RPC 的目标是隐藏通信细节，使远程过程调用看起来和本地调用一样。除了一些例外，如无法处理全局变量，以及使用复制/ 恢复而不是变参调用传递指针参数而引起的细微差别等，我们和目标已经很接近了。实际上，只要客户与服务器都正常运转，RPC 将工作良好。但是一旦出现了错误，就会产生一些问题。这时，远程过程调用与本地调用的一些差别是不容易掩盖的。本节中我们将讨论以下可能发生的一些错误及其解决方法。

为使我们的讨论结构化，让我们来区分 RPC 系统中可能出现的五类问题：

- 客户无法定位服务器。
- 客户发给服务器的请求消息丢失。
- 服务器发给客户的应答消息丢失。
- 服务器在收到请求后崩溃。
- 客户机在发送请求后崩溃。

以上五类问题均各不相同，因此需要不同的解决方法。

- **客户无法定位服务器**

客户可能无法定位合适的服务器。例如，服务器可能已关闭。此外，假设客户进程已用某一版本的客户存根编译好了，但其二进制代码已经有很长时间没有使用了。在这期间，服务器可能已产生一个新版本的接口，且产生了新的存根程序并投入使用。这样，当客户进程运行时，binder 就无法将客户进程与服务器进程相匹配，只能报告出错。尽管这种机制可以防止客户与参数不匹配或被认为不匹配的服务器通信，但仍然存在着如何处理错误的问题。在 UNIX 系统中，有一个全局变量 errno，它的不同返回值说明了错误类型。在这样的系统中，加入一条新的错误类型“无法定位服务器”是很简单的。但问题是这种解决办法并不总有效的。比如一个 SUM 过程。如果参数是 7 与-8 的话，其返回值-1 不再是一个出错标志而是一个合法的值。一种可能的解决办法是在出错时产生一个异常。在一些语言中(例如，Ada)，程序员可以编写在发生特定错误（如除 0 错误）时激活的特殊过程。在 C 语言中，信号处理程序的目的就在于此。换句话说，我们可以定义新的信号类型 SIGNOSERVER，让它像其它信号一样处理错误。这种方法也有缺陷。首先，并不是所有语言都有异常和信号处理程序。PASCAL 就是这样一个例子。另外，编写异常和信号处理程序破坏了透明性。

- **客户请求消息丢失**

这是最容易解决的一个问题：客户内核在发送请求时启动计时器。如果在计时器时满之前无应答或无确认消息返回，内核重发消息。如果请求消息确实丢失了，服务器是无法区分收到的请求是原来的还是重发的，而一切运行良好。当然，如果消息被多次重发而得不到应答，客户会放弃请求并认为服务器已经关闭。这就是前面所说的“无法定位服务器”的情况。

- **服务器应答消息丢失**

应答丢失的处理要困难一些。一个显而易见的办法是根据计时器重传。如果在合理的时间内未收到应答，那么就重发请求。这种方法存在的问题是客户内核不知道为什么没有收到

应答。是请求丢失还是应答丢失，或是服务器速度太慢？对不同原因引起的错误，其处理的方法也不相同。

特别是有些操作是幂等的能多次安全地重复执行而不产生危害。例如，从某文件读出1024 个字节的请求就不产生负面影响，它可以按需要多次执行而不发生错误。但是，假设有一个发到银行服务器的请求，它要求一个帐户调拨一百万美元到另一个账户。如果请求到达且被执行，但应答丢失，客户就不会知道这种情况并将在计时器到时后重发请求。服务器会认为这是一个新的请求，因而再次执行该请求。那么将有二百万美元调拨到那个帐户上。如果应答重复丢失，后果将是非常可怕的。显然，这是一个非幂等请求。

要解决这个问题，一种方法是将每个请求构造成幂等的。然而，有些请求（如汇钱）事实上是非幂等的，因此需要其它的措施。比如客户内核给要发送的请求消息分配一个序号。服务器内核保留那些最近来自每个客户的请求序号。这样服务器内核可以区别第一次发送的请求和重发的请求，从而排除了两次执行某个请求的可能性。一个附加的保护措施是在消息头上增加一位以区分是原来的还是重发的消息，能提醒服务器在看到不是原发消息时谨慎处理。

- **服务器崩溃**

服务器崩溃也与可幂等次执行有关，但现在不能使用给消息加序号的方法来解决。图1(a)是一服务器进程正常工作处理一个事件的顺序；一个来自客户的请求到达，执行该请求，服务器进程返回一个应答。如图 1(b)所示是一个客户请求到达后，服务器进程执行，在返回应答之前服务器崩溃。最后，如图 1(c)所示，一个请求到达后，在执行前服务器崩溃。

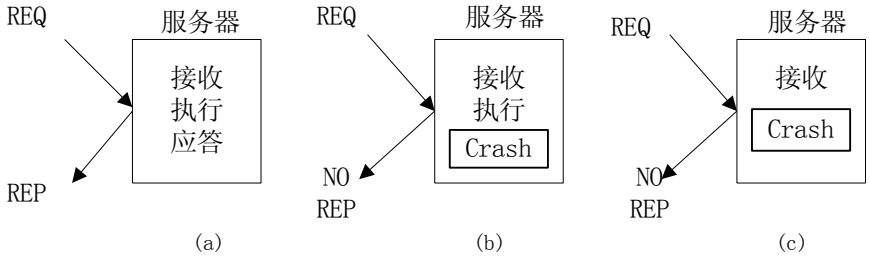

图 1（a）正常状态；（b）执行后崩溃；（c）执行前崩溃

图 1 中(b)、(c)的处理方法是不相同的。在(b)中，系统不得不向客户报告失败（例如，引起一个异常中断），而在(c)中只需重发请求。问题是客户的内核不能区分这两种情况。它只知道计时器到时。

有三种方法可以解决这个问题。第一种方法是等待服务器重新启动（或与另一个新服务器捆绑），然后重发请求。这种方法要求不断重试直至应答消息到来并传给客户。这种技术称作至少一次语义（at least once semantics），它保证 RPC 至少要执行一次，但也有可能执行多次。第二种方法是立即放弃并报告失败。这是最多一次语义(at most once semantics)，它确保了 RPC 最多执行一次，但可能没有执行。第三种方法不做任何保证。当服务器崩溃时，客户得不到任何帮助和保证。RPC 可以不被执行或执行相当多次。这种方法最大的优点就是

易实现。

这三种方法都不是成熟的方法。人们需要的是精确的执行一次的语义(exactly once semantics)，但通常它是不容易实现的。设想一个远程操作包括打印一些文本，它把文件调入打印机的缓冲区，然后在某个控制寄存器中设置一位后准备开始打印。 可能在置位的那一微秒的前后服务器崩溃而无法打印。客户由于无法知道崩溃发生在置位前还是置位后，因此也就无法有针对性地进行恢复处理。

简而言之，服务器崩溃的可能性在很大程度上改变了 RPC 的性质。单处理机系统和多处理机系统发生服务器崩溃的情况是截然不同的。在单处理机系统中，服务器的崩溃往往意味着客户的崩溃。所以恢复既不可能也没必要。而在分布式系统中则可以采取一些措施来处理这种情况。

- **客户机崩溃**

如果客户已发出请求但在应答到来之前崩溃了，这时会发生什么？此时已经激活了服务器中的相应计算，但没有客户在等待结果。这样的计算称为孤儿（orphan）。孤儿会导致一系列的问题。起码它浪费了 CPU 周期， 同时又锁住了文件或其他宝贵的资源。此外，如果客户重新启动并再次调用了这个 RPC，客户会很快得到那个孤儿的返回值，这将引起调用结果的混淆。

为解决孤儿问题，Nelson（1981）提出了四种解决方法。方法一，在客户存根发送一个 RPC 前，在日志文件中记下要执行操作的信息。 该日志文件保存在不受崩溃影响的磁盘和其他媒介上。当客户重新启动后，系统检查日志文件，并准确地清除孤儿。这种方法称为（extermination)。根绝方法的缺陷是对每个 RPC 都要进行磁盘记录，极大增加了系统开销。此外，这种方法也可能不起作用，因为孤儿还可以执行 RPC， 这样又生成了一层或更多层的子 RPC，这些子 RPC 很难找到和清除。最后，由于网络或许是分段的，如果网关失败，即使找到这些孤儿也无法清除。总之，这种方法是没有前途的。方法二，称为再生（reincarnation）。这种方法不必做磁盘记录。该方法将时间划分成顺序编号的纪元（epochs）。当一客户重新启动时，它向所有机器广播一个新纪元的开始。广播后，所有远程计算被终止。当然，如果网络是分段的，有些孤儿还会遗留下来。但是当这些孤儿的应答返回时，返回的消息上带有它们过时的纪元号。这些应答还是容易识别和清除的。方法三，是对第二种方法的改进，称作温和再生(gentle reincarnation)。当接到某客户开始新纪元的广播后，每台机器检查自己是否有远程计算，若有则试图去找到该远程计算的调用者。若没有找到该计算的调用者，则终止该计算。方法四，这种方法叫过期(expiration)。每一个 RPC 执行时事先分给一个标准时间段 T。当 T 到期而调用未完成时就必须再申请一个 T。这样做确实麻烦。另一方面，如果客户崩溃，服务器在客户重新启动前等候了一个 T 后，所有孤儿都被清除。由于 RPC 有各种不同的请求，如何选择 T 的合适值呢？

实际上，这些方法都不尽人意。终止一个孤儿可能会造成不可预见的后果。例如，假设一个孤儿正锁住一个或多个文件或数据记录等，如果突然清除该孤儿，那么这些资源可能会一直

处于被占用的状态。另外，一个孤儿可能已在某些远程的进程调用队列中等待，期望将来能调用其他进程。这样，即使去除了这个孤儿也不能去除孤儿遗留下的轨迹。