第 10 章　分布式文件系统

10.1　简　　介

从 70 年代以来，计算机网络在整个计算机工业中掀起了一场革命。随着网络的发展，人们迫切地需要在不同的计算机之间共享文件。最早的解决方法局限于将整个文件从一个机器复制到另外一个机器上，比如 UNIX 到 UNIX 复制(uucp)[Nowi 90]和文件传输协议(ftp)[Post 85]。然而这些解决方案还没有达到如同访问本机磁盘那样访问远程机器上的文件的要求。

80 年代中期出现了一些允许通过网络访问远程主机上的文件进行透明访问的分布式文件系统。主要有 Sun 的网络文件系统(NFS)[Sand 85a]，AT&T 的远程文件系统(RFS)[Rifk 86]和卡内基·梅隆大学的 Andrew 文件系统(AFS)[Saty 85]。尽管它们致力于解决同一基本问题，这三种文件系统在它们的设计目标结构以及语法上都大相径庭。RFS 现在已经在大多数基于 System 5 的系统上实现了。NFS 已经在无数的 UNIX 系统和非 UNIX 系统中得到了更加广泛的接受。AFS 的开发工作后来由 Transarc 公司接手，现在已经发展成了 OSF 的分布式计算环境(DCE)的分布式文件系统(DFS)组件。

本章首先讲述分布式文件系统的特性。接着描述上面所说的每个分布式文件系统的设计和实现，同时分析它们各自的优缺点。

10.2　分布式文件系统的一般特征

传统的集中式文件系统可以让多个用户共享存储在本地机器上的所有文件。分布式文件系统对此做了扩展，允许通过网络互联的不同机器上的用户共享文件。分布式文件系统的实现是基于客户/服务器模型的。客户就是要访问文件的机器，而服务器便是存储文件并且允许用户访问这些文件的机器。有些系统要求客户机和服务器是不同的机器，而另外一些则没有这种要求，一台机器既可以是客户机又可以是服务器。

我们必须要明了分布式文件系统和分布式操作系统之间的区别[Tann85]。一个像 V [Cher 88]或 Amoeba[Tann 90]之类的分布式操作系统从其用户角度来看是一个集中式的操作系统，但实际上操作系统同时在不同机器上运行着。它可能提供一个让运行它的所有主机共享的文件系统。而一个分布式文件系统是一个软件层，它可以管理操作系统和文件系统之间的通信。分布式文件系统一般被集成到主机的操作系统中并且为有集中内核的系统提供文件访问服务。

下面是一些分布式文件系统的重要特性[Levy 90]。每个文件系统都可能有其中某些或全部特性。这些特性为我们评价和比较不同的结构提供了一个依据。

- **网络透明性**　客户访问远程文件主机上的文件的操作应该和访问本地文件的操作相同。
- **位置透明性**　通过文件的名字不能判断出该文件在网络中什么地方。

- **位置独立性** 如果一个文件的物理位置改变了，其名字不应该发生变化。
- **用户灵活性** 用户可以在网络的任一节点上访问共享文件。
- **容错能力** 如果一个组成部分(一个服务器或一个网络)发生错误，系统应该能够继续工作。不过这样有可能降低性能或使得文件系统的某一部分不能再用了。
- **可扩展性** 当负载重时，系统应该能够很好地扩展。同时，也可以通过增加组件的方式渐近式地进行扩展。
- **文件可动性** 可以从一个运行着的系统中将文件从某个物理位置移到另外的位置。

10.2.1 设计考虑

在设计分布式文件系统时有几个重要的问题要解决。这涉及到在功能、语法和性能之间进行平衡。我们可以根据各个文件系统怎样解决这些问题来比较它们。

- **名字空间** 一些分布式文件系统提供统一的名字空间，所有客户可以使用同样的路径名来访问同一给定文件。其他文件系统允许用户将共享的文件子树安装到该文件系统上，这样用户便可以自己定制名字空间。这两种方法现在都有一定的吸引力。
- **有状态或无状态操作** 一个有状态的服务器保存客户在执行两个请求之间的状态信息并且使用这些信息来保证以后的请求能够正确地工作。一些像 open 和 seek 的请求都是有状态的，因为服务器必须要记住客户打开了哪一个文件以及每个打开文件的指针偏移。在一个无状态系统中，每个请求都是自我包含的(self-contained)，服务器不保存任何客户操作的状态信息。例如，服务器可能要求客户指定每次文件读写的指针偏移，而不是维护一个指针偏移。有状态服务器较无状态服务器速度快，因为服务器可以利用客户状态信息减少大量的网络信量。不过，有状态服务器必须具备复杂的一致性维护和崩溃恢复机制。无状态服务器易于设计和实现，但是性能不太高。
- **共享的语义** 当多个用户并发地访问同一个文件时，分布式文件系统必须定义语义。如果一个客户对文件系统做出某些改变，UNIX 语义要求在其他客户发出下一个读或写系统调用时这个改变对这些用户必须是可见的。有些文件系统提供会话语义(session semantics)，该语义要求一个客户对文件系统的改变在其他客户读或写调用时必须传播到这些客户那里去。
- **远程文件访问方法** 纯粹的客户/服务器模型使用远程服务方法访问文件。在这种方法中，每一个动作都是由客户发出来的，而服务器只是简单地响应客户的请求。在很多分布式文件系统特别是有状态分布式文件系统中，服务器的作用比上述更大。它可能不仅响应客户请求，而且也对高速缓存一致性作出预测，一旦客户的数据变为无效时，它便通知客户。

现在我们看一下在 UNIX 系统中常见的分布式文件系统，看它们是如何解决这些问题的。

10.3 网络文件系统(NFS)

Sun 在 1985 年引入了 NFS，作为解决对远程文件系统进行透明性访问的一种方法。除

了发布协议外，Sun 同时也发表了一个参考实现，供应商使用该参考实现将 NFS 移植到另外几个操作系统中。NFS 也因此变为事实上的工业标准的一部分。几乎所有的 UNIX 变体和几种像 VMS 和 MS-DOS 的非 UNIX 系统都支持 NFS。

NFS 体系结构是基于客户-服务器模型的。文件服务器是输出一组文件的机器，而客户是访问这些文件的机器。一台机器既可以是不同文件系统的客户机，也可以是服务器。不过 NFS 代码分为客户和服务器两个部分，用以支持只有客户或只有服务器的系统。

客户和服务通过远程过程调用通信，远程过程调用是以同步请求的方式执行的。当客户机上的一个应用程序试图访问一个远程文件时，内核向服务器发送一个请求，客户进程被阻塞，直到服务器返回一个应答。服务器一直处于等待状态，如果有客户请求到来，便处理它们，并将应答返回给客户。

10.3.1　用户透视

NFS 服务器输出一个或多个文件系统。每个输出的文件系统既可以是整个磁盘分区，也可以是一个文件子树①。服务器可以通过在/etc/exports 中的项来指定哪个客户可以访问各个输出的文件系统以及其权限是否是只读的或者是可读写的。

就像安装一个本地文件系统一样，客户机可以将一服务器输出的文件系统或子树安装到已存的目录树中。尽管服务器可能将其输出为可读写的，客户机可以按只读方式安装目录。NFS 支持两种类型的安装 —— 硬安装和软安装。如果服务器不对客户请求做出反应，安装类型便可以决定客户机采取什么样的行动。如果文件系统是硬安装的，客户会一直重试，直到返回一个应答，如果文件系统是软安装的，客户收不到应答，过一会儿就会放弃请求，返回一个错误。一旦 mount 命令成功，客户便可以使用访问本地文件系统的命令来访问远程文件系统。有些文件系统也支持 spongy 安装，这种类型在安装时同硬安装一样，但在随后的 I/O 请求中则跟软安装一样。

NFS 安装比本地文件系统安装更加自由。尽管很多客户要求调用安装命令的用户必须是特权用户②，协议本身并没有这么规定。客户可以在目录树的多个位置安装同一个文件系统，甚至可以安装在已安装的文件系统的子目录上。服务器可以仅仅输出它的本地文件系统，并且在遍历路径名时不通过它自己的安装点。因此，客户要想看到服务器上的所有文件，必须要将服务器上的所有文件系统都安装上。

这在图 10-1 中作了说明。服务器系统 nfssrv 有两个磁盘。它将 dsk1 安装在 dsk0 的/usr/local 目录上并且将目录/usr 和/msr/local 输出。假设一个客户执行下面四个 mount 操作：

```
mount -t nfs    nfssrv: /usr          /usr
mount -t nfs    nfssrv:/usr/ul        /ul
mount -t nfs    nfssrv: /usr          /users
mount -t nfs    nfssrv: /usr/local    /usr/local
```

① 不同的 UNIX 变体有自己的管理输出粒度的规则。某些只允许输出整个文件系统，而某些则允许输出文件系统的一个子树。

② 例如，在 Digital 的 ULTRIX 中只要用户对安装点目录有写权限，用户就可以安装 NFS 文件系统。

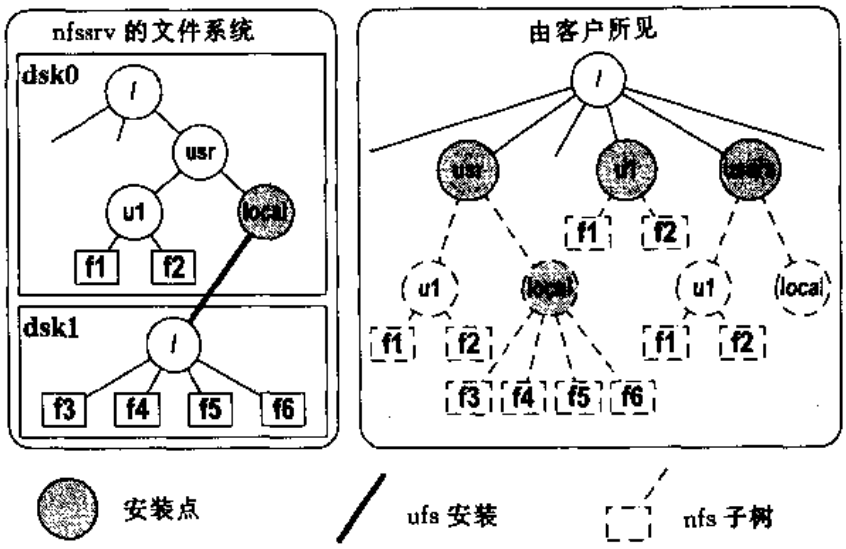

图 10-1 安装 NFS 文件系统

假设四个 mount 命令全都执行成功。在客户端，子树/usr 反映了 nfssrv 的整个/usr 子树，因为客户也安装了/usr/local。客户上的子树/ul 映射了 nfssrv 上的/usr/u1 子树。这说明可以安装输出文件系统的子目录③。最后，在客户机上/users 子树仅仅映射了 nfssrv 上的/usr 子树在 dsk0 上的部分；在 dsk1 上的文件系统在客户机上的/users/local 上是不可见的。

10.3.2 设计目标

NFS 最初的设计目标如下：

- NFS 应该不仅仅局限于 UNIX 上。任何操作系统都应该能够实现 NFS 服务器和客户。
- 协议不能依赖任意特定的硬件。
- 如果服务器或者客户崩溃，应该有很简单的机制来恢复。
- 应用程序应该能够透明地访问远程文件，而不用特定的路径名和库，也不用重编译。
- 必须为 UNIX 客户维护 UNIX 文件系统语义。
- NFS 的性能必须与本地磁盘差不多。
- NFS 的实现必须是可移植的。

10.3.3 NFS 组成

NFS 的实现有几个组成部分。其中有些是仅仅局限于客户机或服务器，而另外一些则是由两者共享的。有一些组件不属于内核功能，而是属于 NFS 扩展接口的一部分：

- NFS 协议，它定义了一组客户可能向服务器发送的请求以及请求中的参数和可能返回的应答。协议版本 1.0 仅仅在 Sun 中存在，从来没有发行过。NFS 2.0[NFSv 2]④

③ 并不是所有的实现都允许这样。

④ 这包括那些支持 NFSv3 的实现。

最先于 1985 年在 SunOS 中推出[Sand 85b]，现在所有的 NFS 实现都支持它，因此本章主要讲述 NFS 2.0。10.10 讲述协议的 3.0 版，3.0 版于 1993 年推出并且已被很多供应商所支持。表 10-1 列举了所有的 NFSv2 请求。

- 远程过程调用(RPC)协议定义了客户和服务器之间所有的交互的格式。每个 NFS 请求是以一个 RPC 包的形式被发送的。
- 扩展数据表达(XDR)提供了一种与机器无关的通过网络传输数据的方法。所有的 RPC 请求都使用 XDR 来传送数据。注意，XDR 和 RPC 也被除 NFS 以外的其他服务所使用。
- NFS 服务器代码，负责处理所有的客户请求，提供对输出文件系统的访问。
- NFS 客户代码通过向服务器发送一个或多个 RPC 请求来实现所有的客户系统对远程文件的调用。
- 安装协议定义了安装和卸载 NFS 文件系统的语义。表 10-2 描述了协议的主要内容。
- 用于 NFS 的几个守护进程。在服务器端，一组 nfsd 守护进程监听以及响应客户机的 NFS 请求，mountd 守护进程处理安装请求。在客户端，一组 biod 守护进程处理 NFS 文件块的异步输入/输出。
- 网络锁定管理器(NLM)和网络状态监视器(NSM)：它们一起实现通过网络锁定文件的功能。这些应用程序尽管并不是 NFS 的正式组成部分，还是有很多 NFS 的实现包含了它们，并提供基本协议不可能提供的服务。NLM 和 NSM 分别通过守护进程 lockd 和 statd 来实现服务器功能。

表 10-1　NFSv2 操作

Proc	输入 args	结果
NULL	void	void
GETATTR	fhandle	status,fattr
SETATTR	fhandle,sattr	status,fattr
LOOKUP	dirfh,name	status,fhandle,fattr
READLINK	fhandle	status,link_value
READ	fhandle,offset,count,totcount	status,fattr,data
WRITE	fhandle,offset,count,totcount,data	status,fattr
CREATE	dirfh,name,sattr	status,fhandle,fattr
REMOVE	dirfh,name	status
RENAME	dirfh1,name1,dirfh2,name2	status
LINK	fhandle,dirfh,name	status
SYMLINK	dirfh,name,linkname,sattr	status
MKDIR	dirfh,name,sattr	status,fhandle,fattr
REDIR	dirfh,name	status
READDIR	fhandle,cookie,count	status,dir_entries
STATFS	fhandle	status,file_stats

表 10-2 安装协议(1 版)

Proc	输入 args	结果
NULL	void	void
MNT	pathname	status,fhandle
DUMP	voidp	mount list
UMNT	pathname	void
UMNTALL	void	void
EXPORT	void	export list

10.3.4 无状态

NFS 协议的最主要的特点是服务器是无状态的,因此不需要维护关于客户操作正确与否的任何信息。每个请求都是独立的,其中包含所有处理该请求所需的信息。除了可选择的为了缓冲和统计收集目的需要保存某些执行信息外,服务不需要维护任何客户请求的执行信息。

例如,NFS 协议不提供打开和关闭一个文件的请求,因为这样会产生一些服务器必须要记住的状态信息。基于同样的原因,READ 和 WRITE 请求将起始偏移作为参数传到服务器,而不像对本地文件的 read 和 write 操作,这些操作可以从打开文件对象中得到该次读写操作的起始偏移(见 8.2.3 小节)⑤。

无状态的协议使得崩溃恢复变得很简单。当客户崩溃时,没有必要进行任何恢复,因为服务器没有保存客户的任何一致性信息。当客户机重启时,客户端可能重新安装文件系统,运行访问远程文件的应用程序。服务器不需要知道也不需关心客户的崩溃。

当服务器崩溃时,客户发现其请求得不到任何响应。于是它便持续地发送请求,直到服务器重启⑥。因为服务器处理任何请求都不依赖于请求以外的信息,所以此时服务器便可以接收请求并且处理它们。当服务器将回答信息反送到客户时,客户端停止重复发送请求。客户机不知道服务器是否已经崩溃,是否已经重启还是简单地因为反应慢。

与上述相反,有状态协议需要复杂的崩溃恢复机制。服务器必须检测到客户机崩溃,并且将为该客户保存的所有状态信息丢弃掉。当服务器崩溃重启时,它必须要通知客户机,以便将各个客户在服务器上的状态重新建立起来。

无状态的主要缺陷是服务器在应答一个请求之前必须将所有的修改都提交到稳定的存储设备中。这就意味着服务器在返回一应答之前,不仅要把文件数据,而且也要把 i 节点和间接块之类的元数据信息写回磁盘。否则,服务器崩溃时可能将客户确信已经写入磁盘的信息丢失掉(系统崩溃也有可能使本地文件系统中的数据丢失掉,但是这种情况用户知道崩溃也知道数据可能丢失)。无状态也有其他一些缺点。它需要一个单独的协议(NLM)对文件进行锁定。同样,为了解决同步写时的性能问题,大多数客户将数据和元数据存放在本地高速

⑤ 某些系统提供了 pread 和 pwrite 调用,偏移是其中一个参数。这个调用在多线程系统中非常有用(见 3.3.2 小节)。

⑥ 这个只对硬安装(通常缺省就是硬安装)有效。对于软安装,客户在一段时间后将放弃操作并返回给应用程序一个错误信息。

缓存中。这就很难保证协议的一致性(10.7.2 小节将更加详细地讨论这个问题)。

10.4 协　议　族

NFS 协议族的基本协议包括 RPC,NFS 和 Mount,它们都使用 XDR 进行数据编码。其他相关的协议有 NLM,NSM 和 portmapper。本节主要描述 XPR 和 RPC。

10.4.1 扩展数据表示(XDR)

处理基于网络的不同计算机之间通信的程序必须要考虑在网络上传输的信息的解译问题。因为在不同端的计算机的硬件和操作系统可能迥然不同,它们对于数据元素的内部解释也相差很大。这种差别主要有字节顺序,数据类型的大小以及字符串和数组的格式。如果在相同机器上或者在两台类似的机器上通信则不会遇到这样的问题,但是在异质环境中这个问题必须加以妥善的解决。

在计算机间传输的数据分为两种类型— 不透明的和有类型的。不透明的数据也就是字节流类型,如文件传输和调制解调器通信传输的都是不透明数据。接收者简单地将接受到的数据看作是字节流,而不试图去解释它。但是对于有类型数据来说,接受者必须要解释,所以发送者和接受者都必须知道数据格式。例如,一个小印第安(little—endian)的机器可能会发送一个值为 0x0103(十进制数为 259)的两字节整数。如果一个大印第安(big—endian)机器接收到了该数字,它会将其解释为 0x0301(十进制数为 769)。显然,这两台机器不能相互理解。

XDR 标准定义了在网络上传输的数据的机器无关的表达方式。它定义了一些基本数据类型,同时又定义了构造复杂数据以及类型的规则。因为它是由 Sun 引入的,所以它在字节顺序之类的问题上不免受到 Motorala680x0 体系结构的影响(Sun-2 和 Sun-3 工作站都是基于 680x0 的)。XDR 的一些基本定义如下:

- **整数**　是一个 32 位的实体。第 0 个字节(从左到右)是最高位字节,有符号整数用 2 的补码记数法表示。
- **可变长度的不透明数据**　由 length 域(4 字节长整数)来描述,随后是数据本身。如果不够四个字节用 NULL 填充。对于固定长度不透明数据,field 域被省略掉了。
- **字符串**　有一个 length 描述其长度,随后跟着一串由 ASCII 组成的字符串。每四个字节组成一组,如果不满一组,可用 NULL 填充。如果字符串长度正好是 4 的倍数,则不用 NULL 填充。
- **数组**　一般是同构元素,由一个 size 域描述数组大小,随后跟着一组按其本来顺序排列的元素。size 域是一个四字节整数,对于固定长度的数组,该域被省略掉。每个元素的大小必须是 4 字节的倍数。元素必须是同种类型,它们可以有不同的尺寸,比如字符串的数组便是这种情况。
- **结构**　将其中的元素按它们本来的顺序排列,组成一个结构。每个组成部分都是以 4 字节为界的。

图 10-2 所示是一些 XDR 编码的例子。除了上述定义外,XDR 也提供一种描述数据的

形式化语言规范。在下面小节将要描述的 RPC 规范语言就是简单地对 XDR 语言进行了扩充。同样地，rpcgen 编译器会理解 XDR 规范，然后产生将数据以 XDR 形式编码和解码的例程。

值	XDR 表示			
0x203040	00	20	30	40
Array of 3 integers {0x30,0x40,0x50}	00	00	00	30
	00	00	00	40
	00	00	00	50

Variable length array of strings
{
"Monday",
"Tuesday"
}
	00	00	00	02

	00	00	00	06
	'M'	'o'	'n'	'd'
	'a'	'y'	00	00
	'T'	'u'	'e'	's'
	'd'	'a	'y'	00

图 10-2 XDR 编码的例子

XDR 是用于任意计算机之间通信的通用语言。其主要缺点是如果某台计算机的数据表达语义同 XDR 规范不匹配，那么就需要繁琐的转换过程，使性能降低。最不合适的是如果两台机器同种类型，本来不需要对数据编码便可以通信的，可是由于它们的格式不符合 XDR 规范，两台计算机还要进行繁琐的数据转换工作。

例如，考虑两台使用基于 XDR 编码的协议的 VAX-11 计算机。因为 VAX 是小印第安(字节 0 是最低位)，发送者必须将每个整数转换成大印第安形式(XDR 强制如此)，同时接收者也必须将大印第安形式数据转换成小印第安形式。这显然是多此一举。如果规范能够提供某种手段使收发双方知道对方的特征，如果类型相同就不用 XDR 编码，这样会避免浪费时间进行不必要的转换。DCE RPC[OSF 92]使用这种编码方案替代了 XDR。

10.4.2 远程过程调用(RPC)

RPC 协议指定了不同计算机之间通信的格式。客户发送一个 RPC 请求到服务器，服务器处理该请求，向客户返回一个 RPC 应答。该协议解决了消息格式，传输和身份验证之类的不依赖于特定应用程序和服务的问题。有些服务是建立在 RPC 之上的，比如 NFS，Mount，NLM，NSM，portmapper 和网络信息服务(NIS)。

注意 PRC 有几种不同的实现。NFS 使用 Sun 公司的 RPC 协议[Sun 88](Sun RPC 或 ONC-RPC，ONC 代表 Open Network Computing)，在本书中，除特别指定，RPC 特指 Sun RPC。在本书中唯一提到的其他 RPC 实现是 OSF 的分布计算环境，也就是 DCE RPC。

不像 DCE RPC 那样提供同步和异步操作，Sun RPC 仅使用同步请求。当一个客户发送一个 RPC 请求时，调用进程会阻塞在那里，直到收到一个应答。这样，RPC 的行为跟本地过

程调用类似。

RPC 协议提供请求的可靠性传输，这意味着它必须保证请求已经到达目的地，也必须保证能够收到一个应答。尽管 RPC 本质上是传输独立的，它却是建立在不可靠的 UDP/IP 协议之上的。如果 RPC 发现有的请求没有收到应答，便重发该请求，直到收到一个应答。通过这种方法 RPC 可以实现可靠的数据报服务。

图 10-3 描述一个典型的 RPC 请求和应答。xid 是标记请求的传输 ID。客户端为每个请求产生一个独特的 xid，服务器在应答中返回同样的 xid。这样客户便可以判断是哪一个请求的应答信息到达了，服务器也可以据此鉴别重复请求(由客户端重发造成)。direction 域标明信息是请求还是应答。rpc_vers 域表明 RPC 协议的版本号。prog 和 vers 分别是特定 RPC 服务的程序和版本号。一个 RPC 服务可能注册多个协议版本。例如 NFS 协议程序号是 100003，它支持两种版本。proc 标明在服务程序中要调用的特定过程。在应答中，reply_stat 和 accept_stat 中包含了状态信息。

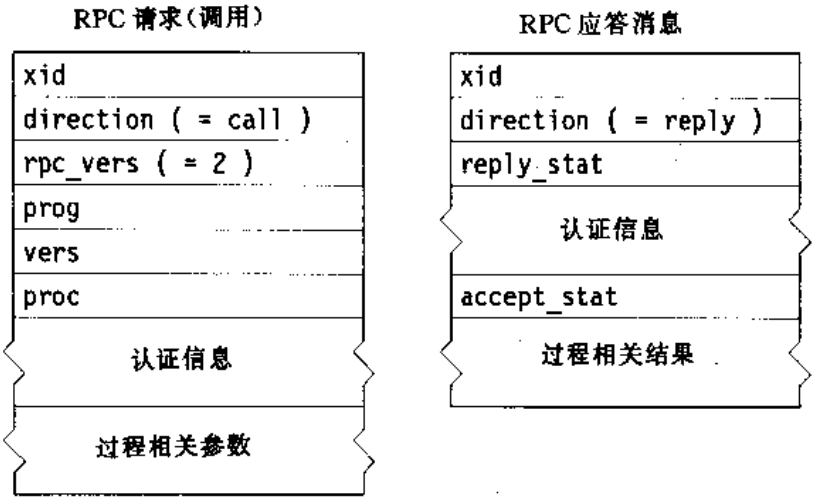

图 10-3　RPL 消息格式

RPC 使用五种确认机制来标识远程过程调用的调用者——AUTH_NULL，AUTH_UNIX，AUTH_SHORT，AUTH_DES 和 AUTH_KERB。AUTH_NULL 表示没有认证。AUTH_UNIX 由 UNIX 方式的凭证组成，包括客户机名称，UID 和一个或多个 GIDS。服务器在收到一个 AUTH_UNIX 的凭证之后产生一个 AUTH_SHORT，然后将其返回给调用者，并在以后的请求中使用。其思想是服务器可以很快地解密 AUTH_SHORT 凭证来识别已知客户，因此可以提供认证。这是一种可选的特性，并没有多少服务支持它。AUTH_DES 是一种安全的认证程序，它使用了一种称为私有锁的机制[Sun 89]。AUTH_KERB 是基于 kerberos[Stei 88]凭证机制的另外一种安全认证程序。每一项服务都要决定要接受哪一种认证机制。NFS 支持以上五种认证方式，不过要求仅对空过程才能使用 NUTH_NULL 认证方式。但是大多数 NFS 实现只使用 AUTH_UNIX 这一种认证方式。

Sun 也提供了一个 RPC 编程语言和一个叫做 rpcgen 的 RPC 编译器。基于 RPC 的服务可以利用这种语言进行完整的定义，并产生一个形式化的接口语言定义。当 rpcgen 处理这个规范时，它会产生一组 C 源文件，其中包含 XDR 转换程序以及客户和服务器程序的存根

(stub)版本,同时还有包含客户和服务器共用信息定义的头文件。

10.5 NFS 实现

我们现在来看一下典型的 UNIX 是怎样实现 NFS 协议的。NFS 已经被移植到一些像 MS-DOS 和 VMS 等非 UNIX 操作系统中去了。其中有些实现仅提供只有客户或只有服务器的实现,而另外一些则既提供了客户的实现,又提供了客户机的实现。而且有些提供商提供的 NFS 是专用的,不能运行在通用操作系统之上,这些供提供商有 Auspex,Network Appliance Corporation 和 Novell 等。最后,对于不同的操作系统还有很多 NFS 的用户层实现,通常都是做为免费软件或共享软件的形式发行。当然这样一些实现是完全不同的,10.8 小节描述了其中一些有趣的变体。在本节,我们主要讨论传统的支持 vnode/vfs 接口的 UNIX 系统是怎样实现 NFS 的。

10.5.1 控制流

在图 10-4 中,服务器输出一个 ufs 文件系统,该文件系统被客户所安装。当客户机上的进程调用系统调用来对 NFS 文件系统进行操作,文件系统无关代码识别出文件的 v 节点并且调用相应的 v 节点操作。对 NFS 文件来说,与 v 节点相联系的文件系统相关数据结构是 rnode(远程节点)。v 节点的 v_op 域指向实现不同的 v 节点操作的 NFS 客户例程向量(struct vnodeops)。这些程序首先构造 RPC 请求,然后将这些请求发送到服务器端。客户机上的调用进程阻塞在调用点,直到服务器返回一个应答[见脚注 6]。服务器首先识别对应的本地文件的 v 节点,然后调用相应的由本地文件系统实现的 vnodeops 操作。

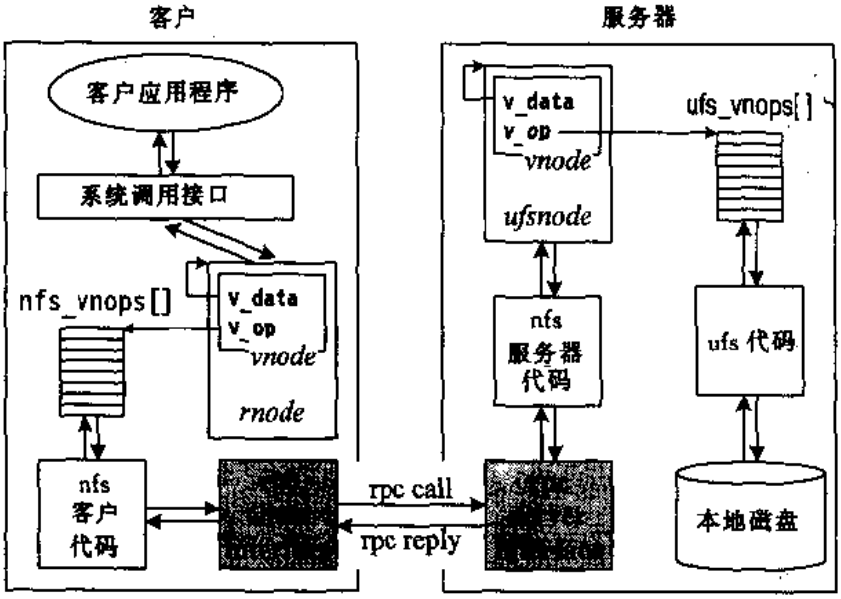

图 10-4 NFS 中的控制流

最后,服务器处理完毕请求,将结果绑定到一个 RPC 应答消息中,然后返回到客户端。客户端的 RPC 层接受到该消息后唤醒睡眠进程。这个进程完成客户的 v 节点操作以及其他

的系统调用然后返回用户。

上面的例子说明，客户必须为每一个活动的 NFS 文件维护一个本地 v 节点，这样就可以很方便地将 v 节点上的操作转换为 NFS 的客户函数。同样，客户也可以缓存远程文件的属性，这样客户不用访问服务器就可以进行某些操作了。与客户缓存有关的内容将在10.7.2中讨论。

10.5.2　文件句柄

当客户向服务器发送一个 NFS 请求时，它必须标识它要访问的文件。如果在每一次访问时都要将路径名传过去，速度会很慢。NFS 协议将每一个文件或目录同一个叫做文件句柄的对象联系起来。当客户第一次访问文件或通过 Lookup，CREATE 以及 MKDIR 请求创建文件时，服务器便为文件分配一个句柄。服务器将该句柄返回给客户，以后客户便可以使用该句柄对文件进行操作。

客户将文件句柄看作是一个不透明的 32 字节的对象，并不去解译其内容。服务器可以随意实现文件句柄，只要它能保证句柄和文件是一一对应关系即可。一般情况下，文件句柄中含有一个文件系 ID，用来唯一地标识一个本地文件系统；还有文件的 i 节点号以及该 i 节点的产生数(generation number)。文件句柄中可能还包含该文件所在的输出目录的产生数。

在 i 节点中加入一个产生数，目的是解决 NFS 所独有的问题。这是因为在客户一开始访问文件(一般通过 lookup，该函数返回一个文件句柄)和客户发送 I/O 请求之间服务器有可能删掉文件并且将该文件的 i 节点用到了其他文件上。这样就有可能发生错误。因此服务器需要有一种方法来判断由客户发送的文件句柄是否已经过时。在 i 节点每次被释放时(与之相关联的文件也被删掉了)服务器将其产生数增加 1。这样服务器就可以分辨出请求是否要求访问过时的文件；如果是则返回一个 ESTALE(过时文件句柄)的错误状态。

10.5.3　Mount 操作

当一个客户安装一个 NFS 文件系统时，内核分配一个新的 vfs 结构，调用 nfs_mount( ) 函数。nfs_mount( )使用 Mount 协议向服务器发送一个 RPC 请求。该请求的参数是被安装目录的路径名。服务器上的守护进程 mountd 接收这个请求，并且对路径名进行转换。它检测路径名是不是一个目录以及该目录是否已经输出。如果是，mountd 返回一个成功完成的应答，在应答中包含有那个目录的文件句柄。

客户接收到成功的应答，完成 vfs 结构的初始化工作。它将服务器的名字和网络地址记录在 vfs 的私有数据对象中。接着为根目录分配 rnode 和 v 节点。如果服务器崩溃重启，客户端仍然包含着被安装文件系统的信息，但是服务器却丢失掉了这些信息。因为客户在 NFS 请求中发送合法文件句柄，所以服务器假设以前已经有过一次成功的安装，于是服务器便重建其内部记录。

服务器需要检查每一个 NFS 请求，查看发送请求的用户是否有访问文件系统的权限。它必须保证正在被操作的文件已经输出到那个客户了(如果请求要修改文件，输出的文件系统必须是可读写的)。为了更加有效地检测，文件句柄中包括输出目录的<inode，generation number>偶对。服务器上维护了一个所有输出目录的内存列表，这样就可以很快地完成检测。

10.5.4 路径名查找

客户通过 mount 操作得到顶层目录的文件句柄。在路径查找过程中或者从 CREATE 或 MKDIR 的操作结果中客户可以得到其他的句柄。在像 open，create 和 stat 之类的系统调用中客户会发出 lookup 操作。

在客户端，lookup 从当前目录或者根目录(要视路径是相对还是绝对来定)开始，每次处理一个分量。如果当前目录是一个 NFS 目录或者通过一个安装点而进入 NFS 目录中，查找操作调用 NFS 的 VOP_LOOKUP 函数。这个函数给服务器发送一个 LOOKUP 请求，将父目录的文件句柄和要搜索的分量的名字作为参数传递。

服务器从文件句柄中得到文件系统 ID，用该 ID 找到文件系统的 vfs 结构。接着在该文件系统上调用 VFS_VGET 操作，该操作会将文件句柄加以转换，将一个指向父目录的 v 节点的指针返回(如果 v 节点不在内存中，首先要分配一个)。接着服务器便可以在那个 v 节点上调用 VOP_LOOKUP 操作，这样会调用本地文件系统的相应函数。该函数搜索目录以查找该文件。如果找到了，将其 v 节点调入到内存中，返回指向它的指针。

服务器接着在目标文件的 v 节点上调用 VOP_GETATTR 操作，然后是 VOP_FID 操作，该操作为文件产生一个文件句柄。最后它构造一个应答信息，其中包含状态、分量的文件句柄和文件属性信息。

当客户收到应答后，它为新的文件分配 v 节点和 rnode 结构(如果该文件已经被查找过了，客户或许已经有它的 v 节点了)，它将文件句柄和文件属性复制到 rnode 中，并且搜索下一个分量。

如果每次只搜索一个分量速度会很慢，因为解析一个路径名需要几次 RPC 请求。客户可以将目录信息缓冲在本地，这样会避免一些 RPC(10.7.2 小节)请求. 尽管在一个 LOOKUP 调用中将路径名全部输入效率似乎更高，但这种方法有严重缺陷。首先，因为客户可能将一个文件系统安装到路径的中间目录上，为了能够正确地解析路径名，服务器需要知道客户的所有的安装点。其次，一次解析整个路径名需要服务器理解 UNIX 路径名语义。这与 NFS 的无状态和操作系统的无关性的设计目标是矛盾的。NFS 服务器已经被移植到各种不同的系统中，像 VMS 和 Novell Netware 等，而这些系统有着不同的路径名约定，因此也不可能全部遵守 UNIX 路径名定义。只有在 mount 操作中才需要使用全路径名，而它却使用了完全不同的协议。

10.6 UNIX 语义

因为 NFS 的基本目的是用于 UNIX 客户的，所以为远程文件访问保持 UNIX 语义很重要。然而，因为 NFS 协议是无状态的，所以客户不能在服务器上维护打开文件信息，这会导致一些与 UNIX 不兼容的地方，下面我们将分别讨论。

10.6.1 打开文件权限

UNIX 系统是在进程第一次打开文件的时候检查访问权限的，而不是在每一次读或写的时候。假设在一个用户打开文件准备往里写东西时，文件的所有者将其权限变为只读，在

UNIX 系统中，用户可以继续写文件，直到他关闭了该文件为止。而在 NFS 中，由于缺乏打开文件的概念，所以在每次读写的时候都要检查权限。因此在上面的例子中，用户如果继续访问就会产生一个错误，这是客户所不愿意发生的。

尽管没有办法充分地解决这个问题，NFS 提供了一种调和的办法。不管权限如何，服务器总是允许文件所有者对文件进行读写。从表面上看，这似乎更进一步违反了 UNIX 语义，因为这个方法看起来允许所有者修改本来是写保护的文件。不过 NFS 客户代码可防止这种情况的发生。当客户打开文件时，LOOKUP 操作返回文件属性和文件句柄。属性中包含着文件被打开时许可。如果文件是写保护的，客户代码会从 open 调用中会返回一个 EACCESS（拒绝访问）错误。必须指出的是，服务器的安全机制依赖于客户适当的行为。在这个例子中，问题并不是很严重，因为它仅仅导致属主没有写权限。10.9 节中我们将讨论 NFS 安全性的主要问题。

10.6.2　删除打开文件

如果一个 UNIX 进程试图删除其他进程已打开的文件，内核便将该文件标识为删除，将其登记项从父目录中删除。尽管现在没有任何新的进程可以打开这个文件，那些以前打开该文件的进程依然可以访问它。当打开该文件的最后一个进程将文件关闭后，内核便物理地删除该文件。这个特性被很多应用程序用来实现临时文件。

这又将导致 NFS 的一个问题，因为服务器并不知道打开文件的信息。此时的折衷方法是修改 NFS 客户代码，这是因为 NFS 客户的确知道打开文件。如果一个客户检测到有的进程试图删除一个打开文件，它将删除操作改为 RENAME 操作，给文件一个新的名字和位置。客户一般选择不常见的和长的文件名，以防止与已存文件的名字重复。当文件最后被关闭的时候，客户发送一个 REMOVE 请求来删除文件。

当打开文件的进程和删除文件的进程在同一台机器上时上述方法十分奏效。但是当两个进程不在同一个机器上时，就不能确保文件不被删除。如果是这样，用户再次试图读写文件时，就会产生一个不可预料的错误（stale file handle）。这种折衷方法产生的另外一个问题是当客户机在 RENAME 和 REMOVE 操作之间崩溃时，服务器上就会留下一个垃圾文件。

10.6.3　读和写

在 UNIX 中，read 和 write 系统调用在 I/O 一开始时便锁定文件的 v 节点。这样在系统调用粒度级上文件 I/O 操作是原子性的。如果针对同一个文件的两个 write 同时发生，内核对它们进行排序，先完成一个，然后是另一个。同样的，内核要保证 read 系统调用正在进行的时候不能有其他进程改变文件。本地文件系统相关代码在一个单独的 vop_rdwr 操作上下文中处理文件锁定问题。

对于一个 NFS 文件来说，对于同一客户机上的并发访问同一个文件的两个进程，客户将它们串行化。但是在不同机器上的两个进程访问服务器上同一文件时，它们是独立地访问的。一个 read 或者 write 操作可能需要几个 RPC 请求（最大的 RPC 请求是 8192 字节），并且由于服务器是无状态的，所以它不能在两个请求之间维护锁。NFS 对这一类的互相重叠的 I/O 请求没有提供任何保护措施。

合作进程可以使用 NLM 协议锁定整个文件或者文件的一部分。但是这个协议只提供

建议性加锁功能，这样如果别的进程试图强行访问文件时该锁就被跳过去了。

10.7 NFS 性能

NFS 的一个设计目标就是要使其性能能够与本地磁盘性能差不多。现在的问题不是吞吐量问题，而是完成正常工作需要的时间。现在有好多基准程序试图在 NFS 文件系统上模拟一个正常的工作负荷，其中最著名的有 LADDIS[Witt 93]和 nhfsstone[Mora 90]。本节主要讨论 NFS 的限制以及解决这些问题的方法。

10.7.1 性能瓶颈

NFS 服务器都是功能强大的机器，一般都有大的高速缓存和快速磁盘，用来补偿往返的 RPC 请求造成的时间延迟。然而 NFS 的设计中有几个方面却直接导致了性能的低下。

因为 NFS 协议是一个无状态协议，所以它要求所有的写操作在返回前都要将结果写回磁盘。这些结果不仅有对文件元数据的修改 i 节点和间接块，也有对文件数据本身的修改。结果任何一个对文件系统做出某种修改的 NFS 请求(如 WRITE，SETATTR 或 CREATE)执行速度都很慢。

读取文件属性要求针对每个文件都要调用一次 RPC 请求。结果像 ls-l 之类对目录进行操作的命令导致大量的 RPC 请求。在本地情况下，这种操作是很快的，因为 i 节点都在高速缓冲区中，因此 stat 调用仅需引用一下内存即可。

如果服务器不能迅速地响应请求，客户会假设服务器崩溃了或者请求在网络传输过程中丢失了，因而会重传。处理重传请求可能会增加服务器的负担，这使问题变得更加严重。最严重的后果就是服务器由于到来的通信量太大而崩溃。

下面我们看一下解决这些 NFS 性能问题的方法以及对这些解决方法的反响。

10.7.2 客户端高速缓存

如果客户对远程文件的每一次访问都要通过网络进行，NFS 的性能会变得让人难以忍受。因此大多数 NFS客户分类缓存文件数据块和文件属性。它们将文件数据块放在磁盘缓存区中，将属性放在 rnode 中。这种缓存策略是有危险的，因为客户没有办法知道缓存的内容是否有效，每次必要时缺少对服务器的查询。

客户端采取了一些预防措施来防止使用陈旧的数据。内核在 rnode 中维护了一个过期时间，这个时间规定属性缓存的期限。一般情况下，客户每 60 秒或更少的时间要重新从服务器读取。如果过了过期时间访问这些文件属性，客户要重新从服务器加载这些属性。同样地，对于文件数据块，客户检查文件修改时间是否从服务器读取后就没有变化，以此来检查缓存的一致性。客户可能会使用该时间戳的缓存值，当时间过期时会发出一个 GETATTR 命令。

客户端缓存是获得较好性能的必要条件。在这里我们介绍的预防措施只能减少一致性问题，但是不能完全消除此问题。事实上，它们又引入了新的问题[Mack 91]和 [Jusz 89]。

10.7.3 延迟写

NFS 同步写的要求只适用于服务器。客户端可以随便延迟写，这是因为如果客户机崩

溃而导致数据丢失，而这用户是知道的。所以，客户机对整个数据块采用异步写（发出 WRITE 请求但不用等待应答），而对部分数据块则延迟一段时间再写（在其后的某一时间写）。大多数的 UNIX 实现每 30 秒或在文件关闭后将延迟写的内容写回服务器。客户机上的守护进程 biod 专门负责这一类写操作。

尽管服务器在返回应答前必须将写的内容刷新到稳定存储，服务器并不一定非要将数据写回磁盘。服务器可以使用专用硬件保存数据，确保在系统崩溃时数据不会丢失掉。例如，有些服务器使用特殊的电池驱动的非易失性存储器（NVRAM）来存储数据。WRITE 操作将数据写到 NVRAM 缓冲区中。服务器在稍后的时间把 NVRAM 内容写回磁盘。这样服务器就可以很快地响应客户的写请求，因为将数据写入 NVRAM 要比将数据写入磁盘快。磁盘驱动程序可以优化从 NVRAM 到磁盘的写操作顺序，从而可以使磁头移动距离最小。另外，同一个缓冲区多次更新结果可以使用一个操作写回磁盘。[Mora 90]和[Hiz 94]描述了使用 NVRAM 的 NFS 服务器和实现。

[Jusz 94]讲述了一种称为收集写（write-gathering）的技术，它可以在不使用特殊硬件的情况下，减少同步写操作的次数。它是基于 NFS 客户要使用很多 biod 守护进程来处理写操作这一事实的。当一个客户进程打开一个文件并且向里面写数据时，内核将写的数据存放在高速缓存中并且将其标识为延迟写。当客户关闭文件时，内核便将它缓冲的数据块写回磁盘中去。如果在客户机上有足够的 biod 守护进程，这些守护进程可以并行发送写命令。结果服务器经常收到针对同一个文件的大量的写请求。

使用收集写方法，服务器不用立即处理 WRITE 请求。相反地，它将写操作延迟，期望同时能收到针对同一文件的其他写请求。于是服务器便可以将对同一文件的写请求收集起来，将它们一起处理。完成所有这些写请求后，服务器每个客户都发送应答信息。当客户使用 biods 时这种技术很有效，当客户都使用大量的 biods 时性能则最高。尽管这种方法看起来使得单个的写操作延时增大，实际上极大地提高了性能，因为它减少了服务器上写操作的次数。例如，如果服务器收集了对同一个文件的几个写请求，它可以仅仅使用一次数据写和一次 i 节点写操作便将写的内容写入到磁盘中去（而不是需要 n 次写操作）。收集写方法提高了服务器的吞吐量，减少了其磁盘负担。这种方法使得单个写操作的延迟增加，其总体效果却是使得平均写的时间减少了。

有些服务器使用 UPS 负责在系统崩溃时将数据写回磁盘。有些服务器预料到很少发生系统崩溃，因此忽略了 NFS 同步写的要求。对这个问题有很多的解决方案，但作用都不大，反而加重了其严重性，在 10.10 节中描述的 NFSv3 对协议的内容作了改变，容许客户机和服务器可以安全地使用异步写。

10.7.4　重传高速缓存

为了提供可靠性传输，RPC 客户在收到应答之前重新发送请求。一般地，第一次重传的等待时间很短（可配置，大约 1～3 秒），以后重传一次其等待时间呈指数增加。如果经过一定次数的重传之后仍然没有收到应答，客户会发送一个新的请求，该请求除传输 ID 不同外其余都和老请求相同。

重传的原因一般是由于网络上数据包丢失或者服务器没有及时地应答请求。很多情况下对原来请求的应答还在向客户传输的过程，客户便将请求又发送了一次。当服务器崩溃或

者网络极度拥挤时重传会变得很频繁。

服务器必须正确有效地解决请求重传问题。NFS 请求分两种类型——幂等(idempotent)和非幂等(nonidempotent)的[Jusz 89]。像 READ 和 GETATTR 之类的幂等请求可以被重复执行两次而没有任何副作用。而对于非幂等请求来说,重复执行会引发问题。以任何方法修改文件系统的请求都是非幂等请求。

举一个例子。下面是重发 REMOVE 请求所产生的各种事件。

1. 客户发送一个 REMOVE 请求,请求删除某一个文件。
2. 服务器成功地将文件删除掉。
3. 服务器向客户送回一个应答。该应答在网络上丢失了。
4. 客户发送重复的 REMOVE 请求。
5. 服务器第二次处理 REMOVE 请求。因为文件已被删除,所以产生一个错误。
6. 服务器将错误消息发送给客户。该错误信息成功到达客户。

结果,客户得到了一个错误信息,以为文件没有被删除。但实际上文件已经被正确的删除掉了。

重新处理重复的请求也会降低服务器的性能,因为服务器花很多时间去做它不应该做的事情。有时候这会使情况变得更加糟糕,因为重传一般都发生在服务器过载的时候。

因此,有必要检测以及正确地处理重传的请求。为了达到这个目的,服务器将最近收到的请求放到高速缓存中。如果一个请求的 xid,过程号以及客户 ID 同高速缓存中的一个请求完全相同,那么服务器便将其视为是重传的(这并不总是对的,因为有些客户可能为两个不同的用户产生出相同的 xid)。这个高速缓存一般叫做"重传高速缓存"或"xid 高速缓存"。

最初的 Sun 参考端口仅为 CREATE,REMOVE,LINK,MKDIR 和 RMDIR 请求维护了这样一个高速缓存。只有在请求失效时它才检测高速缓存,看是否由于重传请求才导致失效。如果是,它向客户发回一个成功应答。但这是不够的,因为这种方法仅堵住了一部分循环漏洞却又引发了其他的一致性问题。而且,它也没有解决性能问题,因为服务器直到在它处理了请求之后才去检测高速缓存的。

[Jusz89]详细地分析了要解决重传尚存在的问题。基于此,Digital 在 Ultrix 中修改了 xid 高速缓存。新的实现将所有的请求都缓存起来,并且在每次处理一个新请求之前都要检查高速缓存。每个高速缓存项中都包含着请求标识(客户 ID,xid,过程代号),一个 state 域和一个 timestam 域。如果服务器发现请求已在高速缓存中,并且请求的状态是正在进行,它便将重发的请求丢弃掉。如果状态是"完成",并且如果时间戳表明请求是最近完成的(在宽度为 3～6 秒的丢弃窗口内),服务器丢弃掉重复的请求。如果超过了丢弃窗口范围,且请求是幂等请求,服务器处理该请求。如果是非幂等请求,服务器向客户发送一个成功应答;否则,它重试请求。高速缓存项按最近最少使用方式循环使用;这样,如果客户继续重传的话,服务器最终将重新处理请求。

xid 高速缓存可以帮助消除重复操作,因此提高了服务器的性能和正确性。还可以进一步改善。如果服务器缓存了应答信息和 xid 信息,它可以通过将缓存的应答重传的方法处理重复的请求。在丢弃窗口之内到达的重复请求仍然可以被丢弃掉。即使对于幂等请求,这也会进一步降低由于重复处理带来的时间浪费。这种方法需要有足够大的高速缓存来存储所

有的应答信息。像 READ 和 READDIR 请求的应答信息可能会很大。因此最好将请求从 sid 高速缓存中去掉，在必要时重新处理它们即可。

10.8　专用 NFS 服务器

大多数的 UNIX 供应商通过为工作站在原有的框架上改组，加上更多的磁盘，网络适配器和内存的方法设计文件服务器。这些系统运行的操作系统大多是 UNIX 的变体，而这些变体主要是为了用在多道程序环境中而设计的，并不适合用于高吞吐量的 NFS 服务中。有些供应商设计了一些专门用于 NFS 服务器的系统。这些系统有的是在原来的 UNIX 上增加某些功能，有的则是将操作系统整个替换。本节介绍两种这样的体系结构。

10.8.1　Auspex 功能性多处理器结构

Auspex 系统是高端 NFS 服务器，它进入市场时采用一个叫做 NS5000 的功能多处理器(FMP)系统。FMP 体系结构[Hitz 90]将 NFS 服务器分为两个主要子系统 —网络和文件系统⑦。它使用了很多共享同一底板的 Motorola 680x0 处理器，每一个处理器专用于一个子系统。这些处理器上运行着一个小的功能多处理内核(FMK)，处理器之间通过高速度的消息传递来通信。每个处理上器运行着一个修改了的 SunOS 4.1(加入了 FMK 支持)，并且提供了管理功能。图 10-5 显示了其基本设计。

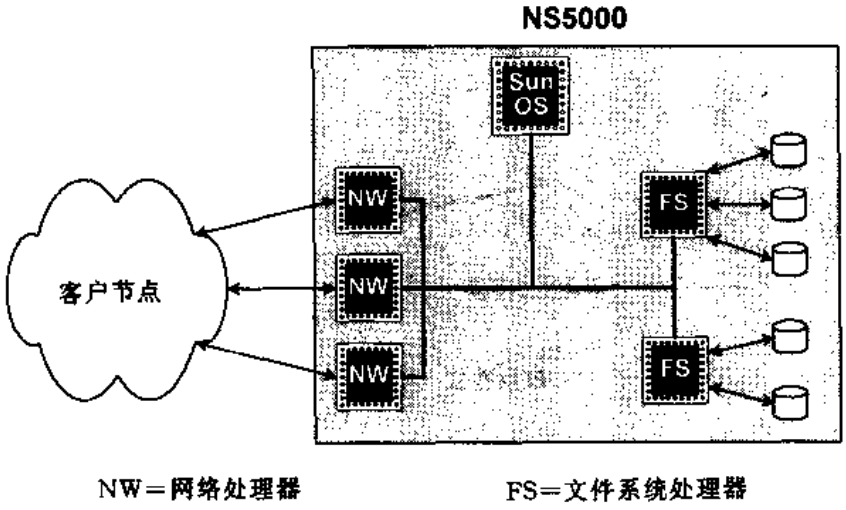

图 10-5　Auspex NS5000 体系结构

UNIX 前端可以直接同每个功能处理器通信。它通过标准网络驱动程序同网络处理器对话，通过一个实现了 vnode/vfs 接口的特殊的本地文件系统跟文件系统处理器对话。UNIX 处理器也可以直接通过一个特殊的设备驱动程序访问存储器，该设备驱动程序代表一个 Auspex 磁盘，并将磁盘 I/O 请求转化为 FMK 消息。这样，像 fsck 和 newfs 之类的工具无需改变就可以工作了。

---

⑦　最初的设计将存储功能分离到另一个子系统中。最新的产品线不再有独立的存储处理器。

正常的 NFS 请求都绕过 UNIX 处理器。请求一般先到达一个实现了 IP,UDP,RPC 和 NFS 层的网络处理器。接着网络服务器将请求传递到文件系统处理器,文件系统处理器可以向存储处理器发送 I/O 请求。最后,网络处理器将应答消息传回客户。

FMK 内核支持一组基本原语,包括轻量级进程,消息传递和内存分配。FMK 去除了与传统 UNIX 有关的很多东西后,它的上下文切换和消息传递变得非常快。例如,FMK 没有内存管理,其进程永远不终止。

这种结构为高吞吐量 NFS 提供了基础,Auspex 系统也因此成为高端 NFS 市场的佼佼者。最近其地位受到了来自 Sun 和 DEC 提供的基于簇的 NFS 服务器的挑战。

10.8.2 IBM 的 HA-NFS 服务器

HA-NFS 是由 IBM 设计的高可用 NFS 服务器,[Bhid91]描述了其原型实现。HA-NFS 将高可用 NFS 服务遇到的问题分为三类——网络可靠性,磁盘可靠性和服务器可靠性。它使用磁盘镜像和可选的网络备份来解决前两个问题,而使用协作服务器来解决第三个问题。

图 10-6 展示了 HA-NFS 的设计。每个服务器都有两个网络接口,也因此有两个 IP 地址。一个服务器可以指定其中一个网络接口为其主要接口,用它来进行正常的各项操作。只有当其他服务器失效时它才使用其第二个接口。

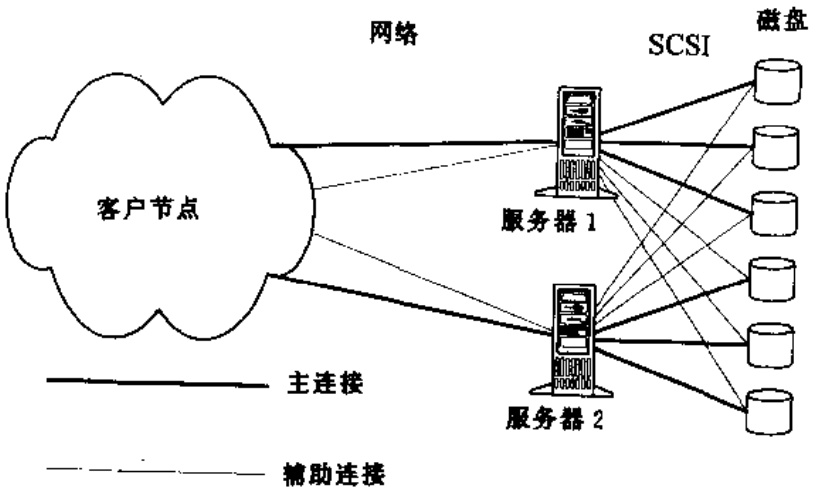

图 10-6 HA-NFS 配置

HS-NFS 也采用了双端口磁盘,这些磁盘通过一个共享的 SCSI 总线同两个服务器连在一起。每个磁盘都有一个主服务器。在正常情况下,只有该主服务器才能访问这个磁盘。当主服务器失效时,第二服务器便接管磁盘。这样磁盘便分为两组,分别为不同的服务器服务。

两个服务器,通过周期性的心跳消息实现彼此之间的通信。当其中一个服务器没有从另外一个服务器收到心跳消息时,它发起一系列的检测,以确认另外一个服务器是不是真的失效了。如果是,该服务器要进行一些失效恢复工作。它首先控制住失效服务器的磁盘,然后将其第二个网络接口的 IP 地址设为失效服务器的主网络接口地址。这样它便可以接受以及处理原本要发往那个失效服务器的消息。

这个接管过程对客户是透明的,客户可能只能感到性能有所下降。当错误恢复过程正在

进行时服务器可能不响应消息。一旦恢复过程结束，存留下来的服务器的速度可能会变慢，因为它要承担原本由两个服务器承担的任务。不过其服务不会受到损害。

每个服务器上都运行IBM'S AIX操作系统，该操作系统使用元数据日志文件系统。HA-NFS加上了有关RPC请求的信息，以便为NFS操作增加日志项。当一个服务器在失效恢复中接管了一个磁盘，它重新运行日志，将文件系统恢复到一致性状态，同时从日志中的RPC信息中恢复其重传高速缓存。这就在失效恢复中防止了由于重传而造成的不一致。两个服务器还交换使用NLM和NSM发出文件锁定请求的客户的信息[Bhid 92]。这样在其中一个服务器崩溃时加锁管理器状态仍然可以得到恢复。

有两种方法可以使得IP地址接管对用户是透明的。一种是使用特殊的硬件地址可以改变的网络接口卡。在失效恢复过程中，服务器同时改变即将成为失效服务器的主接口的第二接口的IP地址和硬件地址。如果没有这样的硬件，服务器可以利用地址解析协议(ARP)请求[Plum 82]来更新在客户中的<硬件地址，IP地址>映射项。

10.9 NFS安全性

任何一种网络应用要想提供跟本地系统一样的安全性是十分困难的。尽管NFS竭力提供UNIX语义，其安全性十分不足。本节主要看一下NFS在安全方面主要漏洞和一些解决方案。

10.9.1 NFS访问控制

NFS在两种情况下进行访问控制，一是在文件系统被安装时，一是在每一次NFS请求时。服务器维护了一个输出表，该表指定了哪一个客户机可以访问输出的文件系统以及访问权限是只读的还是可读写的。当一个客户试图安装一个文件系统，服务器的守护进程mountd检查输出表，拒绝非法客户的访问。这里没有针对特定用户的限制，也就是说在合法客户机上的任何一个用户都可以安装文件系统⑧。

在每一次NFS请求时，客户都要发送确认信息，一般是以AUTH_UNIX形式。该信息中包含着发出该请求的的进程的所有者的用户和组ID。NFS服务器使用该信息初始化一个凭证(credentials)结构。该结构可以被本地文件系统用作访问控制。

为了使上述方法有效，服务器和所有的客户必须共享同一个<UID,GID>空间。这意味着所有共享NFS的机器上的所有用户，一给定的UID只能是其中一个客户，而不能是其他人。如果在机器m1上的用户u1和机器之上的用户u2有同样的UID，NFS服务器会将两者混淆，并且会允许其中一个访问另一个的文件。

这是典型的工作组的主要问题。工作组中每一个用户都有其私有工作站，还有一个中心NFS服务器维护普通文件(在很多情况下，包括登录目录)。因为每个用户对他自己的工作站一般都有root权限，他可以使用任意<UID,GID>去创建新的帐户，这样就可能冒充了工作组中的其他人。这样一个冒充者可以随便访问被冒充者在服务器上的文件，而被冒充者并不知道这一切。冒充者不用编写复杂的程序，也不用修改内核或网络使可以做到这些。唯

⑧ 许多客户实现版本只允许特权用户安装NFS文件系统。

一的保护措施是只让那些可以信任和被监视的客户访问 NFS 文件系统。这个例子有力地说明了 NFS 不存在安全性这样一个事实。

还有其他的方法可以侵入 NFS。因为 NFS 依赖于在没有安全保护的网络上传输的数据，所以很容易写程序来模仿 NFS 客户，发送包含有假的认证数据的数据包，甚至可以使数据看似来自不同的机器。这样即使在机器上没有 root 许可的用户或在对服务器没有 NFS 权限的机器上的用户都可以侵入 NFS。

10.9.2 UID 重新映射

有很多方法可以阻止上面所说的侵入。第一种保护方法涉及到 UID 重新映射。也就是服务器为每一个客户都维护一个转换映射表，而不是所有的客户和服务器共享一个表。这个映射表定义了从由网络上收到的凭证到一个可以用于服务器的实体的转换。这个实体也是由一个<UID,GID>定义，但与客户发送的<UID,GID>有所不同。

例如，转换表可以指定从客户 c1 来的 UID 17 被转换到服务器上的 UID 33，而从客户 c2 来的 UID 17 被转换到服务器上的 UID 17，等等。它也定义了 GID 的转换。该表还定义了一些缺省的通配符的转换(如，来自一组可信用户的凭证不要求转换)。一般地，如果进来的凭证与任何表项或缺省规则不能匹配，那么该证明便被映射为用户 nobody，这样用户就只能访问任何人都可以访问的那些文件。

这样的一种 UID 映射可以在 RPC 层上实现。这意味着转换可以用于任何基于 RPC 的服务，并且这种转换在 NFS 解释请求之前便已完成。因为这种转换需要附加的处理，所以会降低基于 RPC 的服务的性能，即使某一服务不需要安全层也得进行转换。

另外一种方法是在 NFS 层通过将映射表同/etc/exports 文件合并的方式实现 UID 重新映射。这样服务器就可以为不同的文件系统实行不同的映射。例如，如果服务器将其/bin 目录输出为只读，将/usr 目录输出为可读写，它可能想将 UID 映射仅仅应用于/usr 目录，因为其中可能包含一些用户希望保护的敏感文件。这种方法的缺陷是每一种 RPC 服务都要实现它自己的 UID 映射(或者其他的安全机制)，这就可能导致重复的劳动和编码。

很少有主流 NFS 不提供任何形式的 UID 映射。NFS 的安全版本一般都使用有 AUTH_DES 或 AUTH_KERB 认证的 RPC(在 10.4.2 小节中描述)。

10.9.3 root 重新映射

一个相关的问题涉及到来自客户机的根访问。让所有客户机上的超级用户都对服务器上的文件有根访问权限显然不是一个好主意。一般的方法是服务器将来自所有客户的超级用户映射到用户 nobody。另外，有些实现使用/etc/exports 文件来指定 root 应该被映射哪个 UID。

尽管这种方法解决了超级用户访问的问题，以 root 登录的用户往往会遇到一些奇怪的限制。这些用户对 NFS 文件的权限比普通用户要少。例如，他们可能不能访问服务器上他们自己的文件。

上面说的很多问题并不仅仅局限于 NFS。传统的 UNIX 安全性框架是为单机、多用户环境设计的，仅就在这个范围它也是不足的。在网络中节点之间是相互信任的，因此网络的引入严重地危害了安全性，同时也引入了其他漏洞。这就导致了一些网络安全性和认证服务

的出现，其中在UNIX中最有名的是Kerberos[Stei 88]。有关网络安全性的详细讨论不在本书范围之内。

10.10 NFSv3

NFSv2早已变得十分流行，已经被移植到很多不同的硬件平台和操作系统上。但同时也进一步暴露了它的缺陷。尽管可以通过灵巧的设计解决某些问题，大部分问题是由协议的固有特性决定的。在1992年，来自几个公司的工程师们在波士顿、马塞诸塞州召开了几次会议，决定开发NFSv3.0[Pawl 94 Sun 95]。现在NFSv3已经出现商业版本了。DEC在其DEC OSF中支持了NFSv3，成为第一个支持NFSv3的公司⑨。Silicon和其他一些公司紧随其后。Guelph大学的Rick Macklem为4.4BSD实现了NFSv3.0，该实现可以以anonyomous ftp从snowwhite.cis.uoguelph.ca:/pub/nfs得到。

NFSv3解决了很多NFSv2的局限性。NFSv2性能问题是要求服务器在返回应答信息之前一定要将所有的修改提交给可靠的存储介质。这主要由于协议的无状态造成的，因为客户没有其他办法知道数据是否已被安全地存储了。NFSv3增加了一个COMMIT请求，这样NFSv3便可以异步写了。其工作过程如下：当一个客户进程往一个NFS文件中写入数据时，内核向服务器发送一个异步写请求，服务器可以将数据存在本地高速缓存中，并且迅速向客户机返回一个应答信息。客户内核保留写数据的一份复制，直到进程关闭文件。这时内核向服务器发送一个COMMIT请求，将数据写入磁盘，然后向客户返回success。当客户接受到服务器对其COMMIT请求的回答，便将它的写入数据的备份丢弃掉。

异步写在NFSv3中是可选的，有些客户或服务器可能不支持异步写。例如，如果客户没有足够的资源来缓存写入数据，那么它仍然可以使用原来的方式写入，也即服务器必须立即将数据写入。甚至对于异步写请求，服务器也可以以同步的方式完成，不过这要在应答消息中说明这一点。

NFSv2的另外一个问题是它仅仅使用32位的数据域来指定文件长度以及读写偏移量。这样，协议就不支持4G以上的文件。对于有些需要处理更大的文件的应用上述限制是不可接受的。NFSv3中将该域扩展为64位，因此它可以支持1.6×10^{19}字节大小的文件。

NFSv2的协议会产生过多的LOOKUP和GETATTR请求。例如，当一个用户通过ls-l读取一个目录时，客户向服务器发送一个READDIR请求，服务器便返回目录中所有文件的列表。于是客户对列表中的每一个文件都发送一个LOOKUP和GETATTR请求。如果一个目录很大，这种方法会导致过大的网络通信量。

NFSv3提供了一个READDIRPLUS操作，它可以返回目录中的文件的名字，文件句柄和属性。这样便可以使用一个NFSv3请求来替代原先的一系列NFSv2请求。READDIRPLUS请求必须小心使用，因为它将返回大量的数据。如果客户仅仅需要一个文件的信息，还是使用老的请求为好。

支持NFSv3的实现必须也要支持NFSv2。客户和服务器一般是使用双方都支持的协议的最高版本。当它们第一次接触时，客户使用其最高版本的协议，如果服务器不理解其请求，

⑨ DEC OSF/1现在称为Digital UNIX。

客户试着使用较低版本的协议，依此类推，直到它们找到一个双方都支持的版本。

NFSv3 成功与否只有让时间来说明。上面所描述的各种改变对 NFS 协议来说是很大的改进，应该能够产生出比较好的性能。NFSv3 也消除了 NFSv2 中的一些小问题。有些小的改变实际上降低了 NFSv3 的性能，但是预计使用异步写和 READDIRPLUS 获得的好处会远远超过这些小的性能降低。

10.11 远程文件共享(RFS)文件系统

AT&T 在 SVR3 UNIX 中引入了远程文件共享(RFS)文件系统，目的在于通过网络对远程文件进行访问。RFS 和 NFS 的基本目标类似，但是它们的体系结构和设计是截然不同的。

RFS 的主要设计目标是对远程文件和设备提供安全透明的访问，访问方式应该保留所有的 UNIX 语义。这就意味着 RFS 要支持包括设备文件和命名管道(FIFO 文件)的所有类型的文件，也要支持文件和记录加锁。其他重要的目标包括二进制兼容性和网络独立性。有了前者，已有的应用程序不需修改便可以使用 RFS(除了要处理一些新的错误代码)；而有了后者，RFS 就既可以用于局域网(LAN)，也可以用于广域网上(WAN)。

RFS 的最初实现只移植到了运行于不同硬件上的其他 SVR3 UNIX 系统上。这主要是因为 RFS 使用了 SVR3 中的文件系统开关表。SVR4 将 RFS 同 vnode/vfs 接口集成起来，从而使得 RFS 可以移植到其他的 UNIX 变体上去。这种重新设计是基于一次早期的从 RFS 到 SunOS 的移植[Char 87]。我们主要集中在 RFS 的基于 v 节点的实现上。

10.12 RFS 结构

跟 NFS 类似，RFS 也是基于客户/服务器模型的。服务器向外输出目录，客户安装这些输出的目录。任何一台机器可以是客户，也可以是服务器，还可以同时是客户和服务器。但 RFS 和 NFS 的相同点也仅止于此。RFS 体系结构是有状态的，而这正是正确地提供 UNIX 开放文件语义所必须的。这个特点对 RFS 的实现和功能都有深远的影响。

NFS 使用可靠的虚电路传输服务，TCP/IP 就是这样的一种协议。每一对客户/服务器在使用一条在第一次安装操作时建立一条虚电路。如果客户又安装了服务器上的其他目录，所有的安装多路复用同一条虚电路。虚电路在安装操作期间一直保持打开状态。如果客户或服务器之一崩溃了，虚电路便会中断，对方也会知道这次崩溃，因而可以采取合适的行动。

网络独立性可以通过在 STREAMS 框架之上实现 RFS(见第 7 章)以及使用 AT&T 的传输提供者接口(TPI)来实现。RFS 可以通过多个流来通信，因此可以使用同一台机器上的多个不同的传输提供者。图 10-7 显示了在客户和服务器之间通信建立的过程。

RFS 将服务器输出的每个目录同一个符号的资源名字联系起来。一个集中式的名字服务器将资源名映射到它们的网络地址。这样资源便可以在网络中移动；客户不用知道资源当前的位置便可以访问它们[10]。

---

[10] 当然，当有客户安装了它时，资源就不能移动。

因为 RFS 有可能用于大规模网络上，资源管理可能变得非常复杂。RFS 提供了域(domain)的概念，它是网络中的一组计算机的逻辑组合。这样，资源便可以以域名和资源名标识了，而且现在资源名可以仅在域内唯一，而不必全局唯一。如果没有指定域名，就使用当前域的名字。名字服务器可以仅仅保存它自己域中的资源信息而将其他请求发往它们对应域的名字服务器。

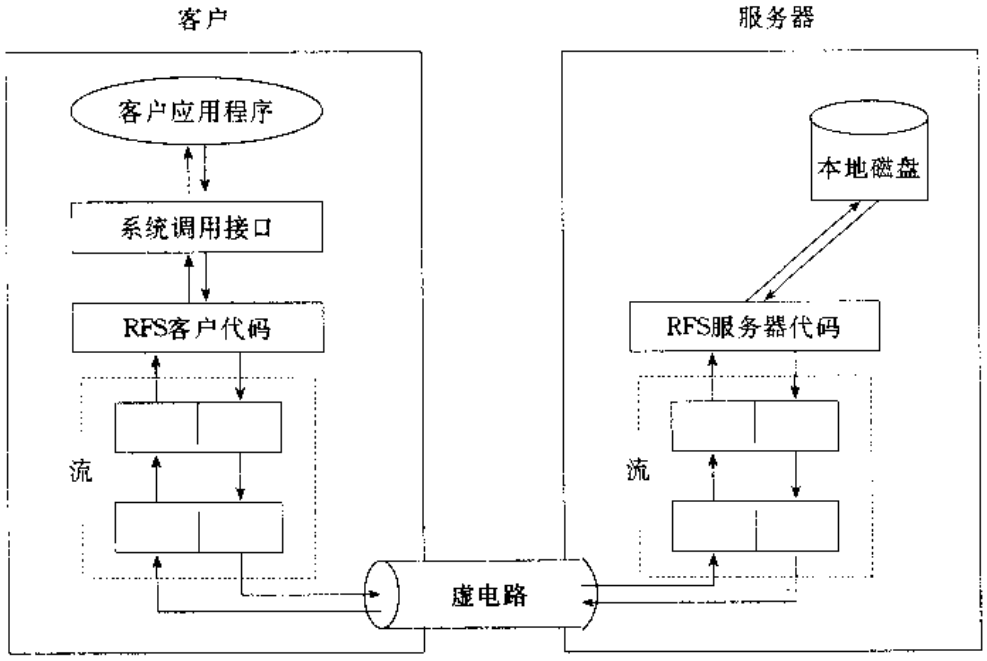

图 10-7　RFS 中的通信

10.12.1　远程消息协议

RFS 的最初设计使用了远程系统调用模型，该模型为每一个对远程文件操作的系统调用提供了一个 RFS 操作。对于每一个这样的操作，客户将系统调用的参数和有关客户进程的环境信息封装到一个 RFS 请求中。接受到请求后，服务器重新创建客户的环境，执行系统调用。客户进程此时被阻塞在调用点，直到服务器处理完请求，送回一个包含有系统调用执行结果的响应消息时它才被唤醒。接着客户解释返回来的结果，完成系统调用，将控制权返回给用户。这种实现被称为 RFS1.0 协议。

当将 RFS 同 vnode/vfs 集成在一起时，有必要为 RFS 实现每一个 v 节点操作。在移植到 SunOS 上时[Char 87]，每一个 v 节点操作都由一个或多个 RFS1.0 请求实现。例如，vn_open 可以只使用一个 RFS_OPEN 请求即可；而 vn_setattr 则需要一个 RFS_OPEN，然后一个或多个 RFS_CHMOD，RFS_CHOWN 和 RFS_UTIME。

SVR4 引入了 RFS 协议的一个新版本 RFS2.0。它提供了一组请求，可以直接映射 v 节点和 vfs 操作，因此可以更加简洁地同 vnode/vfs 集成在一起。不过，这带来了向前兼容的问题，因为网络上不同的机器可能运行着不同的 UNIX 发行版，因此也就有不同版本的 RFS。

为了解决这个问题，SVR4 客户和服务器必须同时理解旧版本和新版本的 RFS 协议。当连接建立时(在第一次 mount 操作期间)，客户和服务器交换它们都能处理和理解的协议的信息。因此只有当两台机器都支持 RFS2.0 时，它们才能使用 RFS2.0。如果一台机器运行

SVR4，而另外一台运行 SVR3，那么它们只能使用 RFS1.0 协议。

这就要求 SVR4 以两种方式实现每一个 vnode 和 vfs 操作。当同其他 SVR4 机器对话时使用一种，而同老版本的系统通话时，使用另外一种。

10.12.2 有状态操作

RFS 是有状态的文件系统，这意味着服务器必须维护客户的状态。服务器记录哪些文件已被客户打开，并且将这些文件的引用计数增加。另外，服务器跟踪由每个客户保留的文件/记录锁以及每个命名管道的读者/写者计数。服务器还维护着所有安装了其文件系统的客户表。这个表保存了客户的互联网地址，安装的文件系统和虚电路连接的参数。

RFS 的有状态的特征要求服务器和客户在其对方崩溃时能被通知到，从而可以采取合适的恢复行动。这将在 10.13.3 小节中详细描述。

10.13 RFS 实现

RFS 协议允许把客户和服务器功能彻底地分开。安装操作由一个远程安装程序和名字服务器共同完成。下面我们逐个看一下各个组成部分。

10.13.1 远程安装

RFS 服务器可以使用系统调用 advfs 来输出一个目录。advfs 的参数包括输出目录的路径名，同该目录相联系的资源名，和所有被授权可以访问资源的客户机列表。另外，在虚电路建立的过程中，服务器可能要进行口令检查。

服务器调用 advfs 输出一个目录。advfs 为目录在内核的资源列表中创建一个表项（见图 10-8）。该表项包含了资源名，指向输出目录的 v 节点的指针和被授权的客户列表。它也包含所有安装该资源的客户组成的列表的头部。在 SVR4 中系统调用 advfs 被 rfsfs 调用替代，rfsfs 包含有几个子功能，其中包括输出文件系统。

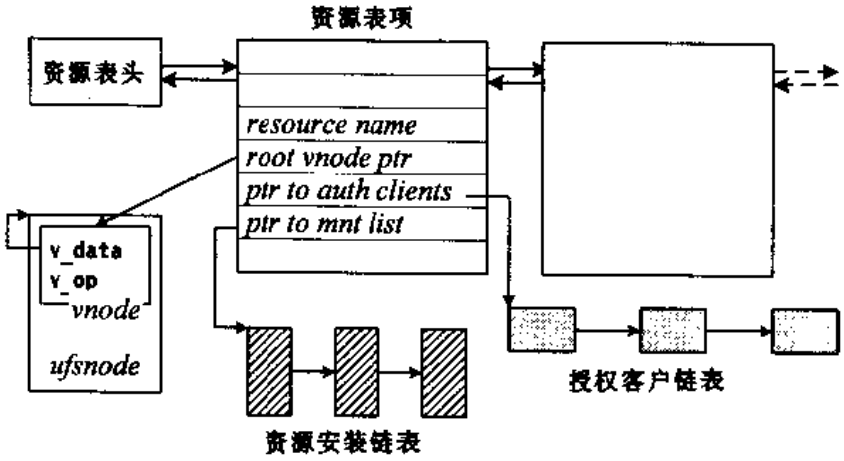

图 10-8 RFS 资源链表

图 10-9 描述了 RFS 服务器，名字服务器和客户之间的交互。服务器调用 adv(1)命令，

在名字服务器上为它输出的资源注册。在稍后的某个时候，客户可以通过下列命令来安装一个 RFS 资源：

```
mount -d <RNAME> /mnt
```

其中<RNAME>是资源的名字。mount 命令要求名字服务器为它的资源查到网络地址，并且在需要时建立一条虚电路。接着它使用本地安装点的路径名和一个指定 RFS 类型的标志来调用 mount 系统调用。系统调用中与 RFS 有关的参数包括虚电路指针和资源名字。

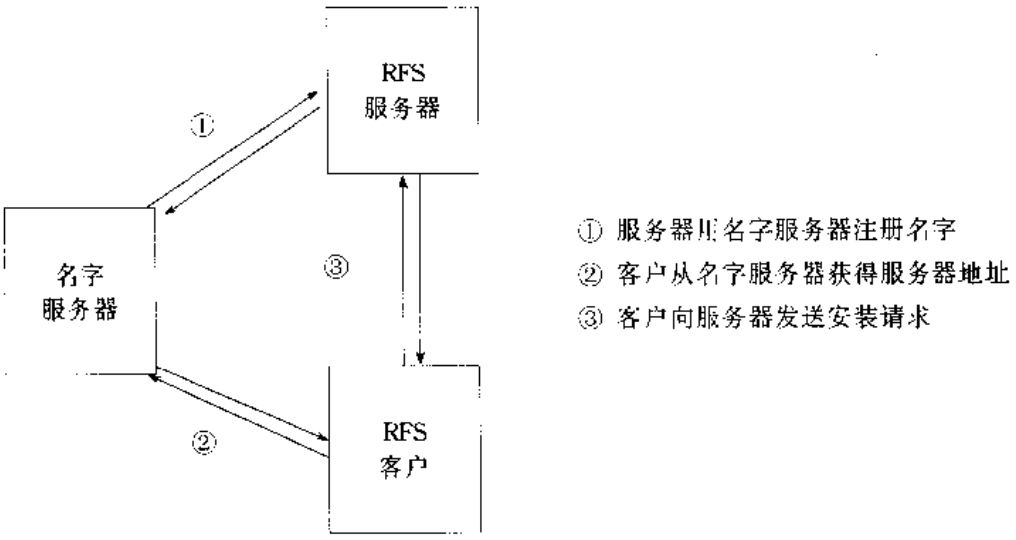

图 10-9　RFS 文件系统中的安装

系统调用 mount 向服务器发送一个 MOUNT 请求，其中用其符号资源名作为参数，服务器找到资源的表项，确认客户有权安装资源，然后便在其资源安装列表中为客户加入一项。它把输出目录的引用计数添加到远程安装的总数中，并向客户发回一个成功的应答，该应答中包含有一个安装 ID，客户可以在将来对该文件系统的请求中将该 ID 送回来，服务器可以使用安装 ID 迅速找到相应的资源。

当客户收到 MOUNT 响应，它结束其处理过程，建立 vfs 项，将安装 ID 存放在文件系统特定的数据结构中。同时它也为资源的根目录建立 v 节点。一个 RFS 的 v 节点中的 v. data 域指向一个叫做发送描述符的数据结构，它包含有关虚电路和服务器的信息，可以用来定位相应的本地 v 节点的文件句柄。

客户和服务器的第一个安装操作建立一条虚电路。随后的所有安装(和 RFS 操作)都多路复用该虚电路。直到最后一个资源卸出之后，虚电路才会断开。安装命令使用传输接口同服务器上的一个守护进程建立一个链接。一旦链接建立，客户和服务器便协商运行时参数，其中包括协议版本号，硬件结构类型等。如果两台机器有着不同的体系结构，它们要用 XDR 来对数据进行编码。

最初的虚电路建立过程是以用户态使用系统上的标准网络接口来进行的，一旦建立之后，用户便使用系统调用 rfsys 的 FWFD 功能将虚电路传到内核中。

10.13.2 RFS 客户和服务器

客户既可以使用路径名也可以使用文件描述符来访问一个 RFS 文件。内核转换路径名时可能会遇到一个 RFS 安装点。在 RFS1.0 中，客户将路径名的剩余部分发送到服务器，服务器将其解析出来，将结果返回给客户。因为 RFS2.0 允许在 RFS 目录中安装其他文件系统，于是在路径名解析中客户要检测是否有这种情况，如果有，客户为了保证能够正确地处理安装，每次只解析一个分量，如果不是这样，客户便可以将整个路径名送到服务器去。服务器返回一个句柄，客户将其存放在 v 节点的私有数据结构中。图 10-10 中显示了在客户和服务器上的数据结构。

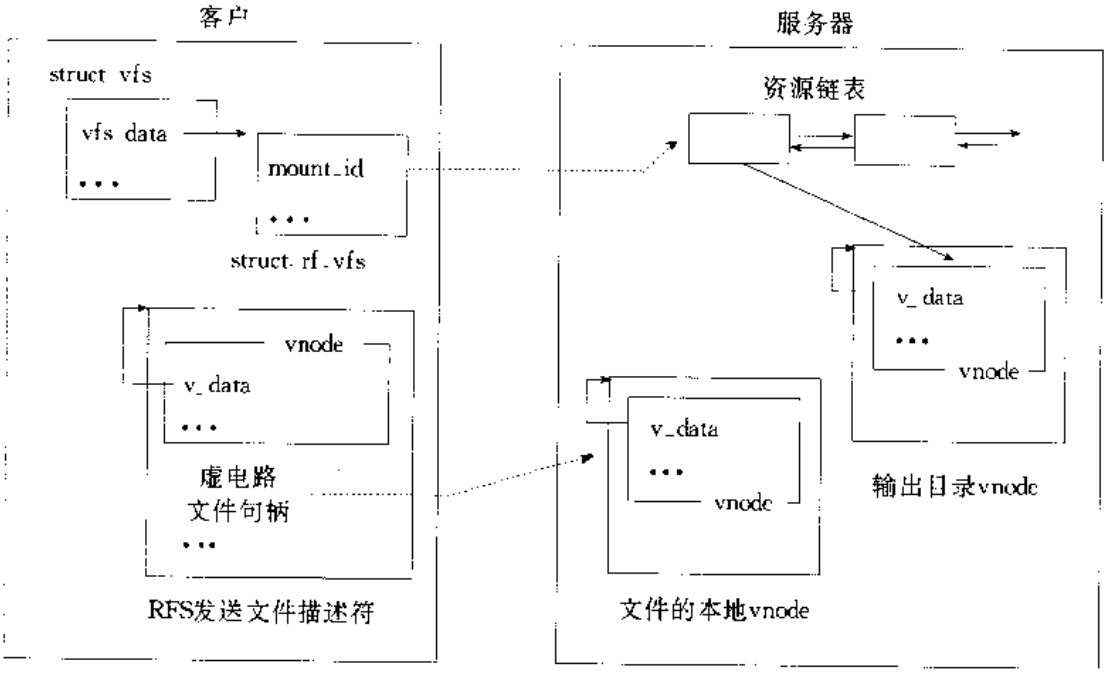

图 10-10 RFS 数据结构

对文件随后的操作通过文件描述符来访问，这是因为内核可以使用文件描述符来找到 v 节点。v 节点操作会调用 RFS 客户函数，从 v 节点中抽取出文件句柄并且将其通过 RFS 请求传递到服务器。文件句柄对客户来说是不透明的。一般来说，它只包含一个指向服务器上的 v 节点的指针。

RFS 服务器是作为一个或多个独立的守护进程运行的。这些守护进程全部在内核中运行，以避免在用户态和内核态之间进行上下文切换。每个守护进程监听到来的请求，在为一个请求服务完毕之前不能为另外一个服务。当为一个请求服务时，守护进程使用证明，资源限制和消息传来的其他一些属性来假设客户进程的身份。守护进程在等待资源时可能会进入睡眠状态，守护进程和其他进程一样接受进程调度。

10.13.3 崩溃恢复

有状态的系统需要复杂的崩溃恢复机制。客户和服务器都要能够检测和处理对方的失效。当其中的一个崩溃时，它们之间的虚电路会断开，于是底层的传输机制会将这件事通知

RFS。当网络失效时虚电路也会中断，这可以用跟RFS同样的方法对待。客户和服务器的恢复机制是不同的。

当一个客户崩溃时，服务器必须要取消与该客户有关的状态。为了做到这一点，服务器为每个i节点维护一个客户级的状态信息，并且用它来进行以下的操作：

1. 将i节点的引用计数减一，该值应该和崩溃掉的客户所保持的引用计数相等。
2. 将客户使用的在已命名管道上读者/写者锁释放掉。
3. 将客户持有的文件/记录锁释放掉。

当服务器崩溃之后，所有正在等待服务器响应的客户进程都被唤醒，系统调用返回一个ENOLINK错误信息。所有指向该服务器上的文件的RFS i节点都被标记，这样任何想对这些文件进行访问的操作都会返回一个错误。一个用户级的守护进程(rfudaemon)被唤醒以处理其余的恢复任务。

10.13.4 其他问题

进程池 因为RFS允许客户透明地访问远程设备和已命名管道，所以客户进程有可能因为等待设备和管道I/O而阻塞很长时间[11]。因此服务器需要维护一个进程的动态池。如果当一个客户请求到达而所有的服务器都很忙，进程池会产生一个新的进程。这个动态池有上限(可动态调节的)，最后一个进程不允许睡眠。

远程信号 UNIX中，信号可以终止一个系统调用。这个功能可以被延伸用于RFS环境中。假设一个客户进程由于等待服务器的请求而被阻塞，如果该进程收到一个信号，客户内核向服务器发送一个signal请求。该请求使用系统ID和PID标识一个进程。服务器使用这些信息来找到处理该客户请求的正在睡眠的进程并且将信号发送给它。

数据传输 对于本地read和write系统调用来说，copyin()和copyout()例程将数据在缓冲区和用户地址空间之间移动。而RFS操作必须要从请求消息将数据移出或将数据移入到请求消息中。在进程表项中有一特殊标志可以将进程标识为一个RFS服务器，copyin()和copyout()操作例程检查该标志以便进行合适的数据传输。

10.14 客户端高速缓存

如果每一个I/O操作都需要远程访问，那么RFS的性能会变得让人难以忍受。为了维护合理的吞吐量，使用某种形式的高速缓存是必要的。分布式高速缓存的问题在于必须维护可能同时在多个客户缓冲区和服务器中的数据的一致性。正如我们在10.7.2小节中所看到的，NFSv2对此采取了相当大胆的态度，这对于并不想保留完全的UNIX语义的无状态系统来说是可以接受的。而RFS是有状态协议，因此需要更加谨慎，必须要建立一个可操作的一致性协议。

在SVR3.1[Bach 87]的RFS中引入了客户端高速缓存。这个高速缓存在安装时被调

[11] 实际上，除非在完全的同类系统中，否则使用RFS来共享设备是很容易出问题的。系统调用语义间的细小差别就不可能共享设备。

用，不过服务器也可以根据需要使其无效(这对于那些想自己进行数据缓冲的应用程序是有意义的)。RFS 高速缓存被设计为可以提供很强程度的一致性，这样就可以保证用户永远不可能从缓存中取到过时的数据。

缓存是严格写通的，客户在将本地缓冲的数据复制更新之后，迅速将写的数据送往服务器。虽然这并不能提高写操作的性能，但是对于数据一致性是非常重要的。当且仅当要求的数据都在高速缓冲区时，系统调用 read 才能将缓冲的数据返回。如果任何部分数据不在缓冲区内，客户在一次操作中将数据从服务器上全部读取出来。如果只有在缓冲区找不到的块从服务器上读取，我们就不能保证读操作的原子性，因为从高速缓存中读取的块和从服务器上读取的数据可能不是文件的同一状态。

RFS 高速缓存共享本地块高速缓冲资源。缓冲区的一部分预留给 RFS，有些用于本地文件，而其余的则两者都可以用。这就防止了本地文件独占缓冲池。缓冲区按最近最少使用方式重用。

10.14.1 高速缓存一致性

强高速缓存一致模型可以保证每一次读取返回的数据等价于服务器在此时的文件映像。服务器的映像反映了文件的磁盘副本，同时也反映了服务器的高速缓冲区中最近使用的块。

任何对文件的修改，不管是由服务器上的用户还是由客户上的用户发出，都会使其他客户机高速缓冲区中的数据块变为无效。解决这个问题的方法是每次写操作都要通知受到影响的客户，但是这会导致过多的网络通信，从而使得这种方法不可行。RFS 提供了一种更加有效的解决一致性问题的办法。它将客户分成已经打开文件和已经关闭文件两类。

只有被多个客户共享的文件才需要一致性协议。如果一个远程文件被同一台机器上不同进程所共享，高速缓冲一致性可由客户自己提供，不用涉及到服务器。

图 10-11 显示了 RFS 是怎样维持高速缓冲一致性的。当服务器收到第一个写操作且写的对象文件正被多个客户打开，那么服务器将写操作挂起，向所有打开该文件的客户发送一个清除消息。这些客户将文件所有被缓存的数据置为无效并且暂时关闭对该文件的缓存。随后的对该文件的读操作将绕过高速缓存区而直接从服务器上读取。当写进程关闭文件或当自从上次修改已经过去了一段时间后，缓存重新变得有效。一旦所有的客户将高速缓存置为无效并且承认消息，写请求便被恢复。

可能有些客户虽然已经关闭掉了文件，但是文件的一些数据块仍然在高速缓存中。那么当他们重新打开文件时他们可能会读取到过时的缓存数据。我们必须防止这种情况的发生。解决方法是将每一个文件都与一个版本号联系起来。每次文件被修改时，这个号都会增加。对每一次打开请求，服务器都要返回一个版本号，客户将该版本号存在它的高速缓存中。如果文件在客户上一次关掉它之后被修改了，那么当客户试图重新打开文件时会得到一个不同的版本号。此时客户要刷新所有与文件相关的块，这样就可以保证客户不会访问到陈旧的数据。

在正常情况下，RFS 的一致性机制可以实现很强的一致性，而且代价也不太高。但是当其中的一个客户崩溃或者变得不响应时，就会发生问题。因为此时该客户响应高速缓存无效的请求要花费很长的时间，这可能会妨碍其他节点完成它们的操作。这样，单个的不正常的

客户可能会使整个网络产生问题。从总体效果来看，使用高速缓存带来的好处比维持一致性所花的代价更值得。对于有一个到五个客户的工作组，使用 RFS 缓存可能会使性能提高两倍(相对于以往的实现)。

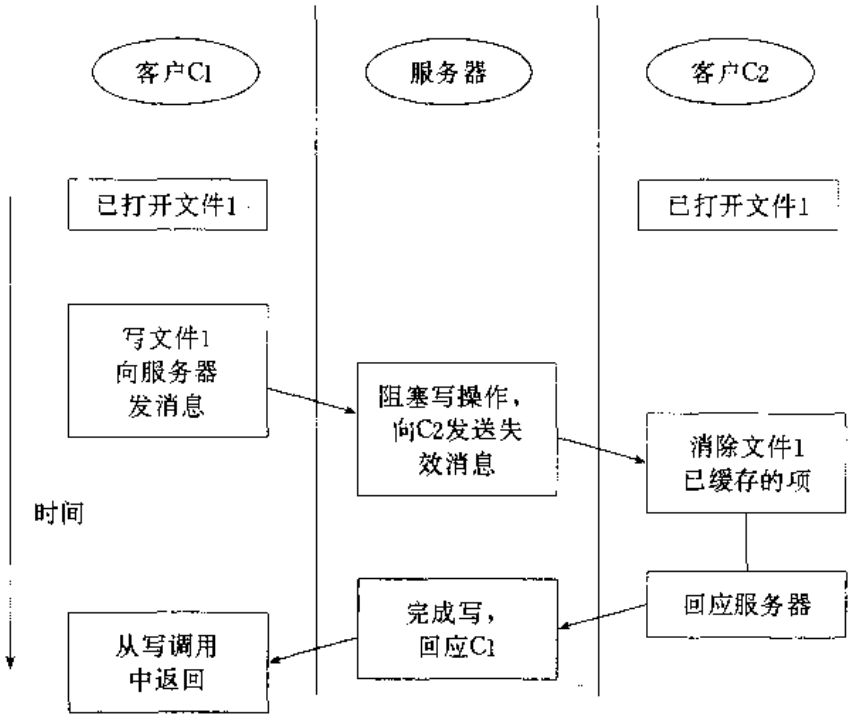

图 10-11　RFS 的缓存一致性算法

10.15　Andrew 文件系统

NFS 和 RFS 主要用于小型局域网上，并且客户机数目有限。当将它们用于跨越若干建筑或者成百上千个客户时(比如大学校园)，其效果很不理想。1982 年卡内基·梅隆大学和 IBM 联合建立了信息技术中心(ITC)为教育计算开发一些底层结构。它们的重要目标就是 Andrew 文件系统(AFS)。Andrew 是一个能够扩展到几千个用户的分布式文件系统[Morr 86]。

ITC 发行了几个版本的 AFS，AFS3.0 是其中的顶峰之作。其后，针对 AFS 的工作转移到 Transarc 公司去，该公司由 Andrew 的最初的一些开发者组成。在 Transarc，AFS 演变为 OSF 的分布式计算环境(DCE)的分布文件系统(DFS)组成部分。下节描述 AFS 的设计和实现。10.18 节讲述 DCE DFS。

除了可扩展性外，AFS 的设计者们指定了几个重要的目标[Saty 85]。AFS 必须是 UNIX 兼容的，因此 UNIX 二进制代码不用修改便可以在 AFS 客户上运行。它还必须为共享文件提供一个统一的位置无关的名字空间。用户可以在网络上任意节点访问他们的文件，不用停机就可以将文件从一个位置移到另外一个位置。它必须是容错的，一台服务器或者网络组件坏了之后系统仍然是可用的。错误必须被隔离在错误发生点附近。它应该不用信任客户工作站或网络便可以提供安全性。最后，其性能要与分时系统接近。

10.15.1 可扩展的结构

要想使得一个分布式文件系统具有可扩充性会碰到三个重要问题。如果用一个服务器来处理很多个客户，那么服务器会拥塞，网络也会超载；同样，如果客户的高速缓存容量不足也会导致网络通信量大增；最后，如果服务器负责所有操作的处理，那么服务器也很快会超载。一个可扩展的系统必须要很好地解决上面三个问题。

AFS通过将网络分成不同的互不依赖的簇来解决网络拥塞和服务器超载的问题。Andrew不像NFS和RFS那样，它使用专用的服务器。每台机器要么是客户，要么是服务器，却不能同样既为客户又为服务器。图10-12显示了一个AFS网络的组织。每个簇都包含一些客户，还包含有一个服务器，其上包含着这些客户感兴趣的文件，比如客户工作站所有者的用户目录。

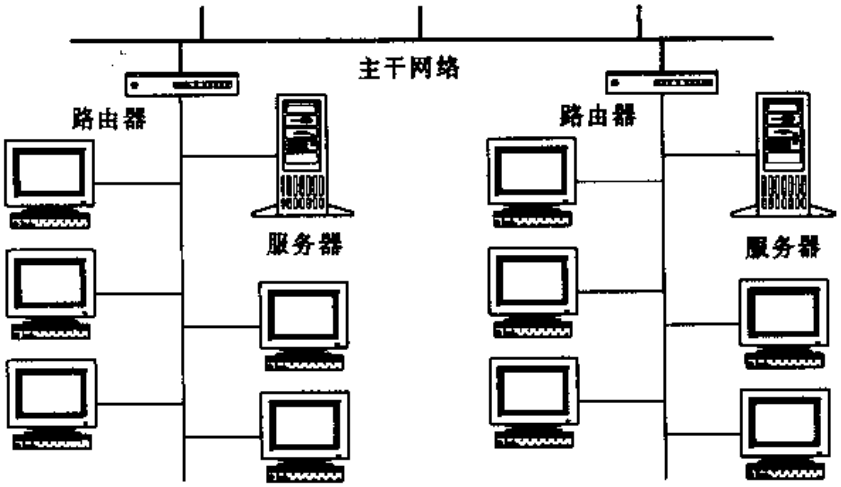

图10-12 AFS网络组织

这种配置使得客户访问在同一网段上的服务器中的文件变得十分迅速。当然用户也可以访问其他服务器上的文件，但是速度会慢很多。网络可以动态配置，以使各个网络段和服务器的负载能够平衡。

AFS使用主动式文件缓存和一个有状态协议尽量减少网络通信量。客户将它们最近访问的文件放在本地磁盘中。AFS的最初实现将整个文件都缓存起来。但是这对大型文件来说是不实际的，于是AFS3.0将文件分为64K大小的块，然后单独缓存这些块。AFS服务器主动地参于客户的高速缓存区管理，其方法是每当客户缓冲的数据变为过时时，服务器便通知客户。10.16.1小节更加深入地描述了该协议。

AFS将名字查找的任务从服务器转移到了客户，这样就减轻了服务器的负担。客户将所有的目录都缓存起来，自己进行路径名解析。10.16.2小节对此作了详细说明。

10.15.2 存储和名字空间组织

AFS服务器集合(称为Vice，据传言是代表高度集成的计算环境之意)维护着所有共享文件。AFS以逻辑单位"卷"来组织文件。卷是相关文件和目录的集合[Side 86]，它形成共享

文件树中的子树。例如，一个卷中可能包含有一个用户的所有文件，一般地，一个磁盘分区中可以包含几个小的卷。而一个大的卷可能要跨越几个磁盘。

卷是不同于分区的另外一个文件系统存储单位。这种划分有很多好处。卷可以从一个位置移到另一个位置，而不用影响活动的用户。这可以用来调节负载平衡或者调整用户持续移动。如果一个用户将其工作站从网络中一个位置移动到另一个位置，系统管理员可以将用户的卷移到用户所在的本地服务器上去。卷的出现也使得文件的大小可以超过一个磁盘的容量。可以将只读的卷复制放在不同的服务器上以提高性能。最后，每个卷可以被单独备份及恢复。

AFS 提供了一个与存储位置无关的统一的名字空间。文件同一个 fid 来标识，fid 中含有卷 ID、v 节点号，和一个 v 节点唯一性标识符。由于历史原因，AFS 术语 v 节点表示一个 Vice i 节点；因此 v 节点号是卷的 i 节点列表的索引。唯一性标识符是一个产生数，每次 v 节点被重用时都会增加一定值。

卷位置数据库提供了位置独立性和位置透明性。该数据库在卷 ID 和卷的物理位置之间作了一次映射。该数据库在每个服务器上都有副本，使其不至于成为瓶颈资源。如果卷被移到其他位置，原来的服务器上会保留有其去向的信息，这样其他服务器上的数据库就不需马上更新。当卷正在被传输到其他机器的过程中，原来的服务器仍然处理更新操作。有时候为了传输最近的改变而停止卷的工作。

每个客户工作站必须有一个本地磁盘。该磁盘上包含一些本地文件和一个用于将共享文件安装于其上的目录。一般情况下，每个工作站将共享树安装到同一个目录底下。本地文件包括为了一些最基本操作而必须保留的系统文件，还有用户为了安全和性能等原因而设置为本地的文件。因此每个客户看到的是一个共享名字空间和它自己的本地文件。本地磁盘也充当最近被访问的共享文件的高速缓存区。

10.15.3 会话语义

在一个集中式 UNIX 系统中，如果一个进程修改一个文件，其他进程在下一次读系统调用就会看到新的数据。在一个分布式文件系统中要强制使用 UNIX 语义会导致网络流量过载并带来性能下降。AFS2.0 使用了一个不甚严格的一致性维护协议，会话语义。该协议在每次文件打开或关闭时才进行高速缓存一致性操作。只有在文件被关闭时客户才将被修改数据刷新到服务器上。此时，服务器通知其他客户告诉它们其缓存数据已经无效。客户也不是在每次对文件进行读写访问时都要进行一致性检查，而是等到下一次文件打开时才进行。因此在不同机器上的用户在 open 和 close 系统调用的粒度下看到对共享文件的修改，而不是像在 UNIX 中那样是在 READ 和 WRITE 系统调用粒度下。

因为元数据的更新必须立即传递到服务器，所以 AFS 为元数据操作提供了很强的保护措施。例如，如果在一个客户上完成了一个 rename 系统调用，那么网络上的任何机器都不能使用旧文件名字将文件打开，而必须使新的文件名字。

会话语义的一致性保证要比 UNIX 语义弱得多。AFS3.0 在每次读或写的时候检查数据有效性，因此就会提供更好的一致性。即使如此，其语义较 UNIX 语义仍然有缺陷，因为客户直到文件关闭时才将数据改变刷新到服务器。AFS 的新化身 DFS 通过令牌传递机制达到了 UNIX 语义要求，我们将在 10.18.2 小节中详细描述。

10.16 AFS 实现

AFS 中客户和服务器是分开的。服务器的集合被称为 Vice，而客户工作站被称为 Virtue。服务器和客户都使用某种风格的 UNIX 作为其底层操作系统。AFS 的最初实现使用了简单的用户级进程来实现客户和服务器的功能。内核被稍微修改了一下，以便识别对共享文件的引用并且将它们转发到被称为 Venus 的 AFS 客户进程。在 AFS3.0 中，客户内核包括一个 AFS 缓冲管理器，它通过 vnode/vfs 接口提供 AFS 功能。服务器就是一个单一的，多线程的用户进程(在传统的系统之上使用用户级线程库)。在最近版本中，大部分服务器的功能都迁移到客户上去了，并且在守护进程的上下文中运行。

10.16.1 缓存以及一致性

高速缓存管理器[Spec 89]为客户上的 AFS 文件实现了 v 节点操作。当一个客户首次打开一个 AFS 文件时，高速缓存管理器将整个文件读过来(对于大于 64K 的文件，每次传回 64K 块)并且将它作为客户本地文件系统的一个文件缓存起来。高速缓存管理器将所有的 read 和 write 调用重定向到该缓冲的备份中。当客户关闭掉文件时，高速缓存管理器将改变的数据刷新到服务器上。

当高速缓存管理器从服务器上将文件取回时，服务器也提供一个与数据有关的回调。回调可以保证数据是合法的。如果另外一个客户修改了文件并且将改变写回服务器，服务器必须通知其他所有对该文件持有回调的客户，这叫做废除回调。客户便将陈旧的数据丢弃掉，如果需要的话就从服务器上再取一次数据⑫。

客户将文件属性和文件数据分开来缓存。文件属性被缓存在内存中，而文件数据被缓存在磁盘中。对于文件属性，客户和服务器使用同样的回调机制。当客户改变了任何文件属性时客户便会通知服务器，服务器就会立即中断所有相关的回调。

在竞争条件下，回调废除机制会有一些问题[Kaza 88]。假设正当服务器发送一个消息废除对某个文件的回调，而此时客户从服务器取这个文件。如果文件数据到达晚了一点，客户就不知道这个废除的回调是应用于当前数据还是应用于文件的前一个回调的。因此也就不能辨认该回调是否有效。AFS 的解决方法是简单地将数据丢弃掉，从服务器中再取一遍。这可能会带来额外的网络通信，使操作减慢。不过这种情形需要几个事件同时出现才能发生，而这种概率是相当低的，因此不会经常发生。

如果暂时的错误阻止了回调废除消息的传送，则会发生严重的问题。在 AFS 中，客户可能在不与服务器接触的情况下运行很长时间。在这段时间内，客户错误地假设它缓存的数据是正确的。为了解决这个问题，客户定期检测与其有回调协定的文件服务器(缺省为每 10 分钟 1 次)。

回调机制意味着服务器是有状态的，并且服务器必须跟踪所有它为每一个文件发出的回调。如果一个客户修改了文件，服务器必须废除所有该文件的未完成的回调。如果该信息

---

⑫ 在 AFS 3.0 之前，客户只是在下一次打开操作时丢弃过时数据。这一特性连同很大的数据块尺寸使得这些实现并不能很好地适用于事务处理和数据库系统。

大到没法再管理的地步，服务器必须要废除一些已存的回调并且回收存储。客户必须为缓存的文件维持有效性信息。

10.16.2 路径名查找

一个可扩充的系统必须要防止服务器超载。方法之一是将一些操作从服务器迁移到客户上。比如，路径查找是一项极耗CPU时间的操作，AFS可以直接在客户机上处理。

客户机中缓存有符号链接和目录的信息。它也缓冲有卷位置数据库的项。客户每次只转换路径名中的一个分量。如果在其本地缓存中没有查到某目录，那么客户便从服务器中将整个目录都取回来。接着客户便可以在该目录中查找所需要的分量。目录项将分量名映射到其fid上，fid中包含有卷ID，v节点号和v节点唯一标识符。

如果客户知道卷的位置，它便同相应的服务器联系，从那里得到下一个分量(除非该分量已经在本地缓存中了)。如果不知道卷的位置，客户便从最近的服务器上查询卷位置数据库，并且将得到的应答缓存起来。客户将缓存的数据库表项作为线索。如果信息已经改变了，服务器将拒绝请求，客户必须查询服务器数据库以得到正确的位置。还有一种可能是卷已经被移走了，并且最近的服务器还不知道这件事。在这种情况下，客户首先尝试卷以前的位置；旧位置的服务器会有卷的迁移信息，因而会做出恰当的反应。

10.16.3 安全性

AFS把Vice(服务器的集合)作为安全性的边界。AFS认为用户工作站和网络是不安全的。AFS不在网络上传输未加密的口令，因为通过潜伏在网络上的计算机获取口令是轻而易举的事情。

AFS使用由MIT开发的Kerberos认证系统[Stei 88]。Kerberos客户通过应答来自服务器的提问来认证自己，而不是采用传输一个客户和服务器都知道的密码来认证。服务器将提问加密，加密使用的密钥服务器和客户都知道。客户将提问解密，然后使用同样的密钥将应答信息加密，然后将加密的应答信息送回服务器。因为服务器每次都使用不同的提问，因此客户不可能重用同一个响应。

[Hone 92]指出了AFS3.0使用Kerberos的几个漏洞。客户在其地址空间中保留了几个重要的未加密的数据结构，这些数据结构很容易受到在自己的工作站获得root权限的用户的攻击。这种用户可以遍历内核的数据结构获得其他用户已得到认证的Kerberos许可。而且，AFS3.0的提问响应协议容易受到网络上其他节点的攻击。这些节点可以向客户发回假冒的提问。在AFS3.1中Transarc公司弥补了这些漏洞。

AFS也为目录(但是不为单个的文件)提供了访问控制表(ACL)。每个ACL是一个偶对组成的数组。每个偶对中的第一项是一个用户或组名，第二项定义了赋予该用户和组的权限。对目录，ACL支持四种类型的权限— lookup，insert，delete和admihiste(修改该目录的ACL)。另外，对目录中的文件ACL允许三种类型权限- —read，write和lock。AFS也保留了标准UNIX权限位，用户必须同时通过这两种检查(ACL和UNIX权限)才能操作一个文件。

10.17 AFS 的缺陷

AFS 是一个高度可扩展的体系结构。在 CMU，AFS 在 1985 年中期投入使用。到 1989 年 2 月，AFS 支持 9000 个用户账号，30 个文件服务器，1000 个客户机和大约 45G 的存储空间。它同样也适用于大范围的文件共享。到了 1992 年春天，一共有 67 个 AFS 单元可以在世界范围内安装[Gerb 92]。用户可以使用 cd，ls 和 cat 等常规 UNIX 命令来访问分布在各个不同位置的文件，比如法国 Grenoble 的 OSF 研究机构，日本的 Keio 大学和华盛顿特区的国家康复中心。[Howa88]描述的一些功能测试表明，AFS 减少了服务器的 CPU 利用，减少了网络流量和远程操作的总时间。

不过客户的性能却不能令人满意[Stol 93]。AFS 客户使用本地文件系统来缓存最近被访问的文件块。当访问这些数据时，它必须要进行一些附加的极为耗时操作。除了访问本地文件外，高速缓存管理器在确保被缓存的数据的有效性的同时还要将 AFS 文件或块映射到本地文件。而且，如果请求跨越多个块，那么高速缓存管理器必须要将其分为几个小的请求，使得每个请求只对单独的一个块操作。结果，即使数据已经在高速缓存区中了，要访问一个 AFS 文件仍然要比访问一个本地文件多花一倍的时间。通过合适的调整，AFS 中文件数据无效和文件跨越几个块的情况可减少到 10～15%。

有状态的模型很难实现。高速缓存一致性维护算法必须要处理几种竞争条件和潜在的死锁情况。实现完成时，模型就已经远离 UNIX 语义了。它的一致性保证要比 UNIX 语义弱的多，因为客户只有在进程关闭文件时才将数据改变写回服务器。这就会导致很多意料不到的问题。由于服务器崩溃，网络失效或者其他一些像磁盘满等错误，客户不能将数据刷新到服务器中去。这有两个致命的后果。首先，当关闭 AFS 文件时失效的概率可能要比关闭本地文件时出错的概率更大。在很多情况下，应用程序在文件终止时才隐式地将文件关闭。其次，写操作常常在其不应该成功的时候成功(比如，使用写操作扩展文件但是磁盘却满了)。这两种情况都会对客户有意料不到的后果。

最后，将路径名查找的工作转移到客户减少了服务器的负担，但是却需要客户理解服务器的目录格式。在 NFS 中，目录信息是独立于硬件和操作系统的。

DFS 解决了 AFS 的一些缺陷。我们将在下一节介绍 DFS。

10.18 DCE 分布式文件系统(DCE DFS)

在 1989 年，Transarc 公司接管了 AFS 的开发和产品化工作。经过他们的努力，OSF 接纳了 AFS 技术，将其做为 OSF 分布式计算环境(DCE)的分布式文件系统的基础。这个新的实体通常称为 DCE DFS，或简单地称为 DFS。在本章以后的部分，我们都称之为 DFS。

DFS 是从 AFS 中演化而来，因此在很多方面同 AFS 相似。DFS 较 AFS 在以下方面有所改进：

1. DFS 中，一台机器既可以是客户机又可以是服务器。
2. DFS 提供了更强的类似 UNIX 的共享语义和一致性保证机制。
3. DFS 可以跟其他文件系统有更好的互操作性。

Transarc 为 DFS 服务器开发了 Episode 文件系统[Chut 92]作为其本地文件系统。我们将在 11.8 节中详细地讲述 Episode，它提供了高可用性(通过使用记日志)，同时也支持逻辑卷(AFS 中的卷在 Episode 中被称为文件集)及与 POSIX 兼容的访问控制表。在本章中，我们主要集中在 DFS 的分布式组成部分。

10.18.1　DFS 体系结构

DFS 结构在很多方面同 AFS 类似。它使用有状态的客户/服务器模型，活动服务器发出高速缓存无效的信息。它将整个文件(如果文件大于 64K，则为 64K 的块)缓存在客户的本地文件系统中。DFS 使用一个卷位置数据库来提供名字透明性。

DFS 也在很多方面比 AFS 有所改进。服务器和客户上都使用了 vnode/vfs 模型，这样 DFS 就可以同其他文件系统和访问协议实现互操作了。它也允许服务器的本地用户访问 DFS 文件系统。DFS 客户和服务器通过 DCE RPC 通信[OSF 92]，DCE RPC 提供了诸如同步和异步模式、Kerberos 认证等有用特性，也支持长距离(long haul)操作和面向连接的传输。

图 10-13 显示了整个 DFS 的体系结构。客户端和 AFS 的客户非常类似，其主要区别在于它们处理目录的形式。AFS 和 DFS 客户都缓存目录信息。然而 DSF 允许服务器输出很多不同种类的文件系统(不过最好是 Episode，因为它是专为 DFS 而做的)，因此客户可能不理解服务器的目录格式。因此 DFS 客户将单独一次查找的结果而不是整个目录的信息缓存起来。

DFS 服务器的设计同 AFS 大相径庭。在 AFS 中，访问协议和文件系统是一个集合的实体。而在 DFS 中，两者是分开的，通过 vnode/vfs 接口交互。这就允许 DFS 输出服务器的本地文件系统。它也允许服务器上的本地应用程序访问输出的文件系统。DFS 服务器使用一个称为 VFS+的扩展 vfs 接口，vfs+有支持卷和访问控制列表的附加功能。Episode 支持所有 VFS+的操作，因此提供了所有 DFS 的功能。其他本地文件系统可能不支持 vfs 的扩展，因此只能提供 DFS 功能的一部分。

协议导出程序处理来自 DFS 客户的请求。它为每个客户维护状态信息，并且每当客户缓存的数据变为无效时服务器便通知客户。VFS 接口中的连接层维护协议导出程序和其他文件访问方法(本地访问和服务器支持的其他分布式协议)的一致性，我们将在 10.18.2 小节中解释这些其他文件访问方法。

10.18.2　高速缓冲区一致性

DFS 对访问共享文件提供了严格的 UNIX 单系统语义。如果一个客户向一个文件写入数据，那么任何读取该文件的客户都应该看到新的数据。DFS 在 read 和 write 系统调用粒度上保证缓存一致性，而 AFS 是在 open 和 close 系统调用粒度上保证一致性的。

为了实现这些语义，DFS 服务器中有一个令牌管理器，主要用来跟踪所有引用文件的活动客户。对于每一次引用，服务器给客户一个或者多个令牌，用来保证文件数据和属性的有效性。服务器可以在任何时候通过回收令牌来取消有效性的保证。此时客户必须将相应的缓存数据视为无效，如果需要时再从服务器上取一次数据。

DFS 支持四种类型的令牌，每一种令牌处理不同的一些文件操作：

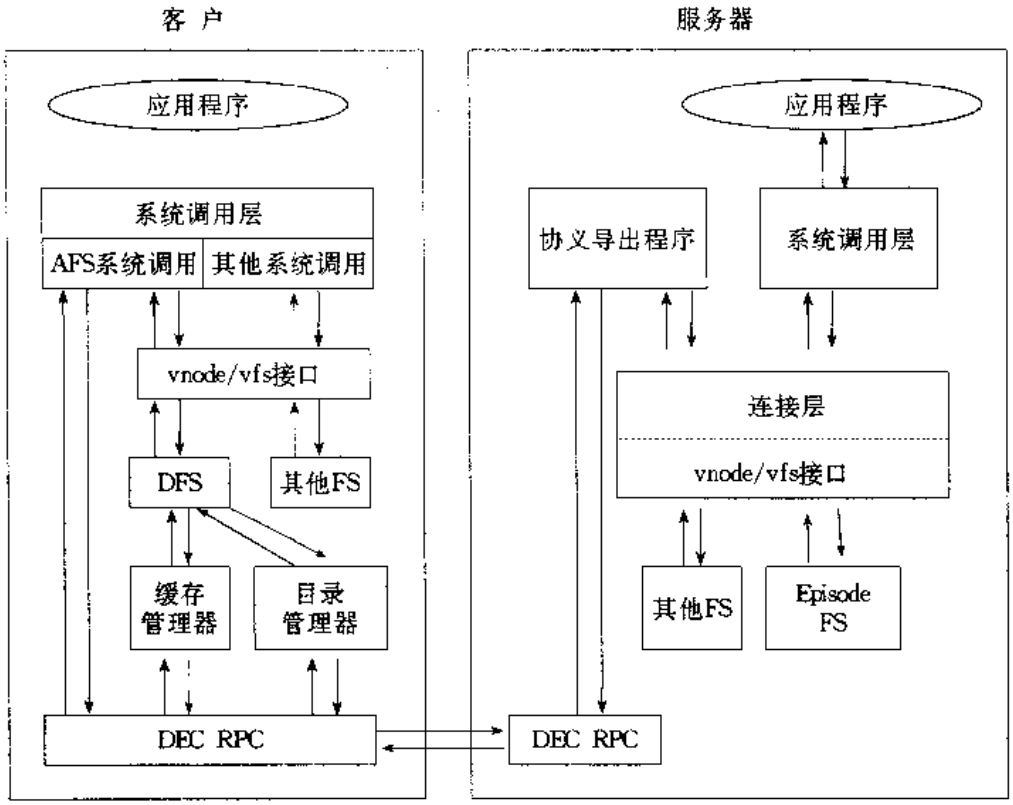

图 10-13 DFS 体系结构

数据令牌 有两种类型的数据令牌:读令牌和写令牌。每一个令牌都应用于文件中一定范围的字节。如果客户持有一个读令牌,那么它所缓存的文件的那部分数据副本就是有效的。如果它持有一个写令牌,它可以对缓存的数据进行修改,而不用将其刷新到服务器上。当服务器回收一个读令牌时,客户必须要将其缓存的数据丢弃掉。如果服务器会收一个写令牌,客户必须将所有的修改写回服务器,然后将数据丢弃掉。

状态令牌 这些令牌主要为了保证客户缓存的文件属性是有效的。此种令牌也分为两种类型——状态读令牌,状态写令牌。它们的语义跟数据读和数据读写令牌的语义类似。如果客户持有一个文件的写令牌,服务器便阻塞其他试图读文件属性的客户。

锁令牌 这种令牌可以使其持有者在文件的字节范围内加各种类型的锁。只要客户持有一个锁令牌,它就不需要联络服务器就可以将文件锁定,因为服务器不会让另外一个客户也同时为该文件加锁。

打开令牌 允许客户可以打开一个文件。根据文件打开的不同模式,打开令牌有几种不同类型——读,写,执行和独占性写。例如,如果有一个客户持有一个打开执行类的令牌,那么其他的客户就不能再修改文件。其他的分布式文件系统是很难支持这种特性的。但这种特性非常有必要,因为大多数的 UNIX 系统每次访问可执行文件的一页(分页,见 13.2 节)。如果当一个文件正在执行时被修订了,那么客户将会得一部分新程序,一部分旧程序,这将导致很多奇怪和不可预料的结果。

不同的客户可能需要同一个文件的令牌,DFS 为此定义了一组互相兼容规则。不同类型的令牌是相互兼容的,因为它们都跟一个文件的不同部分相关。对于同种类的令牌,规则

依令牌类型而有所不同。对于数据和锁令牌，如果读和写令牌的字节范围是重叠的，那么它们是不兼容的。对于打开令牌，独占性写令牌同其他任何类型的打开令牌都不兼容，执行打开令牌同一正常写令牌也是不兼容的。其他种类的组合相互之间都是互相兼容的。

令牌同 AFS 回调类似，因为它们都提供一致性保证，而服务器随时可以取消这种保证。不过同回调不同的是，令牌是一种分类的对象。AFS 为文件数据和文件属性名定义了不同种类的回调。而正如前面所述，DFS 提供几种不同种类的令牌。这就可以允许文件系统在更大程度上能够保持一致，同时也使对共享文件的访问可以达到 UNIX 方式的单用户语义。

10.18.3　令牌管理器

每个服务器都有一个驻留在 v 节点接口连接层中的令牌管理器。连接层中有一个为每个 v 节点操作的包装例程。该例程获得所有完成一个操作所需要的令牌，然后调用文件系统相关的 v 节点例程。

在很多情况下，一个 v 节点操作需要几个令牌。有些操作需要进行目录查找，找到在执行操作前必须获得令牌的那些 v 节点集。例如，对 rename 操作来说，如果源目录和目的目录不同，那么该操作需要对这两个目录的状态写令牌。同时也必须搜索目的目录。查看在目的目录中是否已经有一个以新名字命名的文件。如果有，rename 必须首先获得令牌以将该文件删掉。令牌管理器必须小心翼翼地操作以避免死锁，它所使用的机制类似于物理文件系统所使用的机制。

有时候客户请求会跟已经赋予给其他客户的令牌冲突。令牌管理器必须首先阻塞请求，然后通知持有当前令牌的客户，将该令牌回收。如果令牌正在用于读取，其所有者将令牌返回，将它缓存的数据标识为无效。如果令牌正在用于写，其所有者将所有已修改的数据刷新到服务器，然后将令牌返回。如果令牌是用于锁和打开文件，那么所有者直到解锁和关闭掉文件之后才能将令牌返回。当令牌管理器得到令牌后，它结束请求，将令牌返回给调用者。

令牌管理器必须在 v 节点层中，这是因为 DFS 可能同其他访问方法共存，比如通过系统调用使用的本地方法和其他一些像 NFS 之类的分布式协议。令牌管理器无需考虑访问方法便可获得令牌，这样就可以保证 DFS 在混合模式访问时的正确性。例如，假设两个客户并发地访问同一个文件，一个客户使用 NFS 而另外一个使用 DFS。如果 NFS 不需要获得或者不需要检测令牌，那么它将违反对 DFS 客户的保证。通过将令牌管理器放在连接层中，服务器可以同步所有对文件的操作。

10.18.4　其他 DFS 服务

DFS 远不止协议导出程序(protocol exporter)和客户高速缓存管理器。它提供很多同文件服务操作协作的辅助功能。这些功能包括：

文件集位置数据库(fldb)　这是一个全局的、包含有每个卷位置的数据库的备份。它保存了有关文件集和其复件的位置信息，它同 AFS 的卷位置数据库类似。

文件集服务器(ftserver)　实现了整个文件集的操作，比如文件集迁移。

认证服务器　提供了基于 Kerberos 的认证服务。

备份服务器(rpserver)　为了提高重要数据的可用性，DFS 支持文件集备份。这种方法可以防止网络和服务器损耗，同时也可通过将一些重复使用的文件集分散到不同的机

器而减轻服务器的负担。数据副本是只读的而原始数据是可读写的。DFS 支持两种形式的备份　　松弛的和被调度的。对于松弛备份，客户必须发出显式 fts 释放命令来更新原件的备份。而被调度备份以固定间隔自动更新备份。

10.18.5 分析

DFS 为访问分布式文件提供了一组广泛的工具。它的 Episode 文件系统使用日志来减小崩溃恢复时间，因此提高了系统的可用性。它使用两个分立的抽象　·集合体和文件集来组织文件系统。集合体是物理存储的单位，而文件集是文件系统的逻辑划分。通过这种方法，数据的逻辑和物理组织的工作量都减少了一倍。

Episode 对细粒度的文件保护，使用 POSIX 兼容的访问控制表，。尽管这可以提供比 UNIX 更加灵活和健壮的安全策略，系统管理员和用户对此却很不熟悉。类似的，Kerberos 为 DFS 提供了一个安全框架，但是却需修改一些程序，比如 login，ftp 和各种不同的批处理和邮件程序。

DFS 使用文件集复制来提高数据可用性并且通过将负载分布到不同的服务器上从而减少访问时间。文件集备份也允许对单个文件集进行备份，因为文件集的每个副本都是文件集的一个一致的快照。文件集位置数据库提供了位置独立性和透明性。

DFS 体系结构是基于客户端高速缓存的结构，该结构中由服务器向客户发送数据无效的提示。这种方法适合于大规模网络，因为它减少了在正常使用模式下的网络拥塞。通过在客户端和服务器同时实现 vnode/vfs 接口，DFS 可以同其他物理文件系统以及其他本地和远程文件访问协议实现互操作。

然而，DCE DFS 体系结构很复杂，不容易实现。它不仅需要 DCE RPC，而且需要很相关的服务，比如 X.500 全局目录服务[OSF 93]。特别地，在小的机器和简单的操作系统上很难支持 DFS。这将成为它被异质环境所接纳的障碍。

高速缓存的一致性维护机制和死锁避免机制也是高度复杂的。算法也必须能从单个客户、服务器或者网络段的失效中正确恢复过来。这对于那些提供细粒度的并发访问语义的分布式文件系统是一个很大的问题。

10.19 小　　结

本章主要描述了几种重要的分布式文件系统的结构和实现，它们是 NFS，RFS，AFS 和 DFS。NFS 是最容易实现和移植的结构。它已经被移植到很多平台和操作系统上去了，使得它成为异质环境中的最佳选择协议。但是，其扩展性不好，并且达不到 UNIX 的语义，写性能也十分不好。RFS 提供了 UNIX 语义，并且可以共享设备，但是只能运行于系统 UNIX 和从其发展出来的变体中。AFS 和 DFS 都是具有高扩展性的结构。DFS 提供了 UNIX 语义，并且可以同其他访问协议实现互操作。然而，它很复杂且不灵巧，因此难以实现。DFS 是一种新出现的技术，只有时间才能验证其成功与否。

现在很少有公认的可以测试这些文件系统相对性能的标准。[Howa 88]比较了 NFS 和 DFS 在相同硬件环境下的性能。结果表明，对于服务器，NFS 在低负载情况下速度很快，但是当负载加重时性能急剧下降。这两个系统从那时起已经获得了很大的发展，但是影响可扩

展性的因素没有显著地改变。

10.20　练　　习

1. 为什么在分布式文件系统中网络透明性很重要？
2. 位置透明性和位置独立性有什么区别？
3. 无状态文件系统的优点是什么？缺点是什么？
4. 哪一个分布式文件系统为共享访问文件提供了 UNIX 语义？哪一种提供了会话语义？
5. 为什么安装协议同 NFS 协议是分离的？
6. 一个同步 RPC 请求是怎样工作的？试设计一个客户接口发送一个同步请求，接受应答。
7. 写一个 RPC 程序，可以让用户向服务器发送一个字符串，然后在服务器上打印出来。为这种服务找到一个可以应用的地方。
8. 假设一个 NFS 服务器崩溃重启。它怎么知道客户机已经安装了什么文件系统？它的确关心这些吗？
9. 考虑下面一个从 NFS 安装目录执行的 shell 命令：

   ```
   echo hello> krishna. txt
   ```

   执行该命令将导致哪些 NFS 请求？它们的顺序是什么？假设 krishna. txt 原来不存在。
10. 在练习 9 中，如果 hello. txt 已经存在了，则 NFS 请求的顺序是什么？
11. NFS 客户通过将服务器上的文件重命名而假装将打开文件删除，而实际的删除工作是在打开文件关闭之后才进行的。如果在文件删除之前文件被删除了将发生什么情况？设计一个解决方案。
12. write 系统调用是异步的，不用等到数据被写到稳定存储器便返回来。为什么 NFS 的写操作必须是同步的？服务器或者客户崩溃对那些尚未完成的写操作将造成什么样的影响？
13. 为什么 NFS 不使用于大范围区域的操作？
14. 为什么 RFS 只使用于同质的环境中？
15. RFS 在何种程度上达到了网络透明性，位置独立性和位置透明性的要求？
16. DFS 比 AFS 在哪些方面有所提高？
17. DFS 协议导出程序的作用是什么？
18. DFS 令牌和 AFS 的回调有什么区别？
19. 在本章中讨论的分布式文件系统中哪一种提供了用户和文件的可迁移性？
20. 比较在 NFS，RFS，AFS 和 DFS 环境中用户看到的名字空间。
21. 比较在 NFS，RFS，AFS 和 DFS 中客户端高速缓存的一致性语义。

10.21 参考文献

[Bach 87] Bach, M. J., Luppi, M. W, Melamed, A. S., and Yueh, K., "A Remote-File Cache for RFS," Proceedings of the Summer 1987 USENIX Technical Conference, Jun. 1987, pp. 273-279.

[Bhid 91] Bhide, A., Elnozahy, E., and Morgan, S., "A Highly Available Network File Server," Proceedings of the Winter 1991 USENIX Technical Conference, Jan. 1991, pp. 199-205.

[Bhid 92] Bhide, A., and Shepler, S., "A Highly Available Lock Manager for HA-NFS," Proceedings of the Summer 1992 USENIX Technical Conference, Jun. 1992, pp. 177-184.

[Char 87] Chartok, H., "RFS in SunOS," Proceeding of the Summer 1987 USENIX Technical Conference, Jun. 1987, pp. 281-290.

[Cher 88] Cheriton, D. R., "The V Distributed System," Communications of the ACM, Vol. 31, No. 3, Mar. 1988, pp. 314-333.

[Chut 92] Chutani, S., Anderson, O. T., Kazar, M. L., Leverett, B. W., Mason, W. A., and idebotham, R. N., "The Episode File System," Proceedings of the Winter 1992 USENIX Technilcal Conference, Jan, 1992, pp. 43-59.

[Gerb 92] Gerber, B., "AFS: A Distributed File System that Supports Worldwide Networks," Network Comptuing, May 1992, pp. 142-148.

[Hitz 90] Hitz, D., Harris, G., Lau, J. K., and Schwartz, A. M., "Using UNIX as One Component of a Lightweight Distributed Kernel for Multiprocessor File Servers," Proceedings of the Winter 1990 USENIX Technical Conference, Jan., 1990, pp. 285-295.

[Hitz 94] Hitz, D., Lau, J., and Malcolm, M., "File System Design for an NFS File Server Applicance," Proceedings of the Winter 1994 USENIX Technical Conference, Jan. 1994, pp. 235-245.

[Hone 92] Honeyman, P., Huston, L. B., and Stolarchuk, M. T., "Hijacking AFS," Proceedcings of the Winter 1992 USENIX Technical Conference, Jan. 1992, pp. 175-181.

[Howa 88] Howard, J. H., Kazar, M. L., Menees, S. G., Nichols, D. A., Satyanarayanan, M., and Sidebotham, R. N., "Scale and Performance on a Distributed File System," ACM Transactions on Computer Systems, Vol. 6, No. 1, Feb. 1988, pp. 55-81.

[Jusz 89] Juszczak, C., "Improving the Performance and Correctness of an NFS Server," Proceedings of the Winter 1989 USENIX Technical Conference, Jan. 1989, pp. 53-63.

[Jusz 94] Juszczak, C., "Improving the Write Performance of an NFS Server," Proceedings of the Winter 1994 USENIX Technical Conference, Jan. 1994, pp. 247-259.

[Kaza 88] Kazar, M. L. "Synchronization and Caching Issues in the Andrew File System," Proceedings of the Winter 1988 USENIX Technical Conference, Feb. 1988, pp. 27-36.

[Kaza 90] Kazar, M. L., Leverett, B. W., Anderson, O. T., Apostolides, V., Bottos, B. A., Chutani, S., Everhart, C. F., Mason, W. A., Tu, S.-T., and Zayas, E. R., "Decorum File System Architectural Overview," Prceedings of the Summer 1990 USENIX Technical Conference, Jun. 1990.

[Levy 90] Levy, E., and Silberschatz, A., "Distributed File System: Concepts and Examples," ACM Computing Surveys, Vol. 22, No. 4, Dec. 1990, pp. 321-374.

[Mack 91] Macklem, R., "Lessons Learned Tunings the 4. 3BSD Reno Implementation of the NFS Protocol," Proceedings of the Winter 1991 USENIX Technical Conference, Jan. 1991, pp. 53-64.

[Mora 90] Moran, J., Sandberg, R., Coleman, D., Kepecs, J. and Lyon,, B., "Breaking Through the NFS Performance Barier," Proceedings of the Spring 1990 European UNIX Users Group Conference, Apr. 1990, pp. 199-206.

[Morr 86] Morris, J. H., Satyanarayanan, M., Conner, M. H., Howard, J. H., Rosenthal, D. S. H., and Smith, F., D., "Andrew: A Distributed Personal Computing Environment," Communications of the ACM, Vol. 29, No. 3, Mar. 1986, pp. 184-201.

[Nowi 90] Nowitz, D. A., "UUCP Administration," UNIX Research System Papers, Tenth Edition, Vol. II, Saunders College Publishing, 1990, pp. 563-580.

[OSF 92] Open Software Foundation, OSF DCE Application Environment Specification, Prentice-Hall, Englewood Cliffs, NJ, 1992.

[OSF 93] Open Software Foundation, OSF DCE Administration Guide-Extend Sevices pplication Environment Specification, Prentice-Hall, Englewood Cliffs, NJ, 1993.

[Pawal 94] Pawlowski, B., Juszczak, C., Staubach, P., Smith, C., Lebel, D., and Hitz, D., "NFS Version 3 Design and Implementation," Proceedings of the Summer 1994 USENIX Technical Conference, Jun. 1994, pp. 137-151.

[Post 85] Postel, J., and Reynolds, J., "The File Transfer Protocol," RFC 959, Oct. 1985.

[Plum 82] Plummer, D. C., "An Ethernet Address Resolution Protocol," RFC 826, Nov. 1982.

[Rifk 86] Rifkin, A. P., Forbes, M. P., Hamilton, R. L., Sabrio, M., Shah, S., and Yueh, K., "RFS Architectural Overview," Proceedings of the Summer 1986 USENIX Technical Conference, Jun. 1986, pp. 248-259.

[Sand 85a] Sandberg, R., Goldberg, D., Kleiman, S. R,., Walsh, D., and Lyon, B., "Design and Implementation of the Sun Network Filesystem," Proceedings of the Summer 1985 USENIX Technical Conference, Jun. 1985, pp. 119-130.

[Sand 85b] Sandberg, R., "Sun Network Filesystem Protocol Specification," Sun Microsystems, Inc., Technical Report, 1985.

[Saty 85] Satyanarayanan, M., Howard, J. H., Nichols, D. A., Sidebotham, R. N., Spector, A. Z., and West, M. J., "The ITC Distributed File System: Principles and Design," Tenth ACM Symposium on Operating Systems Principles, Dec, 1985, pp. 35-50.

[Side 86] Sidebotham, R. N., "VOLUMES——The Andrew File System Data Structuring Primitive," Proceedings of the Autumn 1986 European UNIX Users' Group Conference, Oct. 1986, pp. 473-480.

[Spec 89] Spector, A. Z., and Kazar, M. L., "Uniting File Systems," Unix Review, Vol. 7, No. 3, Mar. 1989, pp. 61-70.

[Stei 88] Steiner, J. G., Neuman, C., and Schiller, J. I., "Kerberos: An Authentication Service for Open Network Systems," Proceedings for the Winter 1988 USENIX Technical Cnference, Jan, 1988, pp. 191-202.

[Stol 93] Stolarchuk, M. T., "Faster AFS," proceedings of the Winter d1993 USENIX Technical Conference, Jan. 1993, pp. 67-75.

[Sun 87] Sun Microsystems, Inc., "XDR: External Data Representation Standard," RFC 1014, DDN Network Imfromation Center, SRI International, Jun. 1989.

[Sun 88] Sun Microsystems, Inc., "RPC: Remote Procedure Call, Protocol Specification, Version 2." RFC 1057, DDN Network Information Center, SRI Interational, Jun 1989.

[Sun 89] Sun Microsystems, Inc., "Network File System Protocol Specitfcation," RFC 1094DDN Network Information Center, SRI Intenational, Mar. 1989.

[Sun 95] Sun mICROSYSTEMS, iNC., "NFS Version 3 Protocol Specification," RFC 1813, DDN Network Information Center, SRI International, Jun. 1995.

[Tann 85]　Tannenbaum, A. S., and Van Renesse, R., "Distributed Operation Systems," ACM Computing Surveys, Vol, 17, No. 4, Dec. 1985, pp. 419-470.

[Tann 90]　Tannenbaum, A. S., Van Renesse, R., Van Staveren, H., Sharp. G. J., Mullender, S. J., Jansen, J., and Van Rossum, G., "Experiences with the Amoeba Distributed Operating System," Communications of the ACM, Vol. 33, No. 12, Dec. 1990, pp. 46-63.

[Witt 93]　Wittle, M., and Keith, B., "LADDIS: The Next Generation in NFS File Server Benchmarking," Proceedings of the summer 1993 USENIX Technical conference, Jun. 1993, pp. 111-128.