

TCP 迁移技术报告

汪黎

目录

图目录.....	V
表目录.....	VI
摘要.....	VII
ABSTRACT.....	VIII
第一章 绪论.....	1
§1.1 研究背景.....	1
§1.2 研究现状.....	2
1.2.1 当前流行的集群调度技术.....	2
1.2.2 正在开展的研究项目.....	5
§1.3 报告的组织.....	6
第二章 Linux 的网络实现.....	7
§2.1 网络设备接口层.....	7
§2.2 网络接口核心层.....	8
§2.3 网络协议层.....	9
§2.4 网络接口 Socket 层.....	11
§2.5 本章小结.....	13
第三章 基于 Netfilter 的 TCP 迁移方法.....	14
§3.1 Netfilter 框架简介.....	14
3.1.1 Netfilter 框架的结构和功能.....	14
3.1.2 Netfilter 的应用实例：Linux 防火墙.....	16
§3.2 基于 Netfilter 的 TCP 迁移方法.....	17
3.2.1 技术思想概述.....	17
3.2.2 连接的建立过程.....	17
3.2.3 连接的通信过程.....	20
3.2.4 连接的取消过程.....	20
§3.3 测试.....	21
§3.4 本章小结.....	22

第四章 基于连接修改和传递技术的 TCP 迁移方法.....	23
§4.1 技术思想概述	23
§4.2 连接传递技术	24
§4.3 本章小结	25
第五章 基于重构连接现场的 TCP 迁移方法.....	26
§5.1 技术实现背景	26
5.1.1 网络相关系统调用的内部实现.....	26
5.1.2 TCP 连接的建立过程	30
5.1.3 网络缓冲区的基本操作.....	32
§5.2 技术实现概述	36
§5.3 技术实现难点	37
5.3.1 Handoff 协议	37
5.3.2 连接现场重构.....	37
5.3.3 HTTP 请求报文零拷贝传递.....	38
§5.4 本章小结	39
第六章 对持久连接 (P-HTTP) 的支持	40
§6.1 概述.....	40
6.1.1 提出背景.....	40
6.1.2 持久连接协议 (P-HTTP) 概述	40
6.1.3 持久连接带来的问题.....	41
6.1.4 国际上的解决办法.....	41
§6.2 技术思想概述	42
§6.3 实现算法	42
6.3.1 一次迁移过程.....	42
6.3.2 多次迁移过程.....	43
§6.4 BE 调度技术.....	44
§6.5 本章小结	45
第七章 大规模基于请求内容分发的系统 TCPHA.....	46
§7.1 系统体系结构	46
§7.2 系统工作过程.....	46

§7.3 系统实现的关键技术	47
7.3.1 服务器体系结构	47
7.3.2 ARP 问题及其解决方案: ARP 过滤	50
7.3.3 动态 IP 隧道技术	51
§7.4 性能测试与比较	52
§7.5 本章小结	52
第八章 总结与展望	53
参考文献	54

图目录

图 1.1 服务器集群的结构	1
图 3.1 Netfilter HOOK 位置	15
图 3.2 三方之间的 TCP 连接	19
图 3.3 CommView 显示的测试结果	21
图 3.4 浏览器显示的测试结果	22
图 4.1 Linux 网络相关数据结构实现	24
图 5.1 基本 TCP 客户-服务器程序编程模型	27
图 5.2 socket 系统调用实现	28
图 5.3 TCP 连接建立过程-服务器方	30
图 5.4 TCP 连接建立过程-客户方	31
图 5.5 sk_buff 与网络报文之间的关系	33
图 5.6 sk_buff_head 与 sk_buff 的关系	34
图 5.7 LINUX 2.4 的 sk_buff 与网络报文之间的关系	35
图 5.8 基于连接现场重建的 TCP 迁移方法	37
图 5.9 Handoff 请求报文格式	37
图 5.10 Handoff 响应报文格式	37
图 6.1 支持 P-HTTP 的集群调度技术	41
图 7.1 TCPHA 系统体系结构	46
图 7.2 TCPHA 系统报文交换时序图	47
图 7.3 TCPHA 调度器处理请求的简单步骤	48
图 7.4 多进程结构	48
图 7.5 多线程结构	49
图 7.6 单进程事件驱动结构	49
图 7.7 多线程事件驱动结构	50

表目录

表 7.1 TCPHA 与 Squid,KTCPVS 性能比较	52
---------------------------------------	----

摘要

本文首先介绍了国际上流行的集群调度技术，指出了实现 TCP 迁移技术的较大意义。然后，本文分析了 Linux 网络源代码的框架，逻辑上的各层间的功能和接口；分析了网络服务器的编程模型及相关系统调用的操作系统实现；分析了 TCP 连接的建立过程；接下来，本文提出了三种实现 TCP 迁移的方法，这三种方法是：基于 Netfilter 的 TCP 迁移方法；基于连接修改和传递的 TCP 迁移方法；基于重建连接现场的 TCP 迁移方法；还提出了解决持久连接问题的方法：多重迁移算法（multi-handoff）。最后，本文介绍了基于连接现场重建 TCP 迁移方法的服务器集群调度系统的实现。

关键词：操作系统，Linux，服务器集群系统，TCP 迁移

ABSTRACT

At first,the popular server cluster scheduling technologies is introduced,the significance of implementing TCP Handoff is pointed out.We analysis the framework of Linux network source codes and the function and interface between each layer in logic.and on the analysis of the programming model of network servers and implementing of interrelated system calls and the course of TCP connection establishing,we present three technologies of implementing TCP Handoff,which are TCP Handoff based Netfilter;TCP Handoff based connection modifying and connection transferring;TCP Handoff based connection locale reconstructing。 We also present a algorithm to support P-HTTP effectively,which is named multi-handoff algorithm.At last,a server cluster scheduling system is introduced,which is based the key technology:TCP Handoff based connection locale reconstructing.

Key words: Operating System, Linux, Server Cluster System, TCP Handoff

第一章 绪论

§ 1.1 研究背景

随着 Internet 在世界各地的飞速普及和发展，网络服务器的负载越来越重。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(Server Cluster)已成为实现高可伸缩、高可用网络服务的有效结构。服务器集群的结构如图 1.1 所示，它通常由一台前端调度器（简称为 FE，下同）和若干台后端服务器（简称为 BE，下同）组成，彼此之间通过高性能网络互联。整个集群共享一个虚拟 IP 地址，集群中只有 FE 对客户方可见，集群对客户方看来就像是一台高性能的服务器。所有的客户请求首先到达 FE，由 FE 将请求根据一定的负载平衡算法分发给 BE。采用集群技术的系统有以下优点：

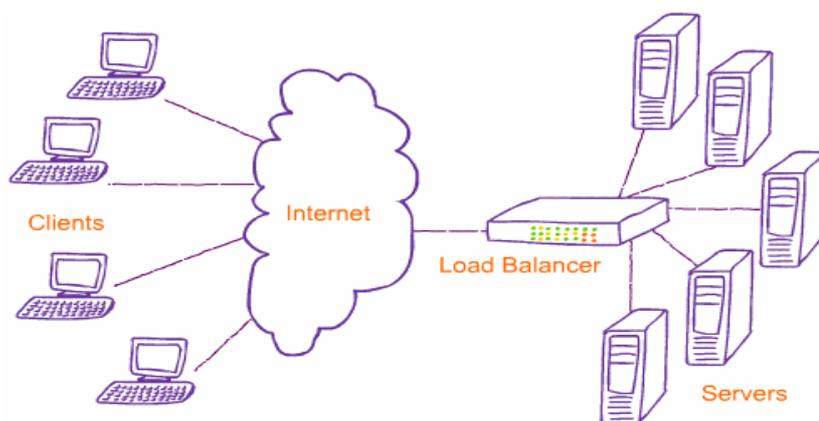


图 1. 1 服务器集群的结构

- 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，（这极大地依赖于集群调度系统）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。

但是，用集群系统来提供可伸缩网络服务是有难度的，表现在：

- 透明性 (Transparency)

如何高效地使得由多个计算机组成的松耦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单个服务器上。

- 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。当发现一个模块失效时，要将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)。

可见，优良的负载平衡技术，是充分发挥服务器集群性能的关键。

§ 1.2 研究现状

1.2.1 当前流行的集群调度技术

Web 服务器集群就是基于集群体系结构的 Web 服务器。对于 Web 服务器集群，当前流行的负载平衡技术，实际就是集群调度技术有：

- 基于 DNS 解析的技术

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^[33,34,35]。它的结构是：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，从而实现负载平衡。

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。当用户将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 域名服务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 间存在多台域名服务器，而它们都会缓冲已解析的名字到 IP 地址的映射，这会导致该域名服务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务器提交请求并进行重新映射。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不好。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，等所有域名服务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。

- 基于客户端的技术

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，进而以负载均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 等其他浏览器时，不可避免的要进行 RR-DNS 解析。

Smart Client^[36]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。

- 基于 IP 层的调度技术

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，由它进行负载均衡调度，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^[37]、Cisco 的 LocalDirector^[38]、Alteon 的 ACEDirector^[39]和 F5 的 Big/IP^[40]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，使得进行负载均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^[41]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^[42,43]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。这种方法具有高可伸缩性，但一套 IBM SP/2 和 NetDispatcher 需要上百万美金。

在贝尔实验室的 ONE-IP^[44]中, 每台服务器都有独立的 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。

- 基于请求内容的调度技术

即通常所称的第七层调度, 又称为基于内容的调度, 它综合考虑客户请求的内容, 如 URL 名称, 类型, Cookies 等。有关的研究表明, 基于内容的调度可以充分利用 Web 访问流中的局部性, 结合其它的相关技术, 如缓存协作技术^[3,18], 可以大大提高 BE 上的 Cache 命中率, 从而能较大幅度地提高 Web 服务器集群的性能。此外, 理论上, 基于请求内容的调度还允许集群提供的服务按内容存放在不同的 BE 上, 即不同的 BE 上的服务可以不同。而采用第四层调度策略, 则 BE 必须是对称的, 即任何一台 BE 都必须能够处理任何一个请求。将服务按内容存放, 可以大大减少资源浪费, 集群系统可以利用有限的资源提供更多的服务。该技术的实现又可细分为:

1. 基于应用层调度程序的技术

这种技术现在采用得较多, 其主要思想是在 FE 上运行一个基于应用层的调度程序, 用户所有的服务请求发给它, 它根据请求的内容或其他调度因素, 选中一台 BE, 与之建立连接, 并把服务请求转发给它, BE 处理完请求后, 将数据发给 FE, 再由其转给用户。应用层负载均衡调度的典型代表有 Zeus 负载调度器^[45]、pWeb^[46]、Reverse-Proxy^[47]和 SWEB^[25,48,49,50,51]等。Zeus 负载调度器是 Zeus 公司的商业产品, 它是由 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。

这种技术的优点是: 可以做到基于请求内容的转发, 可以方便的采用各种调度算法, 以实现较好的性能。但是它的缺陷也是显而易见的: FE 作为数据中转站, 它的数据处理能力将是系统性能的瓶颈。特别是 BE 将处理请求的结果数据均通过 FE 发送给用户, 而这些结果往往是大量的数据 (文件或图片), 这势必占用 FE 大量的处理能力和带宽, 很容易造成 FE 过载而难以解决负载均衡问题, 而且用户态, 核心态的转换太多, 时空开销太大。

2. 基于 TCP 粘合 (TCP Splicing) 的技术

该技术^[12,13,18,23]的实现思想与上一种方法类似, 只是在操作系统内核实现, 并采用一些优化措施, 如通过修改 FE 与客户方的连接和 FE 与 BE 的连接上的某些域, 使两条连接在 FE 处“粘合”起来, 从而可以采用数据零拷贝传递等技术。由于避免了数据报文的上下文切换工作, 减少了报文的拷贝开销, 减少了核心空间到用户进程的通信开销, 整体效率比

上一种方法要好一些。采用该技术的系统如 KNITS^[30], KTCPVS^[52]。

3. 基于 TCP 迁移的技术

TCP 迁移是最新的技术, 它的实现思想是 FE 根据客户的请求内容, 按照一定的调度规则, 调度到一台 BE。然后将客户方与 FE 的 TCP 连接对客户透明的迁移到 BE 上。BE 的响应直接发给客户, 不再经过 FE, 既有效的减少了客户端的响应延迟, 又大大降低了 FE 的负载, 从而有效的提高了整个集群系统的性能。但 TCP 迁移技术的实现有着较大的难度, 目前国际上也仅限于技术讨论阶段, 尚无实用的系统。

由于源代码共享的 GNU 精神和群策群力的集市软件开发模式, Linux 得到了用户、产业界和学术界的普遍认同和支持, 其开发呈现出百花齐放的势头。Linux 的性能非常稳定, 网络性能尤其突出, 已成为 Internet 上的主要服务器操作系统。因此, 基于 Linux 研究 TCP 迁移技术及基于该技术的服务器集群系统, 是非常方便的, 而且有着广阔的应用前景。此外, 随着 TCP 迁移技术研究的成熟, 可以在别的操作系统上实现该技术, 使之成为更加通用和实用的集群调度技术。另外, TCP 迁移技术的研究对于高性能计算领域中进程迁移技术的研究, 高可靠性技术中容错的研究, 以及移动自组网络中主机的透明动态迁移都有很大的帮助。

1.2.2 正在开展的研究项目

- LARD(Locality-Aware Request Distribution) Project

该项目^[1,4,6]是美国的Rice University和IBM在联合开展的项目, 他们在FreeBSD上实现了基于TCP迁移的集群调度系统并提出了LARD调度算法。在他们的论文中, 对TCP迁移的实现方法细节介绍非常少, 而且他们是基于FreeBSD实现的。在支持P-HTTP方面, 他们提出的方法是BE request forwarding, 该方法的开销较大。

- Socket Cloning

该方法^[5]是香港大学的研究人员提出来的, 他们在基于Linux的系统上实现了TCP迁移, 但需要修改操作系统协议栈, 从而要重新编译内核。而且他们实现的迁移对服务器应用程序不是透明的, 需要由服务器应用程序通过其提供的系统调用来显式地发起迁移。因而需要修改服务器应用程序。

- Modular TCP handoff design in STREAMS-based TCP/IP implementation

该方法^[7]是美国Hewlett-Packard实验室和Michigan State University的研究人员提出来的, 他们在HP-UX 11.0系统的基于流的TCP/IP实现上实现了TCP迁移, 但对于P-HTTP, 他们没有提出具体的解决方法。

- The Migrate Internet Mobility Project

麻省理工学院计算机科学实验室(美国) (MIT Lab for Computer Science) 正在开展的研究项目^[20,22], 他们的应用背景是在移动网络领域。他们的方法是在传统TCP报文中加入Migrate Tcp Options, 达到的目标是使移动主机可以在一条连接逻辑上不中断的条件下更换IP地址, 而保持与固定主机的连接。他们的方法也需要修改操作系统协议栈。

- MIGSOCK

MIGSOCK是CMU University开展的研究项目^[8], 该项目在Linux上实现了TCP连接迁移。但该方法主要用于进程迁移环境, 应用背景不同, 而且它需要由源端(客户方)主动发起, 因此对客户方是不透明的。还需要应用程序的参与, 利用MIGSOCK提供的API来显式地传递连接状态等, 因而对应用程序也是不透明的。需要重新编译内核。在传递

连接状态时，是将连接状态写入一个文件，通过网络文件系统NFS传送，这样开销较大。

- M-TCP (Migratory TCP)

M-TCP(Migratory TCP)^[9,10,11]，是由Rutgers University的Distributed Computing Lab的研究人员开发的，他们的方法是提出一种新的传输层协议，该协议是传统TCP协议的扩展，称为M-TCP协议 (Migratory TCP)。使用该协议的服务器集群中服务器之间可实现TCP连接的动态迁移，但迁移是由客户方主动发起的，因此对客户方不透明。使用此方法需要修改主机的操作系统，是在BSD上实现的。并且服务程序也必须使用M-TCP提供的API重新编写。

§ 1.3 报告的组织

全文共分八章，结构如下：

第一章“绪论”，介绍了 Web 服务器集群调度和 TCP 迁移的相关概念。

第二章“Linux 的网络实现”，研究了 Linux 网络代码的总体框架，数据包从物理网卡接收到传送给用户层的全过程。研究了网络代码逻辑上的各层之间的功能和接口。

第三章“基于 Netfilter 的 TCP 迁移方法”，研究了 Linux 中的 Netfilter 框架的结构，功能和实现，介绍了基于 Netfilter 实现 TCP 迁移的方法。

第四章“基于连接修改和传递的 TCP 迁移方法”，提出了通过修改 TCP 连接，以及利用连接传递技术实现 TCP 迁移的方法。

第五章“基于重构连接现场的 TCP 迁移方法”，在研究了传统 Web 服务器编程模型和相关系统调用实现的基础上，提出了基于重构连接现场实现 TCP 迁移的方法。

第六章“对持久连接 (P-HTTP) 的支持”，在研究 P-HTTP 的基础上，提出了基于 TCP 迁移的系统支持 P-HTTP 的算法：多重跳动算法 (Multi-Handoff)。

第七章“大规模基于请求内容分发的系统 TCPHA”，详细地介绍了基于重构连接现场 TCP 迁移方法的服务器集群调度系统 TCPHA 的实现。

第八章“总结与展望”，是对 TCP 迁移技术的总结和展望。

第二章 Linux 的网络实现

Linux 的网络代码源于伯克利 UNIX，大约占内核源代码的百分之二十。Linux 的网络代码从逻辑上大致可分为四层：网络设备接口层，网络接口核心层，网络协议层，以及网络接口 Socket 层。

网络设备接口层主要负责从物理介质接收和发送数据，包含各种网络设备的设备驱动程序。实现的文件在 `linux/driver/net` 目录下面。

网络接口核心层是整个网络接口的关键部分，它为网络协议层提供统一的发送接口，屏蔽各种各样的物理介质的细节，同时负责把来自下层的数据向合适的协议配送。它是网络接口的中枢部份。它的主要实现文件在 `linux/net/core` 目录下，其中 `linux/net/core/dev.c` 为主要管理文件。

网络协议层是各种具体网络协议的实现。Linux 支持 TCP/IP, IPX, X.25, AppleTalk 等多种协议，各种具体协议实现的源码见 `linux/net/` 目录下相应的名称。其中 TCP/IP(IPv4) 协议实现的源码在 `linux/net/ipv4`，其中 `linux/net/ipv4/af_inet.c` 是主要的管理文件。

网络接口 Socket 层为用户提供网络服务的编程接口，它实际上是 BSD 编程套接字接口函数在内核的实现。主要的源码在 `linux/net/socket.c`。

本章主要讲述 Linux 的网络实现的总体框架，对逻辑上各层的源代码和功能以及各层之间的衔接做一简要介绍。

§ 2.1 网络设备接口层

物理层上有许多不同类型的网络接口设备，在文件 `include/linux/if_arp.h` 的 28 行里定义了 ARP 能处理的各种物理设备的标志符。网络设备接口负责具体物理介质的控制，包括从物理介质接收以及发送数据，并对物理介质进行诸如最大数据包之类的各种设置。这里以比较简单的 3Com3c501 以太网卡的驱动程序为例，概述一下这层的工作原理。源码在 `Linux/drivers/net/3c501.c`。

网卡最主要的功能是完成网络数据的接收和发送。发送相对来说比较简单，在 `Linux/drivers/net/3c501.c` 的行 475 开始的 `el_start_xmit()`。这个函数就是实际向 3Com3c501 以太网卡发送数据的函数，具体的发送工作主要是根据具体设备的要求，对一些寄存器的读写，这里从略。

接收的工作相对来说比较复杂。一般说来，一个新的数据包到达，或者一个数据包发送完毕，都会产生一个中断。`Linux/drivers/net/3c501.c` 的 572 行开始的 `el_interrupt()` 函数就

是 CPU 处理该中断的代码。前半部份处理的是包发送完以后向 CPU 传递的相关信息，后半部份处理的是一个新到达的数据包，就是说接收到了新的数据。el_interrupt()函数并没有对新到达的数据包进行太多的处理，就交给了接收处理函数 el_receive()。el_receive()首先检查接收的包是否正确，如果是一个“好”包就会为包分配一个缓冲结构(dev_alloc_skb())，这样驱动程序对包的接收工作就完成了，通过调用上层的函数 netif_rx()(net/core/dev.c 1214 行)，把包交给上层。

数据包由物理设备向上层的传递，是通过驱动程序调用 netif_rx()(net/core/dev.c 1214 行)函数实现的。该函数是网络接口核心层和网络接口设备层联系的桥梁，因而所有的网卡驱动程序都需要调用该函数。

数据包由上往下的传递稍微复杂一点。网络接口核心层需要知道有多少网络设备可以用，以及每个设备的函数的入口地址等。系统将所有可用的网络设备的相关信息都登记在由 dev_base 开始的队列中，其数据结构为 struct net_device *dev_base (Linux/include/linux/netdevice.h 436 行)。对于网络接口核心层来说，所有的设备都是一个 net_device 结构，它在 include/linux/netdevice.h,line 233 里被定义，这是从网络接口核心层的角度看到的一个抽象的设备。该设备具有的功能通过该结构中的函数指针域来调用。

如果网络接口核心层需要由下层发送数据的时候，在 dev_base 队列中找到设备以后，就直接调用该设备的 hard_start_xmit()函数来让下层发送数据包。

驱动程序要让网络接口核心层知道自己的存在，当然要加入 dev_base 所指向的指针链，然后把自己的函数以及各种参数和 net_device 里的相应的域对应起来。加入 dev_base 所指向的指针链是通过函数 register_netdev(&dev_3c50)(linux/drivers/net/net_init.c, line 532)建立的，实际上就是向系统登记的过程。对于 3Com3c501 网卡，填充各种功能函数的操作是在 e11_probe1()(Linux/drivers/net/3c501.c)里进行的。该函数在网卡初始化时调用。进一步的赋值在 ether_setup(dev) (drivers/net/net_init.c, line 405) 里进行。我们注意到 dev->hard_start_xmit =&el_start_xmit，这样发送函数的关系就建立了，上层只需要调用 dev->hard_start_xmit()来发送数据。

§ 2.2 网络接口核心层

如上所述，网络接口核心层是通过 dev_base 指向的设备链知道驱动程序以及驱动程序的函数的入口的，而驱动程序则是通过调用网络接口核心层的函数 netif_rx()(net/core/dev.c 1214 行)把数据传递到上一层的。

网络接口核心层的上层是具体的网络协议，下层是驱动程序。现在讨论一下网络接口核心层和网络协议层部分的关系，这种关系不外乎也是数据包的接收和发送的关系。

网络接口核心层通过 `dev_queue_xmit()`(`net/core/dev.c,line975`)这个函数向网络协议层提供统一的发送接口，也就是说无论是 IP，还是 ARP 协议，以及其它各种底层协议，通过这个函数把要发送的数据传递给网络接口核心层。`dev_queue_xmit()`最后会调用 `dev->hard_start_xmit()`，而 `dev->hard_start_xmit()`会调用实际的驱动程序来完成发送的任务。例如上面的例子中，调用 `dev->hard_start_xmit()`实际就是调用了 `el_start_xmit()`。

现在讨论接收的情况。网络接口核心层通过函数 `netif_rx()`(`net/core/dev.c 1214 行`)接收了网络设备接口层发送来的数据，这时候当然要把数据包往上层派送。所有的协议族的下层协议都需要接收数据，如 TCP/IP 的 IP 协议和 ARP 协议，SPX/IPX 的 IPX 协议，AppleTalk 的 DDP 和 AARP 协议等都需要直接从网络接口核心层接收数据，网络接口核心层是如何将接收到的数据分发给这些协议的呢？与前面类似，也是通过一个登记表来解决的。该登记表就是 `static struct packet_ptype_base[16]`(`net/core/dev.c line 164`)数组。这个数组包含了所有需要接收数据包的协议，以及它们的接收函数的入口。

从该数组中登记的信息可知，IP 协议接收数据是通过 `ip_rcv()`函数的，而 ARP 协议是通过 `arp_rcv()`的，网络接口核心层只要通过这个数组就可以把数据交给上层函数了。

如果有协议想把自己添加到这个数组，是通过 `dev_add_pack()`(`net/core/dev.c, line233`)函数，从数组删除则是通过 `dev_remove_pack()`函数。IP 层的注册是在初始化函数 `ip_init`(`net/ipv4/ip_output.c, line 1003`)中进行的。

如前所述，网络接口核心层通过函数 `netif_rx()`(`net/core/dev.c 1214 行`)接收了驱动程序发送来的数据，由于该函数是在中断服务程序里面，所以并不能够处理太多的东西，剩下的工作就通过 `cpu_raise_softirq(this_cpu, NET_RX_SOFTIRQ)`交给软中断处理。从 `open_softirq(NET_RX_SOFTIRQ, net_rx_action, NULL)`可以知道 `NET_RX_SOFTIRQ`软中断的处理函数是 `net_rx_action()`(`net/core/dev.c, line 1419`)，`net_rx_action()`根据数据包的协议类型在数组 `ptype_base`里找到相应的协议，并从中得到相应的接收函数，然后把数据包交给接收函数，这样就将收到的数据包交给了上层协议，实际调用接收函数是通过 `net_rx_action()`里的 `pt_prev->func()`这一句。例如：如果数据包是 IP 协议的话，就调用 `ptype_base[ETH_P_IP]->func()`函数，该函数实际上就是 `ip_rcv()`，这样就把数据包交给了 IP 协议。

§ 2.3 网络协议层

该层实现各种具体的网络协议。所有 Linux 支持的网络协议的定义都在 `linux/include/linux/socket.h`里面，Linux 的 BSD Socket 定义了多至 32 种支持的协议族，其中 `PF_INET`就是我们最熟悉的 TCP/IP 协议族(IPv4，以下没有特别声明都指 IPv4)。以这个

协议族为例,看看这层是怎么工作的。实现 TCP/IP 协议族的主要文件在 `linux/net/ipv4/` 目录下面, `Linux/net/ipv4/af_inet.c` 为主要的管理文件。

在 Linux2.4.16 里面,实现了 TCP/IP 协议族里面的的 IGMP,TCP,UDP,ICMP,ARP,IP。我们先讨论一下这些协议之间的关系。IP 和 ARP 协议是需要直接和网络设备接口打交道的协议,也就是需要从网络核心层接收数据和发送数据。而其它协议 TCP,UDP,IGMP,ICMP 需要直接利用 IP 协议,从 IP 协议接收数据,以及利用 IP 协议发送数据,同时还要向上层 Socket 层提供直接的调用接口。可以看到 IP 层是一个核心的协议,向下需要和网络接口核心层交互,又要向上层协议提供数据的发送和接收的服务。

先看 IP 协议层。网络接口核心层如果接收到 IP 层的数据,通过 `ptype_base[ETH_P_IP]` 数组的 IP 协议对应的项指向的 IP 协议的 `ip_packet_type->ip_rcv()` 函数把数据包传递给 IP 层,也就是说 IP 层通过函数 `ip_rcv()(linux/net/ipv4/ip_input.c)` 从网络接口核心层接收到数据包。`ip_rcv()` 这个函数只对 IP 数据包做了一些校验和的检查工作,如果包是正确的就把包交给下一个处理函数 `ip_rcv_finish()`(注意调用是通过 `NF_HOOK` 这个宏实现的)。现在, `ip_rcv_finish()` 这个函数真正要完成一些 IP 层的工作了。IP 层要做的主要工作就是路由,要决定把数据包往那里送。路由的工作是通过函数 `ip_route_input()(linux/net/ipv4/route.c,line 1622)` 实现的。对于进来的包可能的路由有:

- 属于本地的数据(即是需要传递给 TCP, UDP, IGMP 这些上层协议的);
- 需要转发的数据包(网关或者 NAT 服务器之类的);
- 不可能路由的数据包(地址信息有误);

我们现在关心的是如果数据是本地数据的时候怎么处理。`ip_route_input()` 调用 `ip_route_input_slow()(net/ipv4/route.c, line 1312)`, 在 `ip_route_input_slow()` 里面的 1559 行 `rth->u.dst.input= ip_local_deliver`, 这就是判断到 IP 包是本地的数据包,并把本地数据包处理函数的地址返回。这样,路由工作完成了,返回到 `ip_rcv_finish()`。`ip_rcv_finish()` 最后调用 `skb->dst->input(skb)`, 从上面可以看到,这其实就是调用了 `ip_local_deliver()` 函数,而 `ip_local_deliver()` 接着就调用了 `ip_local_deliver_finish()`。现在真正到了往上层传递数据包的时候了。

现在的情形和网络接口核心层往上层传递数据包的情形非常相似,怎么从多个协议选择合适的协议,并且往这个协议传递数据呢?网络接口核心层通过一个数组 `ptype_base[16]` 保存了注册了的所有可以接收数据的协议,同样网络协议层也定义了这样一个数组 `struct net_protocol*inet_protos[MAX_INET_PROTOS](/linux/net/ipv4/protocol.c#L102)`,它保存了所有需要从 IP 协议层接收数据的上层协议(IGMP, TCP, UDP, ICMP)的接收处理函数的地址,每种上层协议对应数组中的一项,为一个 `inet_protocol` 结构,该结构中的 `handler` 函数

指针域登记的处理函数就是该协议接收数据包的处理函数。例如 TCP 协议的数据结构的定义是 `static struct inet_protocol tcp_protocol (linux/net/ipv4/protocol.c line67)`，其接收数据包的函数为 `tcp_v4_rcv`。其它协议的处理函数如 `igmp_rcv()`,`udp_rcv()`, `icmp_rcv()`。同样在 `linux/net/ipv4/protocol.c`，往数组 `inet_protos[MAX_INET_PROTOS]` 里面添加协议是通过函数 `inet_add_protocol()` 实现的，删除协议是通过 `inet_del_protocol()` 实现的。`inet_protos[MAX_INET_PROTOS]`初始化的过程在 `linux/net/ipv4/af_inet.c inet_init()`初始化函数里面。

如果在 Linux 启动的时候留意启动的信息，或者在 linux 下打命令 `dmesg` 就可以看到这一段信息：

IP Protocols: ICMP, UDP, TCP, IGMP。

也就是说现在数组 `inet_protos[]`里面有了 ICMP, UDP, TCP, IGMP 四个协议的 `inet_protocol` 数据结构，数据结构中包含了它们接收数据的处理函数。

Linux 2.4.16 在 `linux/include/linux/socket.h` 里定义了 32 种支持的 BSD socket 协议，常见的有 TCP/IP,IPX/SPX,X.25 等，而每种协议还提供不同的服务，例如 TCP/IP 协议通过 TCP 协议支持基于连接的服务，而通过 UDP 协议支持无连接服务。面对这么多的协议，向用户提供统一的接口是必要的，这种统一是通过网络接口 Socket 层来实现的。

§ 2.4 网络接口 Socket 层

在 BSD Socket 网络编程的模型下，利用一系列的统一的函数来实现网络通信。例如，一个典型的利用 TCP 协议通信的程序是这样的：

```
socket_descriptor = socket(AF_INET,SOCK_STREAM,0);
connect(socket_descriptor,地址,端口);
send(socket_descriptor," hello world" );
recv(socket_descriptor,buffer,1024,0);
```

第一个系统调用指定了协议族为 Inet 协议族(由参数 `AF_INET` 指定),即 TCP/IP 协议,同时是利用面向连接的服务(由参数 `SOCK_STREAM` 指定),这样就对应到 TCP 协议,以后的操作就是利用 Socket 的标准函数进行的。

从上面我们可以看到两个问题，首先 Socket 层需要根据用户指定的协议族(上面是 `AF_INET`) 从 32 种协议族中选择一种协议族来完成用户的要求，当协议族确定以后，还要把特定的服务映射到协议族下的具体协议，例如当用户指定的是面向连接的服务时，Inet 协议族会映射到 TCP 协议。

从多个协议中选择用户指定的协议，并把具体的处理交给选中的协议，这与网络接口

核心层向上和向下衔接的问题本质上是一样的，所以解决的方法也是一样的，同样还是通过数组。在 `Linux/net/socket.c` 定义了这个数组 `static struct net_proto_family*net_families[NPROTO]`。其中 `net_families[2]` 是 TCP/IP 协议，`net_families[3]` 是 X.25 协议，具体哪一项对应什么协议，在 `include/linux/socket.h` 有定义。但是每一项的数据结构 `net_proto_family` 的 `ops` 域是空的，也就是具体协议族处理函数的地址是不知道的。协议的处理函数和 `ops` 域建立联系是通过 `sock_register()(Linux/net/socket.c)` 这个函数建立的，例如 TCP/IP 协议是这样建立关系的：

```
int __init inet_init(void) (net/ipv4/af_inet.c)
{
    (void) sock_register(&inet_family_ops);
}
```

只要给出 `AF_INET`(在宏里定义是 2)，就可以找到 `net_families[2]` 里面的处理函数了。

协议的映射完成了，现在要进行服务的映射了。根据系统分层设计的思想，上层不需要知道下层的什么协议能对应特定的服务，这种映射由协议族自己完成。在 TCP/IP 协议族里，这种映射是通过 `struct list_head inetsw[SOCK_MAX](net/ipv4/af_inet.c)` 这个数组进行的，在谈论这个数组之前先看另外一个数组 `inetsw_array[](net/ipv4/af_inet.c)`，该数组是静态定义的，从其定义中可以看到：`SOCK_STREAM` 映射到了 TCP 协议，`SOCK_DGRAM` 映射到了 UDP 协议，`SOCK_RAW` 映射到了 IP 协议。现在只要把 `inetsw_array` 里的三项添加到数组 `inetsw[SOCK_MAX]` 就可以了，添加是通过函数 `inet_register_protosw()` 实现的。在 `inet_init()(net/ipv4/af_inet.c)` 里完成了这些工作。

还有一个需要映射的就是 `socket`，诸如 `accept,send,connect,release,bind` 等的操作函数是怎么映射的呢？是通过上面的 `inetsw_array` 数组中的项的 `ops` 和 `prot` 域来映射的。例如，对应 TCP 协议的项的 `prot` 域为 `tcp_prot`，`tcp_prot` 中的 `connect` 函数指针域赋值为 `tcp_v4_connect`。因此，用户调用 `connect()` 函数，其实就是调用了 `tcp_v4_connect()` 函数。

现在讲述网络接口 `Socket` 层和用户层的衔接。

系统调用 `socket(),bind(),connect(),accept(),send(),release()` 等是在 `Linux/net/socket.c` 里面实现的，系统调用对应的实现函数是相应的函数名加上 `sys_` 的前缀。

现在看看当用户调用 `socket()` 这个函数进行的工作：`socket(AF_INET,SOCK_STREAM,0)` 调用了 `sys_socket(),sys_socket()` 接着调用 `socket_creat(),socket_creat()` 就会根据用户提供的协议族参数在 `net_families[]` 里寻找合适的协议族，如果协议族没有被安装就要请求安装该协议族的模块，然后就调用该协议族的 `create()` 函数的处理句柄。根据参数 `AF_INET`，`inet_creat()` 就被调用了，`inet_creat()` 根据服务类型在数组 `inetsw[SOCK_MAX]` 中选择合适的

协议，并把协议的操作集赋给 socket，根据 SOCK_STREAM，TCP 协议被选中：

```
inet_creat(){
answer=inetsw [用户要求服务服务] ;
sock->ops = answer->ops;
sk->prot = answer->prot
}
```

对于其它的系统调用实现函数，将在后面的章节中讲述。

§ 2.5 本章小结

本章简要介绍了 Linux 内核源代码中网络部分的总体框架。Linux 的网络代码实现非常好地遵循分层设计原则，根据代码功能的逻辑结构，网络代码大致可分为四层：

- 网络设备接口层
- 网络接口核心层
- 网络协议层
- 网络接口 Socket 层

Linux 网络代码的良好的组织和编程风格，非常便于阅读，也非常利于在其上做开发。

第三章 基于 Netfilter 的 TCP 迁移方法

在 Linux 内核 2.4 中，引入了一个最新的编程框架：Netfilter。它提供了非常好的接口用于抓取、截获、分析网络上的数据包。基于 Netfilter 的 Linux 防火墙，有着非常优秀的性能。

本章首先介绍 Netfilter 编程框架，然后提出基于 Netfilter 框架的 TCP 迁移方法，并给出测试结果。

§ 3.1 Netfilter 框架简介

3.1.1 Netfilter 框架的结构和功能

Netfilter 框架是随着 Linux 防火墙技术的发展而诞生的。Linux 的防火墙技术经历了若干代的沿革，最早使用的 ipfwadm 是 Alan Cox 在 Linux 发展的初期，从 FreeBSD 的内核代码中移植过来的。后来经历了 ipchains，再经由 Paul Russell 在 Linux kernel 2.3 系列的开发过程中发展了 Netfilter 这个架构。而用户空间的防火墙管理工具，也相应的发展为 iptables。它提供了对 2.0.x 内核中的 ipfwadm 以及 2.2.x 内核中的 ipchains 的兼容。Netfilter/iptables 这个组合目前相当的令人满意。在经历了 Linux kernel 2.4 和 2.5 的发展以后，的确可以说，Netfilter/iptables 经受住了大量用户广泛使用的考验。

Netfilter 在 Linux 内核中的 IPv4、IPv6 和 DECnet 等网络协议栈中都有相应的实现。在编译 Linux 内核时，Netfilter 作为一个在编译过程中可选的部件。

在 IPv4 协议栈中，为了实现对 Netfilter 架构的支持，在 IP 数据包在 IPv4 协议栈中的游历路线之中，仔细选择了五个钩挂（HOOK）点。在这五个钩挂点上，各引入了一行对 NF_HOOK() 宏函数的一个相应的调用。这五个钩挂点被分别命名为 PREROUTING, LOCAL-IN, FORWARD, LOCAL-OUT 和 POSTROUTING。它们的含义如下：

NF_IP_PRE_ROUTING：在接收到的报文作路由之前；

NF_IP_FORWARD：在接收到的报文转向另一个 NIC（转发）之前；

NF_IP_POST_ROUTING：在报文发送之前；

NF_IP_LOCAL_IN：在接收到的报文作路由，确定是本机接收的报文之后；

NF_IP_LOCAL_OUT：在本地报文作发送路由之前。

如图 3.1 所示：

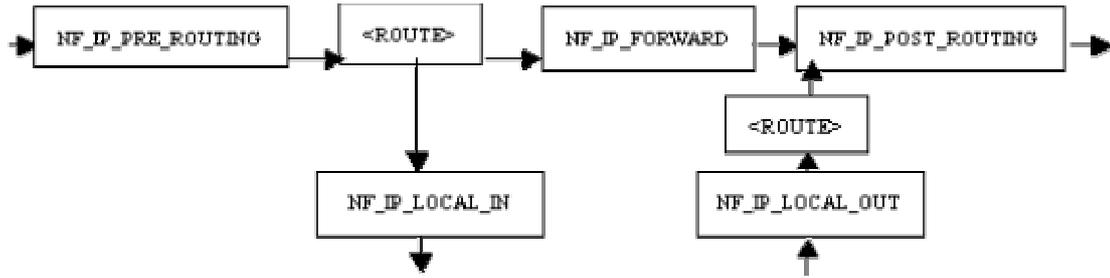


图 3. 1 Netfilter HOOK 位置

NF_HOOK() 这个宏函数，定义在 linux-2.4.19/include/linux/netfilter.h 里面,它的实现代码如下:

```

#ifdef CONFIG_NETFILTER
#define NF_HOOK(pf, hook, skb, indev, outdev, okfn) \
    nf_hook_slow((pf), (hook), (skb), (indev), (outdev), (okfn))
#else
#define NF_HOOK(pf, hook, skb, indev, outdev, okfn) (okfn)(skb)
#endif /*CONFIG_NETFILTER*/

```

从该宏函数的实现可知，当 #ifdef CONFIG_NETFILTER 这个条件编译被满足的时候，就转去调用 nf_hook_slow() 函数；如果 CONFIG_NETFILTER 没有被定义，则调用 NF_HOOK 宏中的参数 okfn，这实际上是一个函数指针，指出从 Netfilter 模块转回到 IPv4 协议栈，继续往下处理时要执行的函数。如果 CONFIG_NETFILTER 被定义，执行 nf_hook_slow() 函数的最后，实际上也会调用 okfn。这样就给了用户在编译内核的时候一个选项，可以通过定义 CONFIG_NETFILTER 与否来决定是否把 Netfilter 支持代码编译进内核。现在来看一下 NF_HOOK 宏即 nf_hook_slow()的调用时机。

其中一个调用的地方是 ip_rcv()函数，它是 IP 层接收报文的总入口，函数的代码如下：

```

ip_rcv() {
.....
return NF_HOOK(PF_INET, NF_IP_PRE_ROUTING, skb, dev, NULL,
    ip_rcv_finish);
}

```

ip_rcv()只是对接收到的 IP 报文进行了一下校验和检查，没有对其进行路由，然后就调用了 nf_hook_slow() 函数。nf_hook_slow()函数的主要功能就是根据 pf 和 hook 参数（这里分别为 PF_INET,NF_IP_PRE_ROUTING），索引数组 nf_hooks。nf_hooks 是一个二维数组，定义如下：struct list_head nf_hooks[NPROTO][NF_MAX_HOOKS]。可见数组中每个元素为一个链表头元素，即每个数组元素可以延伸出一个链表。链表中每个元素为一个 nf_hook_ops 结构。nf_hook_slow()函数在索引得到对应的链表后，它会遍历这个链表，对

链表中的每个元素，执行其 hook 函数。并根据 hook 函数的返回值，决定对 IP 数据包进行的下一步处理。

Hook 函数可以有五种返回值，它们对应的意义如下：

NF_ACCEPT: 继续正常的报文处理；

NF_DROP: 将报文丢弃；

NF_STOLEN: 由 hook 函数处理了该报文，不再继续传送；

NF_QUEUE: 将报文入队，通常交由用户程序处理；

NF_REPEAT: 再次调用该 hook 函数。

如果 hook 函数返回的值是 NF_ACCEPT，则 nf_hook_slow()函数调用它参数中的 okfn 函数，这里为 ip_rcv_finish 函数，继续协议栈对报文的处理。其它返回值对应的处理如上所述。可见，用户可以通过在链表中加入自己的处理函数（hook 函数），使得协议栈在处理报文的典型阶段，调用用户自己的处理函数，并由用户决定协议栈是否对该报文进行继续处理。

向 nf_hooks 数组登记处理函数是通过 nf_register_hook()实现的。用户通过填充一个 nf_hook_ops 结构，指出该函数的调用点（五个钩挂点之一）和对应的处理函数。然后以该结构的指针为参数，调用 nf_register_hook()函数即可实现向系统注册自己的处理函数。

3.1.2 Netfilter 的应用实例：Linux 防火墙

现在来看 Linux 的防火墙的结构。Linux 的防火墙由 iptables 和 Netfilter 两大部分组成。Netfilter 提供在网络协议栈处理流程的典型点上注册处理函数的功能。Iptables 则存储处理规则。Iptables 是通过 table（表）来组织防火墙规则的，一个 table 就是一组类似的防火墙规则的集合。iptables 里面默认定义了三个 table: filter, mangle 和 nat, 绝大部分报文处理功能都可以通过在这些内建（built-in）的表格中填入 rule（规则）完成。

filter, 该模块的功能是过滤报文，不作任何修改，或者接受，或者拒绝。它在 NF_IP_LOCAL_IN、NF_IP_FORWARD 和 NF_IP_LOCAL_OUT 三处注册了 hook 函数，也就是说，所有报文都将经过 filter 模块的处理。

nat, 网络地址转换（Network Address Translation），该模块以 Connection Tracking 模块为基础，仅对每个连接的第一个报文进行匹配和处理，然后交由 Connection Tracking 模块将处理结果应用到该连接之后的所有报文。nat 在 NF_IP_PRE_ROUTING、NF_IP_POST_ROUTING 注册了 hook 函数，如果需要，还可以在 NF_IP_LOCAL_IN 和 NF_IP_LOCAL_OUT 两处注册 hook 函数，提供对本地报文（出/入）的地址转换。nat 仅对报文头的地址信息进行修改，而不修改报文内容，按所修改的部分，nat 可分为源 NAT（SNAT）和目的 NAT（DNAT）两类，前者修改第一个报文的源地址部分，而后者则修改

第一个报文的目的地址部分。SNAT 可用来实现 IP 伪装，而 DNAT 则是透明代理的实现基础。

mangle，属于可以进行报文内容修改的 IP Tables，可供修改的报文内容包括 MARK、TOS、TTL 等，mangle 表的操作函数嵌入在 Netfilter 的 NF_IP_PRE_ROUTING 和 NF_IP_LOCAL_OUT 两处。

内核编程人员还可以通过注入模块，调用 Netfilter 的接口函数创建新的 iptables。

§ 3.2 基于 Netfilter 的 TCP 迁移方法

3.2.1 技术思想概述

该技术的主要思想是通过 Netfilter 框架的支持，模拟成客户，在 BE 上“重现”连接的建立过程，即在 FE 上复制并缓存客户方在连接建立过程中发送来的报文。在决定迁移的 BE 后，通过 IP 隧道，将这些报文封装后发送给 BE，BE 根据 IP 隧道协议解开封装后，报文被 TCP 迁移系统的内核模块所截获，模块从中依次提取出各个报文，然后伪装成客户与 BE 的协议栈进行三次握手过程，使连接从 FE 迁移到 BE 上。其中根据客户方发出的 ACK 报文，可以推测出 FE 的序列号，这样可以通过两种方法使 BE 的序列号与 FE 一致：

(1) 由 BE 上的内核模块每次截获该连接上的报文，修改序列号，使其在客户方和 BE 看来保持一致，然后再传送给 BE 的协议栈正常处理。这种方法的优点是具体操作系统实现无关，对操作系统协议栈透明，对上层应用透明，甚至不需要代码在内核运行，只要操作系统提供截取数据包的相应功能即可。这一优点也使得该技术也可在基于 Windows 的操作系统中采用。

(2) 由 BE 上的内核模块直接修改内存中的 TCP 协议实现的相关数据结构的序列号域。这种方法的优点是“一劳永逸”，但与具体操作系统实现相关。

利用 Netfilter 框架支持，在集群 FE 和 BE 上，分别挂接 IP 报文遍历的 IP_LOCAL_IN 链和 IP_LOCAL_OUT 链两处，用于截取/改写 TCP 报文。挂接 IP_LOCAL_IN 链，将截获所有发送给本机的 IP 报文。而挂接 IP_LOCAL_OUT 链，将截获所有本机将要发出的 IP 报文。在 FE 上，模块的功能主要是记录连接，迁移信息，分析 HTTP 请求内容，转发报文给 BE。在 BE 上，则主要是改写序列号，丢弃某些响应报文，以达到“欺骗”BE 的目的。通过这些方法，来实现 TCP Handoff。

可见，该方法由于完全在模块注册的回调 HOOK 函数中实现，因而对操作系统协议栈是透明的，无需修改操作系统协议栈，因而也就无需重新编译内核。

3.2.2 连接的建立过程

假设有三台主机 A，B，C。A 是客户，B 是集群的前端机，C 是集群后端服务器中迁移的目的结点。实现 TCP Handoff 的模块名为 KTH (Kernel TCP Handoff)，KTH 分别安

装在 B, C 上。

系统的工作过程如下:

Step 1: 客户端 A 向集群发起请求, 典型的请求以一个 SYN (CSEQ 表示客户端报文的初始序列号) 报文为开始, 标记一个 TCP 连接开始并请求回应一个三次握手协议过程。该请求到达前端调度器 B。

Step 2: 前端调度器 B 上的 KTH 系统将截获客户端的 SYN 请求报文, 解析后, 将其复制一份, 缓存下来, 另外记下该连接的发送方地址和端口, 用于计算 hash 值, 将该连接记录到 Hash 表中, 连接的状态记录为 SYN_REVD。再将该报文上传给协议栈正常处理。

Step 3: 前端调度器 B 回应一个 ACK (CSEQ+1, 表示对应于初始序列的应答序列) 到客户端 A。同时, 发起一个 SYN (VSEQ 表示新的报文序列号)。该 ACK+SYN 报文同样在发送前由 KTH 截获, KTH 只是简单地将该连接的状态改为 SYNACK_SENT, 然后让该报文正常发送出去。

Step 4: ACK+SYN 报文回送到客户端 A, A 接收响应。

Step 5: A 在收到集群的正确应答报文后, 认为连接已经建立 (实际上, A 仅仅是和前端调度器 B 建立了连接)。对集群调度器 B 的 SYN (VSEQ) 进行应答, 发送 ACK (VSEQ+1)。至此完成了三次握手的全部过程, 并且开始进行数据通信, 向 B 发出数据请求报文 (这里假设为 HTTP 请求报文) DATA (CSEQ+1)。

Step 6: ACK (VSEQ+1) 报文到达 B 后, 由 B 上的 KTH 系统复制一份缓存起来, 并将该连接的状态改为 ESTABLISHED, 然后交给协议栈正常处理。这里, 记录连接状态的目的主要是为了实现精确的同步和报文的正确处理, 例如, 在接到客户方重传报文的情况下, 不进行不必要的报文复制。另外每次状态转换均重新启动对应新状态的定时器。如果定时器超时, 表明连接超时, 则 B 从 Hash 表中删除该连接对应的表项。

Step 7: 当客户方的 HTTP 请求到达时, 同样由 B 上的 KTH 截获, KTH 分析该 HTTP 请求, 并根据请求内容进行规则匹配, 从而调度到后端的一台服务器 C, 在 Hash 表中登记该服务器地址, 然后将先前复制的 SYN 报文, ACK 报文, 连同该 HTTP 请求报文, 封装为一个报文, 通过 IP 隧道发送给选出的后端服务器 C, 同时, “安静地” 撤消该连接, 所谓 “安静地” 撤消, 是指并不遵循 TCP 协议, 发出一个 FIN 报文, 向连接的对方发出信息表示己方撤消连接, 而只是直接撤消连接的一切相关数据结构, 这类似于一条 TCP 连接因连接的对方不可达 (例如中途将网线拔了), 超时而撤消连接的情况。同时修改 Hash 表中该连接的状态为 HANDOFFED。B 以后只负责将 A 发来的报文在 IP 层封装后通过 IP 隧道转发给 C。

Step 8: B 刚才转发的报文首先被后端服务器 C 上的 KTH 系统截获, KTH 先直接将该 IP 隧道报文交给 IP 隧道设备解开封装。

Step 9: IP 隧道设备解开封装后, 将原始 IP 报文重新放入协议栈中, 又重新被 KTH 系统截获。KTH 系统解析该报文, 从中提取出客户方 A 原始的 SYN (CSEQ) 报文, 记下报文的初始序列号 CSEQ, 然后直接交给协议栈。同时, 将该连接记录到 C 的连接 Hash 表中, 连接的状态记录为 SYN_REVD。

Step 10: C 产生 SYN+ACK 报文, 注意, 这里后端服务器 C 产生的初始序列号与前端调度器 B 产生的是肯定不同的, 该报文同样由 KTH 系统截获, 记下该序列号为 RSEQ,

然后将该 SYN+ACK 报文“悄悄地”丢弃，即不再通过协议栈向下传输发送，但是又让协议栈认为该报文已正常处理完毕。Netfilter 通过 HOOK 函数返回 NF_STOLEN 支持该操作。同时修改连接的状态为 SYNACK_SENT。

Step 11: 接着，KTH 从先前的 IP 报文中又提取出客户方 A 的 ACK (VSEQ+1) 报文，计算偏差 $OFFSET=RSEQ-VSEQ$ ，根据 TCP 协议，该偏差是固定的。利用这个偏差，我们就可以修正序列号，从而将连接在客户方 A 与后端服务器 C 之间建立起来。

Step 12: 修改该 ACK (VSEQ+1) 报文的 TCP 报文头中的 ack_seq 域的值，将其加上 OFFSET，这样，实际上该报文变为了 (RSEQ+1) 报文，重新计算校验和（实际上，这里有个小技巧，只需设一个标志位，可以使 Linux 操作系统协议栈跳过校验和检查，从而不用重新计算校验和。但对要发送出去的报文，在修改序列号后，就必须重新计算校验和了），修改该连接状态为 ESTABLISHED，将报文交给协议栈。以后 C 上的 KTH 截获到的所有该连接上收到的报文，修改序列号的方法均为将其加上 OFFSET，下文不再赘述。这样，C 就以为是它发出去的 SYN+ACK 报文的响应报文到了，这样，三次握手完成，后端服务器 C 与客户 A 也建立了“相同的”连接。这样，实际上该 TCP 连接在客户方和调度器之间，客户方和后端服务器之间，均处于半连接状态，如图 3.2 所示：

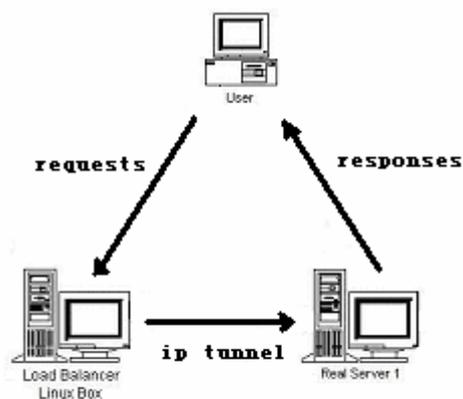


图 3. 2 三方之间的 TCP 连接

Step 13 : KTH 从先前的 IP 报文中最后提取出 HTTP 请求 (seq:CSEQ+1,ack_seq:VSEQ+1) 报文，KTH 将 ack_seq 域的值加上 OFFSET，即将该报文变为 (seq:CSEQ+1,ack_seq:RSEQ+1) 报文，重新计算校验和，交给协议栈。

Step 14: 后端服务器 C 响应该 HTTP 请求，响应报文 (seq:RSEQ+n,ack_seq:CSEQ+2) 同样在发送前由 KTH 系统截获。

Step 15: KTH 将报文的 seq 域减去 OFFSET，变为 (seq:VSEQ+n,ack_seq:CSEQ+2)，重新计算校验和，交给协议栈,发送给客户方 A。以后 C 上的 KTH 截获到的所有该连接上发送的报文，修改序列号的方法均为将其减去 OFFSET，下文不再赘述。A 以为是前端调度器 B 发来的响应，从而客户方也正确地得到了响应。

至此，实际上已完成连接的迁移工作，而该过程对客户 A 是透明的，A 认为连接的另一个端点是 B，而实际上已变成了 C。而对 C 看来，连接的另一个端点是 A，因而 HTTP 响应不通过 B 而直接发送给 A。

3.2.3 连接的通信过程

迁移的连接上的后续报文由 A 发送给 B, B 上的 KTH 模块截获报文后, 查找 Hash 表, 得到迁移的目的后端服务器 C 的地址。KTH 模块将报文使用 IP 隧道协议封装后, 转发给 C。C 上的 KTH 模块截获报文后, 修改报文的序列号, 然后交给协议栈正常处理。C 发出的响应报文同样在发送前由 KTH 模块截获, 修改报文的序列号, 重新计算校验和, 然后交给协议栈正常处理。

3.2.4 连接的取消过程

首先考虑客户方主动关闭连接的情况:

Step 1: 客户方 A 发起一个 FIN, 到达前端调度器 B。

Step 2: 前端调度器 B 记录该连接为 CLOSE_WAIT 状态, 将该报文封装后通过 IP 隧道转发给后端服务器 C。

Step 3: 后端服务器 C 上的 KTH 接到转发的 FIN, 同样记录该连接为 CLOSE_WAIT 状态, 将报文交给协议栈。

Step 4: 后端服务器 C 上的 KTH 接到将要发送的 ACK, 修改序列号, 计算校验和, 将报文交给协议栈发送。

Step 5: 后端服务器 C 上的 KTH 接到将要发送的 FIN', 修改序列号, 计算校验和, 将报文交给协议栈发送。

Step 6: 客户方 A 收到 FIN'后, 撤消连接, 发送响应 ACK 到达前端调度器 B, KTH 将该报文通过 IP 隧道发送给后端服务器 C。

Step 7: 后端服务器 C 上的 KTH 接到 ACK, 修改序列号, 交给协议栈, C 将撤消连接。一段时间后, B, C 由于 CLOSE_WAIT 状态超时, 从 Hash 表中删除连接的表项。这样, 连接在三方之间正确地撤消了。

考虑后端服务器 C 主动关闭连接的情况:

Step 1: C 发起一个 FIN, 到达客户方 A。

Step 2: 前端调度器 B 接到 A 发送的 ACK, 将其通过 IP 隧道转发给后端服务器 C。

Step 3: 前端调度器 B 接到 A 发送的 FIN', 记录该连接为 CLOSE_WAIT 状态, 将该报文封装后通过 IP 隧道转发给后端服务器 C。

Step 4: 后端服务器 C 上的 KTH 接到 ACK, 修改序列号, 交给协议栈。

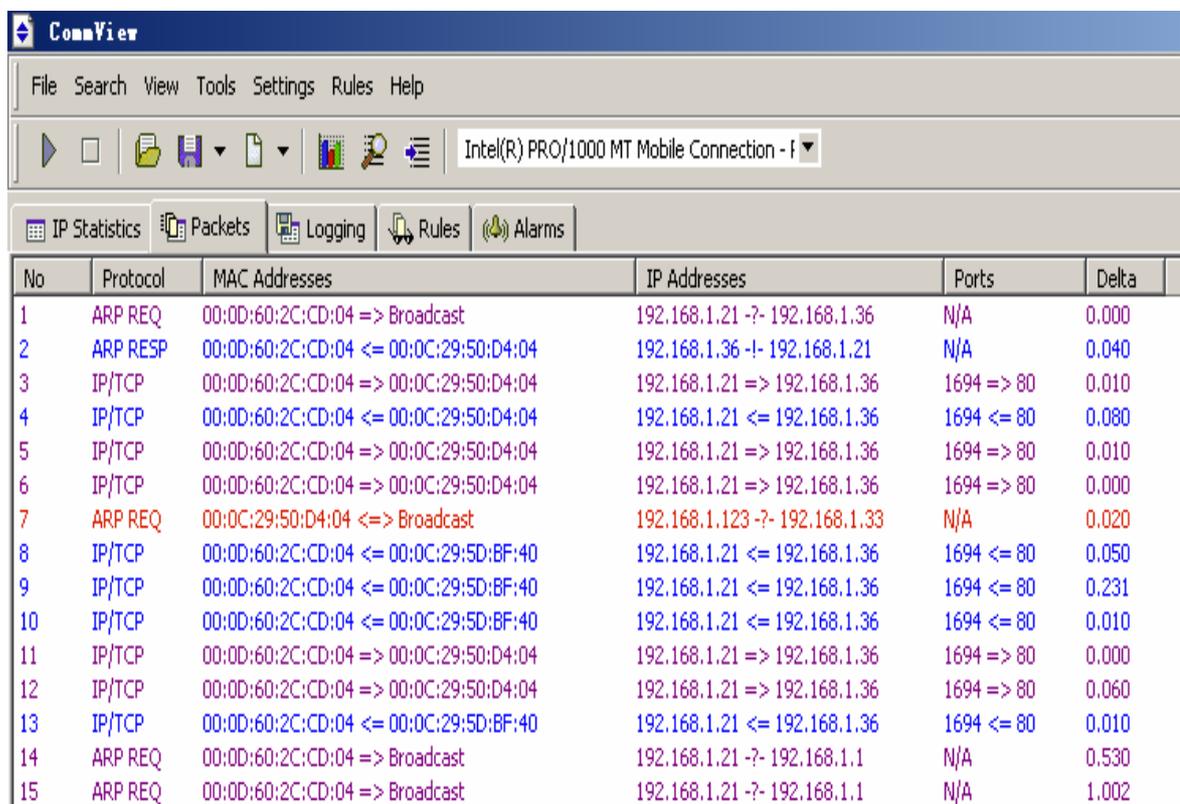
Step 5: 后端服务器 C 上的 KTH 接到 FIN', 记录该连接为 CLOSE_WAIT 状态, 修改序列号, 交给协议栈。C 的协议栈撤消连接, 向 A 发送 ACK。

Step 6: 后端服务器 C 上的 KTH 接到将要发送的 ACK, 修改序列号, 计算校验和, 交给协议栈发送。

Step7: 客户方 A 接到 ACK, 撤消连接。而一段时间后, B, C 由于 CLOSE_WAIT 状态超时删除 Hash 表中该连接的表项。这样, 连接在三方之间正确地撤消了。

§ 3.3 测试

我们采用三台机器 A, B, C。A 作为客户机, 其 IP 地址为 192.168.1.21, 运行 Windows, 并运行 CommView, 这是一个截获并分析网络数据包的软件。B, C 运行 Linux, B 作为集群前端调度器, 其 IP 地址为 192.168.1.36(以太网卡 eth0 的地址), 其 IP 隧道设备地址为 192.168.1.123。C 作为后端服务器, 其 IP 地址为 192.168.1.33, IP 隧道设备地址为 192.168.1.36。这样, B 转发报文时, 封装后的报文源地址是 192.168.1.123, 目的地址为 192.168.1.33。C 在接到报文并解开封装后, 看见的报文则是刚才由 A 发送给 B 的, 源地址为 192.168.1.21, 目的地址为 192.168.1.36。而 C 发现目的地址是它自己(因为它有一个 IP 地址也是 192.168.1.36), 将处理该报文并发送响应给 A。测试时, B、C 都打开 Web 服务器, 并在 Web 服务的根目录分别放一个 index.htm 文件, 文件内容分别为: “hello, this is a htm in 192.168.1.36” 和 “hello, this is a htm in 192.168.1.33”。B, C 都运行 KTH 模块, 图 3.3 是在 A 上的 IE 浏览器地址栏中输入 <http://192.168.1.36/index.htm>, CommView 显示的测试结果:



No	Protocol	MAC Addresses	IP Addresses	Ports	Delta
1	ARP REQ	00:0D:60:2C:CD:04 => Broadcast	192.168.1.21 -?-> 192.168.1.36	N/A	0.000
2	ARP RESP	00:0D:60:2C:CD:04 <= 00:0C:29:5D:D4:04	192.168.1.36 -!-> 192.168.1.21	N/A	0.040
3	IP/TCP	00:0D:60:2C:CD:04 => 00:0C:29:5D:D4:04	192.168.1.21 => 192.168.1.36	1694 => 80	0.010
4	IP/TCP	00:0D:60:2C:CD:04 <= 00:0C:29:5D:D4:04	192.168.1.21 <= 192.168.1.36	1694 <= 80	0.080
5	IP/TCP	00:0D:60:2C:CD:04 => 00:0C:29:5D:D4:04	192.168.1.21 => 192.168.1.36	1694 => 80	0.010
6	IP/TCP	00:0D:60:2C:CD:04 => 00:0C:29:5D:D4:04	192.168.1.21 => 192.168.1.36	1694 => 80	0.000
7	ARP REQ	00:0C:29:5D:D4:04 <=> Broadcast	192.168.1.123 -?-> 192.168.1.33	N/A	0.020
8	IP/TCP	00:0D:60:2C:CD:04 <= 00:0C:29:5D:BF:40	192.168.1.21 <= 192.168.1.36	1694 <= 80	0.050
9	IP/TCP	00:0D:60:2C:CD:04 <= 00:0C:29:5D:BF:40	192.168.1.21 <= 192.168.1.36	1694 <= 80	0.231
10	IP/TCP	00:0D:60:2C:CD:04 <= 00:0C:29:5D:BF:40	192.168.1.21 <= 192.168.1.36	1694 <= 80	0.010
11	IP/TCP	00:0D:60:2C:CD:04 => 00:0C:29:5D:D4:04	192.168.1.21 => 192.168.1.36	1694 => 80	0.000
12	IP/TCP	00:0D:60:2C:CD:04 => 00:0C:29:5D:D4:04	192.168.1.21 => 192.168.1.36	1694 => 80	0.060
13	IP/TCP	00:0D:60:2C:CD:04 <= 00:0C:29:5D:BF:40	192.168.1.21 <= 192.168.1.36	1694 <= 80	0.010
14	ARP REQ	00:0D:60:2C:CD:04 => Broadcast	192.168.1.21 -?-> 192.168.1.1	N/A	0.530
15	ARP REQ	00:0D:60:2C:CD:04 => Broadcast	192.168.1.21 -?-> 192.168.1.1	N/A	1.002

图 3. 3 CommView 显示的测试结果

分析前面 13 个报文序列, 1 是 A 发出的 ARP 请求, 由 B 响应(这可以从 MAC 地址看出来), 对应报文 2。3 是 A 发出的 SYN 报文, B 回应以 SYN+ACK(报文 4), A 再回应以 ACK(报文 5), 接着 A 发出 HTTP 请求报文 6, 7 则是 B 发出的 ARP 请求, 由 C 响应。注意这里的报文 8, 是 C 发给 A 的 ACK 报文, 报文 9 则是 HTTP 响应报文, 是由 C 发出的, 报文 10 是 C 发出的 FIN, 11 是 A 发出的 ACK, 12 则是 A 发出的 FIN, 13 由 C 回以 ACK, 这样, 整个请求过程结束。

A 上浏览器显示的内容如图 3.4 所示:

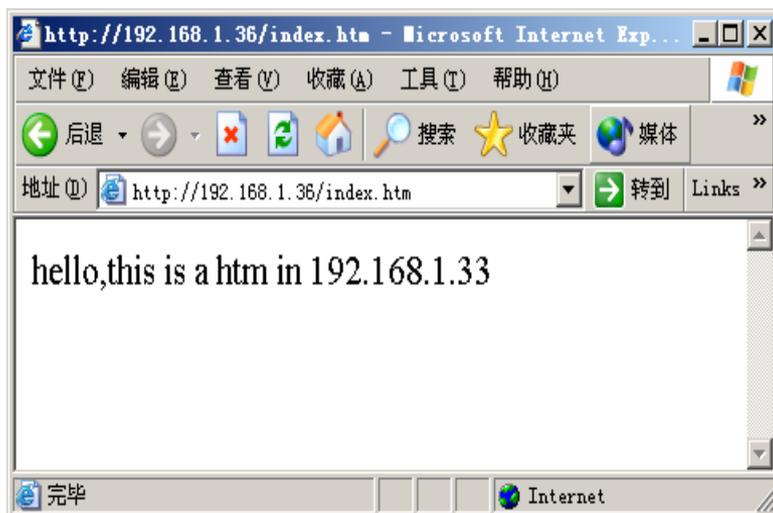


图 3. 4 浏览器显示的测试结果

从以上测试结果可以看出，确实正确地实现了 TCP Handoff 。

§ 3.4 本章小结

本章首先概要介绍了 Linux 内核 2.4 中最新的 Netfilter 框架的结构和功能。Netfilter 对 IPv4 的修改非常小，一是在若干个地方调用了 NF_HOOK(), 二是在 ip_sockopt() 中调用了 nf_sockopt()。Netfilter 框架为用户在协议栈处理过程中加入自己的处理函数提供了方便。

基于 Netfilter 框架的 TCP Handoff 方法有很多优势:

- 无需修改操作系统协议栈，无需重新编译内核，与操作系统实现无关;
 - 无需修改应用层服务程序，对应用层服务程序，客户方均透明;
 - 整个系统作为动态加载的驱动程序模块实现，使用起来非常方便;
 - 通用性很好，只要相应的操作系统提供截取数据包的功能，就可以实现该技术。因此，在 Windows 操作系统上，也可以实现该技术;
- 该技术也有一些缺点:
- 效率不够高;
 - 因为整个系统都在中断服务程序中实现，编程难度较大，调试较困难;
 - 实现对系统的配制管理较困难;

第四章 基于连接修改和传递技术的 TCP 迁移方法

本章在分析 Linux 的 TCP/IP 协议栈实现相关数据结构的基础上，提出基于连接修改和传递技术的 TCP 迁移方法。

§ 4.1 技术思想概述

TCP 迁移实现的关键问题，是如何在 BE 上建立起连接的镜像，即如何在没有客户方参与的情况下，建立起与客户方的连接。所谓 TCP 连接，是一个逻辑上的概念，它对应的是在操作系统内存中的一系列数据结构和一些表格中登记的表项。直观的想法，就是把这些数据结构全部拷贝过来，在 BE 的内存中建立起来，在 BE 的表格中登记相应的表项。这样做的缺点是：

(1) 通信量太大：一般操作系统实现中标识一条 TCP 连接的数据结构是相当大的，将近 1000 字节。FE 需要发送大量的数据，这使得系统的性能不高。而且，一些指针也不好处理。

(2) 细节太复杂：需要完全了解操作系统实现连接建立的每个细节，而且这个过程还可能随操作系统的版本不同而有差异。

实际上，建立连接的大部分工作是相同的，不同的只是不同的连接，一些域的值不同。连接修改和传递技术的思想是屏蔽掉操作系统建立连接的细节，通过修改连接，传递连接来实现 TCP 迁移。

基于连接修改和传递技术的 TCP 迁移方法的具体步骤是：

(1) 当 FE 决定要迁移一个连接时，它首先与 BE 上的迁移代理进程（后面简称为 HAP）建立一条连接；

(2) FE 提取连接的一些特征信息，如客户方的地址和端口。一般说来，特征信息是少量的，大约几十字节。FE 将特征信息通过建立的连接传送给 HAP；

(3) HAP 收到后，向 FE 发送确认；

(4) FE 收到确认后，向 HAP 发送收到的 HTTP 请求报文；

(5) HAP 收到后，向 FE 发送确认；

(6) FE 收到确认后，重置连接（撤消连接的一切数据结构），这里的连接包括与客户方的连接和与 HAP 的连接；

(7) HAP 利用连接特征信息修改与 FE 的连接，使之变为与客户方的连接，这只需要根据特征信息修改连接数据结构中的一些域；

(8) HAP 利用连接传递技术将该连接交给用户层的 Web 服务监听进程。

可见，该技术通过将 HAP 与 FE 的连接改为与客户方的连接，屏蔽了操作系统建立连接的细节，而且通信量很小，性能较高。

§ 4.2 连接传递技术

TCP 连接是一个静态的概念，可以看成是一种资源，它是有属主的。这个属主就是进程。上一步，我们只是得到了资源，还必须把它交给属主，才能为属主所利用。这里的属主，当然就是用户层 Web 服务程序。如何让用户层 Web 服务程序接受这一“凭空”构造出来的 TCP 连接，是一个关键的技术。这里，我们采用了连接传递技术，它可以将一个连接从一个进程传递给另一个监听进程。

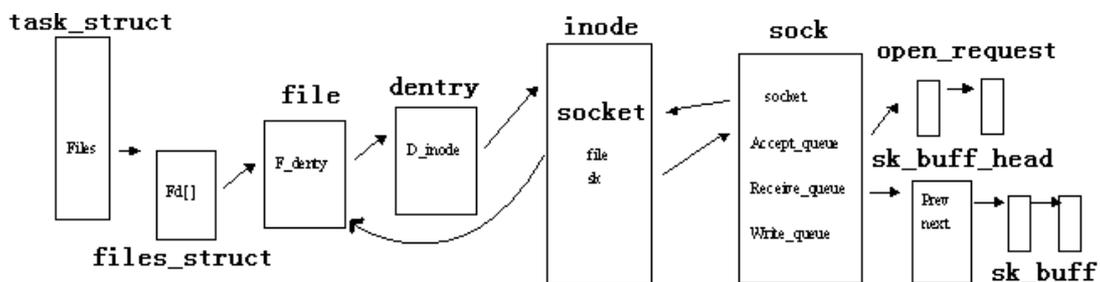


图 4. 1 Linux 网络相关数据结构实现

如图 4.1，在 Linux 操作系统的 TCP / IP 协议栈实现中，有一个非常重要的数据结构 **sock**，在该结构中登记了一条 TCP 连接的所有信息，包括 TCP 协议控制所需要的所有信息，IP 层的路由信息，构造 TCP，IP 报文所需的一切域的值。可以说，TCP 连接和 **sock** 结构是一一对应的。每个 **sock** 结构又通过 **socket** 结构与拥有该连接的进程挂钩。而 **sock** 结构的 **Receive_queue**, **Write_queue** 队列则分别记录了该连接上接收到的报文和将要发送的报文。**Accept_queue** 队列记录了所有已经完成连接建立过程（三次握手）的 TCP 连接，每一个 **open_request** 结构代表了一个已建好的连接，而其 **sk** 域指向的 **sock** 结构中则登记了该连接的全部信息。一般服务程序在监听套接口上等待，一旦客户方与该服务器建立了一个连接，则底层的网络协议栈实现生成该连接对应的 **open_request** 和 **sock** 结构，并将其挂到监听套接口的 **sock** 结构的 **Accept_queue** 队列上，并将监听的进程唤醒，这使得监听进程的 **accept** 系统调用返回。监听进程并不能区分该连接是不是操作系统协议栈建立起来的，它也不需要关心这一点，它只是认为有客户方与之建立起了一条 TCP 连接。

根据以上的分析，可以得出连接传递的步骤：

- (1) 将连接的 **sock** 结构与源进程脱链；
- (2) “伪造”一个 **open_request** 结构，即构造出该连接在初始建立时的环境；
- (3) 建立 **sock** 结构与 **open_request** 结构的联系；
- (4) 将 **open_request** 结构挂到目的监听进程的 **Accept_queue** 上；
- (5) 将目的监听进程唤醒。

另外还有一个问题，就是缺省的接收报文函数，是在读数据时，将报文从 **receive_queue**

摘下，这样，在 HAP 接到 HTTP 请求报文后，该请求报文已从连接的接收队列摘除。这样，当将该连接传递给用户层服务程序后，除非客户方重传 HTTP 请求，否则用户层服务器得不到 HTTP 请求。解决的办法是 Linux 支持带 MSG_PEEK 参数的接收函数，用它接收报文，只是将报文复制一份放入接收缓冲区，并不摘取报文，这样便可完整的将连接和连接上的报文传递给用户层服务程序。

对于该方法的测试，测试的设备和方法与前一章相同，详见 3.3 节。

§ 4.3 本章小结

本章讲述了基于连接修改和传递技术的 TCP 迁移方法，该方法的优点是：

- (1) 屏蔽了操作系统建立连接的细节；
- (2) 无需修改操作系统协议栈，无需重新编译内核；
- (3) 无需修改应用层服务程序，对应用层服务程序，客户方均透明。

该技术的缺点是：

- (1) FE 每次需要与 HAP 重新建立连接，性能不够高；
- (2) FE 每次迁移与 HAP 有两次报文交换。

第五章 基于重构连接现场的 TCP 迁移方法

本章首先介绍网络相关系统调用的内部实现和 TCP 连接建立的详细过程，然后介绍 Linux 报文缓冲区的实现，最后提出基于重构连接现场的 TCP 迁移方法。

§ 5.1 技术实现背景

5.1.1 网络相关系统调用的内部实现

通常的网络服务程序编程模型为套接字编程模型。套接字分为两类：监听套接字和普通套接字。对于普通套接字来说，一旦连接成功建立后，每一个普通套接字都标识一条 TCP 连接。一般的网络服务程序都是这样编写的：一个进程依次执行 `socket`, `bind`, `listen` 系统调用，打开一个监听套接字，初始化，绑定在该套接字上监听。然后调用 `accept` 系统调用进入阻塞，等待连接请求的到来。一旦客户方与之建立了一条 TCP 连接，`accept` 就会返回，返回值是一个新的普通套接字描述符，标识这条新生成的连接。服务程序派生出一个子进程来在该新生成的普通套接字描述符上进行操作，即服务，包括 `recv`（接收数据），`send`（发送数据）等系统调用。而父进程继续在原来的监听套接字上等待新的连接请求。

一次典型的客户与服务器之间采用 TCP 通信的函数调用过程如图 5.1:

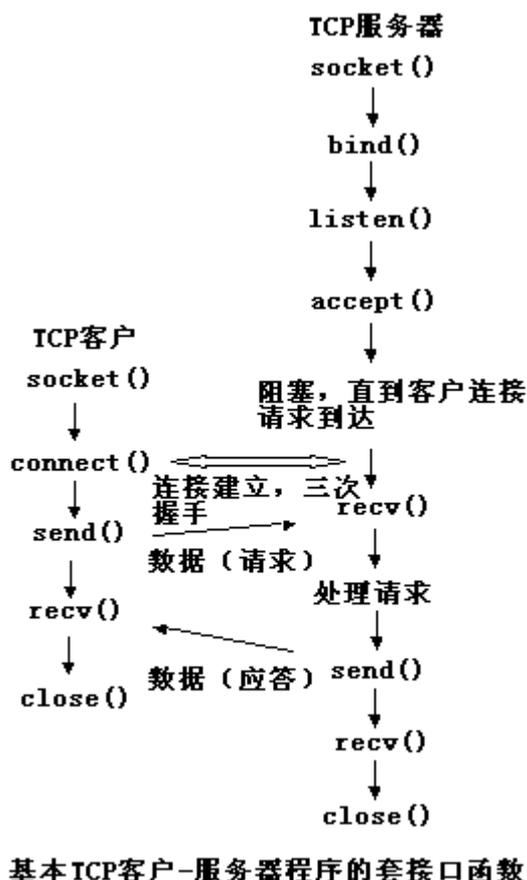


图 5. 1 基本 TCP 客户-服务器程序编程模型

下面分析有关系统调用的实现。

■ **socket**:指定通信协议类型，获得一个套接字描述符
用法:

```
#include <sys/socket.h>
int socket(int family,int type,int protocol)
```

实现:

首先，通过 int 0x80 软中断进入操作系统内核，调用 sys_socket_call,为 socket 系列系统调用的总入口，再根据具体的系统调用号转到 sys_socket。sys_socket 完成 socket 系统调用的所有功能：首先，分配一个新的 inode，并将 inode 中的联合 u 域解释为 socket 结构，表示这个 inode 不是一个普通的打开文件，而是一个用于网络通信的套接口。接着对 inode 和 socket 的部分域进行初始化。置 socket 的状态为未连接状态 (SS_UNCONNECTED)。分配一个 sock 结构,并对其初始化,置 sock 结构的状态为 TCP_CLOSE 状态。然后将 socket 和 sock 结构关联起来。接着查找当前进程的打开文件列表，找到一个未用的表项 fd,再得到一个 file 结构，生成一个 dentry 结构，将 inode 与 dentry，file 结构关联起来，并将 file 结构与当前进程打开文件列表的第 fd 项关联起来，最后返回 fd 作为套接字描述符。建立好后的结构如图 5.2:

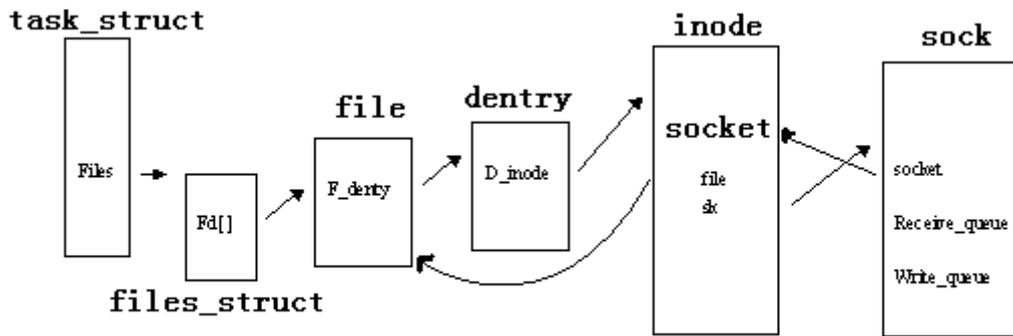


图 5. 2 socket 系统调用实现

■ **bind**:给套接口分配一个本地协议地址

用法:

```
#include <sys/socket.h>
int bind(int sockfd,struct sockaddr *myaddr,socklen_t addrlen)
```

实现:

由 `sys_socketcall` 转到 `sys_bind`,首先通过套接字描述符找到对应的 `socket` 结构,如上图,只要顺着指针链,便可由当前进程的 `task_struct` 找到对应的 `socket` 结构。将要绑定的地址信息拷贝入核心空间,进行一些合法性检查,如果未指定绑定的端口,则分配一个未用端口。并根据该信息填充 `sock` 结构的相关域。然后将该 `sock` 结构加入 `tcp_bhash` 哈希表。

■ **Listen**:将套接口变为一个监听套接口,并指定内核为此套接口排队的最大连接个数

用法:

```
#include <sys/socket.h>
int listen(int sockfd,int backlog)
```

实现:

首先找到套接字描述符对应的 `socket` 结构,置 `sock` 的域:联合 `tp_pinfo` 为 `tcp_opt` 结构,即传输层采用 `TCP` 协议,并初始化该 `tcp_opt` 结构,检测绑定的本地端口是否合法,置 `sock` 的状态为 `TCP_LISTEN` 状态,并将该 `sock` 结构加入 `tcp_listening_hash` 哈希表。

■ **Accept**:从已完成连接队列头返回下一个已完成连接,若已完成连接为空,则进程睡眠(阻塞方式)

用法:

```
#include <sys/socket.h>
int accept(int sockfd,struct sockaddr *cliaddr,socklen_t *addrlen)
```

实现:

首先找到套接字描述符对应的 `socket` 结构,分配一个新的 `inode,socket` 结构,并根据原来的 `socket` 对其赋值。然后计算等待时间,将当前进程加入等待队列,进入睡眠状态,唤醒的条件:已完成连接队列(`accept_queue` 队列)非空;进程收到信号;剩下的等待时间为 0;套接口已关闭;针对一般的情况,`accept_queue` 队列非空,即收到客户的连接请求,则从 `accept_queue` 队列头摘下一项,为一个 `open_request` 结构,根据 `open_request` 结构得到指向该已完成连接的 `sock` 结构的指针,将该 `sock` 结构与先前生成的 `socket` 结构关联起来,再将刚才生成的 `inode` 结构与当前进程关联,并根据 `sock` 结构的信息将客户方的地址

返回用户空间。

■ **Connect:**建立一个与 TCP 服务器的连接

用法:

```
#include <sys/socket.h>
int connect(int sockfd,const struct sockaddr *servaddr,socklen_t addrlen)
```

实现:

首先找到套接字描述符对应的 socket 结构,然后将服务器的地址信息拷贝入核心空间,若本地端口未指定,则分配一个未用的端口,得到路由信息,分配一个 sk_buff 结构,根据路由信息等填充 sock 结构的域,将 sock 结构置为 TCP_SYN_SENT 状态,将其加入 tcp_ehash 哈希表,根据服务器地址信息构造一个 SYN 分节,向服务器发送。接着计算等待时间,置当前进程为睡眠状态,唤醒条件: sock 的状态不是 TCP_SYN_SENT 或 TCP_SYN_RECV;当前进程接到信号;剩下的等待时间为 0;针对一般的情况,sock 的状态不是 TCP_SYN_SENT 或 TCP_SYN_RECV,(为 TCP_ESTABLISHED 状态)则表示已建立和服务器的连接,返回。

■ **Send:**向连接的对方发送数据

用法:

```
#include <sys/socket.h>
ssize_t send(int sockfd,const void*buff,size_t nbytes,int flags)
```

实现:

首先找到套接字描述符对应的 socket 结构,然后将要发送的信息拷贝入核心空间,构造一个 msghdr 结构,然后根据该 msghdr 结构分配 sk_buff 结构,填充 sk_buff 结构,同时根据 sock 结构中记录的 mss 值决定是否分片,最后将生成的 sk_buff 结构挂在 sock 结构的 write_queue 队列尾部,调用 IP 层的函数发送。

■ **Recv:**接收来自连接对方的数据

用法:

```
#include <sys/socket.h>
ssize_t recv(int sockfd,void *buff,size_t nbytes,int flags)
```

实现:

首先找到套接字描述符对应的 socket 结构,生成一个 msghdr 结构,计算等待时间,将当前进程置为睡眠状态,当其 sock 结构的 receive_queue 队列非空时将其唤醒,然后从 receive_queue 摘下 sk_buff,根据其内容填充 msghdr 结构,再根据 msghdr 结构将接收到的数据返回用户空间缓冲区。

■ **Close:**关闭套接口,终止 TCP 连接

用法:

```
#include <unistd.h>
int close(int sockfd)
```

实现:

首先找到套接字描述符对应的 socket 结构,如果该套接口加入了某组播组,则先退出该组播组,将 socket 结构与 sock 结构脱链,如果 sock 结构处于 TCP_LISTEN 状态,则置 sock 的状态为 TCP_CLOSE,将半开连接队列以及完成连接队列清空;如果 sock 结构中还

有未读的数据，则向连接的对方发一个 RST 分节。正常情况，向连接的对方发一个 FIN 分节，如果指定的等待时间不为 0，则将当前进程置入等待队列。如果 sock 结构处于 FIN_WAIT_2 状态，则计算接收 FIN 分节的超时时间，如果大于 TIME_WAIT 状态的时间，则重置保活定时器，否则进入 TIME_WAIT 状态。当 sock 结构的状态为 TCP_CLOSE 后，撤消该 sock 结构。

5.1.2 TCP 连接的建立过程

下面分析 TCP 连接建立的详细过程。

■ 服务器方

首先用户层的应用程序通过 socket,bind,listen,accept 一系列系统调用，建立 sock 结构，并使其处于 TCP_LISTEN 状态。当前进程阻塞等待 accept_queue 队列非空，即客户方连接的建立。接下去是底层网络实现做的工作，其过程如图 5.3:

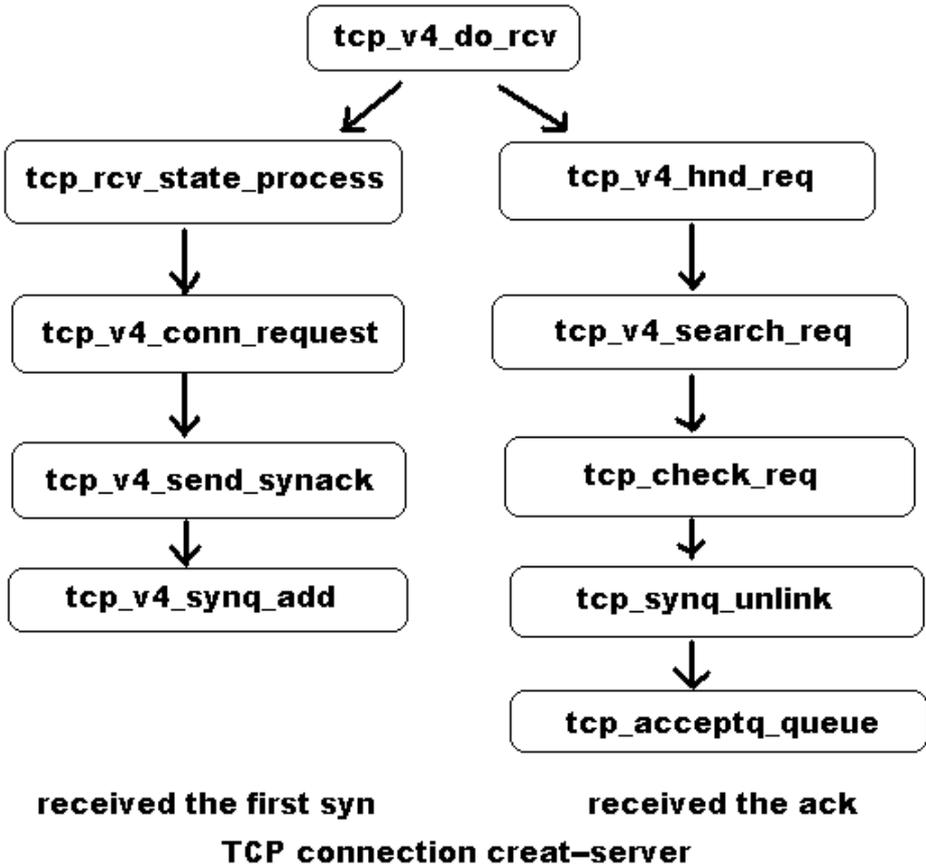


图 5. 3 TCP 连接建立过程-服务器方

当连接请求报文 (SYN 分节) 到达时，首先是 TCP 总的报文接收函数: tcp_v4_do_rcv, 对报文格式，校验和做一些检查。然后调用 tcp_rcv_state_process, 该函数判断如果 sock 结构处于 TCP_LISTEN 状态，并且接到的报文是 SYN 分节 (置了 SYN 标志位)，则调用 tcp_v4_conn_request, 对报文进行进一步检查，如果是广播或多播报文，则丢弃，接着检查半开连接队列及已完成连接队列是否已满，及相关安全性检查。然后分配一个

open_request 结构，对其初始化，如设置 TCP 最大报文段长度，本地地址，对方地址，重传时间，超时时间，初始的发送序列号等。接着向发送 SYN 的地址发送 SYN_ACK 分节。并把刚才生成的 open_request 结构加入 sock 结构的半开连接队列 (syn_table)。

当刚才 SYN_ACK 分节的 ACK 到达时，同样首先是 TCP 总的报文接收函数 tcp_v4_do_rcv 接到由 IP 层交来的报文。该函数判断当 sock 结构处于 TCP_LISTEN 状态时，调用 tcp_v4_hnd_req。该函数首先得到 TCP 报文的头部，然后调用 tcp_v4_search_req，在半开连接队列中查找，是否是相应半开连接的 ACK 报文。如果是，则调用 tcp_check_req，该函数首先调用 tcp_v4_syn_recv_sock 对报文的合法性进行检查，并查看完成连接队列 (accept_queue) 是否已满，若未滿，则调用 tcp_v4_route_req 得到路由信息，接着调用 tcp_create_openreq_child，该函数首先分配一个新的 sock 结构，把新生成的 sock 结构的状态置为 TCP_SYN_RECV 状态。并对该 sock 结构根据该半开连接的信息和原监听套接口的信息进行初始化，把该 sock 结构加入 tcp_eshash 哈希表，并继承原监听套接口的端口。然后将该代表该半开连接队列的 open_request 与该 sock 结构关联，接着把 open_request 从半开连接队列 (syn_table) 摘除，挂入已完成连接队列 (accept_queue)，并唤醒阻塞在 accept 系统调用上的进程。Accept 后半部分的处理如前所述，这样一个 TCP 连接建立起来。

■ 客户方

如前所述，客户通过 socket,connect 生成连接的 socket,sock 结构，向服务器发送一个 SYN 分节，sock 结构处于 TCP_SYN_SENT 状态，接着当前进程进入睡眠状态，同样，连接建立的余下过程由网络底层程序完成，其过程如图 5.4:

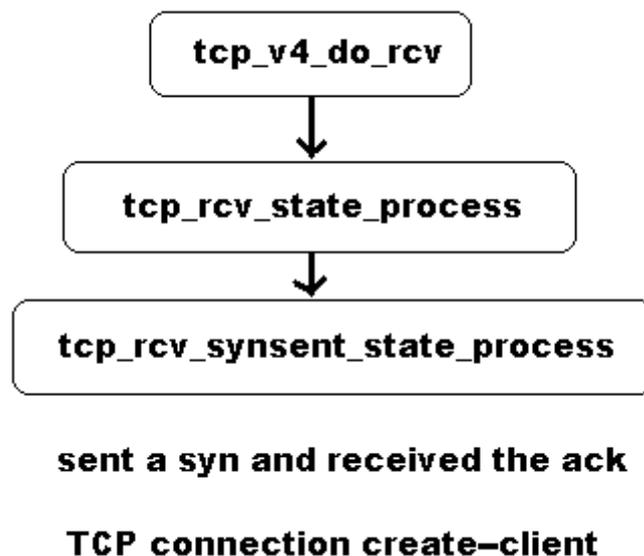


图 5. 4 TCP 连接建立过程-客户方

首先是 TCP 总的报文接收函数: tcp_v4_do_rcv，在对报文的格式进行一些合法性检查后，调用 tcp_rcv_state_process，该函数根据 sock 结构的状态进行散转，如果 sock 结构处于 TCP_SYN_SENT 状态，则调用 tcp_rcv_synsent_state_process，它首先检查 TCP 报头的 ACK 是否置上，接着查看 ACK 的序列号是否合法 (如果 SEG.ACK =< ISS 或者 SEG.ACK

> SND.NXT,发送 RST 分节复位连接)。接着查看,若置了 RST 位,则复位该连接。若设置 SYN 位,则丢弃该报文。接着根据 TCP 协议设置 sock 结构一些域的值,将 sock 结构的状态置为 TCP_ESTABLISHED,并向连接的对方发送一个 ACK 分节,这样该连接便建立起来。如果收到的报文只有 SYN 位,没有 ACK 位,则为双方同时打开的情况,置 sock 结构的状态为 TCP_SYN_RECV,并向连接的对方发一个 SYN_ACK 分节,这样,等到下一个 SYN_ACK 分节收到的时候,连接也建立起来。

5.1.3 网络缓冲区的基本操作

网络协议栈是一个有层次的软件结构,层与层之间通过预定的接口传递网络报文。网络报文中包含了在协议栈各层使用到的各种信息,将在各层间传递。网络报文的长度是不固定的,因此采用什么样的数据结构来存储这些网络报文就显得非常重要。在 BSD 的实现中,采用的数据结构是 mbuf,它所能存储的数据的长度是固定的,如果一个网络报文需要多个 mbuf,这些 mbuf 链接成一个链表。所以同一个网络报文里的数据在内存中的存储可能是不连续的。在 Linux 的实现中,每个网络报文都有一个控制结构,叫做 sk_buff。在 Linux 的协议栈实现中,一般情况下只分配一个网络报文的存储空间,只要不修改网络报文的内容,不同层或不同的处理函数都是通过控制结构 sk_buff 来共享这个网络报文的。只有在需要修改此报文的情况下,才复制一份。这样既节约了存储空间也方便了数据的定位,使得 Linux 的网络协议栈的性能在应用中表现良好。

下面首先讲述在 Linux2.2 中的 sk_buff 的定义和基本操作。sk_buff 是一个控制结构,通过它,才可以访问网络报文里的各种数据。所以在分配网络报文存储空间时,同时也分配它的控制结构 sk_buff。在这个控制结构里,有指向网络报文的指针,也有描述网络报文的变量。同时一个网络报文可以对应多个控制结构,其中只有一个是原始的结构,其他的都是 clone 出来的。Sk_buff 中有一个变量 cloned 表示当前这个 sk_buff 是否是 clone 的,另外通过引用数来记录该 sk_buff 共有多少个拷贝。由于可能存在多个控制结构,所以在释放网络报文时要确定它所有的控制结构都已被释放。

网络报文的存储空间是在网络设备收到网络报文或者应用程序发送数据时分配的,分配的空间以 16 字节对齐。分配成功之后,将网络报文填充到这个存储空间中去。填充时先在存储空间的头部预留了一定数量的空隙,然后将网络报文放到剩余的空间中去。但是网络报文不一定填满整个存储空间,有可能在存储空间的后部还有一定数量的空隙,所以 sk_buff 里面的 head 指针指向存储空间的起始地址, end 指针指向存储空间的结束地址, data 指针指向网络报文的起始地址, tail 指针指向网络报文的结束地址。网络报文在存储空间里的存放的顺序依次是:链路层的头部,网络层的头部,传输层的头部,传输层的数据。在协议栈的不同层, sk_buff 的指针 data 指向这一层的网络报文的头部。同时,在 sk_buff 里,也有相关的数据结构来表示不同层头部信息。sk_buff 和网络报文之间的关系如图 5.5:

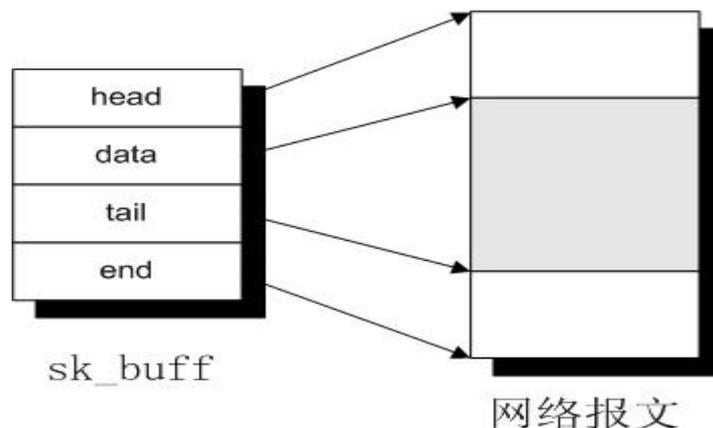


图 5. 5 sk_buff 与网络报文之间的关系

(注：控制结构 sk_buff 和网络报文的存储空间是从两个不同的缓存中分配的，所以它们在内存中不是连续存放的。)

与 sk_buff 相关的函数涉及到网络报文存储结构和控制结构的分配、复制、释放，以及控制结构里的各指针的操作，还有各种标志的检查。重要的函数说明如下：

- struct sk_buff *alloc_skb(unsigned int size,int gfp_mask)

分配大小为 size 的存储空间存放网络报文，同时分配它的控制结构。size 的值是 16 字节对齐的，gfp_mask 是内存分配的优先级。常见的内存分配优先级有 GFP_ATOMIC，代表分配过程不能被中断，一般用于中断上下文中分配内存；GFP_KERNEL，代表分配过程可以被中断，相应的分配请求被放到等待队列中。分配成功之后，因为还没有存放具体的网络报文，所以 sk_buff 的 data，tail 指针都指向存储空间的起始地址，len 的大小为 0，cloned 的值是 0。

- struct sk_buff *skb_clone(struct sk_buff *skb, int gfp_mask)

从控制结构 skb 中 clone 出一个新的控制结构，它们都指向同一个网络报文。clone 成功之后，将新的控制结构和原来的控制结构的 cloned 标记都置位。同时还增加网络报文的引用计数（这个引用计数存放在存储空间的结束地址的内存中，由函数 atomic_t *skb_datarefp(struct sk_buff *skb)访问，引用计数记录了这个存储空间有多少个控制结构）。由于存在多个控制结构指向同一个存储空间的情况，所以在修改存储空间里面的内容时，先要确定这个存储空间的引用计数为 1，或者用下面的拷贝函数复制一个新的存储空间，然后才可以修改它里面的内容。

- struct sk_buff *skb_copy(struct sk_buff *skb, int gfp_mask)

复制控制结构 skb 和它所指的存储空间的内容。复制成功之后，新的控制结构和存储空间与原来的控制结构和存储空间相对独立。所以新的控制结构里的 cloned 标记为 0，而且新的存储空间的引用计数是 1。

- void kfree_skb(struct sk_buff *skb)

释放控制结构 skb 和它所指的存储空间。由于一个存储空间可以有多个控制结构，所以只有在存储空间的引用计数为 1 的情况下才释放存储空间，一般情况下，只释放控制结构 skb。

- unsigned char *skb_put(struct sk_buff *skb, unsigned int len)

将 tail 指针下移，并增加 skb 的 len 值。data 和 tail 之间的空间就是可以存放网络报

文的空间。这个操作增加了可以存储网络报文的空间，但是增加不能使 tail 的值大于 end 的值，skb 的 len 值大于 truesize 的值。

- `unsigned char *skb_push(struct sk_buff *skb, unsigned int len)`

将 data 指针上移，并增加 skb 的 len 值。这个操作在存储空间的头部增加了一段可以存储网络报文的空间，上一个操作在存储空间的尾部增加了一段可以存储网络报文的空间。但是增加不能使 data 的值小于 head 的值，skb 的 len 值大于 truesize 的值。

- `unsigned char *skb_pull(struct sk_buff *skb, unsigned int len)`

将 data 指针下移，并减小 skb 的 len 值。使 data 指针指向下一层网络报文的头部。

- `void skb_reserve(struct sk_buff *skb, unsigned int len)`

将 data 指针和 tail 指针同时下移。这个操作在存储空间的头部预留 len 长度的空隙。

- `void skb_trim(struct sk_buff *skb, unsigned int len)`

将网络报文的长度缩减到 len。这个操作丢弃了网络报文尾部的填充值。

- `int skb_cloned(struct sk_buff *skb)`

判断 skb 是否是一个 clone 的控制结构。如果是 clone 的，它的 cloned 标记是 1，而且它指向的存储空间的引用计数大于 1。

在网络协议栈的实现中，有时需要把许多网络报文放到一个队列中做异步处理。Linux 为此定义了相关的数据结构 `sk_buff_head`。这是一个双向链表的头，它把 `sk_buff` 链接成一个双向链表，如图 5.6:

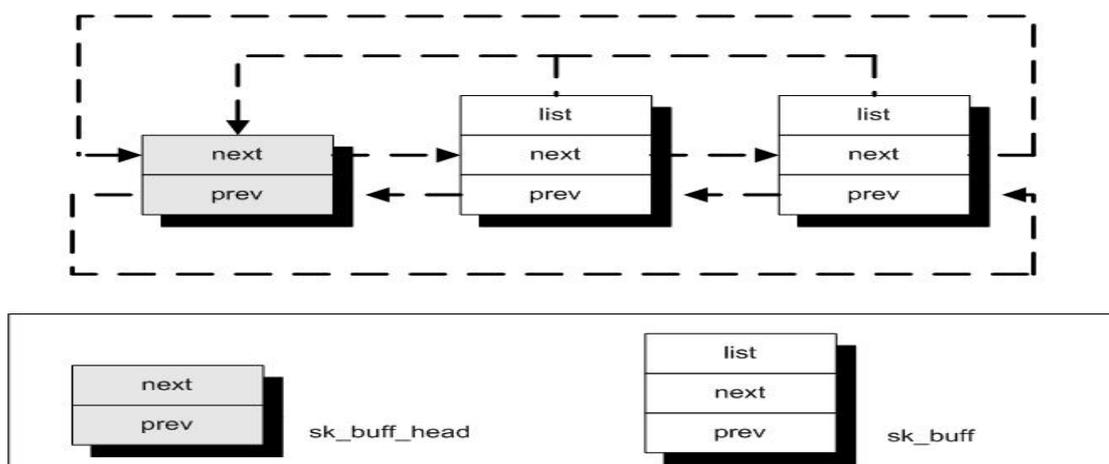


图 5. 6 `sk_buff_head` 与 `sk_buff` 的关系

与链表相关的函数，其功能无非是添加，删除链表上的节点，重要的函数说明如下：

- `void skb_queue_head(struct sk_buff_head *list, struct sk_buff *newsk)`

将 `newsk` 加到链表 `list` 的头部。

- `void skb_queue_tail(struct sk_buff_head *list, struct sk_buff *newsk)`

将 `newsk` 加到链表 `list` 的尾部。

- `struct sk_buff *skb_dequeue(struct sk_buff_head *list)`

从链表 `list` 的头部取下一个 `sk_buff`。

- `struct sk_buff *skb_dequeue_tail(struct sk_buff_head *list)`

从链表 `list` 的尾部取下一个 `sk_buff`。

- `skb_insert(struct sk_buff *old, struct sk_buff *newsk)`

将 newsk 加到 old 所在的链表上，并且 newsk 在 old 的前面。

- void skb_append(struct sk_buff *old, struct sk_buff *newsk)
将 newsk 加到 old 所在的链表上，并且 newsk 在 old 的后面。
- void skb_unlink(struct sk_buff *skb)
将 skb 从它所在的链表上取下。

以上的链表操作都是先关中断的。这在中断上下文中是不需要的，所以另外有一套与上面函数同名但是有前缀“__”的函数供运行在中断上下文中的函数调用。

Linux 2.4 中的网络报文在内存中不一定是连续存储的，同一个网络报文有可能被分成几片存放在内存的不同位置，这一点与 Linux 2.2 不同。一个大概的示意图如图 5.7:

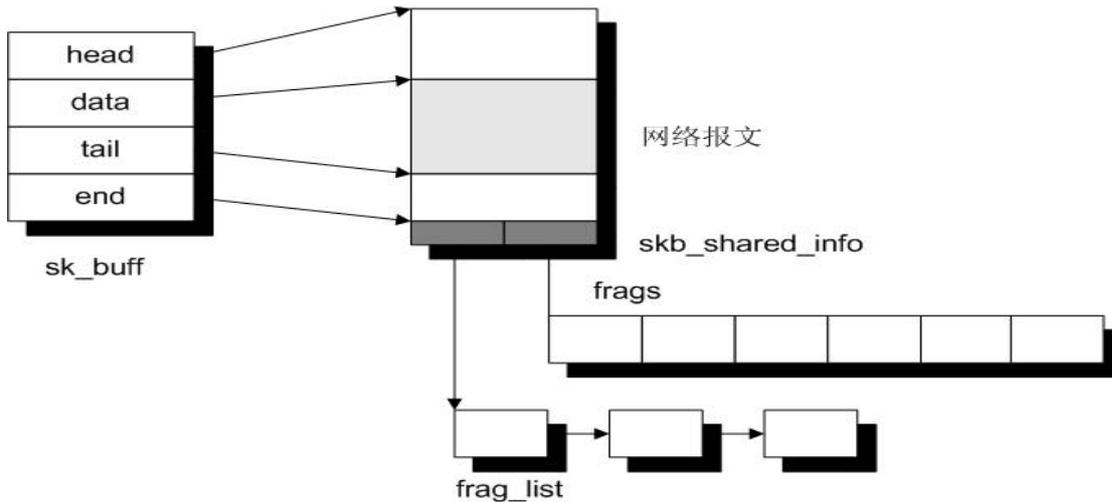


图 5. 7 LINUX 2.4 的 sk_buff 与网络报文之间的关系

图中的 frags 是一个数组，frag_list 是一个单向链表。它们所指向的存储空间是一个页的大小（即 4k）。这些额外的存储空间并不是一开始就使用的，只有在 data 所指的存储空间不够用的情况下才使用这些存储空间。以页为单位划分的存储空间有利于和用户空间的程序共享这一块内存的数据。

为了记录网络报文的长度，在 sk_buff 里增加了一个变量 data_len。这个变量记录的是在 frags 和 frag_list 里面存储的报文的长度。原有的变量 len 记录网络报文的总长度。truesize 是 head 所指的存储区的大小。

Linux 2.2 里分配，复制，释放 sk_buff 以及存储区的函数在 Linux 2.4 中的涵义没有变化，只是在操作时增加了对 frags 和 frag_list 的分配，复制和释放，并且在需要的时候将分散存储的网络报文整合成一个连续存储的网络报文。重要的函数说明如下：

- int skb_is_nonlinear(const struct sk_buff *skb)
判断 skb 是否是非线性化的，即报文的内容是否是非连续存放的
- int skb_linearize(struct sk_buff *skb, int gfp_mask)
将 skb 线性化，即重新分配报文的存储空间，使之在内存中连续

Linux 2.4 中对 sk_buff_head 的操作与 Linux 2.2 基本相同，只是多加了一个 spinlock 使队列可以在 SMP 的机器上更好地共享。具体地例子可以参考源代码，在此不做赘述。

§ 5.2 技术实现概述

在基于 TCP 迁移的集群系统中，由于客户方已经与 FE 建立了连接，不可能要求客户方再重新与 BE 进行三次握手建立相同的连接，这就需要考虑如何在 BE 上生成与 FE 上相同的连接数据结构而不经三次握手。以前，国际上讨论的一些 TCP Handoff 实现方法，都是采用“伪装三次握手”的技术来解决该问题。即在协议栈中与 TCP 逻辑上平行的层次添加一个模块，它负责“伪装”成客户，与 TCP 协议栈进行三次握手，即它产生与客户方最初发送给 FE 的相同的 SYN, ACK 报文，依次发送给 TCP 协议栈。实际上，该方法开销是比较大的，而且是不必要的。在第三章中，提出了基于 Netfilter 实现 TCP Handoff 的思想，这一思想的优点是通用性好，但性能不够高。本章提出的技术思想是：“一步到位”，即直接重构整个连接现场，包括 sock 结构，open_request 结构，然后将其挂入服务进程的 accept_queue 队列，将其唤醒。实际上，是取代了协议栈的功能。而用户进程并不知道唤醒它的是谁，也不知道连接的相关数据结构是由谁生成的，它也并不需要关心这一点。

基于此思想的 TCP Handoff 的实现方法如图 5.8,主要的功能模块逻辑上可划分为三个：SHS (SH sender), PR(packet router), SHR(SH Receiver)。工作步骤如下：

1. 当 FE 决定迁移一个连接时，它通知 SHS，SHS 获取连接的有关信息，然后改写 HTTP 请求报文，添加上 Handoff 报文头，生成一个 Handoff 请求报文；
2. SHS 从持久连接池中选择一条空闲的与选定的 BE 预先建好的持久连接，通过该连接，将 Handoff 请求报文发送给 BE 上的 SHR；
3. BE 上的 SHR 收到 Handoff 请求后，根据 Handoff 报文头中的信息，重构连接现场，利用 HTTP 报文零拷贝传递技术将 HTTP 请求报文挂在重构的连接的接收队列上，利用连接传递技术将新生成的连接传递给应用层服务程序。应用层服务程序将处理该 HTTP 请求，并将响应直接发送给客户方；
4. SHR 构造 Handoff 响应报文，通过刚才的持久连接发送给 FE；
5. FE 上的 SHS 得到迁移成功的消息后，撤消 FE 上该连接的一切数据结构，并向 PR 通知该连接的四元组和迁移后的 BE 地址，以便 PR 将该连接的后继报文直接在 IP 层转发给 BE。

可见，整个 TCP Handoff 过程只有五步，FE 和 BE 间只有一次报文交换，其实现是比较高效的。并且整个过程对客户端，应用层服务程序均透明，因而无需修改客户端，应用层服务程序，有着良好的通用性。

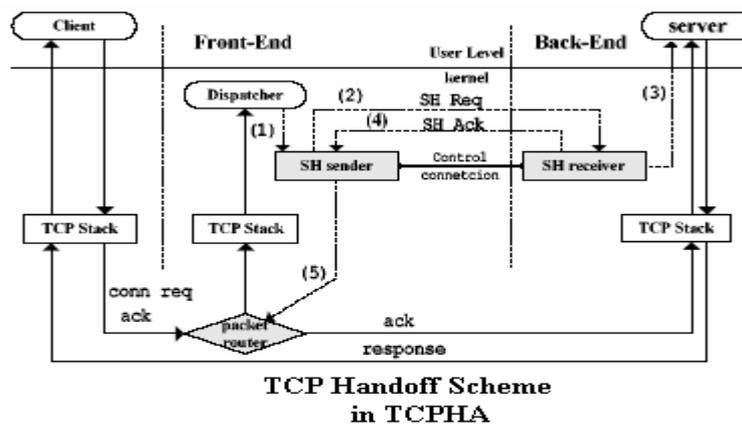


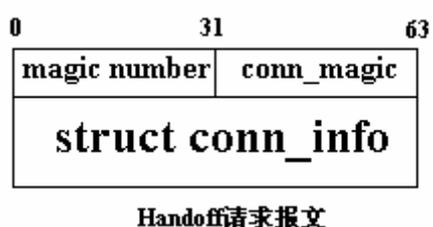
图 5. 8 基于连接现场重建的 TCP 迁移方法

§ 5.3 技术实现难点

5.3.1 Handoff 协议

FE 要将连接特征信息连同 HTTP 请求报文一起传递给 BE，BE 需要给 FE 确认，这需要一套协议。我们设计的协议称为 Handoff 协议，它的细节如下。

首先是 Handoff 请求报文，它将连接特征信息和 HTTP 请求报文一起传递。并基于 HTTP 请求报文进行修改。在报文头部空间中，即 TCP 报文头部和 TCP 负载（即 HTTP 请求）之间扩展出一个 Handoff 请求头部，Handoff 请求头部的格式如图 5.9 所示：首先是 32 位的 TCP Handoff 报文标识，取值为 0x12968b9。接着是 32 位的连接标识，取值为该连接上即将收到的对方的下一个 TCP 序列



号。然后是一个 conn_info 结构，包含所有用于 TCP Handoff 的必要信息，即连接的特征信息。这样，相当于在 TCP 协议和 HTTP 协议之间增加了一层应用层协议：TCP Handoff 协议。这样，Handoff 请求报文的结构为 Handoff 协议头部和 HTTP 请求报文。Handoff 请求报文由 FE 构造，发送给 BE。

图 5. 9 Handoff 请求报文格式

然后是 Handoff 响应报文，Handoff 响应报文的格式如图 5.10 所示：



magic number 域和 conn_magic 域的意义与 Handoff 请求报文中的意义相同。msg 域是一个枚举类型，用于标识 Handoff 操作成功与否。报文中的 conn_magic 域的值置为接收到的 Handoff 请求报文中该域的值，用来标识迁移的连接，以便 FE 确认是某一个 Handoff 请求的响应报文。Handoff 响应报文由 BE 在重建连接现场完毕后，发送给 FE 作为确认。

图 5. 10 Handoff 响应报文格式

可见，根据 Handoff 协议，FE 与 BE 之间只需一次报文交换，性能是较高的。

5.3.2 连接现场重构

研究典型的 Web 服务器编程模型可以发现，一个典型的 Web 服务器程序，总是处在一个循环之中：在一个监听套接字上等候客户方的连接请求（accept 系统调用），如果接到客户方的连接请求，就由操作系统协议栈按照 TCP 协议与客户方进行三次握手过程，建立好 TCP 连接后，返回给应用程序一个新的套接字描述符（accept 系统调用的返回值）标识这个新建立的连接。然后在新建立的套接字描述符上，与用户进行交互，为用户服务。服务完毕后关闭该套接字，继续在监听套接字上等待客户方新的连接请求。当然，根据编程

方式的不同，可以采用多进程或多线程结构将等待连接请求和服务请求并行起来。

从上面的分析可以看到，TCP 连接的建立过程实际是由 `accept` 系统调用完成的，并且该过程是由操作系统“包办”的，对用户程序是透明的。用户程序只是通过 `accept` 系统调用的返回值来引用新建立的 TCP 连接。这就启迪我们，要实现连接现场的重构，只需模拟 `accept` 的操作，构造连接数据结构，在系统相应的 Hash 表中登记。同时与 FE 合作，从 FE 处得到要迁移的连接的相关信息，如客户方的 IP 地址，端口等，根据这些信息修改刚才建立的连接数据结构。这样，一条连接就这样“一步到位”地构造出来了。

连接现场重构技术的主要步骤如下：

- (1) 分配记录连接信息的 `sock` 结构并初始化；
- (2) 查找到客户方的路由，将路由信息填入 `sock` 结构；
- (3) 根据 Handoff 协议从 Handoff 请求报文提取连接信息；
- (4) 根据提取得到的连接信息修改 `sock` 结构的一些域；
- (5) 将该 `sock` 结构登记到系统的相关表格中。

5.3.3 HTTP 请求报文零拷贝传递

在建好连接现场后，需要把 HTTP 请求报文挂在该连接的接收队列上，然后通过上一章讲述的连接传递技术（当然，这里这个连接现场是一步制造出来的，并不是源进程所拥有的连接，即连接的 `sock` 结构并没有与进程的数据结构建立关联，并没有连接到某个 `socket` 结构上，因而也就无需脱链操作）将该连接传递给用户层服务程序。

如前所述，在 Linux 的实现中，每个报文在拷贝到用户缓冲区之前，在操作系统网络协议栈各层间传递时，都放在一个缓冲区中，且由一个 `sk_buff` 结构作为其控制结构。各层通过 `sk_buff` 结构来读取自己这一层的报文内容，而不去管上层报文的内容。又由于我们的系统运行在操作系统内核空间，可以直接访问内核的一些相关数据结构，包括报文的 `sk_buff` 结构，这样，我们就可以通过修改 `sk_buff` 控制结构，来修改报文的属主，让报文从一条连接的报文变为另一条连接的报文，而报文本身的缓冲区不需做任何操作。这样就可以实现将 HTTP 请求报文零拷贝地交给用户层服务程序。

HTTP 请求报文零拷贝传递技术的主要步骤如下：

- (1) 将 `skb` 的 `data` 指针后移，跳过 `handoff` 请求报文头；
- (2) 修改 `copied_seq` 域，标识该报文已由协议栈读取；
- (3) 丢弃报文原来的路由；
- (4) 将报文的属主改为新的连接；
- (5) 将报文加在新的连接的接收队列的尾部。

这样，通过对 `sk_buff` 结构的一些操作，使报文对上层应用程序看来为客户方最早发送给 FE 的 HTTP 请求报文，并且报文的属主由 BE 与 FE 的连接变为了构造出的 BE 与客

户方的连接。

对于该方法的测试，测试的设备和方法和第三章中介绍的相同，详见 3.3 节。

§ 5.4 本章小结

本章主要讲述了基于连接现场重构的 TCP 迁移方法，该方法有以下优点：

- (1) 连接现场是“一步到位”建立起来的，效率很高；
- (2) 对客户方，服务器应用程序均透明；
- (3) 每次 FE 与 BE 间只有一次报文交换，而且通信量较小。

该技术的缺点：

- (1) 该技术与操作系统协议栈实现有关，因而可能需要随着操作系统的修改而做少量修改。

第六章 对持久连接（P-HTTP）的支持

HTTP/1.1 中提出了持久连接（P-HTTP）的概念。本章首先讲述 P-HTTP 对基于 TCP 迁移的集群调度系统的影响，然后提出一种新的算法：多重跳动算法（multi-handoff），用于在基于 TCP 迁移的集群调度系统中有效的支持 P-HTTP。

§ 6.1 概述

6.1.1 提出背景

几乎所有已知的 HTTP 实现都使用 TCP 作为传输协议。然而，对于 HTTP 消息交换中常见的短时连接，TCP 却没有得到充分的优化。使用 TCP 作为传输协议，要求采用三次握手的方式建立连接，并用 4 个数据包关闭连接。在 HTTP/1.0 及以前的版本中，浏览器在每个 TCP 连接中向服务器只发出一个请求。对于每个请求，HTTP 消息的大小通常在 10 个数据包之内。因而，在一次 HTTP 会话的 17 个数据包中，有 7 个都属于附加开销。这也意味着 Web 传输总是不能跨越 TCP 的慢启动阶段。在 TCP 的数据窗口能明显增大以前，连接便已关闭了，这意味着网络带宽不能得到充分的利用。

随着 Web 的日益流行，Web 页面中常常含有嵌入图像。为了下载这样的组合文档，需要几个 HTTP 事务处理，也即要建立多个 TCP 连接。在显示完整的文档之前，由于需要依次建立每一个串行的连接，所以在用户那里会感觉到明显的延迟。以前十分流行的浏览器（Mosaic）采用的便是这样的实现方法。为缩短延迟，一种方法是引入并行的 HTTP 连接。这一技术首先由 Netscape 投入使用，最多可同时打开 4 个并行连接以下载图像，因而加快了整个文档下载的速度。然而，这同时也加重了整个网络的拥塞，服务器不得不付出额外的开销来接受多个 TCP 连接。

6.1.2 持久连接协议（P-HTTP）概述

解决这个问题的另一个办法是让建立起来的 TCP 连接能超越单个请求-响应交换，仍保持其开放状态。因此，HTTP/1.1 以上版本提供了对持久连接的支持。持久连接最基本的思想就是减少 TCP 连接打开和关闭的次数。HTTP/1.1 协议使得浏览器在一个 TCP 连接中向服务器发多个请求。服务器完成一个请求后，保持连接时间一般为 15 秒，这个时间是可设置的。客户方接到响应后，并不关闭连接，而是通过该连接发送第二个请求。服务器同样利用该连接来返回响应数据。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，减轻了网络和服务器的负载，缓解了拥塞情况。同一连接上，后续的响应也能从减轻的拥塞中受益，也充分利用了网络带宽。并且，由于后续请求没有连接的建立和关闭以及 TCP 慢启动

带来的延迟，客户所感知的延迟也明显缩短了，服务质量得以提高。

6.1.3 持久连接带来的问题

在基于 TCP 迁移的系统中，持久连接将带来问题。设想 FE 与客户方建立了一条 TCP 连接，在收到客户方的 HTTP 请求后，FE 解析该请求，调度到一台 BE，设为 BE₁，然后启动连接迁移过程，将该连接迁移到 BE₁ 上。BE₁ 服务该请求，将响应直接发送给客户。客户并不关闭连接，而是发送第二个请求，按照以前的实现技术，该请求将被 FE 直接转发给 BE₁，以后的请求均是如此，直至此次 HTTP 会话结束。这样，实际上持久连接上只有第一个请求是按内容调度的。其它请求在调度时均没有考虑请求内容。这样，基于请求内容的调度系统与四层调度系统相比，已经没有什么优势。而且，还有 TCP 迁移和解析报文内容的附加开销。BE 之间也必须是对称的，不能将服务资源在 BE 上分类存放，对于提高 BE 上的 Cache 命中率，也没有明显的优势。

6.1.4 国际上的解决办法

最简单的方法是让 FE 重定向客户方浏览器到调度到的 BE，这可以简单地通过发送一个 Redirect 响应来实现或通过返回一个 Java Applet，在客户方执行时与特定的 BE 通信来实现。但这些方法有较大的缺陷，重定向带来较大的附加延迟，整个集群的组成都暴露给了客户方，这可能带来安全问题。另外，早期的浏览器可能不支持重定向。因此，性能较高，通用性较好的实现方式应是对客户方透明的。

满足该要求的实现方法有：

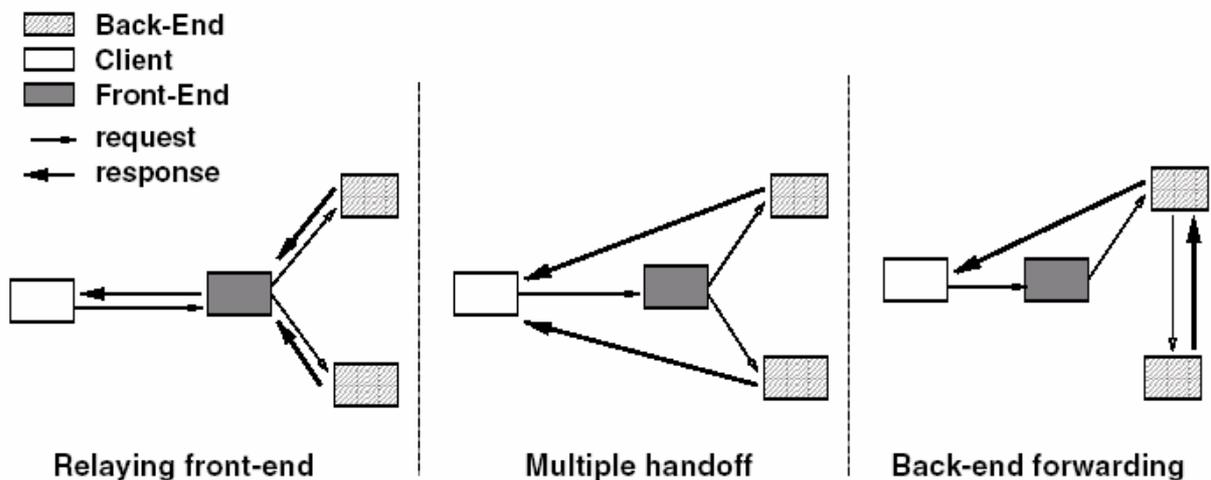


图 6. 1 支持 P-HTTP 的集群调度技术

(1) Relaying FE

该方法如图6.1所示，FE维护所有与BE的持久连接。在一个与客户方的连接上，当一个请求到达时，FE分发这个请求，将这个请求转发给调度到的BE。当响应从BE返回时，FE将响应转发给客户，必要时，FE可以在Cache中缓存响应数据或通过TCP粘合技术将两条连接粘合起来以提高性能。这种方法最大的优势就是简单，并且对客户方和BE均透明。该方法最大的缺陷是所有的响应数据必须由FE转发。这使得FE很容易成为系统的瓶颈，而且也增加了响应的延迟。目前，使用该方法的系统有KNITS^[30]，KTCVPS^[52]。

(2) BE request forwarding

该方法如图6.1所示，当持久连接上的第一个请求到达时，FE将该连接迁移到调度到的BE上。当持久连接上的后续请求到达，由FE调度，如果发现调度到的不是刚才选中的BE，设为A。则FE通知A，A向B发出请求，并将响应直接发给客户方。该方法的缺陷是增加了BE之间的通信，消耗了BE之间的网络带宽，响应延迟也增加了。目前，使用该方法的系统有LARD^[1]。

(3) Multiple TCP connection handoff

该方法如图6.1所示，FE将连接在BE之间迁移，所有的响应数据均不通过前端而直接返回给客户方。该方法的优势是性能很好，但实现较为复杂。目前Dept. of Computer Sciences University of Texas和Austin Research Lab IBM的Ravi Kokku, Ramakrishnan Rajamony, Lorenzo Alvisi, Harrick Vin等人提出了一种实现方法，称为Half-pipe Anchoring^[2]。他们的思想是把TCP连接看成由两条半双工的管道组成，一条是数据管道，一条是控制管道。从集群到客户的是数据管道，从客户到集群的是控制管道。他们的方法是把控制管道固定在一个固定的BE上，而让数据管道在BE间迁移。他们通过自己设计的称为split-stack的通信协议来使两个管道处在不同的节点上。该方法的缺点是维护控制管道和数据管道之间的一致性很复杂，开销很大。

§ 6.2 技术思想概述

我实现的方法是基于 Multiple TCP connection handoff，我称之为多重跳动算法 (multi-handoff)。其具体思想是：让 BE 也参与调度，这样可以避免 FE 的性能瓶颈问题。即 FE，BE 均参与调度，其中 FE 只调度连接上的第一个请求，并迁移连接到选中的 BE。如果该连接上有后续请求报文，则由该连接迁移到的目的 BE 来调度。这时 BE 的功能与 FE 完全相同，它解析新的 HTTP 请求，按调度规则进行匹配，如果调度到自己，则直接处理。如果调度到的是另一台 BE，则它发起一个迁移过程，将该连接迁移到新的 BE 上，然后重置连接，通知 FE 转发的目的地址改变。这要求 BE 也知道整个集群系统的构成。如果连接在 BE 之间存在多个镜象，那么这些镜象之间的状态同步将是非常复杂的，而且可能对客户方带来某些不可预测的影响，例如某个镜象关闭连接。因此，方法的一个关键是保证连接在 BE 之间只有一个镜象。

§ 6.3 实现算法

基本假设：

FE：集群前端机

BE：集群后端机，BE_A：后端机 A,依次类推

6.3.1 一次迁移过程

1. 客户方 C 与 FE 上一个监听的内核线程 T 建立一条 TCP 持久连接，T 生成一个关

于该连接的控制结构。客户方向 FE 发出第一个 HTTP 请求，该请求由 FE 上的内核线程 T 接收，T 将该连接的状态置为 HANDOFFING，并将该连接纪录到 Hash 表中。由于系统启动时钩挂了 NF_IP_LOCAL_IN,NF_IP_FORWARD 两条链，底层 BH 程序通过 Hash 表就可以对该连接上的报文依据状态进行处理。对于 HANDOFFING 状态，BH 将丢弃所有该连接上的输入输出报文，这主要是为了保证 TCP 迁移过程的原子性，即在迁移过程中，该 TCP 连接的状态不能改变。T 解析请求内容，进行调度，假设调度到 BE_A,T 从该连接的 sock 结构得到连接信息，将其和 HTTP 请求报文一起构造 handoff 请求报文通过 FE 和 BE_A 间的持久连接发给 BE_A。

2. BE_A 收到连接信息和 HTTP 请求后，重构连接现场，并将该连接转交给用户层的 HTTP 服务器，向 FE 发出迁移完毕消息，FE 上的线程 T 接到该消息后，将 BE_A 的地址记录在该连接的控制结构中，将连接的状态置为 HANDOFFED。处于该状态，BH 对输入报文一律通过 IP 隧道将报文转发给 BE_A，输出报文则 STOLEN。然后，FE 重置(reset)该连接。

3.FE 上的输入报文钩子发现该连接上客户方的 FIN 或 RST 报文，则将 Hash 表中连接状态置为 TIME_WAIT。一定时间没有收到该连接的报文后，自动从 Hash 表中撤消连接的控制结构。

容错：

1. 在迁移过程中，若 FE 给 BE_A 的报文丢失或 BE_A 不可达，则在一定时间内，FE 将收不到 BE 的响应，超时后，FE 关闭与客户方的连接，并撤消 Hash 表中的连接控制结构。
2. 若 BE_A 的响应丢失，处理方法同上。

6.3.2 多次迁移过程

前面的工作过程同上。此外，BE 也使用 Hash 表记录迁移到它的连接并且也具有调度能力。系统启动时也钩挂 NF_IP_LOCAL_IN,NF_IP_FORWARD 两条链，以便底层 BH 程序通过 Hash 表对连接上的报文依据状态进行处理。初始时，即刚迁移完毕时，BE_A 上记录的连接的状态为 ESTABLISHED。对于该状态，BH 的处理算法为：接收一切 TCP 控制报文（即数据为 0 的报文），如果报文中置了 FIN 或 RST，则将连接状态改为 TIME_WAIT。如果 TCP 报文数据不为 0，则认为是持久连接上的新 HTTP 请求，BH 对该报文进行解析，调度，如果调度到的是自己，则直接将该报文接收。否则，假设调度到 BE_C。则 BH 该报文交给 BE_A 上的一个内核线程 T'，将连接的状态置为 WILLHANDOFF，返回 NF_STOLEN。对于 WILLHANDOFF 状态，BH 接收一切 TCP 控制报文（即数据为 0 的报文），如果报文中置了 FIN 或 RST，则将连接状态改为 TIME_WAIT。如果 TCP 报文数据不为 0，则将其丢弃。这主要是为了保证在重新迁移之前，不再接收新的 HTTP 请求。内核线程 T' 收到该 HTTP 请求报文后，连接新调度到的 BE_C（当然可以考虑将 BE 间也建立持久连接），并在协议栈中查找该连接的 sock 结构，如果该 sock 结构指示连接的状态为 TCP_ESTABLISHED，并且 Hash 表中该连接的控制结构指示连接状态为 WILLHANDOFF，则 T' 将该连接的状态改为 HANDOFFING。对于该状态，BH 丢弃所有该连接上的输入输出报文，这样，BE_A 就和前面的 FE 工作过程相同，获得连接信息，连同 HTTP 请求一起发送给 BE_C，BE_C 执行连接重构，并将其传递给 BE_C 上的 HTTP 服务器，然后通知 BE_A 迁移成功。BE_A 收到 BE_C 的确认后，重置该连接（reset），将该连接状态置为 HANDOFFED。对于该状态，BH 将所有该连接上的输入报文通过 IP 隧道转发给 BE_C，输出报文则 STOLEN。但该状态有一定的时间限制，时间到了以后，BH 从 Hash 表中删除该连接。接着，BE_A 上的 T' 与 FE 建

立连接, 通知它将转发地址变为 BE_C , 得到 FE 的确认消息后, BE_A 从 Hash 表中删除连接。

算法的关键: 让连接安全地在 BE 间迁移的关键就是必须保证在 $HTTP$ 响应的 ACK 收到后再执行迁移。例如, 客户方请求 `index.htm`, FE 将连接迁移到 BE_A , BE_A 响应该请求, 向客户发送该文件, 客户方收到后, 响应以 ACK 。假设在该 ACK 没收到之前, BE_A 就将连接迁移到了 BE_C (根据 $HTTP/1.1$, 这是完全可能的, 例如, 客户方同时发出两个 $HTTP$ 请求), 这样 BE_C 会发现需要重传 `index.htm`, 但如果文件在 BE 之间分类存放的话, BE_C 上根本没有该文件。这还不考虑迁移后重传缓冲区的指针处理问题。事实上, 关键也就是让 BE_C 看到的是一个新的 $HTTP$ 请求需要服务, 而不是需要重传某个缓冲区的内容。当然, 这个问题也可以通过完全重构整个连接现场来解决, 例如, 将重传缓冲区的内容也传递给 BE_C 。然后根据 BE_C 的情况对指针重新赋值。还有, 如果 BE_A 的响应丢失, 客户重传的是对 `index.htm` 的请求, 这种情况在当前这个算法应该不会有问题, 因为这会导致调度到 BE_A 自己, 然后 BH 直接将报文接收。在算法中, 为了保证这一点, 在 BE 上设置了 $WILLHANDOFF$ 状态, 是因为从 BH 将报文交给内核线程 T' , 到 T' 真正决定迁移, 这中间肯定有一定的延时。在这段时间中, 可能连接已经关闭了, 这就是迁移前检查 `sock` 结构的状态和 Hash 表中该连接状态的原因。另一个原因, 在该状态只是丢弃新的 $HTTP$ 请求, 但接收所有控制报文, 这也是尽量使得迁移前, BE_A 收到 ACK 。同样, 这也是设置 $HANDOFFED$ 状态的原因, 就是尽可能的将 ACK 送到 BE_A 。 BE_A 在得到 FE 的确认后, 马上从 Hash 表中删除该连接的原因是, 有可能很快 BE_C 又将连接迁移回 BE_A , 这时, BE_A 上 Hash 表中该连接的存在可能对处理造成影响。

§ 6.4 BE 调度技术

BE 调度技术是实现多重跳动技术的关键。要让 BE 参与调度, 与 FE 调度相比, 有一个技术难题: FE 在调度时, 因为与客户方建立了连接, $HTTP$ 请求报文可以直接由 FE 上的内核线程收到。而 BE 在调度时, 已经将构造出的连接通过连接传递技术交给了用户层服务程序, 这样连接上的新的 $HTTP$ 请求将直接交给用户层服务程序, 而 BE 上的内核线程收不到。就是说, 连接的属主是用户层 Web 服务程序, 而不是 BE 上的系统内核线程。这里, 是通过 $Netfilter$ 框架的支持来解决这个问题的。即在 BH 层, 截获连接上的报文, 判断是否是 $HTTP$ 请求报文的算法是: 判断该报文的 TCP 数据是否为 0, 如果不为 0, 则认为是 $HTTP$ 请求报文。如果判断不是 $HTTP$ 请求报文, 则交给协议栈正常处理。若是 $HTTP$ 请求报文, 解析该报文, 注意, 这里因为是在 IP 层。所以还需要解析 TCP 层的内容。然后, 解析 $HTTP$ 协议层的内容, 匹配调度规则, 如果调度到一台不同的 BE , 则接管该报文, 不再向协议栈上层传递。将调度到的 BE 地址和端口记录在该报文中, 将报文挂到一个异步操作队列中, 向上层内核线程发信号, 由上层内核线程对其进一步处理。

相关的步骤如下:

(1) 检查报文的合法性, 由于这里系统工作在 IP 层, 需要的检查工作有:

- 报文是否是发往本机的包, 若不是, 则交给协议栈正常处理;

- 报文是否是发给环回接口（loopback）的包，若是，交给协议栈正常处理；
- 上层协议是否为 TCP 协议，若不是，交给协议栈正常处理；
- 报文完整性检查。

(2) 根据 IP 报文头中的客户方地址和端口信息查找 Hash 表，得到该连接的相关数据结构；

(3) 得到连接状态；

(4) 根据连接状态进行处理：

- 若为 ESTABLISHED 状态：
 - I. 若为 FIN 或 RST 报文，将连接状态置为 TIME_WAIT；
 - II. 若为 TCP 控制报文，即 TCP 数据为零，则交给协议栈正常处理；
 - III. 否则，认为是 HTTP 请求报文，进行校验和检查，根据调度算法进行调度，如果调度到本机，交给协议栈正常处理；否则将报文从协议栈中脱链，将调度到的新的 BE 的地址和端口记录在报文中。将报文加在一个异步接收队列尾部，通知内核线程有报文需要处理。
- 若为 WILL_HANDOFF 状态：
 - I. 若为 FIN 或 RST 报文，将连接状态置为 TIME_WAIT；
 - II. 若 TCP 数据不为零，认为是新的 HTTP 请求报文，丢弃；
 - III. 否则，交给协议栈正常处理。
- 若为 HANDOFFING 状态：丢弃；
- 若为 HANDOFFED 状态，从协议栈中脱链，封装后转发给新调度到的 BE。

(5) 若内核线程得到通知，则检查异步缓冲队列，如果发现队列非空，则从中取下一个元素，它是一个 HTTP 请求报文。从中提取出调度到的 BE 的地址和端口，与之建立连接，然后将 HTTP 请求报文提取出来，在协议栈中查找要迁移的连接的数据结构，从中得到连接的特征信息，构造一个 handoff 请求报文，将其发送给新调度到的 BE。

(6) 从新调度到的 BE 获取响应，如果迁移成功，则将新的 BE 的地址和端口通知 FE。得到 FE 的响应后，重置该连接，从 Hash 表中删除该连接的对应表项。

§ 6.5 本章小结

本章讲述了持久连接（P-HTTP）给基于 TCP 迁移的集群调度系统带来的问题，并提出了解决方法。该方法有以下优点：

- (1) 采用分布式调度算法，让 BE 也参与调度，减轻了 FE 的性能瓶颈问题；
- (2) 保证了每个请求按内容调度，这样可以充分发挥基于内容调度的优势；
- (3) 保持了 TCP 迁移技术的优势。

第七章 大规模基于请求内容分发的系统 TCPHA

本章主要讲述基于重构连接现场 TCP 迁移方法的 Web 服务器集群调度系统 TCPHA 的实现。

§ 7.1 系统体系结构

TCPHA 的体系结构如图 7.1 所示：

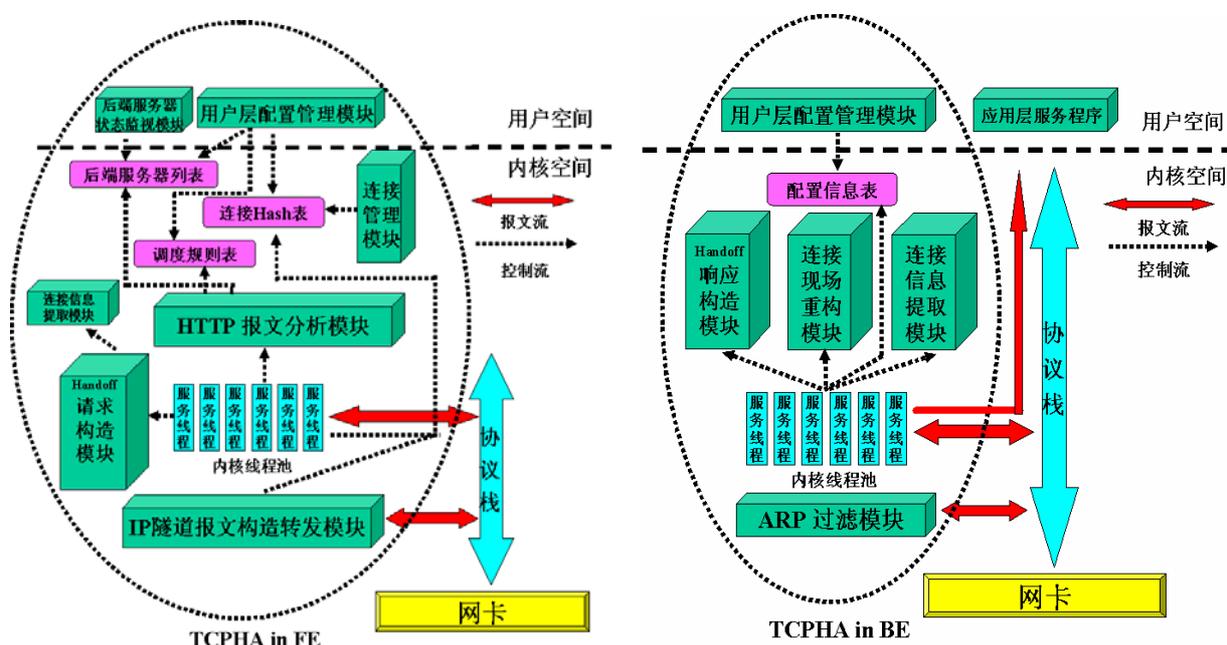


图 7. 1 TCPHA 系统体系结构

整个 TCPHA 系统分为在 FE 上安装的模块和 BE 上安装的模块两大部分。整个系统基本上在操作系统内核工作，并作为可动态加载的设备驱动程序模块实现，安装非常方便，无需修改操作系统内核。用户空间的模块只有两个：用户层配置管理模块和后端服务器状态监视模块。其中用户层配置管理模块用于向集群管理员提供配置该系统的接口；后端服务器状态监视模块则用于负载均衡及容错，它定期地探查后端服务器的运行情况，负载情况，并通过 `ioctl` 接口修改内核中的后端服务器列表信息。

§ 7.2 系统工作过程

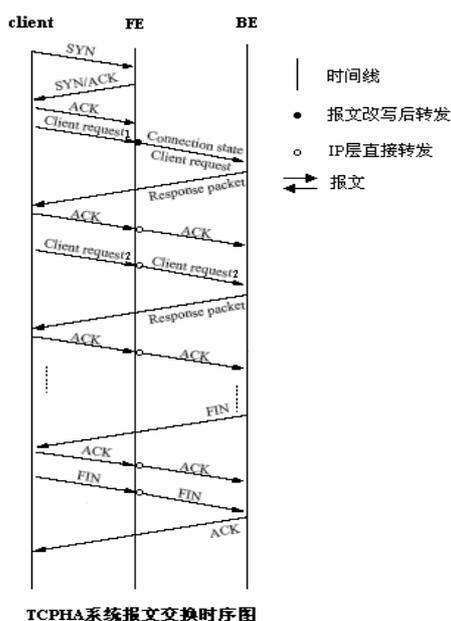
系统的工作过程如下：

客户方向 FE 发起一个 TCP 连接请求，FE 上的 TCPHA 从内核线程池选择一个空闲

的服务线程服务该请求，与客户方建立起一条 TCP 连接。然后，客户方发送 HTTP 请求。该请求由服务线程收到后，调用 HTTP 报文分析模块，根据 HTTP 协议解析 HTTP 请求报文内容，提取用于调度的有关信息，如 URL 等。然后查询调度规则表，进行规则匹配，从而调度到一台 BE。接着，查询后端服务器列表，得到该 BE 的具体信息，如 IP 地址，端口，处理能力，负载情况等。接着，调用 Handoff 请求构造模块，根据 handoff 协议构造 handoff 请求报文。Handoff 请求报文构造完成后，服务线程从持久连接池中，选择一条空闲的预先建好的与选中的 BE 的持久连接，通过该持久连接，将 Handoff 请求报文转发给 BE。

BE 上的服务线程通过持久连接接收到 Handoff 请求报文后，首先核对 magic number 域，确认为 TCP Handoff 报文。然后，调用连接信息提取模块，从报文中提取用于 TCP Handoff 的所有信息，并通过指针的移动，将报文恢复为对上层应用程序看来与客户方初始发送过来的 HTTP 请求报文完全相同的报文。然后，调用连接现场重构模块，重建连接现场。在重构连接后，调用 Handoff 响应报文构造模块，构造 Handoff 响应报文，该 Handoff 响应报文构造好后，通过刚才的持久连接发送给 FE。

FE 上的 TCPHA 系统收到该 Handoff 响应报文后，检查 magic number 域和 conn_magic 域，然后读取迁移操作消息，如果得到的是迁移成功消息，就在 FE 上“静悄悄的”撤消



该 TCP 连接。所谓“静悄悄的”撤消，就是不通过 TCP 协议规定的握手过程撤消连接，而是“单方面的”撤消，即直接撤消该连接在 FE 上的一切数据结构，该过程对客户方不可知。这就类似于连接的一方收到 RST 报文的情况或是因对方不可达，连接超时而撤消连接的情况。然后 FE 上的 TCPHA 将该连接的四元组信息以及迁移到的 BE 的地址记录到连接 Hash 表中，该连接 Hash 表可以被工作于 IP 层的 IP 隧道报文构造发送模块所读取。该模块会根据连接 Hash 表截获流经 FE 的协议栈的已迁移的连接上的后续报文，然后直接在 IP 层将报文转发给 BE。

TCPHA 系统的报文交换时序如图 7.2:

首先，客户方和 FE 通过正常的三次握手过程建立起一条 TCP 连接，如图中前三个报文序列。接着，客户方发出请求 1，该报文在 FE 被改写，然后转发给调

图 7. 2 TCPHA 系统报文交换时序图

度到的 BE。在 BE 上，连接被重建，响应报文不经过 FE 而直接发送给客户方。该连接上的后续报文由 FE 在 IP 层直接转发给 BE，BE 发出的响应报文则直接发送给客户方。

§ 7.3 系统实现的关键技术

7.3.1 服务器体系结构

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。在本小节

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 TCPHA 调度器中采用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 TCPHA 服务器程序。

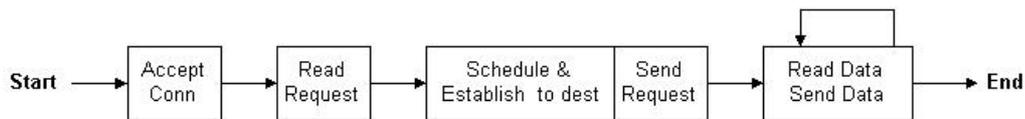


图 7. 3 TCPHA 调度器处理请求的简单步骤

图 7.3 说明 TCPHA 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, 从持久连接池中选择一条空闲的到目标服务器的持久连接。如果没有空闲的连接, 建立一个 TCP 连接到该服务器。
- 迁移连接: 通过该持久连接将与客户方的连接的信息传送到目标服务器, 目标服务器根据这些信息重建连接。
- 撤消连接, 登记连接: 得到目标服务器的迁移成功响应后, 重置与客户方的连接, 将该连接的四元组信息登记到 Hash 表中。

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, 接受连接和读入请求都会阻塞; 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; 接收目标服务器的响应会阻塞。所以, 我们必须设计良好的服务器结构, 将 CPU 的处理时间重叠起来, 提高调度器的性能。

■ 多进程结构

在多进程结构（Multiple-Process Architecture）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 7.4 所示。

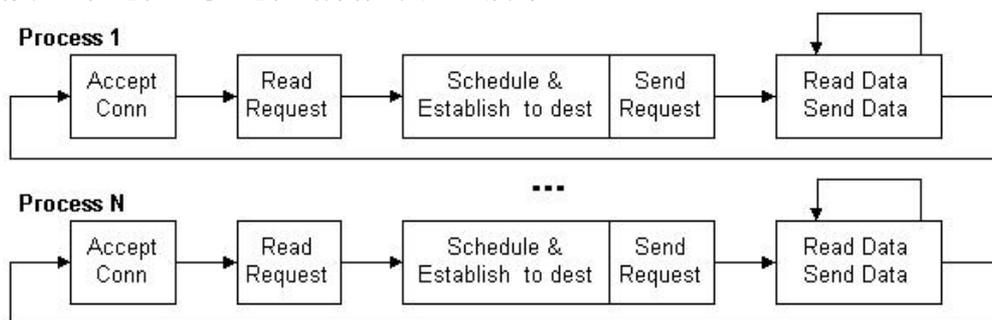


图 7. 4 多进程结构

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。

■ 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 7.5 所示。



图 7. 5 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 TCPHA 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。

■ 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 7.6 所示。

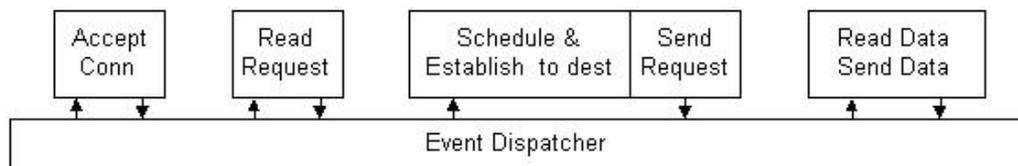


图 7. 6 单进程事件驱动结构

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 和 boa 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。

■ 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 7.7 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。

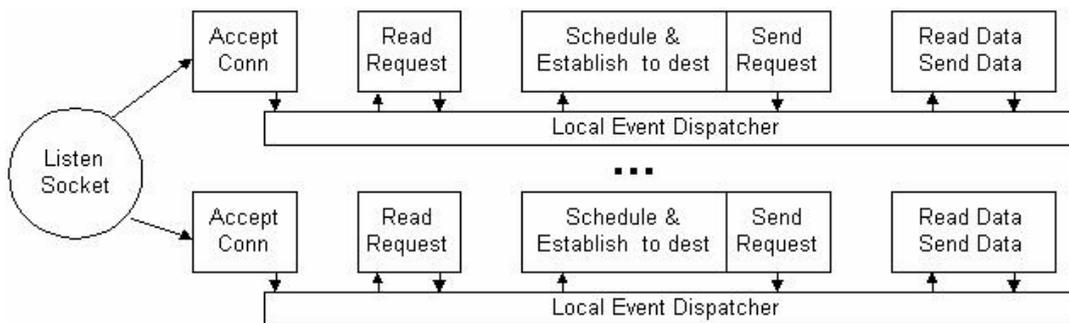


图 7. 7 多线程事件驱动结构

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，引入多个线程，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 个事件驱动线程，在双处理器的系统上启动 4~6 个事件驱动线程。

在 TCPHA 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。而且内核线程在 ring0 权限工作，使用的是内核内存空间，因此不会存在页面交换和系统调用引起的权限切换等的开销，具有较好的性能。

7.3.2 ARP 问题及其解决方案：ARP 过滤

正如前文所述，由于前端调度器和后端所有服务器有一个相同的 IP 地址，我们称之为 VIP 地址（Virtual IP）。当客户方想与 VIP 建立连接时，它会首先发送一个 ARP 请求报文，来得到 VIP 对应的 MAC 地址。我们的目标是让客户与前端调度器建立连接，这样，我们就必须控制后端服务器池中的服务器不要响应 ARP 请求，否则，如果一台后端服务器响应了 ARP 请求，客户方将直接与其建立连接，这样，前端调度器就被旁路(bypass)了，就无法起到调度的作用。还有更糟糕的事情，由于 ARP 的缓存，可能导致大量客户同时访问一台服务器，而别的都闲着，导致负载不均衡。或者，当客户正在与一台服务器进行 TCP

会话，这时接到了另一台服务器的 ARP 响应报文，这样，ARP 表项变了，客户的 TCP 报文转而被发向另一台服务器，当然，这只会接到另一台服务器发来的 RST。还有，在基于内容调度的服务器集群中，后端服务器存放的数据可能并不相同，这样，客户如果与之直接建立了连接，可能得不到请求的数据。因此，解决 ARP 问题是一个技术上的关键问题。在 Linux 2.0 版的内核以前，这个问题的解决很简单，因为某些设备本来就不响应 ARP 请求（称为 nonarp 设备，例如 tunnel0,dummy0,lo:0），这样，只要把 VIP 配置在这些设备上就可以了，但现在 Linux 内核已升级到 Linux 2.4，而从 Linux 2.2 开始，所有这些设备都要响应 ARP 请求。当前的系统，例如 IPVS，都是通过对内核打补丁，然后重新编译内核来解决 ARP 问题。这里，我利用内核 2.4 的支持，用另一种方法解决了 ARP 问题，我称其为 ARP 过滤。利用内核 2.4 提供的 Netfilter 框架支持，TCPHA 系统挂接 ARP 报文遍历的 ARP_IN 链和 ARP_OUT 链两处，用于截取/改写 ARP 报文。当一个 ARP 报文到达时，ARP 报文会转到 ARP_IN 链上。这样，当一个 ARP 报文到达时，该报文会被挂接在 ARP_IN 链上的 TCPHA 系统捕获，若该 ARP 报文是 ARP 请求报文，并且请求的目标 IP 地址是 VIP，TCPHA 将其丢弃，不再上传到操作系统协议栈继续处理。如果 ARP 报文的源地址是 VIP（即报文是前端调度器发来的），这里需要进行特殊的处理，因为后端服务器会认为该报文是自己发出的，从而直接由协议栈丢弃该报文。这样，前端调度器就无法得知后端服务器的存在，当然也就不能向其转发报文。同时，也为了不让后端服务器“知道”有主机的 IP 地址与自己的相同，以免引起可能的麻烦（例如，在 WINDOWS 系统下，会告知用户 IP 地址冲突，甚至禁用该 IP 地址），因此，将该报文的源 IP 地址改为一个不存在的地址 X。同样，当一个 ARP 报文送出时，该报文会被挂接在 ARP_OUT 链上的 TCPHA 系统捕获，如果该 ARP 报文的源地址为 VIP，则将其改为 IP 隧道设备的地址，如果报文的目标地址为 X，则将其改回前段调度器的 IP 地址 VIP。这样，既可以防止后端服务器“主动地”通知自己有一个地址为 VIP，也可以让前端调度器知道该后端服务器的存在。实际上，就是使 VIP 的存在在后端服务器上成为了一个“孤岛”，只有它自己才“知道”有这样一个地址。这样，就成功地解决了 ARP 问题。而且，由于 TCPHA 系统以动态加载的模块方式实现，无需重新编译内核。因此，只要卸载该模块，便取消 ARP 过滤，不影响协议栈原来的行为。

7.3.3 动态 IP 隧道技术

IP 隧道（IP tunneling）是将一个 IP 报文封装在另一个 IP 报文的技术，这可以使得目标为一个 IP 地址的数据报文能被封装和转发到另一个 IP 地址。IP 隧道技术亦称为 IP 封装技术（IP encapsulation）。IP 隧道主要用于移动主机和虚拟私有网络（Virtual Private Network），在其中隧道都是静态建立的，隧道一端有一个 IP 地址，另一端也有唯一的 IP 地址。

我们利用 IP 隧道技术将请求报文封装转发给后端服务器，这是因为如果直接转发，由于客户方报文的目标地址是它自己，报文还是会被它接收进来或者根本就不到达物理网络，因而无法转发给后端服务器。但在这里，后端服务器有一组而非一个，所以我们不可能静态地建立一一对应的隧道，而是动态地选择一台服务器，将请求报文封装和转发给选出的服务器。这里，我阅读了 Linux 内核中实现 IP 隧道协议的源代码，参考它实现了 IP 隧道的动态建立和报文的发送。

技术的主要步骤如下：

- （2）查找到目的地址的路由；

- (3) 扩展 IP 报文的头部空间，留出一个 IP 报文头的空间；
- (4) 填充 IP 隧道报文头部；
- (5) 计算校验和；
- (6) 发送。

§ 7.4 性能测试与比较

我们对目前流行的两种采用不同技术的基于内容调度系统和我们的 TCPHA 系统进行了性能对比测试。它们是 Squid 和 KTCPVS，核心技术分别采用 TCP 网关和 TCP 粘合。另外，KTCPVS 还采用了数据零拷贝等技术。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。测试的硬件环境是：三台 PC 构成一个集群，它们的配置均为：Pentium IV 2.5GHZ 处理器，1G 内存，100M 网卡。三台机器均连接到一台 100M 快速以太网交换机，运行的操作系统均为 Linux-2.4.18 的内核。一台机器作为集群前端机，分别运行 squid-2.5.stable4,ktcpvs-0.0.15,tcpa，另外两台作为集群后端机。一台 PC 作为客户机，配置与前面相同。测试软件采用 httpperf-0.8。为了使测试结果更加正确，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，测试结果如下表所示：

请求文件长度	平均每秒完成请求数		
	Squid	KTCPVS	TCPHA
0.3K	352	390	385
10K	59	98	110
1000K	5	10	13

表 7. 1 TCPHA 与 Squid, KTCPVS 性能比较

从以上测试结果可以看出，当请求文件长度较小时，TCPHA 的性能基本与 KTCPVS 的性能持平，因为此时 TCP 迁移的开销相对较大。但随着请求文件长度的增加，TCPHA 的性能开始明显的超过 KTCPVS，当然，更大大超过 Squid。当请求文件长度为 10K 时，TCPHA 的性能比 KTCPVS 高出 12%，比 Squid 高出 86%。当请求文件长度为 1000K 时，TCPHA 的性能比 KTCPVS 高出 30%，比 Squid 高出 160%。由于测试环境的限制，没有足够多的 BE，而 TCP 迁移技术的一个很重要的优势就体现在它的可扩展性好。因此，不难想象，当在有更多的 BE 的集群环境下，以及有更多的客户的并发访问下，TCPHA 会比 KTCPVS 和 Squid 有更明显的性能提高。

§ 7.5 本章小结

本章主要详细地讲述了基于重构连接现场 TCP 迁移方法的服务器集群调度系统 TCPHA 的实现。与当前流行的基于内容调度系统的性能测试比较说明，TCPHA 有着更好的性能。

第八章 总结与展望

TCP 迁移技术是目前国际上一种最新的集群调度技术，有着很好的性能。

本文对 TCP 迁移技术的实现进行了研究，提出了三种实现 TCP 迁移的方法，并提出了解决持久连接问题的方法。并介绍了基于连接现场重构 TCP 迁移方法的集群调度系统原型的实现。

本文还讲述了很多对 Linux 网络源代码的分析成果。

在 TCP 迁移技术的实现和应用等方面，还有下列方面可以改进：

第一，支持持久连接（P-HTTP）的方法还需要进一步完善，其实现时还需要克服一些技术问题；

第二，可以考虑在基于 TCP 迁移的集群调度系统中结合别的方法，如 TCP 粘合技术，这样，如果请求的响应比较小（这可以通过对 HTTP 请求解析得出），则不必用 TCP 迁移技术，使用 TCP 粘合技术就足够了；

第三，研究更好的调度算法。

参考文献

- [1] Mohit Aron, Peter Druschel, and Willy Zwaenepoel, “Efficient Support for P-HTTP in Cluster-Based Web Servers” , Proceedings of the USENIX Annual Technical Conference, Monterey, California, USA, June 6-11, 1999.
- [2] Ravi Kokku, Ramakrishnan Rajamony, Lorenzo Alvisi, Harrick Vin, “Half-pipe Anchoring: An Efficient Technique for Multiple Connection Handoff”, Dept. of Computer Sciences University of Texas, Austin Research Lab Austin IBM.
- [3] 林曼筠, 钱华林, 基于标记的缓存协作分布式 Web 服务器系统, 软件学报, 2003, Vol.14, No.1.
- [4] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-aware request distribution in cluster-based network servers. ACM SIGPLAN Notices, 33(11):205–216, Nov. 1998.
- [5] Yiu-Fai Sit, Cho-Li Wang, Francis Lau, Socket Cloning for Cluster-Based Web Servers, Department of Computer Science and Information Systems, The University of Hong Kong.
- [6] M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel. Scalable content-aware request distribution in cluster-based network servers. In Proceedings of the USENIX 2000 Annual Technical Conference, June 2000.
- [7] W. Tang, L. Cherkasova, L. Russell, and M.W. Mutka. Modular TCP handoff design in STREAMS-based TCP/IP implementation. In Proceedings of the First International Conference on Networking, pages 71–81, July 2001.
- [8] Bryan Kuntz and Karthik Rajan, MIGSOCK Migratable TCP Socket in Linux, Carnegie Mellon University, Information Networking Institute.
- [9] Rutgers University, M-TCP, <http://discolab.rutgers.edu/mtcp/>
- [10] Florin Sultan, Kiran Srinivasan, Deepa Iyer and Liviu Iftode, Highly Available Internet Services Using Connection Migration, Rutgers University Technical Report DCS-TR-462, December 2001.
- [11] Kiran Srinivasan, MTCP: Transport Layer Support for Highly Available Network Services, Master of Science Thesis. Rutgers University Department of Computer Science Technical Report DCS-TR-459, October 2001.
- [12] Saugata Das Purkayastha, Symmetric TCP Splice: A Kernel Mechanism For High Performance Relaying, Department of Computer Science & Engineering, Indian Institute of Technology, Kanpur.
- [13] Saibal Kumar Adhya, Asymmetric TCP Splice: A Kernel Mechanism to Increase the Flexibility of TCP Splice, Department of Computer Science & Engineering, Indian Institute of Technology, Kanpur.
- [14] L. Aversa and A. Bestavros. Load balancing a cluster of web servers using

- distributed packet rewriting. Technical Report 1999-001, CS Department, Boston University, Jan. 6 1999.
- [15] A. Bestavros, M. Crovella, J. Liu, and D. Martin. Distributed packet rewriting and its application to scalable server architectures. Technical Report 1998-003, CS Department, Boston University, Feb. 1 1998.
- [16] L. Cherkasova and M. Karlsson. Web server cluster design with workload-aware request distribution strategy with ward. In Proceedings of the 3rd International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems, pages 212–221, June 2001.
- [17] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, A. Karlin, H. Levy, On the scale and performance of cooperative Web proxy caching, In Proceedings of the 17th ACM Symposium on Operating Systems, Principles (SOSP-99), March 1999.
- [18] A. Cohen, S. Ramgarakam, H. Slye, On the Performance of TCP Splicing for URL-aware Redirection, In Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems, Boulder, CO, Oct 1999.
- [19] M.Y. Luo, C.S. Yang, Constructing Zero-loss Web Services, Published in the proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies (Infocom-2001), Anchorage, AK, April 2001.
- [20] A.C. Snoeren and H. Balakrishnan, An end-to-end approach to host mobility, Published in the proceedings of the 6th Annual International Conference on Mobile Computing and Networking (Mobicom-00), Boston, Ma, August 2000.
- [21] C.S. Yang, M.Y. Luo, Efficient Support for Content-based Routing in Web Server Clusters, Published in the proceedings of the 2nd USENIX Symposium on Internet Technologies (USITS-99), Boulder, CO, October 1999.
- [22] A. C. Snoeren, D. G. Andersen, H. Balakrishnan, Fine-Grained Failover Using Connection Migration, Published in the proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS-01), pp. 221-232, San Francisco, CA, March 2001.
- [23] D.A. Maltz, P. Bhagwat, TCP Splicing for Application Layer Proxy Performance, IBM Research Report RC 21147, 1998.
- [24] J. C. Mogul. The Case for Persistent-Connection HTTP.
- [25] D. Andresen et al. SWEB: Towards a Scalable WWW Server on MultiComputers.
- [26] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A Hierarchical Internet Object Cache. In Proceedings of the 1996 USENIX Technical Conference, Jan. 1996.
- [27] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In Proceedings of the Sixteenth ACM Symposium on Operating System Principles, SanMalo, France, Oct. 1997.
- [28] G. Hunt, E. Nahum, and J. Tracey. Enabling content-based load distribution for scalable services. Technical report, IBM T.J. Watson Research Center, May 1997.

- [29] G. R. Malan, F. Jahanian, and S. Subramanian. Salamander: A push-based distribution substrate for Internet applications. In Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS), Monterey, CA, Dec. 1997.
- [30] Athanasios E. Papatheodou, Eric Van Hensbergen, KNITS: Switch-based Connection Hand-off, University of Rochester Department of Computer Science, International Business Machines, Austin Research Lab.
- [31] Aline Baggio, Maarten van Steen, Transparent Distributed Redirection of HTTP Requests, Vrije Universiteit – Department of Computer Science.
- [32] Mauro Andreolini, Michele Colajanni, Marcello Nuccio, Scalability of contentaware server switches for Clusterbased Web information systems, Department of Information, Systems and Production University of Tor Vergata, Department of Information Engineering University of Modena.
- [33] Eric Dean Katz, Michelle Butler, and Robert McGrath, “A Scalable HTTP Server: The NCSA Prototype”, Computer Networks and ISDN Systems, pp155-163, 1994.
- [34] 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.
- [35] T. Brisco, “DNS Support for Load Balancing”, RFC 1794, <http://www.internic.net/ds/>
- [36] 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.
- [37] Eric Anderson, Dave Patterson, and Eric Brewer, “The Magicrouter: an Application of Fast Packet Interposing”, <http://www.cs.berkeley.edu/~eanders-/magicrouter/>, May, 1996.
- [38] Cisco Systems Inc. Cisco LocalDirector. <http://www.cisco.com/warp/public-/751/lodir/index.html>.
- [39] Alteon Networks Inc. Alteon ACEDirector. <http://www.alteon.com>.
- [40] F5 Networks Inc. F5 Big/IP. <http://www.f5networks.com>.
- [41] 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.
- [42] IBM Corporation. IBM interactive network dispatcher. <http://www.ics.raleigh.ibm.com/ics/isslearn.html>.
- [43] Guerny 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.
- [44] 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.
- [45] Zeus Technology, Inc. Zeus Load Balancer v1.1 User Guide. <http://www.zeus.com/>
- [46] Edward Walker, “pWEB - A Parallel Web Server Harness”, <http://www.ihpc.nus.edu>.

- sg/STAFF/edward/pweb.html, April, 1997.
- [47] 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.
 - [48] 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.
 - [49] 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>.
 - [50] 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>.
 - [51] 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.
 - [52] KTCPVS.<http://www.linux-vs.org/software/ktcpvs/index.html>
 - [53] M. F. Arlitt and C. L. Williamson. Web Server Workload Characterization: The Search for Invariants. In Proceedings of the ACM SIGMETRICS '96 Conference, Philadelphia, PA, Apr. 1996.
 - [54] W. Stevens. TCP/IP Illustrated Volume 1 : The Protocols. Addison-Wesley, Reading, MA, 1994.
 - [55] 李元佳, 网络接口源码导读, <http://www.douzhe.com/joyfire/kernel/5.html>
 - [56] 赵蔚, Netfilter:Linux 防火墙在内核中的实现, <http://www-900.ibm.com/developerWorks/cn/linux/network/l-netip.html>
 - [57] 杨沙洲, Linux Netfilter 实现机制和扩展技术, <http://www-900.ibm.com/developerWorks/cn/linux/l-ntflt/index.shtml>
 - [58] 朱小平, Linux 协议栈 skbuff 分析
 - [59] 章文嵩, 可伸缩网络服务的研究与实现, 国防科技大学 2000 年工学博士学位论文
 - [60] Wensong Zhang, Shiyao Jin and Quanyuan Wu, "LinuxDirector: A Connection Director for Scalable Internet Services", Journal of Computer Science and Technology, 2000, 15
 - [61] 毛德操, 胡希明, Linux 内核源代码情景分析, 浙江大学出版社, 2001 年 9 月
 - [62] Daniel P. Bovet, Marco Cesati, "Understanding the Linux Kernel, 2nd Edition" , December 2002
 - [63] Balachander Krishnamurthy, Jennifer Rexford 著, 范群波, 沈金河译, Web 协议与实现, 科学出版社, 2003