所需的所有信息。不需要花哨的崩溃恢复，服务器只是再次开始运行，最糟糕的情况下，客户端可能必须重试请求。

48.5 NFSv2 协议

下面来看看 NFSv2 的协议定义。问题很简单：

关键问题：如何定义无状态文件协议

如何定义网络协议来支持无状态操作？显然，像 open()这样的有状态调用不应该讨论。但是，客户端应用程序会调用 open()、read()、write()、close()和其他标准 API 调用，来访问文件和目录。因此，改进该问题：如何定义协议，让它既无状态，又支持 POSIX 文件系统 API？

理解 NFS 协议设计的一个关键是理解文件句柄（file handle）。文件句柄用于唯一地描述文件或目录。因此，许多协议请求包括一个文件句柄。

可以认为文件句柄有 3 个重要组件：卷标识符、inode 号和世代号。这 3 项一起构成客户希望访问的文件或目录的唯一标识符。卷标识符通知服务器，请求指向哪个文件系统（NFS 服务器可以导出多个文件系统）。inode 号告诉服务器，请求访问该分区中的哪个文件。最后，复用 inode 号时需要世代号。通过在复用 inode 号时递增它，服务器确保具有旧文件句柄的客户端不会意外地访问新分配的文件。

图 48.4 是该协议的一些重要部分的摘要。完整的协议可在其他地方获得（NFS 的优秀详细概述，请参阅 Callaghan 的书[C00]）。

```
NFSPROC_GETATTR
  expects: file handle
  returns: attributes
NFSPROC_SETATTR
  expects: file handle, attributes
  returns: nothing
NFSPROC_LOOKUP
  expects: directory file handle, name of file/directory to look up
  returns: file handle
NFSPROC_READ
  expects: file handle, offset, count
  returns: data, attributes
NFSPROC_WRITE
  expects: file handle, offset, count, data
  returns: attributes
NFSPROC_CREATE
  expects: directory file handle, name of file, attributes
  returns: nothing
NFSPROC_REMOVE
  expects: directory file handle, name of file to be removed
  returns: nothing
NFSPROC_MKDIR
  expects: directory file handle, name of directory, attributes
  returns: file handle
NFSPROC_RMDIR
  expects: directory file handle, name of directory to be removed
  returns: nothing
```

```
NFSPROC_READDIR
  expects: directory handle, count of bytes to read, cookie
  returns: directory entries, cookie (to get more entries)
```

图 48.4 NFS 协议：示例

我们简单强调一下该协议的重要部分。首先，LOOKUP 协议消息用于获取文件句柄，然后用于访问文件数据。客户端传递目录文件句柄和要查找的文件的名称，该文件（或目录）的句柄及其属性将从服务器传递回客户端。

例如，假设客户端已经有一个文件系统根目录的目录文件句柄（/）[实际上，这是 NFS 挂载协议（mount protocol），它说明客户端和服务器开始如何连接在一起。简洁起见，在此不讨论挂载协议]。如果客户端上运行的应用程序打开文件/foo.txt，则客户端文件系统会向服务器发送查找请求，并向其传递根文件句柄和名称 foo.txt。如果成功，将返回 foo.txt 的文件句柄（和属性）。

属性就是文件系统追踪每个文件的元信息，包括文件创建时间、上次修改时间、大小、所有权和权限信息等，即对文件调用 stat()会返回的信息。

有了文件句柄，客户端可以对一个文件发出 READ 和 WRITE 协议消息，读取和写入该文件。READ 协议消息要求传递文件句柄，以及文件中的偏移量和要读取的字节数。然后，服务器就能发出读取请求（毕竟，该文件句柄告诉了服务器，从哪个卷和哪个 inode 读取，偏移量和字节数告诉它要读取该文件的哪些字节），并将数据返回给客户端 （如果有故障就返回错误代码）。除了将数据从客户端传递到服务器，并返回成功代码之外，WRITE 的处理方式类似。

最后一个有趣的协议消息是 GETATTR 请求。给定文件句柄，它获取该文件的属性，包括文件的最后修改时间。我们将在 NFSv2 中看到，为什么这个协议请求很重要（你能猜到吗）。

48.6 从协议到分布式文件系统

希望你已对该协议如何转换为文件系统有所了解。客户端文件系统追踪打开的文件，通常将应用程序的请求转换为相关的协议消息集。服务器只响应每个协议消息，每个协议消息都具有完成请求所需的所有信息。

例如，考虑一个读取文件的简单应用程序。表 48.1 展示了应用程序进行的系统调用，以及客户端文件系统和文件服务器响应此类调用时的行为。

关于该表有几点说明。首先，请注意客户端如何追踪文件访问的所有相关状态（state），包括整数文件描述符到 NFS 文件句柄的映射，以及当前的文件指针。这让客户端能够将每个读取请求（你可能注意到，读取请求没有显式地指定读取的偏移量），转换为正确格式的读取协议消息，该消息明确地告诉服务器，从文件中读取哪些字节。成功读取后，客户端更新当前文件位置，后续读取使用相同的文件句柄，但偏移量不同。

其次，你可能会注意到，服务器交互发生的位置。当文件第一次打开时，客户端文件系统发送 LOOKUP 请求消息。实际上，如果必须访问一个长路径名（例如/home/remzi/foo.txt），客户端将发送 3 个 LOOKUP：一个在/目录中查找 home，一个在 home 中查找 remzi，最后一个在 remzi 中查找 foo.txt。

第三，你可能会注意到，每个服务器请求如何包含完成请求所需的所有信息。这个设计对于从服务器故障中优雅地恢复的能力至关重要，接下来将更详细地讨论。这确保服务

器不需要状态就能够响应请求。

表 48.1 读取文件：客户端和文件服务器的操作

客户端	服务器
fd = open(“/foo”,…); 发送 LOOKUP (rootdir FH, “foo”)	
	接收 LOOKUP 请求 在 root 目录中查找“foo” 返回 foo 的 FH + 属性
接收 LOOKUP 回复 在打开文件表中分配文件描述符在表中保存 foo 的 FH 保存当前文件位置 (0) 向应用程序返回文件描述符	
read(fd, buffer, MAX); 用 fd 检索打开文件列表 取得 NFS 文件句柄(FH) 使用当前文件位置作为偏移量 发送 READ(FH, offset=0, count=MAX)	
	接收 READ 请求 利用 FH 获取卷/ inode 号 从磁盘（或缓存）读取 inode 计算块位置（利用偏移量） 从磁盘（或缓存）读取数据 向客户端返回数据
接收 READ 回复 更新文件位置（+读取的字节数） 设置当前文件位置= MAX 向应用程序返回数据/错误代码	
read(fd, buffer, MAX); 除了偏移量=MAX，设置当前文件位置= 2*MAX 外，都一样	
read(fd, buffer, MAX); 除了偏移量=2*MAX，设置当前文件位置= 3*MAX 外，都一样	
close(fd); 只需要清理本地数据结构 释放打开文件 表中的描述符“fd” （不需要与服务器通信）	

提示：幂等性很强大

在构建可靠的系统时，幂等性（idempotency）是一种有用的属性。如果一次操作可以发出多次请求，那么处理该操作的失败就要容易得多。你只要重试一下。如果操作不具有幂等性，那么事情就会更困难。

48.7 利用幂等操作处理服务器故障

当客户端向服务器发送消息时，有时候不会收到回复。这种失败可能的原因很多。在某些情况下，网络可能会丢弃该消息。网络确实会丢失消息，因此请求或回复可能会丢失，

所以客户端永远不会收到响应，如图 48.5 所示。

也可能服务器已崩溃，因此无法响应消息。稍后，服务器将重新启动，并再次开始运行，但所有请求都已丢失。在所有这些情况下，客户端有一个问题：如果服务器没有及时回复，应该怎么做？

在 NFSv2 中，客户端以唯一、统一和优雅的方式处理所有这些故障：就是重试请求。具体来说，在发送请求之后，客户端将计时器设置为在指定的时间之后关闭。如果在定时器关闭之前收到回复，则取消定时器，一切正常。但是，如果在收到任何回复之前计时器关闭，则客户端会假定请求尚未处理，并重新发送。如果服务器回复，一切都很好，客户端已经漂亮地处理了问题。

客户端之所以能够简单重试请求（不论什么情况导致了故障），是因为大多数 NFS 请求有一个重要的特性：它们是幂等的（idempotent）。如果操作执行多次的效果与执行一次的效果相同，该操作就是幂等的。例如，如果将值在内存位置存储 3 次，与存储一次一样。因此“将值存储到内存中”是一种幂等操作。但是，如果将计数器递增 3 次，它的数量就会与递增一次不同。因此，递增计数器不是幂等的。更一般地说，任何只读取数据的操作显然都是幂等的。对更新数据的操作必须更仔细地考虑，才能确定它是否具有幂等性。

NFS 中崩溃恢复的核心在于，大多数常见操作具有幂等性。LOOKUP 和 READ 请求是简单幂等的，因为它们只从文件服务器读取信息而不更新它。更有趣的是，WRITE 请求也是幂等的。例如，如果 WRITE 失败，客户端可以简单地重试它。WRITE 消息包含数据、计数和（重要的）写入数据的确切偏移量。因此，可以重复多次写入，因为多次写入的结果与单次的结果相同。

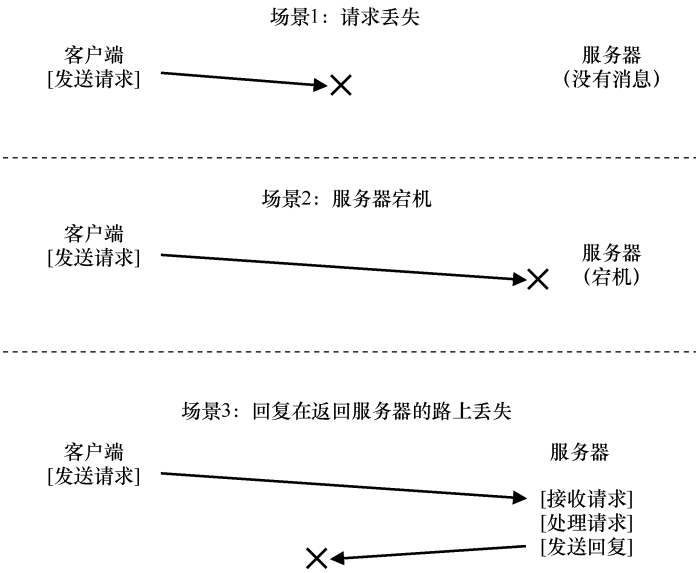

图 48.5 3 种类型的丢失

通过这种方式，客户端可以用统一的方式处理所有超时。如果 WRITE 请求丢失（上面的第一种情况），客户端将重试它，服务器将执行写入，一切都会好。如果在请求发送时，服务器恰好关闭，但在第二个请求发送时，服务器已重启并继续运行，则又会如愿执行（第

二种情况）。最后，服务器可能实际上收到了 WRITE 请求，发出写入磁盘并发送回复。此回复可能会丢失（第三种情况），导致客户端重新发送请求。当服务器再次收到请求时，它就会执行相同的操作：将数据写入磁盘，并回复它已完成该操作。如果客户端这次收到了回复，则一切正常，因此客户端以统一的方式处理了消息丢失和服务器故障。漂亮！

一点补充：一些操作很难成为幂等的。例如，当你尝试创建已存在的目录时，系统会通知你 mkdir 请求已失败。因此，在 NFS 中，如果文件服务器收到 MKDIR 协议消息并成功执行，但回复丢失，则客户端可能会重复它并遇到该故障，实际上该操作第一次成功了，只是在重试时失败。所以，生活并不完美。

提示：完美是好的敌人（Voltaire 定律）

即使你设计了一个漂亮的系统，有时候并非所有的特殊情况都像你期望的那样。以上面的 mkdir 为例，你可以重新设计 mkdir，让它具有不同的语义，从而让它成为幂等的（想想你会怎么做）。但是，为什么要这么麻烦？NFS 的设计理念涵盖了大多数重要情况，它使系统设计在故障方面简洁明了。因此，接受生活并不完美的事实，仍然构建系统，这是良好工程的标志。显然，这种智慧应该要感谢伏尔泰，他说：“一个聪明的意大利人说，最好是好的敌人。”因此我们称之为 Voltaire 定律。

48.8 提高性能：客户端缓存

分布式文件系统很多，这有很多原因，但将所有读写请求都通过网络发送，会导致严重的性能问题：网络速度不快，特别是与本地内存或磁盘相比。因此，另一个问题是：如何才能改善分布式文件系统的性能？

答案你可能已经猜到（看到上面的节标题），就是客户端缓存（caching）。NFS 客户端文件系统缓存文件数据（和元数据）。因此，虽然第一次访问是昂贵的（即它需要网络通信），但后续访问很快就从客户端内存中得到服务。

缓存还可用作写入的临时缓冲区。当客户端应用程序写入文件时，客户端会在数据写入服务器之前，将数据缓存在客户端的内存中（与数据从文件服务器读取的缓存一样）。这种写缓冲（write buffering）是有用的，因为它将应用程序的 write()延迟与实际的写入性能分离，即应用程序对 write()的调用会立即成功（只是将数据放入客户端文件系统的缓存中），只是稍后才会将数据写入文件服务器。

因此，NFS 客户端缓存数据和性能通常很好，我们成功了，对吧？遗憾的是，并没完全成功。在任何系统中添加缓存，导致包含多个客户端缓存，都会引入一个巨大且有趣的挑战，我们称之为缓存一致性问题（cache consistency problem）。

48.9 缓存一致性问题

利用两个客户端和一个服务器，可以很好地展示缓存一致性问题。想象一下客户端 C1 读取文件 F，并将文件的副本保存在其本地缓存中。现在假设一个不同的客户端 C2 覆盖文

件 F，从而改变其内容。我们称该文件的新版本为 F（版本 2），或 F [v2]，称旧版本为 F [v1]，以便区分两者。最后，还有第三个客户端 C3，尚未访问文件 F。

你可能会看到即将发生的问题（见图 48.6）。实际上，有两个子问题。第一个子问题是，客户端 C2 可能将它的写入缓存一段时间，然后再将它们发送给服务器。在这种情况下，当 F[v2]位于 C2 的内存中时，来自另一个客户端（比如 C3）的任何对 F 的访问，都会获得旧版本的文件（F[v1]）。因此，在客户端缓冲写入，可能导致其他客户端获得文件的陈旧版本，这也许不是期望的结果。实际上，想象一下你登录机器 C2，更新 F，然后登录 C3，并尝试读取文件：只得到了旧版本！这当然会令人沮丧。因此，我们称这个方面的缓存一致性问题为“更新可见性（update visibility）”。来自一个客户端的更新，什么时候被其他客户端看见？

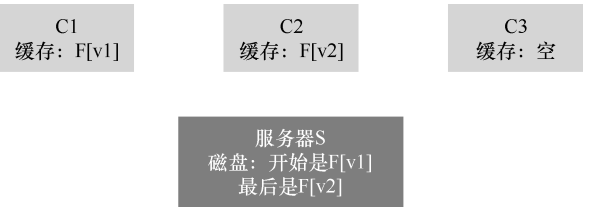

图 48.6 缓存一致性问题

缓存一致性的第二个子问题是陈旧的缓存（stale cache）。在这种情况下，C2 最终将它的写入发送给文件服务器，因此服务器具有最新版本（F[v2]）。但是，C1 的缓存中仍然是 F[v1]。如果运行在 C1 上的程序读了文件 F，它将获得过时的版本（F [v1]），而不是最新的版本（F [v2]），这（通常）不是期望的结果。

NFSv2 实现以两种方式解决了这些缓存一致性问题。首先，为了解决更新可见性，客户端实现了有时称为“关闭时刷新”（flush-on-close，即 close-to-open）的一致性语义。具体来说，当应用程序写入文件并随后关闭文件时，客户端将所有更新（即缓存中的脏页面）刷新到服务器。通过关闭时刷新的一致性，NFS 可确保后续从另一个节点打开文件，会看到最新的文件版本。

其次，为了解决陈旧的缓存问题，NFSv2 客户端会先检查文件是否已更改，然后再使用其缓存内容。具体来说，在打开文件时，客户端文件系统会发出 GETATTR 请求，以获取文件的属性。重要的是，属性包含有关服务器上次修改文件的信息。如果文件修改的时间晚于文件提取到客户端缓存的时间，则客户端会让文件无效（invalidate），因此将它从客户端缓存中删除，并确保后续的读取将转向服务器，取得该文件的最新版本。另外，如果客户端看到它持有该文件的最新版本，就会继续使用缓存的内容，从而提高性能。

当 Sun 最初的团队实现陈旧缓存问题的这个解决方案时，他们意识到一个新问题。突然，NFS 服务器充斥着 GETATTR 请求。一个好的工程原则，是为常见情况而设计，让它运作良好。这里，尽管常见情况是文件只由一个客户端访问（可能反复访问），但该客户端必须一直向服务器发送 GETATTR 请求，以确没人改变该文件。客户因此“轰炸”了服务器，不断询问“有没有人修改过这个文件？”，大部分时间都没有人修改。

为了解决这种情况（在某种程度上），为每个客户端添加了一个属性缓存（attribute cache）。客户端在访问文件之前仍会验证文件，但大多数情况下只会查看属性缓存以获取属性。首次访问某文件时，该文件的属性被放在缓存中，然后在一定时间（例如 3s）后超时。

因此，在这 3s 内，所有文件访问都会断定使用缓存的文件没有问题，并且没有与服务器的网络通信。

48.10 评估 NFS 的缓存一致性

关于 NFS 的缓存一致性还有几句话。加入关闭时刷新的行为是因为“有意义”，但带来了一定的性能问题。具体来说，如果在客户端上创建临时或短期文件，然后很快删除，它仍将被强制写到服务器。更理想的实现可能会将这些短暂的文件保留在内存中，直到它们被删除，从而完全消除服务器交互，提高性能。

更重要的是，NFS 加入属性缓存让它很难知道或推断出得到文件的确切版本。有时你会得到最新版本，有时你会得到旧版本，因为属性缓存没有超时，因此客户端很高兴地提供了客户端内存中的内容。虽然这在大多数情况下都没问题，但它偶尔会（现在仍然如此！）导致奇怪的行为。

我们已经描述了 NFS 客户端缓存的奇怪之处。它是一个有趣的例子，其中实现的细节致力于定义用户可观察的语义，而不是相反。

48.11 服务器端写缓冲的隐含意义

我们的重点是客户端缓存，这是最有趣的问题出现的地方。但是，NFS 服务器也往往配备了大量内存，因此它们也存在缓存问题。从磁盘读取数据（和元数据）时，NFS 服务器会将其保留在内存中，后续读取这些数据（和元数据）不会访问磁盘，这可能对性能有（小）提升。

更有趣的是写缓冲的情况。在强制写入稳定存储（即磁盘或某些其他持久设备）之前，NFS 服务器绝对不会对 WRITE 协议请求返回成功。虽然他们可以将数据的拷贝放在服务器内存中，但对 WRITE 协议请求向客户端返回成功，可能会导致错误的行为。你能搞清楚为什么吗？

答案在于我们对客户端如何处理服务器故障的假设。想象一下客户端发出以下写入序列：

```
write(fd, a_buffer, size); // fill first block with a's
write(fd, b_buffer, size); // fill second block with b's
write(fd, c_buffer, size); // fill third block with c's
```

这些写入覆盖了文件的 3 个块，先是 a，然后是 b，最后是 c。因此，如果文件最初看起来像这样：

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
```

我们可能期望这些写入之后的最终结果是这样：x、y 和 z 分别用 a、b 和 c 覆盖。

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
```

现在假设，在这个例子中，客户端的 3 个写入作为 3 个不同的 WRITE 协议消息，发送给服务器。假设服务器接收到第一个 WRITE 消息，将它发送到磁盘，并向客户端通知成功。现在假设第二次写入只是缓冲在内存中，服务器在强制写入磁盘之前，也向客户端报告成功。遗憾的是，服务器在写入磁盘之前崩溃了。服务器快速重启，并接收第三个写请求，该请求也成功了。

因此，对于客户端，所有请求都成功了，但我们很惊讶文件的内容如下：

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy  <---  oops
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
```

因为服务器在提交到磁盘之前，告诉客户端第二次写入成功，所以文件中会留下一个旧块，这对于某些应用程序，可能是灾难性的。

为了避免这个问题，NFS 服务器必须在通知客户端成功之前，将每次写入提交到稳定（持久）存储。这样做可以让客户端在写入期间检测服务器故障，从而重试，直到它最终成功。这样做确保了不会导致前面例子中混合的文件内容。

这个需求，对 NFS 服务器的实现带来一个问题，即写入性能，如果不小心，会成为主要的性能瓶颈。实际上，一些公司（例如 Network Appliance）的出现，只是为了构建一个可以快速执行写入的 NFS 服务器。一个技巧是先写入有电池备份的内存，从而快速报告 WRITE 请求成功，而不用担心丢失数据，也没有必须立即写入磁盘的成本。第二个技巧是采用专门为快速写入磁盘而设计的文件系统，如果你最后需要这样做[HLM94，RO91]。

48.12 小结

我们已经介绍了 NFS 分布式文件系统。NFS 的核心在于，服务器的故障要能简单快速地恢复。操作的幂等性至关重要，因为客户端可以安全地重试失败的操作，不论服务器是否已执行该请求，都可以这样做。

我们还看到，将缓存引入多客户端、单服务器的系统，如何会让事情变得复杂。具体来说，系统必须解决缓存一致性问题，才能合理地运行。但是，NFS 以稍微特别的方式来解决这个问题，偶尔会导致你看到奇怪的行为。最后，我们看到了服务器缓存如何变得棘手：对服务器的写入，在返回成功之前，必须强制写入稳定存储（否则数据可能会丢失）。

我们还没有谈到其他一些问题，这些问题肯定有关，尤其是安全问题。早期 NFS 实现中，安全性非常宽松。客户端的任何用户都可以轻松伪装成其他用户，并获得对几乎任何文件的访问权限。后来集成了更严肃的身份验证服务（例如，Kerberos [NT94]），解决了这些明显的缺陷。

参考资料

[S86]“The Sun Network File System: Design, Implementation and Experience”Russel Sandberg

USENIX Summer 1986
最初的 NFS 论文。阅读这些美妙的想法是个好主意。

[NT94]“Kerberos: An Authentication Service for Computer Networks”
B. Clifford Neuman, Theodore Ts’o
IEEE Communications, 32(9):33-38, September 1994
Kerberos 是一种早期且极具影响力的身份验证服务。我们可能应该在某个时候为它写上一章……

[P+94]“NFS Version 3: Design and Implementation”
Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, Dave Hitz USENIX Summer 1994, pages 137-152
NFS 版本 3 的小修改。

[P+00]“The NFS version 4 protocol”
Brian Pawlowski, David Noveck, David Robinson, Robert Thurlow
2nd International System Administration and Networking Conference (SANE 2000)
毫无疑问，这是有史以来关于 NFS 的优秀论文。

[C00]“NFS Illustrated”Brent Callaghan
Addison-Wesley Professional Computing Series, 2000
一个很棒的 NFS 参考，每个协议都讲得非常彻底和详细。

[Sun89]“NFS: Network File System Protocol Specification”
Sun Microsystems, Inc. Request for Comments: 1094, March 1989
可怕的规范。如果你必须读，就读它。

[O91]“The Role of Distributed State”John K. Ousterhout
很少引用的关于分布式状态的讨论，对问题和挑战有更广的视角。

[HLM94]“File System Design for an NFS File Server Appliance”Dave Hitz, James Lau, Michael Malcolm
USENIX Winter 1994. San Francisco, California, 1994
Hitz 等人受到以前日志结构文件系统工作的极大影响。

[RO91]“The Design and Implementation of the Log-structured File System”Mendel Rosenblum, John Ousterhout
Symposium on Operating Systems Principles (SOSP), 1991
又是 LFS。不，LFS，学无止境。

第 49 章　Andrew 文件系统（AFS）

Andrew 文件系统由卡内基梅隆大学（CMU）的研究人员于 20 世纪 80 年代[H+88]引入。该项目由卡内基梅隆大学著名教授 M. Satyanarayanan（简称为 Satya）领导，主要目标很简单：扩展（scale）。具体来说，如何设计分布式文件系统（如服务器）可以支持尽可能多的客户端？

有趣的是，设计和实现的许多方面都会影响可扩展性。最重要的是客户端和服务器之间的协议（protocol）设计。例如，在 NFS 中，协议强制客户端定期检查服务器，以确定缓存的内容是否已更改。因为每次检查都使用服务器资源（包括 CPU 和网络带宽），所以频繁的检查会限制服务器响应的客户端数量，从而限制可扩展性。

AFS 与 NFS 的不同之处也在于，从一开始，合理的、用户可见的行为就是首要考虑的问题。在 NFS 中，缓存一致性很难描述，因为它直接依赖于低级实现细节，包括客户端缓存超时间隔。在 AFS 中，缓存一致性很简单且易于理解：当文件打开时，客户端通常会从服务器接收最新的一致副本。

49.1　AFS 版本 1

我们将讨论两个版本的 AFS [H+88，S+85]。第一个版本（我们称之为 AFSv1，但实际上原来的系统被称为 ITC 分布式文件系统[S+85]）已经有了一些基本的设计，但没有像期望那样可扩展，这导致了重新设计和最终协议（我们称之为 AFSv2，或就是 AFS）[H+88]。现在讨论第一个版本。

所有 AFS 版本的基本原则之一，是在访问文件的客户端计算机的本地磁盘（local disk）上，进行全文件缓存（whole-file caching）。当 open()文件时，将从服务器获取整个文件（如果存在），并存储在本地磁盘上的文件中。后续应用程序 read()和 write()操作被重定向到存储文件的本地文件系统。因此，这些操作不需要网络通信，速度很快。最后，在 close()时，文件（如果已被修改）被写回服务器。注意，与 NFS 的明显不同，NFS 缓存块（不是整个文件，虽然 NFS 当然可以缓存整个文件的每个块），并且缓存在客户端内存（不是本地磁盘）中。

让我们进一步了解细节。当客户端应用程序首次调用 open()时，AFS 客户端代码（AFS 设计者称之为 Venus）将向服务器发送 Fetch 协议消息。Fetch 协议消息会将所需文件的整个路径名（例如/home/remzi/notes.txt）传递给文件服务器（它们称为 Vice 的组），然后将沿着路径名，查找所需的文件，并将整个文件发送回客户端。然后，客户端代码将文件缓存在客户端的本地磁盘上（将它写入本地磁盘）。如上所述，后续的 read()和 write()系统调用在 AFS 中是严格本地的（不与服务器进行通信）。它们只是重定向到文件的本地副本。因为 read()和 write()调用就像调用本地文件系统一样，一旦访问了一个块，它也可以缓存在客户端内存中。因此，AFS 还使用客户端内存来缓存它在本地磁盘中的块副本。最后，AFS 客户端完

成后检查文件是否已被修改（即它被打开并写入）。如果被修改，它会用 Store 协议消息，将新版本刷写回服务器，将整个文件和路径名发送到服务器以进行持久存储。

下次访问该文件时，AFSv1 的效率会更高。具体来说，客户端代码首先联系服务器（使用 TestAuth 协议消息），以确定文件是否已更改。如果未更改，客户端将使用本地缓存的副本，从而避免了网络传输，提高了性能。表 49.1 展示了 AFSv1 中的一些协议消息。请注意，协议的早期版本仅缓存文件内容。例如，目录只保存在服务器上。

表 49.1 AFSv1 协议的要点

TestAuth	测试文件是否已改变（用于验证缓存条目的有效性）
GetFileStat	取得文件的状态信息
Fetch	获取文件的内容
Store	将文件存入服务器
SetFileStat	设置文件的状态信息
ListDir	列出目录的内容

49.2 版本 1 的问题

第一版 AFS 的一些关键问题，促使设计人员重新考虑他们的文件系统。为了详细研究这些问题，AFS 的设计人员花费了大量时间来测量他们已有的原型，以找出问题所在。这样的实验是一件好事。测量（measurement）是理解系统如何工作，以及如何改进系统的关键。实际数据有助于取代直觉，让解构系统成为具体的科学。在他们的研究中，作者发现了 AFSv1 的两个主要问题。

提示：先测量后构建（Patterson 定律）

我们的顾问之一，David Patterson（因 RISC 和 RAID 而著名），过去总是鼓励我们先测量系统并揭示问题，再构建新系统来修复所述问题。通过使用实验证据而不是直觉，你可以将系统构建过程变成更科学的尝试。这样做也具有让你在开发改进版本之前，先考虑如何准确测量系统的优势。当你最终开始构建新系统时，结果两件事情会变得更好：首先，你有证据表明你正在解决一个真正的问题。第二，你现在有办法测量新系统，以显示它实际上改进了现有技术。因此我们称之为 Patterson 定律。

- **路径查找成本过高**。执行 Fetch 或 Store 协议请求时，客户端将整个路径名（例如 /home/remzi/notes.txt）传递给服务器。为了访问文件，服务器必须执行完整的路径名遍历，首先查看根目录以查找 home，然后在 home 中查找 remzi，依此类推，一直沿着路径直到最终定位所需的文件。由于许多客户端同时访问服务器，AFS 的设计人员发现服务器花费了大量的 CPU 时间，只是在沿着目录路径走。
- **客户端发出太多 TestAuth 协议消息**。与 NFS 及其过多的 GETATTR 协议消息非常相似，AFSv1 用 TestAuth 协议信息，生成大量流量，以检查本地文件（或其状态信息）是否有效。因此，服务器花费大量时间，告诉客户端是否可以使用文件的缓存副本。大多数时候，答案是文件没有改变。

AFSv1 实际上还存在另外两个问题：服务器之间的负载不均衡，服务器对每个客户端

使用一个不同的进程，从而导致上下文切换和其他开销。通过引入卷（volume），解决了负载不平衡问题。管理员可以跨服务器移动卷，以平衡负载。通过使用线程而不是进程构建服务器，在 AFSv2 中解决了上下文切换问题。但是，限于篇幅，这里集中讨论上述主要的两个协议问题，这些问题限制了系统的扩展。

49.3 改进协议

上述两个问题限制了 AFS 的可扩展性。服务器 CPU 成为系统的瓶颈，每个服务器只能服务 20 个客户端而不会过载。服务器收到太多的 TestAuth 消息，当他们收到 Fetch 或 Store 消息时，花费了太多时间查找目录层次结构。因此，AFS 设计师面临一个问题。

关键问题：如何设计一个可扩展的文件协议

如何重新设计协议，让服务器交互最少，即如何减少 TestAuth 消息的数量？进一步，如何设计协议，让这些服务器交互高效？通过解决这两个问题，新的协议将导致可扩展性更好的 AFS 版本。

49.4 AFS 版本 2

AFSv2 引入了回调（callback）的概念，以减少客户端/服务器交互的数量。回调就是服务器对客户端的承诺，当客户端缓存的文件被修改时，服务器将通知客户端。通过将此状态（state）添加到服务器，客户端不再需要联系服务器，以查明缓存的文件是否仍然有效。实际上，它假定文件有效，直到服务器另有说明为止。这里类似于轮询（polling）与中断（interrupt）。

AFSv2 还引入了文件标识符（File Identifier，FID）的概念（类似于 NFS 文件句柄），替代路径名，来指定客户端感兴趣的文件。AFS 中的 FID 包括卷标识符、文件标识符和“全局唯一标识符”（用于在删除文件时复用卷和文件 ID）。因此，不是将整个路径名发送到服务器，并让服务器沿着路径名来查找所需的文件，而是客户端会沿着路径名查找，每次一个，缓存结果，从而有望减少服务器上的负载。

例如，如果客户端访问文件/home/remzi/notes.txt，并且 home 是挂载在/上的 AFS 目录（即/是本地根目录，但 home 及其子目录在 AFS 中），则客户端将先获取 home 的目录内容，将它们放在本地磁盘缓存中，然后在 home 上设置回调。然后，客户端将获取目录 remzi，将其放入本地磁盘缓存，并在服务器上设置 remzi 的回调。最后，客户端将获取 notes.txt，将此常规文件缓存在本地磁盘中，设置回调，最后将文件描述符返回给调用应用程序。有关摘要，参见表 49.2。

然而，与 NFS 的关键区别在于，每次获取目录或文件时，AFS 客户端都会与服务器建立回调，从而确保服务器通知客户端，其缓存状态发生变化。好处是显而易见的：尽管第一次访问/home/remzi/notes.txt 会生成许多客户端—服务器消息（如上所述），但它也会为所有目录以及文件 notes.txt 建立回调，因此后续访问完全是本地的，根本不需要服务器交互。因此，在客户端缓存文件的常见情况下，AFS 的行为几乎与基于本地磁盘的文件系统相同。如果多次访问一个文件，则第二次访问应该与本地访问文件一样快。

表 49.2 读取文件：客户端和文件服务器操作

客户端 (C1)	服务器
fd = open(“/home/remzi/notes.txt”,…); 发送 Fetch (home FID, “remzi”)	
	接收 Fetch 请求 在 home 目录中查找 remzi 对 remzi 建立 callback(C1) 返回 remzi 的内容和 FID
接收 Fetch 回复 将 remzi 写入本地磁盘缓存 记录 remzi 的回调状态 发送 Fetch (remzi FID, “notes.txt”)	
	接收 Fetch 请求 在 remzi 目录中查找 notes.txt 对 notes.txt 建立 callback(C1) 返回 notes.txt 的内容和 FID
接收 Fetch 回复 将 notes.txt 写入本地磁盘缓存 记录 notes.txt 的回调状态 本地 open() 缓存的 notes.txt 向应用程序返回文件描述符	
read(fd, buffer, MAX); 执行本地 read()缓存副本	
close(fd); 执行本地 close()缓存副本 如果文件已改变，刷新到服务器	

fd = open(“/home/remzi/notes.txt”,…);
Foreach dir (home, remzi)
if (callback(dir) == VALID)
使用本地副本执行 lookup(dir)
else
Fetch (像上面一样)
if (callback(notes.txt) == VALID)
open 本地缓存副本
return 它的文件描述符
else
Fetch (像上面一样) then open 并 return fd	

补充：缓存一致性不能解决所有问题

在讨论分布式文件系统时，很多都是关于文件系统提供的缓存一致性。但是，关于多个客户端访问文件，这种基本一致性并未解决所有问题。例如，如果要构建代码存储库，并且有多个客户端检入和检出代码，则不能简单地依赖底层文件系统来为你完成所有工作。实际上，你必须使用显式的文件级锁（file-level locking），以确保在发生此类并发访问时，发生“正确”的事情。事实上，任何真正关心并发更新的应用程序，都会增加额外的机制来处理冲突。本章和第 48 章中描述的基本一致性主要用于随意使用，例如，当用户在不同的客户端登录时，他们希望看到文件的某个合理版本。对这些协议期望过多，会让自己陷入挫败、失望和泪流满面的沮丧。

49.5 缓存一致性

讨论 NFS 时，我们考虑了缓存一致性的两个方面：更新可见性（update visibility）和缓存陈旧（cache staleness）。对于更新可见性，问题是：服务器何时用新版本的文件进行更新？对于缓存陈旧，问题是：一旦服务器有新版本，客户端看到新版本而不是旧版本缓存副本，需要多长时间？

由于回调和全文件缓存，AFS 提供的缓存一致性易于描述和理解。有两个重要的情况需要考虑：不同机器上进程的一致性，以及同一台机器上进程的一致性。

在不同的计算机之间，AFS 让更新在服务器上可见，并在同一时间使缓存的副本无效，即在更新的文件被关闭时。客户端打开一个文件，然后写入（可能重复写入）。当它最终关闭时，新文件被刷新到服务器（因此可见）。然后，服务器中断任何拥有缓存副本的客户端的回调，从而确保客户端不再读取文件的过时副本。在这些客户端上的后续打开，需要从服务器重新获取该文件的新版本。

对于这个简单模型，AFS 对同一台机器上的不同进程进行了例外处理。在这种情况下，对文件的写入对于其他本地进程是立即可见的（进程不必等到文件关闭，就能查看其最新更新版本）。这让使用单个机器完全符合你的预期，因为此行为基于典型的 UNIX 语义。只有切换到不同的机器时，你才会发现更一般的 AFS 一致性机制。

有一个有趣的跨机器场景值得进一步讨论。具体来说，在极少数情况下，不同机器上的进程会同时修改文件，AFS 自然会采用所谓的“最后写入者胜出”方法（last writer win，也许应该称为“最后关闭者胜出”，last closer win）。具体来说，无论哪个客户端最后调用 close()，将最后更新服务器上的整个文件，因此将成为“胜出”文件，即保留在服务器上，供其他人查看。结果是文件完全由一个客户端或另一个客户端生成。请注意与基于块的协议（如 NFS）的区别：在 NFS 中，当每个客户端更新文件时，可能会将各个块的写入刷新到服务器，因此服务器上的最终文件最终可能会混合为来自两个客户的更新。在许多情况下，这样的混合文件输出没有多大意义，例如，想象一个 JPEG 图像被两个客户端分段修改，导致的混合写入不太可能构成有效的 JPEG。

在表 49.3 中可以看到，展示其中一些不同场景的时间线。这些列展示了 Client1 上的两个进程（P1 和 P2）的行为及其缓存状态，Client2 上的一个进程（P3）及其缓存状态，以及服务器（Server），它们都在操作一个名为的 F 文件。对于服务器，该表只展示了左边的操作完成后该文件的内容。仔细查看，看看你是否能理解每次读取的返回结果的原因。如果想不通，右侧的“评论”字段对你会有所帮助。

表 49.3 缓存一致性时间线

P1	Client1 P2	Cache	Client2 P3	Cache	Server Disk	评论
open(F)		—		—	—	文件创建
write(A)		A		—	—	
close()		A		—	A	

续表

P1	Client1 P2	Cache	Client2 P3	Cache	Server Disk	评论
	open(F)	A		—	A	
	read()→A	A		—	A	
	close()	A		—	A	
open(F)		A		—	A	
write(B)		B		—	A	
	open(F)	B		—	A	本地进程
	read()→B	B		—	A	马上看到写入
	close()	B		—	A	
		B	open(F)	A	A	远程进程
		B	read()→A	A	A	没有看到写入……
		B	close()	A	A	
close()		B B		~~A~~	B	……直到发生 close()
			open(F)	B	B	
		B	read()→B	B	B	
		B	close()	B	B	
		B	open(F)	B	B	
open(F)		B		B	B	
write(D)		D		B	B	
		D	write(C)	C	B	
		D	close()	C	C	
close()		D D		~~C~~ D	D D	
			open(F)			P3 很不幸
		D	read()→D	D	D	最后写入者胜出
		D	close()	D	D	

49.6 崩溃恢复

从上面的描述中，你可能会感觉，崩溃恢复比 NFS 更复杂。你是对的。例如，假设有一小段时间，服务器（S）无法联系客户端（C1），比方说，客户端 C1 正在重新启动。当 C1 不可用时，S 可能试图向它发送一个或多个回调撤销消息。例如，假设 C1 在其本地磁盘上缓存了文件 F，然后 C2（另一个客户端）更新了 F，从而导致 S 向缓存该文件的所有客户端发送消息，以便将它从本地缓存中删除。因为 C1 在重新启动时可能会丢失这些关键消息，所以在重新加入系统时，C1 应该将其所有缓存内容视为可疑。因此，在下次访问文件 F 时，C1 应首先向服务器（使用 TestAuth 协议消息）询问，其文件 F 的缓存副