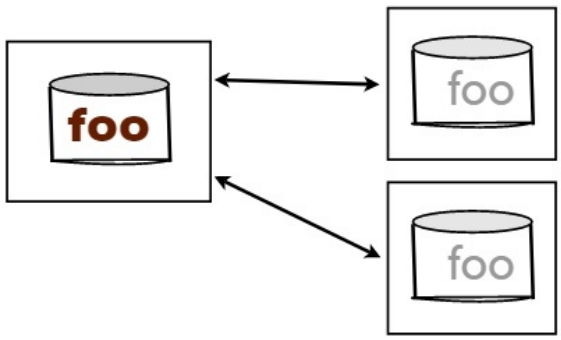
zhuanlan.zhihu.com

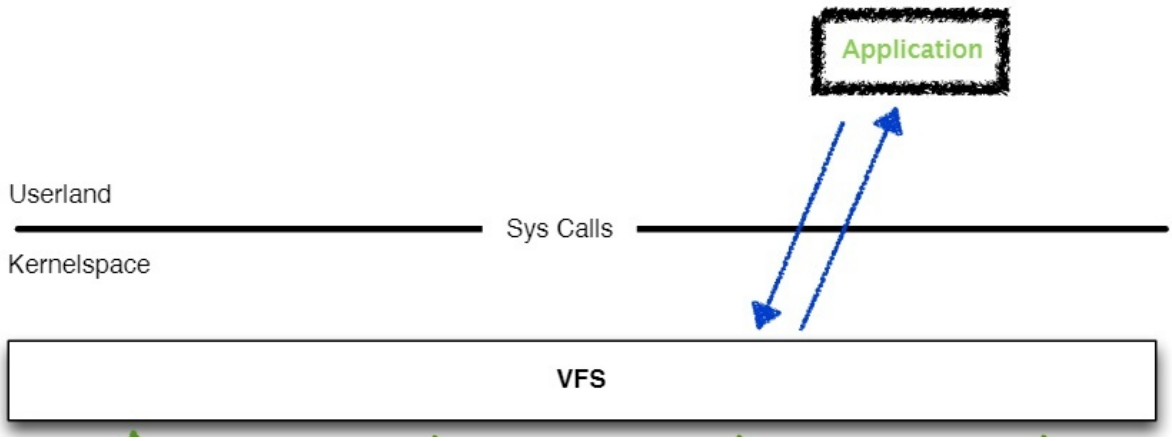
分布式文件系统之NFS

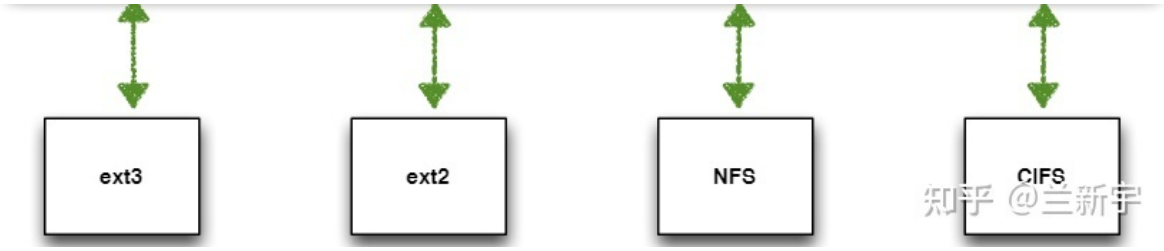
16-20 minutes

「分布式」是现在蛮流行的一个词，而其盛行，离不开底层网络通信能力的迅速发展。在文件系统这个领域，早期的分布式实现更多的是用来实现「共享」，而不是「容错」。传统的集中式文件系统允许单个系统中的多用户共享本地存储的文件，而分布式文件系统将共享的范围扩展到通过网络互连的不同机器的用户中。



最早的分布式文件系统诞生自上世纪70年代，目前依然在使用的FTP算是其中老资格的代表，但FTP的使用涉及文件的上传和下载，而随后出现的NFS，提供了像访问本地文件一样访问远程文件的方式。

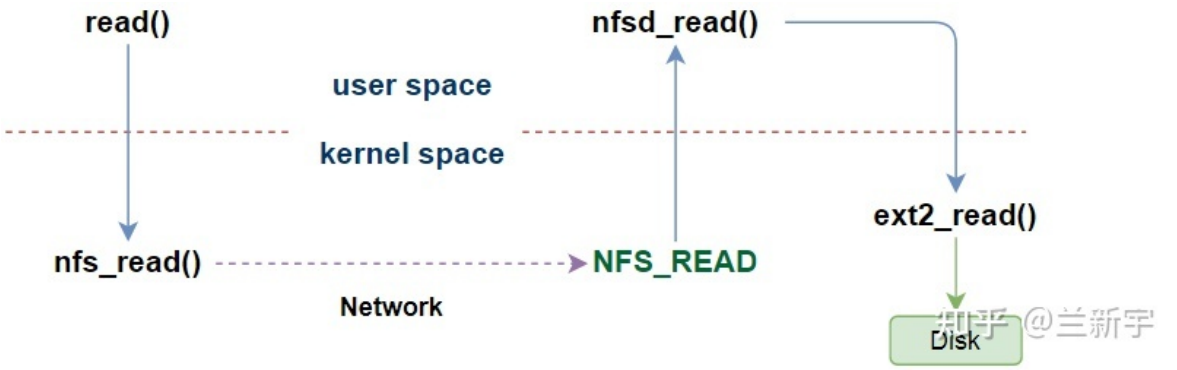




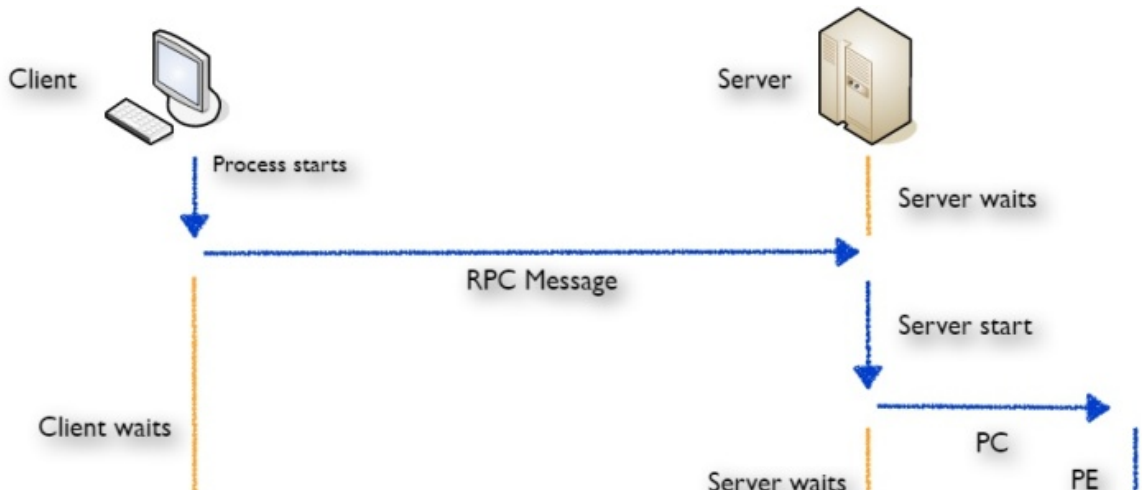
NFS直译过来是「网络文件系统」，但其实网络文件系统是一个大类，为了区分，我们可以把它叫做"Sun Network File System"。

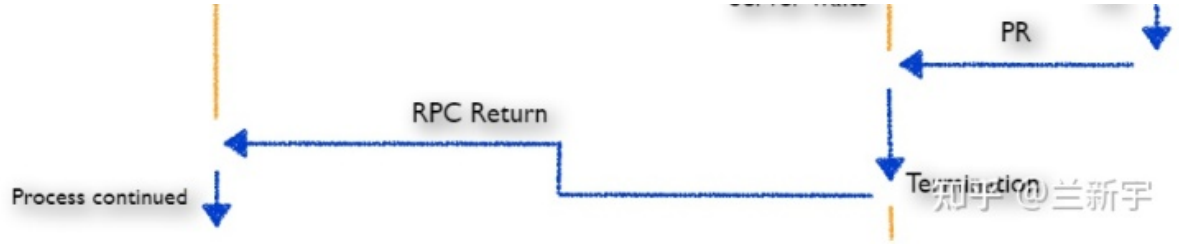
NFS的实现基于**client-server**模型，当client的用户态程序使用read()系统调用后，client位于内核的文件系统将通过网络传输，向server的文件系统发送读取某个block数据的请求。

server收到请求后，从自己的disk（或者page cache）读取对应的block数据，发送给client。然后，client的文件系统将收到的数据复制到read()入参指定的user buffer中。



虽然文件的实体在远端，但从client端用户程序的角度，read()调用返回的结果与对本地文件系统的访问无异，是透明的，这属于一种远程过程调用（**RPC**）。





RPC本身不依赖于任何传输协议，但其通常是基于UDP/IP实现的，为了保证传输服务的可靠性，RPC层会跟踪所有未回复的请求，如果没有收到答复，就按一定的时间间隔重传请求。

- **谁来维护状态**

  当client发送访问文件的请求后，由server返回给client一个文件描述符，之后当client试图访问这个文件时，将文件描述符传给server就可以了，由server来维护这个文件的状态信息，这种模式被称为**stateful**。

  与之相对的另一种方式是：哪个文件被client打开了，访问到文件的哪个位置了（即"offset"），server通通都不记录，由client自己维护，并在发送读写文件请求时，包含所需的全部信息（self-contained），即**stateless**模式。

  stateful模式平时用着没什么问题，但当server和client的任意一方出现**crash**的时候，都将给recovery增加复杂性。比如当server给client返回一个文件描述符后，自己崩溃了，之后重启运行后，client拿着这个文件描述符来找它，它就不记得自己前世的事情（该文件描述符对应哪个文件啊）。

  虽然在崩溃恢复上可以少操很多心，但面对crash的问题，stateless模式也并非就可以高枕无忧。当server崩溃后，client会发现请求没有响应，于是持续地重发请求，直到server重启后回复请求，所以client实际上没法区分server是崩溃重启了，还是运行比较缓慢（或者网络连接存在异常）。

  正因为此，当client发送写block的请求后，server需要等到数据真的写到disk上，才能回复client（其实就是"write-through"方式）。要

不然client会误以为已经写成功了，实际上没有。对于本地文件系统，除非使用sync语义，write()调用只要到达page cache就可以返回，这是因为如果crash发生在本地的话，至少能确切地知道，数据可能是丢失或者不一致的。

- **NFS的选择**

  stateful和stateless各有利弊，作为以fast crash recovery为设计目标的NFS，其v2（1989年）和v3（1995年）版本的实现采用的都是stateless模型。

  ### 打开操作

  既然NFS对于client是“透明的”文件操作，那要访问文件，第一步肯定是open()调用。client的open请求，在server端将转换成一系列的**lookup**操作：

```
fd = open(“/usr/joe/6360/list.txt”)

lookup(rootfh, “usr”) returns (fh0, attr)
lookup(fh0, “joe”) returns (fh1, attr)
lookup(fh1, “6360”) returns (fh2, attr)
lookup(fh2, “list.txt”) returns (fh, attr)
```

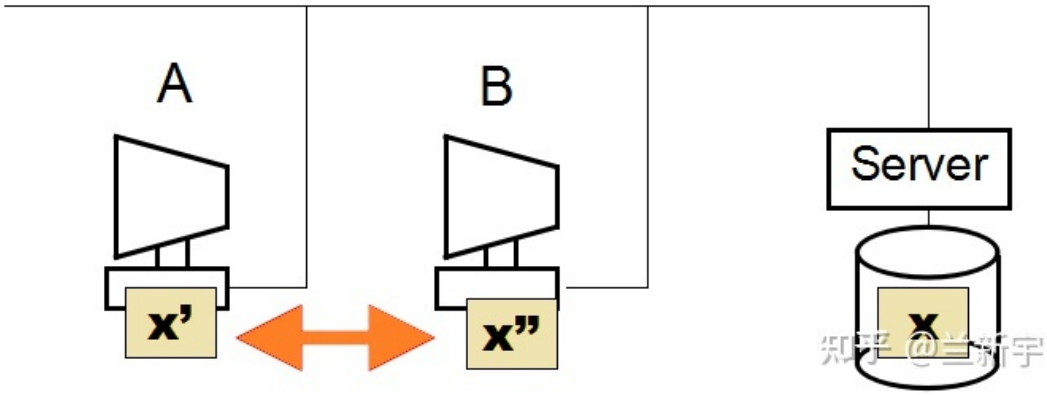

  比如client是通过"mount -t nfs server:/home/share /usr"命令挂载的，那么文件路径中的"usr"在server端就会被转换成"/home/share"。那为什么需要lookup那么多次呢？因为路径中的每一个目录都可能是一个不同的挂载点。

  对于没有记忆（stateless）的server来说，面对open请求，不能只给client返回一个file descriptor，而是需要包含能定位文件的信息，包括文件所在的文件系统ID、文件的inode编号等。这些信息被封装在**file handle**里，交给client，之后client发送针对这个文件的读写请求时，再把对应的file handle拿给server，server才能知道怎么去磁盘上找到这个文件。

  ### 读写操作

访问本地文件系统，通常打开时需要检查一下权限，但读写时就不再需要了，但由于stateless的NFS没有维护状态，所以理论上每次读写都得进行**权限检查**。从这个角度来说，server端其实并没有真正地实现open()对应的操作。

为了提高读写性能，client可以将一些文件block的数据缓存在本地。但有cache就不可避免地带来**cache一致性**的问题，当client读取cache中的数据时，还是得向server发送查询该文件attribute（也就是meta data）的请求，以询问该block数据是否有被其他的client更改。



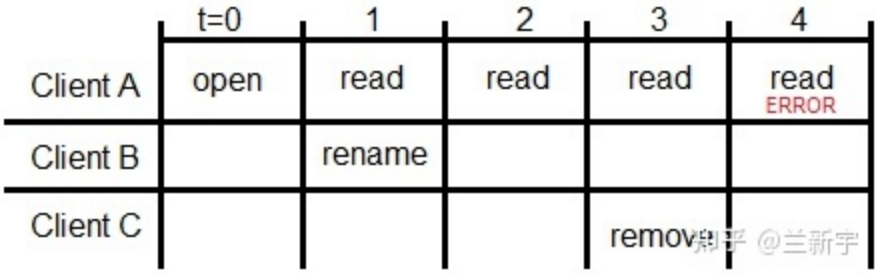
这种方案虽然原理简单，但会导致server收到大量的meta data查询请求，而且可能大部分的时间里，这个block的数据并没有被更改。一个改进的做法是：client将文件的meta data也缓存起来，每隔一段时间更新一次（比如3秒）。当然，这可能造成不一致的问题（读到stale的数据），但算是性能和cache一致性之间的一种平衡吧。

删除操作

根据Unix的规则，只要文件还在open状态（有object指向它），那么执行remove操作只会删除文件路径并将文件标记为"deleted"，文件的内容并不会从磁盘移除，已经打开文件的进程依然可以继续访问文件（参考这篇文章）。

但NFS有所不同，假设Client B对共享文件进行了rename/move操作，Client A依然可以访问这个文件，因为访问基于的是inode而不

是文件名，但如果Client C在server端把文件remove了，之后A的读取操作就会失败。



究其原因，同样因为stateless。作为server，它并不知道文件被remove的时候，有没有client端的进程还在"open"这个文件，client端的close()操作在server端也没有对应的实现。

那Client A的"read error"又是怎么来的（怎么被server判断的）呢？文件被Client C移除后，对应的inode也将随之被server回收，并且可能很快就被reuse以创建新的文件，但Client A不知道啊，它还继续发送read这个inode的请求。

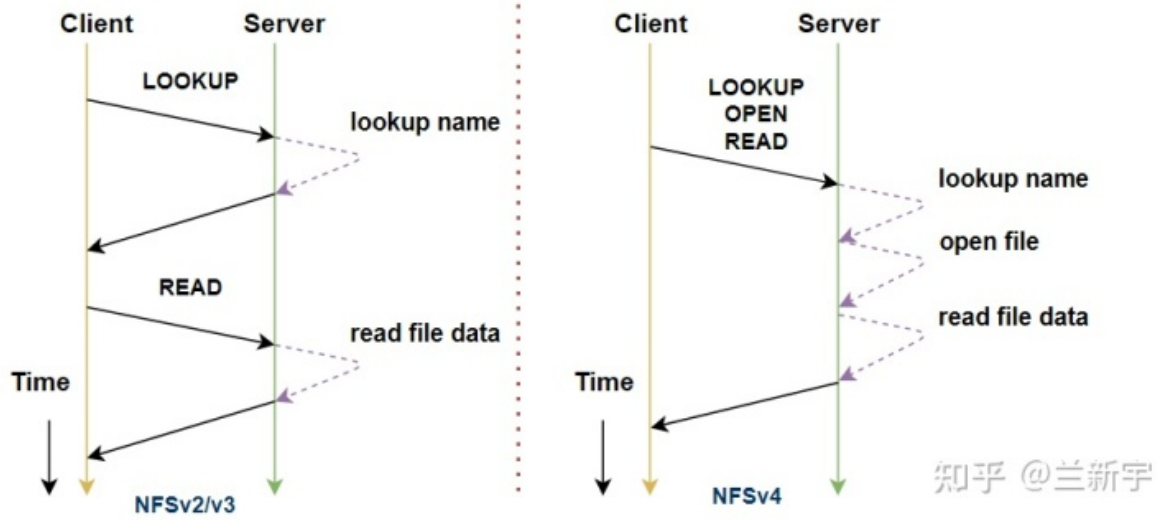
这时的inode已经不是Client A想要的那个inode了，看来啊，得对inode加上一些额外的信息，比如一个单调递增的数字，或者是文件创建时的timestamp，反正能作为唯一性的区别就好。NFS选择的是递增数字的方式，称为"generation number"[注1]，包在前面讲的file handle里，在之后client和server的交互中传递，过期的handle意味着文件已不复存在。

- **不断演进的NFS**

到了新世纪的2000年，NFS演进到了v4版本，其中一个重大的变化就是：它投靠了stateful阵营。除了解决上述stateless模式存在的一些固有问题，stateful模式还让client端的close()操作有了意义：通知server丢弃关于这个文件的状态信息。此外，NFSv4还在诸多方面进行了优化。

在client向server发送的请求中，很多操作常常是一前一后地伴随出现，比如client对NFS中的文件执行"ls -l"命令，"READDIR"之后往

往紧接着"GETATTR"。对此NFSv3采用的方法是在协议中增加一个新的"READDIRPLUS"操作，到了v4则更加灵活，一个RPC message可以传递多个NFS操作，操作可以合并发送，在避免协议改动的情况下，增加了协议的可扩展性。



- **联想**

在我们自己的代码设计中，要不要记录状态（和过去发生关系），由哪个模块来记录状态，通常也是重要的考量之一。比如要计时，就得在过去采样一个timestamp，现在再采样一个timestamp，这就形成了对过去时刻状态的依赖。如果能构造出不依赖的场景，则往往可以减少很多判断，使得逻辑更加简洁清晰，也有利于减少异常造成的影响（这次错误不影响下次）。

注1：这主要是依靠"inode"结构体中的"i_version"来实现的，但考虑到重启后"i_version"可能回滚，所以还加上了ctime，以确保唯一性。

> We could use i_version alone as the change attribute. However, i_version can go backwards after a reboot. On its own that doesn't necessarily cause a problem, but if i_version goes backwards and then is incremented again it could reuse a value that was previously used before boot, and a client who queried the two values might incorrectly assume nothing changed. By using both

> ctime and the i_version counter we guarantee that as long as time doesn't go backwards we never reuse an old value.

参考：

- DESIGN AND IMPLEMENTATION OF THE SUN NETWORK FILESYSTEM