电子科技大学

硕士学位论文

分布式并行操作系统中调度的研究和实现

姓名：刘流

申请学位级别：硕士

专业：计算机系统结构

指导教师：刘心松

2001.1.1

中文摘要

学科专业：计算机系统结构

论文题目：

分布式并行操作系统中任务调度的设计与实现

硕士生：刘流 **99S6-035** 导师：刘心松教授

随着计算机技术的广泛应用，对分布式并行操作系统的需求越来越大。分析当前国内外操作系统的发展趋势，我们迫切需要开发一个分布式并行操作系统。因此，本课题以开放源码的 Linux 操作系统为基础，对它进行改造，开发出分布式并行调度系统。

本论文首先介绍了分布式并行操作系统的概念和特点，以及设计分布式并行操作系统需要注意的问题。接着专门讨论了分布式并行操作系统中的调度，主要是处理机的分配算法，并对若干种现有的调度算法进行了深入分析。论文的主要部分是关于分布式并行调度的设计与实现，在介绍了分布式并行操作系统的设计结构后，开始主要介绍分布式并行调度的结构组成。对其中的每一个组成部分都分别介绍了设计思想和实现方案。最后，论文从响应时间和负载均衡两个方面的测试数据对分布式并行调度的性能进行了评估。

关键词：分布式并行调度；负载均衡；发送者发起算法；

Abstract

Major: Computer Architecture

Subject:

Design & Implementation of Task Scheduling in Distributed Parallel Operating System

Master: Liu Liu **99S6-035** Advisor: Prof. Liu Xinsong

With the using of computer technology abroad, the need of distributed parallel operating system highly increases. According to analyze the international progress direction of distributed parallel operating system, we need to develop an distributed parallel operating system of ours urgently. So we develop a distributed parallel operating system based on modifying the Linux of opening source.

In this thesis, we firstly introduce the concept and characteristic of distributed parallel and problems of designing distributed parallel operating system. Then it study the scheduling of distributed parallel operating system specially, that is the algorithm of assigning processor and analyze the existent algorithm of scheduling. The body of thesis introduces the design and implementation of distributed parallel scheduling. After designing the architecture of distributed parallel operating system, the thesis introduces the design and implementation of each part of distributed parallel scheduling. At last the thesis evaluates the capability of distributed parallel scheduling according to analyze the data coming from response time and load balancing.

Key Words: Distributed parallel scheduling; Load balancing; Sender-initiated algorithm

第一章 引言

随着时代的发展，计算机技术在各行各业得到了广泛的应用，操作系统作为各种应用软件的底层平台，有着十分重要的作用和地位。操作系统的稳定性、安全性和效率直接影响了应用软件的使用。

从 20 世纪 80 年代中期开始，技术上出现了两大进步。

首先是功能更强的微处理机的开发，开始出现了 8 位的机型，随后不久 16 位，32 位，甚至 64 位的 CPU 也开始普及。其中许多机器具有较大主机（即大型机）的计算能力，但价格却只是它的几分之一。

第二个进步是高速计算机网络的出现。局域网 LAN 使得同一建筑内的数十甚至上百台计算机连接起来，使少量的信息能够在大约 1 毫秒左右的时间里在计算机之间传送。更大量的数据则以 10^7~10^8 比特/秒或者更高的速率进行传送。广域网 WAN 使得全球范围内的数百万台计算机连接起来，传输速率从 64Kbps 到用于一些先进的实验型网络中的 1Gbps。

应用这些技术的结果，使得将大量 CPU 组成的计算机系统通过高速网络连接在一起不仅成为可能，而且变得十分容易。

有了这两个基础，就有可能在上面开发出优秀的分布式并行操作系统。但是非常遗憾，国外从 80 年代就开始对分布式并行操作系统进行研究，至今已经出现了四十多个实验系统，也有几个成熟的形成产品的系统。国内也对分布式并行操作系统作了多年的研究，也产生了多个实验系统，但是至今没有成熟的产品系统。

鉴于国内外分布式并行操作系统的发展状况，从 80 年代就在美国致力于分布式并行操作系统研究的刘心松教授毅然决定开发出一个能形成产品的分布式并行操作系统，这将不仅使操作系统国产化成为现实，而且使国内在分布式并行操作系统的研究上处于国际领先地位。

设计一个分布式并行操作系统，可以有两种途径。一是设计一个全新的系统。这样，由于针对性强，可以获得优秀的性能。但是，设计周期长，而且难以继承现有的成功软件。第二种途径是通过对集中式操作系统进行扩充改造，它的优点是：集中式操作系统中的许多部分可以直接继承，研制周期短，而且原来的用户程序可以不加修改地在新系统中运行。

经过详尽考虑和深入研究，我们毅然决定改造 Linux 操作系统，使之变为一个分布式并行操作系统。作为一个分布式并行操作系统，它必然有其基本的组成部分：分布式并行通信、分布式并行调度、分布式并行文件系统

等。在整个系统中，我主要负责分布式并行调度部分。因此，本论文的主要内容是分布式并行操作系统中调度的研究与实现。论文内容安排如下：

第一章　引言

第二章　分布式并行操作系统概述 简要介绍分布式并行操作系统的概念以及各个组成部分。

第三章　分布式并行操作系统调度的研究 深入对分布式并行操作系统的调度进行了研究，详细分析了分布式并行操作系统调度的实现目标。

第四章　分布式并行操作系统调度的设计与实现 针对分布式并行操作系统调度的实现目标，详细设计了调度的实现方案。

第五章　系统性能分析 通过获得的实验数据对分布式并行操作系统进行性能分析。

第六章　结论 对整个论文所涉及的工作做了总结，并提出了对分布式并行操作系统调度可能的进一步改进。

第二章 分布式并行操作系统概述

2.1 操作系统简介

2.1.1 什么是操作系统

尽管多数计算机用户都有一些使用操作系统的体验，但是要给出操作系统的正确定义却很困难。部分原因是操作系统执行两个相对独立的任务。

2.1.1.1 作为扩展机器的操作系统

在机器语言一级上，多数计算机的体系结构（指令集、存储组织、I/O和总线结构）是原始的而且编程是很困难的。一般程序员并不想涉足对硬件的直接编程，程序员需要的是一种简单的，高度抽象的可以与之打交道的设备。

将硬件细节与程序员隔离开来，并提供一个可以读写的命名文件的简洁的方式，这就是操作系统。操作系统屏蔽了磁盘、定时器、存储器管理等底层硬件的特性，提供一个简明的、面向文件的接口，这些抽象都较底层硬件本身更简单、更容易使用。

从这个角度来看，操作系统的作用是为用户提供一台等价的扩展机器，或者称为虚拟机，它比底层硬件更容易编程[1][3]。

2.1.1.2 作为资源管理器的操作系统

把操作系统看作是提供给用户的基本的方便接口的概念是一种自顶向下的观点。按照另一种自底向上的观点，操作系统则用来管理一个复杂系统的各个部分。现代计算机包含处理器、存储器、时钟、磁盘、终端、磁带设备、网络接口、打印机以及许多其他设备。从这个角度来看，操作系统的任务是在相互竞争的程序之间有序地控制对处理器、存储器以及其他 I/O 接口设备的分配。

当计算机有多个用户时，特别是用户还需要共享诸如磁带机和照片输出机等昂贵的资源时，就更有必要管理和保护存储器、I/O 设备以及其他设备。经济考虑是一方面，还要让用户共享信息。总之，这种观点认为操作系统的主要任务是跟踪谁在使用什么资源、满足资源请求、记录使用状况，以及协调各程序和用户对资源使用请求的冲突[3]。

2.1.2 操作系统基本概念

操作系统与用户程序的界面由操作系统提供的“扩展指令”集定义，这些扩展指令传统上称为“系统调用（system call）”。系统调用创建、删除和使用由操作系统管理的各种软件对象，其中最重要的是进程和文件。

2.1.2.1 进程

在所有操作系统中，一个重要概念是进程（process）。一个进程基本上是一个执行的程序。它包括可执行的程序、程序数据、堆栈、程序计数器、其他寄存器以及所有运行程序所需要的信息。

在分时系统中，操作系统周期性地挂起一个进程，然后启动运行另一个进程，因为在过去的一个时间周期内，第一个进程已经运行完分配给它的时间片。在上述进程被暂时挂起的情况下，之后的某个时刻它再次启动时的状态与先前暂停时的状态完全相同，这就意味着在将其挂起时该进程的所有信息都要被记录下来。在许多操作系统中，进程的所有信息，除了进程地址空间的内容以外，均被存放在操作系统管理的一张表中，称为进程表（process table）。进程表是一个数组（或链表），当前存在的每个进程都要占用其中一项。所以，一个进程包括进程的地址空间以及对应的进程表项，其中包括所用的寄存器及其他信息。

一个进程能够创建一个或多个进程，这些被创建的进程称为子进程（child process），而且这些子进程又可以创建它们的子进程，这样就可以得到一棵进程树。

有时，需要向一个进程传送信息，而该进程并没有等待接收信息，那么发送者可以要求操作系统给接收者一个通知。操作系统向这个进程发送一个信号，接收到信号将导致该进程暂时挂起，将寄存器值保存到堆栈中，并启动一个特殊的信号处理例程。信号处理程序结束后，进程从接收到信号的断点处继续执行。信号是对硬件中断的软件模拟，有很多原因可以导致产生信号。信号可以用于进程间通信。

2.1.2.2 文件

大多数的操作系统支持将一组文件形成目录的形式来存放文件。目录项可以是文件或目录，这样就产生了层次-文件系统。

文件可以形成树状层次结构，这种层次可以很深，常常多达四、五层，且能持久存在。

目录层次结构中的每一个文件都可以用从根目录开始的路径名来确定，绝对路径名包含了从根目录到该文件的所有目录清单。

不同的操作系统提供不同的手段来对文件进行保护。

多数操作系统提供一种抽象，以允许在不了解硬件细节的情形下让用户完成 I/O 操作。这种抽象把 I/O 设备视作特殊文件，提供特殊文件的目的是使 I/O 设备使用起来更象文件。在这种方式下，可以使用与普通文件相同的系统调用来对特殊文件进行读写。

管道是一种虚拟的文件，用来连接两个进程。进程 A 要向进程 B 发送数据时，把管道文件当作输出文件，往里写数据。进程 B 则将管道文件当作输入文件，从中读数据。于是，进程间通信就如同是在读写普通文件。

2.1.3 操作系统结构

2.1.3.1 整体式系统

最常用的结构方式是整体式系统，整个操作系统是一堆过程的集合，每个过程都可以任意调用其他过程。使用这种技术时，系统中的每个过程都有一个完好定义的接口，即入口参数和返回值，而且相互之间的调用不受约束。

这种组织方式提出了一种操作系统的基本结构：有处理服务过程请求的一个主程序，有执行系统调用的一套服务过程，有支持服务过程的一套实用程序。

在该模型中，每一系统调用都由一服务过程完成；有一组实用程序完成一些服务过程需要用到的功能。

2.1.3.2 层次式系统

将整体式系统进一步通用化，就变成了层次式系统，上层软件都基于下一层软件之上。

THE 系统就是典型的整体式系统，该系统分为五层。处理器分配在第 0 层中进行，内存管理在第 1 层中进行，第 2 层软件处理进程与操作员控制台之间的通信，第 3 层软件则管理 I/O 设备和相关的信息流缓冲区，第 4 层是用户程序层。

MULTICS 采用了通用层次化的更一步概念，它由许多同心的环构成，内层环比外层环有更高的级别。

2.1.3.3 虚拟机

基于如下思想设计，一个分时系统应该提供多道程序的功能，以及一个比裸机具有更方便的扩展界面的计算机。

系统的核心被称为虚拟机监控程序，它在裸机上运行并且具备了多道程序功能。该系统向上层提供了若干台虚拟机，它不同于其他操作系统的是这些虚拟机不是那种具有文件等优良特征的扩展计算机，它们仅仅是精确复制的裸机硬件。它包含核心态/用户态，I/O 功能，中断，以及其他真实硬件所具有的全部内容。

2.1.3.4 客户机/服务器系统

现代操作系统的一个发展趋势是，进一步发展将代码移到更高层次的思想，即尽可能多地从操作系统中去掉东西，只留下一个很小的内核（kernel）。通常采用的方法是，由用户进程来实现大多数操作系统的功能。

在这种模型中，操作系统内核的全部工作是处理客户机与服务器之间的通信。操作系统被分成了多个部分，每个部分仅仅处理一个方面的功能。这样每个部分更小，更容易管理。而且，所有的服务都以用户进程的形式运行，它们不在核心态下运行，所以不直接访问硬件。

客户机/服务器模型的另一个优点是，它适用于分布式系统。

2.2 分布式计算机系统的概念和特点

2.2.1 分布式计算机系统的定义

“一个分布式计算机系统是一些独立的计算机的集合，但是对这个系统的用户来说，系统就象一台计算机一样。”这个定义有两方面的含义，第一，从硬件角度来讲，每台计算机都是自主的；第二，从软件角度来讲，用户将整个系统看作是一台计算机[2]。

2.2.2 分布式计算机系统的优点

分布式计算机系统的特点在于系统中具有多台计算机且所有的计算机都有相等的地位，由于有这个特点，所以与传统的集中式系统相比，分布式计算机系统具备许多优越的性能：

经济性：多个计算机能提供比大型机更好的性能价格比。

速度：分布式计算机系统能提供比大型机更强大的计算能力。

固有的分布性：有一些应用包含了物理上分散的机器。

可靠性：当某台机器崩溃时，整个系统仍能正常工作。

可扩展性：处理能力可按需增加。

2.3 设计分布式并行操作系统需要注意的问题

2.3.1 透明性

所谓透明性就是让用户觉得一群机器和一台普通的单处理器系统是一样的。

可以在两种层次上达到透明性。最容易的作法就是对用户隐藏分布性；在更低的一个层次，也可以让系统对程序透明，即系统调用的接口被设计成看不出有多个处理器存在。

透明包括几个方面的含义，如表 2-1 所示。

表 2-1 透明性种类的说明

种　类	说　明
位置透明性	用户不清楚资源的位置
迁移透明性	资源可以任意移动，名字不改变
复制透明性	用户不清楚有多个拷贝存在
并发透明性	多个用户可以自动共享资源
并行透明性	任务被并行地执行，用户不需要知道

位置透明性指的是在一个真正的分布式系统中，用户说不出软、硬件资源的位置，如 CPU、打印机、文件、数据库等资源，资源的名字不能隐含资源的位置。

迁移透明性指的是资源可以自由地从一处迁移到另一处，它们的名字无需改变。

复制透明性指的是操作系统可以任意复制文件和其他资源，形成多个拷贝，而用户却不用关心这点。

并发透明性指的是分布式系统通常有多个相互独立的用户，当有多个用户同时访问同一资源时，每个用户不会注意到有其他用户正在访问资源。

并行透明性指的是所有任务在用户不知道的情况下，在多个处理器上被并行地执行。

2.3.2 灵活性

大多数分布式系统采用微内核结构，微内核具有很高的灵活性，它一般只提供 4 种最小的服务：进程间通信机制、某些内存管理工作、有限的低级进程管理和调度、低级的输入/输出。大部分操作系统的服务通过用户级的服务器来实现。

2.3.3 可靠性

建立分布式系统的目的之一是提高系统的可靠性。它的基本思想是，如果有一台机器坏了，其他机器能够接替它进行工作。高可靠性的分布式系统必须有高可用性和高安全性，另外还应该有容错功能，即一个节点出了故障，修复以后应该能够恢复到原有的正常状态。

2.3.4 性能

性能问题是任何时候都需要考虑的问题，一个分布式系统的性能在某些情况下可能比单一的计算机性能略差，这是因为分布式系统的性能和系统中的通信紧密相关。如果通信质量很差，则系统性能相应会很差。在任务调度的同时必须要考虑通信开销，这样才能保证系统的性能。

2.3.5 可扩展性

分布式系统应该有很好的可扩展性，系统规模可大可小。总的设计原则是避免集中式的部件、表和算法。

2.4 本章小结

本章首先介绍了操作系统的基本概念，主要对进程和文件进行了介绍。进而介绍了分布式系统的基本概念和优点，最后就分布式系统的设计需要注意的问题作了介绍。

第三章 分布式并行操作系统中的调度

根据分布式系统的定义，分布式系统由多个处理机组成，这些处理机的组织形式可能是工作站集合、一个公共的处理机池或者是某种组合的形式。但无论是哪种形式，都要有一种算法来决定在哪台处理机上运行哪个进程，这就是分布式并行操作系统的调度。

3.1 问题的提出

分布式计算机系统（Distributed Computing System - DCS）有许多潜在的优点，它能大大提高系统的可用性、可靠性、灵活性及总体性能，而且可以获得很高的性能/价格比。但是，要充分发挥分布式计算机的潜在优势，必须解决一些问题。其中最关键的一个问题是必须采取某些手段，使整个系统的负载均衡分配在所有的处理机上，以避免出现某些节点机分配的任务很多（我们称之为重载），而其他一些节点机却是空闲的（我们称之为轻载或空载）现象。

分布式计算机系统这种负载不公平分配的现象，是由于任务到达的随机性，以及各个节点机处理能力上的差异造成的。在系统运行过程中，一方面，要使重载节点机上的任务尽可能快地完成；另一方面，要使空闲节点机尽可能忙起来。

如何避免这种忙闲并存的情况，从而有效地提高系统的资源利用率，减少任务的平均响应时间呢？负载平衡就是这样一种有效的方法。它在任务到达时设法给任务分配一个轻载节点机，使各个节点机的负载大致相等，以达到平衡整个系统负载的目的。这种负载平衡的方法就是分布式并行操作系统的调度。

3.2 分布式并行调度算法概述

3.2.1 处理机的分配模型

在讨论特定的算法和设计原则之前，要先讨论一些关于处理机分配的基本模型、假设及其目标的问题。

在这个领域中几乎所有的模型都假设所有的计算机都是一样的，或者至少是代码兼容的，最多在速度上有所不同。

几乎所有的已有模型都假设系统是完全连接的，即每台处理机都能与其他任何一台处理机进行通信。

处理机的分配策略可以分为两大类：一类是不可迁移的，另一类是可迁移的。在不可迁移的策略下，当创建一个进程时，系统就决定为该进程分配哪台处理机。一旦分配完毕，进程将一直在这台处理机上运行，直至结束。无论这台处理机是多么严重地超载，也不管其他处理机有多空闲，它都不能迁移到其他的处理机上。相反，在可迁移的策略下，可以将已经开始运行的进程迁移到其他处理机上继续运行。可迁移策略能够提供更好的负载平衡，但是同时也增加了系统的复杂性，并对系统的设计有很大的影响。

对于处理机分配算法，我们总是希望对它进行一些优化。然而准确地说各个系统想要优化的内容各不相同，即它们的目标是不一样的。

一个可能的目标是使 CPU 的利用率最大化，即每小时中，使实际运行用户任务的 CPU 周期数最大，也即尽可能地减少 CPU 的空闲时间，让每个 CPU 都在运行。

另一个有价值的目标是使平均响应时间最小化。如图 3-1 所示，有两个处理机和两个进程。处理机 1 以 10MIPS 的速度运行，处理机 2 以 100MIPS 的速度运行，但是有一个后备进程的等待列表需要耗时 5 秒才能完成其中的进程。现在有两个进程，进程 A 有 1 亿条指令，进程 B 有 3 亿条指令。每个进程在每个处理机上的响应时间（包括等待时间）在图中已经显示出来。如果让处理机 1 运行进程 A，处理机 2 运行进程 B，那么平均响应时间是（10+8）/2=9 秒。如果反过来，平均等待时间是（30+6）/2=18 秒。很显然，前者方案在最小化平均响应时间上要优于后者。

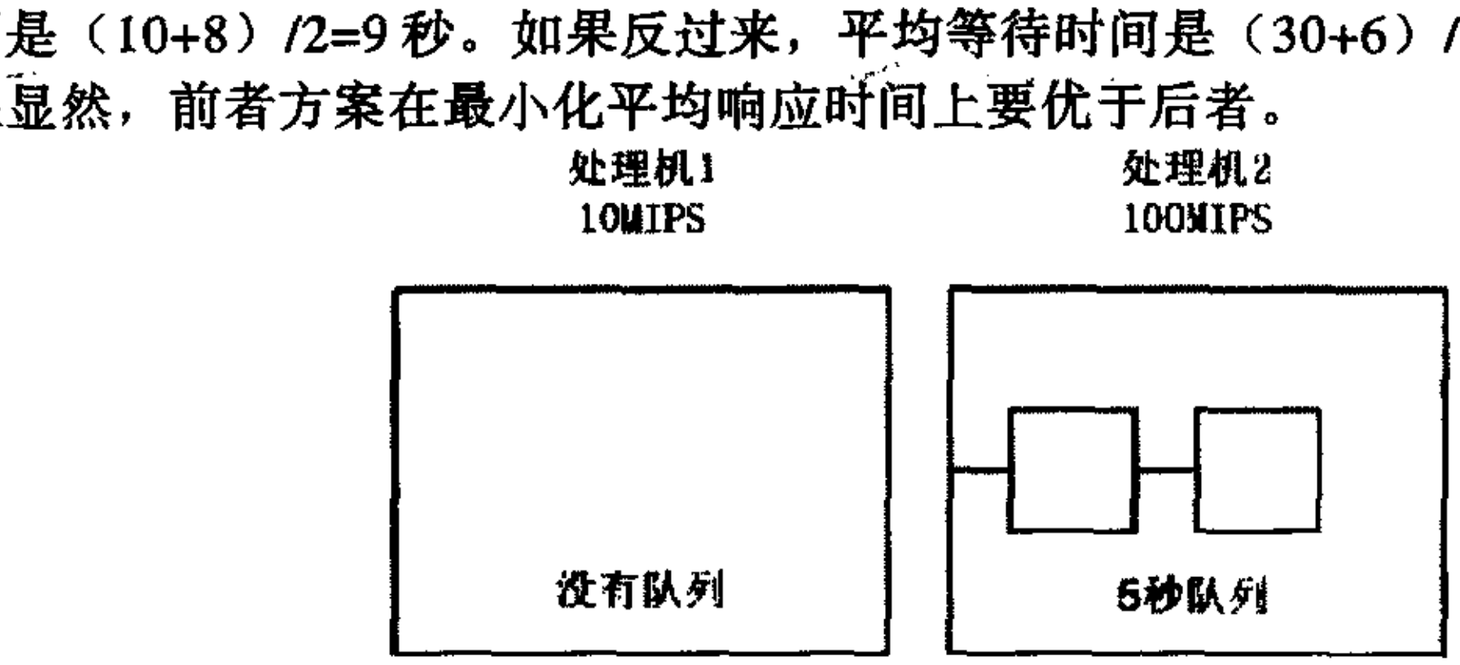

图3-1 在两个处理机中两个进程的响应时间

最小化响应时间的一个变形是最小化响应率，响应率的定义是在一台机器上运行一个进程的时间除以这个进程在一个无负载的标准处理机上运行时应该花的时间。对于很多用户来说，响应率是一个比响应时间更加有效的参数，因为它考虑了大的作业会比小的作业花更多的时间这样一个事实。比

如，一个 1 秒的任务花了 5 秒时间完成，而一个 1 分钟的任务花了 70 秒，用响应时间来算，前者要好一些，而用响应率来算，则是后者要好得多，因为 5/1 远远大于 70/60。

3.2.2 处理机分配算法的设计问题

在设计算法时，设计者要考虑的问题可以概括为以下五个方面：

(1) 确定性与启发性（试探性）算法：

确定性算法适用于当进程的所有行为都是预先可知的情况。如果知道所有进程的计算需求、文件需求、通信需求等情况，就可以设计出一个完美的调度方案。但是只有极少数系统的所有信息是预先可知的，而大多数的系统信息是不可知的，每个小时甚至每分钟都有可能发生很大的变化。在这种系统中，处理机分配不能用数学的、确定性的算法来实现，而只能用特别的启发性（试探性）算法。

(2) 集中式算法与分布式算法：

集中式的算法是把所有的信息搜集到一个地点便于作出更好的决定，但是这使得系统不够健壮，而且使中心机器的负载过重。通常倾向于采用分布式算法。

(3) 最优与次优算法：

用集中式和分布式算法都能够得到最优解，但是都比得到次优解的代价大得多。要得到最优解必须搜集更多的信息，并进行更全面的处理。在实际实现中，大多数分布式系统都将目标定位在启发性的、分布式的、次优的解决方案上，因为要得到最优解太困难了。

(4) 本地与全局算法：

这个问题与转移策略有关。当一个任务到达时，系统要决定是否让这个任务在接收它的机器上运行。如果那台机器太忙，这个任务就应该被转移到其他机器上运行。这时就要决定是否只根据本地的信息来决定转移策略。本地算法主张，如果机器的负载低于某个阈值，就让该任务在这台机器上运行，否则就要转移该任务。全局算法主张，在判断本地机器是否太忙以前，先收集一些全局信息，再决定是否在本地机器上运行该进程。本地算法简单，但是远远到不达最优，而全局算法只不过稍好一些，却付出了更大的代价。

(5) 发送者发起与接收者发起算法。

这个问题与定位策略有关。一旦转移策略决定要转移一个任务，定位策略就要决定将该任务转移到何处。定位策略不能是本地的，它需要别处的负

载信息来对任务转移到何处作出明智的决定。这个信息可以用两种方法来传递，一种是发送者发起，另一种是接收者来启动。

例如，在图 3-2(a)中，这里有一台过载的机器，它向别的机器发送请求希望得到帮助，它希望将一些新任务卸载到别的机器上。在这里，是由发送者主动定位其他 CPU 的。相反，在图 3-2(b)中，这里有一台空闲的或者说是欠载的机器，它向别的机器发送信息，声明自己是空闲的，可以承担一些额外的工作。它的目的是找到一台愿意给它一些事情作的机器。

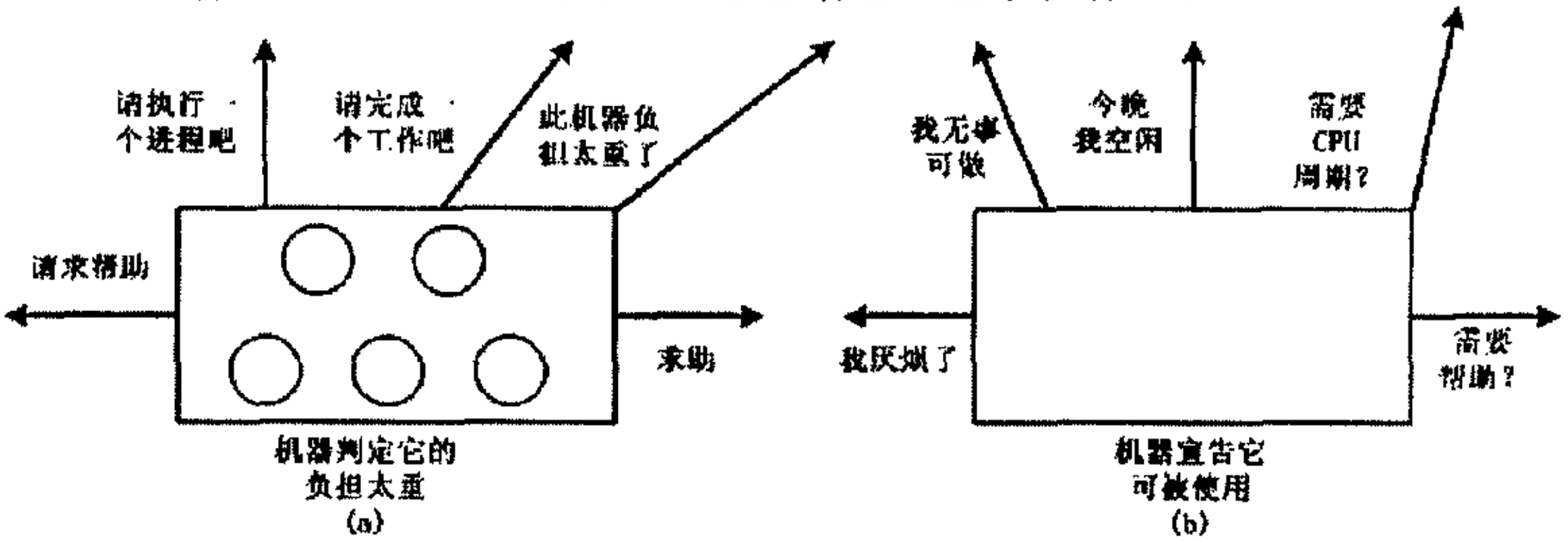

图3-2 (a)发送者寻找空闲机器
(b)接收者寻找工作做

3.3 分布式并行调度算法分析

上面我们讨论了分布式调度算法的设计问题，这里简要分析几种常见的调度算法。

3.3.1 图论确定性算法

一类广泛研究的算法是基于这样的一些系统，构成它的进程知道它所要求的 CPU 和内存要求，并且知道系统中每对进程之间要求的平均通信量。如果系统的 CPU 数量小于进程数目，则要求将多个进程指派到同一个 CPU 上。整个算法的思想是使这种指派能够使网络的通信量最小化。

整个系统可以表示为一张带权图，每个节点表示一个进程，每条边表示两个进程之间的通信量。从数学的角度来讲，整个问题就变成了如何根据特定的限制将图划分为 k 个不相连的子图。对于每种满足限制的解决方案，子图内部的边意味着机器内部的通信，可以忽略。从一个子图连向另一个子图的边表示网络通信。算法的目标就是在满足所有的限制下，找到一种划分方式使网络通信量最小。

显而易见，我们是在寻找紧耦合的聚集。但是这些聚集之间很少相互作用（集合内的通信量大，集合之间的通信量小）。

3.3.2 集中式算法

图论算法应用非常有限，因为它需要事先了解完整的信息。这里讨论一种不需要事前了解任何信息的启发性算法。

算法中有一个协调者，它保存着一张使用情况表，每个工作站在表中都有一个条目，初值为 0。当有重要的事情发生时，将给协调者发消息以更新使用情况表。算法将根据使用情况表决定处理机的分配。这些决定发生在调度事件发生时，即有进程请求处理机、处理机进入空闲状态或者是发生了时钟中断时。

这个算法所以是集中式的，是因为它所考虑的是使每台工作站的用户公平地享用系统的计算能力，而不是试图使处理机的利用率最大化。

3.3.3 层次式算法

集中式的算法，不能很好地适应大型系统。中央节点很快会成为系统的瓶颈，更不用说是某个节点失效了。这些问题可以用层次式算法来解决。层次式算法基本保持了集中式算法的简单性，并且能够很好地适应大型系统。

提出一个监视处理机集合的方法，就是将所有的处理机以一种与网络物理结构无关的方式组织成一个逻辑分层结构。

对每组 k 个工作，分配给管理者一个任务，即跟踪各个工作者谁在忙，谁正空闲。如果系统很大，将会有众多的管理者，所以有的机器要变为管理者的管理者。这种层次可以无限扩展下去。

如果一个管理者停止了正常的工作，一个方案是选择一个它的直接下属成为临时管理者，取代它的位置。这个选择可以由它的下属共同作出，也可以由它的同级作出，还可以由它的上级作出。

为了避免单个管理者在层次的最上层，可以截去最上层，而在最上层形成一个委员会，拥有最高权利。如果委员会中的一个成员无法工作，其他成员会在下一层中选出一名成员取代其位置。

3.3.4 发送者发起分布式启发性算法

一个典型的算法是 Eager 等人在 1986 年提出的，在他们研究的“最有效代价”的算法中，当创建进程时，创建进程的机器将对一个随机选择的机器发送询问，询问那台机器的负载是否低于某个阈值。如果是，将发送进程，否则，将选择另一台机器并发送询问。这个过程不会一致持续下去，如果在 N 次询问内还没有找到合适的机器，算法将停止，新进程将在创建它的机器上运行。

然而，在负载十分重的情况下，所有的机器都会不停地毫无意义地向其他机器发送询问，想找到一台愿意接受更多工作的机器。在这种情况下，几乎没有机器会被减轻负载，但却引起系统相当可观的额外开销。

3.3.5 接收者发起分布式启发性算法

根据这个算法，当一个进程结束时，系统将检查自己是否有足够的工作可以做，如果没有，将随机向一台机器申请工作。如果那台机器没有要给予的工作，系统将继续询问第二、第三台机器。如果连续 N 次询问都没有申请到工作，系统将暂时停止申请，开始处理系统队列中一个等待的进程，当这个进程结束后，再开始下一轮申请。如果系统无事可做，则将进入空闲状态，一定的时间后，它从新开始申请。

这个算法的好处就是它不会在系统非常繁忙的时候给系统增加额外的负载，当然，在系统几乎没有事情可做的时候，会拼命向别的机器申请工作，会造成相当大的询问负载。然而，在系统欠载时使系统开销增大总比在系统过载时这样做要好得多。

3.3.6 投标算法

这种算法试图将计算机系统变成小型的经济社会，由买卖双方和需求关系确定的价格组成。算法的核心参与者是进程，为了完成工作，进程必须购买 CPU 时间，而 CPU 则拍卖它的处理机周期给出价最高的买主。

每个处理机在一个公共可读文件中公布它们的近似价格。这个价格并不是确定的，而是为了表示所提供服务的大概价格。根据处理机速度、内存容量、浮点运算能力以及其他一些特性，不同的处理机有不同的价格。所提供的服务的指示，如预期的响应时间也可以同时公布。

当一个进程要启动一个子进程时，它将查询现在有谁在提供它所需要的服务，然后确定一个它可以支付得起的处理机集。通过计算，它将从这个处理机集中选出一个最好的。根据应用程序的不同，最好的可以指最便宜的、最快的或最高性能价格比的。然后它将向第一选中的处理机发出一个出价，所出的价格可能会高于或低于已公布的价格。

处理机收集向它们发出的所有出价，然后作出决定，可能是选择出价最高的。然后通知选中的未选中的进程，并开始执行被选中的进程。公共文件中此处理机的价格将被更新，以反映处理机的最新价格。

3.4 本章小结

本章首先介绍了分布式操作系统中调度的概念，然后介绍了分布式操作系统中的调度模型以及设计的若干问题。最后，分别介绍了常见的几种调度算法，为下面的分布式操作系统中调度的设计和实现做好了准备。

第四章 分布式并行操作系统调度的设计与实现

作为一个操作系统，它应该支持绝大多数的应用服务。所以，我们的目标是要设计一个通用的调度模块，使得任何应用服务都可以在其上运行。当然，这些应用服务必须要遵循一些规约。

调度模块有高度的灵活性，它的各个组成部分可以自由地替换，这是我们设计的重要基础之一。例如，负载计算、副本管理、任务调度算法这些部分，当有了更好的设计思想或者效率更高的方法，便可以将相应的部分重新设计并予以替换，以期达到更高的性能指标。

4.1 分布式并行操作系统的结构

4.1.1 开发平台

我们的分布式并行操作系统是基于 Linux 开发的，这是有深刻原因的。目前常用的商业操作系统有 Microsoft 的 DOS、Windows9x、WindowsNT、SCO 公司的 SCO UNIX、SUN 公司的 Solaris OS、Apple 公司的 Mac OS 等等。它们都是比较成熟的产品，而且稳定性都有保证。但是它们有一个特点，都是国外的产品，我们不了解它们的技术细节。因此，当使用这些操作系统的计算机系统与 Internet 相连接时，我们不能保证是否会泄露一些重要的信息，也不能保证系统是否会存在安全漏洞。越来越多的报道披露，Microsoft 的 Windows 系列操作系统都为美国国家安全局提供了“后门”。这些事情让人们开始重视自己正在使用的操作系统的安全性问题。我国已经要求某些政府重要部门的连网计算机不能使用 Microsoft 的操作系统，其原因是这些国外的商业操作系统在我们看来就像一个“黑匣子”。

在这种情况下，符合 GNU、GPL 的 Linux 操作系统公开的源码对我们来说无疑是一个很好的系统。因为我们可以通过源码了解 Linux 操作系统的实现机制，从而保证它没有安全“后门”的存在，而且在该操作系统上已经有了很多成熟的应用能够为我们方便地使用。另外，Linux 内核和其上的许多系统软件以及应用软件的源码是公开且免费的。Linux 除了具有一般 UNIX 的工具外，还支持多种系统语言（如 C、C++等）、多种脚本语言（如 Perl 等）、多种自然语言（如中文、英文等）、多种免费数据库、多种网络应用，另外还支持与其他操作系统的共享（如 Windows 系列）。并且，Linux 对硬件的要求也不高。所以，Linux 是一种很有发展前途的操作系统。

近年来，国内系统软件对 Linux 开发热情逐渐膨胀起来，中文化 Linux 有了 Turbo Linux、蓝点 Linux 和红旗 Linux 等，并且它们的使用范围也在逐渐扩展。

我们研究室作为计算机系统结构的资深研究室，对 Linux 的内核结构作了全面、深刻的分析，并发表了一个关于 Linux 分析结果的专辑。这些工作对我们选择在 Linux 平台上进行分布式并行操作系统的开发提供了便利。

4.1.2 分布式并行操作系统结构介绍

分布式并行操作系统的结构如图 4-1 所示。整个系统不改变原来 Linux 系统的硬件驱动部分，对原来 Linux 系统的调度、文件和通信子系统进行修改，变为相应的分布式并行调度、分布式并行文件和分布式并行通信三个子系统，它们和原来的硬件驱动组成一个新的分布式并行操作系统。

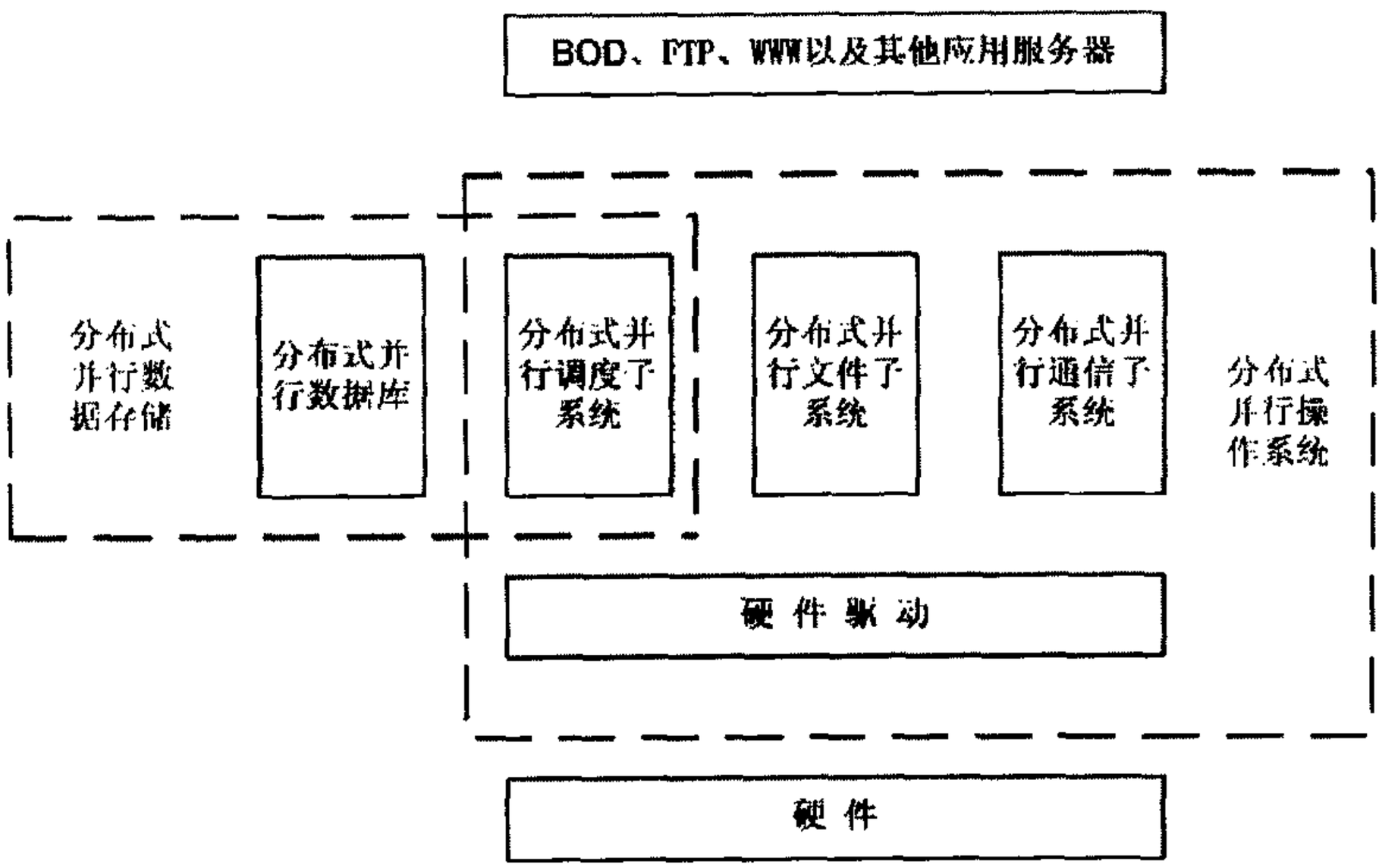

图 4-1 分布式并行操作系统结构图

在修改操作系统的同时，我们还基于 MySQL 研发了一个分布式并行数据库 DPSQL。MySQL 是一个起源于北欧的一个 SQL 客户机/服务器关系数据库管理系统，它包括一个 SQL 服务器、访问服务器的客户机程序、管理工具和一个编写用户自己的程序的编程接口。MySQL 并不是一个开放源代码的产品，因为在某些条件下使用它需要许可证。但是，MySQL 很愿意在开放源代码的团体内得以普及，所以除非通过出售 MySQL 或出售需要它的

服务来挣钱，否则，大体上说 MySQL 一般是免费的。MySQL 的普及并不限于开放源代码团体内，虽然它在个人计算机上运行，但它是可移植的，并且运行在商用操作系统（如 Solaris、Irix 和 Windows）和一直到企业服务器的各种硬件上。此外，它的性能也足以和任何其他系统相匹敌，而且它还可以处理具有数百万个记录的大型数据库[5]。基于这些原因，所以我们选择了 MySQL 来开发我们的分布式并行数据库。

4.2 分布式并行调度子系统

作为分布式并行操作系统的核心，分布式并行调度子系统要与其他的子系统进行交互，涉及的面很广，这就要求我们对调度子系统的设计要尽可能的完整。整个调度子系统的结构如图 4-2。

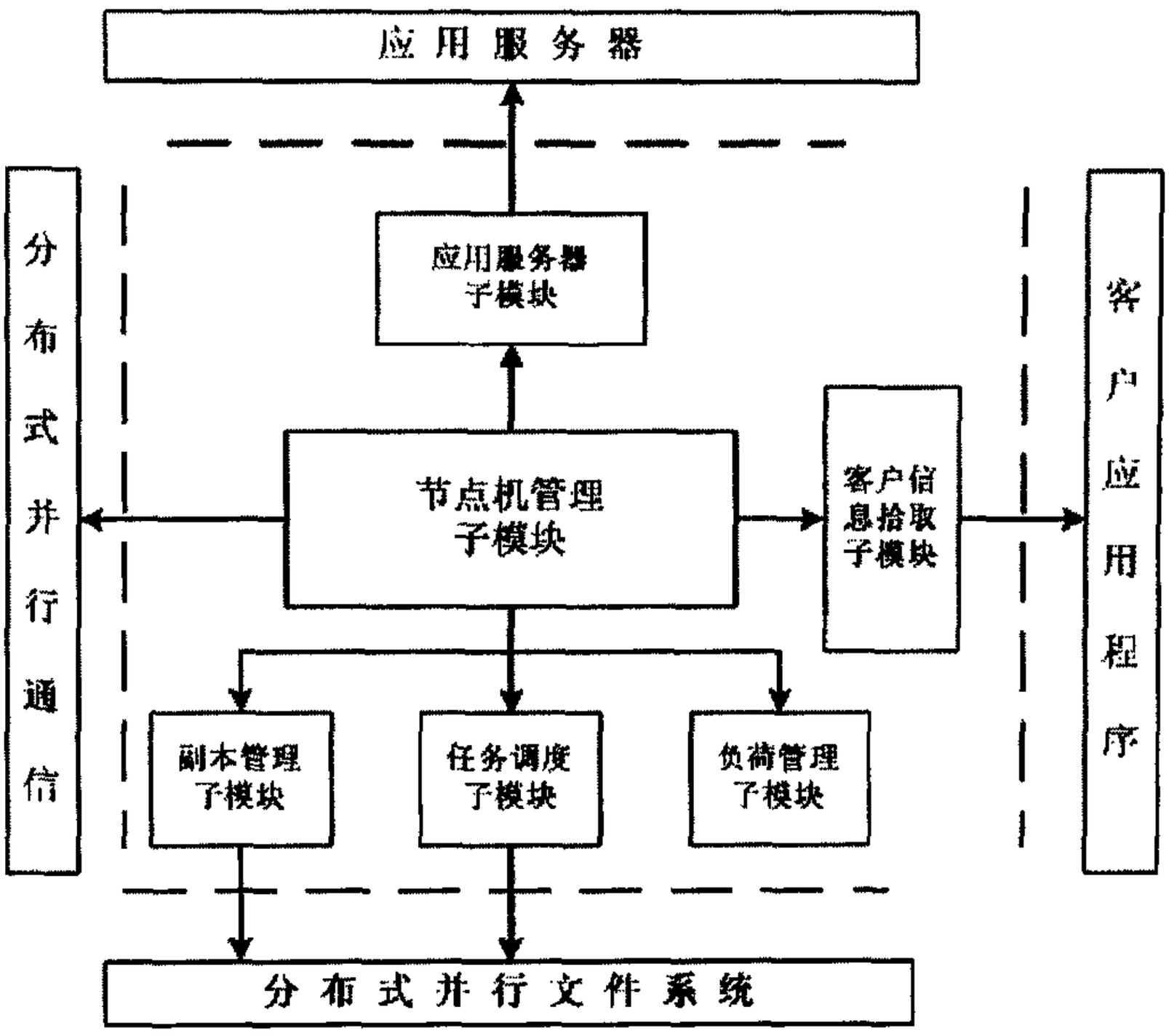

图 4-2 分布式并行调度子系统结构图

从图 4-2 中我们可以看出，分布式并行调度子系统处于分布式并行操作系统的核心地位。它向下与分布式并行文件系统进行交互，调用分布式并行文件系统提供的接口；向上与应用服务器进行交互，向应用服务器提供接

口；另外，它还要调用分布式并行通信提供的接口进行节点机之间的通信；最后，它还要向远程客户应用程序提供接口，以便于客户调用。

整个调度子系统分为六个子模块：节点机管理子模块、应用服务器子模块、客户信息拾取子模块、副本管理子模块、任务调度子模块和负荷管理子模块。在后面的部分，将对这些子模块逐一介绍。

4.2.1 节点机管理子模块

节点机管理子模块是分布式并行调度子系统的核心子模块，其他的子模块都要在它的基础上运行。为了实现调度子系统的高度模块化，各部分均可替换，程序之间必须定义尽可能简洁的接口，尽量独立，以便在需要替换其中某个子模块的同时其他的子模块不需要改动。为了实现高速的响应，应该在每个节点机上收集尽可能多的信息，以便在进行调度决策时能有更多的信息根据。同时为了减少通信开销，每个节点机应尽可能地减少全局信息的维护，因为要对全局信息进行一次同步更新的通信量是非常大的，特别是在节点机数目很大的时候，这个开销是相当惊人的。

4.2.1.1 节点机管理子模块的设计

节点机管理子模块是一个消息收发平台，它接收各类消息，转发给相应的处理程序，同时接收各个处理程序需要发送的消息，把他们发送出去。

在分布式并行调度子系统结构图中可以看到，该子系统要与四个部分交互，即应用服务器、分布式并行通信、客户应用程序和分布式并行文件系统。其中，涉及到通信的有三个部分，他们是应用服务器、分布式并行通信和客户应用程序。节点机管理子模块主要功能就是接收由它们发送的消息，和发送消息给它们。

节点机管理子模块与三个部分进行交互，由于这三个部分与节点机管理子模块的物理位置的不同，因此采用的通信方法也不同。具体来说，节点机管理子模块与客户应用程序的物理位置一般很远，为了保证它们之间的可靠通信，所以采用 TCP 协议进行通信；分布式并行通信子系统的目标是解决节点机之间的高速可靠通信，主要是提供给分布式并行调度子系统所使用，而且由于节点机（这里所说的节点机是指分布式并行服务器）一般所处的物理位置较近，处于同一个子网，因此在节点机之间采用 UDP 协议进行通信；节点机管理子模块与应用服务器的关系比较特殊，它们一般同时处于一台机器上，它们之间的通信就需要采用进程间通信的方法了，进程间通信的方法很多，有消息、信号灯、共享内存等，为了与前面两种通信协议保持一定的统一，我们采用了 UNIX 域协议来实现节点机管理子模块与应用服务器之间的通信。

除了进行消息的收发工作外，节点机管理子模块还要进行一些日常维护工作。这些工作包括定时检查应用服务器是否活动，检查其他节点机是否活动等。整个节点机管理子模块的结构如图 4-3 所示：

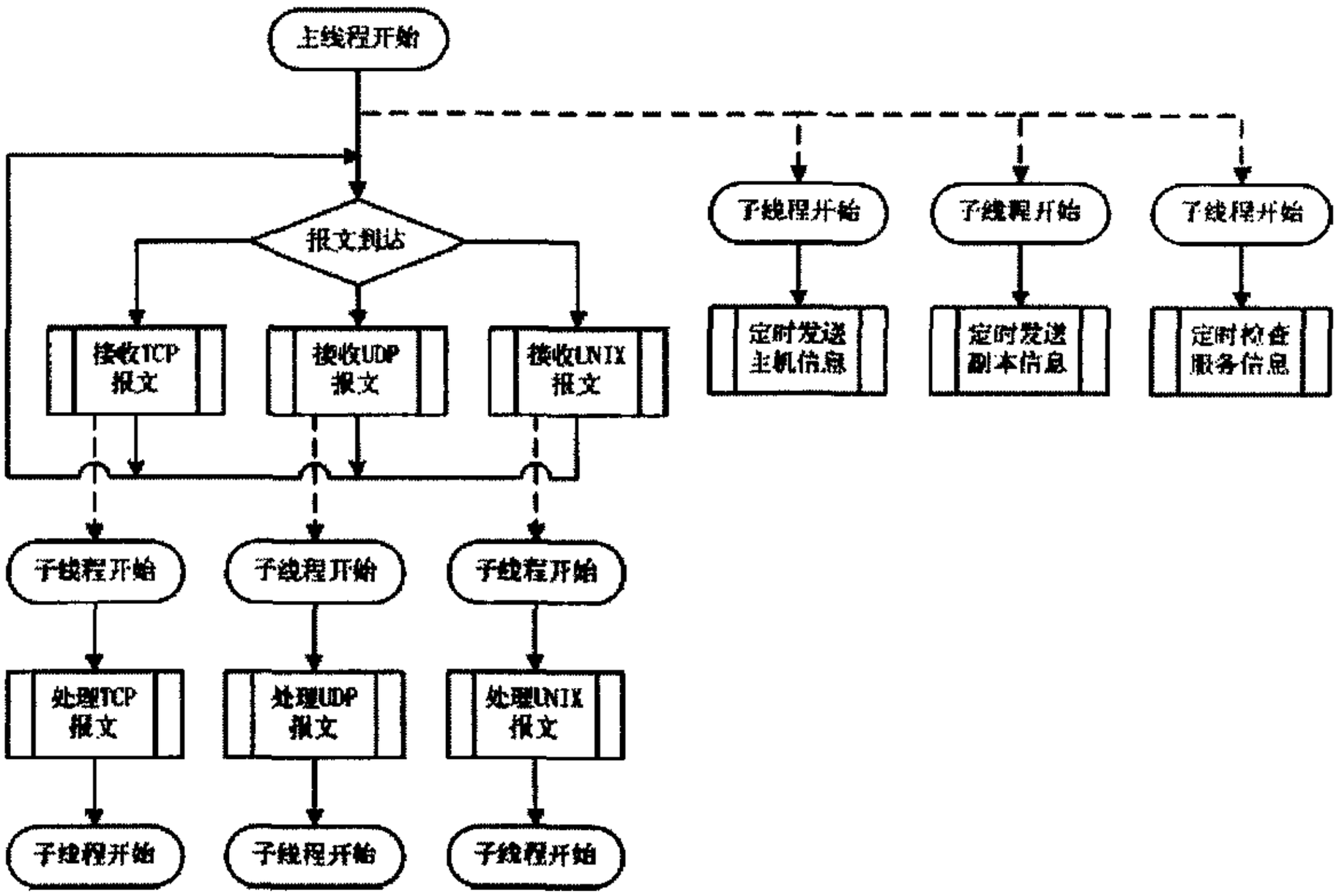

图 4-3 节点机管理子模块结构图

4.2.1.2 节点机管理子模块的实现

(1) 初始化

初始化工作包括两点，一是将整个分布式并行调度子系统初始化为守护进程，二是对所有通信套接字进行初始化，为整个子系统的正常运行奠定基础。

一个守护进程（daemon）是一个在后台运行的进程，它独立于任何的控制终端。例如，如果从一个控制终端启动了一个守护进程，然后退出终端，其他用户又登录这个终端，那么我们不希望在这个用户使用终端期间有任何的守护进程产生的错误信息出现在终端上。相似的，终端产生的信号不会影响到先前在这个终端上启动的守护进程。这就是为什么它必须独立于任何控制终端的原因。

有很多方法可以启动守护进程：

a. 通过系统初始化脚本来启动一个守护进程 这些脚本通常在/etc 目录下，或者在以/etc/rc 名字开始的目录下，但是它们的位置和上下文是独立于实现的。由系统初始化脚本启动的守护进程拥有超级用户的权限。

b. 由 inetd 超级服务器来启动的很多网络服务器也是守护进程 守护进程 inetd 本身是由系统初始化脚本启动的，它监听网络请求（Telnet，FTP 等），当请求到达时，它调用真实的服务器（Telnet 服务器，FTP 服务器等）。

c. 由 cron 守护进程来启动 守护进程 cron 本身是由系统初始化脚本在系统启动时启动的。

d. 由 at 命令说明在将来的某一时刻启动守护进程 守护进程 cron 通常会在时间到达的时候初始化这些程序，所以这些程序就作为守护进程来运行。

e. 从用户终端启动（前台或后台） 这样做的原因是测试守护进程，或者由于一些原因而被终止后需要重新启动守护进程。

综合考虑，我采用了最后一种方法来设计守护进程，下面介绍它的实现细节。

a. 创建一个子进程 首先调用 fork 来创建一个子进程，然后将父进程终止，让子进程继续运行。如果一个进程在前台由 shell 命令启动，当父进程终止时，shell 会认为命令被执行完成了，自动地在后台运行子进程。子进程继承了父进程的进程组 ID，同时拥有自己的进程 ID，这确保了子进程不是一个进程组的首领，它需要进行下一步的处理。

b. 创建一个新会话 子进程调用 setsid 来创建一个新的会话，这样，子进程成为了新会话和新的进程组的首领，它不再拥有控制终端。

c. 忽略 SIGHUP 信号并再次创建子进程 当调用 fork 返回时，第一个子进程终止，让第二个子进程继续运行。第二次调用 fork 的目的是确保守护进程不能自动获得一个控制终端，因为它有可能在将来打开一个终端设备。在 SVR4 中，当一个没有控制终端的会话首领打开一个终端设备时，这个终端就会成为会话首领的控制终端。但是，通过第二次调用 fork 就能确保第二个子进程不再是一个会话的首领，所以它就不能获得一个控制终端。还必须要忽略 SIGHUP 信号，因为当一个会话的首领（第一个子进程）在终止时，它会向属于这个会话的所有进程（包括第二个子进程）发送 SIGHUP 信号。

d. 改变工作目录并清除文件模式创建掩码 要改变工作目录的一个原因是守护进程可以从文件系统的任何地方启动，如果它保持在原来的目录下，那么那个文件系统就不能被卸载。此时，文件模式创建掩码要被设置为 0，

这样如果守护进程创建自己的文件，被继承的文件模式创建掩码中的许可位则不会影响新文件的许可位。

e. 关闭任何打开的描述符 要关闭任何从执行守护进程的进程（通常是shell）中继承的打开的描述符。问题是要确定使用的最大的描述符，系统并没有提供一个函数来获取这个最大值。虽然也有一些方法来确定这个最大值，但是相当复杂，因为限制是无限的。我采用的解决方法是关闭前 64 个描述符，尽管它们大多数没有被打开。

f. 使用日志 调用 openlog 来打开日志，第一个参数是来自调用者，通常是程序名字（例如：argv[0]）。第二个参数设为 LOG_PID，指定要将进程 ID 加入到每一行日志信息中。

经过以上六个步骤的工作，程序就完成了守护进程的初始化工作[5][6]。

初始化的第二步是进行通信套接字的初始化，要生成 TCP、UDP 和 UNIX 域三个套接字，并将它们绑定在熟知的端口上。

对于 TCP 和 UDP 套接字，大家用得非常多，这里就不多说了。而 UNIX 域套接字一般很少使用，这里作一些介绍。

UNIX 域协议并不是一套真正的协议，而是一种在单个主机上执行客户-服务器通信的方法，它使用与在不同的主机之间执行客户-服务器通信相同的 API：套接字或 XTI。当客户和服务器都在同一台主机上的时候，UNIX 域协议是 IPC 方法的一种代替品。

UNIX 域提供两种类型的套接字：流套接字（与 TCP 相似）和数据报套接字（与 UDP 相似），甚至还提供原始套接字。

在 Ipv4 中，使用 32 位地址和 16 位端口号的组合作为它的协议地址；在 Ipv6 中，使用 128 位地址和 16 位端口号的组合作为它的协议地址；而在 UNIX 域中，用来识别客户和服务器的协议地址是普通文件系统中的路径名，这些路径名不是通常的文件，因为我们不能读写它们，除非程序将这些路径名与 UNIX 域套接字联系起来。

UNIX 域套接字的地址结构定义在<sys/un.h>系统头文件中：

```
Struct    sockaddr_un {
  Short int       sun_family;
  Char            sun_path[104];
};
```

成员变量 sun_family 必须设置为 AF_UNIX，成员变量 sun_path 指定文件系统中一个路径名，这个路径名必须以空字符结束。

初始化 UNIX 域套接字的步骤如下：

a. 创建 UNIX 域套接字 程序调用 socket 创建一个 UNIX 域套接字，第一个参数必须设置为 AF_UNIX，第二个参数视其需要可以设置为 SOCK_STREAM 或者 SOCK_DGRAM，我们需要在这个 UNIX 域套接字上进行数据报传输，所以将它设置为 SOCK_DGRAM。

b. 删除路径名 如果路径名已经在文件系统中存在，那么在绑定时会失败。以防万一，不管路径名是否存在，首先调用 unlink 删除要绑定的路径名。如果路径名不存在，unlink 返回错误，忽略它。

c. 填写服务器地址并绑定到套接字上 将上面删除的路径名复制到地址中，然后调用 bind 将地址与套接字绑定在一起。这样就声明了一个为大家熟知的地址，以利于客户进行连接。

这样，UNIX 域套接字的初始化就完毕了，程序可以通过它进行数据报的发送和接收[6]。

(2) 产生日常维护工作的线程

日常的维护工作指不需要程序的干预，在程序启动后就自动进行的一些工作。这些工作主要有三个：检查服务活跃性、发送本地信息和发送副本信息。

由于这些工作不是定时执行的，因此将它们设计为由子线程来执行。一则不受主程序逻辑的制约，二则可以访问主程序的数据结构，三则增加的系统开销很小。主程序调用 pthread_create 来创建三个线程，分别执行以上三个工作如下。

a. 检查服务活跃性是一个非常重要的日常工作 一个服务在进行注册以后就开始正常运行，在运行过程中，服务可能会因为一些意外的事件而中断，在它们中断后，系统要及时知道这些信息，并将它们注销，否则系统会认为它们能继续提供服务，而得到错误的信息。所以，检查这些服务是否活跃是系统必须要做的工作。

服务是否活跃应该有一个标志来确定，子线程通过检查这个标志就可以判断指定的服务活跃与否。服务在运行过程中有自己的状态，这些状态中就包含了我们需要的信息，但是这必须要从内核中得到相关进程的数据。从内核中取得数据，是比较困难的，尤其对应用程序来说。经过细心研究发现，在 Linux 系统中，存在一个 proc 文件系统，它是一个伪文件系统，它只存在内存当中，而不占用外存空间。它以文件系统的方式为访问系统内核数据的操作提供接口。用户和应用程序可以通过 proc 得到系统的信息，并可以改变内核的某些参数[12]。由于系统的信息，如进程，是动态改变的，所以用户或应用程序读取 proc 文件时，proc 文件系统是动态从系统内核读出所

需信息并提交的。除了这些，proc 文件系统还有一些以数字命名的目录，它们是进程目录。系统中当前运行的每一个进程都有对应的一个目录在/proc 下，以进程的 PID 号为目录名，它们是读取进程信息的接口，proc 文件系统的名字就是由之而起[10] [11]。有了这种机制，我们的应用程序就可以通过读普通文件的方式轻松获得进程的有关信息。进程目录的结构如表 4-1 所示。

表 4-1 进程目录结构表

目录名称	目录内容
cmdline	命令行参数
environ	环境变量值
root	连接进程的 root 目录
statm	进程内存状态信息
maps	内存印象
exe	进程的可执行连接
cwd	当前工作目录的连接
status	人可读的进程状态信息
stat	二进制的进程状态信息
mem	内存利用情况
fd	包含所有文件描述符的目录

在这些文件中，我们要读取的是 status，其中包含了可读的进程状态信息，status 文件的前四行如下：

```
Name:   bash            // 进程名字
State:  S (sleeping)    // 进程状态
Pid:    7705            // 进程 ID
PPid:   7704            // 父进程 ID
```

在这里，第二行是关键，它的值说明了进程的状态。一般进程可以是以下几个状态：

```
S (sleeping)            // 睡眠
R (running)             // 运行
Z (zombie)              // 僵尸
```

如果一个进程的状态为 S 或者 R，那么我们就认为这个服务进程是活跃的。反之，如果进程状态不是 S 或者 R，甚至在/proc 下没有服务进程的

PID 目录，那么我们就认为该服务进程已经不是活跃的了，应该发送消息通知主线程从服务表中注销该服务。

检查服务是否活跃的子线程定时执行，对服务表中的所有服务进行检查。

b. 发送本地信息也是一个需要定时执行的日常工作 因为每个节点机为了调度的需要都得在本地维护一定的全局信息，所以节点机之间就要定时交换自己的本地信息，这样才可能获得一定的全局信息。

容易想到的发送信息的方法是广播，因为这样做是最省事的，只需要发送一次信息，所有的节点机都可以接收到。但是我们在实现中并没有采用广播方式，而是采用单点传送进行多次发送，这是有原因的。要了解广播与单点传送的区别，就必须了解主机对由信道传送过来的帧的过滤过程。

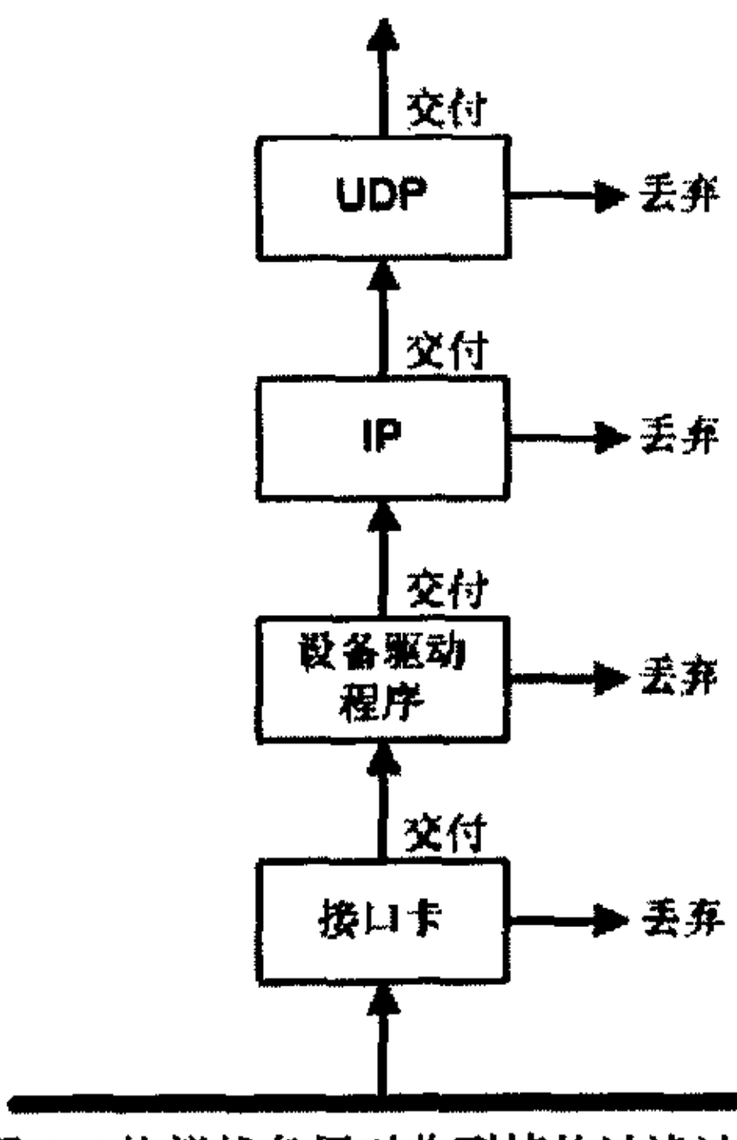

图 4-4 协议栈各层对收到帧的过滤过程

如图 4-4 所示，首先，网卡查看由信道传送过来的帧，确定是否接收该帧，若接收则接收后就将它送往设备驱动程序。通常网卡仅仅接收那些目的地址为网卡物理地址或广播地址的帧。另外，多数接口均被设置为混合模式，这种模式能接收每个帧的一个复制。

如果网卡收到一个帧，这个帧将被传送给设备驱动程序（如果帧校验和错，网卡将丢弃该帧）。设备驱动程序将进行另外的帧过滤。首先，帧类型

中必须指定要使用的协议（IP、ARP 等）。其次，进行多播过滤来检测该主机是否属于多播地址说明的多播组。

设备驱动程序随后将数据帧传送给上一层，比如，当帧类型指定为 IP 数据报时，就传往 IP 层。IP 层根据 IP 地址中的源地址和目的地址进行更多的过滤检测。如果正常，就将数据报传送给上一层（如 TCP 或 UDP）。

每次 UDP 收到由 IP 传送来的数据报，就根据目的端口号，有时还有源端口号进行数据报过滤。如果当前没有进程使用该目的端口号，就丢弃该数据报并产生一个 ICMP 不可到达报文（TCP 根据它的端口号作相似的过滤）。如果 UDP 数据报存在校验和错，将被丢弃。

使用广播的问题在于它增加了对广播数据不感兴趣的主机的处理负荷。拿一个使用 UDP 广播应用作为例子：如果子网内有 50 个主机，但是仅有 20 个参与了该应用，每次这 20 个主机中的一个发送 UDP 广播数据时，其余 30 个主机不得不处理这些广播数据报。一直到 UDP 层，收到的 UDP 广播数据报才会被丢弃。这 30 个主机丢弃 UDP 广播数据报是因为这些主机没有使用这个目的端口。这是一次发送 UDP 数据报的情形，考虑我们的实际情况是要频繁地在主机之间发送 UDP 数据报，如果每一次发送都要出现以上的情况，增加其他主机的负荷，那么会造成资源的极大浪费，这是我们不愿意看到的[8]。

综上所述，发送本地信息我们采用了单点传送的方式发送 UDP 数据报。发送本地信息是定时进行的，采用的方式又是单点传送，所以要确定每次传送的主机数目。这是一个难点问题，因为每次传送的主机数目多少直接影响到主机维护全局信息的效率。如果每次传送的主机数目太少，例如只传送到一台主机，那么必须要在若干个周期以后，才能将本地信息传送给所有的主机，其效率显然是不符合实际需要的。如果每次传送的主机数目过多，甚至是传送给所有的主机，那么在每次传送的时候花费的时间会过长，同样效率低下。应该是一个什么数目比较合适呢？

由于在子模块运行过程中，活动主机的数目是动态变化的，所以每次发送信息的目的主机数目也是动态变化的。在每次发送信息的时候，有两个阈值需要比较，一个是 1 倍发送周期，另一个是 3 倍发送周期。每向一个目的主机发送主机信息，都会收到由它返回的应答，而且因为主机信息在每个发送周期都要发送一次，所以如果在 1 倍发送周期内接收到了一个目的主机返回的应答，就说明已经在 1 倍发送周期内向该目的主机发送过了，就不需要再对它发送一次；如果在 3 倍发送周期时还没有接收到目的主机的应答，就说明该主机已经死亡，这是因为在 3 倍发送周期内已经向该目的主机进行了两次发送，都没有接收到它的应答。这里的两次发送，就是对一些偶然事故

进行的补救措施，就是一种超时处理。只有在 1 倍发送周期到 3 倍发送周期之间还没有接收到应答的目的主机才需要进行发送。不管在每个发送周期时发送的目的主机的数目多少，必须要在 3 倍周期内覆盖到它，因为主机只有在接收到发送信息的前提下才有可能发送应答信息，如果在 3 个周期内没有覆盖到一些主机，那么就有可能错误地认为这些没有覆盖到的主机已经死亡，这是不应该出现的情况。所以系统内所有的活动主机都应该在 3 倍发送周期的时间内覆盖完毕。好了，讨论到这里，我们已经得到了问题的答案，那就是在每个发送周期要发送的目的主机的数目是所有主机数目的 1/3，由于所有的主机数目是动态变化的，我们得到的这个相对数目也是动态变化的。

定时更新算法是节点机管理子模块的核心算法，它的功能是要确定在一次发送主机信息的时候，往哪些主机发送。为此，节点机管理子模块设计了一个称为主机更新索引表的数据结构，这个表里的每一项都是一个类型。定义如下：

```
class host_update {
private:
  unsigned long      IPAddr;      // 主机 IP 地址，网络字节顺序
  time_t             SendTime;    // 上次发送信息给该主机的时间
  time_t             RecvTime;    // 上次接收该主机发送信息的时间
public:
  host_update() {};
  host_update(unsigned long ip, time_t st = 0, time_t rt = 0)
        :IPAddr(ip), SendTime(st), RecvTime(rt) {};
  // 重载 < 符号，对表按照 RecvTime 升序进行排序
  bool operator < (const host_update& hu) const
  { return RecvTime < hu.RecvTime; }
  // 重载 == 符号，用来在表中查找特定的项
  bool operator == (const host_update& hu) const
  { return IPAddr == hu.IPAddr; }
};
vector<host_update>  vector_host_update;
```

主机更新索引表中的项总是按照一定的顺序进行排列，具体来说就是按照主机的接收时间的递增进行排序。每次进行发送的时候，只需要从主机更新索引表中取得最前面的几个主机，向它们发送主机信息就可以了。

算法能确保每次发送的目的主机都是最久没有发送的。这是因为RecvTime 的值是在接收到源主机信息时调用 time 取得的，RecvTime 的值越大表示时间越近。而且不管哪个子模块对主机更新索引表进行了修改，都一定会在修改后调用 STL 的标准算法 sort 对主机更新索引表按照 RecvTime 的升序进行排序，这样在每次发送主机信息的时候，只要取表最前面的主机，就能保证它们是最久没有接收到信息的源主机。如果一直没有接收到该主机发送来的信息，那么该主机就一直排在主机更新索引表的最前面，到下一次发送的时候，还是会往该主机发送主机信息。如果 RecvTime 与当前调用 time 得到值的差值超过 3 个发送周期，我们就可以认为该主机已经死亡，应该从本地保存的全局信息中删除该主机。算法如下：

```
while (1)          // 死循环
{
  for (依次取主机更新索引表前 1/3 的主机)
  {
        if (该主机的 RecvTime 与当前时间的差距 < 一个发送周期)
              // 不必发送，等待下一个周期
              break;
        if (该主机的 RecvTime 与当前时间的差距 > 三个发送周期)
        {
              // 该主机已经死亡
              从本地的全局信息中删除该主机；
              continue;
        }
        发送主机信息更新请求报文给该主机；
        设置该主机的 SendTime 为当前时间；
  }
  sleep(UPD_HOST_TIME);          // 睡眠一个发送周期
};
```

该日常工作发送的主机信息更新请求报文的格式如下：

报文头	报文类型	服务数目	服务信息	主机负载	剩余空间

报文头（1 字节）：标明此报文是本系统专用的，应该设置为 MSG_HEADER；

报文类型（1 字节）：标明此报文是何种报文，这里应该设置为 HOST_INFO_REQ；

服务数目（2 字节）：本主机提供的服务数目；

服务信息（服务数目*27 字节）：本主机提供的服务映射信息；

主机负载（浮点数）：本主机当前的平均负载；

剩余空间（长整数）：本主机的磁盘剩余空间。

c. 发送副本信息是另一个非常重要的日常工作 这个工作牵涉到文件副本的创建、删除，它将定时调用副本管理子模块提供的接口取得主机的副本信息，然后将副本信息发送出去。由于副本信息是一个大概的数字，它不要求非常精确，也不要求所有的节点机随时保持同步，所以发送副本信息的周期可以长一点，例如 5 分钟或者 10 分钟。因为副本信息的发送周期延长了，所以发送副本信息可以采用广播数据报的方式，这样既方便又不会频繁地干扰其他主机接收信息的工作。

(3) 监听套接字并接收、处理套接字上的报文

在初始化工作中，我们已经准备好了 3 个套接字，即 TCP 套接字、UDP 套接字和 UNIX 域套接字，那么就要监听这 3 个套接字，一旦有任何报文到达，就可以立即感知。

监听的方式有很多，循环监听是其中一种。它的原理是不断循环检测 3 个套接字上有无报文到达，如果有，则可以接收、处理报文；如果没有，则继续进行检测。这种方式有一个最大的缺点：CPU 始终处于忙的状态，哪怕套接字上没有任何的报文到达，也要不断地进行检测；还有一点是不能立即感知套接字上到达报文，因为一个报文在套接字上到达时，不一定恰好检测到这个套接字，所以会造成一定的延迟，对并行处理也要造成一些影响。

还有一种监听方式是多路复用。它可以同时监听多个文件描述符，判断它们是否可读、可写或者可执行。一旦其中的一个文件描述符出现上述的某种情况，会立即通知系统，以便于系统进行处理。这种方式不像循环监听，它非常节约系统资源，因为在套接字上没有报文到达的时候，它处于睡眠状态，不占用 CPU 资源；多路复用还可以实现最大的并行度，因为它同时监

听多个套接字，任何一个套接字上有报文到达，系统都会立即感知，并即时进行处理。因此，输入/输出多路复用是一个非常适合的方案。

输入/输出多路复用的具体实现是调用 select 函数，它有 5 个参数，原形如下：

```
int select(int maxfd, fd_set* rdset, fd_set* wrset, fd_set* exset,
        struct timeval* timeout);
```

参数 maxfd 指定测试的描述符的最大值，从 0 到 maxfd-1 的描述符都将被测试；参数 rdset、wrset 和 exset 分别指定要检查的可读描述符集、可写描述符集和异常描述符集；参数 timeout 指定测试的超时时间。有描述符就绪时函数返回就绪的描述符个数，超时时间内没有描述符就绪时返回 0，函数失败时返回-1。

我们只关心套接字上是否有报文到达，也就是套接字是否可读，所以只需要设置一个可读描述符集，将 3 个套接字描述符放入其中，而设置可写描述符集和异常描述符集为空。在没有报文到达时，我们希望程序永远阻塞，进入睡眠状态，所以超时时间设置为空。

当有报文到达时，根据不同的套接字，有 3 种不同的处理方法。如果是 TCP 套接字上有连接请求，立即接受请求，生成一个连接套接字，并启动一个子线程来接收、处理连接套接字上的报文，主线程继续执行 select，等待下一个报文到达。如果是 UDP 套接字或 UNIX 域套接字上有报文到达，立即启动一个子线程来接收、处理套接字上的报文，主线程继续执行 select，等待下一个报文到达。算法如下：

```
for (;;)   // 死循环
{
  设置可读描述符集;
  调用 select;  // 阻塞，睡眠
  if (TCP 套接字可读)
  {
        // 请求 TCP 连接
        accept;        // 生成一个连接套接字
        启动子线程在生成的连接套接字上接收、处理报文;
}
if (UDP 套接字可读)
{
```

```
        // UDP 套接字上有报文到达
        启动子线程在 UDP 套接字上接收、处理报文;
}
if (UNIX 域套接字可读)
{
        // UNIX 套接字上有报文到达
        启动子线程在 UNIX 套接字上接收、处理报文;
    }
}
```

下面分别讨论 3 种不同的处理方法。

a. TCP 处理方法

当 TCP 套接字可读时，只是说明有 TCP 的连接请求到达，并不是有真正的 TCP 报文到达。这时，在 TCP 套接字的 listen 队列中就存在一个连接请求，服务器调用 accept 响应这个请求，生成一个连接好的 TCP 套接字，在这个套接字上就可以接收、发送 TCP 报文了。主线程启动一个子线程，将这个新生成的连接好的 TCP 套接字通过堆变量传递给子线程，主线程则继续调用 select，进入睡眠状态，等待下一个报文的到达。

子线程启动后，首先将堆变量保存在局部变量中，把父线程申请的堆变量释放，这样既不会造成内存的泄露，又保证了该子线程对堆变量独占。

然后，子线程要做一步比较重要的工作，就是调用 pthread_detach 来将自己与父线程分开。一个线程要么是 joinable，要么是 detach。当一个 joinable 线程终止时，它的线程 ID 和退出状态将一直保留，直至其他的线程调用了 pthread_join。但是，一个 detach 线程就像一个守护进程一样，当它终止时，所有的资源都要被释放并且不能等待它终止。之所以要做这个工作，是因为处理 TCP 报文的线程在完成工作后是异步退出的，其他线程不能等待它终止。

这些准备工作完成以后，子线程就接收 TCP 套接字上到达的报文。将 TCP 报文处理完成之后，关闭该 TCP 套接字。最后，子线程退出。

TCP 报文是服务器与远程客户应用程序进行通信所使用的报文，服务器在 TCP 套接字上接收远程客户应用程序发送来的更新请求报文，报文的格式如下：

报文头	报文类型	服务名字-端口映射版本号	服务-服务器映射版本号

报文头（1 字节）：标明此报文是本系统专用的，应该设置为 MSG_HEADER；

报文类型（1 字节）：标明此报文是何种报文，这里应该设置为 CLI_UPD_REQ；

服务名字-端口映射版本号（4 字节）：这个版本号是客户本地的信息版本号；

服务-服务器映射版本号（4 字节）：这个版本号是客户本地的信息版本号。

服务器接收到远程客户应用程序发送来的 TCP 报文，在判断报文是合法之后，首先发送服务器地址列表应答报文给远程客户应用程序，报文的格式如下：

报文头	报文类型	信息长度	服务器地址列表信息

报文头（1 字节）：标明此报文是本系统专用的，应该设置为 MSG_HEADER；

报文类型（1 字节）：标明此报文是何种报文，这里应该设置为 CLI_HOST_UPD_ACK；

信息长度（2 字节）：说明后面的信息长度；

服务器地址列表信息（最大 64K 字节）。

接下来，服务器将要发送服务名字-端口映射信息应答报文给远程客户应用程序，但是在发送之前，必须要判断远程客户端的服务名字-端口映射信息版本号与服务器维护的服务名字-端口映射信息版本号是否一致，如果一致，说明服务器的服务名字-端口映射信息从上次发送给远程客户应用程序以来，没有改动，不需要再次发送，直接发送下一个报文。如果不一致，则说明服务器的服务名字-端口映射信息已经更新了，因此必须发送给远程客户应用程序，更新远程客户。服务名字-端口映射信息应答报文的格式如下：

报文头	报文类型	信息长度	服务名字-端口映射信息版本号	服务名字-端口映射信息

报文头（1 字节）：标明此报文是本系统专用的，应该设置为 MSG_HEADER；

报文类型（1 字节）：标明此报文是何种报文，这里应该设置为 CLI_NAME_SERVICE_UPD_ACK；

信息长度（2字节）：说明后面的信息长度（不包括版本号），如果长度为0，则表示服务名字-端口映射信息不需要更新，报文中就没有后面的信息了；

服务名字-端口映射版本号（4字节）：这个版本号是服务器端的信息版本号；

服务名字-端口映射信息（最大64K字节）。

服务器要发送的第三个应答报文是服务-服务器映射信息应答报文，但是在发送之前，必须要判断远程客户端的服务-服务器映射信息版本号与服务器维护的服务-服务器映射信息版本号是否一致，如果一致，说明服务器的服务-服务器映射信息从上次发送给远程客户应用程序以来，没有改动，不需要再次发送，直接发送下一个报文。如果不一致，则说明服务器的服务-服务器映射信息已经更新了，因此必须发送给远程客户应用程序，进行更新。服务-服务器映射信息应答报文的格式如下：

报文头	报文类型	信息长度	服务-服务器映射信息版本号	服务-服务器映射信息

报文头（1字节）：标明此报文是本系统专用的，应该设置为MSG_HEADER；

报文类型（1字节）：标明此报文是何种报文，这里应该设置为CLI_SERVICE_HOST_UPD_ACK；

信息长度（2字节）：说明后面的信息长度（不包括版本号），如果长度为0，则表示服务-服务器映射信息不需要更新，报文中就没有后面的信息了；

服务-服务器映射版本号（4字节）：这个版本号是服务器端的信息版本号；

服务-服务器映射信息（最大64K字节）。

b. UDP处理方法

当UDP套接字可读时，说明有UDP报文到达。这时，主线程立即启动一个子线程，将UDP套接字通过堆变量传递给子线程，主线程则继续调用select，进入睡眠状态，等待下一个报文的到达。

子线程启动后，首先将堆变量保存在局部变量中，把父线程申请的堆变量释放，这样既不会造成内存的泄露，又保证了该子线程对堆变量独占。

然后，子线程要做一步比较重要的工作，就是调用pthread_detach来将自己与父线程分开。

这些准备工作完成以后，子线程就接收 UDP 套接字上到达的报文。将 UDP 报文处理完成之后，子线程退出。

UDP 报文是服务器与服务器之间进行通信所使用的报文，服务器首先在 UDP 套接字上接收其他服务器发送来的更新请求报文，报文的格式如下：

报文头	报文类型	信息内容

报文头（1 字节）：标明此报文是本系统专用的，应该设置为 MSG_HEADER；

报文类型（1 字节）：标明此报文是何种报文，这里设置有 3 种类型，如果报文是主机信息请求报文，则设置为 HOST_INFO_REQ；如果报文是主机信息应答报文，则设置为 HOST_INFO_ACK；如果报文是副本信息请求报文，则设置为 REPETITION_REQ；

信息内容：根据报文类型有不同的内容。

服务器接收到 UDP 报文后，根据报文的类型进行相应的处理，算法如下：

```
if (主机信息应答报文)
{
   在主机更新索引表中查找源主机；
   if (存在源主机)
         将源主机的 RecvTime 设置为当前时间；
   else
         在主机更新索引表中添加源主机，并设置 SendTime 和
         RecvTime 为当前时间；
   更新全局服务名字-端口映射表；
   更新主机负载表；
   更新服务-服务器映射表；
}
if (主机信息请求报文)
{
   在主机更新索引表中查找源主机；
   if (源主机不存在)
   {
```

```
            发送应答报文给源主机;
            在主机更新索引表中添加源主机，并设置 SendTime 和
            RecvTime 为当前时间;
        }
        if (源主机的 SendTime 与当前时间的差值大于 1 倍主机信息发送周期)
        {
            发送应答报文给源主机;
            将源主机的 RecvTime 和 SendTime 设置为当前时间;
        }
        if (源主机的 SendTime 与当前时间的差值小于 1 倍主机信息发送周期)
        {
            将源主机的 RecvTime 设置为当前时间;
        }
        调用 STL 的标准算法 sort 对主机更新索引表进行排序;
        更新全局服务名字-端口映射表;
        更新主机负载表;
        更新服务-服务器映射表;
          更新两个信息版本号;
        }
        if (副本信息请求报文)
        {
          调用副本管理子模块提供的接口处理该报文;
        }
```

因为算法表达得比较简洁，所以在这里要对算法做一些说明。当接收到主机信息应答报文时，应该在主机更新索引表中查找源主机，如果查找到了，则说明源主机是在接收到本机发送给它的请求报文后返回一个应答报文，对这种情况，直接在主机更新索引表中修改源主机的 RecvTime；如果没有查找到该源主机，则说明它是在接收到本机发送的广播数据报后返回一个应答报文，在这种情况下，本机就已经感知到系统内有源主机的存在，就必须要在主机更新索引表中添加源主机，并设置源主机的 RecvTime 和 SendTime 为当前时间。当接收到主机信息请求报文时，也要在主机更新索引表中查找源主机，如果没有查找到，则说明源主机是一个新机器，就必须

要在主机更新索引表中添加源主机，并设置源主机的 RecvTime 和 SendTime 为当前时间；如果查找到该源主机，就需要进一步判断源主机的 SendTime 与当前时间的差值，只有在这个差值大于 1 倍的主机信息发送周期的情况下（即上一次发送的主机信息已经过时），才会发送应答报文，然后将源主机的 RecvTime 和 SendTime 设置为当前时间。接下来，要调用标准算法中的 sort 对整个主机更新索引表按照 RecvTime 的升序进行一次排序，给以后的处理提供正确的信息。然后要更新全局服务名字-端口映射表、主机负载表、服务-服务器映射表以及两个信息版本号。当接收到副本信息请求报文时，就需要调用副本管理子模块提供的一个接口来处理这个报文。

c. UNIX 处理方法

当 UNIX 域套接字可读时，说明有 UNIX 报文到达。这时，主线程立即启动一个子线程，将 UNIX 域套接字通过堆变量传递给子线程，主线程则继续调用 select，进入睡眠状态，等待下一个报文的到达。

子线程启动后，首先将堆变量保存在局部变量中，把父线程申请的堆变量释放，这样既不会造成内存的泄露，又保证了该子线程对堆变量独占。

然后，子线程要做一步比较重要的工作，就是调用 pthread_detach 来将自己与父线程分开。

这些准备工作完成以后，子线程就接收 UNIX 域套接字上到达的报文。将 UNIX 报文处理完成之后，子线程退出。

UNIX 报文是分布式并行调度子系统与应用服务器之间进行通信所使用的报文，分布式并行调度子系统首先在 UNIX 域套接字上接收应用服务器发送来的报文，报文的格式如下：

报文头	报文类型	信息内容

报文头（1 字节）：标明此报文是本系统专用的，应该设置为 MSG_HEADER；

报文类型（1 字节）：标明此报文是何种报文，这里设置有 2 种类型，如果报文是服务注册请求报文，则设置为 SERV_LOG_REQ；如果报文是服务调度请求报文，则设置为 SERV_SCHED_REQ；

信息内容：根据报文类型有不同的内容，如果是服务注册请求报文，则格式如下：

操作类型	服务端口	服务名字	应用服务器进程号

操作类型（1 字节）：标明服务注册的具体操作，如果是注册，则设置为 LOGIN；如果是注销，则设置为 LOGOUT；

服务端口（2字节）：标明要操作的服务端口号；

服务名字（25字节）：标明要操作的服务名字；

应用服务器进程号（4字节）；

如果报文是服务调度请求报文，则格式如下：

调度类型	调度数据

调度类型（枚举类型）：标明应用服务器要如何调度分布式并行调度子系统；

调度数据（根据调度类型的不同有不同的长度）：要调度的信息内容。

分布式并行调度子系统在接收到UDP报文后，根据报文的类型进行相应的处理，算法如下：

```
if (是服务注册请求报文)
{
  if (是注册)
  {
        在本地服务名字-端口映射表中查找要注册的服务；
        if (存在该服务)
        {
              将服务的进程号设置为注册者的进程号；
              结束线程；
        }
        将该服务添加入本地服务名字-端口映射表中；
        将本机添加入服务-服务器映射表中；
        将该服务添加入全局服务名字-端口映射表中；
        结束线程；
  }
  if (是注销)
  {
        从本地服务名字-端口映射表中删除该服务；
        从服务-服务器映射表中删除本机；
        从全局服务名字-端口映射表中删除该服务；
  }
```

```
}
if (是服务调度请求报文)
{
   调用任务调度子模块提供的接口进行处理;
}
```

节点机管理子模块是分布式并行调度子系统的核心模块，它要负责创建、维护一些共享资源，这些共享资源在子系统的范围内提供给所有子模块所共享，除了前面提到的主机更新索引表外，还有以下的一些数据结构，在这些表中，我们采用了 STL 提供的 vector 来实现动态表，它有许多优点，例如可以动态改变表的长度，可以对它使用 STL 的标准算法[4]。

a. 本地服务名字-端口映射表

```
class name_service {
private:
   char              ServiceName[NAME_MAX_LEN];
   unsigned short    ServicePort;
   pid_t             ProcID;
public:
   name_service()
{ memset((void*)ServiceName, 0, NAME_MAX_LEN); }
name_service(char* name, unsigned short port = 0, pid_t pid = 0)
:ServicePort(port), ProcID(pid)
{
     memset((void*)ServiceName, 0, NAME_MAX_LEN);
     strncpy(ServiceName, name, NAME_MAX_LEN);
}
// 重载 == 符号，根据服务端口号进行查找
bool operator==(const name_service& ns) const
{ return (!strcmp(ServiceName, ns.ServiceName))
     || (ServicePort == ns.ServicePort); }
};
vector<name_service>        vector_local_name_service;
```

b. 全局服务名字-端口映射表

```
vector<name_service>        vector_global_name_service;
```

c. 服务-服务器映射表

```
class service_host {
private:
   unsigned short              ServicePort;
   vector<unsigned long>       vector_IPAddr;
public:
   service_host(){}
   service_host(unsigned short port):ServicePort(port){}
   //重载 == 符号，根据服务端口号进行查找
   bool operator==(const service_host& sh) const
   { return ServicePort == sh.ServicePort; }
};
vector<service_host>           vector_service_host;
```

d. 主机负载表

```
class host_load {
private:
   unsigned long        IPAddr;
   float                LoadInfo;
   long                 FreeDisk;
public:
   host_load(){};
   host_load(unsigned long ip):IPAddr(ip){};
   //重载 == 符号，根据主机 IP 地址进行查找
   bool operator==(const host_load& hl) const
   { return IPAddr == hl.IPAddr; }
};
    vector<host_load>      vector_host_load;
```

所有这些共享资源都要被分布式并行调用子系统内的所有子模块所共享，所以必须要针对它们设计相应的互斥锁。各个子模块在使用它们的时候，首先要将它们加锁，在使用完后，要将它们解锁。如果在对共享资源加

锁的时候，有其他的子模块在使用它，那么就会阻塞，直至其他子模块使用完毕解锁，才可以加锁使用。

4.2.2 应用服务器子模块

应用服务器子模块是分布式并行调度子系统与应用服务器之间进行交互的一个接口。应用服务器通过该子模块进行两种类型的工作，一是向分布式并行调度子系统注册自己，二是根据特定的调度信息从分布式并行调度子系统得到一台合适的主机 IP 地址。

4.2.2.1 应用服务器子模块的设计

根据应用服务器子模块的功能，应该提供 2 个接口。一个接口是服务注册，另一个接口是服务调度。应用服务器子模块的结构如图 4-5：

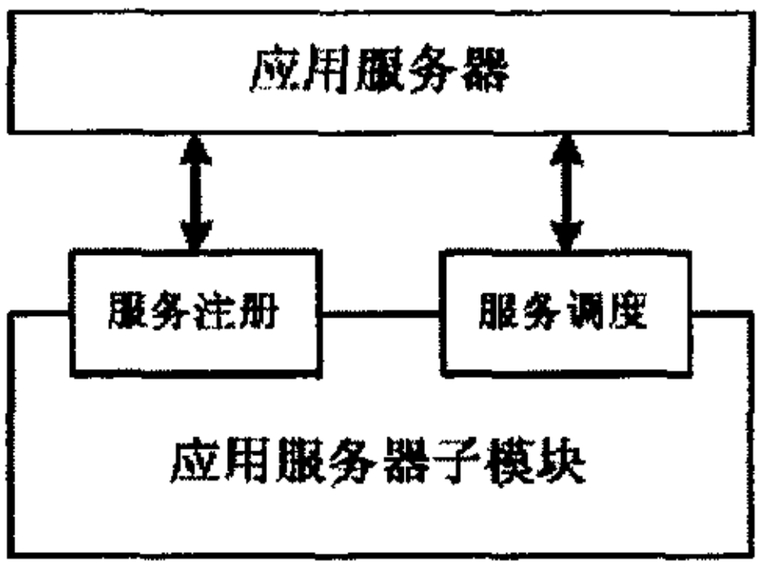

图 4-5 应用服务器子模块结构

由于应用服务器与分布式并行调度子系统在同一台主机上，所以它们之间的通信应该使用进程间通信的方法。我们具体采用了 UNIX 域套接字的方法来传递 UNIX 报文。

服务注册接口将应用服务器需要注册的服务信息，即服务名字、服务端口号以及应用服务器的进程号传递给分布式并行调度子系统，由它进行登记注册。这个接口分为两种操作，一是注册，二是注销。

服务调度接口将应用服务器指定的调度类型和调度信息传递给分布式并行调度子系统，由它进行调度，在调度完毕后，返回给应用服务器一台合适的主机 IP 地址。

尽管应用服务器和分布式并行调度子系统之间采用的是 UNIX 域套接字通信方式，但是这种通信与一般的 TCP 和 UDP 套接字通信不一样。UNIX 域套接字通信的双方都存在于一台主机上，可靠性高，因此它不用进行一些通信的差错控制等工作，效率就非常高。另外，采用 UNIX 域套接字通信

方式，与 TCP 和 UDP 套接字通信方式能够达到形式上的一致，便于统一处理。

4.2.2.2 应用服务器子模块的实现

(1) 服务注册

服务注册接口提供给应用服务器向分布式并行调度子系统登记注册自己，原型如下：

int RegisterService(char* name, unsigned short port, char op, pid_t pid)

参数：name 为服务名字；port 为服务端口号；op 为操作类型，可以取两个值，LOGIN 为注册，LOGOUT 为注销；pid 为应用服务器的进程号。

返回值：函数调用成功返回 0，失败返回-1。

该接口接收应用服务器传进的参数，首先构造一个 UNIX 报文，报文的格式如下：

报文头	报文类型	操作类型	服务端口号	服务名字	进程号

报文头（1 字节）：标明此报文是本系统专用的，应该设置为 MSG_HEADER；

报文类型（1 字节）：标明此报文是何种报文，这里设置为 SERV_LOG_REQ；

操作类型（1 字节）：标明作何种操作，可以取 LOGIN 或 LOGOUT；

服务端口号（2 字节）：要注册的服务端口号；

服务名字（25 字节）：要注册的服务名字；

进程号（4 字节）：应用服务器自身的进程号。

在构造完 UNIX 报文后，生成一个 UNIX 域套接字，然后通过这个套接字将报文发送到分布式并行调度子系统监听的 UNIX 域套接字上去就完成了服务注册工作。

(2) 服务调度

服务调度接口是实现动态调度的关键接口，应用服务器通过这个接口向分布式并行调度子系统传递调度的信息，分布式并行调度子系统经过调度，得到一个合适的主机 IP 地址，然后把这个 IP 地址返回给应用服务器。服务调度接口原型如下：

unsigned long FindServer(enum ScheduleType st, void* info, int len)

参数：st 为调度类型；info 为调度数据的首地址；len 为调度数据的长度。

返回值：函数调用成功返回合适的主机 IP 地址，失败返回 0。

该接口接收应用服务器传进的参数，首先构造一个 UNIX 报文，报文的格式如下：

报文头	报文类型	调度类型	调度数据

报文头（1 字节）：标明此报文是本系统专用的，应该设置为 MSG_HEADER；

报文类型（1 字节）：标明此报文是何种报文，这里设置为 SERV_SCHED_REQ；

调度类型（4 字节）：标明了采用何种方式进行调度；

调度数据（调度类型指定）：调度信息。

调度类型是一个枚举变量，当前只设计了两种调度类型：缺省调度类型和文件调度类型，当然，随着系统的扩展，也可以扩展该类型，使之具有更强大的功能。

enum ScheduleType {ST_DEFAULT, ST_FILE};

每种调度类型分别对应于不同的调度数据结构，缺省调度类型的结构为

```
struct sched_default {
  unsigned short      ServicePort;
  unsigned long       IPAddr;
};
```

文件调度类型的结构为

```
struct sched_file {
  unsigned short      ServicePort;
  unsigned long       IPAddr;
  char                FileName[1024];
};
```

两种调度结构中都有一项为 IPAddr，这一项要求应用服务器填写不可用的主机 IP 地址。因为在有些情况下，分布式并行调度子系统觉得一台主机可用，但是应用服务器却觉得这个主机不可用，它不希望分布式并行调度

子系统返回这个主机给它，它就可以将这个主机的 IP 地址填入调度结构，那么分布式并行调度子系统在进行调度时就会避免返回这个 IP 地址给它。

有一点特殊的是，分布式并行调度子系统在进行调度以后，会返回给应用服务器一个主机 IP 地址。不像 UDP 的客户，当使用 UNIX 域数据报协议时必须要明确地将一个路径名绑定到套接字上，这样服务器才能有路径名来发送应答。所以该接口要做一些特殊的操作。

声明一个本地 UNIX 域地址 struct sockaddr_un cliaddr，然后填写地址信息：

strcpy(cliaddr.sun_path, tmpnam(NULL));

调用 tmpnam(NULL)来分配一个唯一的路径名，接着将该地址绑定在创建的套接字上：

bind(sockfd, (struct sockaddr*)&cliaddr, sizeof(cliaddr));

这样，该接口在发送报文的时候就通知接收方自己的地址信息，接收方就可以根据对方的地址信息来返回调度的结果。

该接口要取分布式并行调度子系统返回的结果，在接收报文时由于采用的是阻塞接收，所以在有些情况下可能造成永久阻塞，与死机的情况一样。为了避免出现这样的情况，设置超时，时间一到，不管有无接收到结果报文，都要强制返回。

4.2.3 客户信息拾取子模块

客户信息拾取子模块是分布式并行调度子系统与远程客户进行交互的一个接口。客户应用程序向该子模块提供一个服务类型的标识，该子模块从本地维护的信息中选择一台适合的服务器的地址，将这个地址返回给客户应用程序。而不需要现场从远端获取服务器信息，这样就能节省通信开销，提高效率。

4.2.3.1 客户信息拾取子模块的设计

根据需求，客户信息拾取子模块应该提供给客户应用程序三类接口：初始化、取服务器地址和取服务器地址列表。其中，初始化接口是单向的，而其他两类是双向的。子模块结构如图 4-6 所示。

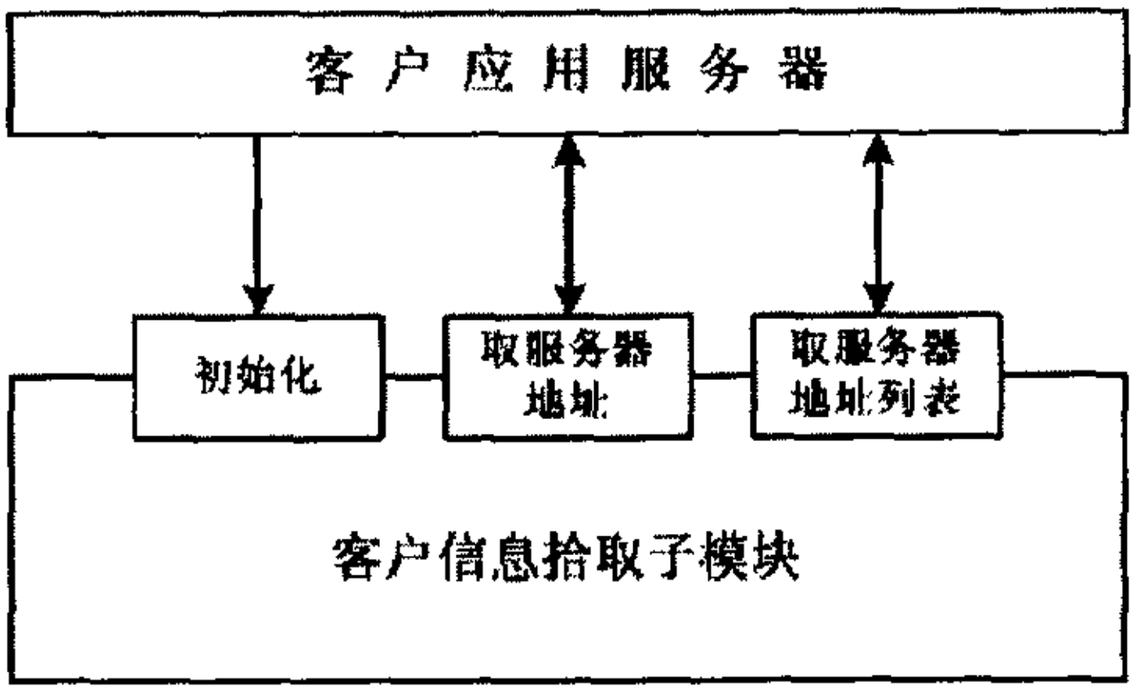

图 4-6 客户信息拾取子模块结构

初始化工作应该在客户应用程序启动的时候进行，它的主要功能是从远程服务器端下载最新的信息来更新本地存储的服务器信息。由于这个工作可能耗费的时间比较长，因此应该设计为一个后台执行的任务。初始化接口由客户应用程序单向调用，它立即启动一个线程，然后退出。被启动的线程开始与远程的服务器进行通信，更新本地信息。

取服务器地址接口接收客户应用程序提供的请求服务的标识，它根据这个服务标识，在提供这种服务的服务器中选择一台适合的服务器，将这台服务器的 IP 地址返回给客户应用程序。在这里，服务标识分为两种：服务名字和服务端口号。因此，取服务器地址接口相应分为两个版本。这个接口是从本地存储的服务器信息中进行拾取，所以执行效率非常高。

和上述接口类似，取服务器地址列表接口也是接收客户应用程序提供的请求服务的标识，它根据这个服务标识，选择所有提供这种服务的服务器，将它们的 IP 地址全部返回给客户应用程序。同样，服务标识分为两种：服务名字和服务端口号。因此，取服务器地址列表接口同样分为两个版本。这个接口也是从本地存储的服务器信息中进行拾取，所以执行效率非常高。

客户信息拾取子模块驻留在客户机上，而客户机一般是使用 Windows 系列的操作系统，所以该子模块将把接口作为动态链接库的形式提供给客户应用程序[13]。

4.2.3.2 客户信息拾取子模块的实现

(1) 信息本地存储

客户信息拾取子模块要求尽可能快速地根据客户应用程序提供的服务标识返回一个服务器的 IP 地址。在这种要求下，不可能让客户应用程序在调用接口时，现场从远程服务器端获取信息。这样会造成时间上的浪费，影响客户应用程序的效率。

鉴于这个原因，所以要在本地存储一份服务器信息的副本。

服务器的信息主要有以下内容：服务器的 IP 地址列表、服务名字-端口映射列表、服务-服务器映射列表和服务器信息版本号。

服务器的 IP 地址列表信息存在于一个文本文件中，具体为 C:\ServerList。这个文件是用来存储活动的可以连接的服务器 IP 地址，文件的每一行只有一个服务器的 IP 地址，IP 地址的格式为点分十进制。以“#”符号开始的行为注释行，例如，

```
# 服务器 IP 地址列表
192.168.0.1
192.168.0.2
192.168.0.3
```

服务名字-端口映射列表信息存在于一个文本文件中，具体为 C:\ServiceName。这个文件存储的内容是服务的名字与端口的映射关系，用来将客户提供的服务名字解析为对应的服务端口号。文件的每一行只有两项，第一项为服务名字，类型是长度为 25 个字符的字符串；第二项为该服务的端口号，是小于 65536 的正整数。它们之间由空格符或者制表符分隔，以“#”符号开始的行为注释行，例如，

```
# 服务名字-端口映射列表
WWW         80
FTP         21
DPSQL       3306
```

服务-服务器映射列表信息存在于一个文本文件中，具体为 C:\ServiceHost。这个文件是用来查找某一个特定的服务，它存在于哪些服务器上的信息。文件的每一行只有两项，第一项为服务的端口号，是小于 65536 的正整数；第二项为提供该服务的服务器的 IP 地址列表，类型是长度可变的字符串，格式为逗号分隔的点分十进制的 IP 地址。它们之间由空格符或者制表符分隔，以“#”符号开始的行为注释行，例如，

```
# 服务-服务器映射列表
80      192.168.0.13，192.168.0.25
21      192.168.0.8，192.168.0.2，192.168.0.31
3306    192.168.0.13，192.168.0.25，192.168.0.2
```

服务器信息版本号存在于一个文本文件中，具体为 C:\ServiceVer。这个文件是用来记录当前本地保存的服务信息的版本号的，这个版本号与远程服务器端的版本号是一一对应的，它表示了服务器信息的新旧程度。文件的每一行只有一个版本号，一共只有两个版本号。第一个是服务名字-端口映射列表信息的版本号，第二个是服务-服务器映射列表信息的版本号，它们的格式均为正整数。以“#”符号开始的行为注释行，例如，

```
# 服务器信息版本号
# 服务名字-端口映射列表信息的版本号
15
# 服务-服务器映射列表信息的版本号
22
```

(2) 初始化

考虑到客户应用程序从启动到调用取服务器地址接口之间有一段初始化的工作要做，本子模块也提供一个初始化接口，做一些初始化工作。该接口的主要工作是产生一个子线程后马上返回，然后由子线程来更新本地信息。客户程序应该在启动时调用此接口，这样做的原因是等到调用取服务器地址接口的时候，本地信息已经与远程服务器上的信息同步了，这时取出的服务器地址是最新的。初始化接口的原型为

```
void InitLocalInfo(void);
```

初始化接口的流程图如图 4-7 所示，首先本地必须至少要存储一台可用的服务器的 IP 地址，否则初始化接口将不能连接远程服务器进行信息更新工作。然后要取得本地信息的版本号，这个版本号将要被发送到远程服务器端，与远程服务器保存的版本号进行比较，以此来决定客户端是否有必要从远程服务器端下载更新信息。如果不能取得这个版本号，则可以说明本地没有保存任何服务器的信息，于是将版本号设置为 0，发送到远程服务器端，强制下载服务器信息进行更新。接着，初始化接口开始连接远程服务器，如果所有知道的服务器均不能连接，则退出；否则向第一台连接上的服务器发送请求报文，然后依次接收服务器地址列表信息、服务-服务器映射信息、服务名字-端口映射信息。最后，用接收的信息来更新本地保存的信息。

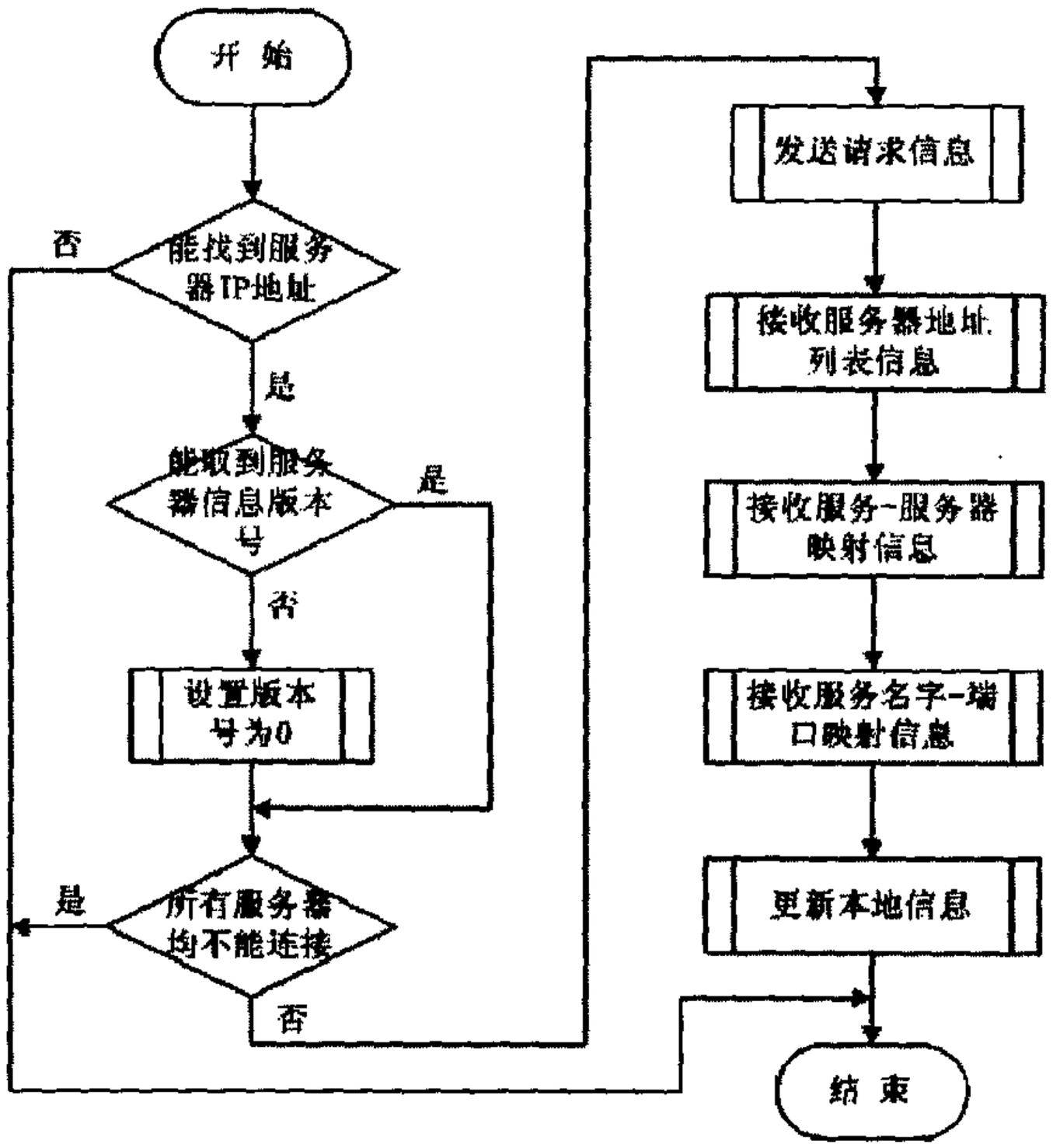

图 4-7 客户端初始化接口流程图

初始化接口发送的更新请求报文是通过 TCP 连接发送给服务器的，目的是更新本地的名字服务信息与服务主机信息，格式如下：

报文头	报文类型	服务名字-端口映射版本号	服务-服务器映射版本号

报文头（1 字节）：标明此报文是本系统专用的，应该设置为 MSG_HEADER；

报文类型（1 字节）：标明此报文是何种报文，这里应该设置为 CLI_UPD_REQ；

服务名字-端口映射版本号（4 字节）：这个版本号是客户本地的信息版本号；

服务-服务器映射版本号（4 字节）：这个版本号是客户本地的信息版本号；。

在发送更新请求报文后，初始化接口就等待接收三个服务器的应答报文。第一个应答报文是服务器地址列表应答报文，格式如下：

报文头	报文类型	信息长度	服务器地址列表信息

报文头（1 字节）：标明此报文是本系统专用的，应该设置为 MSG_HEADER；

报文类型（1 字节）：标明此报文是何种报文，这里应该设置为 CLI_HOST_UPD_ACK；

信息长度（2 字节）：说明后面的信息长度；

服务器地址列表信息（最大 64K 字节）。

初始化接口接收的第二个应答报文是服务名字-端口映射信息应答报文，格式如下：

报文头	报文类型	信息长度	服务名字-端口映射信息版本号	服务名字-端口映射信息

报文头（1 字节）：标明此报文是本系统专用的，应该设置为 MSG_HEADER；

报文类型（1 字节）：标明此报文是何种报文，这里应该设置为 CLI_NAME_SERVICE_UPD_ACK；

信息长度（2 字节）：说明后面的信息长度（不包括版本号），如果长度为 0，则表示服务名字-端口映射信息不需要更新，报文中就没有后面的信息了；

服务名字-端口映射版本号（4 字节）：这个版本号是服务器端的信息版本号；

服务名字-端口映射信息（最大 64K 字节）。

初始化接口接收的第三个应答报文是服务-服务器映射信息应答报文，格式如下：

报文头	报文类型	信息长度	服务-服务器映射信息版本号	服务-服务器映射信息

报文头（1 字节）：标明此报文是本系统专用的，应该设置为 MSG_HEADER；

报文类型（1 字节）：标明此报文是何种报文，这里应该设置为 CLI_SERVICE_HOST_UPD_ACK；

信息长度（2字节）：说明后面的信息长度（不包括版本号），如果长度为0，则表示服务-服务器映射信息不需要更新，报文中就没有后面的信息了；

服务-服务器映射版本号（4字节）：这个版本号是服务器端的信息版本号；

服务-服务器映射信息（最大64K字节）。

在接收三个报文之后，初始化接口要开始更新本地保存的服务器信息，以达到和远程服务器上的信息保持同步。更新过程主要包括两个步骤，一是解析报文内的信息，二是将解析后的信息写入文件中。下面对各个报文中的信息所表示的意义进行介绍。

服务器地址列表应答报文中的信息格式如下：

服务器地址1	服务器地址2	……

每个服务器IP地址大小为4字节，类型为无符号的长整数，IP地址格式为网络字节顺序。

服务名字-端口映射信息应答报文中的信息格式如下：

服务名字1	服务端口1	服务名字2	服务端口2	……

每条服务名字-端口映射信息分为两部分，第一部分为服务名字，大小为25字节，类型为字符串；第二部分为服务端口，大小为2字节，类型为无符号的短整数。后面接着是下一条服务名字-端口映射信息。

服务-服务器映射信息应答报文中的信息格式如下：

服务端口	服务器数目	服务器地址1	服务器地址2	……

每条服务-服务器映射信息分为三部分，第一部分为服务端口，大小为2字节，类型为无符号的短整数；第二部分为服务器数目，大小为1字节，类型为字符，表明提供此类服务的服务器数目；第三部分为服务器的地址列表，每个服务器地址大小为4字节，类型为无符号的长整数，IP地址格式为网络字节顺序，服务器地址的数目由前一部分的值指出。后面接着是下一条服务-服务器映射信息。

(3) 取服务器地址

客户应用程序调用客户信息拾取子模块的最终目的是要获得一台可用的服务器地址，因此，本子模块提供了实现这一功能的接口。另外，由于客户应用程序在传入的参数上有一定的灵活性，所以客户信息拾取子模块提供了

两个版本的接口，以适应客户应用程序的不同需要。取服务器地址接口原形如下：

bool GetServerIPByPort(unsigned short ServicePort,

unsigned long* IPAddr);

参数：ServicePort 为服务端口号；IPAddr 输入时指向不可用的服务器地址，输出时指向返回服务器地址的指针。

返回值：函数调用成功返回 1，失败返回 0。

bool GetServerIPByName(const char* ServiceName,

unsigned long* IPAddr);

参数：ServiceName 为服务名字；IPAddrIPAddr 输入时指向不可用的服务器地址，输出时指向返回服务器地址的指针。

返回值：函数调用成功返回 1，失败返回 0。

取服务器地址的两个版本的接口执行流程如图 4-8 所示：

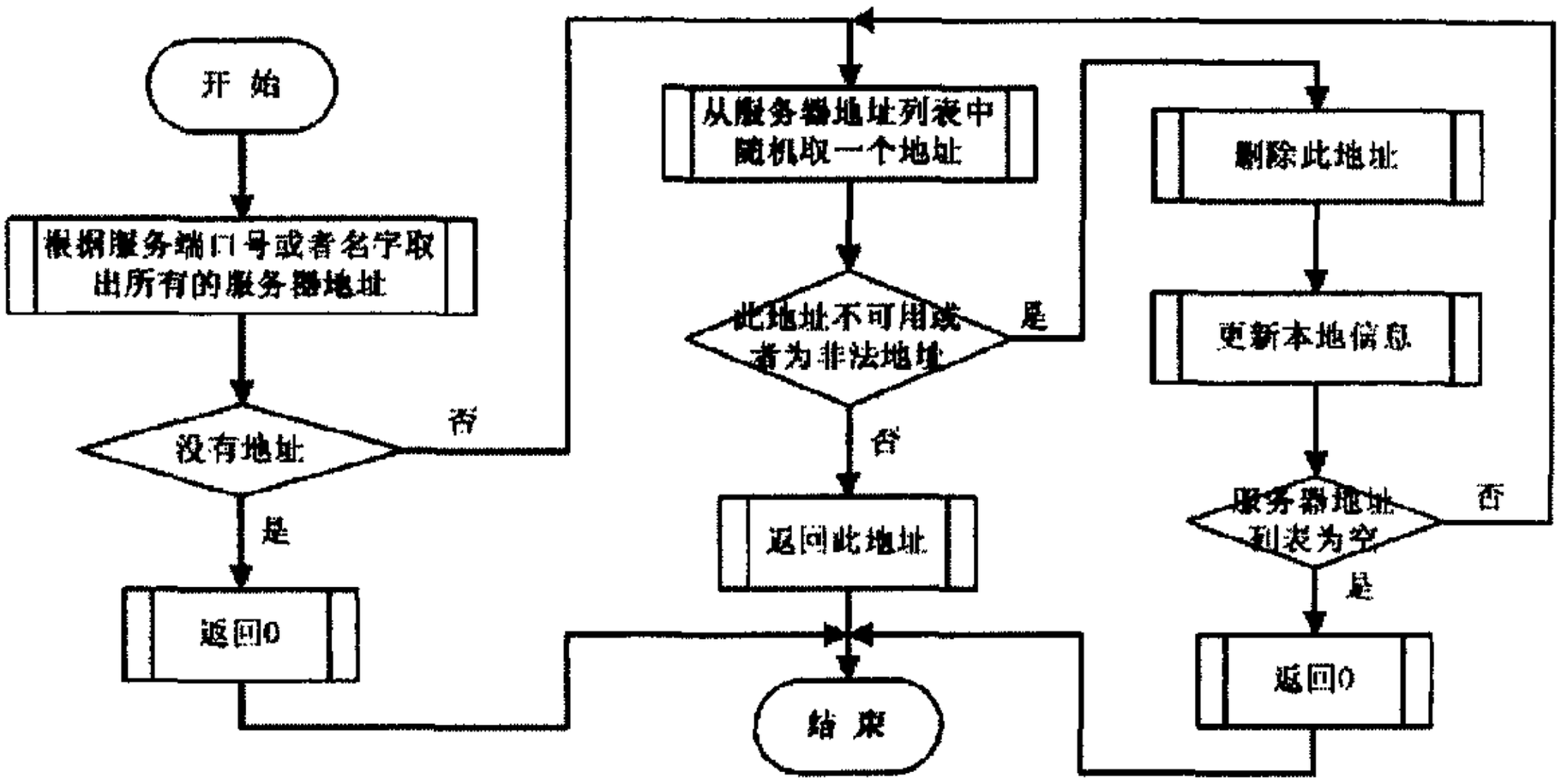

图 4-8 客户端取服务器地址接口流程图

该接口首先取出所有提供指定服务的服务器地址，然后随机选择一个地址，如果选择的地址恰好是不可用的，则删除它。在随机取下一个地址之前，要执行一个非常重要的步骤，那就是强制更新本地存储的服务器信息。因为这时候发现了一个不可用的服务器地址，说明本地存储的服务器信息已经过时了，再在这个基础上进行选择出的服务器地址是过时的可能性非常大。虽然有一个程序在后台定时更新本地存储的服务器信息，但是更新总有一个频度。如果在上一次更新之后，服务器状态立刻出现了改变，客户端应

用程序发现服务器不可用，于是将这个信息通知该接口。该接口这时可以有两种选择：一是立刻强制更新本地存储的服务器信息；二是什么也不做，等待下一次后台程序的定时更新。两种选择带来的后果是不一样的，前者虽然花费一些时间进行更新，但是它的好处是使本地信息重新与服务器保持一致，为以后的客户应用服务提供可信赖的服务器地址。而后者虽然可以从剩下的可用服务器中挑选一个，且不花费时间更新，但是它不能保证现在本地信息中有多少是可以信赖的，从而增加了以后客户应用程序选择失败的可能性。综合两者的利弊，我们认为前者的处理方法是可取的，随着宽带网络的发展，它的缺点将不再是问题。

(4) 取服务器地址列表

有时候，客户应用程序需要调用客户信息拾取子模块得到所有的可用的服务器地址，因此，本子模块提供了实现这一功能的接口。同样，由于客户应用程序在传入的参数上有一定的灵活性，所以客户信息拾取子模块提供了两个版本的接口，以适应客户应用程序的不同需要。取服务器地址列表接口原形如下：

bool GetServersIPByPort(unsigned short ServicePort,

unsigned long* IPList, int* IPNum);

参数：*ServicePort* 为服务端口号；*IPList* 为指向返回服务器地址列表的指针，IPNum 为指向存放服务器数目的地址。

返回值：函数调用成功返回 1，失败返回 0。

bool GetServersIPByName(const char* ServiceName,

unsigned long* IPList, int* IPNum);

参数：ServiceName 为服务名字；IPList 为指向返回服务器地址列表的指针，IPNum 为指向存放服务器数目的地址。

返回值：函数调用成功返回 1，失败返回 0。

取服务器地址列表接口的流程相对来说要比取一个服务器地址接口简单一些，因为它不处理服务器地址是否可用这些信息，只是将存储在本地的服务器信息取出来，返回给客户应用程序。

4.2.4 副本管理子模块

副本管理子模块的功能主要是根据调度信息自动增加或者删除文件的副本，它与整个系统的效率高低有非常大的关系。

假设某台主机上有一个文件，这个文件在整个系统中是唯一的。当多个用户访问这个文件时，这台主机就要同时为这些用户提供服务。如果用户的

数量非常大，那么这台主机就力不从心了。在这个时候，副本管理子模块就要根据用户的调度信息自动在其他的主机上创建新的副本，以分担主机的负载，从而提高了整个系统的效率。一段时间后，用户对这个文件的访问频度下降，当下降到一定的程度后，副本管理子模块又能够自动地删除这个文件的副本，以节约主机的磁盘空间。

4.2.4.1 副本管理子模块的设计

每访问一次文件，就将该文件的副本记录信息中的访问次数加 1，而后为了准确反映文件当前的副本信息，又周期性地将副本访问次数减少，每一个周期（如一天为一个周期）执行一次统计工作，每次统计将副本文件记录信息表中的各个记录访问次数按权值降低，时间越早的副本信息权值越低。

每个副本信息记录了最近若干个周期的访问次数，根据权值计算综合访问次数，如果综合访问次数大于阈值，则创建副本。

副本管理子模块分别与节点机管理子模块和任务调度子模块都有接口，结构如图 4-9：

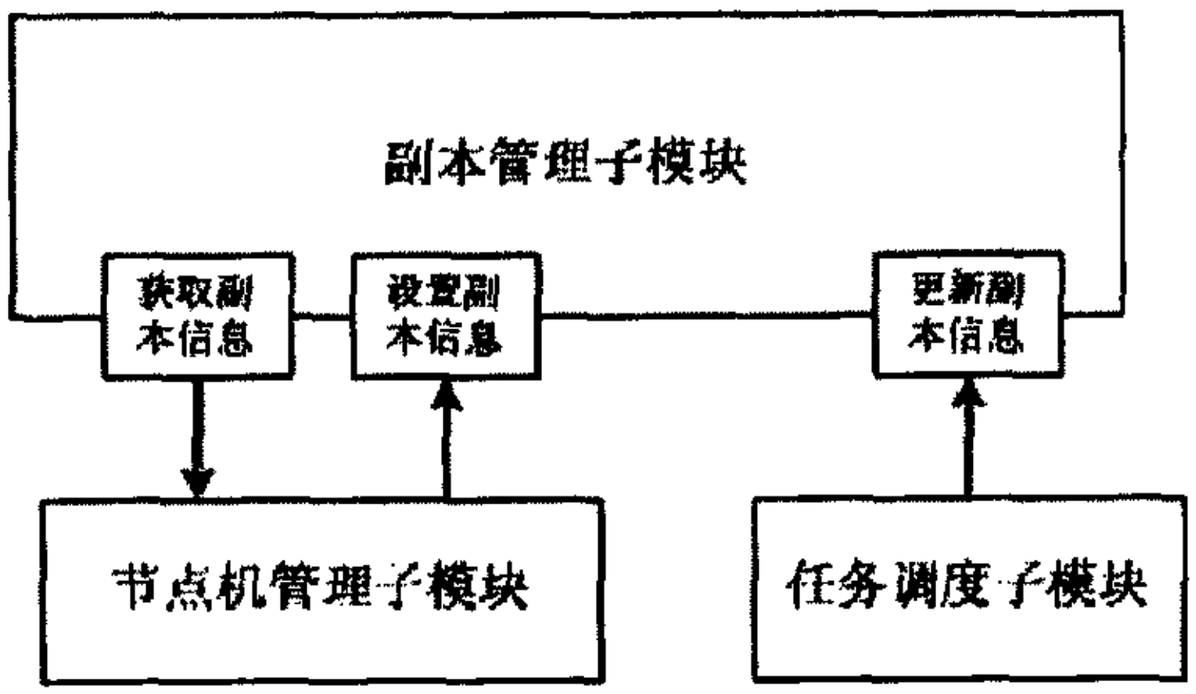

图 4-9 副本管理子模块结构图

4.2.4.2 副本管理子模块的实现

文件副本信息在内存中以哈希表形式进行存储，每个被访问过的文件用一个节点表示，节点的数据结构如下：

```
struct    repinfo {
  char*  filename;      // 访问的文件名
  int      recentnum[10];      // 最近 10 个周期的访问次数
  struct repinfo*        next;  // 下一个节点
};
```

文件副本信息在内存中存储的同时也要记录在文件中，在文件中的存放格式和在内存中略有差别，数据结构如下：

```
struct    repinfo_file {
  int     rec_len;        // 记录长度
  int     recentnum[10];      // 最近 10 个周期的访问次数
  char    filename[256];      // 访问的文件名
};
```

副本管理子模块首先提供了一个更新副本信息的接口，原型如下：

```
int update_file_info(char *filename, unsigned long *ipgroup,
                                    int *repnum)
```

参数：filename 为访问的文件名；ipgroup 为活动主机 IP 地址列表首地址；repnum 为文件副本数目的指针；

返回值：成功返回 0，且 ipgroup 中存储存在副本的主机 IP 地址；失败返回-1。

该接口的算法如下：

```
if (副本表中不存在该文件)
  在表中新建一个节点，节点的访问次数为 1；
else
  将该节点的访问次数加 1；
计算该文件的综合访问次数；
if (综合访问次数大于阈值)
{
  if (找到没有副本的主机)
        在该主机上创建一个副本；
}
将该文件访问信息填入更新链表；
查询副本表，将有副本的主机 IP 地址和副本数填写入函数传递的参数；
```

该接口的传入参数 ipgroup 指向的活动主机列表在传入函数之前就有具体内容，它包括了系统中的所有活动主机。这些活动主机的排列顺序有一定的规则，提供指定服务的主机排列在表的前部，这样在创建副本时将优先在

提供服务的主机上创建。算法中计算文件的综合访问次数是一个加权计算公式：

$$文件综合访问次数=\sum_{i=0}^{9}(10-i)a_i$$

算法只涉及到最近 10 个周期（i 从 0 至 9，当前周期为 0）的访问次数，a_i 为指定周期内的文件访问次数，(10-i)为指定周期的权值，周期越近，权值越大。通过这个公式计算出来的文件综合访问次数能够反映出最近对文件的访问频度的大小，能够满足我们的需要。

阈值是经过我们在实际的环境中测量出来的。经过我们的测算，一台主机的磁盘随机 I/O 速度不超过 2MB/s，而一个用户对文件的访问速度将不小于 500Kb/s，那么一台主机只能同时支持 30 个左右的用户对文件的访问。所以如果在一台主机上对一个文件的综合访问次数超过 30，那么就应该对该文件创建副本。

在前面节点机管理子模块中曾经谈到，节点机管理子模块要定时发送主机的文件访问信息。在发送前就要通过一个接口来获取主机的文件访问信息，这个接口就是由副本管理子模块提供的第二个接口，原型如下：

char* get_new_file_info(int* buffer_len);

参数：buffer_len 是指向副本信息长度的指针；

返回值：成功返回副本信息的首地址，失败返回空。

当节点机管理子模块接收到该文件访问信息时，还要调用副本管理子模块提供的另一个接口来更新本地的副本表，以保持副本表的全局一致性。该接口原型如下：

int update_new_file_info(char* new_block);

参数：new_block 为接收到的文件访问信息首地址；

返回值：成功为 0，失败为-1。

副本管理子模块的设计和实现为整个系统的负载均衡起到了非常重要的作用，它将同时访问一台或几台主机上文件的大量请求自动分布到多台主机上，以达到将负载分流的目的，提高整个系统的性能。

4.2.5 任务调度子模块

任务调度子模块是动态调度的核心，该子模块主要目的是为了提供调度接口。调用者提供一个调度类型和一个调度信息的数据结构，该子模块根据这些信息返回给它一个合适的服务器 IP 地址。

4.2.5.1 任务调度子模块的设计

在节点机管理子模块接收到用户的调度请求时，要调用任务调度子模块提供的接口进行动态调度，得到一个合适的主机 IP 地址后，再返回给用户。

任务调度子模块包含了不同类型的调度算法，根据节点机管理子模块传递来的调度类型和调度数据来选择适当的调度算法，结构如图 4-10 所示。

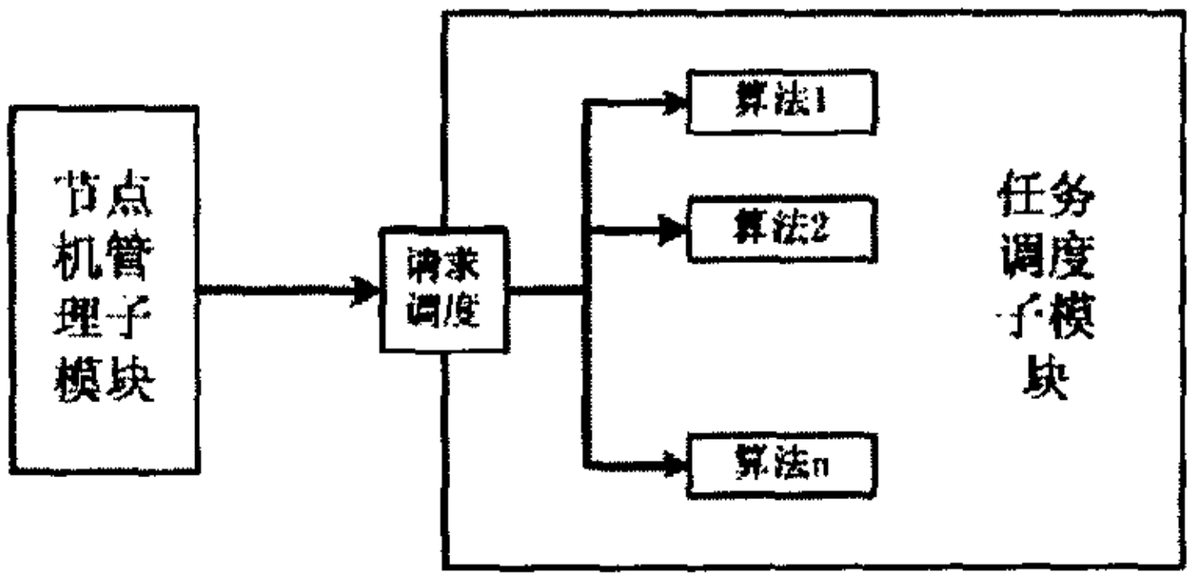

图 4-10 任务调度子模块结构图

4.2.5.2 任务调度子模块的实现

由于节点机管理子模块负责接收用户的调度请求，它并不知道调度数据的细节，所以它只需要将调度数据直接传递给任务调度子模块提供的接口就可以了。这样还有一个好处，就是使子模块之间相互独立，减小它们的相关性。任务调度子模块提供的接口原型如下：

unsigned long request_sched(const char* buf)

参数：调度数据字符串；

返回值：成功返回无符号长整数，代表一个合适的主机 IP 地址；失败返回 0。

不同用户有不同的调度要求，针对这点我们的设计能够实现多种调度方案。用户只需要提供调度的类型和相应的调度数据，就能够获得正确的结果。调度的类型和相应的数据结构是由本子模块与调度的用户双方共同协议商定的，随着用户要求的增多，系统的扩展，可以增加更多的调度类型。

当前的实现中，我们提供了两种调度类型和相应的数据结构，它们是缺省调度和文件调度，实现如下：

缺省调度：ST_DEFAULT

struct sched_default {

```
  unsigned short      ServicePort;
  unsigned long       IPAddr;
};
```

文件调度：ST_FILE

```
struct sched_file {
  unsigned short      ServicePort;
  unsigned long       IPAddr;
  char                FileName[1024];
};
```

接口 request_sched 如同一个开关，根据调度类型进行不同的动态调度，算法如下：

首先从报文缓冲区中取得调度类型和调度数据；

if (调度类型为 ST_DEFAULT)

  将调度数据传递给缺省调度函数；

if (调度类型为 ST_FILE)

  将调度数据传递给文件调度函数；

返回主机 IP 地址；

如果需要扩展系统规模，增加调度的类型，只需要增加新的调度标识，在上述算法中增加对该标识的判断，还需要增加新的处理函数。

缺省调度函数的算法如下：

取得要求的服务端口；

if (服务-服务器映射表中不存在该服务)

  返回 0；

if (该服务映射的服务器中有些不可用)

{

  删除该服务器；

  发送主机信息到该服务器，更新它；

}

if (该服务映射的服务器中包括本地服务器并且本地服务器的负载在限定范围内)

  返回本地服务器 IP 地址；

else

返回服务器中负载最小的服务器 IP 地址；

文件调度函数的算法如下：

调用分布式并行文件子系统提供的接口 update_file_info 得到存储有指定文件的主机列表；

if (没有主机存储有该文件)

返回 0；

取得要求服务映射的所有服务器列表；

if (该服务映射的服务器中有些不可用)

{

删除该服务器；

发送主机信息到该服务器，更新它；

}

取服务映射的服务器与存储有文件的主机之间的交集；

if (该交集中包括本地服务器并且本地服务器的负载在限定范围内)

返回本地服务器 IP 地址；

else

返回服务器中负载最小的服务器 IP 地址；

任务调度子模块是分布式并行调度子系统非常重要的一个子模块，它决定了动态调度的策略和效果。根据子模块可以自由替换的设计要求，任务调度子模块的独立度很高，提供的接口很清晰，便于在扩展系统规模时进行修改和替换。

4.2.6 负荷管理子模块

负荷管理子模块主要是获取本地主机的负荷参数，以此作为动态调度的依据。

4.2.6.1 负荷管理子模块的设计

之所以将负荷管理模块独立出来，是因为评估机器的负荷是一个涉及方面很宽的工作，它的影响因素很多，如任务队列的长度，内存使用率，网络使用率，浮点运算能力等。对于不同的策略来说，我们关注的因素也不同。

在这种情况下，就有必要将负荷管理独立出来，使它在不同的策略中可以被替换。

4.2.6.2 负荷管理子模块的实现

因为负荷的表现是多种多样的，但是我们又需要一个统一的负荷来进行动态调度，所以在本实现中，我们采用过去 1 分钟内节点机的平均负荷作为节点机的负荷。

取主机的平均负荷作为调度的依据是因为平均负荷是系统综合了各方面的负荷而得到的有说服力的指数，对于偏重不同的负荷指标的调度都有一定满足度。而取 1 分钟内的平均负荷则是最能体现系统当前的负荷水平的。

本子模块提供了一个取得主机 1 分钟内的平均负荷的接口，原型如下：

float getload(void);

参数：无；

返回值：成功返回 1 分钟内的平均负荷；失败返回 0。

取系统的平均负荷也是一项比较困难的工作，因为这些参数都是内核参数，应用无法得到。所幸，系统有一个/proc 文件系统，它将一些内核参数以文件的形式输出，这样普通应用也能够使用了。

系统的平均负荷存在于/proc/loadavg 文件中，文件的格式如下：

浮点数 1　　浮点数 2　　浮点数 3 ……

文件只有一行，最前面的 3 个浮点数代表了系统在最近 1 分钟，5 分钟，15 分钟内的平均负荷[9]，因此接口只需要读取该文件的第一行的第一个浮点数就可以了。

在需要更改取系统负荷的方法时，只改变算法，不必改变接口的原型。这样就能使负荷管理子模块做到最大限度的独立，系统一部分的改变并不影响到整体的改变。

4.3 本章小结

本章主要介绍了分布式并行调度子系统的设计和实现，对每个子模块的结构，功能都作了详细的说明，对今后的修改和替换奠定了基础。

第五章 系统性能分析

一个系统的性能可以用很多指标来体现，对于分布式并行调度子系统来说，最重要的指标是响应时间和负载均衡，以下我们就对这两个方面对分布式并行调度子系统来进行分析。

5.1 响应时间

响应时间指的是客户从发起调度到获得一个可用的主机 IP 地址的时间。响应时间的长短可以衡量分布式并行调度的优劣，时间越短越好。

测试响应时间的环境假定是 4 个客户应用程序，其中两个为缺省调度，记为缺省 1，缺省 2；两个为文件调度，记为文件 1，文件 2；3 台主机组成的服务器系统，每个客户应用程序均发起 5 个调度请求，测试结果如表 5-1 所示。

表 5-1 响应时间测试结果表（单位：ms）

	第 1 次	第 2 次	第 3 次	第 4 次	第 5 次
缺省 1	1.686	1.669	1.614	1.681	1.602
缺省 2	1.677	1.654	1.638	1.669	1.666
文件 1	657.984	665.257	706.118	672.363	688.590
文件 2	678.901	689.340	697.630	722.151	701.463

从测试结果可以看出：缺省调度的响应时间在 1 毫秒级别，文件调度由于需要与文件子系统进行交互，花费时间应该比缺省调度要长，响应时间在 100 毫秒级别。从这两种调度的结果来看，分布式并行调度子系统的性能还是比较好的。

5.2 负载均衡

负载均衡指的是提供服务的主机不能始终集中于某一台主机，而是随机分布在各个主机上。

测试负载均衡的环境是由 30 台客户机同时向 2、3、4 台主机发起调度请求，观察最终由哪台主机提供服务。测试结果如表 5-2 所示。

表 5-2 负载均衡测试结果表

主机 1	主机 2
11 台客户机	19 台客户机

主机 1	主机 2	主机 3
8 台客户机	10 台客户机	12 台客户机

主机 1	主机 2	主机 3	主机 4
8 台客户机	8 台客户机	8 台客户机	6 台客户机

从测试结果可以看出：4 台主机的负载均衡效果好于 3 台主机，3 台主机的负载均衡效果好于 2 台主机。于是我们可以推测：主机的数量越大，负载达到均衡的效果就越明显。这说明我们预期的设计目标已经达到了。

5.3 本章小结

根据对分布式并行调度子系统的性能分析可以看出，系统的响应时间和负载均衡都比较好，特别是在节点机数目很大的情况下，负载均衡的效果更好。

第六章 结束语

任务调度是操作系统中一个非常重要的子系统，针对传统的集中式调度子系统的缺点，我们开发成功了分布式并行调度子系统。本文论述了分布式并行调度的技术，分析了现有的若干调度算法，以发送者发起的启发性算法为基础，设计并实现了新型的分布式并行调度算法。

本课题为实验室奠定分布式并行调度子系统平台的目的已基本达到，成为实验室分布式并行操作系统的组成部分。

通过测试数据分析，分布式并行调度子系统在响应时间和负载均衡两个方面都达到了很好的性能，完全达到了设计时的指标。

但是，该子系统仍有一些有待改进的地方，需要后来者继续作不懈的努力。由于分布式并行调度子系统的设计原则是高度模块化，其中任意一个模块都可以进行修改和替换，这为后来者在该子系统上进一步开发提供了方便。

参考文献

（1） Adrzej Goscinski. Distributed Operating System: The Logical Design, Addison-Wesley Publisher Ltd, 1991.

（2） Andrew S. Tanenbaum. Distributed Operating Systems, 1995.

（3） Andrew S. Tanenbaum. Modern Operating Systems, 1992.

（4） Tome Swan. Tom Swan's GNU C++ for Linux, 2000.

（5） Paul DuBois. MySQL 网络数据库指南, 2000.

（6） W. Richard Stevens. UNIX Network Programming Volum 1, 1998.

（7） W. Richard Stevens. UNIX 环境高级编程，机械工业出版社，2002.

（8） W. Richard Stevens. TCP/IP 详解，卷 1：协议，机械工业出版社，2000.

（9） Scott Maxwell. Linux 内核源代码分析，机械工业出版社，2000.

（10） 李善平、郑扣根，Linux 操作系统及实验教程，机械工业出版社，1999.

（11） 周巍松，Linux 系统分析与高级编程技术，机械工业出版社，1999.

（12） Alessandro Rubini. Linux 设备驱动程序，中国电力出版社，2000.

（13） David Bennett. Visual C++ 5 开发人员指南，机械工业出版社，1998.

致 谢

首先，感谢导师刘心松教授给我指明了课题的方向，使我有机会深入学习 Linux 操作系统的内核结构，进一步研究了分布式并行操作系统中任务调度的设计与实现。在毕业设计和论文的撰写过程中，他还给我提出了很多好的建议，提供了优越的科研开发环境。在近三年的学习研究期间，他以渊博的学识、丰富的经验和勤勉严谨的治学态度深深地感染了我，让我终生获益。在此，我向刘老师致以最真诚的感谢。

此外，我要向在论文期间给予我帮助的王玉英老师表示深深感谢，她对我的学习和工作给予了很大的支持和帮助。在与 8010 研究室的同学们共事期间，大家一起讨论，解决难题，相互支持鼓励，它们给予了我很大的帮助，使我少走了很多弯路，谢谢大家的支持与帮助。

最后，我要向我的父母和家人致以崇高的敬意，正是他们毫无保留的支持才使我得以顺利地完成学业。

感谢所有关心和帮助我的老师、同学、朋友们！

个人简历

出生日期：1974 年 6 月 14 日

学位情况：于 1997 年 6 月在北京航空航天大学获得学士学位

攻读硕士学位期间作为骨干参与的科研项目有：

1.分布式并行调度系统

2.分布式数据库系统

3.分布式 VOD 系统

4.基于宽带网的分布式数据广播系统

攻读硕士学位期间发表的论文有：

《Linux 消息通信系统的分析》（《电脑技术信息》2000 年第 7 期）

攻读硕士学位期间获得的奖励有：

研究生二等奖学金一次