (LBLC) scheduling algorithm, which is to

improve the access locality on individual server when load on the servers is basically balanced. This algorithm can improve hit rate on backend servers' main memory cache, thus increase the performance of the whole cluster system. In order to meet high concurrency requirement, we introduce a multiple-thread event-driven server architecture, and efficiently implement a kernel Layer-7 switch called KTCPVS inside the Linux kernel. Performance testing results show that the kernel Layer-7 switching method has higher throughput.

Combining the IP load balancing techniques of IPVS and the content-based request distribution technique of KTCPVS, we design and implement a large-scale content-based request distribution system. A VS/DR or VS/TUN load balancer acts as a single entry point of the system, and distributes the incoming requests to a number of KTCPVS distributors. KTCPVS distributor queries the central Content Location Service to decide the right backend server to process the request, and then forwards the request to that server. We give a locality-based least loaded (LBLL) scheduling algorithm. Furthermore, we apply this architecture to build a large-scale Web cache cluster, and present cache-oriented LBLL algorithm. This algorithm uses content-based request distribution to improve the access locality on individual cache server. KTCPVS distributor can bypass cache server on non-cacheable object requests, and send the requests to the original servers directly. It can shorten the processing time, and improve the utilization of cache server. The algorithm can also guarantee that cache server operates under normal load.

Based on the above works, we provide a LVS framework, which includes IPVS of IP load balancing, KTCPVS of content-based request distribution and corresponding cluster management software. The LVS framework can be used to build highly scalable and highly available Web, Cache, Mail and Media services, and to develop large-scale and reliable e-commerce applications. The LVS framework has already had really big real applications all over the world, such as Red Hat Cluster Server software, VA Linux Cluster solution, UK National JANET Web Cache, Linux portal, and real.com site.

KEYWORDS: Server Cluster, Scalable Network Services, Scalability, High Availability, Load Balancing, Scheduling Algorithms

第一章 绪 论

在过去的十几年中，Internet 从几个研究机构相连为信息共享的网络发展成为拥有大量应用和服务的全球性网络，它正成为人们生活中不可缺少的一部分。虽然 Internet 发展速度很快，但建设和维护大型网络服务依然是一项挑战性的任务，因为系统必须是高性能的、高可靠的，尤其当访问负载不断增长时，系统必须能被扩展来满足不断增长的性能需求。由于缺少建立可伸缩网络服务的框架和设计方法，这意味着只有拥有非常出色工程和管理人才的机构才能建立和维护大型的网络服务。

针对这种情形，本文主要研究了可伸缩网络服务的体系结构、实现技术和相关算法，并实现了一个可伸缩网络服务的框架。

§1.1 研究背景

当今计算机技术已进入以网络为中心的计算时期。由于客户/服务器模型的简单性、易管理性和易维护性，客户/服务器计算模式在网上被大量采用。在九十年代中期，万维网（World Wide Web）[1]的出现以其简单操作方式将图文并茂的网上信息带给普通大众，Web 也正在从一种内容发送机制成为一种服务平台，大量的服务和应用(如新闻服务、网上银行、电子商务等)都是围绕着 Web 进行。这促进 Internet 用户剧烈增长和 Internet 流量爆炸式地增长，图 1.1 显示了 1995 至 2000 年与 Internet 连接主机数的变化情况[2]，可见增长趋势较以往更迅猛。

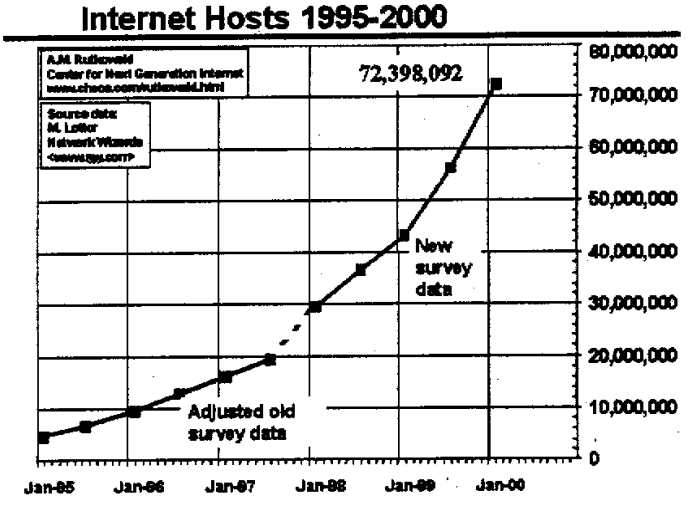

图 1.1：1995 至 2000 年 Internet 主机数的变化

Internet 的飞速发展给网络带宽和服务器带来巨大的挑战。从网络技术的发展来看，网络带宽的增长远高于处理器速度和内存访问速度的增长，如 100M Ethernet、ATM、Gigabit Ethernet 等不断地涌现，10Gigabit Ethernet 即将就绪，在主干网上密集波分复用将成为宽带 IP 的主流技术[3,4]，Lucent 已经推出在一根光纤跑 800Gigabit 的 WaveStar™ OLS 800G 产品[5]。所以，我们深信越来越多的瓶颈会出现在服务器端。很多研究显示 Gigabit Ethernet 在服务器上很难使得其吞吐率达到 1Gb/s 的原因是协议栈（TCP/IP）和操作系统的低效，以及处理器的低效，这需要对协议的处理方法、操作系统的调度和 IO 的处理作更深入的研究。在高速网络上，重新设计单台服务器上的网络服务程序也是个重要课题[6,7]。

比较热门的站点会吸引前所未有的访问流量，例如根据 Yahoo 的新闻发布，Yahoo 已经每天发送 6.25 亿页面[8]。一些网络服务也收到巨额的流量，如 American Online 的 Web Cache 系统每天处理 50.2 亿个用户访问 Web 的请求，每个请求的平均响应长度为 5.5Kbytes。与此同时，很多网络服务因为访问次数爆炸式地增长而不堪重负，不能及时处理用户的请求，导致用户进行长时间的等待，大大降低了服务质量。如何建立可伸缩的网络服务来满足不断增长的负载需求成为迫在眉睫的问题。

大部分网站都需要提供每天 24 小时、每星期 7 天的服务，对电子商务等网站尤为突出，任何服务中断和关键性的数据丢失都会造成直接的商业损失。例如，根据 Dell 的新闻发布[9]，Dell 现在每天在网站上的交易收入为一千四百万美元，一个小时的服务中断都会造成平均五十八万美元的损失。所以，这对网络服务的可靠性提出了越来越高的要求。

现在 Web 服务中越来越多地使用 CGI、动态主页等 CPU 密集型应用，这对服务器的性能有较高要求。未来的网络服务会提供更丰富的内容、更好的交互性、更高的安全性等，需要服务器具有更强的 CPU 和 I/O 处理能力。例如，通过 HTTPS（Secure HTTP）取一个静态页面需要的处理性能比通过 HTTP 的高一个数量级，HTTPS 正在被电子商务站点广为使用。所以，网络流量并不能说明全部问题，要考虑到应用本身的发展也需要越来越强的处理性能。

因此，解决网络服务的可伸缩性和可靠性已是非常紧迫的问题。通过高性能网络或局域网互联的服务器集群正成为实现高可伸缩、高可用网络服务的有效结构。服务器集群系统的优点：

- 性能

  网络服务的工作负载通常是大量相互独立的任务，通过一组服务器分而治之，可以获得很高的整体性能。

- 性能/价格比

  组成集群的 PC 服务器或 RISC 服务器和标准网络设备因为大规模生产，价格低，具有很高的性能价格比。若整体性能随着结点数的增长而接近线性增加，该系统的性能/价格比接近于 PC 服务器。所以，这种松耦合结构比紧耦合的多处理器系统具有更好的性能/价格比，

- 可伸缩性
  集群中的结点数目可以增长到成千上万个，其伸缩性远超过单台超级计算机。
- 高可用性
  在硬件和软件上都有冗余，通过检测软硬件的故障，将故障屏蔽，由存活结点提供服务，可实现高可用性。

用集群系统来提供可伸缩网络服务的难点：

- 透明性（Transparency）
  如何高效地使得由多个独立计算机组成的松藕合的集群系统构成一个虚拟服务器；客户端应用程序与集群系统交互时，就像与一台高性能、高可用的服务器交互一样，客户端无须作任何修改。部分服务器的切入和切出不会中断服务，这对用户也是透明的。
- 性能（Performance）
  性能要接近线性加速，这需要设计很好的软硬件的体系结构，消除系统可能存在的瓶颈。将负载较均衡地调度到个服务器上。
- 高可用性（Availability）
  需要设计和实现很好的系统资源和故障的监测和处理系统。当发现一个模块失败时，要这模块上提供的服务迁移到其他模块上。在理想状况下，这种迁移是即时的、自动的。
- 可管理性（Manageability）
  要使集群系统变得易管理，就像管理一个单一系统一样。在理想状况下，软硬件模块的插入能做到即插即用（Plug & Play）。
- 可编程性（Programmability）
  在集群系统上，容易开发应用程序。

§1.2 研究现状

下面简述当前用服务器集群实现高可伸缩、高可用网络服务的几种负载调度方法，并列举几个在这方面有代表性的研究项目。

1.2.1 网络服务的负载调度方法

在网络服务中，一端是客户程序，另一端是服务程序，在中间可能有代理程序。由此看来，可以在不同的层次上实现多台服务器的负载均衡。用集群解决网络服务性能问题的现有方法主要分为以下四类。

1.2.1.1 基于 RR-DNS 的解决方法

NCSA 的可伸缩的 WEB 服务器系统就是最早基于 RR-DNS（Round-Robin Domain Name System）的原型系统[10,11,12]。它的结构和工作流程如下图所示：

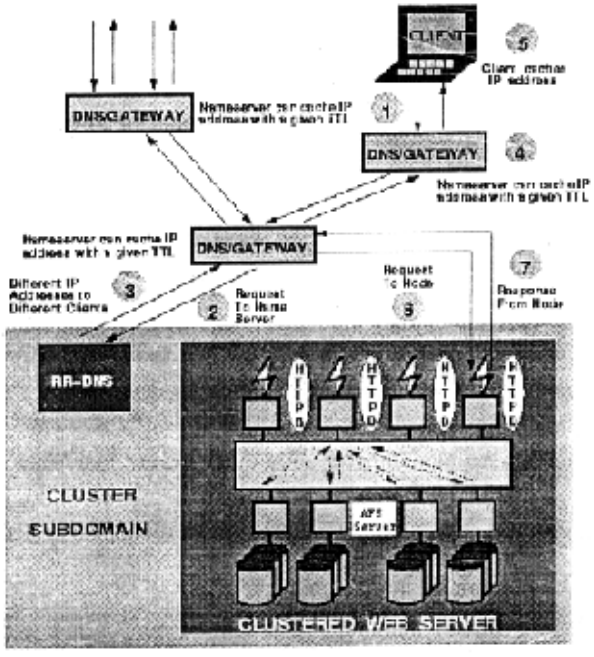

图 1.1：基于 RR-DNS 的可伸缩 WEB 服务器

有一组 WEB 服务器，他们通过分布式文件系统 AFS(Andrew File System)来共享所有的 HTML 文档。这组服务器拥有相同的域名，当用户按照这个域名访问时，RR-DNS 服务器会把域名轮流解析到这组服务器的不同 IP 地址，从而将负载分到各台服务器上。

这种方法带来几个问题。第一，域名服务器是一个分布式系统，是按照一定的层次结构组织的。当用户就域名解析请求提交给本地的域名服务器，它会因不能直接解析而向上一级域名服务器提交，上一级域名服务器再依次向上提交，直到 RR-DNS 域名服器把这个域名解析到其中一台服务器的 IP 地址。可见，从用户到 RR-DNS 间存在多台域名服器，而它们都会缓冲已解析的名字到 IP 地址的映射,这会导致该域名服器组下所有用户都会访问同一 WEB 服务器，出现不同 WEB 服务器间严重的负载不平衡。为了保证在域名服务器中域名到 IP 地址的映射不被长久缓冲，RR-DNS 在域名到 IP 地址的映射上设置一个 TTL(Time To Live)值，过了这一段时间，域名服务器将这个映射从缓冲中淘汰。当用户请求，它会再向上一级域名服器提交请求并进行重新影射。这就涉及到如何设置这个 TTL 值，若这个值太大，在这个 TTL 期间，很多请求会被映射到同一台 WEB 服务器上，同样会导致严重的负载不平衡。若这个值太小，例如是 0，会导致本地域名服务器频繁地向 RR-DNS 提

交请求，增加了域名解析的网络流量，同样会使 RR-DNS 成为系统中一个新的瓶颈。

第二，用户机器会缓冲从名字到 IP 地址的映射，而不受 TTL 值的影响，用户的访问请求会被送到同一台 WEB 服务器上。由于用户访问请求的突发性和访问方式不同，例如有的人访问一下就离开了，而有的人访问可长达几个小时，所以各台服务器间的负载仍存在倾斜（Skew）而不能控制。假设用户在每个会话中平均请求数为 20，负载最大的服务器获得的请求数额高于各服务器平均请求数的平均比率超过百分之三十。也就是说，当 TTL 值为 0 时，因为用户访问的突发性也会存在着较严重的负载不平衡。

第三，系统的可靠性和可维护性差。若一台服务器失效，会导致将域名解析到该服务器的用户看到服务中断，即使用户按“Reload”按钮，也无济于事。系统管理员也不能随时地将一台服务器切出服务进行维护，如进行操作系统和应用软件升级，这需要修改 RR-DNS 服务器中的 IP 地址列表，把该服务器的 IP 地址从中划掉，然后等上几天或者更长的时间，等所有域名服器将该域名到这台服务器的映射淘汰，和所有映射到这台服务器的客户机不再使用该站点为止。

1.2.1.2 基于客户端的解决方法

基于客户端的解决方法需要每个客户程序都有一定的服务器集群的知识，进而把以负载平衡的方式将请求发到不同的服务器。例如，Netscape Navigator 浏览器访问 Netscape 的主页时，它会随机地从一百多台服务器中挑选第 N 台，最后将请求送往 wwwN.netscape.com。然而，这不是很好的解决方法，Netscape 只是利用它的 Navigator 避免了 RR-DNS 解析的麻烦，当使用 IE 等其他浏览器不可避免的要进行 RR-DNS 解析。

Smart Client[13]是 Berkeley 做的另一种基于客户端的解决方法。服务提供一个 Java Applet 在客户方浏览器中运行，Applet 向各个服务器发请求来收集服务器的负载等信息，再根据这些信息将客户的请求发到相应的服务器。高可用性也在 Applet 中实现，当服务器没有响应时，Applet 向另一个服务器转发请求。这种方法的透明性不好，Applet 向各服务器查询来收集信息会增加额外的网络流量，不具有普遍的适用性。

1.2.1.3 基于应用层负载平衡调度的解决方法

多台服务器通过高速的互联网络连接成一个集群，在前端有一个基于应用层的负载平衡调度器。当用户访问请求到达调度器时，请求会提交给作负载平衡调度的应用程序，分析请求，根据各个服务器的负载情况，选出一台服务器，重写请求并向选出的服务器访问，取得结果后，再返回给用户。

应用层负载平衡调度的典型代表有 Zeus 负载调度器[14]、pWeb[15]、Reverse-Proxy[16]和 SWEB[17, 18, 19, 20]等。Zeus 负载调度器是 Zeus 公司的商业产品，它是在 Zeus Web

服务器程序改写而成的，采用单进程事件驱动的服务器结构。pWeb 就是一个基于 Apache 1.1 服务器程序改写而成的并行 WEB 调度程序，当一个 HTTP 请求到达时，pWeb 会选出一个服务器，重写请求并向这个服务器发出改写后的请求，等结果返回后，再将结果转发给客户。Reverse-Proxy 利用 Apache 1.3.1 中的 Proxy 模块和 Rewrite 模块实现一个可伸缩 WEB 服务器，它与 pWeb 的不同之处在于它要先从 Proxy 的 cache 中查找后，若没有这个副本，再选一台服务器，向服务器发送请求，再将服务器返回的结果转发给客户。SWEB 是利用 HTTP 中的 redirect 错误代码，将客户请求到达一台 WEB 服务器后，这个 WEB 服务器根据自己的负载情况，自己处理请求，或者通过 redirect 错误代码将客户引到另一台 WEB 服务器，以实现一个可伸缩的 WEB 服务器。

基于应用层负载平衡调度的多服务器解决方法也存在一些问题。第一，系统处理开销特别大，致使系统的伸缩性有限。当请求到达负载平衡调度器至处理结束时，调度器需要进行四次从核心到用户空间或从用户空间到核心空间的上下文切换；需要进行二次 TCP 连接，一次是从用户到调度器，另一次是从调度器到真实服务器；需要对请求进行分析和重写。这些处理都需要不小的ＣＰＵ、内存和网络等资源开销，且处理时间长。所构成系统的性能不能接近线性增加的，一般服务器组增至 3 或 4 台时，调度器本身可能会成为新的瓶颈。所以，这种基于应用层负载平衡调度的方法的伸缩性极其有限。第二，基于应用层的负载平衡调度器对于不同的应用，需要写不同的调度器。以上几个系统都是基于 HTTP 协议，若对于 FTP、Mail、POP3 等应用，都需要重写调度器。

1.2.1.4 基于 IP 层负载平衡调度的解决方法

用户通过虚拟地址（Virtual IP Address）访问服务时，访问请求的报文会到达虚拟服务器主机，由它进行负载平衡调度，从一组真实服务器选出一个，将报文的目标地址 Virtual IP Address 改写成选定服务器的地址，报文的目标端口改写成选定服务器的相应端口，最后将报文发送给选定的服务器。真实服务器的回应报文经过虚拟服务器主机时，将报文的源地址和源端口改为 Virtual IP Address 和相应的端口，再把报文发给用户。Berkeley 的 MagicRouter[21]、Cisco 的 LocalDirector[22]、Alteon 的 ACEDirector[23]和 F5 的 Big/IP[24] 等都是使用网络地址转换方法。MagicRouter 是在 Linux 1.3 版本上应用快速报文插入技术，使得进行负载平衡调度的用户进程访问网络设备接近核心空间的速度，降低了上下文切换的处理开销，但并不彻底，它只是研究的原型系统，没有成为有用的系统存活下来。Cisco 的 LocalDirector、Alteon 的 ACEDirector 和 F5 的 Big/IP 是非常昂贵的商品化系统，它们支持部分 TCP/UDP 协议，在 ICMP 处理上存在问题。

IBM 的 TCP Router[25]使用修改过的网络地址转换方法在 SP/2 系统实现可伸缩的 WEB 服务器。TCP Router 修改请求报文的目标地址并把它转发给选出的服务器，服务器能把响应报文的源地址置为 TCP Router 地址而非自己的地址。这种方法的

好处是响应报文可以直接返回给客户，坏处是每台服务器的操作系统内核都需要修改。IBM 的 NetDispatcher[26, 27]是 TCP Router 的后继者，它将报文转发给服务器，而服务器在 non-ARP 的设备配置路由器的地址。这种方法与 LVS 中的 VS/DR 类似，它具有高可伸缩性，但一套在 IBM SP/2 和 NetDispatcher 需要上百万美金。

在贝尔实验室的 ONE-IP[28]中，每台服务器都独立的 IP 地址，但都用 IP Alias 配置上同一 VIP 地址，采用路由和广播两种方法分发请求，服务器收到请求后按 VIP 地址处理请求，并以 VIP 为源地址返回结果。这种方法也是为了避免回应报文的重写，但是每台服务器用 IP Alias 配置上同一 VIP 地址，会导致地址冲突，有些操作系统会出现网络失效。通过广播分发请求，同样需要修改服务器操作系统的源码来过滤报文，使得只有一台服务器处理广播来的请求。

微软的 Windows NT 负载均衡服务（Windows NT Load Balancing Service，WLBS）[29]是 1998 年底收购 Valence 研究公司获得的，它与 ONE-IP 中的基于本地过滤方法一样。WLBS 作为过滤器运行在网卡驱动程序和 TCP/IP 协议栈之间，获得目标地址为 VIP 的报文，它的过滤算法检查报文的源 IP 地址和端口号，保证只有一台服务器将报文交给上一层处理。但是，当有新结点加入和有结点失效时，所有服务器需要协商一个新的过滤算法，这会导致所有有 Session 的连接中断。

1.2.2 正在开展的研究项目

1.2.2.1 EDDIE Project

EDDIE Project[30]是爱立信研究室正在开展的项目，目标是实现一个鲁棒的、可伸缩的 Internet 服务器。它使用的主要技术是改进的 RR-DNS 和应用层调度的方法。正如所说，这些方法分别存在负载不平衡和伸缩能力有限的问题。

1.2.2.2 Berkeley's Cluster-Based Scalable Network Services

在文献[31]中，加州大学 Berkeley 分校的研究人员给出了基于集群系统的可伸缩网络服务的三层模型，它们依次从底向上为可伸缩网络服务 SNS (Scalable Network Services)、TACC(Transformation, Aggregation, Caching, Customization)和 Service，讲述了怎样基于该模型构造 Cluster-aware 可伸缩的网络服务，将网络服务进行分割，分发到各个结点运行。这种方法需要重新开发网络服务应用程序，而且由应用程序本身提供负载平衡调度和容错处理。

1.2.2.3 Ninja Project

Ninja 项目是加州大学 Berkeley 分校正在的研究项目，它的目标是开发一种软件体系结构来支持下一代 Internet 应用[32]。Ninja 方法的核心是在集群环境建立一个软件平台 bases，它提供可伸缩、高可用、容错和数据持久性等服务，来简化可伸缩网络服务的建设。整个平台是用 Java 语言开发，这需要用 Java 语言重写网络服务应用。目前，这个项目正在进展中。Steven Gribble 刚完成的博士论文[33]主要讲述了为 Ninja 项目设计和完成分布式数据结构（Distributed Data Structure）支持数据持久性服务。

1.2.2.4 Microsoft Cluster Service

微软的集群服务 MSCS[34]目前正处在第一个开发阶段，采用 Share-nothing 的集群模型，每个结点运行 NT 操作系统。MSCS 主要实现了 NT 集群系统的统一管理，结点的加入和离开，结点的故障检测和故障后（Failover）处理。目前 MSCS 主要实现了集群系统的高可用性，但没有涉及到集群系统的性能上的伸缩性，没有事务处理，若一个结点失效，另一个结点取而代之，但失效结点当前处理的请求会被丢掉，需要用户重新提交请求。

1.2.2.5 Microsoft's Scalable Servers Research Group

这是 Jim Gray 在微软研究院领导的可伸缩服务器研究小组，研究基于 Windows NT Clusters 的可伸缩服务器[35, 36, 37]。Jim Gray 的 NT Clusters 定义如下：

NT Clusters: I believe you can build supercomputers as a cluster of commodity hardware and software modules. A cluster is a collection of independent computers that is as easy to use as a single computer. Managers see it as a single system, programmers see it as a single system, and users see it as a single system. The software spreads data and computation among the nodes of the cluster. When a node fails, other nodes provide the services and data formerly provided by the missing node. When a node is added or repaired, the cluster software migrate some data and computation to that node.

在 1997 年，他们演示了在两个结点的 NT Clusters 上高可用的 SQL Server，故障后时间（Failover time）为 15 秒。在伸缩性方面，1997 年 5 月 20 日他们演示了 45 个结点的 NT Clusters 系统，每天能处理 10 亿个 Transaction。但是，在这个系统中每个结点服务于不同的客户，并行性是静态分割好的，所以可伸缩服务器有待于深入研究。

Jim Gray 等在 1999 向美国总统提交的“信息技术研究：投资我们的未来”报告，指出政府在以后十年应加强在软件技术、可伸缩信息基础设施和高端计算的研

究投入[38]。他们还认为可伸缩服务器将会取代伸缩性差的多处理器服务器[39]。

§1.3 主要研究的问题

本文研究的主要问题是可伸缩网络服务的体系结构和实现技术，针对如何提高基于集群网络服务的可伸缩性和高可用性，主要研究了如下几方面的内容：

- 可伸缩网络服务体系结构的研究和设计。
- 研究提高网络服务可伸缩性的实现技术
- 负载平衡调度算法
- 系统高可用性的研究和实现
- 系统实现并提供一个可伸缩网络服务的框架

§1.4 研究工作的主要贡献

本文的主要贡献及创新工作包括以下几个方面：

1. 在分析现在和将来网络服务需求的基础上，提出可伸缩网络服务的体系结构 Linux Virtual Server，分为负载调度器、服务器池和后端存储三层结构。负载调度器采用 IP 负载均衡技术和基于内容请求分发技术。LVS 集群提供了负载平衡、可伸缩性和高可用性，可以应用于建立很多可伸缩网络服务。在此基础上，我们提出了地理分布的 LVS 集群系统，它可以节约网络带宽，改善网络服务质量，具有很好的抗灾害性。
2. 在 IP 负载均衡技术上，针对网络服务非对称性的特点，为克服 VS/NAT 伸缩能力差的缺点，我们提出了通过 IP 隧道实现虚拟服务器的方法 VS/TUN，和通过直接路由实现虚拟服务器的方法 VS/DR，它们可以极大地提高系统的可伸缩性。在负载调度研究中，针对请求的服务时间变化大，我们提出一个动态反馈负载均衡算法，结合内核中的加权连接调度算法，根据动态反馈回来的负载信息来调整服务器的权值，来调整服务器间处理请求数的比例，从而有效地避免服务器间的负载不平衡。最后，我们在 Linux 内核中高效地实现了 IP 虚拟服务器软件，支持三种 IP 负载均衡技术和多种连接调度算法。
3. 在对已有基于内容请求分发（即 Layer-7 交换）的方法进行分析基础上，提出在操作系统内核中利用内核线程高效地实现 Layer-7 交换的解决方法，可以避免了多次内核与用户空间的切换和内存复制开销。并提出基于局部性的最小连接（LBLC）调度算法，在服务器间负载基本平衡情况下，将相似的请求发到同一服务器，提高单个服务器的访问局部性。LBLC 算法可提高后端服务器的主存 Cache 命中率，从而提高整个集群系统的性能。针对高并发度的需求，我们引入多线程事件驱动的服务器结构，在 Linux

内核中高效地实现了内核 Layer-7 交换机 KTCPVS。实际的性能测试说明内核 Layer-7 交换方法具有良好的性能。

4. 结合 IPVS 的 IP 负载均衡技术和 KTCPVS 的基于内容请求分发技术，我们设计和实现了大规模基于内容请求分发的系统。一个 VS/DR 或 VS/TUN 调度器为集群系统的单一入口点，将请求分发给一组 KTCPVS 分发器，它们使用中心的内容位置服务来确定处理该请求的后端服务器；并提出适用于内容位置服务中的基于局部性最小负载（LBLL）调度算法。
5. 我们将大规模基于内容请求分发结构应用到 Cache 集群系统中，提出了面向 Cache 应用的 LBLL 算法。该算法使得 KTCPVS 将不可 Cache 的对象请求跳过 Cache 服务器，直接发给源服务器，这样可以缩短处理时间同时可提高 Cache 服务器的使用效率；通过基于内容请求分发提高单个 Cache 服务器上请求的局部性，即将整个请求的工作集分割成单个 Cache 服务器上基本上不重叠的子集，可使得整个系统性能超线性地增加；算法还保证 Cache 服务器在正常负载下工作，若 Cache 服务器超载，KTCPVS 分发器将跳过 Cache 服务器。

在以上工作基础上，我们提供了一个 Linux Virtual Server 框架，如图 1.2 所示。在 LVS 框架中，提供了含有三种 IP 负载均衡技术的 IP 虚拟服务器软件、基于内容请求分发的内核 Layer-7 交换机 KTCPVS 和集群管理软件。可以利用 LVS 框架实现高可伸缩的、高可用的 Web、Cache、Mail 和 Media 等网络服务，在此基础上，可以开发支持庞大用户数的、高可伸缩的、高可用的电子商务应用。LVS 框架已在世界各地得到很好的实际应用，如 Red Hat 集群服务器软件、VA Linux 集群解决方案、红旗 Linux LVS 集群软件、英国国家 JANET Cache 网（wwwcache.ja.net）、Linux 门户站点（linux.com）、开发源码站点（sourceforge.net）、real.com 网站等。

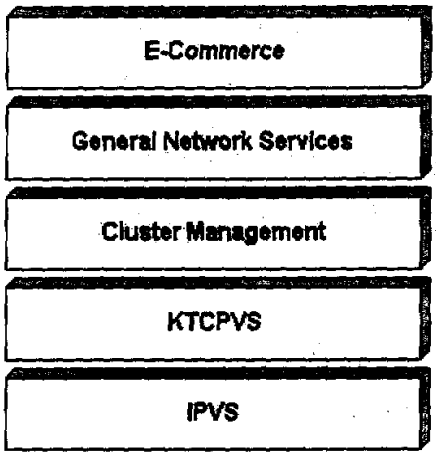

图 1.2：Linux 虚拟服务器框架

§1.5 论文结构

论文共分为八章。各章节的组织结构如图 1.3 所示：

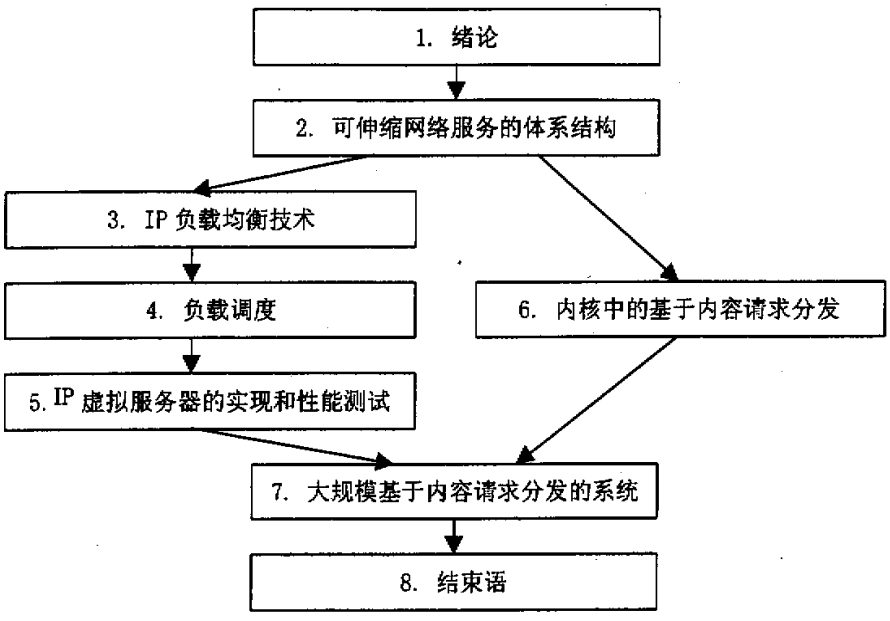

图 1.3：论文的组织结构

第一章为绪论。本章首先分析了 Internet 的飞速发展对网络带宽和服务器处理性能带来巨大的挑战，和利用服务器集群提高服务处理能力的优点和难点。简述了服务器集群系统的研究现状，最后介绍了本文的研究内容及主要贡献。

第二章分析了现在和将来网络服务的需求，提出可伸缩网络服务的体系结构 LVS，提供了负载平衡、可伸缩性和高可用性，可以应用于建立很多可伸缩网络服务。在此基础上，我们提出了地理分布的 LVS 系统，它可以节约网络带宽，改善网络服务质量，具有很好的抗灾害性。

第三章讲述了三种 IP 负载均衡技术。在分析 VS/NAT 的缺点和网络服务非对称性的基础上，我们提出通过 IP 隧道实现虚拟服务器的方法 VS/TUN，和通过直接路由实现虚拟服务器的方法 VS/DR,它们可以极大地提高系统的伸缩性。

第四章描述了 IP 虚拟服务器在内核中的各种连接调度算法。针对请求的服务时间变化很大，我们提出一个动态反馈负载均衡算法，结合内核中的加权调度算法，根据动态反馈回来的负载信息来调整服务器的权值，来避免服务器间负载不平衡。

第五章主要讲述了 IP 负载均衡技术和连接调度算法在 Linux 内核中的实现，以及实现中遇到的相关问题和优化。对 IP 虚拟服务器软件进行性能测试，并列举该软件的应用情况。

第六章在对已有的基于内容请求分发方法（即 Layer-7 交换）进行分析基础上，提出在操作系统内核中利用内核线程高效地实现 Layer-7 交换的解决方法，可以避免了多次内核与用户空间的切换和内存复制开销。详细阐述内核 Layer-7 交换机 KTCPVS 的实现，提出基于局部性的最小连接负载调度算法，采用多进程事件驱动的服务器结构。实际的性能测试说明内核 Layer-7 交换方法具有良好的伸缩性。

第七章讲述大规模基于内容请求分发系统的体系结构和实现，并对其进行简要的性能测试。应用该体系结构到大规模 Cache 集群系统中，并对有关算法进行改进。

第八章对本文的工作进行总结，并对下一步工作进行展望。

第二章 可伸缩网络服务的体系结构

针对网络服务的可伸缩性、高可用性、可维护性和价格有效性需求，本章提出了可伸缩网络服务的体系结构和设计方法，它提供了负载平衡、可伸缩性和高可用性。

§2.1 可伸缩网络服务的定义

可伸缩性（Scalability）是在当今计算机技术中经常用到的词汇。对于不同的人，可伸缩性有不同的含义。 现在，我们来定义可伸缩网络服务的含义。

可伸缩网络服务是指网络服务能随着用户数目的增长而扩展其性能，如在系统中增加服务器、内存或硬盘等；整个系统很容易被扩展，无需重新设置整个系统，无需中断服务。换句话说，系统管理员扩展系统的操作对最终用户是透明的，他们不会知道系统的改变。

可伸缩系统通常是高可用的系统。在部分硬件（如硬盘、服务器、子网络）和部分软件（如操作系统、服务进程）的失效情况下，系统可以继续提供服务，最终用户不会感知到整个服务的中断，除了正在失效点上处理请求的部分用户可能会收到服务处理失败，需要重新提交请求。Caching 和复制是建立高可用系统的常用技术，建立多个副本会导致如何将原件的修改传播到多个副本上的问题。

实现可伸缩网络服务的方法一般是通过一对多的映射机制，将服务请求流分而治之（Divide and Conquer）到多个结点上处理。一对多的映射可以在很多层次上存在，如主机名上的 DNS 系统、网络层的 TCP/IP、文件系统等。虚拟（Virtual）是描述一对多映射机制的词汇，将多个实体组成一个逻辑上的、虚拟的整体。例如，虚存（Virtual Memory）是现代操作系统中最典型的一对多映射机制，虚存建立一个虚拟内存空间，将它映射到多个物理内存上[40]。

§2.2 网络服务的需求

随着 Internet 的飞速发展和对我们生活的深入影响，越来越多的个人在互联网上购物、娱乐、休闲、与人沟通、获取信息；越来越多的企业把他们与顾客和业务伙伴之间的联络搬到互联网上，通过网络来完成交易，建立与客户之间的联系。互联网的用户数和网络流量正以几何级数增长，这对网络服务的可伸缩性提出很高的要求。例如，比较热门的 Web 站点会因为被访问次数急剧增长而不能及时处理用户的请求，导致用户进行长时间的等待，大大降低了服务质量。另外，随着电子商务等关键性应用在网上运行，任何例外的服务中断都将造成不可估量的损失，服务

的高可用性也越来越重要。所以，对用硬件和软件方法实现高可伸缩、高可用网络服务的需求不断增长，这种需求可以归结以下几点：

- 可伸缩性（Scalability），当服务的负载增长时，系统能被扩展来满足需求，且不降低服务质量。
- 高可用性（Availability），尽管部分硬件和软件会发生故障，整个系统的服务必须是每天 24 小时每星期 7 天可用的。
- 可管理性（Manageability），整个系统可能在物理上很大，但应该容易管理。
- 价格有效性（Cost-effectiveness），整个系统实现是经济的、易支付的。

单服务器显然不能处理不断增长的负载。这种服务器升级方法有下列不足：一是升级过程繁琐，机器切换会使服务暂时中断，并造成原有计算资源的浪费；二是越往高端的服务器，所花费的代价越大；三是一旦该服务器或应用软件失效，会导致整个服务的中断。

通过高性能网络或局域网互联的服务器集群正成为实现高可伸缩的、高可用网络服务的有效结构。这种松耦合结构比紧耦合的多处理器系统具有更好的伸缩性和性能价格比，组成集群的 PC 服务器或 RISC 服务器和标准网络设备因为大规模生产，价格低，具有很高的性能价格比。但是，这里有很多挑战性的工作，如何在集群系统实现并行网络服务，它对外是透明的，它具有良好的可伸缩性和可用性。

针对上述需求，我们给出了基于 IP 层和基于内容请求分发的负载平衡调度解决方法，并在 Linux 内核中实现了这些方法，将一组服务器构成一个实现可伸缩的、高可用网络服务的服务器集群，我们称之为 Linux 虚拟服务器（Linux Virtual Server）。在 LVS 集群中，使得服务器集群的结构对客户是透明的，客户访问集群提供的网络服务就像访问一台高性能、高可用的服务器一样。客户程序不受服务器集群的影响不需作任何修改。系统的伸缩性通过在服务机群中透明地加入和删除一个节点来达到，通过检测节点或服务进程故障和正确地重置系统达到高可用性。

§2.3 LVS 集群的体系结构

下面先给出 LVS 集群的通用结构，讨论了它的设计原则和相应的特点；然后将 LVS 集群应用于建立可伸缩的 Web、Media、Cache 和 Mail 等服务。

2.3.1 LVS 集群的通用结构

LVS 集群采用 IP 负载均衡技术和基于内容请求分发技术。调度器具有很好的吞吐率，将请求均衡地转移到不同的服务器上执行，且调度器自动屏蔽掉服务器的故障，从而将一组服务器构成一个高性能的、高可用的虚拟服务器。

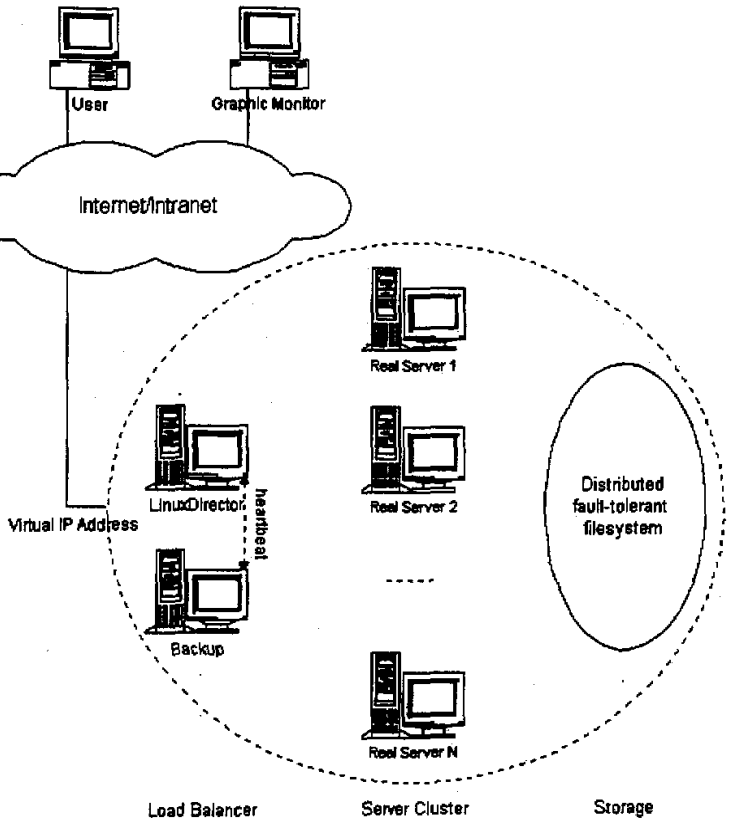

图 2.1：LVS 集群的体系结构

为此，在设计时需要考虑系统的透明性、可伸缩性、高可用性和易管理性。LVS 集群的体系结构如图 2.1 所示，它有三个主要组成部分：

- 负载调度器（load balancer），它是整个集群对外面的前端机，负责将客户的请求发送到一组服务器上执行，而客户认为服务是来自一个 IP 地址上的。它可以是用 IP 负载均衡技术的负载调度器，也可以是基于内容请求分发的负载调度器，还可以是两者的结合。
- 服务器池（server pool），是一组真正执行客户请求的服务器，执行的服务有 WEB、MAIL、FTP 和 DNS 等。
- 后端存储（backend storage），它为服务器池提供一个共享的存储区，这样很容易使得服务器池拥有相同的内容，提供相同的服务。

调度器采用 IP 负载均衡技术、基于内容请求分发技术或者两者相结合。在 IP 负载均衡技术中，需要服务器池拥有相同的内容提供相同的服务。当客户请求到达时，调度器只根据负载情况从服务器池中选出一个服务器，将该请求转发到选出的服务器，并记录这个调度；当这个请求的其他报文到达，也会被转发到前面选出的服务器。在基于内容请求分发技术中，服务器可以提供不同的服务，当客户请求到达时，调度器可根据请求的内容和服务器的情况选择服务器执行请求。因为所有的

操作都是在操作系统核心空间中将完成的，它的调度开销很小，所以它具有很高的吞吐率。

服务器池的结点数目是可变的。当整个系统收到的负载超过目前所有结点的处理能力时，可以在服务器池中增加服务器来满足不断增长的请求负载。对大多数网络服务来说，结点与结点间不存在很强的相关性，所以整个系统的性能可以随着服务器池的结点数目增加而线性增长。

后端存储通常用容错的分布式文件系统，如 AFS[41, 42]、GFS[43, 44, 45, 46]、Coda[46, 47, 48, 49, 50, 51]和 Intermezzo[52]等。分布式文件系统为各服务器提供共享的存储区，它们访问分布式文件系统就像访问本地文件系统一样。同时，分布式文件系统提供良好的伸缩性和可用性。然而，当不同服务器上的应用程序同时访问分布式文件系统上同一资源时，应用程序的访问冲突需要消解才能使得资源处于一致状态。这需要一个分布式锁管理器（Distributed Lock Manager），它可能是分布式文件系统内部提供的，也可能是外部的。开发者在写应用程序时，可以使用分布式锁管理器来保证应用程序在不同结点上并发访问的一致性。

负载调度器、服务器池和分布式文件系统通过高速网络相连，如 100Mbps 交换机、Myrinet[53]、CompactNET[54]和 Gigabit 交换机等。使用高速的网络，主要为避免当系统规模扩大时互联网络成为瓶颈。

Graphic Monitor 是为系统管理员提供整个集群系统的监视器，它可以监视系统中每个结点的状况。Graphic Monitor 是基于浏览器的，所以无论管理员在本地还是异地都可以监测系统的状况。为了安全的原因，浏览器要通过 HTTPS（Secure HTTP）协议和身份认证后，才能进行系统监测，并进行系统的配置和管理。

2.3.1.1 为什么使用层次的体系结构

层次的体系结构可以使得层与层之间相互独立，允许在一个层次的已有软件在不同的系统中被重用。例如，调度器层提供了负载平衡、可伸缩性和高可用性等，在服务器层可以运行不同的网络服务，如 Web、Cache、Mail 和 Media 等，来提供不同的可伸缩网络服务。

2.3.1.2 为什么是共享存储

共享存储如分布式文件系统在这个 LVS 集群系统是可选项。当网络服务需要有相同的内容，共享存储是很好的选择，否则每台服务器需要将相同的内容复制到本地硬盘上。当系统存储的内容越多，这种不共享结构（Shared-nothing Structure）的代价越大，因为每台服务器需要一样大的存储空间，所有的更新需要涉及到每台服务器，系统的维护代价也很高[55]。

共享存储为服务器组提供统一的存储空间，这使得系统的维护工作比较轻松，如 Webmaster 只需要更新共享存储中的页面，对所有的服务器都有效。分布式文件

系统提供良好的伸缩性和可用性，当分布式文件系统的存储空间增加时，所有服务器的存储空间也随之增大。对于大多数 Internet 服务来说，它们都是读密集型（Read-intensive）的应用，分布式文件系统在每台服务器使用本地硬盘作 Cache（如 2Gbytes 的空间），可以使得访问分布式文件系统本地的速度接近于访问本地硬盘。

2.3.1.3 高可用性

集群系统的特点是它在软硬件上都有冗余。系统的高可用性可以通过检测节点或服务进程故障和正确地重置系统来实现，使得系统收到的请求能被存活的结点处理。通常，我们在调度器上有资源监视进程来时刻监视各个服务器结点的健康状况，当服务器对 ICMP ping 不可达时或者她的网络服务在指定的时间没有响应时，资源监视进程通知操作系统内核将该服务器从调度列表中删除或者失效。这样，新的服务请求就不会被调度到坏的结点。资源监测程序能通过电子邮件或传呼机向管理员报告故障，一旦监测到服务进程恢复工作，通知调度器将其加入调度列表进行调度。另外，通过系统提供的管理程序，管理员可发命令随时将一台机器加入服务或切出服务，很方便进行系统维护。

现在前端的调度器有可能成为系统的单一失效点。为了避免调度器失效导致整个系统不能工作，我们需要设立调度器的备份。两个心跳进程（Heartbeat Daemon）[56]分别在主、从调度器上运行，它们通过串口线和 UDP 等心跳线来相互汇报各自的健康情况。当从调度器不能听得主调度器的心跳时，从调度器会接管主调度器的工作来提供负载调度服务。这里，一般通过 ARP 欺骗（Gratuitous ARP）[57]来接管集群的 Virtual IP Address。当主调度器恢复时，这里有两种方法，一是主调度器自动变成从调度器，二是从调度器释放 Virtual IP Address，主调度器收回 Virtual IP Address 并提供负载调度服务。然而，当主调度器故障后或者接管后，会导致已有的调度信息丢失，这需要客户程序重新发送请求。

2.3.2 可伸缩 Web 和媒体服务

基于 LVS 可伸缩 Web 和媒体服务的体系结构如图 2.2 所示：在前端是一个负载调度器，一般采用 IP 负载均衡技术来获得整个系统的高吞吐率；在第二层是服务器池，Web 服务和媒体服务分别运行在每个结点上；第三层是数据存储，通过分布式文件系统使得每个服务器结点共享相同的数据。集群中结点间是通过高速网络相连的。

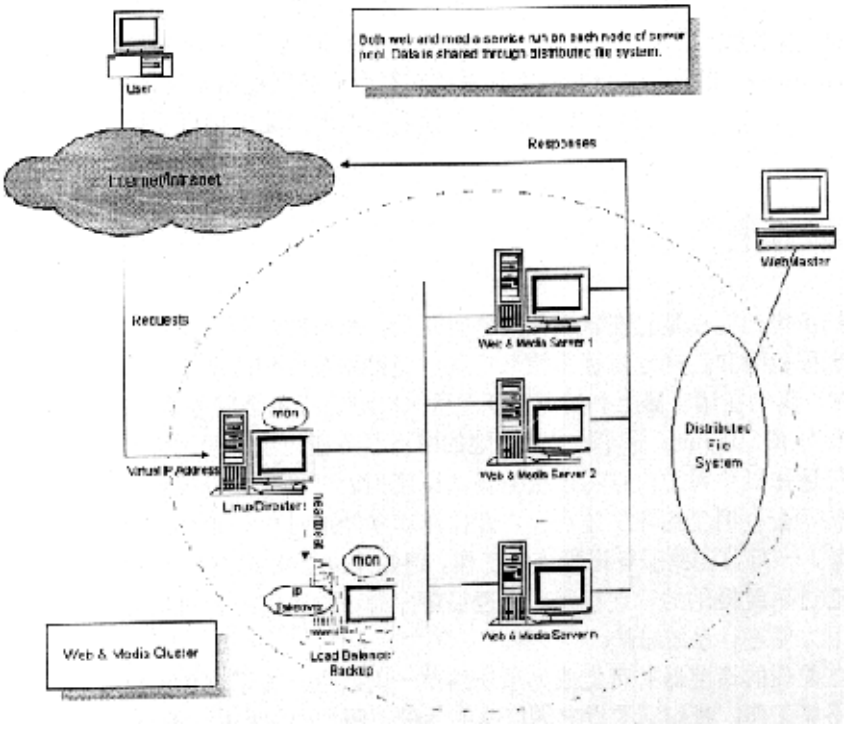

图 2.2：基于 LVS 的可伸缩 Web 和媒体集群

分布式文件系统提供统一的存储空间，这使得系统的维护工作比较方便，且系统运行比较高效。当所有服务器结点超载时，管理员可以很快地加入新的结点来处理请求，而无需将 Web 文档等复制到结点的本地硬盘上。Webmaster 可以看到统一的文档存储空间，维护和更新页面比较方便，对分布式文件系统中页面的修改对所有的服务器都有效。大的媒体文件（如视频文件）分段存储在分布式文件系统的多个结点上，可以提高文件系统的性能和文件服务器间的负载均衡。

IP 负载调度器（即 VS/DR 方法，将在下一章详细叙述）可以分别将 Web 服务和媒体服务负载均衡地分发到各个服务器上，服务器将响应数据直接返回给客户，这样可以极大地提高系统的吞吐率。

Real 公司以其高压缩比的音频视频格式和音频视频播放器 RealPlayer 而闻名。Real 公司正在使用以上结构将由 20 台服务器组成的 LVS 可伸缩 Web 和媒体集群，为其全球用户提供 Web 和音频视频服务。Real 公司的高级技术主管声称 LVS 击败所有他们尝试的商品化负载均衡产品[58]。

2.3.3 可伸缩 Cache 服务

有效的网络 Cache 系统可以大大地减少网络流量、降低响应延时以及服务器的负载。但是，若 Cache 服务器超载而不能及时地处理请求，反而会增加响应延时。

所以，Cache 服务的可伸缩性很重要，当系统负载不断增长时，整个系统能被扩展来提高 Cache 服务的处理能力。尤其，在主干上的 Cache 服务可能需要几个 Gbps 的吞吐率，单台服务器（如 SUN 目前最高端的 Enterprise 10000 服务器）远不能达到这个吞吐率。可见，通过 PC 服务器集群实现可伸缩 Cache 服务是很有效的方法，也是性能价格比最高的方法。

基于 LVS 可伸缩 Cache 集群的体系结构如图 2.3 所示：在前端是一个负载调度器，一般采用 IP 负载均衡技术来获得整个系统的高吞吐率；在第二层是 Cache 服务器池，一般 Cache 服务器放置在接近主干 Internet 连接处，它们可以分布在不同的网络中。调度器可以有多个，放在离客户接近的地方，可实现透明的 Cache 服务。

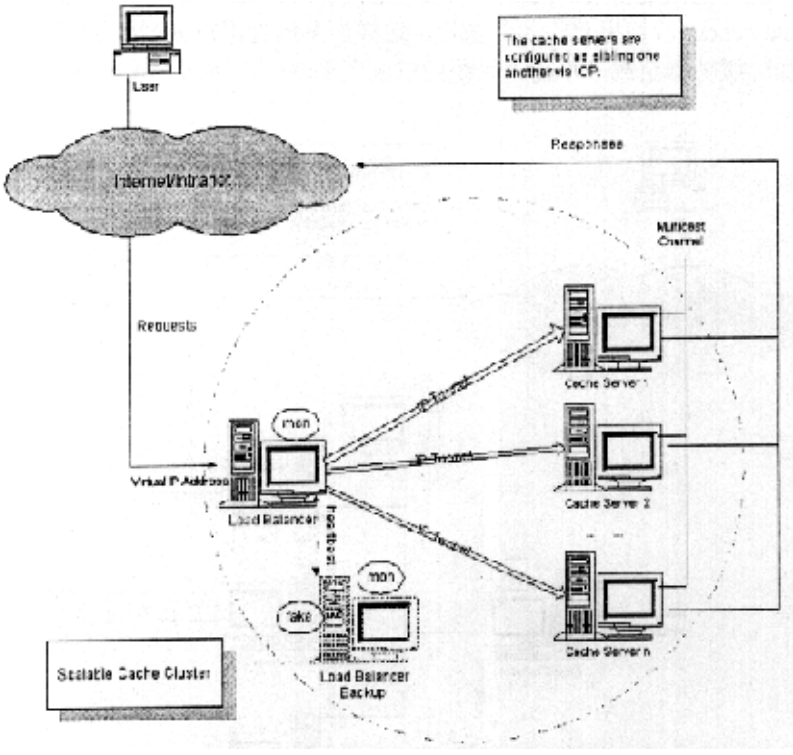

图 2.3：基于 LVS 的可伸缩 Cache 集群

Cache 服务器采用本地硬盘来存储可缓存的对象，因为存储可缓存的对象是写操作，且占有一定的比例，通过本地硬盘可以提高 I/O 的访问速度。Cache 服务器间有专用的多播通道，通过 ICP 协议（Internet Cache Protocol）来交互信息。当一台 Cache 服务器在本地硬盘中未命中当前请求时，它可以通过 ICP 查询其他 Cache 服务器是否有请求对象的副本，若存在，则从邻近的 Cache 服务器取该对象的副本，这样可以进一步提高 Cache 服务的命中率。

为 150 多所大学和地区服务的英国国家 Cache 网在 1999 年 11 月用以上 LVS 结构实现可伸缩的 Cache 集群[59]，只用了原有 50 多台相互独立 Cache 服务器的一半，用户反映网络速度跟夏天一样快（学生放暑假）。可见，通过负载调度可以摸

平单台服务器访问的毛刺（Burst），提高整个系统的资源利用率。

2.3.4 可伸缩邮件服务

随着 Internet 用户不断增长，很多 ISP 面临他们邮件服务器超载的问题。当邮件服务器不能容纳更多的用户帐号时，有些 ISP 买更高档的服务器来代替原有的，将原有服务器的信息（如用户邮件）迁移到新服务器是很繁琐的工作，会造成服务的中断；有些 ISP 设置新的服务器和新的邮件域名，新的邮件用户放置在新的服务器上，如上海电信现在用不同的邮件服务器 public1.sta.net.cn、public2.sta.net.cn 到 public9.sta.net.cn 放置用户的邮件帐号，这样静态地将用户分割到不同的服务器上，会造成邮件服务器负载不平衡，系统的资源利用率低，对用户来说邮件的地址比较难记。

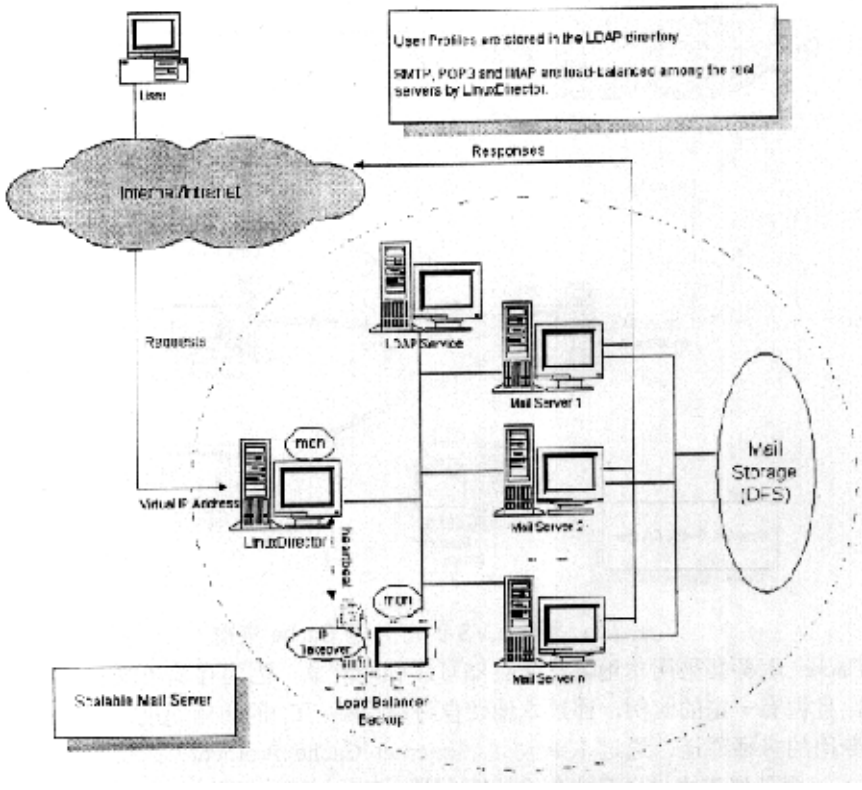

图 2.4：基于 LVS 的可伸缩邮件集群

可以利用 LVS 框架实现可伸缩邮件服务。它的体系结构如图 2.4 所示：在前端是一个采用 IP 负载均衡技术的负载调度器；在第二层是服务器池，有 LDAP（Light-weight Directory Access Protocol）服务器和一组邮件服务器。第三层是数据存储，通过分布式文件系统来存储用户的邮件。集群中结点间是通过高速网络相连的。

用户的信息如用户名、口令、主目录和邮件容量限额等存储在 LDAP 服务器中，可以通过 HTTPS 让管理员进行用户管理。在各个邮件服务器上运行 SMTP（Simple Mail Transfer Protocol）[60]、POP3（Post Office Protocol version 3）[61]、IMAP4（Internet Message Access Protocol version 4）[62]和 HTTP 服务。SMTP 接受和转发用户的邮件，SMTP 服务进程查询 LDAP 服务器获得用户信息，再存储邮件。POP3 和 IMAP4 通过 LDAP 服务器获得用户信息，口令验证后，处理用户的邮件访问请求。SMTP、POP3 和 IMAP4 服务进程需要有机制避免用户邮件的读写冲突。HTTP 服务是让用户通过浏览器可以访问邮件。负载调度器将四种服务请求负载均衡地调度到各个服务器上。

系统中可能的瓶颈是 LDAP 服务器，对 LDAP 服务中 B+树的参数进行优化，再结合高端的服务器，可以获得较高的性能。若分布式文件系统没有多个存储结点间的负载均衡机制，则需要相应的邮件迁移机制来避免邮件访问的倾斜。

这样，这个集群系统对用户来说就像一个高性能、高可靠的邮件服务器（上海电信只要用一个邮件域名 public.sta.net.cn）。当邮件用户不断增长时，只要在集群中增加服务器结点和存储结点。

§2.4 地理分布 LVS 集群的体系结构

由于互联网用户分布在世界各地，通过地理分布的服务器让用户访问就近的服务器，来节省网络流量和提高响应速度。以下，我们提出地理分布的 LVS 集群系统，通过 BGP 路由插入使得用户访问离他们最近的服务器集群，并提供服务器集群之间的负载平衡。

2.4.1 体系结构

地理分布 LVS 集群的体系结构如图 2.5 所示：有三个 LVS 集群系统分布在 Internet 上，他们一般放置在不同区域的 Internet 数据中心（Internet Data Center）中，例如他们分别放在中国、美国和德国的三个不同的 IDC 中。三个 LVS 集群系统都有自己的分布式文件系统，它们的内容是相互复制的，提供相同的网络服务。它们共享一个 Virtual IP Address 来提供网络服务。当用户通过 Virtual IP Address 访问网络服务，离用户最近的 LVS 集群提供服务。例如，中国的用户访问在中国的 LVS 集群系统，美国的用户使用美国的 LVS 集群系统，这一切对用户来说是透明的。

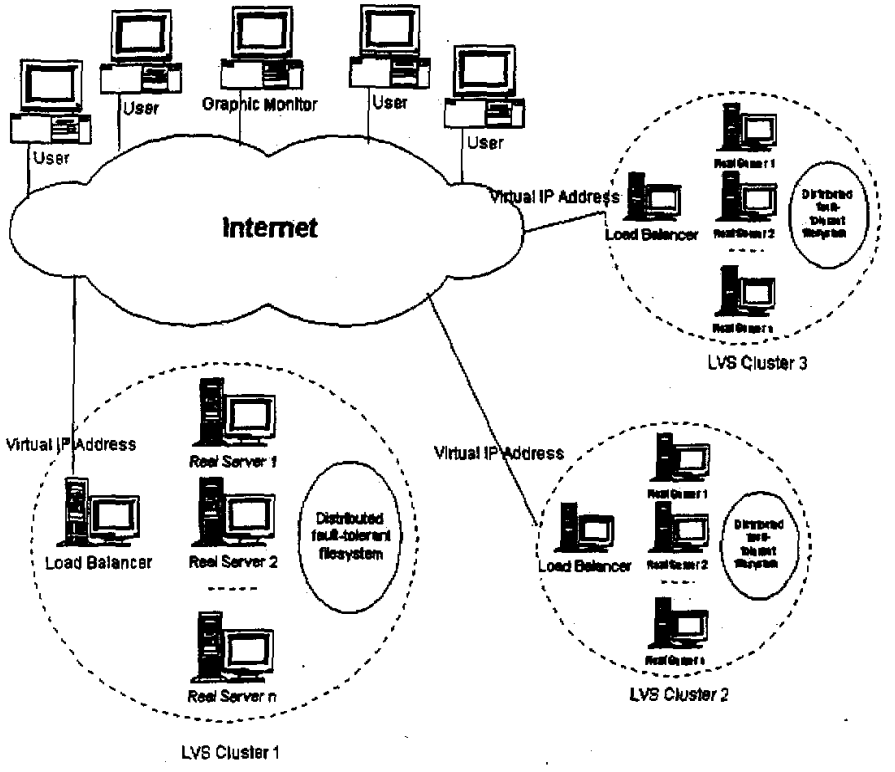

图 2.5：地理分布 LVS 集群的体系结构

地理分布 LVS 集群系统可以带来以下好处：

- 使得用户访问离他们最近的系统，对用户来说体验到更快的响应速度，对服务提供商来说节约网络带宽，降低成本。
- 避免灾难导致系统中止服务。当一个地点发生地震、火灾等使得系统或者网络连接瘫痪时，所有的用户访问可以很快由其他地点的 LVS 集群来提供。除了已建立的连接中断以外，这一切对用户来说都是透明的。

2.4.2 基于 BGP 的地理分布服务器集群调度

BGP（Border Gateway Protocol）[63]是用于自治系统（Autonomous Systems）之间交换路由信息的协议，BGP 可以设置路由策略，如政策、安全和经济上的考虑。

我们可以利用 BGP 协议在 Internet 的 BGP 路由器插入到 Virtual IP Address 的路由信息。在不同区域的 LVS 集群向它附近的 BGP 路由器广播到 Virtual IP Address 的路由信息，这样就存在多条到 Virtual IP Address 的路径，Internet 的 BGP 路由器会根据评价函数选出最近的一条路径。这样，我们可以使得用户访问离他们最近的 LVS 集群。当一个 LVS 集群系统失效时，它的路由信息自然不会在 Internet 的 BGP 路由器中交换，BGP 路由器会选择其他到 Virtual IP Address 的路径。这样，可以做到抗灾害性（Disaster Tolerance）。

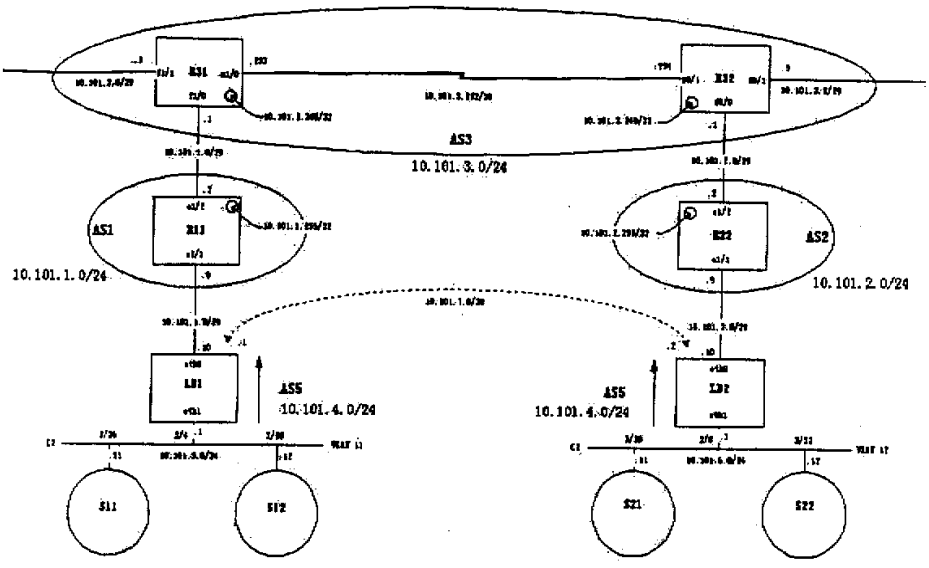

图 2.6：基于 BGP 的地理分布服务器集群调度例子

下面我们举一个基于 BGP 的地理分布服务器集群调度例子，它也是我们在实验室中测试的。测试的例子如图 2.6 所示，R11、R22 分别在不同自治系统 AS1 和 AS2 中的 Internet 服务提供商（Internet Service Provider），R31 和 R32 表示在自治系统 AS3 中两个 ISP。两个 LVS 集群系统分别放置在两个不同数据中心 IDC1 和 IDC2 中，LB1 和 LB2 分别是两个 LVS 集群系统的负载调度器，它们对外提供网络服务的 IP 地址为 10.101.4.1。在第一个集群中，请求被调度到服务器 10.101.5.11 和 10.101.5.12 执行。在第二个集群中，请求被调度到服务器 10.101.6.11 和 10.101.6.12。10.101.4.1 是在自治系统 AS5 的网络 10.101.4.0/24 中。LB1 上的 BGP 服务进程将网络 10.101.4.0/24 的路由信息广播到邻近的 AS1，LB2 上的 BGP 服务进程将网络 10.101.4.0/24 的路由信息广播到邻近的 AS2 中。

我们在 R31 端口 10.101.3.1 相连的 BGP 路由器上查到 AS5 的自治系统路径为 3→1→5。在与 R32 端口 10.101.3.9 相连的 BGP 路由器上查到 AS5 的自治系统路径为 3→2→5，并在该路由器上访问 10.101.4.1 上提供的 Web 服务，是由服务器 10.101.6.11 处理的。当我们关掉 LB2 后，在该 BGP 路由器上查到 AS5 的自治系统路径变为 3→1→5。

2.4.3 服务器集群间的负载均衡

通过 BGP 插入路由信息的方法可以使得用户访问邻近的服务器集群，但是用户访问存在突发性，在某个区域的访问高峰可能会导致该区域的服务器集群系统超载，而其他服务器集群系统可能处于低负载状态，这时与其将请求在超载系统上排队等候，不如将请求送到远处的低负载系统上执行，可以提高响应速度。例如，中国用户在白天时间的一段访问高峰使得在中国的镜像服务器集群系统超载，而此时

美国是晚上时间其镜像服务器集群系统处于低负载状态。

我们提出通过 IP 隧道的方法将请求从一个调度器转发到另一个调度器，再调度到真实服务器上执行。在下一章中将详细描述如何通过 IP 隧道作 IP 负载均衡调度。在通过 IP 隧道转发请求前，各个服务器集群需要定时交换负载信息（如 2 分钟交换一次），当确信远处的集群系统处于低负载状态，再转发请求。例如，当本地集群系统的综合负载大于 1.1 和远处集群系统的负载小于 0.7 时，调度器通过 IP 隧道将新的请求转发到远处的集群系统。若远处集群系统的负载超过 0.7 时，停止转发请求。当本地集群系统的负载降至 1.0 时，也停止转发请求，由本地服务器处理。这样，基本上可以避免两个调度器间相互转发一个请求。即使两个调度器间相互转发一个请求报文的例外情况发生，报文的 TTL 会降到零，报文被丢掉。

§2.5 本章小结

我们分析了现在和将来网络服务的需求，提出可伸缩网络服务的体系结构，分为负载调度器、服务器池和后端存储三层结构。负载调度器采用 IP 负载均衡技术和基于内容请求分发技术。它的实现将在 Linux 操作系统进行，将一组服务器组成一个高可伸缩的、高可用的服务器，故称之为 Linux Virtual Server。它提供了负载平衡、可伸缩性和高可用性，可以应用于建立很多可伸缩网络服务，如 Web、Cache、Mail 和 Media 等服务。

在此基础上，我们还提出地理分布的 LVS 集群系统，通过 BGP 插入路由信息的方法可以使得用户访问邻近的服务器集群，通过 IP 隧道实现服务器集群间的负载均衡，进一步提高响应速度。地理分布的 LVS 集群系统可以节约网络带宽，改善网络服务质量，有很好的抗灾害性。

第三章 IP 负载均衡技术

上一章讲述了可伸缩网络服务的几种结构，它们都需要一个前端调度器。在调度器的实现技术中，IP 负载均衡技术是效率最高的。在已有的 IP 负载均衡技术中有通过网络地址转换（Network Address Translation）将一组服务器构成一个高性能的、高可用的虚拟服务器，我们称之为 VS/NAT 技术（Virtual Server via Network Address Translation），大多数商品化的 IP 负载均衡调度器产品都是使用此方法，如 Cisco 的 LocalDirector、F5 的 Big/IP 和 Alteon 的 ACEDirector。在分析 VS/NAT 的缺点和网络服务的非对称性的基础上，我们提出通过 IP 隧道实现虚拟服务器的方法 VS/TUN（Virtual Server via IP Tunneling），和通过直接路由实现虚拟服务器的方法 VS/DR（Virtual Server via Direct Routing），它们可以极大地提高系统的伸缩性。

本章节将描述三种 IP 负载均衡技术 VS/NAT、VS/TUN 和 VS/DR 的工作原理，以及它们的优缺点。在以下描述中，我们称客户的 socket 和服务器的 socket 之间的数据通讯为连接，无论它们是使用 TCP 还是 UDP 协议。

§3.1 Virtual Server via Network Address Translation

由于 IPv4 中 IP 地址空间的日益紧张和安全方面的原因，很多网络使用保留 IP 地址（10.0.0.0/255.0.0.0、172.16.0.0/255.128.0.0 和 192.168.0.0/255.255.0.0）[64, 65, 66]。这些地址不在 Internet 上使用，而是专门为内部网络预留的。当内部网络中的主机要访问 Internet 或被 Internet 访问时，就需要采用网络地址转换（Network Address Translation，以下简称 NAT），将内部地址转化为 Internets 上可用的外部地址。NAT 的工作原理是报文头（目标地址、源地址和端口等）被正确改写后，客户相信它们连接一个 IP 地址，而不同 IP 地址的服务器组也认为它们是与客户直接相连的。由此，可以用 NAT 方法将不同 IP 地址的并行网络服务变成在一个 IP 地址上的一个虚拟服务。

VS/NAT 的体系结构如图 3.1 所示。在一组服务器前有一个调度器，它们是通过 Switch/HUB 相连接的。这些服务器提供相同的网络服务、相同的内容，即不管请求被发送到哪一台服务器，执行结果是一样的。服务的内容可以复制到每台服务器的本地硬盘上，可以通过网络文件系统（如 NFS）共享，也可以通过一个分布式文件系统来提供。

VS/DR 负载调度器也只处于从客户到服务器的半连接中，按照半连接的 TCP 有限状态机进行状态迁移。

§3.4 三种方法的优缺点比较

三种 IP 负载均衡技术的优缺点归纳在下表中：

	VS/NAT	VS/TUN	VS/DR
Server	any	Tunneling	Non-arp device
server network	private	LAN/WAN	LAN
server number	low (10~20)	High (100)	High (100)
server gateway	load balancer	own router	Own router

注：以上三种方法所能支持最大服务器数目的估计是假设调度器使用 100M 网卡，调度器的硬件配置与后端服务器的硬件配置相同，而且是对一般 Web 服务。使用更高的硬件配置（如千兆网卡和更快的处理器）作为调度器，调度器所能调度的服务器数量会相应增加。当应用不同时，服务器的数目也会相应地改变。所以，以上数据估计主要是为三种方法的伸缩性进行量化比较。

3.4.1 Virtual Server via NAT

VS/NAT 的优点是服务器可以运行任何支持 TCP/IP 的操作系统，它只需要一个 IP 地址配置在调度器上，服务器组可以用私有的 IP 地址。缺点是它的伸缩能力有限，当服务器结点数目升到 20 时，调度器本身有可能成为系统的新瓶颈，因为在 VS/NAT 中请求和响应报文都需要通过负载调度器。 我们在 Pentium 166 处理器的主机上测得重写报文的平均延时为 60us，性能更高的处理器上延时会短一些。假设 TCP 报文的平均长度为 536 Bytes，则调度器的最大吞吐量为 8.93 MBytes/s. 我们再假设每台服务器的吞吐量为 600KBytes/s，这样一个调度器可以带动 16 台服务器。

基于 VS/NAT 的的集群系统可以适合许多服务器的性能要求。如果负载调度器成为系统新的瓶颈，可以有三种方法解决这个问题：混合方法、VS/TUN 和 VS/DR。在 DNS 混合集群系统中，有若干个 VS/NAT 负载调度器，每个负载调度器带自己的服务器集群，同时这些负载调度器又通过 RR-DNS 组成简单的域名。但 VS/TUN 和 VS/DR 是提高系统吞吐量的更好方法。

对于那些将 IP 地址或者端口号在报文数据中传送的网络服务，需要编写相应的应用模块来转换报文数据中的 IP 地址或者端口号。这会带来实现的工作量，同时应用模块检查报文的开销会降低系统的吞吐率。

3.4.2 Virtual Server via IP Tunneling

在 VS/TUN 的集群系统中，负载调度器只将请求调度到不同的后端服务器，后

端服务器将应答的数据直接返回给用户。这样，负载调度器就可以处理大量的请求，它甚至可以调度百台以上的服务器（同等规模的服务器），而它不会成为系统的瓶颈。即使负载调度器只有 100Mbps 的全双工网卡，整个系统的最大吞吐量可超过 1Gbps。所以，VS/TUN 可以极大地增加负载调度器调度的服务器数量。VS/TUN 调度器可以调度上百台服务器，而它本身不会成为系统的瓶颈，可以用来构建高性能的超级服务器。

VS/TUN 技术对服务器有要求，即所有的服务器必须支持“IP Tunneling”或者“IP Encapsulation”协议。目前，VS/TUN 的后端服务器主要运行 Linux 操作系统。因为“IP Tunneling”正成为各个操作系统的标准协议，所以 VS/TUN 也会适用运行其他操作系统的后端服务器。

3.4.3 Virtual Server via Direct Routing

跟 VS/TUN 方法一样，VS/DR 调度器只处理客户到服务器端的连接，响应数据可以直接从独立的网络路由返回给客户。这可以极大地提高 LVS 集群系统的伸缩性。

跟 VS/TUN 相比，这种方法没有 IP 隧道的开销，但是要求负载调度器与实际服务器都有一块网卡连在同一物理网段上，服务器网络设备（或者设备别名）不作 ARP 响应，或者能将报文重定向（Redirect）到本地的 Socket 端口上。

§3.5 本章小结

本章主要讲述了可伸缩网络服务 LVS 框架中的三种 IP 负载均衡技术。在分析网络地址转换方法（VS/NAT）的缺点和网络服务的非对称性的基础上，我们提出通过 IP 隧道实现虚拟服务器的方法 VS/TUN，和通过直接路由实现虚拟服务器的方法 VS/DR，极大地提高了系统的伸缩性。

第四章 负载调度

本章讲述 IPVS 在内核中的各种连接调度算法。针对请求的服务时间变化很大，我们提出一个动态反馈负载均衡算法，结合内核中的加权调度算法，根据动态反馈回来的负载信息来调整服务器的权值，来避免服务器间的负载不平衡。

§4.1 内核中的连接调度算法

IPVS 在内核中的负载均衡调度是以连接为粒度的。在 HTTP 协议（非持久）[72]中，每个对象从 WEB 服务器上获取都需要建立一个 TCP 连接，同一用户的不同请求会被调度到不同的服务器上，所以这种细粒度的调度可以避免单个用户访问的突发性引起的负载不平衡。

在调度算法上，我们已实现了以下五种调度算法：

- 轮叫调度（Round-Robin Scheduling）
- 加权轮叫调度（Weighted Round-Robin Scheduling）
- 最小连接调度（Least-Connection Scheduling）
- 加权最小连接调度（Weighted Least-Connection Scheduling）
- 全球服务器调度（Global Server Scheduling）

4.1.1 轮叫调度

轮叫调度算法就是以轮叫的方式依次将请求调度不同的服务器。它假设所有服务器处理性能均相同，不管服务器的当前连接数和响应时间。该算法简单，不适用于服务器组中处理性能不一的情况，而且当请求服务时间变化比较大时，轮叫调度算法容易导致服务器间的负载不平衡。虽然轮叫 DNS 方法也是以轮叫的方式将一个域名解析到多个 IP 地址，但轮叫 DNS 方法的调度粒度是基于每个域名服务器的，域名服务器对域名解析的缓存会妨碍轮叫解析域名生效，这会导致服务器间负载的严重不平衡。这里，轮叫调度算法的粒度是基于每个连接的，同一用户的不同连接都会被调度到不同的服务器上，所以这种细粒度的轮叫调度要比 DNS 的轮叫调度优越很多。

4.1.2 加权轮叫调度

加权轮叫调度算法可以解决服务器间性能不一的情况，它用相应的权值表示服务器的处理性能，服务器的缺省权值为 1。加权轮叫调度算法是按权值的高低和轮

叫方式分配请求到各服务器。权值高的服务器先收到的连接，权值高的服务器比权值低的服务器处理更多的连接，相同权值的服务器处理相同数目的连接数。加权轮叫调度算法如下：

假设有一组服务器 $S = \{S_0, S_1, \ldots, S_{n-1}\}$，一个指示变量 i 表示上一次选择的服务器，变量 cw 表示当前的权值。变量 i 和 cw 最初都被初始化为零。当所有服务器的权值为零，即 $W(S_i)=0$，则所有的新连接都会被丢掉。

```
while (true) {
  if (i == 0) {
    cw = cw - 1;
    if (cw <= 0) {
      set cw the maximum weight of S;
      if (cw == 0)
        return NULL;
    }
  } else i = (i + 1) mod n;
  if (W(Si) >= cw)
    return Si;
}
```

例如，有三个服务器 A、B 和 C 分别有权值 4、3 和 2，则在一个调度周期内(mod sum(W_i))调度序列为 AABABCABC。加权轮叫调度算法还是比较简单和高效。当请求的服务时间变化很大，单独的加权轮叫调度算法依然会导致服务器间的负载不平衡。

4.1.3 最小连接调度

最小连接调度是把新的连接请求分配到当前连接数最小的服务器。最小连接调度是一种动态调度算法，它通过服务器当前所活跃的连接数来估计服务器的负载情况。调度器需要记录各个服务器已建立连接的数目，当一个请求被调度到某台服务器，其连接数加 1；当连接中止或超时，其连接数减一。

当各个服务器有相同的处理性能时，最小连接调度能把负载变化大的请求分布平滑到各个服务器上，所有处理时间比较长的请求不可能被发送到同一台服务器上。但是，当各个服务器的处理能力不同时，该算法并不理想，因为 TCP 连接处理请求后会进入 TIME_WAIT 状态，TCP 的 TIME_WAIT 一般为 2 分钟，此时连接还占用服务器的资源，所以会出现这样情形，性能高的服务器已处理所收到的连接，连接处于 TIME_WAIT 状态，而性能低的服务器已经忙于处理所收到的连接，还不断地收到新的连接请求。

4.1.4 加权最小连接调度

加权最小连接调度是最小连接调度的超集，各个服务器用相应的权值表示其处理性能。服务器的缺省权值为 1，系统管理员可以动态地设置服务器的权值。加权最小连接调度在调度新连接时尽可能使服务器的已建立连接数和其权值成比例。加权最小连接调度的算法如下：

假设有一组服务器 $S = \{S_0, S_1, \ldots, S_{n-1}\}$，每台服务器的权值为 $W_i(i=0, 1, .., n-1)$，已建立的连接数目为 $C_i(i=0, 1, .., n-1)$，服务器当前连接数的总和为 $S = \Sigma C_i$ $(i=0, 1, .., n-1)$。当前的新连接请求会被发送服务器 j，服务器 j 满足以下条件

$$(C_i / S) / W_j = \min \{ (C_i / S) / W_i \} \quad (i=0, 1, .., n-1)$$

因为 S 在这一轮查找中是个常数，所以判断条件可以优化为

$$C_j / W_j = \min \{ C_i / W_i \} \quad (i=0, 1, .., n-1)$$

因为除法所需的 CPU 周期比乘法多，在 Linux 内核中为提高处理速度不允许浮点除法，服务器的权值都大于零，所以判断条件 C_j / W_j 〈 C_i / W_i 可以进一步优化为 $C_j * W_i$ 〈 $C_i * W_j$ 。

同时保证服务器的权值为零时，服务器不被调度。算法的流程如下：

```
for (j=0; j<n; j++) {
  if (W_j > 0)
    break;
}
for (i=j+1; i<n; j++) {
  if (C_j * W_i > C_i * W_j)
    j = i;
}
if (j < n) return S_j;
else return NULL;
```

4.1.5 全球服务器调度

全球服务器调度（Global Server Scheduling）是根据客户位置和服务器的负载情况将连接调度到地理分布的服务器。由于多数 Internet 服务的请求与响应非对称性，若通过 VS/TUN 的方法将请求调度到离客户最近的服务器，响应直接从该服务器返回给客户，这种调度仍然是十分有效的。

在 IP 层进行调度时，所能获得的信息只有客户的 IP 地址和服务器的相应负载指标，所以我们根据 IP 地址分配的情况在操作系统的内核中组成一张表，将 IP 地址和处理来自该 IP 地址的服务器序列关联起来。当一个客户的连接请求到达时，调度器可以获得客户的 IP 地址，查询该表获得处理该客户请求的服务器序列，即

离客户较近的服务器序列，再通过 IP 隧道方法将连接调度到该序列中负载最轻的服务器上。

§4.2 动态反馈负载均衡算法

动态反馈负载均衡算法考虑服务器的实时负载和响应情况，不断调整服务器间处理请求的比例，来避免服务器超载时依然收到大量请求，从而提高整个系统的吞吐率。图 4.1 显示了该算法的工作环境，在调度器 LinuxDirector 上运行 Monitor Daemon，Monitor Daemon 来监视和收集各个服务器的负载信息。Monitor Daemon 可根据多个负载信息算出一个综合负载值。Monitor Daemon 将各个服务器的综合负载值和当前权值算出一组新的权值，若新权值和当前权值的差值大于设定的阀值，Monitor Daemon 将该服务器的权值设置到内核中的 IPVS 调度中，而在内核中连接调度一般采用基于权值轮叫算法。

4.2.1 连接调度

当客户通过 TCP 连接访问网络访问时，服务所需的时间和所要消耗的计算资源是千差万别的，它依赖于很多因素。例如，它依赖于请求的服务类型、当前网络带宽的情况、以及当前服务器资源利用的情况。一些负载比较重的请求需要进行计算密集的查询、数据库访问、很长响应数据流；而负载比较轻的请求往往只需要读一个 HTML 页面或者进行很简单的计算。

请求处理时间的千差万别可能会导致服务器利用的倾斜（Skew），即服务器间的负载不平衡。例如，有一个 WEB 页面有 A、B、C 和 D 文件，其中 D 是大图像文件，浏览器需要建立四个连接来取这些文件。当多个用户通过浏览器同时访问该页面时，最极端的情况是所有 D 文件的请求被发到同一台服务器。所以说，有可能存在这样情况，有些服务器已经超负荷运行，而其他服务器基本是闲置着。同时，有些服务器已经忙不过来，有很长的请求队列，还不断地收到新的请求。反过来说，这会导致客户长时间的等待，觉得系统的服务质量差。

4.2.1.1 简单连接调度

简单连接调度会使得服务器倾斜的发生。在上面的例子中，若采用轮叫调度算法，且集群中正好有四台服务器，必有一台服务器总是收到 D 文件的请求。这种调度策略会导致整个系统资源的低利用率，因为有些资源被用尽导致长时间的等待，而其他资源空闲着。

4.2.1.2 实际 TCP/IP 流量的特征

文献[73]说明网络流量是呈波浪型发生的，在一段较长时间的小流量后，会有一段大流量的访问，然后是小流量，这样跟波浪一样周期性地发生。文献[74,75,76,77]揭示在 WAN 和 LAN 上网络流量存在自相似的特征，在 WEB 访问流也存在自相似性。这就需要一个动态反馈机制，利用服务器组的状态来应对访问流的自相似性。

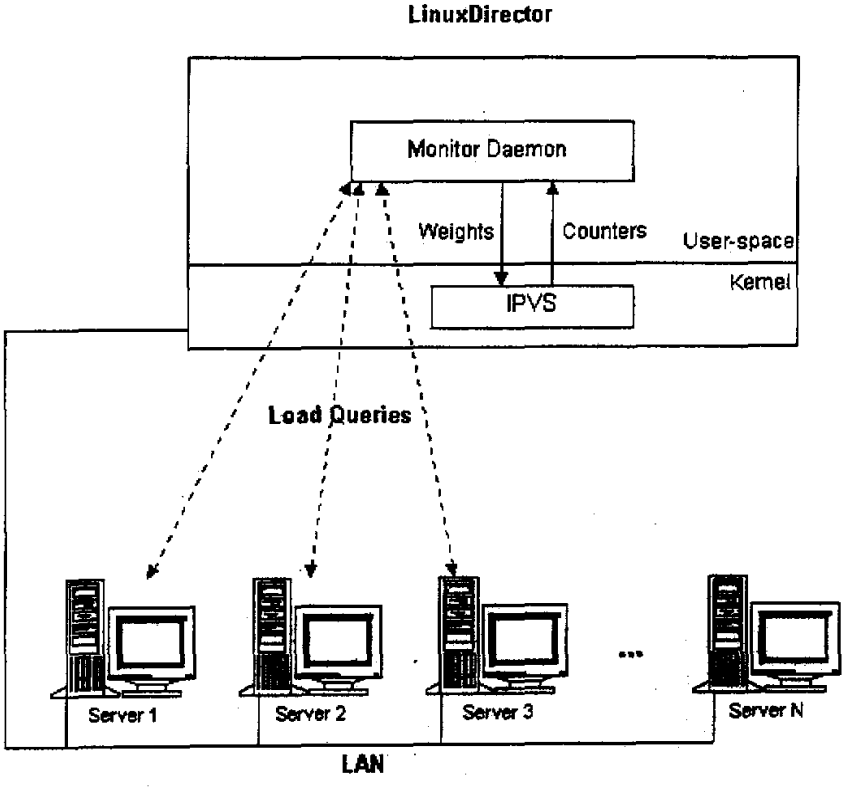

图 4.1：动态反馈负载均衡算法的工作环境

4.2.2 动态反馈负载均衡算法

TCP/IP 流量的特征通俗地说是有许多短事务和一些长事务组成，而长事务的工作量在整个工作量占有较高的比例。所以，我们要设计一种负载均衡算法，来避免长事务的请求总被分配到一些机器上，而是尽可能将带有毛刺（Burst）的分布分割成相对较均匀的分布。

我们提出基于动态反馈负载均衡机制，来控制新连接的分配，从而控制各个服务器的负载。在 LinuxDirector 调度器的内核中使用基于权值轮叫调度算法（Weighted Round-Robin Scheduling）来分配新的连接；在 LinuxDirector 的用户空间中运行

Monitor Daemon。Monitor Daemon 定时地监视和收集各个服务器的负载信息，根据多个负载信息算出一个综合负载值。Monitor Daemon 将各个服务器的综合负载值和当前权值算出一组新的权值。当综合负载值表示服务器比较忙时，新算出的权值会比其当前权值要小，这样新分配到该服务器的请求数就会少一些。当综合负载值表示服务器处于低利用率时，新算出的权值会比其当前权值要大，来增加新分配到该服务器的请求数。若新权值和当前权值的差值大于设定的阀值，Monitor Daemon 将该服务器的权值设置到内核中的 IPVS 调度中。过了一定的时间间隔（如 2 秒钟），Monitor Daemon 再查询各个服务器的情况，并相应调整服务器的权值；这样周期性地进行。可以说，这是一个负反馈机制，使得服务器保持较好的利用率。

在基于权值轮叫调度算法中，当服务器的权值为零，已建立的连接会继续得到该服务器的服务，而新的连接不会分配到该服务器。系统管理员可以将一台服务器的权值设置为零，使得该服务器安静下来，当已有的连接都结束后，他可以将该服务器切出，对其进行维护。维护工作对系统都是不可少的，比如硬件升级和软件更新等，零权值使得服务器安静的功能很主要。所以，在动态反馈负载均衡机制中我们要保证该功能，当服务器的权值为零时，我们不对服务器的权值进行调整。

4.2.3 综合负载

在计算综合负载时，我们主要使用两大类负载信息：输入指标和服务器指标。输入指标是在调度器上收集到的，而服务器指标是在服务器上的各种负载信息。我们用综合负载来反映服务器当前的比较确切负载情况，对于不同的应用，会有不同的负载情况，这里我们引入各个负载信息的系数，来表示各个负载信息在综合负载中轻重。系统管理员根据不同应用的需求，调整各个负载信息的系数。另外，系统管理员设置收集负载信息的时间间隔。

输入指标主要是在单位时间内服务器收到新连接数与平均连接数的比例，它是在调度器上收集到的，所以这个指标是对服务器负载情况的一个估计值。在调度器上有各个服务器收到连接数的计数器，对于服务器 S_i，可以得到分别在时间 T_1 和 T_2 时的计数器值 C_{i1} 和 C_{i2}，计算出在时间间隔 T_2-T_1 内服务器 S_i 收到新连接数 $Ni = C_{i2} - C_{i1}$。这样，得到一组服务器在时间间隔 T_2-T_1 内服务器 S_i 收到新连接数{N_i}，服务器 S_i 的输入指标 $INPUT_i$ 为

$$INPUT_i = \frac{N_i}{\sum_{i=1}^{n} N_i / n}$$

服务器指标主要记录服务器各种负载信息，如服务器当前 CPU 负载 $LOAD_i$、服务器当前磁盘使用情况 D_i、当前内存利用情况 M_i 和当前进程数目 P_i。在所有的服务器上运行着 SNMP（Simple Network Management Protocol）服务进程，而在调度器上的 Monitor Daemon 通过 SNMP 向各个服务器查询获得这些信息。若服务器在设定的时间间隔内没有响应，Monitor Daemon 认为服务器是不可达的，将服务器

在调度器中的权值设置为零，不会有新的连接再被分配到该服务器；若在下一次服务器有响应，再对服务器的权值进行调整。再对这些数据进行处理，使其落在[0, ∞)的区间内，1 表示负载正好，大于 1 表示服务器超载，小于 1 表示服务器处于低负载状态。获得调整后的数据有 $DISK_i$、$MEMORY_i$ 和 $PROCESS_i$。

另一个重要的服务器指标是服务器所提供服务的响应时间，它能比较好地反映服务器上请求等待队列的长度和请求的处理时间。调度器上的 Monitor Daemon 作为客户访问服务器所提供的服务，测得其响应时间。例如，测试从 WEB 服务器取一个 HTML 页面的响应延时，Monitor Daemon 只要发送一个“GET /”请求到每个服务器，然后记录响应时间。若服务器在设定的时间间隔内没有响应，Monitor Daemon 认为服务器是不可达的，将服务器在调度器中的权值设置为零。同样，我们对响应时间进行如上调整，得到 $RESPONSE_i$。

这里，我们引入一组可以动态调整的系数 R_i 来表示各个负载参数的重要程度，其中 $\Sigma R_i = 1$。综合负载可以通过以下公式计算出：

$$AGGREGATE\_LOAD_i = R_1 * INPUT_i + R_2 * LOAD_i + R_3 * DISK_i + R_4 * MEMORY_i + R_5 * PROCESS_i + R_6 * RESPONSES_i$$

例如，在 WEB 服务器集群中，我们采用以下系数{0.1, 0.3, 0.1, 0.1, 0.1, 0.3}，认为服务器的 CPU 负载和请求响应时间较其他参数重要一些。若当前的系数 Ri 不能很好地反映应用的负载，系统管理员可以对系数不断地修正，直到找到贴近当前应用的一组系数。

另外，关于查询时间间隔的设置，虽然很短的间隔可以更确切地反映各个服务器的负载，但是很频繁地查询（如 1 秒钟几次）会给调度器和服务器带来一定的负载，如频繁执行的 Monitor Daemon 在调度器会有一定的开销，同样频繁地查询服务器指标会服务器带来一定的开销。所以，这里要有个折衷（Tradeoff），我们一般建议将时间间隔设置在 2 到 10 秒之间。

4.2.4 权值计算

当服务器投入集群系统中使用时，系统管理员对服务器都设定一个初始权值 DEFAULT_WEIGHTi，在内核的 IPVS 调度中也先使用这个权值。然后，随着服务器负载的变化，对权值进行调整。为了避免权值变成一个很大的值，我们对权值的范围作一个限制[0, $SCALE*DEFAULT\_WEIGHT_i$]，SCALE 是可以调整的，它的缺省值为 10。

Monitor Daemon 周期性地运行，若 $DEFAULT\_WEIGHT_i$ 不为零，则查询该服务器的各负载参数，并计算出综合负载值 $AGGREGATE\_LOAD_i$。我们引入以下权值计算公式，根据服务器的综合负载值调整其权值。

$$w_i \leftarrow \begin{cases} w_i + A * \sqrt[3]{0.95 - AGGREGATE\_LOAD_i} & \text{当} AGGREGATE\_LOAD \neq 0.95 \\ w_i & \text{当} AGGREGATE\_LOAD = 0.95 \end{cases}$$

在公式中，0.95 是我们想要达到的系统利用率，A 是一个可调整的系数（缺省值为 5）。当综合负载值为 0.95 时，服务器权值不变；当综合负载值大于 0.95 时，权值变小；当综合负载值小于 0.95 时，权值变大。若新权值大于 SCALE*DEFAULT_WEIGHT，我们将新权值设为 SCALE*DEFAULT_WEIGHT。若新权值与当前权值的差异超过设定的阀值，则将新权值设置到内核中的 IPVS 调度参数中，否则避免打断 IPVS 调度的开销。

在实现使用中，若发现所有服务器的权值都小于他们的 DEFAULT_WEIGHT，则说明整个服务器集群处于超载状态，这时需要加入新的处理结点到集群中；若所有服务器的权值都接近于 SCALE*DEFAULT_WEIGHT，则说明当前系统的负载比较轻。

§4.3 性能模拟

为了研究调度算法对整个系统性能的影响，我们主要对加权轮叫调度、加权最小连接调度和动态反馈负载均衡算法进行性能模拟。以上算法都假设各个后端服务器能处理任一请求。

我们实验用的 Web 服务器是 Pentium II 350MHz 和 128M 内存的机器，运行 Linux 内核 2.2.17。一个 TCP 连接的建立和挂断分别需要 140us 的 CPU 时间，传输 512K 字节需要 40us。一个磁盘的寻道延时为 28ms，磁盘每 4K 字节的传输时间为 410us。Web 服务器采用 LRU 的淘汰策略。

我们将实验室多台 Web 服务器的两个月日志记录，去掉 CGI 等动态的访问请求，归并成一个大的日志文件。日志中含有 37804 个请求目标，覆盖的数据为 1629M 字节。根据文件的长度可以算出文件命中和不命中时的处理时间。在动态反馈负载均衡算法模拟中，各个服务器的负载是在过去一分钟内平均的正在运行进程数，我们每隔 5 秒钟对服务器负载按以下公式计算：

$$load = load * \frac{11}{12} + Active\_processes * \frac{1}{12}$$

根据服务器的负载值，再对服务器的权值进行相应的调整。

最后，根据上述模型对算法进行性能模拟，模拟结果如图 4.2 所示。我们可以看出 WRR 和 WLC 的性能很相近，动态反馈负载均衡算法 DFLB 性能要它们高一些，随着服务器结点数目增加，这种优势越明显。当结点数为 7 时，DFLB 的性能比 WRR 和 WLC 高出 20%。这是因为在 WRR 和 WLC 算法中会出现服务器间负载不平衡，有些服务器超载后还收到请求，而有些服务器闲置着，使得系统的吞吐率受影响；DFLB 通过动态反馈机制及时地调整服务器的权值，来避免服务器间负载不平衡，从而提高整个系统的吞吐率。

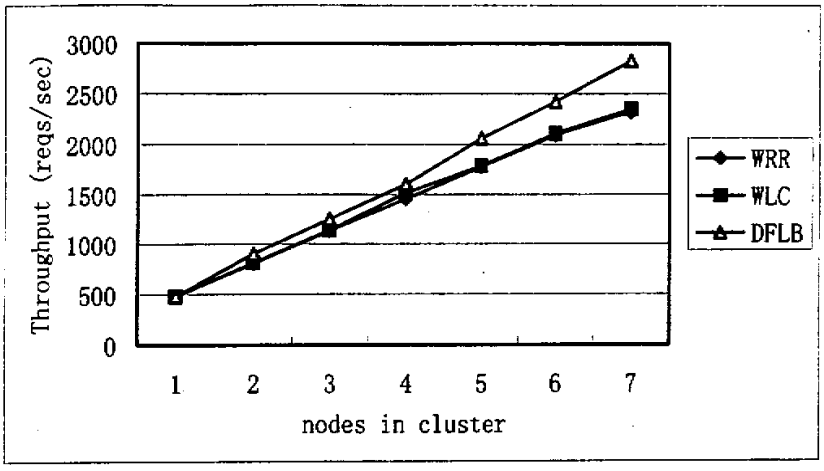

图 4.2：调度算法的性能模拟

§4.4 本章小结

IP 虚拟服务器在内核中的以下连接调度算法：

- 轮叫调度(Round-Robin Scheduling)
- 加权轮叫调度(Weighted Round-Robin Scheduling)
- 最小连接调度（Least-Connection Scheduling）
- 加权最小连接调度（Weighted Least-Connection Scheduling）
- 全球服务器调度（Global Server Scheduling）

因为请求的服务时间差异较大，内核中的连接调度算法容易使得服务器运行出现倾斜。为此，我们提出了一个动态反馈负载均衡算法，结合内核中的加权连接调度算法，根据动态反馈回来的负载信息来调整服务器的权值，来调整服务器间处理请求数的比例，从而避免服务器间的负载不平衡。性能模拟结果也说明动态反馈负载算法可以较好地避免服务器的倾斜，提高系统的资源使用效率，从而提高系统的吞吐率。

第五章 IP 虚拟服务器的实现和性能测试

本章主要讲述 IP 负载均衡技术和连接调度算法在 Linux 内核[78, 79, 80]中的实现，称之为 IP 虚拟服务器（IP Virtual Server，简写为 IPVS），再叙述了实现中所遇到关键问题的解决方法和优化。最后，对 IPVS 软件进行性能测试，并列举该软件的应用情况。

§5.1 系统实现的基本框架

我们分别在 Linux 内核 2.0 和内核 2.2 中修改了 TCP/IP 协议栈，在 IP 层截取和改写/转发 IP 报文，实现了三种 IP 负载均衡技术，并提供了一个 ipvsadm 程序进行虚拟服务器的配置和管理。在 Linux 内核 2.4 中，我们把它实现为 NetFilter 的一个模块，很多代码作了改写和进一步优化，目前版本已在网上发布，根据反馈信息该版本已经较稳定。

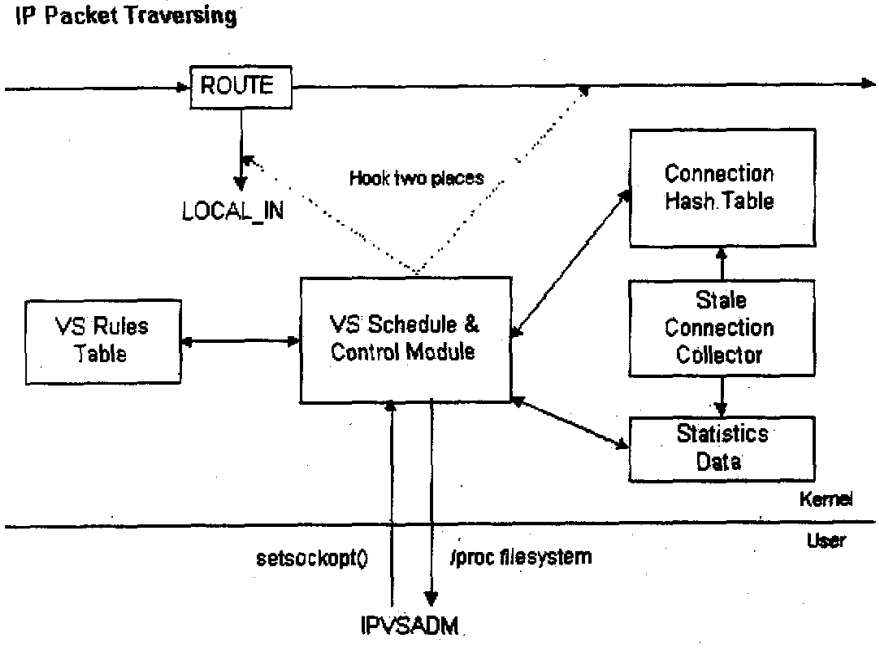

图 5.1：系统的主要功能模块

系统的主要功能模块如图 5.1 所示，“VS Schedule & Control Module”是虚拟服务器的主控模块，它挂接在 IP 报文遍历的 LOCAL_IN 链和 IP_FORWARD 链两处，用于截取/改写 IP 报文；“VS Rules Table”用于存放虚拟服务器的规则，“Connections

Hash Table”表是用于记录当前连接的 Hash 表；“Stale Connection Collector”模块用于回收已经过时的连接；“Statistics Data”表记录 IPVS 的统计信息。用户空间的 ipvsadm 管理程序通过 setsockopt()函数将虚拟服务器的规则写入“VS Rules Table”表中，通过/proc 文件系统把“VS Rules Table”表中的规则读出。

当一个 IP 报文到达时，若报文的目标地址是本地的 IP 地址，IP 报文会转到 LOCAL_IN 链上，否则转到 IP_FORWARD 链上。IPVS 模块主要挂接在 LOCAL_IN 链和 IP_FORWARD 链两处。当一个目标地址为 Virtual IP Address 的报文到达时，该报文会被挂接在 LOCAL_IN 链上的 IPVS 程序捕获，若该报文属于在连接 Hash 表中一个已建立的连接，则根据连接的信息将该报文发送到目标服务器，否则该报文为 SYN 时，根据连接调度算法从一组真实服务器中选出一台服务器，根据 IP 负载调度设置的规则将报文发送给选出的服务器，并在连接 Hash 表中记录这个连接。挂接在 IP_FORWARD 链上的 IPVS 程序是改写 VS/NAT 中服务器响应报文的地址。

连接的 Hash 表可以容纳几百万个并发连接，在 Linux 内核 2.2 和内核 2.4 的 IP 虚拟服务器版本中每个连接只占用 128Bytes 有效内存，例如一个有 256M 可用内存的调度器就可调度两百万个并发连接。连接 Hash 表的桶个数可以由用户根据实际应用来设定，来降低 Hash 的冲突率。

在每个连接的结构中有连接的报文发送方式、状态和超时等。报文发送方式有 VS/NAT、VS/TUN、VS/DR 和本地结点，报文会被以连接中设定的方式发送到目标服务器。这意味着在一个服务器集群中，我们可以用不同的方式（VS/NAT、VS/TUN 或 VS/DR）来调度不同的服务器。连接的状态和超时用于记录连接当前所在的状态，如 SYN_REC、ESTABLISHED 和 FIN_WAIT 等，不同的状态有不同的超时值。

§5.2 系统实现的若干问题

本节讲述实现时所遇到的若干主要问题和它们的解决方法或者优化处理。

5.2.1 Hash 表

在系统实现中，我们多处用到 Hash 表，如连接的查找和虚拟服务的查找。选择 Hash 表优先 Tree 等复杂数据结构的原因是 Hash 表的插入和删除的复杂度为 O(1)，而 Tree 的复杂度为 O(log(n))。Hash 表的查找复杂度为 O(n/m)，其中 n 为 Hash 表中对象的个数，m 为 Hash 表的桶个数。当对象在 Hash 表中均匀分布和 Hash 表的桶个数与对象个数一样多时，Hash 表的查找复杂度可以接近 O(1)。

因为连接的 Hash 表要容纳几百万个并发连接，并且连接的 Hash 表是系统使用最频繁的部分，任何一个报文到达都需要查找连接 Hash 表，所以如何选择一个高效的连接 Hash 函数直接影响到系统的性能。连接 Hash 函数的选择要考虑到两个因素，一个是尽可能地降低 Hash 表的冲突率，另一个是 Hash 函数的计算不是很复杂。

一个连接有客户的<protocol,address,port>、虚拟服务的<address,port>和目标服务器的<address,port>等元素，其中客户的<protocol,address,port>是每个连接都不相同的，后两者在不同的连接经常重叠。所以，我们选择客户的<protocol,address,port>来计算 Hash Key。在 IPVS 版本中，我们用以下快速的移位异或 Hash 函数来计算。

```
#define IP_VS_TAB_BITS    CONFIG_IP _VS_TAB_BITS
#define IP_VS_TAB_SIZE    (1 << IP_VS_TAB_BITS)
#define IP_VS_TAB_MASK    (IP_VS_TAB_SIZE - 1)
inline unsigned ip_vs_hash_key(unsigned proto, unsigned addr, unsigned port)
{
    return (proto ^ addr ^ (addr>>IP_VS_TAB_BITS) ^ port)
        & IP_VS_TAB_MASK;
}
```

为了评价 Hash 函数的效率，我们从一个运行 IPVS 的真实站点上取当前连接的样本，它一共含有 35652 个并发连接。在有 64K 桶的 Hash 表中，连接分布如下：

桶的长度（L_j）	该长度桶的个数（N_j）
5	16
4	126
3	980
2	5614
1	20900

通过以下公式算出所有连接查找一次的代价：

$$\sum_j \left(N_j * \sum_{i=1}^{L_j} i \right) = \frac{1}{2} \sum_j N_j * L_j * (L_j + 1)$$

所有连接查找一次的代价为 45122，每个连接查找的平均代价为 1.266（即 45122/35652）。我们对素数乘法 Hash 函数[81]进行分析，素数乘法 Hash 函数是通过乘以素数使得 Hash 键值达到较均匀的分布。

```
inline unsigned ip_vs_hash_key(unsigned proto, unsigned addr, unsigned port)
{
    return ((proto+addr+port)* 2654435761UL) & IP_VS_TAB_MASK;
}
```

其中，2654435761UL 是 2 到 2^32 间黄金分割的素数，

$$\frac{\sqrt{5}-1}{2} = 0.618033989$$

$$2654435761 / 4294967296 = 0.618033987$$

在有 64K 桶的 Hash 表中，素数乘法 Hash 函数的总查找代价为 45287。可见，现在 IPVS 中使用的移位异或 Hash 函数还比较高效。

5.2.2 垃圾回收

为了将不再被使用的连接单元回收，我们在连接上设置一个定时器，当连接超时，将该连接回收。因为系统中有可能存在几百万个并发连接，若使用内核中的定时器，几百万个连接单元系统的定时器的列表中，系统每隔 1/100 秒进行单元的迁移和回收已超时的连接单元，这会占用很多系统的开销。

因为连接的回收并不需要很精确，我们可以让系统的定时器每隔 1 秒启动连接回收程序来回收那些超时的连接。为此，我们设计了一个慢定时器，连接的定时都是以 1 秒钟为单位。用三个时间大转盘，第一个转盘有 1024 个刻度，定时在 1024 秒钟之内的连接都挂接在第一个转盘的各个刻度上；第二个转盘有 256 个刻度，定时在$[2^{10}, 2^{18})$区间的落在第二个转盘上；第三个转盘有 256 个刻度，定时在$[2^{18}, 2^{26})$区间的落在第三个转盘上。

慢定时器处理程序每隔 1 秒由系统定时器启动运行一次，将第一个转盘当前指针上的连接进行回收，再将指针顺时针转一格。若指针正好转了一圈，则对第二个转盘当前指针上的连接进行操作，根据他们的定时迁移到第一个转盘上，再将指针顺时针转一格。若第二个转盘的指针正好转了一圈，则对第三个转盘当前指针上的连接进行操作，根据他们的定时迁移到第二个转盘上。

使用这种慢定时器极大地提高了过期连接的回收问题。在最初的版本中，我们是直接使用系统的定时器进行超时连接的回收，但是当并发连接数增加到 500,000 时，系统的 CPU 使用率已接近饱和，所以我们重新设计了这种高效的垃圾回收机制。

5.2.3 ICMP 处理

调度器需要实现虚拟服务的 ICMP 处理，这样进来的虚拟服务 ICMP 报文会被改写或者转发给正确的后端服务器，出去的 ICMP 报文也会被正确地改写和发送给客户。ICMP 处理对于客户和服务器间的错误和控制通知是非常重要的。

ICMP 消息可以发现在客户和服务器间的 MTU（Maximum Transfer Unit）值。在客户的请求被 VS/DR 调度到一台服务器执行，服务器将执行结果直接返回给客户。例如响应报文的 MTU 为 1500 个字节，在服务器到客户的路径中有一段线路的 MTU 值为 512 个字节，这时路由器会向报文的源地址（即虚拟服务的地址）发送一个需要分段为 512 个字节的 ICMP 消息。该 ICMP 消息会到达调度器，调度器需要将 ICMP 消息中原报文的头取出，再在 Hash 表中找到相应的连接，然后将该 ICMP 消息转发给对应的服务器。这样，服务器就会将原有的报文分段成 512 个字节进行发送，客户得到服务的响应。

5.2.4 可装卸的调度模块

为了提高系统的灵活性，我们将连接调度做成可装卸的模块（Loadable Modules），如 ip_vs_rr.o、ip_vs_wrr.o、ip_vs_lc.o、ip_vs_wlc.o 和 ip_vs_gs.o。当虚拟服务设置时，会将相应的模块调到内核中。这样，有助于提高系统的使用效率，不装载不被使用的资源。

5.2.5 锁的处理和优化

在系统中虚拟服务规则的读和写需要锁来保证处理的一致性。在连接的 Hash 表中，同样需要锁来保证连接加入和删除的一致性。连接的 Hash 表是系统使用最频繁的资源，任何一个报文到达都需要查找连接 Hash 表。如果只有一个锁来管理连接 Hash 表的操作，锁的冲突率会很高[82]。为此，我们引入有 n 个元素的锁数组，每个锁分别控制 1/n 的连接 Hash 表，增加锁的粒度，降低锁的冲突率。在两个 CPU 的 SMP 机器上，假设 CPU 操作 Hash 表的部位是随机分布的，则两个 CPU 同时操作同一区域的概率为 1/n。在系统中 n 的缺省值为 16。

5.2.6 连接的相关性

到现在为止，我们假设每个连接都相互独立的，所以每个连接被分配到一个服务器，跟过去和现在的分配没有任何关系。但是，有时由于功能或者性能方面的原因，一些来自同一用户的不同连接必须被分配到同一台服务器上。

FTP[83]是一个因为功能设计导致连接相关性的例子。在 FTP 使用中，客户需要建立一个控制连接与服务器交互命令，建立其他数据连接来传输大量的数据。在主动的 FTP 模式下，客户通知 FTP 服务器它所监听的端口，服务器主动地建立到客户的数据连接，服务器的端口一般为 20。IPVS 调度器可以检查报文的内容，可以获得客户通知 FTP 服务器它所监听的端口，然后在调度器的连接 Hash 表中建立一个相应的连接，这样服务器主动建立的连接可以经过调度器。但是，在被动的 FTP 模式下，服务器告诉客户它所监听的数据端口，服务器被动地等待客户的连接。在 VS/TUN 或 VS/DR 下，IPVS 调度器是在从客户到服务器的半连接上，服务器将响应报文直接发给客户，IPVS 调度器不可能获得服务器告诉客户它所监听的数据端口。

SSL（Secure Socket Layer）[84]是一个因为性能方面原因导致连接相关性的例子。当一个 SSL 连接请求建立时，一个 SSL 的键值（SSL Key）必须要在服务器和客户进行选择和交换，然后数据的传送都要经过这个键值进行加密，来保证数据的安全性。因为客户和服务器协商和生成 SSL Key 是非常耗时的，所以 SSL 协议在 SSL Key 的生命周期内，以后的连接可以用这个 SSL Key 和服务器交换数据。如果 IPVS 调

度器将以后的连接调度到其他服务器，这会导致连接的失败。

我们现在解决连接相关性的方法是持久服务（Persistent Service）的处理。使用两个模板来表示客户和服务器之间的持久服务，模板（protocol, client_ip, 0, virtual_ip, virtual_port, dest_ip, dest_port）表示来自同一客户 client_ip 到虚拟服务（virtual_ip, virtual_port）的任何连接都会被转发到目标服务器（dest_ip, dest_port），模板（protocol, client_ip, 0, virtual_ip, 0 dest_ip, 0）表示来自同一客户 client_ip 到虚拟服务器 virtual_ip 的任何连接都会被转发到目标服务器 dest_ip，前者用于单一的持久服务，后者用于所有端口的持久服务。当一个客户访问一个持久服务时，IPVS 调度器会在连接 Hash 表中建立一个模板，这个模板会在一个可设置的时间内过期，如果模板有所控制的连接没有过期，则这个模板不会过期。在这个模板没有过期前，所有来自这个客户到相应服务的任何连接会被发送到同一台服务器。

持久服务还可设置持久的粒度，即可设置将来自一个 C 类地址范围的所有客户请求发送到同一台服务器。这个特征可以保证当使用多个代理服务器的客户访问集群时，所有的连接会被发送到同一服务器。

虽然持久服务可能会导致服务器间轻微的负载不平衡，因为持久服务的一般调度粒度是基于每个客户机的，但是这有效地解决连接相关性问题，如 FTP、SSL 和 HTTP Cookie[85]等。

5.2.7 本地结点

本地结点（Local Node）功能是让调度器本身也能处理请求，在调度时就相当一个本地结点一样，在实现时就是根据配置将部分连接转交给在用户空间的服务进程，由服务进程处理完请求将结果返回给客户。该功能的用处如下：

当集群中服务器结点较少时，如只有三、四个结点，调度器在调度它们时，大部分的 CPU 资源是闲置着，可以利用本地结点功能让调度器也能处理一部分请求，来提高系统资源的利用率。

在分布式服务器中，我们可以利用 IPVS 调度的本地结点功能，在每台服务器上加载 IPVS 调度模块，在一般情况下，利用本地结点功能服务器处理到达的请求，当管理程序发现服务器超载时，管理程序将其他服务器加入调度序列中，将部分请求调度到其他负载较轻的服务器上执行。

在地理上分布的服务器镜像上，镜像服务器利用本地结点功能功能处理请求，当服务器超载时，服务器通过 VS/TUN 将请求调度到邻近且负载较轻的服务器上。

5.2.8 数据统计

在虚拟服务使用情况的统计上，我们实现了如下计数器：

- 调度器所处理报文的总数
- 调度器所处理连接的总数

- 调度器中所有并发连接的数目
- 每个虚拟服务处理连接的总数
- 每个服务器所有并发连接的数目

在单位时间内，我们可以根据调度器所处理报文总数之差得出调度器的报文处理速率，根据调度器所处理连接总数之差得出调度器的连接处理速率。同样，我们可以算出每个虚拟服务的连接处理速率。

5.2.9 防卫策略

调度器本身可以利用 Linux 内核报文过滤功能设置成一个防火墙，只许可虚拟服务的报文进入，丢掉其他报文。调度器的脆弱之处在于它需要记录每个连接的状态，每个连接需要占用 128 个字节，一些恶意攻击可能使得调度器生成越来越多的并发连接，直到所有的内存耗尽，系统出现拒绝服务（Denial of Service）[86, 87, 88]。但是，一般 SYN-Flooding 攻击调度器是非常困难的，假设系统有 128Mbytes 可用内存，则系统可以容纳一百万个并发连接，每个处理接受 SYN 连接的超时（Timeout）为 60 秒，SYN-Flooding 主机需要生成 16,666 Packets/Second 的流，这往往需要分布式 SYN-Flooding 工具，由许多个 SYN-Flooding 主机同时进行攻击调度器。

为了避免此类大规模的恶意攻击，我们在调度器中实现三种针对 DoS 攻击的防卫策略。它们是随机丢掉连接、在调度报文前丢掉 1/rate 的报文、使用更安全的 TCP 状态转换和更短的超时。在系统中有三个开关分别控制它们，开关处于 0 表示该功能完全关掉。1 和 2 表示自动状态，当系统的有效内存低于设置的阀值时，该防卫策略被激活，开关从 1 状态迁移到 2 状态；当系统的有效内存高于设置的阀值时，该防卫策略被关掉，开关从 2 状态迁移到 1 状态。3 表示该策略永远被激活。

5.2.10 调度器间的状态复制

尽管 IP 虚拟服务器软件已被证明相当鲁棒，但调度器有可能因为其他原因而失效，如机器的硬件故障和网络线路故障等。所以，我们引入一个从调度器作为主调度器的备份，当主调度器失效时，从调度器将接管 VIP 等地址进行负载均衡调度。在现在的解决方案中，当主调度器失效时，调度器上所有已建立连接的状态信息将丢失，已有的连接会中断，客户需要向重新连接，从调度器才会将新连接调度到各个服务器上。这对客户会造成一定的不便。为此，我们考虑一种高效机制将主调度器的状态信息及时地复制到从调度器，当从调度器接管时，绝大部分已建立的连接会持续下去。

因为调度器的连接吞吐率是非常高的，如每秒处理一万多个连接，如何将这些变化非常快的状态信息高效地复制到另一台服务器？我们设计利用内核线程实现主从同步进程，在操作系统的内核中，直接将状态信息发送到从调度器上，可以避免用户空间和核心的切换开销。其结构如图 5.2 所示：

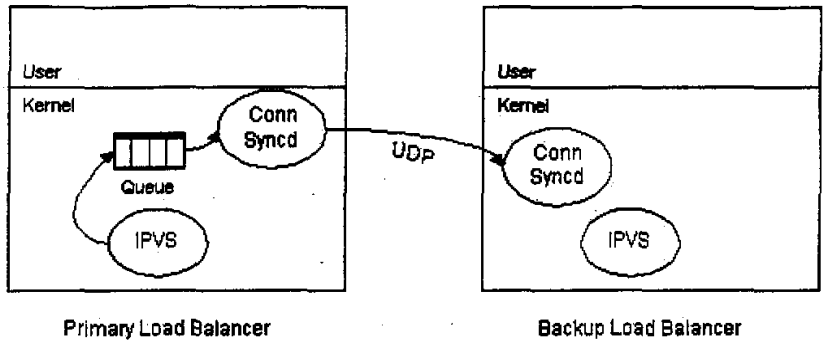

图 5.2：主从调度器间的状态复制

在主从调度器的操作系统内核中分别有两个内核线程 ConnSyncd，主调度器上的 ConnSyncd 每隔 1/10 秒钟唤醒一次从更新队列中将更新信息读出，将更新信息发给从调度器上的 ConnSyncd，然后在从调度器内核中生成相应的状态信息。为了减少主从调度器间的通讯开销，在主调度器的更新队列中只放新连接生成的信息，在从调度器中生成连接信息，设置定时器，当连接超时，该连接会自动被删除。

主从调度器间状态复制的代码正在编写中。因为主从调度器间的状态复制会降低调度器的吞吐率，所以主从调度器间状态复制会以模块的形式出现，当用户特别需要时，可以将该模块加入内核中。

§5.3 性能测试

在美国 VA Linux 公司[2]的高级工程师告诉我，他们在实验室中用一个 IPVS 调度器（VS/DR 方式）和 58 台 WEB 服务器组成一个 WEB 集群，想测试在真实网络服务负载下 IPVS 调度器的性能，但是他们没有测试 IPVS 调度器的性能，当 58 台 WEB 服务器都已经满负荷运行时，IPVS 调度器还处于很低的利用率（小于 0.2）。他们认为系统的瓶颈可能在网卡的速度和报文的转发速度，估计要到几百台服务器时，IPVS 调度器会成为整个系统的瓶颈。

我们没有足够的物理设备来测试在真实网络服务负载下 IPVS 调度器的性能，况且在更高的硬件配置下（如两块 1Gbps 网卡和 SMP 机器），调度器肯定会有更高的性能。为了更好地估计在 VS/DR 和 VS/TUN 方式下 IPVS 调度器的性能，我们专门写一个测试程序 testlvs，程序不断地生成 SYN 的报文发送给调度器上的虚拟服务，调度器会生成一个连接并将 SYN 报文转发给后端服务器，我们设置后端服务器在路由时将这些 SYN 报文丢掉，在后端服务器的网卡上我们可以获得进来的报文速率，从而估计出调度器的报文处理速率。

我们的测试环境如图 5.3 所示：有四台客户机、三台服务器和一个 Pentium III

[2] VA Linux 公司是目前最大的 Linux 系统提供商之一，他们为客户提供基于 IPVS 的服务器集群系统和相关的技术服务。

500MHz、128M 内存和 2 块 100M 网卡的调度器，四台客户机和调度器通过一个 100M 交换机相连，三台服务器和调度器也通过一个 100M 交换机相连。它们都运行 Linux 操作系统。

在 VS/NAT 的性能测试中，我们分别在三台服务器启动三个 Netpipe 服务进程，在调度器上开启三个虚拟服务通过网络地址转换到三台服务器，用三台客户机运行 Netpipe 分别向三个虚拟服务进行测试，调度器已经满负荷运行，获得三个 Netpipe 的累计吞吐率为 89Mbps。在正常网络服务下，我们假设每个连接的平均数据量为 10Kbytes，VS/NAT 每秒处理的连接数为 1112.5 Connections/Second。

在 VS/DR 和 VS/TUN 的性能测试中，我们设置后端服务器在路由时将这些 SYN 报文丢掉，后端服务器就像一个黑洞将报文吸掉，它的处理开销很小，所以我们在后端服务器只用两台。我们在后端服务器上运行程序来测试进来报文的速率，在调度器上将一虚拟服务负载均衡到两台后端服务器，然后在四台客户机上运行 testlvs 不断地向虚拟服务发 SYN 报文，报文的源地址是随机生成的，每个报文的大小为 40 个字节。测试得 VS/DR 的处理速率为 150,100 packets/second，VS/TUN 的处理速率为 141,000packets/second，可见将 IP 隧道的开销要比修改 MAC 地址要大一些。在实际实验中，我们测得平均文件长度为 10K 的 HTTP 连接，从客户到服务器方向的报文为 6 个。这样，我们可以推出 VS/DR 或 VS/TUN 调度器的最大吞吐率为 25,000 Connections/Second。

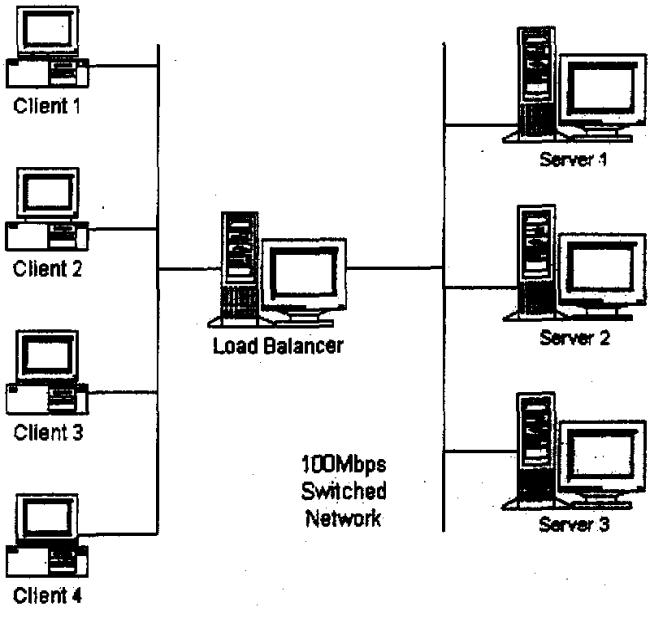

图 5.3：IPVS 调度器的性能测试环境

§5.4 LVS 集群的应用

Linux 虚拟服务器项目（Linux Virtual Server Project）的网址是 http://www.LinuxVirtualServer.org/，LVS 中的 IPVS 第一个版本源程序于 1998 年 5 月在网上发布。至今，本项目受到不少关注，LVS 系统已被用于很多重负载的站点，就我们所知该系统已在美、英、德、澳等国的十几个站点上正式使用。

我们没有上百台机器和高速的网络来测试 LVS 的终极性能，所以举 LVS 的应用实例来说明 LVS 的高性能和稳定性。我们所知的一些大型 LVS 应用实例如下：

- 英国国家 JANET Cache Service（wwwcache.ja.net）是为英国 150 所以上的大学提供 Web Cache 服务。他们用 28 个结点的 LVS 集群代替了原有现 50 多台相互独立的 Cache 服务器，用他们的话说现在速度就跟夏天一样，因为夏天是放假期间没有很多人使用网络。
- Linux 的门户站点（www.linux.com）用 LVS 将很多台 VA Linux SMP 服务器组成高性能的 WEB 服务，已使用将近一年。
- SourceForge（sourceforge.net）是在全球范围内为开发源码项目提供 WEB、FTP、Mailing List 和 CVS 等服务，他们也使用 LVS 将负载调度到十几台机器上。
- 世界上最大的 PC 制造商之一采用了两个 LVS 集群系统，一个在美洲，一个在欧洲，用于网上直销系统。
- 以 RealPlayer 提供音频视频服务而闻名的 Real 公司（www.real.com）使用由 20 台服务器组成的 LVS 集群，为其全球用户提供音频视频服务。在 2000 年 3 月时，整个集群系统已收到平均每秒 20,000 个连接的请求流。
- NetWalk（www.netwalk.com）用多台服务器构造 LVS 系统，提供 1024 个虚拟服务，其中本项目的一个美国镜像站点（www.us.linuxvirtualserver.org）。
- RedHat（www.redhat.com）从其 6.1 发行版起已包含 LVS 代码，他们开发了一个 LVS 集群管理工具叫 Piranha，用于控制 LVS 集群，并提供了一个图形化的配置界面。
- VA Linux（www.valinux.com）向客户提供基于 LVS 的服务器集群系统，并且提供相关的服务和支持。
- TurboLinux 的“世界一流 Linux 集群产品”TurboCluster 实际上是基于 LVS 的想法和代码的，只是他们在新闻发布和产品演示时忘了致谢[3]。
- 红旗 Linux 和中软都提供基于 LVS 的集群解决方案，并在 2000 年 9 月召开的 Linux World China 2000 上展示。

[3] TurboLinux 总裁 Cliff Miller 先生于 1999 年 3 月在北京当面道歉，但后来他们还是忘了致谢。:)

§5.5 本章小结

本章主要讲述了 IP 虚拟服务器在 Linux 内核中的实现、关键问题的解决方法和优化处理、以及 IP 虚拟服务器的性能测试和应用情况。IP 虚拟服务器具有以下特点：

- 三种 IP 负载均衡技术，在一个服务器集群中，不同的服务器可以使用不同的 IP 负载均衡技术。
- 可装卸连接调度模块，共有五种连接调度算法。
- 高效的 Hash 函数
- 高效的垃圾回收机制
- 虚拟服务的数目没有限制，每个虚拟服务有自己的服务器集。
- 支持持久的虚拟服务
- 正确的 ICMP 处理
- 拥有本地结点功能
- 提供系统使用的统计数据
- 针对大规模 DoS 攻击的三种防卫策略

通过 IP 虚拟服务器软件和集群管理工具可以将一组服务器组成一个高性能、高可用的网络服务。该系统具有良好的伸缩性，支持几百万个并发连接。无需对客户机和服务器作任何修改，可适用任何 Internet 站点。该系统已经在很多大型的站点得到很好的应用。

第六章 内核中的基于内容请求分发

前面几章讲述了在 Linux 虚拟服务器(Linux Virtual Server)的框架下，先在 Linux 内核中实现了含有三种 IP 负载均衡技术的 IP 虚拟服务器，可将一组服务器构成一个实现高可伸缩、高可用的网络服务的服务器集群。在 IPVS 中，使得服务器集群的结构对客户是透明的，客户访问集群提供的网络服务就像访问一台高性能、高可用的服务器一样。客户程序不受服务器集群的影响不需作任何修改。系统的伸缩性通过在服务机群中透明地加入和删除一个节点来达到，通过检测节点或服务进程故障和正确地重置系统达到高可用性。

IPVS 基本上是一种高效的 Layer-4 交换机，它提供负载平衡的功能。当一个 TCP 连接的初始 SYN 报文到达时，IPVS 就选择一台服务器，将报文转发给它。此后通过查发报文的 IP 和 TCP 报文头地址，保证此连接的后继报文被转发到相同的服务器。这样，IPVS 无法检查到请求的内容再选择服务器，这就要求后端的服务器组是提供相同的服务，不管请求被送到哪一台服务器，返回结果都应该是一样的。但是在有一些应用中后端的服务器可能功能不一，有的是提供 HTML 文档的 Web 服务器，有的是提供图片的 Web 服务器，有的是提供 CGI 的 Web 服务器。这时，就需要基于内容请求分发 (Content-Based Request Distribution)，同时基于内容请求分发可以提高后端服务器上访问的局部性。

§6.1 基于内容的请求分发

根据应用层（Layer-7）的信息调度 TCP 负载不是很容易实现，对于所有的 TCP 服务，应用层信息必须等到 TCP 连接建立（三次握手协议）以后才能获得。这就说明连接不能用 Layer-4 交换机收到一个 SYN 报文时进行调度。来自客户的 TCP 连接必须在交换机上被接受，建立在客户与交换机之间 TCP 连接，来获得应用的请求信息。一旦获得请求信息，分析其内容来决定哪一个后端服务器来处理，再将请求调度到该服务器。

现有两种方法实现基于内容的调度。一种是 TCP 网关（TCP Gateway），交换机建立一个到后端服务器的 TCP 连接，将客户请求通过这个连接发到服务器，服务器将响应结果通过该连接返回到交换机，交换机再将结果通过客户到交换机的连接返回给客户。另一种是 TCP 迁移（TCP Migration），将客户到交换机 TCP 连接的交换机端迁移到服务器，这样客户与服务器就可以建立直接的 TCP 连接，但请求报文还需要经过交换机调度到服务器，响应报文直接返回给客户。

就我们所知，Resonate[89]实现了 TCP 迁移方法，他们称之为 TCP 连接跳动（TCP Connection Hop），他们的软件是专有的。TCP 迁移需要修改交换机的 TCP/IP 协议

栈，同时需要修改所有后端服务器的 TCP/IP 协议栈，才能实现将 TCP 连接的一端从一台机器及其迁移到另一台上。在文献[90]中，Rice 大学和 IBM 的研究人员简要地描述了它们的 TCP Handoff Protocol，在交换机和后端服务器上安装 TCP Handoff 协议，交换机获得客户请求后，通过 TCP Handoff 协议将 TCP 连接的交换机端转给后端服务器。但是，他们没有提供任何其他文档或研究报告描述他们的 TCP Handoff 协议和实现。他们提出的提高局部性调度算法 LARD 只考虑静态文档。虽然在交换机获得客户请求后，TCP 迁移方法比较高效，但是该方法实现工作量大，要修改交换机和后端服务器的操作系统，不具有一般性。

因此，TCP 网关方法被绝大部分 Layer-7 交换的商业产品和自由软件所使用，如 ArrowPoint 的 CS-100 和 CS-800 Web 交换机（Web Switch）[91]、Zeus 负载调度器[14]、爱立信实验室的 EDDIE[30]、自由软件 Apache[92]和 Squid[93]等。TCP 网关方法不需要更改服务器的操作系统，服务器只要支持 TCP/IP 即可，它具有很好的通用性。但是，TCP 网关一般都是在用户空间实现的，其处理开销比较大，如图 6.1 所示。

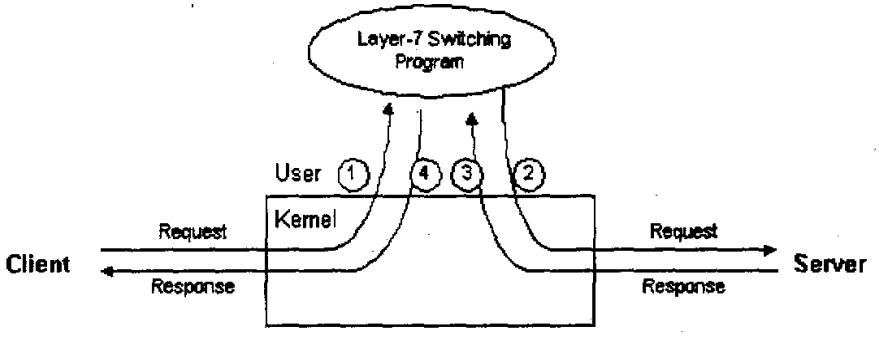

图 6.1：用户空间的 TCP Gateway

从图中我们可以看出，对于任一 TCP 请求，客户将请求发到交换机，计算机将请求报文从内核传给用户空间的 Layer-7 交换程序，Layer-7 交换程序根据请求选出服务器，再建一个 TCP 连接将请求发给服务器，返回的响应报文必须先由内核传给用户空间的 Layer-7 交换程序，再由 Layer-7 交换程序通过内核将结果返回给客户。对于一个请求报文和对应的响应报文，都需要四次内核与用户空间的切换，切换和内存复制的开销是非常高的，所以用户空间 TCP Gateway 的伸缩能力很有限，一般来说一个用户空间 TCP Gateway 只能调度三、四台服务器，当连接的速率到达每秒 1000 个连接时，TCP Gateway 本身会成为瓶颈。所以，在 ArrowPoint 的 CS-800 Web 交换机中集成了多个 TCP Gateway。

虽然 Layer-7 交换比 Layer-4 交换处理复杂，但 Layer-7 交换带来以下好处：

- 相同页面的请求被发送到同一台的服务器，所请求的页面很有可能会被服务器缓存，可以提高单台服务器的主存 Cache 使用效率。
- 一些研究[94]表明 WEB 访问流中存在空间的局部性。Layer-7 交换可以充分

利用访问的局部性，将相同类型的请求发送到同一台服务器，使得每个后端服务器收到的请求相似性好，有利于进一步提高单台服务器的主存 Cache 使用效率，从而在有限的硬件配置下提高系统的整体性能。

- 后端的服务器可运行不同类型的服务，如文档服务，图片服务，CGI 服务和数据库服务等。

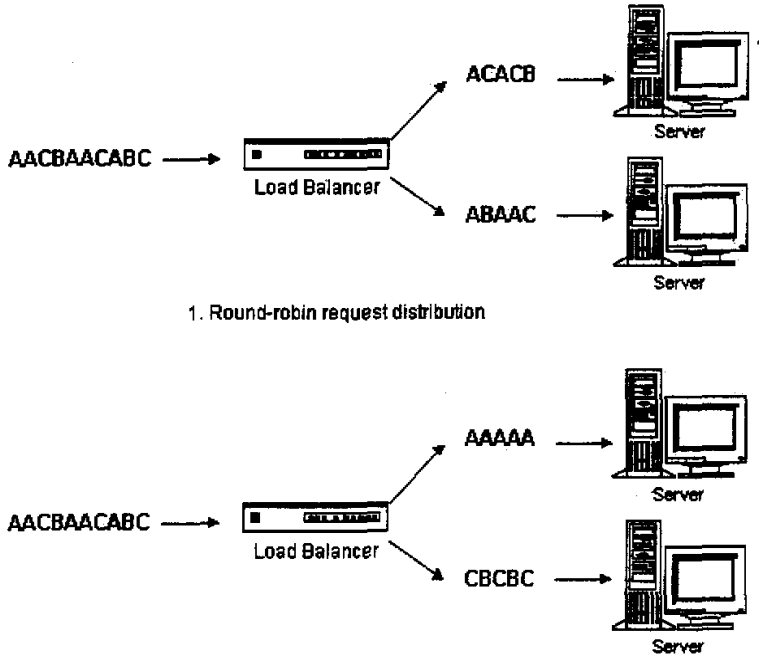

图 6.2：基于内容的请求分发

图 6.2 举例说明了基于内容请求分发可以提高后端服务器的 Cache 命中率。在例子中，一个有两台后端服务器的集群来处理进来的请求序列 AACBAACABC。在基于内容的请求分发中，调度器将所有请求 A 发到后端服务器 1，将请求 B 和 C 序列发到后端服务器 2。这样，有很大的可能性请求所要的对象会在后端服务器的 Cache 中找到。相反在轮叫请求分发中，后端服务器都收到请求 A、B 和 C，这增加了 Cache 不命中的可能性。因为分散的请求序列会增大工作集，当工作集的大小大于后端服务器的主存 Cache 时，导致 Cache 不命中。

在基于内容的请求分发中，不同的请求被发送到不同的服务器，当然这有可能导致后端服务器的负载不平衡，也有可能导致更差的性能。所以，在设计基于内容请求分发的服务器集群中，我们不仅要考虑如何高效地进行基于内容的请求分发，而且要设计有效的算法来保证后端服务器间的负载平衡和提高单个服务器的 Cache 命中率。

§6.2 内核中的基于内容请求分发 KTCPVS

由于用户空间 TCP Gateway 的开销太大，导致其伸缩能力有限。为此，我们提出在操作系统的内核中实现 Layer-7 交换方法，来避免用户空间与核心空间的切换开销和内存复制的开销。在 Linux 操作系统的内核中，我们实现了 Layer-7 交换，称之为 KTCPVS（Kernel TCP Virtual Server）。以下几小节将介绍 KTCPVS 的体系结构、实现方法、负载平衡和高可用特征等。

6.2.1 KTCPVS 的体系结构

KTCPVS 集群的体系结构如图 6.3 所示：它主要由两个组成部分，一是 KTCPVS 交换机，根据内容不同将请求发送到不同的服务器上；二是后端服务器，可运行不同的网络服务。KTCPVS 交换机和后端服务器通过 LAN/WAN 互联。

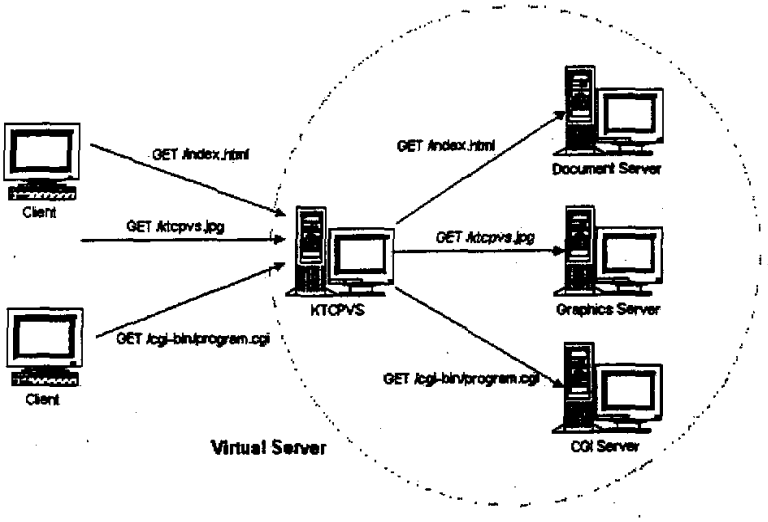

图 6.3 ：KTCPVS 集群的体系结构

KTCPVS 交换机能进行根据内容的调度，将不同类型的请求发送到不同的后端服务器，再将结果返回给客户，后端服务器对客户是不可见的。所以，KTCPVS 集群的结构对客户是透明的，客户访问集群提供的网络服务就像访问一台高性能、高可用的服务器一样，故我们也称之为虚拟服务器。客户程序不受服务器集群的影响不需作任何修改。

6.2.2 KTCPVS 实现

在 Linux 内核 2.4 中，我们用内核线程（Kernel thread）实现 Layer-7 交换服务程序，并把所有的程序封装在可装卸的 KTCPVS 模块。KTCPVS 的主要功能模块

如图 6.4 所示：当 KTCPVS 模块被加载到内核中时，KTCPVS 主内核线程被激活，并在/proc 文件系统和 setsockopt 上挂接 KTCPVS 的内核控制程序，用户空间的管理程序 tcpvsadm 通过 setsockopt 函数来设置 KTCPVS 服务器的规则，通过/proc 文件系统把 KTCPVS 服务器的规则读出。基于内容调度的模块被做成可装卸的模块（Loadable Module），不同的网络服务可以使用不同的基于内容调度模块，如 HTTP 和 RTSP 等。另外，系统的结构灵活，用户也可以为自己不同类型的网络服务编写相应的模块。

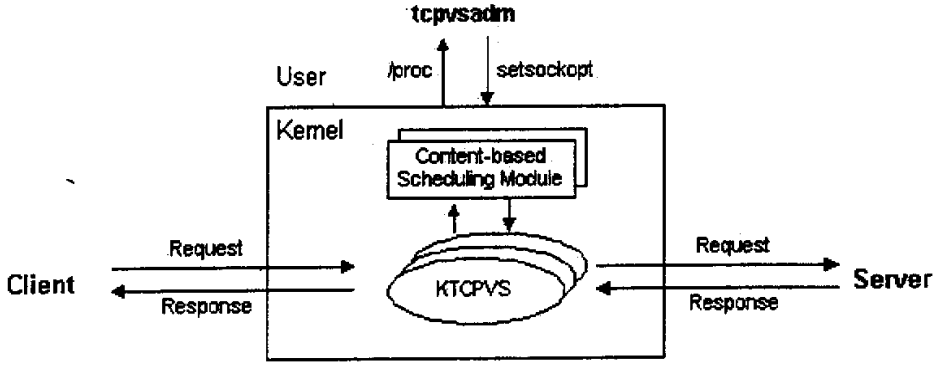

图 6.4：KTCPVS 系统的主要功能模块

可以通过 tcpvsadm 命令使得 KTCPVS 主线程生成多个子线程监听在某个端口。这组子线程可以接受来自客户的请求，通过与其绑定的基于内容的调度模块获得能处理当前请求的服务器，并建立一个 TCP 连接到该服务器，将请求发到服务器；子线程获得来自服务器的响应结果后，再将结果返回给客户。所有的请求和响应数据处理都是在操作系统的内核中进行的，所以没有用户空间与核心空间的切换开销和内存复制的开销，其处理开销比用户空间的 TCP Gateway 小很多。

6.2.3 高可用性

我们采用了两种方法来检测故障。第一，资源监测器每隔 t 个毫秒对每个服务器发 ARP（Address Resolve Protocol）请求，若有服务器过了 r 毫秒没有响应，则说明该服务器已发生故障，资源监测器通知调度器将该服务器的所有服务进程调度从调度列表中删除。第二，资源监测器定时地向每个服务进程发请求，若不能返回结果则说明该服务进程发生故障，资源监测器通知调度器将该服务进程调度从调度列表中删除。资源监测器能通过电子邮件或传呼机向管理员报告故障，一旦监测到服务进程恢复工作，通知调度器将其加入调度列表进行调度。通过检测节点或服务进程故障和正确地重置系统，可以将部分结点或软件故障对用户屏蔽掉，从而实现系统的高可用性。

§6.3 调度算法

KTCPVS 的负载平衡调度是以 TCP 连接为粒度的。同一用户的不同连接可能会被调度到不同的服务器上，所以这种细粒度的调度可避免不同用户的访问差异引起服务器间的负载不平衡。

在调度算法上，我们先实现了以下调度算法：

- 轮叫调度（Round-Robin Scheduling）
- 加权轮叫调度（Weighted Round-Robin Scheduling）
- 最小连接调度（Least-Connection Scheduling）
- 加权最小连接调度（Weighted Least-Connection Scheduling）

这些算法都较容易实现。另外，KTCPVS 交换机网络配置比 Layer-4 交换机的简单，KTCPVS 交换机和服务器只要有网络相连，能进行 TCP/IP 通讯即可，安装使用比较方便，所以在整个系统规模不大（例如不超过 10 个结点）且结点提供的服务相同时，可以利用以上算法来调度这些服务器，KTCPVS 交换机仍是不错的选择。同时，在实际的性能测试中，这些调度算法可以用作比较。

我们提出基于局部性的最小连接调度（Locality-Based Least-Connection Scheduling）和基于内容的调度（Content-based Scheduling）算法。基于局部性的最小连接调度是假设后端服务器都是相同的，在后端服务器的负载基本平衡情况下，尽可能将相同的请求分到同一台服务器，以提高后端服务器的访问局部性，从而提高后端服务器的 Cache 命中率。基于内容的调度是考虑后端服务器不相同时，若同一类型的请求有多个服务器可以选择时，将请求负载均衡地调度到这些服务器上。

6.3.1 轮转调度算法

轮转调度算法是假设所有服务器处理性能均相同，依次将请求调度不同的服务器，算法简单，但不适用于服务器组中处理性能不一的情况。

6.3.2 加权轮叫调度算法

加权轮叫调度算法可以解决服务器间性能不一的情况，它用相应的权值表示服务器的处理性能，服务器的缺省权值为 1。加权轮叫调度算法是按权值的高低和轮叫方式分配请求到各服务器。权值高的服务器先收到的连接，权值高的服务器比权值低的服务器处理更多的连接，相同权值的服务器处理相同数目的连接数。

6.3.3 最小连接调度

最小连接调度（Least-Connection Scheduling）是把新的连接请求分配到当前连

接数最小的服务器。最小连接调度是一种动态调度算法，它通过服务器当前所活跃的连接数来估计服务器的负载情况。调度器需要记录各个服务器已建立连接的数目，当一个请求被调度到某台服务器，其连接数加 1；当连接中止或超时，其连接数减一。

6.3.4 加权最小连接调度

加权最小连接调度是最小连接调度的超集，各个服务器用相应的权值表示其处理性能。服务器的缺省权值为 1，系统管理员可以动态地设置服务器的权值。加权最小连接调度在调度新连接时尽可能使服务器的已建立连接数和其权值成比例。

6.3.5 基于局部性的最小连接调度

在基于局部性的最小连接负载调度（Locality-Based Least-Connection Scheduling）中，我们假设任何后端服务器都可以处理任一请求。算法的目标是在后端服务器的负载平衡情况下，提高后端服务器的访问局部性，从而提高后端服务器的主存 Cache 命中率。

在 WEB 应用中，来自不同用户的请求很有可能重叠，WEB 访问流中存在空间的局部性，容易获得 WEB 请求的 URL，所以，LBLC 算法主要针对 WEB 应用。静态 WEB 页面的请求格式如下：

<scheme>://<host>:<port>/<path>

其中，区分页面不同的变量是文件的路径<path>，我们将之记为请求的目标。CGI 动态页面的格式如下[95, 96]：

<scheme>://<host>:<port>/<path>?<query>

我们将其中的 CGI 文件路径<path>记为请求的目标。因为 CGI 程序一般要读一个或者多个文件来生成一个动态页面，若在服务器负载基本平衡情况下将相同的 CGI 请求发送到同一服务器，可以提高服务器文件系统的 Cache 命中率。

LBLC 算法的基本流程如下：

```
while (true) {
  get next request r;
  r.target = { extract path from static/dynamic request r };
  if (server[r.target] == NULL) then {
    n = { least connection node };
    server[r.target] = n;
  } else {
    n = server[r.target];
    if (n.conns > n.high && a node with node.conns < node.low) ||
      n.conns >= 2*n.high then {
```

```
        n = { least connection node };
        server[r.target] = n;
      }
    }
    if (r is dynamic request) then
      n.conns = n.conns + 2;
    else
      n.conns = n.conns + 1;
    send r to n and return results to the client;
    if (r is dynamic request) then
      n.conns = n.conns - 2;
    else
      n.conns = n.conns - 1;
  }
```

在算法中，我们用后端服务器的活跃连接数（n.conns）来表示它的负载，server是个关联变量，它将访问的目标和服务器对应起来。在这里，假设动态页面的处理是静态页面的 2 倍。开销算法的意图是将请求序列进行分割，相同的请求去同一服务器，可以提高访问的局部性；除非出现严重的负载不平衡了，我们进行调整，重新分配该请求的服务器。我们不想因为微小或者临时的负载不平衡，进行重新分配服务器，会导致 Cache 不命中和磁盘访问。所以，“严重的负载不平衡”是为了避免有些服务器空闲着而有服务器超载了。

我们定义每个结点有连接数目的高低阀值，当结点的连接数（node.conns）小于其连接数低阀值（node.low）时，表明该结点有空闲资源；当结点的连接数（node.conns）大于其连接数高阀值（node.high）时，表明该结点在处理请求时可能会有一定的延时。在调度中，当一个结点的连接数大于其高阀值并且有结点的连接数小于其低阀值时，重新分配，将请求发到负载较轻的结点上。还有，为了限制结点过长的响应延时，当结点的连接数大于 2 倍的高阀值时，将请求重新分配到负载较轻的结点，不管是否有结点的连接数小于它的低阀值。

我们在调度器上进行限制后端服务器的正在处理连接数必须小于Σnode.high，这样可以避免所有结点的连接数大于它们的 2 倍的高阀值。这样，可以保证当有结点的连接数大于 2 倍的高阀值时，必有结点的连接数小于其高阀值。

正确选择结点的高低阀值是根结点的处理性能相关的。在实践中，结点的低阀值应尽可能高来避免空闲资源，否则会造成结点的低吞吐率；结点的高阀值应该一个较高的值并且结点的响应延时不大。选择结点的高低阀值是一个折衷平衡的过程，结点的高低阀值之差有一定空间，这样可以限制负载不平衡和短期负载不平衡，而不破坏访问的局部性。对于一般结点，我们选择高低阀值分别为 30 和 60；对于性能很高的结点，可以将其阀值相应调高。

在基本的 LBLC 算法中，在任何时刻，任一请求目标只能由一个结点来服务。

然而，有可能一个请求目标会导致一个后端服务器进入超载状态，这样比较理想的方法就是由多个服务器来服务这个文档，将请求负载均衡地分发到这些服务器上。这样，我们设计了带复制的 LBLC 算法，其流程如下：

```
while (true) {
  get next request r;
  r.target = { extract path from static/dynamic request r };
  if (serverSet[r.target] ==  ∅) then {
    n = { least connection node };
    add n to serverSet[r.target];
  } else {
    n = {least connection node in serverSet[r.target]};
    if (n.conns > n.high && a node with node.conns < node.low) ||
      n.conns >= 2*n.high then {
      n = { least connection node };
      add n to serverSet[r.target];
    }
    if |serverSet[r.target]| >1
      && time()-serverSet[r.target].lastMod > K then {
      m = {most connection node in serverSet[r.target]};
      remove m from serverSet[r.target];
    }
  }
  if (r is dynamic request) then
    n.conns = n.conns + 2;
  else
    n.conns = n.conns + 1;
  send r to n and return results to the client;
  if (r is dynamic request) then
    n.conns = n.conns - 2;
  else
    n.conns = n.conns - 1;
  if (serverSet[r.target] changed) then
    serverSet[r.target].lastMod = time();
}
```

这个算法与原来算法的差别是调度器维护请求目标到一个能服务该目标的结点集合。请求会被分发到其目标的结点集中负载最轻的一个，调度器会检查是否发生结点集中存在负载不平衡，若是，则挑选所有结点中负载最轻的一个，将它加入该目标的结点集中，让它来服务该请求。另一方面，当请求目标的结点集有多个服务

器，并且上次结点集的修改时间之差大于 K 秒时，将最忙的一个结点从该目标的结点集中删除。在实验中，K 的缺省值为 60 秒。

6.3.6 基于内容的调度

在基于内容的调度（Content-based Scheduling）中，不同类型的请求会被送到不同的服务器，但是同一类型的请求有多个服务器可以选择，例如，CGI 应用往往是 CPU 密集型的，需要多台 CGI 服务器去完成，这时需要将请求负载均衡地调度到这些服务器上。其基本算法如下：

```
while (true) {
  get next request r;
  extract path from static/dynamic request and set r.target;
  if (definedServerSet[r.target] == ∅) then
    n = {least connection node in defaultServerSet};
  else
    n = {least connection node in definedServerSet[r.target]};
  send r to n and return results to the client;
}
```

当请求目标有定义的结点集，即该类型的请求有一些服务器能处理，从该结点集中挑选负载最轻的一个，把当前的请求发到该服务器。当请求目标没有定义好的结点集，则从缺省的结点集中选负载最轻的结点，由它来处理该请求。

在上面的算法中，只考虑了结点间的负载平衡和结点上静态分割好的局部性，没有考虑动态访问时的局部性。我们对该算法改进如下：

```
while (true) {
  get next request r;
  r.target = { extract path from static/dynamic request r };
  if (definedServerSet[r.target] == ∅) then
    staticServerSet = defaultServerSet;
  else
    staticServerSet = definedServerSet[r.target];
  if (serverSet[r.target] ==  ∅) then {
    n = { least connection node in statisServerSet};
    add n to serverSet[r.target];
  } else {
    n = {least connection node in serverSet[r.target]};
    if (n.conns > n.high && a node in staticServerSet with
        node.conns < node.low) ||
       n.conns >= 2*n.high then {
```

```
            n = { least connection node in staticServerSet};
            add n to serverSet[r.target];
        }
        if |serverSet[r.target]| >1
            && time()-serverSet[r.target].lastMod > K then {
            m = {most connection node in serverSet[r.target]};
            remove m from serverSet[r.target];
        }
    }
    if (r is dynamic request) then
        n.conns = n.conns + 2;
    else
        n.conns = n.conns + 1;
    send r to n and return results to the client;
    if (r is dynamic request) then
        n.conns = n.conns - 2;
    else
        n.conns = n.conns - 1;
    if serverSet[r.target] changed then
        serverSet[r.target].lastMod = time();
}
```

6.3.7 调度算法的性能模拟

为了研究各个调度算法对整个系统性能的影响，我们对 WRR、WLC、LBLC 和 LBLC/R 的性能进行模拟。以上算法都假设各个后端服务器能处理任一请求，我们也假设调度器不会成为系统的瓶颈。

我们实验用的 Web 服务器是 Pentium II 350MHz 和 128M 内存的机器，运行 Linux 内核 2.2.17。一个 TCP 连接的建立和拆断分别需要 140us 的 CPU 时间，传输 512K 字节需要 40us。一个磁盘的寻道延时为 28ms，磁盘每 4K 字节的传输时间为 410us。Web 服务器采用 LRU 的淘汰策略。

我们将实验室多台 Web 服务器的两个月日志记录，去掉 CGI 等动态的访问请求（因为动态页面很难估计它的处理时间），归并成一个大的日志文件。日志中含有 37804 个请求目标，覆盖的数据为 1629M 字节。然后，用它对以上算法进行性能模拟。模拟结果如图 6.5 所示。

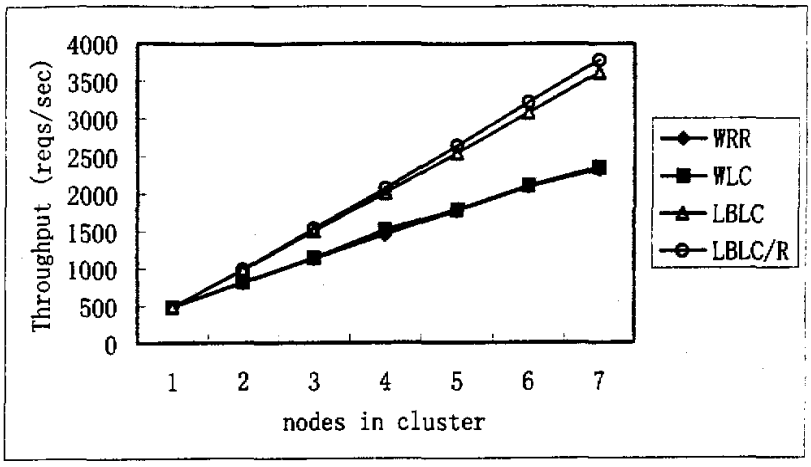

图 6.5 ：调度算法的性能模拟

在图中，我们可以看出 WRR 和 WLC 的性能较低，因为过高的 Cache 不命中率使得个服务器的性能降低，导致整个系统性能较低。随着集群中的结点数目增加，LBLC 的性能优势越来越明显，这归功于 LBLC 将请求集分割多个子集，每个子集有较好的局部性，后端服务器的 Cache 命中率高，使得系统的整体性能较高。当结点数为 7 时，LBLC 的性能比 WRR 高出 62.9%。LBLC/R 略优于 LBLC，因为避免负载不平衡出现时过多的服务器重新分配。

§6.4 KTCPVS 服务器程序的体系结构

操作系统和服务器程序的体系结构对服务器程序的性能有着重大的影响[97, 98, 99, 100, 101]。在这小节中，将简要地描述几种服务器的体系结构，针对高并发度的应用，我们在 KTCPVS 调度器中引入对称多线程事件驱动的体系结构（Symmetric Multiple-Thread Event-Driven Architecture），结合 Linux 内核提供的机制高效地实现 KTCPVS 服务器程序。

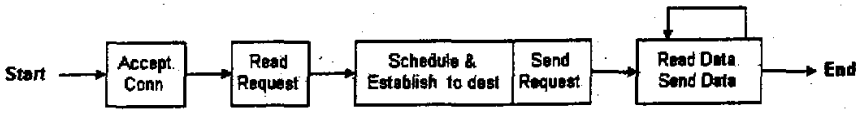

图 6.6：调度器处理请求的简单步骤

图 6.6 说明 KTCPVS 服务器程序处理一个请求到结束的几个步骤，稍详细的步骤如下：

- 接受连接：通过 accept 操作来接受进来的客户连接请求，并为之建立相应的 socket。
- 读入请求：从建立连接的 socket 中读出客户的请求。
- 调度和建立连接到目标服务器：分析请求的内容，选出相应的目标服务器，并建立一个 TCP 连接到该服务器。

- 发送请求：将请求发送到目标服务器。
- 读出数据与发送数据：从到服务器的连接中读出响应数据，通过到客户的连接返回数据。

在以上的几个步骤中，都有可能造成阻塞，例如，当期望的数据没有到达时，接受连接和读入数据都会阻塞；建立到目标服务器的连接会阻塞；当 TCP 的发送缓冲不够，发送数据会阻塞。所以，我们必须设计良好的服务器结构 CPU 的处理时间重叠起来，提高调度器的性能。

6.4.1 多进程结构

在多进程结构（Multiple-Process Achitecture）中，每个进程串行地执行一个请求从开始到结束的几个步骤，然后再处理一个新的请求。这样，多个进程可以并发地执行多个服务请求，通过操作系统的调度将 CPU 的时间重叠起来，当一个活跃进程阻塞时，操作系统会切换到另一个进程。多进程结构如图 6.7 所示。

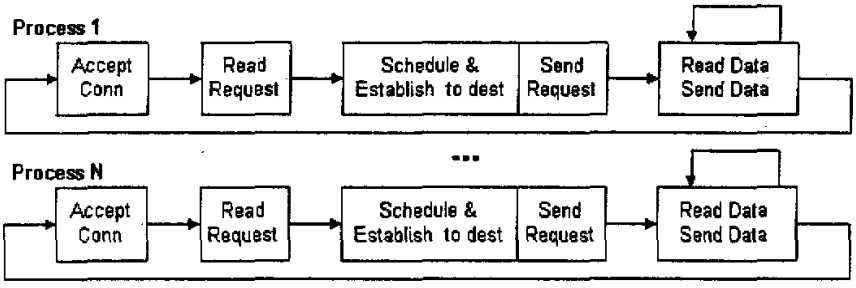

图 6.7：多进程结构

在多进程结构中，每个进程都有它私有的地址空间，相互独立，不存在任何同步问题。进程间进行共享信息的开销比较大，比较难进行系统的优化。系统的性能严重依赖于操作系统的调度，一般来说进程的切换开销比较大。

从编程的角度来看，多进程结构很自然，容易实现。

6.4.2 多线程结构

在多线程结构（Multiple-Thread Architecture）中，线程执行一个请求从开始到结束的几个步骤，然后再处理一个新的请求，许多线程共享一个地址空间，并发地执行多个服务请求。其结构如图 6.8 所示。

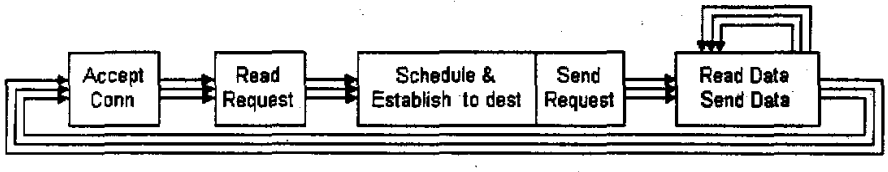

图 6.8：多线程结构

多个线程间可以共享全局变量，共享信息比较容易，但线程间需要同步机制来

访问共享变量。当一个线程网络访问阻塞时，相同空间中的其他线程可以照样执行。多线程结构需要操作系统在内核中提供线程支持，但目前 Linux 和 FreeBSD 操作系统都只提供用户层的线程支持。

虽然线程的切换开销要比进程的切换开销要小一些，但对于高并发度的 KTCPVS 服务器程序，它会处理成千上万个并发连接，成千上万个线程间的切换开销会非常大，当系统的负载比较高，线程的数目超过系统支持的能力（一般为几百个），系统的性能会剧烈下降。

从编程的角度来看，多线程结构也比较自然，只要考虑线程间同步问题，也比较容易实现。

6.4.3 单进程事件驱动结构

单进程事件驱动结构（Single-Process Event-Driven Architecture）使用一个事件驱动的进程来并发地处理多个请求。进程执行非阻塞的系统调用，如 BSD Unix 中的 select 和 System V 中的 poll，来查看 I/O 事件。当 I/O 事件就绪时，再进行 I/O 操作。单进程事件驱动结构如图 6.9 所示。

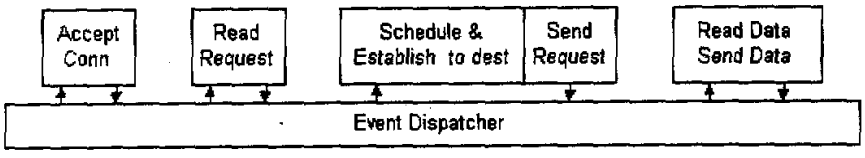

图 6.9：单进程事件驱动结构

在原理上，单进程事件驱动可以将 CPU 执行时间重叠起来，同时避免上下文切换的开销和线程间同步的开销，在负载比较高时，事件匹配可以摸平访问的毛刺，系统提供一个稳定的吞吐率。所以说单进程事件驱动结构比较高效，比较快的 Web 服务器程序如 Zeus[102]和 boa[103]等都采用此结构。

但是，操作系统往往不能提供完全异步的操作。建立到后端服务器的 TCP 连接时，会阻塞。当 TCP 发送缓冲满时，发送数据会阻塞。页面故障和垃圾回收导致进程被挂起往往是不可避免的。当事件队列很长时，事件匹配的开销也会增大，一些研究[104, 105]提出优化 select 操作从而提高整体性能。另外，单进程事件驱动不适合在 SMP 结构的机器。

从编程的角度来看，事件驱动结构需要较高编程技巧，必须将所有可能会导致阻塞的操作找出来，使用非阻塞的系统调用；找出任务的不同状态；维护一个事件队列，进程不断地从队列取出任务，根据任务的状态作相应的处理。

6.4.4 多线程事件驱动结构

为充分利用单进程事件驱动结构和多线程结构的优点，避免它们的缺点，我们引入一个混合结构——多线程事件驱动结构（Multiple-Thread Event-Driven

Architecture)。多线程事件驱动结构如图 6.10 所示，系统中有多个线程，每个线程有本地的事件匹配，它从共享的监听 Socket 上接受连接后，将它加入本地的事件匹配器中，线程以事件驱动的方式处理请求。线程间只有在接受连接时需要同步，在本地的事件匹配时不需要同步机制，可以避免竞争条件（Race Condition）和死锁（Deadlock）的出现。当一个线程阻塞后，系统会切换到另一个线程。它可以利用 SMP 结构，让多个线程在多个处理器上执行，提高整体性能。

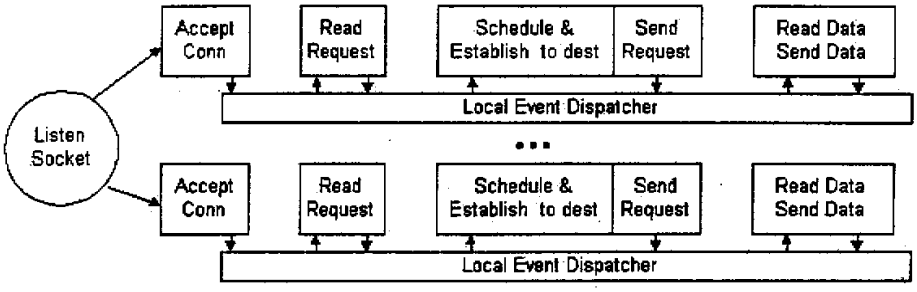

图 6.10：多线程事件驱动结构

多线程事件驱动结构带来的问题是多个线程切换有开销。这是一个折衷的问题，我们要利用事件驱动的高效，避免单个线程因为有些 I/O 操作导致阻塞，引入多个进程，同时要避免多个线程频繁切换开销。所以，我们一般在单处理器的系统上执行 2~3 个事件驱动进程，在双处理器的系统上启动 4~6 个事件驱动进程。

在 KTCPVS 的实现中，我们采用 Linux 的内核线程（Kernel Thread）来实现多线程事件驱动结构。Linux 的内核线程不同于一般的线程概念，它是介于进程和线程之间的执行体。在统一的操作系统内核空间中，多个内核线程共享操作系统的全局变量，但它们的执行是相互独立的，不会因为一个内核线程被挂起，所有的内核线程被挂起。

§6.5 持久 HTTP 连接的处理

在这小节中，我们将说明持久 HTTP 连接带来的问题，和 KTCPVS 中的相应解决方法。

6.5.1 HTTP/1.1 持久连接

一个 HTML 文档往往含有多个对象，如文档中的图像文件等，所以得到一个 HTML 文档需要向 Web 服务器发送多个请求。在 HTTP/1.0 协议中，浏览器需要为每一个请求建立一个 TCP 连接。因为 TCP 的三次握手的建立和挂断开销大，这会导致增加客户方的延时，增加网络传输的负担，也增加服务器的资源需求。

HTTP/1.1 协议[106]使得浏览器在一个 TCP 连接中向服务器发多个请求。服务器完成一个请求后，保持连接一般为 15 秒，这个时间是可设置的。当第二个请求到

达时，服务器将响应数据通过已有的 TCP 连接返回给客户。同时，HTTP/1.1 协议许可客户将一系列请求通过流水线方式发给服务器，服务器将多个响应数据通过已有的 TCP 连接返回给客户。这样，可以避免多个 TCP 的建立开销和慢启动特性(Slow Startup)，提高网络的使用效率，客户也会感知到好的服务质量。

6.5.2 持久 HTTP 连接对基于内容请求分发带来的问题

在基于内容请求分发中，不同的请求可能被发送到不同的服务器，一方面是为提高后端服务器上的访问局部性，另一方面也可能是一些请求只能由某些服务器处理。在 HTTP/1.1 协议中，多个请求可能来自一个 TCP 连接，而基于内容请求分发一般是以 TCP 连接为粒度的，这会使得这些请求都被发送到同一台服务器。这会影响 LBLC 等算法的性能；在静态设置中，一些请求只能在某些服务器处理，这会造成严重的问题。

6.5.3 解决的策略

调度器尽可能维持一系列持久的连接到各个后端服务器。当多个请求从一个 TCP 连接到达时，调度器根据请求的目标来选择服务器，若调度器到该服务器存在持久连接，调度器利用该连接将请求发给服务器；若不存在，调度器建立一个持久连接到该服务器，并记录给连接，将请求发给该服务器。当响应结果从服务器返回到调度器时，调度器再依次从到客户的连接将结果返回给客户。

这种方法对客户是透明的，也不需要修改服务器程序。同时，利用持久 TCP 连接可以降低调度器到后端服务器之间 TCP 连接的建立和挂断的开销。

§6.6 性能测试与比较

在性能测试上，我们对用户空间的 TCP Gateway 和 KTCPVS 交换机的吞吐率和系统利用率进行测试对比。测试的硬件环境是八台 PC 机通过一个 Intel EtherExpress 10/100 交换机相连。一台 PC 作为 Layer-7 交换机，运行 KTCPVS 或者用户空间的 TCP Gateway 程序，它的配置为 Pentium III 550MHz 处理器、128M 内存和两块 Intel ExpressPro 100B 网卡。四台 PC 作为客户机，来生成 Web 访问的工作负载，运行 Linux 操作系统，它们的配置不一，处理器的配置都在 Celeron 400MHz 以上。三台 PC 为服务器，运行 RedHat Linux 6.2 操作系统和 Apache 1.3.9 Web 服务器软件，它们的配置为 Pentium III 550MHz 处理器、128M 内存和 Intel ExpressPro 100B 网卡。

WEB 服务器测试软件采用惠普实验室的 httperf 程序[107]，原因是 httperf 能从一个文件中读出一个 URL 列表，按照 URL 列表对 WEB 服务器进行连续访问，或者

根据 URL 列表和相应的参数生成访问序列，并且 httperf 能接受一些参数，如生成并发访问请求的个数和访问间隔的思考时间等。我们在 httperf 程序基础上实现了分布式测试控制程序 DHP（Distributed Httperf），它有一个主控进程和多个执行进程，例如在四台运行 DHP 的执行进程，在任一台机器上运行主控进程。主控进程接受用户输入的参数和指令，将这些参数和命令发送四个执行进程，四个执行进程将同时根据相应的参数执行 httperf，生成访问序列，对设定的 WEB 服务器进行访问。

用户空间的 TCP Gateway 程序我们采用 Squid-2.3-Stable1，操作系统为 Red Hat Linux 6.2，我们对 Squid 的配置文件进行设置，使之成为一个用户空间的 Layer-7 交换机。在 KTCPVS 交换机上，我们采用 Linux 内核 2.4.0-test7，将 KTCPVS 模块加载内核中，并启动 30 个子线程来处理到达交换机的 WEB 请求。

我们在不同的请求文件长度下，测试基于 Squid 的 Layer-7 交换机和 KTCPVS 交换机的吞吐率和系统利用率。为了使用测试结果更加正确，对于每一次测试我们都要重复进行多次，最后对测试结果进行归并，如表 6.1 所示。

表 6.1: 基于 Squid 的 Layer-7 交换机和 KTCPVS 交换机的性能测试结果

请求文件长度	基于 Squid 的 Layer-7 交换机		KTCPVS 交换机	
（Bytes）	每秒连接数	CPU 利用率	每秒连接数	CPU 利用率
1,000	1,021	98.67%	2,124	54.56%
2,000	950	99.21%	1,715	51.27%
5,000	667	98.37%	1,306	46.89%
10,000	472	96.80%	1,032	38.39%
20,000	334	97.83%	536	27.23%
30,000	248	99.14%	364	13.42%
300,000	27	99.77%	39	1.81%

在表 6.1 上，我们可以看到基于 Squid 的 Layer-7 交换机已经满负荷运行了，CPU 利用率的利用率接近 100%；随着请求文件长度的增长，每秒的连接数逐渐下降，但是系统的吞吐率（Bytes/Second）呈上升趋势；可见，连接生成和不同连接之间调度的开销是很大的。在 KTCPVS 交换机上，我们没有能够使得 CPU 满负荷运转，测试它的最大吞吐率和每秒连接数，这是由于我们的服务器数目和 100Mbps 的网络受限。当请求文件长度增长时，系统的吞吐率也增长。当请求文件长度等于 10,000 字节时，系统的吞吐率为 10.32Mbytes/Second，这时 100Mbps 网络的带宽几乎被用尽。此时，Linux 操作系统的 TCP/IP 协议栈也可能到达极限，我们会在这方面继续探索、优化和测试 KTCPVS 的性能。

根据经验统计数据显示平均的 WEB 访问对象长度为 10K 字节[108]。在表 1 上，我们可以看到 KTCPVS 交换机比基于 Squid 的 Layer-7 交换机，在每秒连接数上有 118.64%的增长，在 CPU 利用率上下降了 58.41%。

§6.7 本章小结

本章对基于内容请求分发（Layer-7 交换）的已有方法进行分析比较，指出了

它们存在的不足；并提出了在操作系统内核中利用内核线程高效地实现 Layer-7 交换的解决方法；请求的处理和调度以及响应的处理都是在内核进行的，避免了多次内核和用户层的切换和内存复制开销，从而提高系统的吞吐率。

提出了基于局部性的最小连接调度算法 LBLC，算法假设后端服务器都是相同的，算法的目标是在后端服务器的负载基本平衡情况下，尽可能将相同的请求分到同一台服务器，以提高后端服务器的访问局部性，从而提高后端服务器的主存 Cache 命中率。在不同类型服务器的调度中，我们给出基于内容的调度算法，若同一类型的请求有多个服务器可以选择时，将请求负载均衡地调度到这些服务器上。

详细阐述了内核 Layer-7 交换机 KTCPVS 的实现。分析了多进程、多线程和单进程事件驱动的服务器体系结构的优缺点，引入多线程和单进程事件驱动的混合结构——多线程事件驱动结构，并采用该结构在内核中实现了 KTCPVS。KTCPVS 支持持久 HTTP 连接处理，并通过检测结点或服务进程故障和正确地重置系统达到高可用性也。无需对客户机和服务器作任何修改，可适用任何 Internet 站点。

最后，对用户空间的 TCP Gateway 和 KTCPVS 交换机的性能进行测试和比较，实验数据表明在内核中实现基于内容请求分发的方法具有良好的性能。

第七章 大规模基于内容请求分发的系统

在上一章中，讲了在内核中基于内容请求分发的技术和实现。虽然它的性能远比其他类似系统高，但其伸缩性依然有所限制。本章将讲述大规模基于内容请求分发系统的体系结构和实现，及其该体系结构在大规模 Cache 集群系统中的应用。

§7.1 引言

在基于内容请求分发中，调度器可以根据请求的内容或者请求的类型决定哪一台后端服务器来处理服务请求。这可以带来以下好处：

- 相同页面的请求被发送到同一台的服务器，很有可能页面的请求会被服务器缓存，可以提高单台服务器的 Cache 使用效率。
- 一些研究表明 WEB 访问流中存在空间的局部性。基于内容请求分发可以充分利用访问的局部性，将相同类型的请求发送到同一台服务器，使得每个后端服务器收到的请求相似性好，有利于进一步提高单台服务器的 Cache 使用效率，从而在有限的硬件配置下提高系统的整体性能。
- 后端的服务器可运行不同类型的服务，如文档服务，图片服务，CGI 服务和数据库服务等。

为了分析请求的内容，必须在调度器建立与客户之间的 TCP 连接，来获得请求的内容。我们提出在内核中进行基于内容请求分发的技术，其性能比其他在用户空间的基于内容请求分发高出一倍多。但是，调度器必须在内核中建立一个到后端服务器的 TCP 连接和一个客户之间的 TCP 连接，来自客户的请求数据要进过调度器到后端服务器，从后端服务器返回的响应数据要通过调度器转发给客户，这些操作的开销比较大。一般来说调度器和后端服务器在相同的硬件配置情况下，当后端服务器的数目上升到 10 台时，基于内容请求分发的调度器会成为系统的瓶颈。

当一个基于内容请求分发的调度器成为系统的瓶颈时，一般的解决方法是在后端服务器组前引入多个调度器，然后用轮叫 DNS 将一个域名映射到多个调度器的 IP 地址上，如图 7.1 所示。但是，这种方法存在以下问题：

- 轮叫 DNS 存在严重的负载不平衡问题
- 还有多个调度器都是相互独立的，每台调度器所记录的访问局部性和结点的负载都是不一样的，它们的调度进一步导致后端服务器组的负载不平衡。

所以，轮叫 DNS 和多个调度器并不容易解决基于内容请求分发系统的伸缩性问题。

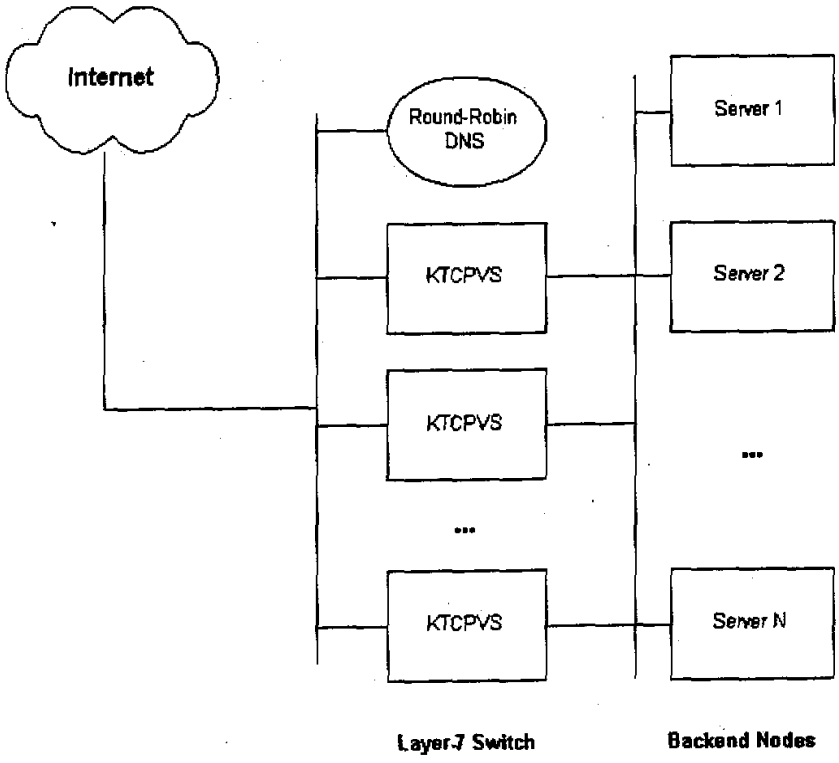

图 7.1：轮叫 DNS 和多 KTCPVS 调度器的结构

在下一节中，我们提出两种大规模基于内容请求分发系统的体系结构，其可伸缩性比单个基于内容请求分发的调度器高出一个数量级。

§7.2 体系结构

这一节解决基于内容请求分发的可伸缩性问题。首先我们要分析当前系统的瓶颈在哪里。

图 7.2 显示了基于内容请求分发的主要组成部分。在一组后端服务器前有一个 KTCPVS 调度器，调度器主要包含 KTCPVS 分发器（KTCPVS Distributor）和基于内容调度模块两部分。KTCPVS 分发器负责客户和服务器之间的数据传输，将客户的请求数据转发给服务器，将来自服务器的响应数据返回给客户。基于内容调度模块是实现基于内容请求分发的策略，它决定哪一台服务器来处理当前的请求。后端服务器运行着真正处理请求的网络服务。

我们可以看到前端调度器的大部分开销是 KTCPVS 分发器，而不是基于内容调度模块；KTCPVS 分发器所执行的任务是完全不相关的。所以，可以将开销大的 KTCPVS 分发器分到多台机器上执行，而一个基于内容调度模块来保证调度的中心控制，使得各个 KTCPVS 分发器看到全局的请求分发和后端服务器忙闲的视图。

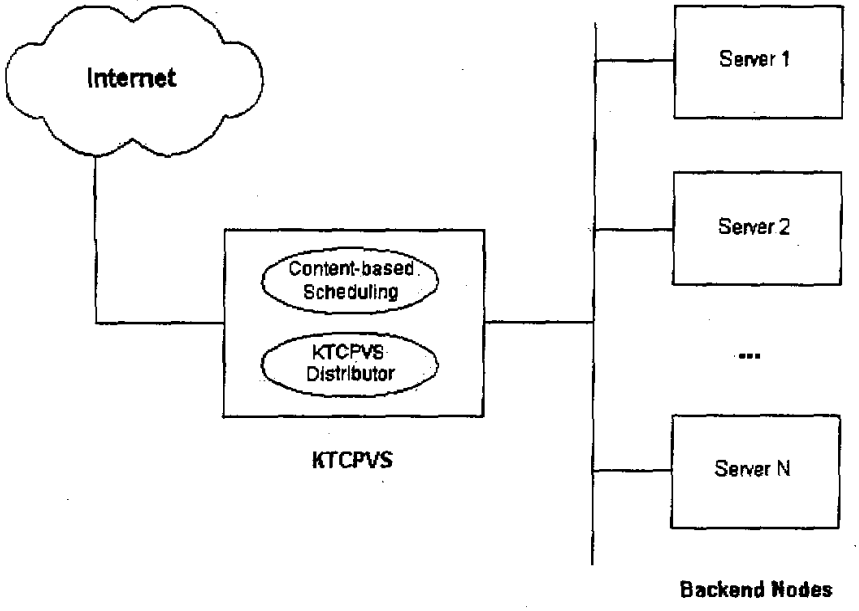

图 7.2：基于内容请求分发的组成部分

众所周知，显式地让客户选择服务器和轮叫 DNS 调度的方法会导致服务器间严重的负载不平衡。我们使用 VS/DR 和 VS/TUN 调度器将请求分发多个 KTCPVS 分发器。在前面一些章节我们知道在相同的硬件配置下 VS/DR 和 VS/TUN 调度器的吞吐率（连接数每秒）比 KTCPVS 调度器的高出将近一个数量级。

在图 7.3 中，我们给出了多个独立 KTCPVS 分发器的结构。在一组服务器前有多个 KTCPVS 分发器，它们运行在独立的机器上。在最前端，有一个 VS/DR 或 VS/TUN 调度器作为 Layer-4 交换机将请求高速地分发到多个 KTCPVS 分发器。在一个专用的服务器上运行内容位置服务（Content Location Service，以下简称 CLS），它实现基于内容请求分发的策略，决定哪一台服务器来处理请求。

当一个来自客户的请求报文到达 Layer-4 交换机时，Layer-4 交换机根据规则将请求报文转发给其中一台 KTCPVS 分发器。KTCPVS 分发器建立与客户之间的 TCP 连接，直到获得客户请求的内容，如 WEB 访问请求中 URL。这时，KTCPVS 分发器将请求的内容发给 CLS 服务器，CLS 服务器根据请求的内容选出一台后端服务器，将结果返回给 KTCPVS 分发器。KTCPVS 分发器再建立一个到后端服务器的 TCP 连接，将请求发到后端服务器。当后端服务器的响应数据返回到 KTCPVS 分发器时，KTCPVS 分发器将数据发给客户。

在这个结构中，KTCPVS 分发器需要多台专用机器。怎样高效地将集群中结点分为 KTCPVS 分发器和后端服务器依赖于工作负载的特征，而工作负载的特征是事先不为所知的。例如，当工作负载中都是负载比较大的任务（如在线数据库查询）时，有效集群配置是一些 KTCPVS 分发器加上大量的后端服务器；当其他负载时，

有可能需要多一些 KTCPVS 分发器。怎样将集群结点分为为 KTCPVS 分发器和后端服务器是需知工作负载的优化问题。不好的分割会导致资源的浪费。

独立 KTCPVS 分发器的好处是后端服务器可以运行任何支持 TCP/IP 的操作系统，系统设置和维护相对简单些，还可以将 Layer-4 交换机、多个 KTCPVS、CLS 服务器封装在一个机箱里。这样，这个机箱对用户来说就是一个大容量的 Layer-7 交换机。用户只要安装后端服务器即可。

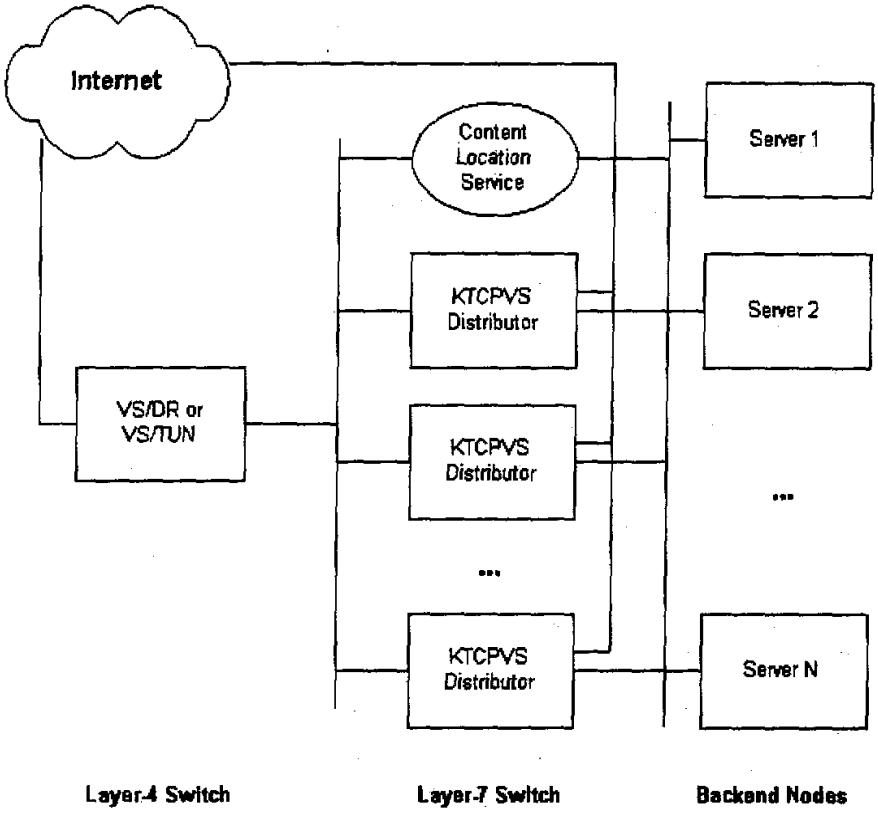

图 7.3：多个独立分发器的结构

图 7.4 显示了另一种结构，KTCPVS 分发器和服务器程序共存在集群结点上。这样，KTCPVS 分发器存在所有的服务器结点上，消除 KTCPVS 分发器成为系统瓶颈的可以，基本上不存在 KTCPVS 分发器和后端服务器的资源浪费问题。KTCPVS 分发器和服务器程序分别监听在不同的 socket 端口上，例如，建立一个用此结构的 WEB 服务集群，我们可以让 KTCPVS 分发器监听在 80 端口上，而 WEB 服务器程序监听在 8080 端口上。每个 KTCPVS 分发器建立一个持久的 TCP 连接到 CLS 服务器，KTCPVS 分发器通过该连接向 CLS 服务器查询执行当前请求的服务器。

当一个来自客户的请求报文到达 Layer-4 交换机时，Layer-4 交换机不作基于内容的调度，而是尽可能地将请求连接负载均衡转发服务器结点。在服务器结点上的 KTCPVS 分发器首先建立与客户之间的 TCP 连接，直到获得客户请求的内容，如

WEB 访问中 URL。这时，KTCPVS 分发器将请求的内容发给 CLS 服务器，CLS 服务器根据请求的内容选出一台后端服务器，将结果返回给 KTCPVS 分发器。若返回的结果是本地服务器，KTCPVS 分发器在内核中将这个 TCP 连接直接转给监听在不同端口上的服务器程序，以后不再需要 KTCPVS 分发器干预，这种转接是非常高效的。若返回的结果是异地服务器，KTCPVS 分发器建立一个到该服务器的 TCP 连接，将请求发到该服务器。当服务器的响应数据返回到 KTCPVS 分发器时，KTCPVS 分发器将数据发给客户。

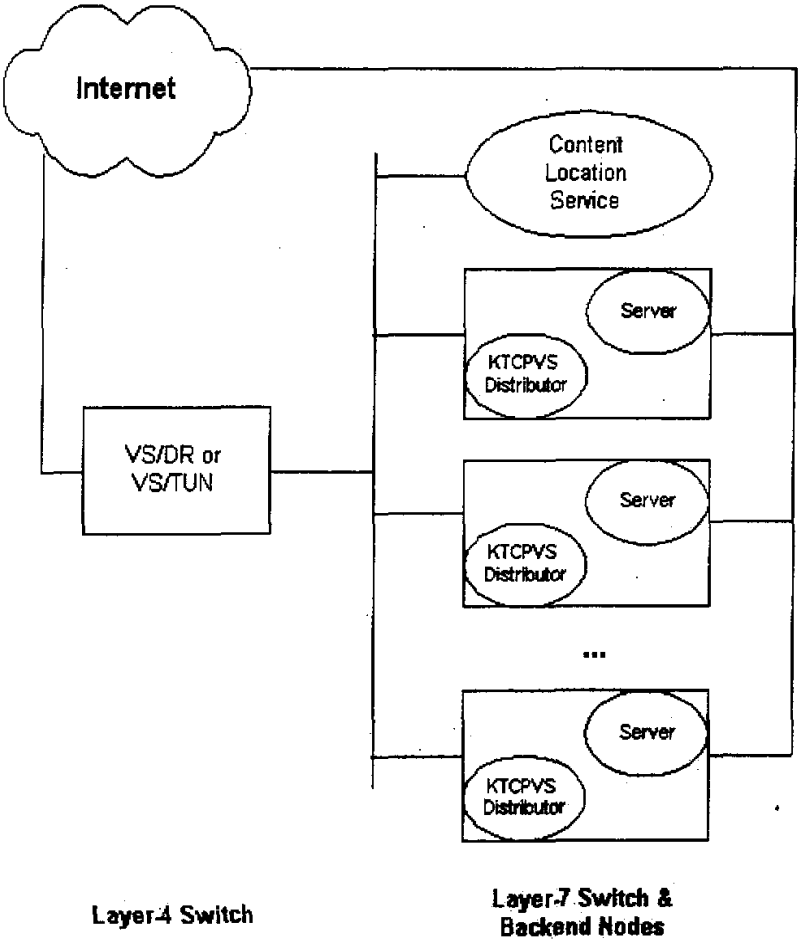

图 7.4：分发器和服务器共存的结构

§7.3 内容位置服务

在前面多个 KTCPVS 分发器需要一个中心服务，根据请求的内容分配一个合适的服务器来执行，我们将这个服务抽象为内容位置服务（Content Location Service）。内容位置服务就像 Internet 上的域名服务（Domain Name Service）一样，

域名服务是将一个域名解析到一个 IP 地址，而内容位置服务是根据输入的请求内容（如 URL 或者 URI）动态地确定该内容所在的位置。

7.3.1 内容位置服务的请求分发

CLS 服务器在调度服务器时，需要知道各个服务器的忙闲情况，以及服务器的健康状况。所以，在 CLS 服务器上运行一个资源监测器，它定时地（如每隔 2 秒钟）查询各个服务器的负载信息，计算出相应的负载值，然后将这些值转给 CLS 服务程序。当资源监测器在设定的时间内没有收到服务器的响应，资源监测器认为该服务器已经失效，通知 CLS 服务程序不再调度该服务器；当资源监测器在以后的查询中发现该服务器作正常的响应，资源监测器通知 CLS 服务程序将该服务器加入调度序列。这样，通过检测节点或服务进程故障和正确地重置系统，可以将部分结点或软件故障对用户屏蔽掉，从而实现系统的高可用性。

在多个独立分发器的结构中，我们对 LBLC/R 算法需要进行相应的改进，提出基于局部性最小负载（Locality-Based Least Loaded，以下简称为 LBLL）算法。LBLL 算法的基本流程如下，其中 T 为一个可设定的服务器负载的阀值。

```
while (true) {
  get next request r;
  if (serverSet[r.target] == ∅) then {
    n = { least loaded node };
    add n to serverSet[r.target];
  } else {
    n = {least loaded node in serverSet[r.target]};
    if (n.load > T && a node with node.load < T/2) ||
      n.load >= 2*T then {
      n = { least loaded node };
      add n to serverSet[r.target];
    }
    if |serverSet[r.target]| >1
      && time()-serverSet[r.target].lastMod > K then {
      m = {most loaded node in serverSet[r.target]};
      remove m from serverSet[r.target];
    }
  }
  if serverSet[r.target] changed then
    serverSet[r.target].lastMod = time();
  send n back to the client;
}
```

在分发器和服务器共存的结构中，我们知道 KTCPVS 分发器获得请求后，请求由本地的服务器程序还是由异地的服务器程序处理的开销差别是很大的。由此，我们对 LBLL 算法进行相应的调整，调整后的算法如下：

```
while (true) {
  get next request r;
  if (serverSet[r.target] == ∅) then {
    if (local_node.load < T) then
      n = local_node;
    else
      n = { least loaded node };
    add n to serverSet[r.target];
  } else {
    if (local_node ∈serverSet[r.target] AND local_node.load < T) then
      n = local_node;
    else {
     n = {least loaded node in serverSet[r.target]};
     if (n.load > T ) then {
       if (local_node.load < T) then
         n = local_node;
       else
         n = { least loaded node };
       add n to serverSet[r.target];
     }
     if |serverSet[r.target]| >1
       && time()-serverSet[r.target].lastMod > K then {
       m = {most loaded node in serverSet[r.target]};
       remove m from serverSet[r.target];
     }
  }
  if serverSet[r.target] changed then
    serverSet[r.target].lastMod = time();
  send n back to the client;
}
```

当请求 r 的服务器集为空集且本地结点的负载小于阀值 T 时，由本地结点处理该请求。当请求 r 的服务器集含有本地结点且本地结点的负载小于阀值 T 时，由本地结点处理该请求；否则，当服务器集中所有结点的负载大于阀值 T 时有本地结点且本地结点的负载小于阀值 T 时，由本地结点处理该请求。

7.3.2 内容位置服务的性能优化

因为内容位置服务是一个中心服务，它的性能决定整个系统的伸缩性。提高内容位置服务性能的关键之处是降低通讯开销和执行开销。主要通过以下两种方法来降低通讯开销：

1. 尽可能缩短发到 CLS 服务的消息长度
2. 尽可能在一个报文中放多个需要解析的请求

在第一种方法中，我们为每一内容引入一个内容标识（Content ID），不同的内容尽可能有不同的内容标识。MD5 算法是对文件或对象生成 128 位校验码非常好的算法，两个不同的请求内容产生相同的 MD5 校验码的可能性微乎其微，所以用 MD5 算法来生成内容标识的很好方法。KTCPVS 分发器在发送请求到 CLS 服务前，用 MD5 算法生成一个 128 位的内容标识。CLS 服务中主要对内容标识进行操作。

在第二种方法中，将多个请求放一个 TCP 报文，可以降低中断和协议处理开销。KTCPVS 分发器将一组请求发给 CLS 服务器后，等到其响应报文返回，再发另一组请求。这样，当 CLS 服务器负载比较大时，它的响应时间也比较长，KTCPVS 分发器在一个报文中发的请求数目越多，可以帮助 CLS 服务器降低负载；当 CLS 服务器负载比较小时，KTCPVS 分发器在一个报文中发的请求数目较小，系统的响应时间就短。

降低执行开销的方法是把 CLS 服务变成一个可装卸的内核模块，它完全在内核中执行，没有用户空间和内核间的切换和内存复制的开销。CLS 服务在内核中运行要比在用户空间中快一倍以上。

§7.4 性能测试

在以前章节中，我们测试得一个 VS/DR 或 VS/TUN 运行在 Pentium III 500MHz、128M 内存和 2 块 100M 网卡的机器上，报文的处理速率为 150,000 packets/second。在实际实验中，我们测得平均文件长度为 10K 的 HTTP 连接，从客户到服务器方向的报文为 6 个。这样，我们可以推出 VS/DR 或 VS/TUN 调度器的最大吞吐率为 25,000 Connections/Second。KTCPVS 交换机的吞吐率最高为 3000 Connections/Second。

我们可知整个系统的吞吐率最大可达 25,000 Connections/Second，但是我们还要查看系统其他部分是否存在瓶颈。我们没有足够的设备对整个系统的性能进行真实测试，平均长度为 10K 的连接和 25,000 Connections/Second 的速率会产生 250MBytes/Second 的网络流量，这需要上百台服务器、上百台客户机和很多 100M 交换机或者 1Gbps 交换机，才能完成真正的性能测试。从以上的体系结构中分析，在前端的 Layer-4 交换机成为瓶颈前，系统唯一可能成为瓶颈的是内容位置服务 CLS。所以，我们只对 CLS 服务的性能进行单独测试。

CLS 服务器是一台 Pentium III 500MHz、128M 内存和 100M 网卡的机器，它运

行 Linux Kernel 2.2.17，我们将 CLS 模块加载到内核中。我们编写了一个测试 CLS 服务吞吐率的小程序，可以调整一次发送的请求数目，程序向 CLS 服务发送请求，收到应答后，再向 CLS 服务发送请求，这样持续下去，直到完成所有的请求，最后程序显示 CLS 服务处理请求的速率。我们用三个程序并发地向 CLS 服务发送请求，将三个吞吐率相加起来，并调整一次发送的请求数目，再进行测试。最后，测试的数据如图 7.5 所示：

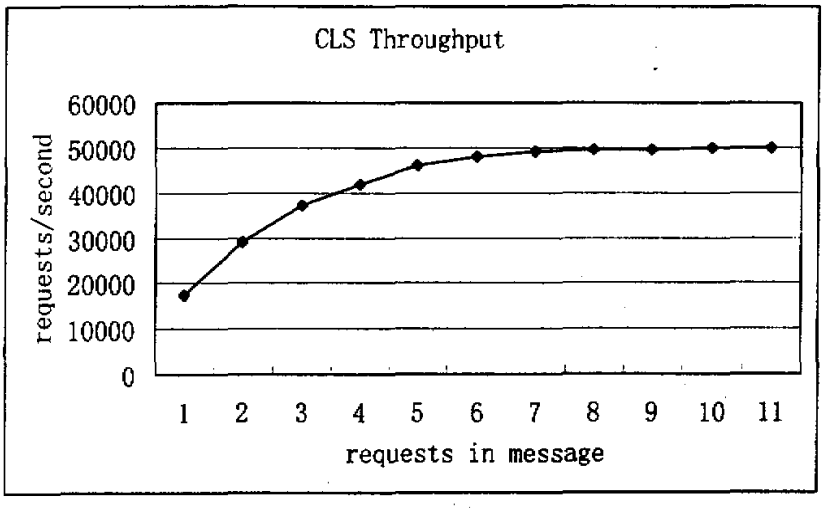

图 7.5：CLS 服务的吞吐率

测试的结果显示在 Pentium III 500MHz 的机器上 CLS 服务可达到接近 50,000 请求每秒。这个速率远超过 VS/DR 或 VS/TUN 调度器的速率，所以在 IPVS 调度器成为瓶颈前，CLS 服务不会成为系统的瓶颈。在更好的硬件配置下，CLS 服务可以达到更高的吞吐率。

§7.5 基于内容分发的大规模 Cache 集群系统

随着 Internet 爆炸式地增长，网络负载和延时的增加一直是令人头疼的问题。在现有网络带宽条件下，有效的网络 Cache 系统可以大大地减少网络流量、降低响应延时以及服务器的负载[109, 110, 111, 112]。以下一节讲述应用大规模基于内容请求分发来建设大容量 Cache 集群系统。

7.5.1 网络 Cache 技术

所谓网络 Cache 技术就是将远处服务器上的内容缓冲到离用户较近的 Cache 服务器，从而减少网络流量、降低响应延时以及服务器的负载。例如，现在有多个用户使用 Cache 服务器访问 www.yahoo.com 的首页，当第一个用户访问时，Cache 服

务器会从 Yahoo 站点取网页返回第一个用户，并将该网页存储在 Cache 中；当第二、第三个用户访问时，Cache 服务器会从本地的 Cache 中取出该网页直接返回给用户，避免了多次访问 Yahoo 站点取该网页。这样，对整个网络来说减少网络流量，对用户来说降低响应延时，对 Yahoo 站点来说降低服务器的负载。

透明 Cache 技术就是将在网络中使用 Cache 服务器，但 Cache 服务器对用户来说是透明的。换句话说，浏览器不需要将代理设置到一个 Cache 服务器。透明 Cache 的实现是通过在用户访问的必经之路上将 WEB 访问的流量转发给 Cache 服务器。不管用户愿意还是不愿意，所有的 WEB 请求被转发到 Cache 服务器。透明 Cache 技术的好处是它可以使用网络容易控制，同时降低网络的维护费用。

7.5.2 通过 Layer-4 交换机的 Cache 集群

在大的网络流量下，一台 Cache 服务器很难胜任调度其中的 WEB 请求。可以通过 Layer-4 交换机将一组 Cache 服务器组成一个 Cache 集群系统，来处理较大的网络负载。Layer-4 交换机只判断 IP 报文的目标地址，将 WEB 请求负载均衡地转发到 Cache 服务器组中。当一个 Cache 服务器失效时，Layer-4 交换机可以将该 Cache 服务器从调度序列中删除，从而将故障屏蔽。但是，Layer-4 交换机不能判断 WEB 请求的内容，这会导致很多文档在 Cache 服务器上重叠，也不能有效地利用访问请求中的局部性。

7.5.3 基于内容分发的 Cache 集群

基于内容分发的 Cache 集群可以解决 Layer-4 交换机的 Cache 集群存在的问题。它的体系结构如图 7.6 所示：KTCPVS 交换机调度一组 Cache 服务器，Cache 服务器直接与 Internet 相连，来自客户的 WEB 访问请求要经过 KTCPVS 交换机，才能访问 Internet。

基于内容分发的 KTCPVS 交换机可以分析 WEB 请求的内容，进行基于内容的调度，充分利用访问请求中的局部性。KTCPVS 交换机建立一个与客户的 TCP 连接和一个与 Cache 服务器的 TCP 连接，来完成请求；当然，KTCPVS 交换机也可以直接与源服务器（真正提供内容的服务器）建立 TCP 连接，从源服务器上直接取得访问对象，从而跳过（Bypass）Cache 服务器。所以，基于内容分发的 Cache 集群可以很好地解决以下三个问题：

- 对不可缓存的对象（Non-cacheable objects），可以跳过 Cache 服务器。
- 优化 Cache 的命中率。
- 保证 Cache 服务器在正常负载下工作，若有 Cache 服务器超载，则跳过该 Cache 服务器。

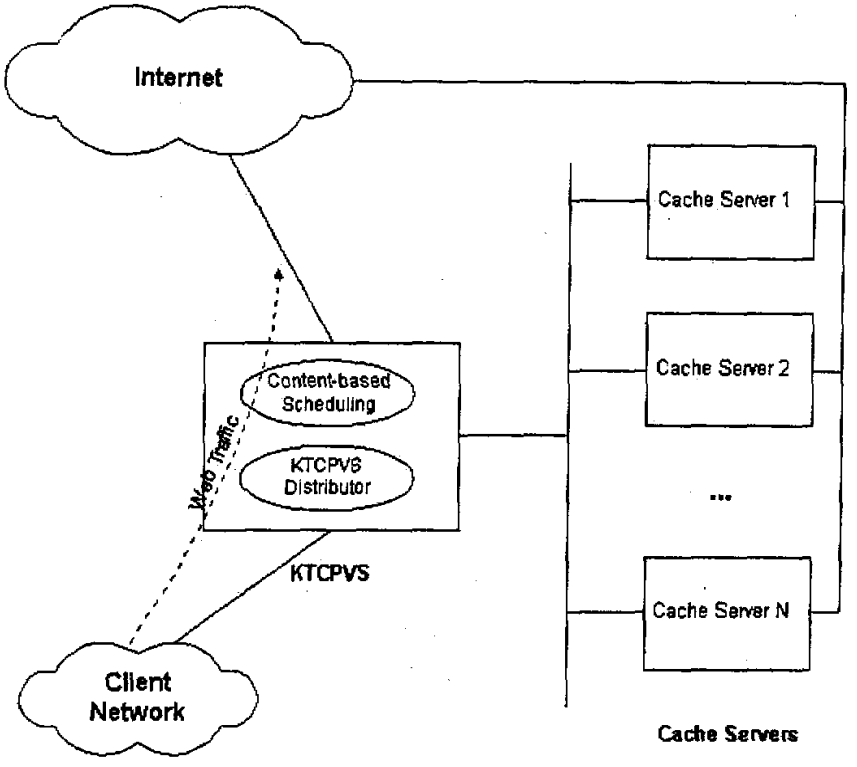

图 7.6：基于 KTCPVS 交换机的 Cache 集群

在现在 WEB 服务中，越来越多的对象是动态生成的，如通过 CGI（Common Gateway Interface）、ASP（Active Server Page）生成的，或者是带 Cookie 的对象。这些动态生成的对象都是不可缓存的对象，可缓存的对象一般有 HTML 文档、GIF、JPEG、PNG 等静态文档。不可缓存的对象访问在 WEB 访问中的比例呈增长趋势，有的比例高达 30%到 50%。如果将不可缓存的对象访问转发给 Cache 服务器，Cache 服务器判断它是不可缓存的，从源服务器取得对象，再将对象返回给客户。这会增加 Cache 服务器的响应延时。若我们能在 KTCPVS 交换机中直接到源服务器取不可缓存的对象，可以避免建立一个从 KTCPVS 交换机到 Cache 服务器连接的开销，因为 KTCPVS 交换机是在操作系统的内核中完成上面操作，它要比用户空间的 Cache 服务高效 2 倍以上。这样不可缓存的对象访问跳过 Cache 服务器，而 Cache 服务器只处理可以缓存的对象请求，可以提高命中率，缩短响应延时。

类似基于局部性的最小连接调度 LBLC/R 算法可以充分挖掘访问请求流中的局部性，将访问请求流分割到不同的 Cache 服务器，而在不同的 Cache 服务器上对象很少有重叠的。这样，可以极大地提高单个服务器的 Cache 命中率，从而当 Cache 服务器数目增加时，整个 Cache 集群系统的性能可以超过线性地增加。

若 KTCPVS 交换机发现有 Cache 服务器超载，则跳过该 Cache 服务器，从源服务器直接取得服务对象。保证 Cache 服务器在正常负载下工作，可以保证 Cache 服务器的正常响应时间。

因为以上 Cache 服务的特殊需求，我们要对 LBLC/R 算法进行相应的调整，调整后的算法如下：

```
while (true) {
  get next request r;
  if (r is for non-cacheable object) {
    fetch object from the original server directly;
    continue loop;
  }
  if (serverSet[r.target] == ∅) then {
    n = { least connection node };
    add n to serverSet[r.target];
  } else {
    n = {least connection node in serverSet[r.target]};
    if (n.conns > n.high) {
      if ( existing a node with node.conns < node.low) {
        n = { least connection node };
        add n to serverSet[r.target];
      } else {
        fetch object from the original server directly;
        continue loop;
      }
    }
    if |serverSet[r.target]| >1
      && time()-serverSet[r.target].lastMod > K then {
      m = {most connection node in serverSet[r.target]};
      remove m from serverSet[r.target];
    }
  }
  n.conns = n.conns + 1;
  fetch object from cache server n;
  n.conns = n.conns - 1;
  if serverSet[r.target] changed then
    serverSet[r.target].lastMod = time();
}
```

从前面的章节中，我们可以知道一个KTCPVS交换机只能调度十台左右的Cache服务器。所以，用一个 KTCPVS 交换机的 Cache 集群的性能还是有限的，一般每秒处理不到 3000 请求。

7.5.4 基于内容分发的大规模 Cache 集群系统

为了实现大规模、高可伸缩的基于内容分发 Cache 集群系统，我们需要用前面讲述的大规模基于内容分发的系统，引入高吞吐率的 Layer-4 交换机来调度多个 KTCPVS 分发器。图 7.7 显示了大规模基于内容分发 Cache 集群系统的体系结构，一组 KTCPVS 分发器和一组 Cache 服务器直接与 Internet 相连，一个 Layer-4 交换机是客户访问 Internet 的必经之路，Layer-4 交换机将 WEB 访问请求负载均衡地分发到各个 KTCPVS 分发器。

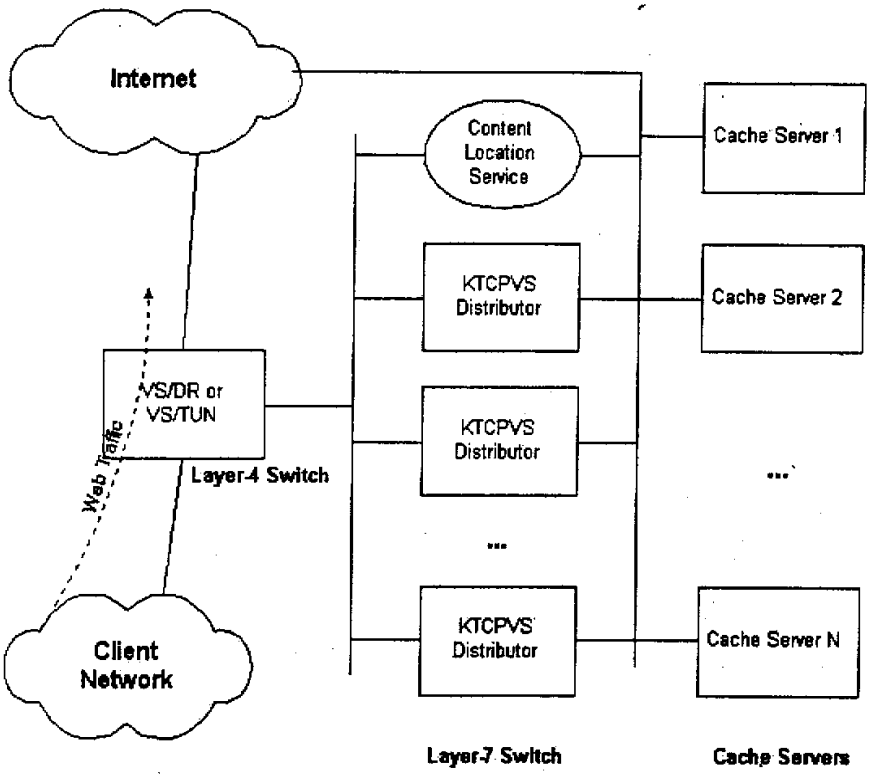

图 7.7：大规模基于内容分发的 Cache 集群系统

我们选用多个独立 KTCPVS 分发器的结构，主要考虑到 KTCPVS 分发器对不可缓存的对象跳过 Cache 服务器需要一定的开销，在这种结构下 Cache 服务器可以是异构的，即 Cache 服务器可以运行不同的操作系统，充分利用原有的服务器资源。

KTCPVS 分发器的处理流程如下：

```
while (true) {
  get next request r;
  if (r is for non-cacheable object) {
    fetch object from the original server directly;
    continue loop;
```

```
    }
    n = query Content Location Sevice with request r;
    if (n is not null)
      fetch object from server n;
  }
```

CLS 服务器上调度算法也作相应的调整，调整后的算法如下：

```
  while (true) {
    get next request r;
    if (serverSet[r.target] == ∅) then {
      n = { least loaded node };
      add n to serverSet[r.target];
    } else {
      n = {least loaded node in serverSet[r.target]};
      if (n.load > T ) then {
        if (a node with node.load < T/2) then {
          n = { least loaded node };
          add n to serverSet[r.target];
        else
          n = the original server in the request r;
        }
        if |serverSet[r.target]| >1
          && time()-serverSet[r.target].lastMod > K then {
          m = {most loaded node in serverSet[r.target]};
          remove m from serverSet[r.target];
        }
      }
      if serverSet[r.target] changed then
        serverSet[r.target].lastMod = time();
      send n back to the client;
  }
```

虽然我们没有足够的物理设备来测试该大规模基于内容分发 Cache 集群系统的实际性能，但我们可以分析到前端的 Layer-4 交换机是系统的唯一瓶颈。在 Pentium III 500MHz、128M 内存和 2 块 100M 网卡硬件配置下，VS/DR 或 VS/TUN 调度器的吞吐率为 25,000 Connections/Second。假设每个连接请求对象的大小为 10Kbytes，则该系统的吞吐率为 250Mbytes/Second，即 2Gbps。它可满足主干网 Cache 系统的需求。在更高的硬件配置下，VS/DR 或 VS/TUN 调度器会有更高的吞吐率，整个系统的吞吐率也会随之提高。

§7.6 本章小结

我们给出大规模基于内容请求分发的服务器集群的体系结构。在这个体系结构中，一个 Layer-4 交换机（VS/DR 或 VS/TUN）作为集群系统的单一入口点（Single-Entry Point），将请求分发给一组 Layer-7 交换机（KTCPVS 分发器），KTCPVS 分发器使用中心的内容位置服务 CLS 来确定处理该请求的后端服务器，再将请求分发到后端服务器。KTCPVS 分发器可以运行在独立的计算机上或者与服务程序共存在后端服务器上，面向这两种结构我们提出了运行在 CLS 中相应的基于局部性最小负载算法，它可以动态地获得各个服务器的负载状况，在负载基本均衡情况下，提高单个服务器的访问局部性。最后，我们实现了该体系结构，对内容位置服务的性能进行了优化，并对其性能进行简要的测试。

我们将该体系结构应用大规模的 Cache 集群系统中，提出面向 Cache 应用的基于局部性最小负载算法。通过基于内容请求分发来提高单个 Cache 服务器上的命中率；将不可 Cache 的对象请求跳过 Cache 服务器，而直接将请求发给源服务器，这样可以缩短处理时间同时可以提高 Cache 服务器的使用效率；算法保证 Cache 服务器在正常负载下工作，若 Cache 服务器超载，KTCPVS 交换机将跳过 Cache 服务器，而直接将请求发给源服务器，提高 Cache 服务器的效率。

第八章 结束语

随着 Internet 的使用爆炸式地增长，越来越多网站担心在访问负载不断增长时系统的可伸缩性和可靠性。新的网络应用需要更高的带宽和响应速度，如连续媒体需要很高的系统吞吐率，电子商务交易系统在任何时候很多（即使网络比较拥挤时）需要很低的响应延时。这些应用对网络基础设施和服务器端提出很大的挑战。本文把重点放在服务器端，主要研究了可伸缩网络服务的体系结构、实现技术和相关算法，并实现了一个可伸缩网络服务的框架。

§8.1 工作总结

8.1.1 可伸缩网络服务的体系结构

我们分析现在和将来网络服务的需求，提出可伸缩网络服务的体系结构，分为负载调度器、服务器池和后端存储三层结构。负载调度器采用 IP 负载均衡技术和基于内容请求分发技术。它的实现将在 Linux 操作系统进行，将一组服务器组成一个高可伸缩的、高可用的网络服务器，故称之为 Linux Virtual Server。它提供了负载平衡、可伸缩性和高可用性，可以应用于建立很多可伸缩的网络服务。

在此基础上，我们还提出地理分布的 LVS 集群系统，通过 BGP 插入路由信息的方法可以使得用户访问邻近的服务器集群，通过 IP 隧道实现服务器集群间的负载均衡，进一步提高响应速度。地理分布的 LVS 集群系统可以节约网络带宽，改善网络服务质量，有很好的抗灾害性。

8.1.2 IP 负载均衡技术

在分析网络地址转换方法（VS/NAT）的缺点和网络服务非对称性的基础上，我们提出通过 IP 隧道实现虚拟服务器的方法 VS/TUN，和通过直接路由实现虚拟服务器的方法 VS/DR，极大地提高了系统的伸缩性。

8.1.3 负载调度

我们先给出 IP 虚拟服务器在内核中几种确实可行的连接调度算法：

- 轮叫调度（Round-Robin Scheduling）
- 加权轮叫调度（Weighted Round-Robin Scheduling）

- 最小连接调度（Least-Connection Scheduling）
- 加权最小连接调度（Weighted Least-Connection Scheduling）
- 全球服务器调度（Global Server Scheduling）

针对请求的服务时间变化很大容易使得服务器运行出现倾斜，我们提出一个动态反馈负载均衡算法，结合内核中的加权调度算法，根据动态反馈回来的负载信息来调整服务器的权值，来调整服务器间处理请求数的比例，从而避免服务器间的负载不平衡。性能模拟结果也说明动态反馈负载算法可以较好地避免服务器的倾斜，提高系统的资源使用效率，从而提高整个系统的吞吐率。

8.1.4 IP 虚拟服务器的实现和性能测试

我们在 Linux 内核中高效地实现了 IP 虚拟服务器软件，支持三种 IP 负载均衡技术和五种连接调度算法。它还有特点：

- 高效的 Hash 函数
- 高效的垃圾回收机制
- 虚拟服务的数目没有限制，每个虚拟服务有自己的服务器集。
- 支持持久的虚拟服务
- 正确的 ICMP 处理
- 拥有本地结点功能
- 提供系统使用的统计数据
- 针对大规模 DoS 攻击的三种防卫策略

IP 虚拟服务器软件系统具有良好的伸缩性，支持几百万个并发连接。无需对客户机和服务器作任何修改，可适用任何 Internet 站点。通过 IP 虚拟服务器软件和集群管理工具可以将一组服务器组成一个高性能、高可用的网络服务。性能测试数据说明在一般的硬件配置下 IP 虚拟服务器中 VS/DR 和 VS/TUN 方法的最大吞吐率可达 25,000 Connections/Second。该软件已在世界各地得到很好的应用。

8.1.5 内核中的基于内容请求分发

在对已有基于内容请求分发（即 Layer-7 交换）的方法进行分析基础上，提出在操作系统内核中利用内核线程高效地实现 Layer-7 交换的解决方法，可以避免了多次内核与用户空间的切换和内存复制开销，从而提高系统的吞吐率。

提出基于局部性的最小连接调度算法 LBLC，算法假设后端服务器都是相同的，算法的目标是在后端服务器的负载基本平衡情况下，尽可能将相同的请求分到同一台服务器，以提高后端服务器的访问局部性，从而提高后端服务器的主存 Cache 命中率。在不同类型服务器的调度中，我们给出基于内容的调度算法，若同一类型的请求有多个服务器可以选择时，将请求负载均衡地调度到这些服务器上。

详细阐述了内核 Layer-7 交换机 KTCPVS 的实现。分析了多进程、多线程和单

进程事件驱动的服务器体系结构的优缺点，提出多线程和单进程事件驱动的混合结构——多线程事件驱动结构，并采用该结构在内核中实现了 KTCPVS。KTCPVS 支持持久 HTTP 连接处理，并通过检测结点或服务进程故障和正确地重置系统达到高可用性也。无需对客户机和服务器作任何修改，可适用任何 Internet 站点。最后，对用户空间的 TCP Gateway 和 KTCPVS 交换机的性能进行测试和比较，实验数据说明在内核中实现基于内容请求分发的方法具有良好的伸缩性。

8.1.6 大规模基于内容请求分发的系统

设计和实现了大规模基于内容请求分发系统。一个 VS/DR 或 VS/TUN 调度器为集群系统的单一入口点，将请求分发给一组 KTCPVS 分发器，它们使用中心的内容位置服务来确定处理该请求的后端服务器，KTCPVS 分发器再将请求分发到后端服务器。KTCPVS 分发器可以运行在独立的机器上，也可以运行在后端服务器上。在内容位置服务的请求分发策略上，提出了基于局部性最小负载算法，对其性能进行了优化。

我们将该体系结构应用大规模的 Cache 集群系统中，提出面向 Cache 应用的基于局部性最小负载算法。通过基于内容请求分发来提高单个 Cache 服务器上的命中率；将不可 Cache 的对象请求跳过 Cache 服务器，而直接将请求发给源服务器，这样可以缩短处理时间同时可以提高 Cache 服务器的使用效率；算法保证 Cache 服务器在正常负载下工作，若 Cache 服务器超载，KTCPVS 交换机将跳过 Cache 服务器，而直接将请求发给源服务器，提高 Cache 服务器的效率。

8.1.7 Linux Virtual Server 框架

在以上工作基础上，我们提供一个 Linux Virtual Server 框架，如图 1.2 所示。在 LVS 框架中，提供了含有三种 IP 负载均衡技术的 IP 虚拟服务器软件、基于内容请求分发的内核 Layer-7 交换机 KTCPVS 和集群管理软件。IP 虚拟服务器需要后端服务器是相同的，有很高的吞吐率，如在一个 Pentium III 500MHz、128M 内存和 2 块 100M 网卡的机器上，VS/DR 或 VS/TUN 的最大吞吐率可高达 25,000 Connections/Second。KTCPVS 可以进行基于内容的调度，后端服务器可以提供不同的网络服务，LBLC 算法可以提高后端服务器的 Cache 使用效率，可以使得整体性能超过线性增长，但是 KTCPVS 的吞吐率有限，接近于 VS/NAT 的吞吐率。可以将一个 VS/DR 或 VS/TUN 调度器、一组 KTCPVS 分发器和内容位置服务组成一个大规模基于内容请求分发系统。

利用 LVS 框架，实现高可伸缩的、高可用的网络服务变得相对容易些。再集成其他网络服务软件、分布式文件系统等，可实现大规模的 Web、Cache、Mail 和 Media 等服务。在此基础上，可以开发支持庞大用户数的、高可伸缩的、高可用的电子商务应用。在过去的一年多时间中，LVS 框架已经在很多国外的大型站点使用，

一些大公司（如 Red Hat）在此基础上开发出服务器集群软件，也有一些大公司（如 VA Linux）提供由他们服务器构成的 LVS 集群和相应的技术支持服务。

§8.2 研究展望

集群技术是一个计算机领域研究的关键技术[113, 114, 115, 116, 117, 118]，它可适用于高性能计算和事务处理。本文只涉及了如何利用服务器集群来建设可伸缩网络服务，主要研究可伸缩网络服务的体系结构、实现技术和相关算法。虽然提供了一个可伸缩网络服务的框架，该框架已经在一些大型站点得到应用，但本文只做了一些基础性的工作，可伸缩网络服务中仍有很多问题值得深入研究。今后的研究工作可以在以下几方面进行：

- 可伸缩网络服务中的服务质量（Quality of Service）

  Internet 的流量有很多类型，如离散媒体（文本和图像）和连续媒体（音频和视频）、实时和非实时、大文件和小文件、静态页面和动态页面、安全事务和非安全事务、付费和非付费服务等。不同的类型需要不同的服务质量，无论在网络层还是服务器层。目前，大部分服务质量的研究是在网络层，有一些研究涉及在单服务器上的区分服务[119, 120, 121]，很少有研究关于服务器集群的区分服务。

- 流量特征研究和可伸缩网络服务的容量规划

  现在有一些研究关于 Internet 流量的特征，都说明 Internet 流量的突发性特点，如 Zipf 分布、自相似（Self-Similar）[73, 74]等。但是，很难预测流量的增长模式，也没有很好理解访问增长和数据大小增长的关系，没有涉及动态页面处理性能需要的复杂性。缺乏有效的工具分析已有的访问记录，来规划可伸缩网络服务现在和将来的容量。

- 分布式锁管理器

  在可伸缩网络服务中，不同服务器上的应用会并发访问一个共享资源，这里需要一种机制来消解访问的冲突，保证共享资源的一致性。分布式锁管理器（Distributed Lock Manager）可以提供这种机制，在开发应用时，可利用 DLM 保证应用并发访问的一致性。在可伸缩网络服务中，若一个服务器结点失效，服务器不再被调度可以将故障屏蔽掉；若 DLM 失效，可能导致整个网络服务失效。所以，DLM 必须有很高的容错性和伸缩性。

  我们曾经基于 CORBA 实现了一个非常简单的 DLM，但在效率和容错上都不理想。需要深入研究高效的、容错的 DLM 的体系结构和相关的实现技术。

- 分布式数据结构（Distributed Data Structures ）

  分布式数据结构[122, 123]是专门为集群中服务器提供共享存储的数据结构，它可以比分布式文件系统和数据库系统具有更好的功能和更高的效率，如可伸缩性和容错处理[124, 125]。Gribble 刚完成的博士论文主要研究分布式数

据结构的实现技术并为 Ninja 平台完成一个基于 Java 语言的分布式 Hash 表，但是分布式数据结构的可伸缩性和容错处理还值得研究，很多数据结构有待实现，如分布式树结构、分布式 B+树结构和分布式日志结构等，用 C 或者 C++实现可以大大提高性能。

- 高速网络下的可伸缩网络服务

  在可伸缩网络服务中，各个服务器的使用效率要比相互独立服务器的效率高，所以利用服务器集群可以减少相互独立服务器的数目。但是，服务器集群的伸缩性一直是值得研究的课题，在许多结点情况下（如 500 个和上千个），结点间的通讯网络有可能首先成为瓶颈。所以，在可伸缩系统区网络（Scalable System Network，如 Infiniband[126, 127]）和轻量级通讯协议（如 Virtual Interface Architecture, VIA[128]）下，可进一步研究可伸缩网络服务的体系结构和实现技术。

此外，很多世界各地 Linux 开发者和我正在做 Linux Cluster Infrastructure 项目，目标是在今后的两、三年内使得 Linux 成为一个真正的集群操作系统。Linux Cluster Infrastructure 主要为集群系统（无论是科学计算集群还是网络服务集群）提出所需的基础设施，如内核中的可靠组通讯（Reliable Group Communication）[129, 130]、成员管理（Membership Management）[131]、选举系统（Quorum Systems）[132, 133]、负载均衡（Load Balancing）[134, 135]、可伸缩的集群文件系统（Scalable Cluster Filesystem）[46, 52]、透明的进程迁移（Transparent Process Migration）[136, 137]和一组集群调用的 API 定义等。

攻博期间发表的部分学术论文

[1] 章文嵩、吴婷婷、金士尧、吴泉源，“一个虚拟 Internet 服务器的设计与实现”，软件学报，2000，11(1):122~125.

[2] Wensong Zhang, Shiyao Jin and Quanyuan Wu, “LinuxDirector: A Connection Director for Scalable Internet Services”, Journal of Computer Science and Technology，2000，15(6):560~571.

[3] 章文嵩、金士尧、吴泉源，“KTCPVS：一个内核的 Layer-7 交换机”，计算机科学与工程，（已录用）

[4] Wensong Zhang, Shiyao Jin and Quanyuan Wu, “Linux Virtual Server: Server Clustering for Scalable Network Services”, Proceeding of World Congress Conference 2000, Beijing, August 21~25, 2000.

[5] Wensong Zhang, Shiyao Jin and Quanyuan Wu, “Creating Linux Virtual Servers”, Proceeding of the 5th Linux Expo Conference, Raleigh, NC, USA, May, 1999.

[6] Wensong Zhang, Shiyao Jin and Quanyuan Wu, “Linux Virtual Servers for Scalable Network Services”, Proceeding of the 2nd Ottawa Linux Symposium, Ottawa, Canada, July, 2000.

[7] Wensong Zhang, Shiyao Jin and Quanyuan Wu, “IP Load Balancing Technologies in LinuxDirector”, Proceeding of the 3rd Workshop on Advanced Paralled Processing Technologies, Changsha, Hunan, China, October, 1999.

[8] Wensong Zhang, Shiyao Jin and Quanyuan Wu, “Scaling Internet Services by LinuxDirector”, Proceeding of IEEE High-Performance Computing / Asia 2000, Beijing, China, May, 2000.

[9] 章文嵩、章文卓、吴泉源，“LVS: 可伸缩网络服务的 Linux 集群”，中国计算机世界报， 2000.08.

致 谢

在本文完成之际，谨向给予我指导、关心和支持的各位老师、领导、同学和亲友们致以衷心的感谢！

首先衷心感谢我的导师金士尧教授！金老师在我的课题研究期间，从选题、研究、设计直到撰写论文的各个阶段，给予我悉心指导和帮助，也给予自由空间去创造和发挥。金老师以其高深的学术造诣、丰富的工程经验、严谨的治学态度、旺盛的工作热情以及对祖国计算机事业的奉献精神都时时刻刻地激励着我。三年多时光匆匆而过，但金老师对我的言传身教和以及各方面能力的培养将使我终身受益！

我也要衷心感谢吴泉源教授！感谢吴老师在我的学习、工作和生活中所给予的无私关心、悉心指导和严格要求。吴老师有着渊博的知识、严谨的学风和敏锐的洞察力，每次探讨他总能很快地找到问题的实质，提出很多宝贵的想法。吴老师经常出一些智力题挑战我的头脑，教导我做研究的方法，帮助我完成从一个求学的学生到一个愿意去发现和创新的研究人员的转变。

感谢计算机学院的专家对本课题的关心，感谢胡守仁教授、杨晓东教授、王怀民教授对本课题的指导。感谢并行与分布处理国家重点实验室的老师们为本课题研究提供良好的实验环境。感谢胡华平博士，在胡师兄从事博士后课题研究期间，他给我的课题研究提供了很多建议。感谢肖侬博士，进行很多有益的探讨。

感谢徐志伟教授邀请我在中科院计算所做访问工作，他提供的良好环境使我工作效率很高，也过得很愉快。感谢南加州大学的金海博士邀请我为 CISC'2001 会议程序委员会成员。

感谢 Joseph Mack 博士编写 LVS-HOWTO 文档和替我在美国的 LinuxExpo'99 会议作报告。感谢 Ottawa Linux Symposium 主席 Andrew J. Hutton 的邀请和 Red Hat 公司的帮助，使我能成行到加拿大作服务器集群的学术报告。感谢 Stephen Tweedie 博士和 Peter Braam 博士，进行很多 Linux 集群的探讨。感谢 Julian Anastasov 和 Simen Horman 对 IPVS 代码的 Bug fix 和改进意见。感谢 IBM Waston 研究中心的 German Goldszmidt 博士提供的研究资料。

感谢同门师兄弟，与他们的讨论经常能够碰撞出创新的火花。肖晓强博士的 MPP 系统互联网、党岗博士生的分布多媒体仿真环境、李宏亮博士生的金融仿真环境、凌云翔博士、刘晓建和王召福博士生的分布仿真系统均使我得到启发。感谢曹林奇、王俊伟、方明、雷刚、王晓川、刘华峰等硕士生的帮助。尤其感谢肖晓强对本文所提的宝贵意见。

感谢博士生队的所有同学，万俊伟、陈渝、姚丹霖等博士，荔建琦、何连跃、王克非、周健、彭伟、陈虎等博士生，与他们的讨论不断地开拓思路。感谢同寝室的杜贵然、王晓东博士生，与他们的生活一直很愉快。感谢黄光奇博士生，进行金融问题的探讨。感谢邓鸥博士生，他的幽默诙谐带来不少笑语。

感谢光华奖学金的精神鼓励和物质支持。

感谢学员大队领导、博士队队领导、学院训练部、校研究生院等有关部门在我攻博期间给予的帮助和支持。

感谢女友吴婷婷小姐，她的相伴使我的生活平添很多乐趣，她的鼓励和帮助一直是我前进的动力。

最后，我要感谢父母的养育之恩。在我二十多年的寒窗生涯中，是他们谆谆教诲和鼓励促使我不断开拓进取，是他们的言传身教，使我知道如何做一个诚实而有用的人。感谢家人对我的关心和爱护，让我时刻牢记身后那一双双期待的目光。在我顺利时他们鼓励我再接再厉，在我失意时他们鼓励我振奋。家人为我付出的一切是我终身难以报答的，唯寄希望于此文能够稍稍有所回报！

参考文献

[1] T. Berners-Lee, R. Cailliau, A. Loutonen, H.F. Nielsen and A. Secret, The world Wide Web, Communications of the ACM, 37(8):76-82, August 1994. http://info.cern.ch/hypertext/WWW/TheProject.html

[2] Information Navigators, Internet Growth Charts, http://navigators.com/stats.html

[3] Srinivasan Seetharaman. IP over DWDM. http://www.cis.ohio-state.edu-/~jain/cis788-99/ip_dwdm/

[4] Lucent Technologies. Web ProForum tutorial: DWDM. October 1999, http://www.webproforum.com/acrobat/dwdm.pdf

[5] Lucent Technologies. Lucent Technologies announces record-breaking 320-channel optical networking system. April 2000, http://www.lucent.com/press-/0400/000417.nsb.html.

[6] James C. Hu, Sumedh Mungee, Douglas C. Schmidt, "Principles for Developing and Measuring High-performance Web Servers over ATM", WUCS Technical Report, September, 1997.

[7] James C. Hu, "Creating a Framework for Developing High-performance Web Server over ATM", Feburay, 1997.

[8] Yahoo! Inc., The Yahoo! Directory and Web Services, http://www.yahoo.com/

[9] Dell Inc. http://www.dell.com/

[10] Eric Dean Katz, Michelle Butler, and Robert McGrath, "A Scalable HTTP Server: The NCSA Prototype", Computer Networks and ISDN Systems, pp155-163, 1994.

[11] Thomas T. Kwan, Robert E. McGrath, and Daniel A. Reed, "NCSA's World Wide Web Server: Design and Performance", IEEE Computer, pp68-74, November 1995.

[12] T. Brisco, "DNS Support for Load Balancing", RFC 1794, http://www.internic.net/ds/

[13] C. Yoshikawa, B. Chun, P. Eastham, A. Vahdat, T. Anderson, and D. Culler. Using Smart Clients to Build Scalable Services. In Proceedings of the 1997 USENIX Technical Conference, January 1997.

[14] Zeus Technology, Inc. Zeus Load Balancer v1.1 User Guide. http://www.zeus.com/

[15] Edward Walker, "pWEB - A Parallel Web Server Harness", http://www.ihpc.nus.edu.sg/STAFF/edward/pweb.html, April, 1997.

[16] Ralf S.Engelschall. Load Balancing Your Web Site: Practical Approaches for Distributing HTTP Traffic. Web Techniques Magazine, Volume 3, Issue 5, http://www.webtechniques.com, May 1998.

[17] Daniel Andresen, Tao Yang, Oscar H. Ibarra. Towards a Scalable Distributed WWW Server on Workstation Clusters. In Proceedings of 10th IEEE International Symposium Of Parallel Processing (IPPS'96), pp.850-856, http://www.cs.ucsb.edu-/Research/rapid_sweb/SWEB.html, April 1996.

[18] Daniel Andresen, Tao Yang, Oscar H. Ibarra, Omer Egecioglu. Adaptive Partitioning and Scheduling for Enhancing WWW Application Performance. Technical Report, University of California Santa Barbara, 1997. http://www.cs.ucsb.edu.

[19] Daniel Andresen, Tao Yang, David Watson, and Athanassios Poulakidas. Dynamic Processor Scheduling with Client Resources for Fast Multi-resolution WWW Image Browsing. Technical Report, University of California Santa Barbara, 1997. http://www.cs.ucsb.edu.

[20] D. Andresen, T. Yang, O. Egecioglu, O. H. Ibarra, and T. R. Smith. Scalability Issues for High Performance Digital Libraries on the World Wide Web Proceedings of ADL `96, Forum on Research and Technology Advances in Digital Libraries, IEEE, Washington D.C., May 1996.

[21] Eric Anderson, Dave Patterson, and Eric Brewer, "The Magicrouter: an Application of Fast Packet Interposing", http://www.cs.berkeley.edu/~eanders-/magicrouter/, May, 1996.

[22] Cisco Systems Inc. Cisco LocalDirector. http://www.cisco.com/warp/public-/751/lodir/index.html.

[23] Alteon Networks Inc. Alteon ACEDirector. http://www.alteon.com.

[24] F5 Networks Inc. F5 Big/IP. http://www.f5networks.com.

[25] D. Dias, W. Kish, R. Mukherjee, and R. Tewari. A Scalable and Highly Available Server. In Proceeding of COMPCON 1996, IEEE-CS Press, Santa Clara, CA, USA, Febuary 1996, pp. 85-92.

[26] IBM Corporation. IBM interactive network dispatcher. http://www.ics.raleigh.ibm.com/ics/isslearn.html.

[27] Guerney D.H. Hunt, German S. Goldszmidt, Richard P. King, and Rajat Mukherjee. Network Dispatcher: a connection router for scalable Internet services. In Proceedings of the 7th International WWW Conference, Brisbane, Australia, April 1998.

[28] Om P. Damani, P. Emerald Chung, Yennun Huang, "ONE-IP: Techniques for Hosting a Service on a Cluster of Machines", http://www.cs.utexas.edu-/users/damani/, August 1997.

[29] Microsoft Corporation. Microsoft Windows NT Load Balancing Service. http://www.micorsoft.com/ntserver/NTServerEnterprise/exec/feature/WLBS/.

[30] A. Dahlin, M. Froberg, J. Walerud and P. Winroth, "EDDIE: A Robust and Scalable Internet Server", http://www.eddieware.org/, May 1998.

[31] A. Fox, S.D. Gribble, Y. Chawathe, E. A. Brewer and P. Gauthier, Cluster-Based Scalable Network Services, In Proceedings of the 16th ACM Symposium on Operating Systems Principles, St.-Malo, France, October 1997.

[32] Steven D. Gribble, Matt Welsh, Rob von Behren, Eric A. Brewer, David Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R.H. Katz, Z.M. Mao, S. Ross, B. Zhao, The Ninja Architecture for Robust Internet-Scale Systems and Services, http://ninja.cs.berkeley.edu/

[33] Steven D. Gribble, A Design Framework and a Scalable Storage Platform to Simplify Internet Service Construction. Ph.D. thesis, U.C. Berkeley, September 2000, http://www.cs.berkeley.edu/~gribble/papers/gribble_thesis.ps.gz

[34] Werner Vogels, Dan Dumitriu, Ken Birman, Rod Gamache, Mike Massa, Rob Short, John Vert, Joe Barrera, Jim Gray, "The design and Architecture of the Microsoft Cluster Service – A Practical Approach to High-Availability and Scalability", The Proceeding of Fault-Tolerant Computer System'98, Munich, Germany, June 23-25, 1998.

[35] Jim Gray, "Jim Gray's NT Cluster's Research Agenda", http://www.research.microsoft.com/~Gray, October 1995.

[36] Jim Gray, "Super-Servers: Commodity Computer Clusters Pose a Software Challenge", http://www.research.microsoft.com/~Gray, 1995.

[37] White Paper, "Clustering Support for Microsoft SQL Server: *High Availability for Tomorrow's Mission Critical Applications*", http://www.research.microsoft.com/, 1997.

[38] President's Information Technology Advisory Committee. Information Technology Research: Investing in Our Future. February 1999. http://www.research.microsoft.com/~Gray

[39] Gordon Bell and Jim Gray. The Revolution Yet to Happen. Microsoft Research Techical Report MSR-TR-98-45, March 1997. http://www.research.microsoft.-com/~Gray

[40] F. J. Corbato and V. A. Vyssotsky. Introduction and Overview of the Multics System. AFIPS Conference Proceedings, 27, 185?96, (1965 Fall Joint Computer Conference), 1965. http://www.lilli.com/fjcc1.html

[41] J.H. Howard. An Overview of the Andrew File System. In Proceedings of the USENIX Winter Technical Conference, Dallas, TX, USA, February 1998.

[42] IBM Corporation. IBM Andrew File System. http://www.ibm.com.

[43] Kenneth W. Preslan, Andrew P. Barry, Jonathan E. Brassow, Grant M. Erickson, Erling Nygaard, Christopher J. Sabol, Steven R. Soltis, David C. Teigland, and

Matthew T. O'Keefe. A 64-bit, Shared Disk File System for Linux. In Proceeding of 16th IEEE Mass Storage Systems Symposium, San Diego, CA, USA. March 15-18, 1999.

[44] Kenneth W. Preslan, Andrew P. Barry, Jonathan E. Brassow, Russell Cattelan, Adam Manthei, Erling Nygaard, Seth Van Oort, David Teigland, Mike Tilstra, and Matthew T. O'Keefe. Implementing Journaling in a Linux Shared Disk File System. To appear in Proceedings of 17th IEEE Symposium on Mass Storage Systems.

[45] Steven R. Soltis. The Design and Implementation of a Distributed File System based on Shared Network Storage. Ph.D. thesis, University of Minnesota, August 1997.

[46] Global File System Website. http://www.globalfilesystem.org.

[47] M. Satyanarayanan. Coda: A Highly Available File System for a Distributed Workstation Environment. In Proceedings of the 2nd IEEE Workshop on Workstation Operating Systems. Pacific Grove, CA, USA, September 1989.

[48] M. Satyanarayanan. Scalable, Secure, and Highly Available Distributed File Access. IEEE Computer, Vol. 23, No. 5, May 1990.

[49] Peter J. Braam. The Coda Distributed File System. Linux Journal, #50, June 1998.

[50] L.B. Mummert, M. Satyanarayanan. Long Term Distributed File Reference Tracing: Implementation and Experience, Vol. 26, No. 6, June 1996.

[51] Coda File System Website. http://www.coda.cs.cmu.edu.

[52] Peter J. Braam, Michael Callahan, Phil Schwan. The InterMezzo Filesystem. In Proceeding of O'Reilly Perl Conference, 1999. http://www.inter-mezzo.org/docu/perlintermezzo.pdf.

[53] Myricom Inc. Myrinet: A Gigabit Per Second Local Area Network, IEEE Micro, Vol 15, No. 1 February 1995, pp. 29-36.

[54] Ziatech Inc. CompactNET: Multiprocessing Open Source Forum. http://www.compact.com.

[55] Bill Devlin, Jim Gray, Bill Laing, George Spix. Scalability Terminology: Farms, Clone, Partitions and Packs: RACS and RAPS. Microsoft Research Technical Report MS-TR-99-85, http://www.research.microsoft.com/~Gray/

[56] Alan Robertson, et al. Linux High Availability Project. http://www.linux-ha.org.

[57] Simon Horman, "Creating Redundant Linux Servers", The 4th Annual Linux Expo, May, 1998, http://linux.zipworld.com.au/fake/.

[58] Ierry Glomph Black. LVS testimonials from Real Networks. March 2000. http://marc.theaimsgroup.com/?l=linux-virtual-server&m=95385809030794&w=2

[59] Michael Sparks. Load Balancing the UK National JANET Web Cache Service Using Linux Virtual Servers. November 1999. http://wwwcache.ja.net/JanetService-/PilotService.html.

[60] Jonathan B. Postel. RFC 821: Simple Mail Transfer Protocol, August 1982. http://www.cis.ohio-state.edu/htbin/rfc/rfc821.html.

[61] John G. Myers and Marshall T. Rose. RFC 1939: Post office protocol version 3, May 1996. http://www.cis.ohio-state.edu/htbin/rfc/rfc1939.html.

[62] Marc Crispin. RFC2060: Internet message access protocol version 4 rev 1, December 1996. http://www.cis.ohio-state.edu/htbin/rfc/rfc2060.html.

[63] Y. Rekhter, T. Li. RFC1771: A Border Gateway Protocol 4 (BGP-4), March 1995. http://www.cis.ohio-state.edu/htbin/rfc/rfc1771.html.

[64] J. B. Postel, DARPA Internet Protocol Specification, RFC 791, September 1981.

[65] J. B. Postel, Transmission Control Protocol, RFC 793, September 1981.

[66] Y. Rekhter and T. Li, An Architecture for IP Address Allocation with CIDR, RFC 1518, September 1993

[67] W. R. Stevens. TCP/IP Illustrated Volume I. Addison-Wesley, Reading, MA, USA, 1994.

[68] W. R. Stevens. UNIX Network Programming. Prentice-Hall, Englewood Cliffs, NJ, USA, 1990.

[69] G. R. Wright and W. R. Stevens. TCP/IP Illustrated Volume II. Addison-Wesley, Reading, MA, USA, 1995.

[70] W. R. Stevens. TCP/IP Illustrated Volume III. Addison-Wesley, Reading, MA, USA, 1996.

[71] A. Rijsinghani, Ed. RFC 1624: Computation of the Internet Checksum via Incremental Update, May 1994. http://www.cis.ohio-state.edu/htbin/rfc-/rfc1624html.

[72] World Wide Web Consortium. HyperText Transfer Protocol(HTTP): a protocol for networked information, http://www.w3.org/hypertext/WWW/Protocols, June, 1995.

[73] William Stalling, Viewpoint: Self-similarity upsets data traffic assumptions, IEEE Spectrum, January 1997.

[74] Kihong Park, Gitae Kim, Mark Crovella, "On the Effect of Traffic Self-similarity on Network Performance", In Proceedings of the 1997 SPIE International Conference on Performance and Control of Network Systems, 1997.

[75] Nicolas D. Georganas, Self-Similar ("Fractal") Traffic in ATM Networks, In Proceedings of the 2nd International Workshop on Advanced Teleservices and High-Speed Communications Architectures (IWACA'94), pages 1-7, Heidelberg, Germany, September 1994.

[76] Mark Crovella and Azer Besavros, Explaining World Wide Web Traffic Self-Similarity. Technical report, Boston University, October 1995, TR-95-015.

[77] Bruce A. Mah. An Empirical Model of HTTP Network Traffic. In Proceedings of INFOCOM 97, Kobe, Japan, April 1997.

[78] Michael Beck (ed.), Harald Bohme, Mirko Dziadzka, Ulrich Kunitz, Robert Magnus. Linux Kernel Internal. Addison-Wesley, ISBN 0201331438, May 1998.

[79] Alessandro Rubini, Andy Oram (ed.). Linux Device Drivers. O'Reilly & Associates, ISBN 1565922921, February 1998.

[80] Scott Maxwell. Linux Core Kernel Commentary. The Coriolis Group, ISBN 1576104699, October 1999.

[81] Chuck Lever. Linux Kernel Hash Table Behavior: analysis and improvements. May 1999. http://www.citi.umich.edu/projects/linux-scalability/reports/hash.html.

[82] J. N. Gray, R. A. Lorie and G. F. Putzolu, Granularity of Locks in a Large Shared Database, In Proceedings of the Conference on Very Large Data Bases, Framingham, MA, USA, September 1995.

[83] A. Bhushan, B. Braden, W. Crowther, E. Harslem, J. Heafner, A. McKenzie, J. Melvin, B. Sundberg, D. Watson and J. White, The File Transfer Protocol, RFC 172, June 1971.

[84] Kipp E.B. Hickman. The SSL Protocol. Internet Draft, February 1995. http://home.netscape.com/eng/security/SSL_2.html.

[85] Netscape Communications Corp. Persistent Client-State Http Cookies: Preliminary Specification. http://home.netscape.com/newsref/std/ cookie_spec.html

[86] Computer Emergencey Response Team, Advisory CA-2000-01 Denial-of-Service Developments, http://www.cert.org/advisories/CA-2000-01.html, January 2000.

[87] Computer Emergency Response Team (CERT), TCP SYN Flooding and IP Spoofing, CERT Advisory CA-96.21, http://www.cert.org/ftp/cert_advisories-/CA-96.21.tcp_syn_flooding, September 1996, Last updated: August 1998.

[88] C. L. Schuba, I. Krsul, M. Kuhn, E. Spafford, A. Sundaram, and D. Zamboni. Analysis of denial of service attacks on TCP. In Proceedings of the 1997 IEEE Symposium on Security and Privacy , Los Alamitos, CA, USA, May 1997.

[89] Resonate Inc., "Resonate Products – Central Dispatch", http://www.resonate.com/

[90] Vivek S. Pai, Mohit Aron, Gaurav Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, Erich Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers. In Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, ACM, San Jose, October 1998.

[91] ArrowPoint Inc. The content smart internet. http://www.arrowpoint.com/solutions-/whitepapers/CSI.asp

[92] Apache Software Foundation. The Apache Project. http://www.apache.org/, 1996-now.

[93] National Laboratory for Applied Network Research, The Squid Internet Object Cache, http://squid.nlanr.net.

[94] Virgilio Almeida, Azer Bestavros, Mark Crovella and Adriana de Oliveira. Characterizing refernce locality in the WWW. In Proceedings of PDIS'96: The IEEE Conference on Parallel and Distributed Information Systems, Miami Beach, Florida, December 1996

[95] The Common Gateway Interface, http://hoohoo.ncsa.uiuc.edu/cgi/

[96] K. A. L. Coar and D. R. T. Robinson, The WWW Common Gateway Interface Version 1.1, Internet Draft, June 1999, http://web.golux.com/coar/cgi/draft-coar-v11-03-clean.html.

[97] D. M. Ritchie and K. Thompson, The Unix time-sharing system, Communications of the ACM, 17(7):365-375, July 1974.

[98] Jeffrey C. Mogul, Operating Systems Support for Busy Internet Servers, In Proceedings HotOS-V, Orcas Island, Washington, May 1995.

[99] S. E. Schechte and J. Sutaria. A Study of the Effects of Context Switching and Caching on HTTP Server Performance. http://www.eecs.harvard.edu/~stuart-/Tarantula/FirstPaper.html.

[100] Erich M. Nahum, Networking Support for High-Performance Servers, Ph.D. thesis, University of Massachusetts Amherst, February 1997.

[101] Gaurav Banga, Operating System Support for Server Applications, Ph.D. thesis, Rice University, January 1999. http://www.cs.rice.edu.

[102] Zeus Technology. Zeus Web Server. http://www.zeus.com.

[103] Larry Doolittle and Jon Nelson. Boa Web Server. http://www.boa.org.

[104] G. Banga and J.C. Mogul, Scalable kernel performance for Internet servers under realistic loads, In Proceedings of the USENIX 1998 Annual Technical Conference, New Orleans, LA, USA, June 1998.

[105] G. Banga, J. C. Mogul and P. Druschel, A scalable and explicit event delivery mechanism for UNIX, Technical Report CS, Rice University, November 1998, http://www.cs.rice.edu.

[106] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2068. January 1997. http://www.w3.org/Protocols-/rfc2068/rfc2068.

[107] David Mosberger, Tai Jin. Httperf – A Tool for Measuring Web Server Performance. http://www.hpl.hp.com/personal/Tai_Jin/.

[108] R. Lee and G. Tomlinson, "Workload Requirements for a Very High-Capacity Proxy Cache Design", Proceeding of the 4th International Web Caching Workshop (NLANR/CAIDA), 1999.

[109] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz and K. J. Worrell, A Hierarchical Internet Object Cache, In Proceedings of the USENIX 1996 Annual Technical Conference, San Diego, CA, USA, January 1996.

[110] Inktomi Corporation, Traffic Server, http://www.inktomi.com/products/traffic/

[111] SingNet (Singapore ISP). Heretical Caching Effort for SingNet Customers. http://www.singnet.com.sg/cache/proxy

[112] N. Smith. The UK National Web Cache - The State of the Art. In Proceedings of the 5th World Wide Web, Paris, May 1996. http://www5conf.inria.fr/fich_html-/papers/P45/Overview.html

[113] Gregory F. Pfister, In Search of Clusters, Second Edition, Prentice Hall PTR, ISBN 0-13-899709-8, New Jersey, 1998.

[114] Kai Hwang, Hai Jin, Edward Chow, Cho-Li Wang and Zhiwei Xu, Designing SSI Clusters with Hierarchical Checkpointing and Single I/O Space, IEEE Concurrency, January-March, 1999, pp. 60-69.

[115] Kai Hwang and Zhiwei Xu, Scalable Parallel Computing: Technology, Architecture, Programming, WCB/McGraw-Hill, New York, 1998.

[116] Ishfaq Ahmad, Cluster computing: A glance at recent events, IEEE Concurrency, January-March, 2000, pp. 67-69.

[117] IEEE Task Force on Cluster Computing, http://www-unix.mcs.anl.gov/~buyya/tfcc/

[118] M. Baker and R. Buyya. Cluster Computing: The Commodity Supercomputing. Software-Practice and Experience, Febuary 1999.

[119] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz. Retrofitting Quality of Service into a Time-Sharing Operating System. In Proceedings of the USENIX 1999 Annual Technical Conference, Monterey, CA, USA, June 1999.

[120] J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz. The Eclipse Operating System: Providing Quality of Service via Reservation Domains. In Proceedings of the USENIX 1998 Annual Technical Conference, Berkeley, CA, USA, June 1998.

[121] M. B. Jones, P. J. Leach, R. P. Draves, and J. S. Barrera. Modular real-time resource management in the Rialto operating systeml. In Proceedings of the 5th Wrokshop on Hot Topics in Operating Systems (HotOS-V), Orcas Island, WA, USA, May 1995.

[122] Witold Litwin and Thomas Schwarz. A High-Availability Scalable Distributed Data Structure using Reed Solomon Codes. In Proceedings of the ACM SIGMOD International Conference on the Management of Data, Dallas, TX, USA, 2000.

[123] Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein and David Culler, Scalable, Distributed Data Structures for Internet Service Construction, http://www.cs.berkeley.edu/~gribble/

[124] Jim Gray, The Transaction Concept: Virtues and Limitations, In Proceedings of VLDB, Cannes, France, September 1981.

[125] Jim Gray, T. Reuter, "Transaction Processing Concepts and Techniques", Morgan Kaufmann, 1994.

[126] Intel Corporation. InfiniBand Architecture: Switched Fabric I/O. http://developer.intel.com/design/servers/future_server_io/index.htm

[127] The InfiniBand Trade Association. InfiniBand Specification & FAQ. http://www.infinibandta.org/

[128] Virtual Interface Architecture website. http://www.viarch.org.

[129] M. G. Hayden. The Ensemble System. Ph.D. thesis, Cornell University, January 1998.

[130] Kenneth P. Birman and Robbert van Renesse. Reliable Distributed Computing with the Isis Toolkit. IEEE Computer Society Press, Loas Alamitos, CA, 1994.

[131] Tushar Deepak Chandra, Vassos Hadzilacos, Sam Toueg, and Bernadette Charron-Bost. On the impossibility of group membership. Technical Report TR95-1548, Cornell University, October 1995.

[132] Avishai Wool. Quorum Systems for Distributed Control Protocols. Ph.D. thesis, Weizmann Institute of Science, Israel, September 1996.

[133] Avishai Wool. Quorum Systems in Replicated Databases: Science or Fiction? IEEE Data Engineering, Vol.21, No.4, pp.3-11, December 1998.

[134] Wensong Zhang, Shiyao Jin and Quanyuan Wu, "Linux Virtual Server: Server Clustering for Scalable Network Services", Proceeding of World Congress Conference 2000, Beijing, August 21~25, 2000.

[135] Wensong Zhang, Shiyao Jin and Quanyuan Wu, "IP Load Balancing Technologies in LinuxDirector", Proceeding of the 3rd Workshop on Advanced Paralled Processing Technologies, Changsha, Hunan, China, October, 1999.

[136] Amnon Barak and Oren Laadan. The MOSIX Multicomputer Operating System for High Performance Cluster Computing. http://www.cs.huji.ac.il/mosix

[137] Mark P. Nuttall. Cluster Load Balancing using Process Migration. Ph.D. thesis, University of London, UK, June 1997.

可伸缩网络服务的研究与实现

作者：章文嵩
学位授予单位：国防科学技术大学
被引用次数：26次

引证文献(26条)

1. 苏命峰 三种LVS负载均衡模式及性能研究[期刊论文]-自动化与信息工程 2011(6)

2. 基于Linux虚拟服务的负载调度方法的研究[期刊论文]-硅谷 2009(19)

3. 高军 虚拟服务集群的分析与实现[期刊论文]-科学技术与工程 2007(5)

4. 夏明波. 王晓川. 金士尧. 李祖秀. 胡光强 ASAS集群的模糊控制策略[期刊论文]-计算机技术与发展 2006(12)

5. 汪黎. 罗宇. 杨明军 TCP连接迁移在Linux环境中的实现[期刊论文]-计算机工程与科学 2003(4)

6. 程伟. 卢泽新. 王宏 一种新的服务器集群系统负载均衡技术[期刊论文]-计算机工程与科学 2006(2)

7. 江浩 主动式集群中执行服务器控制机制的研究与实现[学位论文]硕士 2006

8. 基于内核Layer-7交换的Webcache 加速器的研究与实现[期刊论文]-计算机工程与科学 2005(11)

9. 王晓川. 金士尧. 夏明波 基于多线程事件驱动框架高性能应用层网关的设计与实现[期刊论文]-计算机工程与科学 2008(2)

10. 邵艳明 集群视频服务器容错与流共享策略研究[学位论文]硕士 2004

11. 戴刚. 罗宇 基于DNS的负载平衡技术的设计与实现[期刊论文]-计算机工程 2002(4)

12. 张立岩. 杨国霞. 郑琨 基于集群的Web服务负载均衡算法研究[期刊论文]-河北科技大学学报 2010(3)

13. 喻占武. 李忠民. 郑胜 基于对象存储的新型网络GIS体系结构研究[期刊论文]-武汉大学学报（信息科学版） 2008(3)

14. 刘钊 主动自调度集群集中器队列调度机制的研究与性能分析[学位论文]硕士 2006

15. 刘仲 基于对象存储结构的可伸缩集群存储系统研究[学位论文]博士 2005

16. 周松泉 一种改进的集群动态负载均衡算法[期刊论文]-计算机与现代化 2012(1)

17. 谢红薇. 谢显宇 基于内容的网络集群负载平衡算法模型[期刊论文]-计算机应用与软件 2010(1)

18. 夏明波. 王晓川. 金士尧 自调度集群的研究与实现[期刊论文]-重庆邮电学院学报（自然科学版） 2006(6)

19. 夏明波. 金士尧. 王晓川 集群服务器自调度方法的研究与实现[期刊论文]-计算机应用 2006(z2)

20. 王俊伟. 田俊. 宋振龙. 彭宇行 基于CN-RAID存储系统的集群VOD服务器的CRS调度策略[期刊论文]-计算机研究与发展 2007(z1)

21. 朱巧明. 刘钊. 李培峰. 王汝传 基于随机高级Petri网的主动自调度集群系统的性能分析[期刊论文]-通信学报 2006(12)

22. 王俊伟 大规模多媒体存储系统中数据放置与调度策略的研究[学位论文]博士 2005

23. 乔治 基于Linux负载均衡系统的研究与实现[学位论文]硕士 2005

24. 崔春焱 基于中间件的企业集群系统结构设计[学位论文]硕士 2006

25. 鄢娟 一种开放式网络计算平台[学位论文]硕士 2004

26. 汪黎 TCP连接迁移实现研究[学位论文]硕士 2004