

## 第29章 网络文件系统

### 29.1 引言

本章中我们要讨论另一个常用的应用程序：NFS（网络文件系统），它为客户程序提供透明的文件访问。NFS的基础是Sun RPC：远程过程调用。我们首先必须描述一下RPC。

客户程序使用NFS不需要做什么特别的工作，当NFS内核检测到被访问的文件位于一个NFS服务器时，就会自动产生一个访问该文件的RPC调用。

我们对NFS如何访问文件的细节并不感兴趣，只对它如何使用Internet的协议，尤其是UDP协议，感兴趣。

### 29.2 Sun远程过程调用

大多数的网络程序设计都是编写一些调用系统提供的函数来完成特定的网络操作的应用程序。例如，一个函数完成TCP的主动打开，另一个完成TCP的被动打开，一个函数在一个TCP连接上发送数据，另一个设置特定的协议选项（如激活TCP的keepalive定时器）。在1.15节我们提到过两个常用的用于网络编程的函数集（API）：插口(socket)和TLI。正像客户端和服务端运行的操作系统可能会不相同一样，双方使用的API也可能会不相同。由通信协议和应用协议决定一对客户和服务端是否可以彼此通信。如果两台主机连接在一个网络上，并且都有一个TCP/IP的实现，那么一台主机上的一个使用C语言编写的、使用插口和TCP的Unix客户程序可以和另一台主机上的一个使用COBOL语言编写的、使用其他API和TCP的大型机服务器进行通信。

一般来说，客户发送命令给服务器，服务器向客户发送应答。目前为止，我们讨论过所有应用程序——Ping，Traceroute，选路守护程序、以及DNS、TFTP、BOOTP、SNMP、Telnet、FTP和SMTP的客户和服务端——都是采用这种方式实现的。

远程过程调用RPC (Remote Procedure Call)是一种不同的网络程序设计方法。客户程序编写时只是调用了服务器程序提供的函数。这只是程序员所感觉到的，实际上发生了下面一些动作。

- 1) 当客户程序调用远程的过程时，它实际上只是调用了位于本机上的、由RPC程序包生成的函数。这个函数被称为客户残桩(stub)。客户残桩将过程的参数封装成一个网络报文，并且将这个报文发送给服务器程序。

- 2) 服务器主机上的一个服务器残桩负责接收这个网络报文。它从网络报文中提取参数，然后调用应用程序员编写的服务器过程。

- 3) 当服务器函数返回时，它返回到服务器残桩。服务器残桩提取返回值，把返回值封装成一个网络报文，然后将报文发送给客户残桩。

- 4) 客户残桩从接收到的网络报文中取出返回值，将其返回给客户程序。

网络程序设计是通过残桩和使用诸如插口或TLI的某个API的RPC库例程来实现的，但是

用户程序——客户程序和被客户程序调用的服务器过程——不会和这个API打交道。客户应用程序只是调用服务器的过程，所有网络程序设计的细节都被 RPC程序包、客户残桩和服务器残桩所隐藏。

一个RPC程序包提供了很多好处。

1) 程序设计更加容易，因为很少或几乎没有涉及网络编程。应用程序设计员只需要编写一个客户程序和客户程序调用的服务器过程。

2) 如果使用了一个不可靠的协议，如 UDP，像超时和重传等细节就由 RPC程序包来处理。这就简化了用户应用程序。

3) RPC库为参数和返回值的传输提供任何需要的数据转换。例如，如果参数是由整数和浮点数组成的，RPC程序包处理整数和浮点数在客户机和服务器主机上存储的不同形式。这个功能简化了在异构环境中的客户和服务器的编码问题。

RPC程序设计的细节可以参看参考文献 [Stevens 1990]的第18章。两个常用的RPC程序包是Sun RPC和开放软件基金 (OSF) 分布式计算环境 (DCE) 的RPC程序包。我们对于RPC的兴趣在于想了解 Sun RPC中过程调用和过程返回报文的形式，因为本章中讨论的网络文件系统使用了它们。Sun RPC的第2版定义在RFC 1057 [Sun Microsystems 1988a]中。

## Sun RPC

Sun RPC有两个版本。一个版本建立在插口API基础上，和TCP和UDP打交道。另一个称为 TI-RPC的（独立于运输层），建立在 TLI API基础上，可以和内核提供的任何运输层协议打交道。尽管本章中我们只讨论 TCP和UDP，从讨论的观点来看，两者是一样的。

图29-1显示的是使用 UDP时，一个RPC过程调用报文的格式。IP首部和UDP首部是标准的首部，我们已经在图 3-1和图 11-2中显示过。UDP首部以下是RPC程序包定义的部分。

事务标识符 (XID) 由客户程序设置，由服务器程序返回。当客户收到一个应答，它将服务器返回的 XID与它发送的请求的 XID相比较。如果不匹配，客户就放弃这个报文，等待从服务器返回的下一个报文。每次客户发出一个新的 RPC，它就会改变报文的XID。但是如果客户重传一个以前发送过的 RPC（因为它没有收到服务器的一个应答），重传报文的XID不会修改。

调用(call)变量在过程调用报文中设置为 0，在应答报文中设置为 1。当前的RPC版本是2。接下来三个变量：程序号、版本号和过程号，标识了服务器上被调用的特定过程。

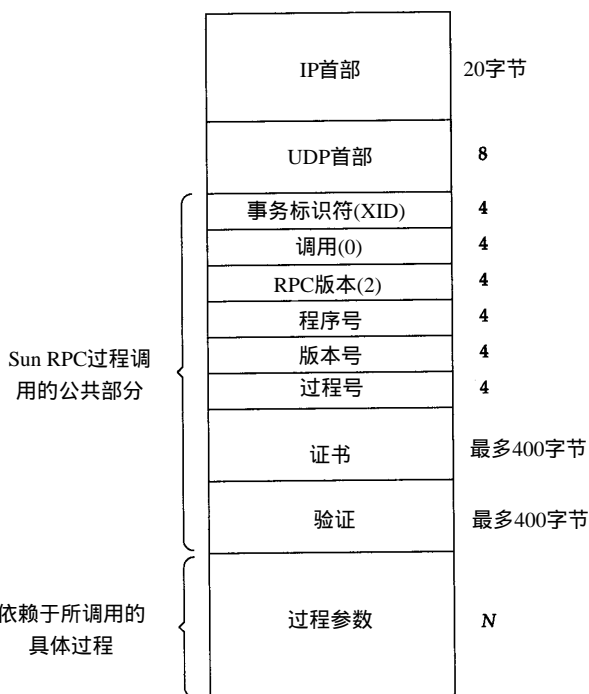


图29-1 RPC过程调用报文作为一个UDP数据报的格式

证书(credential)字段标识了客户。有些情况下,证书字段设置为空值;另外一些情况下,证书字段设置为数字形式的客户的用户号和组号。服务器可以查看证书字段以决定是否执行请求的过程。验证(verifier)字段用于使用了DES加密的安全RPC。尽管证书字段和验证字段是可变长度的字段,它们的长度也作为字段的一部分被编码。

接下来是过程参数(procedure parameter)字段。参数的格式依赖于远程过程的定义。接收者(服务器残桩)如何知道参数字段的大小呢?既然使用的是UDP协议,UDP数据报的大小减去验证字段以上所有字段的长度就是参数的大小。如果使用的不是UDP而是TCP,因为TCP是一个字节流协议,没有记录边界,所以没有固定的长度。为了解决这个问题,在TCP首部和XID之间增加了一个4字节的长度字段,告诉接收者这个RPC调用由多少字节组成。这也使得一个RPC调用报文在必要时可以用多个TCP段来传输(DNS使用了类似的技术,参见习题14-4)。

图29-2显示了一个RPC应答报文的格式。当远程过程返回时,服务器残桩将这个报文发送给客户残桩。

应答报文中的XID字段是从调用报文的XID字段复制而来。应答字段设置为1,以区别于调用报文。如果调用报文被接受,状态字段设置为0(如果RPC的版本号不为2,或者服务器不能鉴别客户的身份,调用报文可能被拒绝)。安全的RPC使用验证字段来标识服务器。

如果远程过程调用成功,接受状态字段置为0。一个非零的值可能表示一个不合法的版本号或者一个不合法的过程号。如果使用的不是UDP而是TCP,如同RPC调用报文一样,在TCP首部和XID字段之间插入一个4字节的长度字段。

### 29.3 XDR: 外部数据表示

外部数据表示XDR(eXternal Data Representation)是一个标准,用来对RPC调用报文和应答报文中的值进行编码。这些值包括RPC首部字段(XID、程序号、接受状态等)、过程参数和过程结果。采用标准化的方法对这些值进行编码使得一个系统中的客户可以调用另一个不同架构的系统中的一个过程。XDR在RFC 1014中定义[Sun Microsystems 1987]。

XDR定义了很多数据类型以及它们如何在一个RPC报文中传输的具体形式(如比特顺序,字节顺序等)。发送者必须采用XDR格式构造一个RPC报文,然后接收者将XDR格式的报文转换为本机的表示形式。例如,在图29-1和图29-2中,我们显示的所有整数值(XID、调用字段、程序号等)都是4字节的整数。在XDR中,所有的整数的确占据4个字节。XDR支持的其他数据类型包括无符号整数、布尔类型、浮点数、定长数组、可变长数组和结构。

### 29.4 端口映射器

包含远程过程的RPC服务器程序使用的是临时端口,而不是知名端口。这就需要某种形

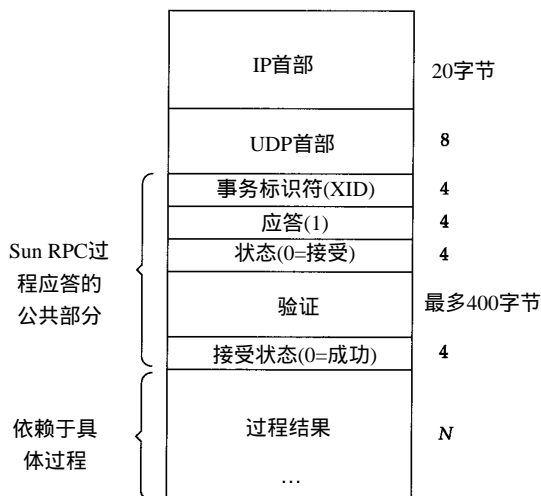


图29-2 RPC应答报文作为一个UDP数据报的格式

式的“注册”程序来跟踪哪一个RPC程序使用了哪一个临时端口。在Sun RPC中, 这个注册程序被称为端口映射器(port mapper)。

“端口”这个词作为Internet协议族的一个特征, 来自于TCP和UDP端口号。既然TI-RPC可以工作在任何运输层协议之上, 而不仅仅是TCP和UDP, 所以使用TI-RPC的系统中(如SVR4和Solaris 2.2), 端口映射器的名字变成了rpcbind。下面我们继续使用更为常见的端口映射器的名字。

很自然地, 端口映射器本身必须有一个知名端口: UDP端口111和TCP端口111。端口映射器也就是一个RPC服务器程序。它有一个程序号(100000)、一个版本号(2)、一个TCP端口111和一个UDP端口111。服务器程序使用RPC调用向端口映射器注册自身, 客户程序使用RPC调用向端口映射器查询。端口映射器提供四个服务过程:

1) PMAPPROC\_SET。一个RPC服务器启动时调用这个过程, 注册一个程序号、版本号和带有一个端口号的协议。

2) PMAPPROC\_UNSET。RPC服务器调用此过程来删除一个已经注册的映射。

3) PMAPPROC\_GETPORT。一个RPC客户启动时调用此过程。根据一个给定的程序号、版本号和协议来获得注册的端口号。

4) PMAPPROC\_DUMP。返回端口映射器数据库中所有的记录(每个记录包括程序号、版本号、协议和端口号):

在一个RPC服务器程序启动, 接着被一个RPC客户程序调用的过程中, 进行了以下一些步骤:

1) 一般情况下, 当系统引导时, 端口映射器必须首先启动。它创建一个TCP端点, 并且被打打开TCP端口111。它也创建一个UDP端点, 并且在UDP端口111等待着UDP数据报的到来。

2) 当RPC服务器程序启动时, 它为它所支持的程序的每一个版本创建一个TCP端点和一个UDP端点(一个给定的RPC程序可以支持多个版本。客户调用一个服务器过程时, 说明它想要哪一个版本)。两个端点各自绑定一个临时端口(TCP端口号和UDP端口号是否一致无关紧要)。服务器通过RPC调用端口映射器的PMAPPROC\_SET过程, 注册每一个程序、版本、协议和端口号。

3) 当RPC客户程序启动时, 它调用端口映射器的PMAPPROC\_GETPORT过程来获得一个指定程序、版本和协议的临时端口号。

4) 客户发送一个RPC调用报文给第3步返回的端口号。如果使用的是UDP, 客户只是发送一个包含RPC调用报文(见图29-1)的UDP数据报到服务器相应的UDP端口。服务器发送一个包含RPC应答报文(见图29-2)的UDP数据报到客户作为响应。

如果使用的是TCP, 客户对服务器的TCP端口号做一个主动打开, 然后在建立的TCP连接上发送一个RPC调用报文。服务器作为响应, 在连接上发送一个RPC应答报文。

程序rpcinfo(8)打印了端口映射器中当前的映射记录(它调用了端口映射器的PMAPPROC\_DUMP过程)。这里给出的是典型的输出:

```
sun % /usr/etc/rpcinfo -p
  program vers proto  port
100005    1   tcp    702  mountd      NFS的安装守护程序
100005    1   udp    699  mountd
100005    2   tcp    702  mountd
100005    2   udp    699  mountd
```

100003	2	udp	2049	nfs	NFS本身
100021	1	tcp	709	nlockmgr	NFS的加锁管理程序
100021	1	udp	1036	nlockmgr	
100021	2	tcp	721	nlockmgr	
100021	2	udp	1039	nlockmgr	
100021	3	tcp	713	nlockmgr	
100021	3	udp	1037	nlockmgr	

可以看出一些程序确实支持多个版本。在端口映射器中，每一个程序号、版本号和协议的组合都有自己的端口号映射。

安装守护程序（mount daemon）的两个版本可以通过同样的TCP端口号（702）和同样的UDP端口号（699）来访问，而加锁管理程序（lock manager）的每个版本都有各自不同的端口号。

## 29.5 NFS协议

使用NFS，客户可以透明地访问服务器上的文件和文件系统。这不同于提供文件传输的FTP（第27章）。FTP会产生文件一个完整的副本。NFS只访问一个进程引用文件的那一部分，并且NFS的一个目的就是使得这种访问透明。这就意味着任何能够访问一个本地文件的客户程序不需要做任何修改，就应该能够访问一个NFS文件。

NFS是一个使用Sun RPC构造的客户服务器应用程序。NFS客户通过向一个NFS服务器发送RPC请求来访问其上的文件。尽管这一工作可以使用一般的用户进程来实现——即NFS客户可以是一个用户进程，对服务器进行显式调用。而服务器也可以是一个用户进程——因为两个理由，NFS一般不这样实现。首先，访问一个NFS文件必须对客户透明。因此，NFS的客户调用是由客户操作系统代表用户进程来完成的。第二，出于效率的考虑，NFS服务器在服务器操作系统中实现。如果NFS服务器是一个用户进程，每个客户请求和服务器应答（包括读和写的数据）将不得不在内核和用户进程之间进行切换，这个代价太大。

本节中，我们考察在RFC1094中说明的第2版的NFS [Sun Microsystems 1988b]。[X/Open 1991] 中给出了Sun RPC、XDR和NFS的一个更好的描述。[Stern 1991] 给出了使用和管理NFS的细节。第3版的NFS协议在1993年发布，我们在29.7节中对它做一个简单的描述。

图29-3显示了一个NFS客户和一个NFS服务器的典型配置，图中有很多地方需要注意。

1) 访问的是一个本地文件还是一个NFS文件对于客户来说是透明的。当文件被打开时，由内核决定这一点。文件被打开之后，内核将本地文件的所有引用传递给名为“本地文件访问”的框中，而将一个NFS文件的所有引用传递给名为“NFS客户”的框中。

2) NFS客户通过它的TCP/IP模块向NFS服务器发送RPC请求。NFS主要使用UDP，最新的实现也可以使用TCP。

3) NFS服务器在端口2049接收作为UDP数据报的客户请求。尽管NFS可以被实现成使用端口映射器，允许服务器使用一个临时端口，但是大多数的实现都是直接指定UDP端口2049。

4) 当NFS服务器收到一个客户请求时，它将这个请求传递给本地文件访问例程，后者访问服务器主机上的一个本地的磁盘文件。

5) NFS服务器需要花一定的时间来处理一个客户的请求。访问本地文件系统一般也需要一部分时间。在这段时间间隔内，服务器不应该阻止其他的客户请求得到服务。为了实现这一功能，大多数的NFS服务器都是多线程的——即服务器的内核中实际上有多个NFS服务器在



运行。具体怎么实现依赖于不同的操作系统。既然大多数的 Unix内核不是多线程的，一个共同的技术就是启动一个用户进程（常被称为 `nfsd`）的多个实例。这个实例执行一个系统调用，使自己作为一个内核进程保留在操作系统的内核中。

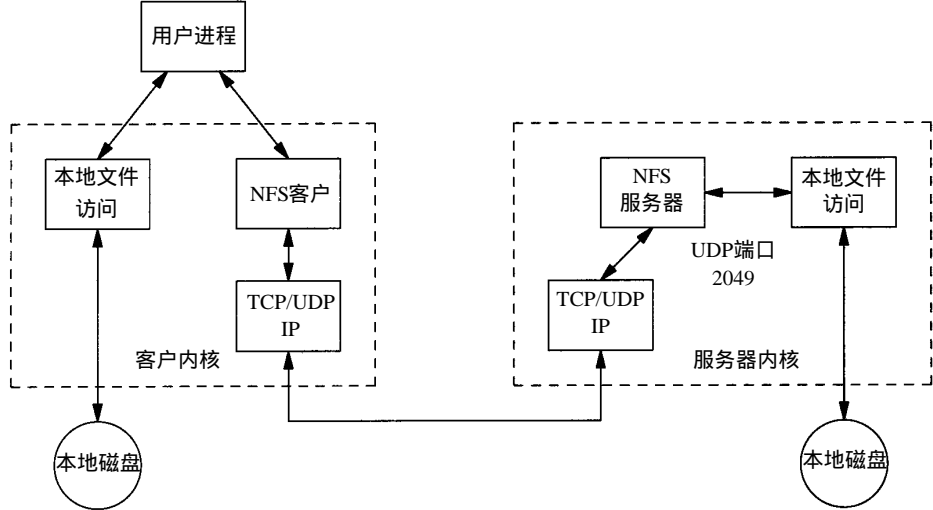


图29-3 NFS客户和NFS服务器的典型配置

6) 同样，在客户主机上，NFS客户需要花一定的时间来处理一个用户进程的请求。NFS客户向服务器主机发出一个RPC调用，然后等待服务器的应答。为了给使用NFS的客户主机上的用户进程提供更多的并发性，在客户内核中一般运行着多个NFS客户。同样，具体实现也依赖于操作系统。Unix系统经常使用类似于NFS服务器的技术：一个叫作**biod**的用户进程执行一个系统调用，作为一个内核进程保留在操作系统的内核中。

大多数的Unix主机可以作为一个NFS客户，一个NFS服务器，或者两者都是。大多数PC机的实现（MS-DOS）只提供了NFS客户实现。大多数的IBM大型机只提供了NFS服务器功能。

NFS实际上不仅仅由NFS协议组成。图29-4显示了NFS使用的不同RPC程序。

应用程序	程序号	版本号	过程数
端口映射器	100000	2	4
NFS	100003	2	15
安装程序	100005	1	5
加锁管理程序	100021	1, 2, 3	19
状态监视器	100024	1	6

图29-4 NFS使用的不同RPC程序

在这个图中，程序的版本是在SunOS 4.1.3中使用的。更新的实现提供了其中一些程序更新的版本。例如，Solaris 2.2还支持端口映射器的第3版和第4版，以及安装守护程序的第2版。SVR4支持第3版的端口映射器。

在客户能够访问服务器上的文件系统之前，NFS客户主机必须调用安装守护程序。我们在下面讨论安装守护程序。

加锁管理程序和状态监视器允许客户锁定一个NFS服务器上文件的部分区域。这两个程

序独立于NFS协议，因为加锁需要知道客户和服务器的状态，而NFS本身在服务器上是无状态的（下面我们对NFS的无状态会介绍得更多）。[X/Open 1991]的第9、10和11章说明了使用加锁管理程序和状态监视器进行NFS文件锁定的过程。

### 29.5.1 文件句柄

NFS中一个基本概念是文件句柄(file handle)。它是一个不透明(opaque)的对象，用来引用服务器上的一个文件或目录。不透明指的是服务器创建文件句柄，把它传递给客户，然后客户访问文件时，使用对应的文件句柄。客户不会查看文件句柄的内容——它的内容只对服务器有意义。

每次一个客户进程打开一个实际上位于一个NFS服务器上的文件时，NFS客户就会从NFS服务器那里获得该文件的一个文件句柄。每次NFS客户为用户进程读或写文件时，文件句柄就会传给服务器以指定被访问的文件。

一般情况下，用户进程不会和文件句柄打交道——只有NFS客户和NFS服务器将文件句柄传来传去。在第2版的NFS中，一个文件句柄占据32个字节，第3版中增加为64个字节。

Unix服务器一般在文件句柄中存储下面的信息：文件系统标识符（文件系统最大和最小的设备号），i-node号（在一个文件系统中唯一的数值）和一个i-node的生成码（每当一个i-node被一个不同的文件重用时就改变的数值）。

### 29.5.2 安装协议

客户必须在访问服务器上一个文件系统上的文件之前，使用安装协议安装那个文件系统。一般情况下，这是在客户主机引导时完成的。最后的结果就是客户获得服务器文件系统的一个文件句柄。

图29-5显示了一个Unix客户发出mount(8)命令所发生的情况，它说明一个NFS的安装过程。

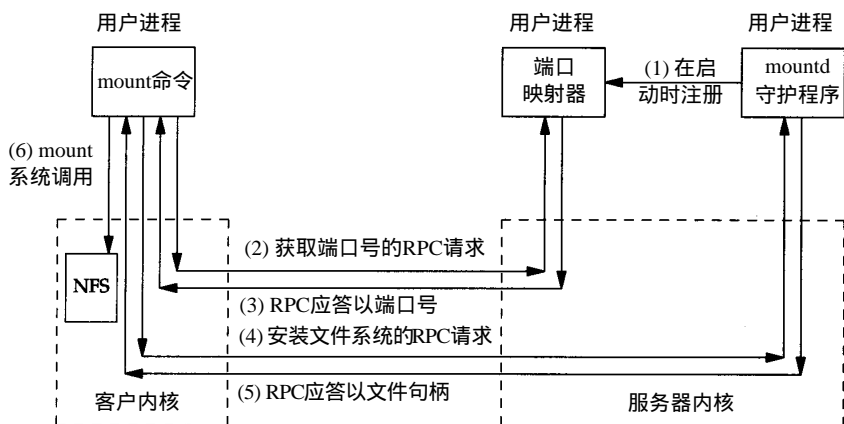


图29-5 使用Unix mount命令的安装协议

依次发生了下面的动作。

- 1) 服务器上的端口映射器一般在服务器主机引导时被启动。
- 2) 安装守护程序（mountd）在端口映射器之后被启动。它创建了一个TCP端点和一个

UDP端点, 并分别赋予一个临时的端口号。然后它在端口映射器中注册这些端口号。

3) 在客户机上执行mount命令, 它向服务器上的端口映射器发出一个RPC调用来获得服务器上安装守护程序的端口号。客户和端口映射器交互既可以使用TCP也可以使用UDP, 但一般使用UDP。

4) 端口映射器应答以安装守护程序的端口号。

5) mount命令向安装守护程序发出一个RPC调用来安装服务器上的一个文件系统。同样, 既可以使用TCP也可以使用UDP, 但一般使用UDP。服务器现在可以验证客户, 使用客户的IP地址和端口号来判别是否允许客户安装指定的文件系统。

6) 安装守护程序应答以指定文件系统的文件句柄。

7) 客户机上的mount命令发出mount系统调用将第5步返回的文件句柄与客户机上的一个本地安装点联系起来。文件句柄被存储在NFS客户代码中, 从现在开始, 用户进程对于那个服务器文件系统的任何引用都将从使用这个文件句柄开始。

上述实现技术将所有的安装处理, 除了客户机上的mount系统调用, 都放在用户进程中, 而不是放在内核中。我们显示的三个程序——mount命令、端口映射器和安装守护程序——都是用户进程。

作为一个例子, 在我们的主机sun (一个NFS客户机) 上执行:

```
sun # mount -t nfs bsdi:/usr /nfs/bsdi/usr
```

这个命令将主机bsdi (一个NFS服务器) 上的/usr目录安装成为本地文件系统/nfs/bsdi/usr。图29-6显示了结果。

当我们引用客户机sun上的/nfs/bsdi/usr/rstevens/hel文件时, 实际上引用的是服务器bsdi上的文件/usr/rstevens/hello.c。

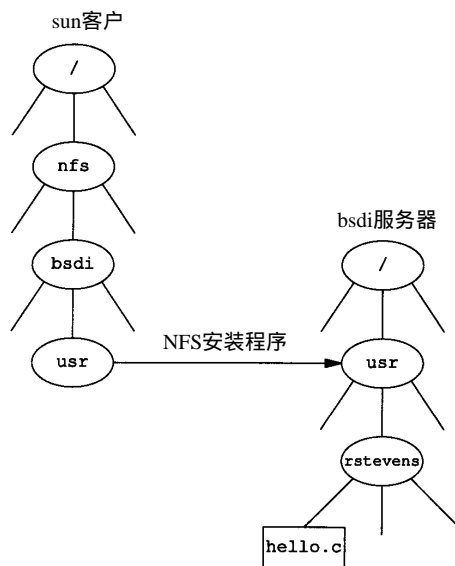


图29-6 将bsdi:/usr 目录安装成主机  
sun上的/nfs/bsdi/usr 目录

### 29.5.3 NFS过程

现在我们描述NFS服务器提供的15个过程 (使用的个数与NFS过程的实际个数不一样, 因为我们把它们按照功能分了组)。尽管NFS被设计成可以在不同的操作系统上工作, 而不仅仅是Unix系统, 但是一些提供Unix功能的过程可能不被其他操作系统支持 (例如硬链接、符号链接、组的属主和执行权等)。[Stevens 1992]的第4章包含了Unix文件系统其他的一些信息, 其中有些被NFS采用。

1) GETATTR。返回一个文件的属性: 文件类型 (一般文件, 目录等)、访问权限、文件大小、文件的属主者及上次访问时间等信息。

2) SETATTR。设置一个文件的属性。只允许设置文件属性的一个子集: 访问权限、文件



的属主、组的属主、文件大小、上次访问时间和上次修改时间。

3) STATFS。返回一个文件系统的状态：可用空间的大小、最佳传送大小等。例如 Unix 的 df 命令使用此过程。

4) LOOKUP。查找一个文件。每当一个用户进程打开一个 NFS 服务器上的一个文件时，NFS 客户调用此过程。

5) READ。从一个文件中读数据。客户说明文件的句柄、读操作的开始位置和读数据的最大字节数（最多 8192 个字节）。

6) WRITE。对一个文件进行写操作。客户说明文件的句柄、开始位置、写数据的字节数 and 要写的数据。

7) CREATE。创建一个文件。

8) REMOVE。删除一个文件。

9) RENAME。重命名一个文件。

10) LINK。为一个文件构造一个硬链接。硬链接是一个 Unix 的概念，指的是磁盘中的一个文件可以有任意多个目录项（即名字，也叫作硬链接）指向它。

11) SYMLINK。为一个文件创建一个符号链接。符号链接是一个包含另一个文件名字的文件。大多数引用符号链接的操作（例如，打开）实际上引用的是符号链接所指的文件。

12) READLINK。读一个符号链接。即返回符号链接所指的文件的名字。

13) MKDIR。创建一个目录。

14) RMDIR。删除一个目录。

15) REaddir。读一个目录。例如，Unix 的 ls 命令使用此过程。

这些过程实际上有一个前缀 NFSPROC\_，我们把它省略了。

#### 29.5.4 UDP 还是 TCP

NFS 最初是用 UDP 写的，所有的厂商都提供了这种实现。最新的一些实现也支持 TCP。TCP 支持主要用于广域网，它可以使文件操作更快。NFS 已经不再局限于局域网的使用。

当从 LAN 转换到 WAN 时，网络的动态特征变化得非常大。往返时间（round-trip time）变动范围大，拥塞经常发生。WAN 的这些特征使得我们考虑使用具有 TCP 属性的算法——慢启动，但是可以避免拥塞。既然 UDP 没有提供任何类似的东西，那么在 NFS 客户和服务器的上加进同样的算法或者使用 TCP。

#### 29.5.5 TCP 上的 NFS

伯克利实现的 Net/2 NFS 支持 UDP 或者 TCP。[Macklem 1991] 描述了这个实现。让我们看一下使用 TCP 有什么不同。

1) 当服务器主机进行引导时，它启动一个 NFS 服务器，后者被动打开 TCP 端口 2049，等待着客户的连接请求。这通常是另一个 NFS 服务器，正常的 NFS UDP 服务器在 UDP 端口 2049 等待着进入的 UDP 数据报。

2) 当客户使用 TCP 安装服务器上的文件系统时，它对服务器上的 TCP 端口 2049 做一个主动打开。这样就为这个文件系统在客户和服务器之间形成了一个 TCP 连接。如果同样的客户安装同样服务器上的另一个文件系统，就会创建另一个 TCP 连接。

3) 客户和服务端在它们连接的两端都要设置 TCP的keepalive选项, 这样双方都能检测到对方主机崩溃, 或者崩溃然后重新启动。

4) 客户方所有使用这个服务器文件系统的应用程序共享这个 TCP连接。例如, 在图 29-6 中, 如果在 bsd1的/usr目录下还有另一个目录 smith, 那么对两个目录 /nfs/bsd1/usr/rstevens和 /nfs/bsd1/usr/smith下所有文件的引用将共享同样的 TCP连接。

5) 如果客户检测到服务器已经崩溃, 或者崩溃然后重新启动 (通过收到一个 TCP差错 “连接超时” 或者 “对方复位连接”), 它尝试与服务器重新建立连接。客户做另一个主动打开, 为同一个文件系统请求重新建立 TCP连接。在以前连接上超时的所有客户请求在新的连接上都会重新发出。

6) 如果客户机崩溃, 那么当它崩溃时正在运行的应用程序也要崩溃。当客户机重新启动时, 它很可能使用 TCP重新安装服务器的文件系统, 这将导致和服务器的另一个连接。客户和服务端之间针对同一个文件系统的前一个连接现在打开了一半 (服务器方认为它还开着), 但是既然服务器设置了 keepalive选项, 当服务器发出下一个 keepalive探查报文时, 这个半开着的TCP连接就会被中止。

随着时间的流逝, 另外一些厂商也计划支持 TCP上的NFS。

## 29.6 NFS实例

我们使用tcpdump来看一下在典型的文件操作中, 客户调用了哪些 NFS过程。当tcpdump检测到一个包含 RPC调用 (在图 29-1中调用字段等于 0) 目的端口是 2049的UDP数据报时, 它把数据报按照一个 NFS请求进行解码。类似地, 如果一个 UDP数据报是一个RPC应答 (在图29-2中应答字段为 1), 源端口是 2049, tcpdump就把此数据报作为一个NFS应答来解码。

### 29.6.1 简单的例子: 读一个文件

第一个例子是使用cat(1)命令将位于一个NFS服务器上的一个文件复制到终端上:

```
sun % cat /nfs/bsd1/usr/rstevens/hello.c      把文件复制到终端
main()
{
    printf("hello, world\n");
}
```

如同图 29-6所示, 主机 sun (NFS客户机) 上的文件系统 /nfs/bsd1/usr 实际上是主机 bsd1 (NFS服务器) 上的 /usr 文件系统。当cat打开这个文件时, sun上的内核检测到这一点, 然后使用NFS去访问文件。图 29-7显示了tcpdump的输出。

当tcpdump解析一个NFS请求或应答报文时, 它打印客户的XID字段, 而不是端口号。第1行和第2行中的XID字段值是0x7aa6。

客户内核中的打开函数一次处理文件名 /nfs/bsd1/usr/rstevens/hello.c 中的一个成员。当处理到/nfs/bsd1/usr时, 它发现这是指向一个已安装的NFS文件系统的一个安装点。

在第1行中, 客户调用GETATTR过程取得客户已经安装的服务器目录的属性 (/usr)。这个RPC请求, 除IP首部和UDP首部之外, 包含 104个字节的数据。第2行中的应答返回了一个OK值, 除了IP首部和UDP首部之外, 包含了 96个字节的数据。在这个图中, 我们可以看出最小的NFS报文包含大约100个字节的数据。

```

1 0.0 sun.7aa6 > bsdi.nfs: 104 getattr
2 0.003587 (0.0036) bsdi.nfs > sun.7aa6: reply ok 96
3 0.005390 (0.0018) sun.7aa7 > bsdi.nfs: 116 lookup "rstevens"
4 0.009570 (0.0042) bsdi.nfs > sun.7aa7: reply ok 128
5 0.011413 (0.0018) sun.7aa8 > bsdi.nfs: 116 lookup "hello.c"
6 0.015512 (0.0041) bsdi.nfs > sun.7aa8: reply ok 128
7 0.018843 (0.0033) sun.7aa9 > bsdi.nfs: 104 getattr
8 0.022377 (0.0035) bsdi.nfs > sun.7aa9: reply ok 96
9 0.027621 (0.0052) sun.7aaa > bsdi.nfs: 116 read 1024 bytes @ 0
10 0.032170 (0.0045) bsdi.nfs > sun.7aaa: reply ok 140

```

图29-7 读一个文件的NFS操作

在第3行中，客户调用 LOOKUP 过程来查看 `rstevens` 文件。在第4行中收到一个 OK 应答。LOOKUP 过程说明了文件名 `rstevens` 和远程文件系统被安装时由内核保存的文件句柄。应答中包含了下一步要使用的一个新的文件句柄。

在第5行中，客户使用第4行中返回的文件句柄对 `hello.c` 调用 LOOKUP 过程。在第6行返回了另一个文件句柄。新的文件句柄就是客户在第7行和第9行中引用文件 `/nfs/bsdi/usr/rstevens/hello.c` 所使用的文件句柄。我们看到客户对于正在打开的路径名的每个成员都调用了一次 LOOKUP 过程。

在第7行中，客户又调用了一次 GETATTR 过程，接着在第9行中调用了 READ 过程。客户请求从偏移0开始的1024个字节，但是接收到的没有这么多（减去 RPC 字段和其他由 READ 过程返回的值的的大小，在第10行中返回了38个字节的数据。这是文件 `hello.c` 的实际大小）。

在这个例子中，应用进程对于内核所做的这些 RPC 请求和应答一点儿也不知道。应用进程只是调用了内核的 `open` 函数，后者引起了3个RPC请求和3个应答（1~6行），然后应用进程又调用了内核的 `read` 函数，它引起了两个请求和两个应答（7~10行）。该文件位于一个 NFS 文件服务器，这一点对客户应用进程来说是透明的。

### 29.6.2 简单的例子：创建一个目录

作为另一个简单的例子，我们将当前工作目录改变为一个 NFS 服务器上的一个目录，然后创建一个新的目录：

```

sun % cd /nfs/bsdi/usr/rstevens      改变当前工作目录
sun % mkdir Mail                     并且创建一个目录

```

图29-8显示了tcpdump的输出。

```

1 0.0 sun.7ad2 > bsdi.nfs: 104 getattr
2 0.004912 ( 0.0049) bsdi.nfs > sun.7ad2: reply ok 96
3 0.007266 ( 0.0024) sun.7ad3 > bsdi.nfs: 104 getattr
4 0.010846 ( 0.0036) bsdi.nfs > sun.7ad3: reply ok 96
5 35.769875 (35.7590) sun.7ad4 > bsdi.nfs: 104 getattr
6 35.773432 ( 0.0036) bsdi.nfs > sun.7ad4: reply ok 96
7 35.775236 ( 0.0018) sun.7ad5 > bsdi.nfs: 112 lookup "Mail"
8 35.780914 ( 0.0057) bsdi.nfs > sun.7ad5: reply ok 28
9 35.782339 ( 0.0014) sun.7ad6 > bsdi.nfs: 144 mkdir "Mail"
10 35.992354 ( 0.2100) bsdi.nfs > sun.7ad6: reply ok 128

```

图29-8 NFS的操作：cd到NFS目录，然后mkdir

改变目录引起客户调用了两次 GETATTR过程 (1~4行)。当我们创建新的目录时, 客户调用了 GETATTR过程 (5~6行), 接着调用 LOOKUP过程 (7~8行, 用来验证将创建的目录不存在), 跟着调用了 MKDIR过程来创建目录 (9~10行)。在第8行中, 应答 OK并不表示目录存在。它只是表示过程返回了。tcpdump并不理解 NFS过程的返回值。它一般打印 OK和应答报文中数据的字节数。

### 29.6.3 无状态

NFS的一个特征 (NFS的批评者称之为 NFS的一个瑕疵, 而不是一个特征) 是 NFS服务器是无状态的 (stateless)。服务器并不记录哪个客户正在访问哪个文件。请注意一下在前面给出的 NFS过程中, 没有一个 open操作和一个 close操作。LOOKUP过程的功能与 open操作有些类似, 但是服务器永远也不会知道客户对一个文件调用了 LOOKUP过程之后是否会引用该文件。

无状态设计的理由是为了在服务器崩溃并且重新启动时, 简化服务器的崩溃恢复操作。

### 29.6.4 例子: 服务器崩溃

在下面的例子中我们从一个崩溃然后重新启动的 NFS服务器上读一个文件。这个例子演示了无状态的服务器是如何使得客户不知道服务器的崩溃。除了在服务器崩溃然后重新启动时一个时间上的暂停外, 客户并不知道发生的问题, 客户应用进程没有受到影响。

在客户机 sun上, 我们对一个长文件 (NFS服务器主机 svr4上的文件 /usr/share/lib/termcap) 执行 cat命令。在传送过程中把以太网的网线拔掉, 关闭然后重新启动服务器主机, 再重新将网线连上。客户被配置成每个 NFS read过程读 1024个字节。图 29-9显示了 tcpdump的输出。

1~10行对应于客户打开文件, 操作类似于图 29-7所示。在第 11行我们看到对文件的第一个 READ操作, 在 12行返回了 1024个字节的数据。这个操作一直继续到 129行 (读 1024个字节的数据, 跟着一个 OK应答)。

在第 130行和第 131行我们看到两个请求超时, 并且分别在 132行和 133行重传。第一个问题是这里为什么会有两个读请求, 一个从偏移 65536开始读, 另一个从偏移 73728开始读? 答案是客户内核检测到客户应用进程正在进行顺序地读操作, 所以试图预先取得数据块 (大多数的 Unix内核都采用了这种预读技术)。客户内核也正在运行多个 NFS块 I/O守护程序, 后者试图代表客户产生多个 RPC请求。一个守护程序正在从偏移 65536处读 8192个字节 (以 1024字节为一组数据块), 而另一个正在从 73728处预读 8192个字节。

客户重传发生在 130~168行。在第 169行我们看到服务器已经重新启动, 在它第 168行的客户 NFS请求做出应答之前, 它发送了一个 ARP请求。对 168行的响应被发送给 171行。客户的 READ操作继续进行下去。

除了从 129行到 171行 5分钟的暂停, 客户应用进程并不知道服务器崩溃然后又重启了。这个服务器的崩溃对于客户是透明的。

为了研究这个例子中的超时和重传时间间隔, 首先要意识到这儿有两个客户守护程序, 分别有它们各自的超时。第 1个守护程序 (在偏移 65536处开始读) 的间隔, 四舍五入到两个十进制小数点, 为 0.68, 0.87, 1.74, 3.48, 6.96, 13.92, 20.0, 20.0, 20.0等等。第 2个守护程序 (在偏移 73728处开始读) 的间隔也是一样的 (精确到两个小数点)。可以看出这些 NFS客户使用了一个这样的超时定时器: 间隔为 0.875秒的倍数, 上限为 20秒。每次超时后, 重传间隔翻倍:

0.875, 1.75, 3.5, 7.0和14.0。

```

1   0.0          sun.7ade > svr4.nfs: 104 getattr
2   0.007653 ( 0.0077) svr4.nfs > sun.7ade: reply ok 96
3   0.009041 ( 0.0014) sun.7adf > svr4.nfs: 116 lookup "share"
4   0.017237 ( 0.0082) svr4.nfs > sun.7adf: reply ok 128
5   0.018518 ( 0.0013) sun.7ae0 > svr4.nfs: 112 lookup "lib"
6   0.026802 ( 0.0083) svr4.nfs > sun.7ae0: reply ok 128
7   0.028096 ( 0.0013) sun.7ae1 > svr4.nfs: 116 lookup "termcap"
8   0.036434 ( 0.0083) svr4.nfs > sun.7ae1: reply ok 128
9   0.038060 ( 0.0016) sun.7ae2 > svr4.nfs: 104 getattr
10  0.045821 ( 0.0078) svr4.nfs > sun.7ae2: reply ok 96
11  0.050984 ( 0.0052) sun.7ae3 > svr4.nfs: 116 read 1024 bytes @ 0
12  0.084995 ( 0.0340) svr4.nfs > sun.7ae3: reply ok 1124

      连续地读
128 3.430313 ( 0.0013) sun.7b22 > svr4.nfs: 116 read 1024 bytes @ 64512
129 3.441828 ( 0.0115) svr4.nfs > sun.7b22: reply ok 1124
130 4.125031 ( 0.6832) sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
131 4.868593 ( 0.7436) sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728
132 4.993021 ( 0.1244) sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
133 5.732217 ( 0.7392) sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728
134 6.732084 ( 0.9999) sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
135 7.472098 ( 0.7400) sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728
136 10.211964 ( 2.7399) sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
137 10.951960 ( 0.7400) sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728
138 17.171767 ( 6.2198) sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
139 17.911762 ( 0.7400) sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728
140 31.092136 (13.1804) sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
141 31.831432 ( 0.7393) sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728
142 51.090854 (19.2594) sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
143 51.830939 ( 0.7401) sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728
144 71.090305 (19.2594) sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
145 71.830155 ( 0.7398) sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728

      连续重传
167 291.824285 ( 0.7400) sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728
168 311.083676 (19.2594) sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536

      服务器重启动
169 311.149476 ( 0.0658) arp who-has sun tell svr4
170 311.150004 ( 0.0005) arp reply sun is-at 8:0:20:3:f6:42
171 311.154852 ( 0.0048) svr4.nfs > sun.7b23: reply ok 1124
172 311.156671 ( 0.0018) sun.7b25 > svr4.nfs: 116 read 1024 bytes @ 66560
173 311.168926 ( 0.0123) svr4.nfs > sun.7b25: reply ok 1124

      连续读

```

图29-9 当一个NFS服务器崩溃然后重启时，客户正在读一个文件的过程

客户要重传多久呢？客户有两个与此有关的选项。首先，如果服务器文件系统是“硬”安装的，客户就会永远重传下去。但是如果服务器文件系统是“软”安装的，客户重传了固定数目的次数之后就会放弃。在“硬”安装的情况下，客户还有一个选项决定是否允许用户中断无限的重传。如果客户主机安装服务器文件系统时说明了中断能力，并且如果我们不想在服务器崩溃之后等5分钟，等着服务器重启动，就可以键入一个中断键以终止客户应用



程序。

### 29.6.5 等幂过程

如果一个RPC过程被服务器执行多次仍然返回同样的结果,那么就把它叫作等幂过程(Idempotent Procedure)。例如,NFS的读过程是等幂的。正像我们在图29-9中看到的,客户只是重发一个特定的READ调用直到它得到一个响应。在我们的例子中,重传的原因是服务器崩溃了。如果服务器没有崩溃,而是RPC应答报文丢失了(既然UDP是不可靠的),客户只是重传请求,服务器再一次执行同样的READ过程。同一个文件的同一部分被重读一次,发送给客户。

这种方法行得通的原因在于每个READ请求指出了读操作开始的偏移位置。如果有一个NFS过程要求服务器读一个文件的下 $N$ 个字节,这种方法就不行了。除非服务器被做成是有状态的(与无状态相反),如果一个应答丢失了,客户重发读下 $N$ 个字节的READ请求,结果将是不一样的。这就是为什么NFS的READ和WRITE过程要求客户说明开始的偏移位置的原因。客户维护着状态(每个文件当前的偏移位置),而不是服务器。

不幸的是并不是所有的文件系统操作都是等幂的。例如,考虑下面的动作:客户NFS发出REMOVE请求来删除一个文件;服务器NFS删除了文件,并回答OK;服务器的回答丢失了;客户NFS超时,然后重传请求;服务器NFS找不到指定的文件,回答指出一个错误;客户应用程序接收到一个错误表示文件不存在。这个返回给客户应用程序的错误是不对的——该文件的确存在并且被删除了。

等幂的NFS过程是:GETATTR、STATES、LOOKUP、READ、WRITE、READLINK和REaddir。不是等幂的过程是:CREATE、REMOVE、RENAME、LINK、SYMLINK、MKDIR和RMDIR。SETATTR过程如果不用来截断文件,一般是等幂的。

既然使用UDP总会发生响应报文丢失的现象,NFS服务器需要一种方法来处理非等幂的操作。大多数的服务器实现了一个最近应答的高速缓存,用于存放非等幂操作最近的应答。每当服务器收到一个请求,它首先检查这个高速缓存,如果找到了一个匹配,就返回以前的应答而不再调用相应的NFS过程。[Juszczak 1989]提供了这种高速缓存的实现细节。

等幂服务器过程的概念可以应用于任何基于UDP的应用程序,而不仅仅是NFS。例如,DNS也提供了一个等幂服务。一个DNS的服务器可以任意多次地执行一个解析者的请求而没有任何不良的后果(如果不考虑网络资源浪费的话)。

## 29.7 第3版的NFS

1993年发布了第3版的NFS协议规范[Sun Microsystem 1994]。其实现有望在1994年成为可能。

我们总结一下第2版和第3版的主要区别。下面把两者分别称为V2和V3。

1) V2中的文件句柄是32字节的固定大小的数组。在V3中,它变成了一个最多为64个字节的可变长度的数组。在XDR中,一个可变长度的数组被编码为一个4字节的数组成员个数跟着实际的数组成员字节。这样在实现时减少了文件句柄的长度,例如Unix只需要12个字节,但又允许非Unix实现维护另外的信息。

2) V2将每个READ和WRITE RPC过程可以读写的数据限制为8192个字节。这个限制在V3

中取消了，这就意味着一个 UDP 上的实现只受到 IP 数据报大小的限制（65535 字节）。这样允许在更快的网络上读写更大的分组。

3) 文件大小以及 READ 和 WRITE 过程开始偏移的字节从 32 字节扩充到 64 字节，允许读写更大的文件。

4) 每个影响文件属性值的调用都返回文件的属性。这样减少了客户调用 GETATTR 过程的次数。

5) WRITE 过程可以是异步的，而在 V2 中要求同步的 WRITE 过程。这样可以提高 WRITE 过程的性能。

6) V3 中删去了一个过程（STATFS），增加了七个过程：ACCESS（检查文件访问权限）、MKNOD（创建一个 Unix 特殊文件）、READDIRPLUS（返回一个目录中的文件名字和它们的属性）、FSINFO（返回一个文件系统的静态信息）、FSSTAT（返回一个文件系统的动态信息）、PATHCONF（返回一个文件的 POSIX.1 信息）和 COMMIT（将以前的异步写操作提交到外存中）。

## 29.8 小结

RPC 是构造客户-服务器应用程序的一种方式，使得看起来客户只是调用了服务器的过程。所有的网络操作细节都被隐藏在 RPC 程序包为一个应用程序生成的客户和服务端残桩以及 RPC 库的例程中。我们显示了 RPC 调用和应答报文的格式，并且提到了使用 XDR 对传输的值进行编码，使得 RPC 客户和服务端可以运行在不同架构的机器上。

最广泛使用的 RPC 应用之一就是 Sun 的 NFS，一个在各种大小的主机上广泛实现的异构的文件访问协议。我们浏览了 NFS 和它使用 UDP 和 TCP 的方式。第 2 版的 NFS 协议定义了 15 个过程。

一个客户对一个 NFS 服务器的访问开始于安装协议，返回给客户一个文件句柄。客户接着可以使用那个文件句柄来访问服务器文件系统中的文件。在服务器上，一次检查文件名的一个成员，返回每个成员的一个新的文件句柄。最后的结果就是要引用的文件的一个文件句柄，它可以在随后的读写操作中被使用。

NFS 试图把它的所用过程都做成等幂的，使得如果响应报文丢失了，客户只需要重发一个请求。我们看到了服务器崩溃然后又重新启动时，一个客户读服务器上的一个文件的例子。

## 习题

- 29.1 在图 29-7 中，我们看到 tcpdump 将分组理解为 NFS 的请求和应答，打印了 XID。tcpdump 可以为任何的 RPC 请求或者应答这样做吗？
- 29.2 在一个 Unix 系统中，你认为为什么 RPC 服务器程序使用的是临时端口，而不是知名端口？
- 29.3 一个 RPC 客户调用了两个服务器过程。第 1 个服务器过程执行花了 5 秒钟的时间，第二个过程花了 1 秒钟。客户有一个 4 秒钟的超时。画出客户与服务器之间在时间轴上交互的信息（假定信息从客户传到服务器或者相反都不花时间）。
- 29.4 在图 29-9 的例子中，如果 NFS 服务器关机时，把它的以太网卡给换掉了，将会发生什么事情？

- 29.5 在图29-9中, 当服务器重新启动后, 它处理了从偏移 65536开始的请求 ( 168行和171行 ), 然后处理了从偏移 66560开始的下一个请求 ( 172行和173行 )。对于从偏移 73728开始的请求怎么处理的呢? ( 167行 )
- 29.6 当描述等幂NFS过程时, 我们给出了一个REMOVE应答在网络中丢失的例子。在这种情况下, 如果使用的是TCP而不是UDP会怎么样呢?
- 29.7 如果NFS服务器使用的是一个临时端口而不是 2049 ,那么当服务器崩溃然后又重新启动时, 一个NFS客户会发生什么情况呢?
- 29.8 每个主机最多只有 1023个保留端口, 所以保留端口是很缺乏的 ( 1.9节 )。如果一个NFS服务器要求它的客户拥有保留端口 ( 公共的端口 ), 一个NFS客户使用TCP安装了 $N$ 个不同的服务器上的 $N$ 个文件系统, 那么客户对每个连接都需要一个不同的保留端口号吗?