# 容错

分布式系统区别于单机系统的一个特征是部分故障的概念:系统的一部分正在故障，而其余部分继续运行，而且结果似乎是正确的。分布式系统设计的一个重要目标是构建一个能够自动从部分故障中恢复而不会严重影响整体性能的系统。特别是，当发生故障时，在进行维修时，系统应继续以可接受的方式运行。换句话说，分布式系统应该是容错的。

在本章中，我们将进一步研究实现容错的技术。在提供了一些一般背景之后，我们将首先通过进程组查看进程弹性。在这种情况下，多个相同的进程协作提供单个逻辑进程，以确保其中一个或多个进程可以在客户端没有注意到的情况下失败。进程组中一个特别困难的地方是在组成员之间就执行客户机请求的操作达成共识。到目前为止，Paxos是一种常用的，但相对复杂的算法，我们通过从头开始构建它来解释。同样，我们仔细审查了在何种情况下能够达成一致意见。

实现容错和可靠的通信是密切相关的。除了可靠的客户机-服务器通信之外，我们还关注可靠的组通信，尤其是原子组播。在后一种情况下，消息被传递给组中的所有非故障进程，或者根本不传递给任何进程。拥有原子多播使容错解决方案的开发变得容易得多。

原子性在许多应用程序中都很重要。在本章中，我们将关注所谓的分布式提交协议，通过该协议，一组进程可以联合提交它们的本地工作，也可以集体中止并返回到以前的系统状态。

最后，我们将研究如何从失败中恢复。特别是，我们要考虑何时以及如何保存分布式系统的状态，以便稍后恢复到该状态。

8.1容错介绍

容错一直是计算机科学研究的热点。在本节中，我们首先介绍与处理故障相关的基本概念，然后讨论故障模型。本文还讨论了冗余是故障处理的关键技术。有关分布式系统容错的更一般信息，请参见，例如[Jalote, 1994;Shooman, 2002]或[Koren和Krishna, 2007]。

**基本概念**

为了理解容错在分布式系统中的作用，我们首先需要更仔细地研究分布式系统容错的实际含义。容错与所谓的**可靠系统**密切相关。可靠性这个术语涵盖了分布式系统的一些有用需求，包括以下内容[Kopetz和Verissimo, 1993]:

•可用性

•可靠性

•安全

•可维护性

**可用性**定义为系统可以立即使用的属性。一般来说，它指的是系统在任何给定时刻正常运行并可以代表用户执行其功能的概率。换句话说，一个高可用性的系统很可能在某个特定的时刻工作。

**可靠性**是指系统能够连续运行而不发生故障的特性。与可用性相反，可靠性是根据时间间隔而不是瞬间来定义的。一个高度可靠的系统很可能在相当长的一段时间内不间断地工作。与可用性相比，这是一个细微但重要的区别。如果一个系统平均每小时宕机一毫秒，看上去是随机的，那么它的可用性超过99.9999%，但仍然是不可靠的。同样，一个系统不会崩溃，但每年8月都会关闭两个星期，它的可靠性很高，但只有96%的可用性。这两个是不一样的。

**安全**是指当系统暂时不能正常运行时，不会发生灾难性事件的情况。例如，许多过程控制系统，例如那些用于控制核电站或将人送入太空的系统，都需要提供高度的安全。如果这样的控制系统发生短时间的失效，后果可能都是灾难性的。过去的许多例子(可能还有更多的例子)表明，构建安全的系统是多么困难。

最后，**可维护性**是指一个失败的系统可以多么容易地修复。一个高度可维护的系统也可能显示出高度的可用性，特别是如果故障可以自动检测和修复。然而，正如我们将在本章后面看到的，从失败中自动恢复说起来容易做起来难。

通常，可靠的系统还需要提供高度的安全性，特别是涉及到完整性等问题时。我们将在下一章讨论安全性。

当一个系统不能兑现它的承诺时，它就会失败。特别是，如果分布式系统的设计目的是为其用户提供许多服务，那么当无法(完全)提供其中一个或多个服务时，系统就会失败。错误是系统状态中可能导致故障的部分。例如，当在网络上传输数据包时，可以预期一些数据包在到达接收端时已经损坏。

错误的原因称为**故障**。显然，找出导致错误的原因是很重要的。例如，错误的或不好的传输介质很容易导致数据包损坏。在这种情况下，相对容易排除故障。然而，在无线网络中，恶劣的天气条件也可能导致传输错误。改变天气来减少或防止错误是有点棘手的。

另一个例子，一个崩溃的程序很明显是一个失败，这可能是因为程序输入了一个包含一个编程错误的代码分支(例如，，程式错误)。这个bug通常是由程序员引起的。换句话说，是程序员的错误导致变成失败的。

建立可靠的系统与控制故障密切相关。正如Avizienis等人[2004]所解释的那样，可以对预防、容忍、消除和预测故障进行区分。就我们的目的而言，最重要的问题是容错，这意味着即使存在故障，系统也可以提供服务。例如，通过对传输包应用纠错码，可以在一定程度上容忍相对较差的传输线路，并降低错误(损坏的包)导致故障的可能性。

故障通常分为瞬态、间歇或永久性故障。**瞬态故障**发生一次，然后消失。如果操作重复，故障就会消失。一只鸟飞过微波发射器的光束可能会导致某些网络上的比特丢失(更不用说一只烤鸟了)。如果传输超时并重试，它可能会在第二次工作。

间歇性故障是从发生开始，然后自动消失，然后重新出现，依此类推。连接器上的松动接触经常会导致间歇性故障。间歇性故障由于诊断困难，造成严重后果。通常，当故障医生出现时，系统可以正常工作。

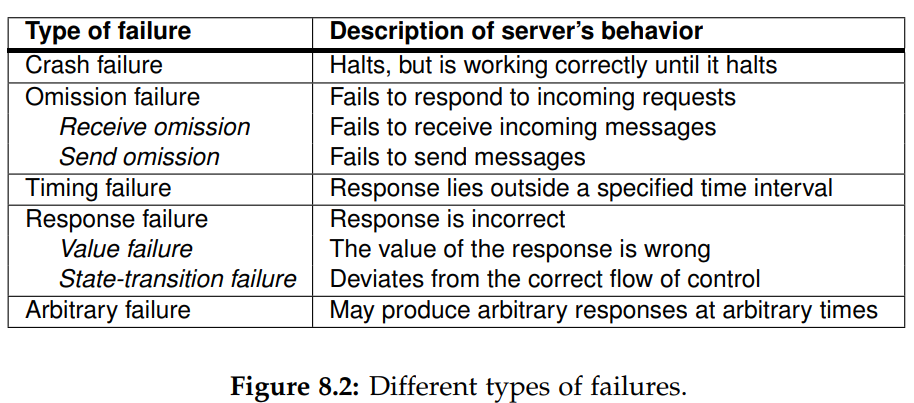
**永久性故障**是指在替换错误组件之前一直存在的故障。烧坏的芯片、软件bug和磁盘头崩溃都是永久性故障的例子。

**故障模型**

一个失败的系统不能充分地提供它所设计的服务。如果我们将分布式系统看作是相互通信的服务器及其客户机的集合，那么没有提供足够的服务就意味着服务器、通信通道或两者都没有完成它们应该做的工作。然而，服务器本身的故障可能并不总是我们要寻找的错误。如果这样的服务器依赖于其他服务器来充分提供其服务，则可能需要在其他地方搜索错误的原因。

这种依赖关系在分布式系统中大量出现。故障磁盘可能会使设计为提供高可用文件系统的文件服务器的工作变得困难。如果这样的文件服务器是分布式数据库的一部分，那么整个数据库的正常工作可能会受到威胁，因为只有部分数据可以访问。

为了更好地了解故障实际上有多严重，开发了几种分类方案。图8.2中显示了一个这样的方案，它基于Cristian[1991]和Hadzilacos和Toueg[1993]描述的方案。



**崩溃失败**发生在服务器过早停止，但在停止之前正常工作的情况下。崩溃失败的一个重要方面是，一旦服务器停止，就再也听不到它发出的任何信息。崩溃失败的一个典型例子是操作系统突然停止，对此只有一个解决方案:重新启动它。许多个人电脑系统经常发生崩溃故障，以至于人们开始认为它们是正常的。因此，将重置按钮从柜子的后面移到前面是有充分理由的。也许有一天它可以被搬到后面去，甚至完全被搬走。

当服务器无法响应请求时，就会发生**遗漏故障。**有几件事可能会出错。在接收-遗漏失败的情况下，服务器可能从一开始就没有收到请求。注意，很可能客户机和服务器之间的连接已经正确建立，但是没有线程监听传入的请求。此外，接收-遗漏故障通常不会影响服务器的当前状态，因为服务器不知道发送给它的任何消息。

同样，当服务器完成其工作，但不知何故未能发送响应时，也会发生**发送遗漏故障**。例如，当发送缓冲区溢出而服务器没有为这种情况做好准备时，可能会发生这种故障。注意，与接收-遗漏失败相反，服务器现在可能处于一种状态，反映它刚刚为客户机完成了一个服务。因此，如果发送响应失败，服务器必须为客户机重新发出之前的请求做好准备。

与通信无关的其他类型的遗漏故障可能是由软件错误引起的，比如无限循环或不正确的内存管理，通过这些错误，服务器被称为“挂起”。

另一类故障与时间有关。当响应位于指定的实时间隔之外时，就会发生**计时故障**。例如，在流媒体视频的情况下，如果没有足够的缓冲空间来容纳所有传入的数据，那么过早地提供数据很容易给接收者带来麻烦。然而，更常见的情况是服务器响应太迟，在这种情况下，会发生性能故障。

一种严重的故障类型是**响应故障**，服务器的响应不正确。可能会发生两种响应失败。在值失败的情况下，服务器只是对请求提供错误的响应（这里指服务器返回值错误）。例如，一个搜索引擎系统地返回与所使用的任何搜索词都不相关的Web页面，但它失败了。

另一种类型的响应失败称为**状态转换失败**。这种故障发生在服务器对传入请求做出意外响应时。例如，如果服务器接收到它无法识别的消息，如果没有采取措施来处理此类消息，则会发生状态转换故障。特别是，有故障的服务器可能错误地采取了它本不应该启动的默认操作。

最严重的是**武断的失败**，也称为**拜占庭式的失败**。实际上，当任意故障发生时，客户端应该做好最坏的准备。特别地，可能会发生这样的情况，服务器正在生成它本不应该生成的输出，但是不能被检测为不正确。Pease等[1980]和Lamport等[1982]首先分析了拜占庭式的故障。下面我们将回到这种失败。

前面提到的许多情况处理的是进程P不再感知来自另一个进程Q的任何操作的情况。然而，P能否感知到Q已经停止了呢？要回答这个问题，我们需要区分两种类型的分布式系统：

在**异步系统**中，不需要对流程执行速度或消息交付时间做任何假设。结果是，当进程P不再感知来自Q的任何操作时，它不能断定Q崩溃了。相反，它可能只是很慢，或者它的消息已经丢失了。

在**同步系统**中，进程执行速度和消息传递时间是有限制的。这也意味着当Q在预期的时候没有显示更多的活动时，进程P可以正确地推断出Q已经崩溃了。

不幸的是，纯同步系统只存在于理论上的可能。另一方面，简单地说每个分布式系统都是异步的也不仅仅是我们在实践中看到的，我们在设计分布式系统时如果假定它们一定是异步的，就会过于悲观。相反，假设分布式系统是**部分同步**会更加现实一些：在大多数情况下，它表现为一个同步系统，但是它表现为异步的时间并没有限制。换句话说，异步行为是一个例外，这意味着我们通常可以使用超时来得出进程确实崩溃的结论，但有时这样的结论是错误的。实际上，这意味着我们必须设计容错的解决方案，能够承受错误地检测进程停止的情况。

在这种情况下，停机故障可以分为以下几类，从最小到最严重(参见Cachin等人[2011])。我们让进程P尝试检测进程Q已经失败。

**故障停止故障**是指可以可靠地检测到的崩溃故障。这可能发生在假设没有错误的通信链接时，以及故障检测过程P可能对Q的响应造成最坏情况下的延迟时。

**故障噪声故障**类似于故障停止故障，只不过P最终只能得出Q崩溃的正确结论。这意味着可能存在一个先验的未知时间，在这个时间里P对Q行为的检测是不可靠的。

在处理**故障静默故障**时，我们假设通信链路没有故障，但是进程P不能区分崩溃故障和遗漏故障。

**故障安全故障**涵盖了按进程Q处理任意故障的情况，但是这些故障是良性的:它们不会造成任何危害。

最后，在处理任意故障时，Q可能以任何可能的方式失败;除了对其他进程的有害行为之外，其他故障是观测不到的。

显然，必须处理故障—任意故障是最糟糕的情况。我们将很快讨论，我们可以设计分布式系统，使它们甚至能够容忍这些类型的故障。

**冗余故障掩蔽**

如果一个系统是容错的，那么它所能做的最好的事情就是试图向其他进程隐藏故障的发生。掩盖故障的关键技术是利用冗余。有三种可能:信息冗余、时间冗余和物理冗余(参见Johnson[1995])。在**信息冗余**的情况下，会添加额外的比特，以便从混乱的比特中恢复。例如，可以将汉明代码添加到传输的数据中，以从传输线上的噪声中恢复。

使用**时间冗余**，执行一个操作，然后，如果需要，再次执行该操作。事务使用这种方法。如果事务中止，可以在不造成任何伤害的情况下重新执行。另一个著名的例子是在缺少预期响应时将请求重新发送到服务器。当故障是瞬态或间歇的时候，时间冗余尤其有用。

有了**物理冗余**，就可以添加额外的设备或过程，使整个系统能够容忍某些组件的丢失或故障。因此，物理冗余可以在硬件或软件中实现。例如，可以将额外的进程添加到系统中，这样，即使少数进程崩溃，系统仍然可以正常运行。换句话说，通过复制过程，可以实现高度的容错。在本章后面，我们将回到这种软件冗余。

8.2进程的弹性

既然已经讨论了容错的基本问题，让我们集中讨论如何在分布式系统中实现容错。我们讨论的第一个主题是防止进程失败，这是通过将流程复制到组来实现的。在接下来的几页中，我们将考虑进程组的一般设计问题，并讨论什么是容错组。此外，我们还将研究如何在流程组中的一个或多个成员不能被信任给出正确答案时，在流程组中达成共识。

**按过程组划分的弹性**

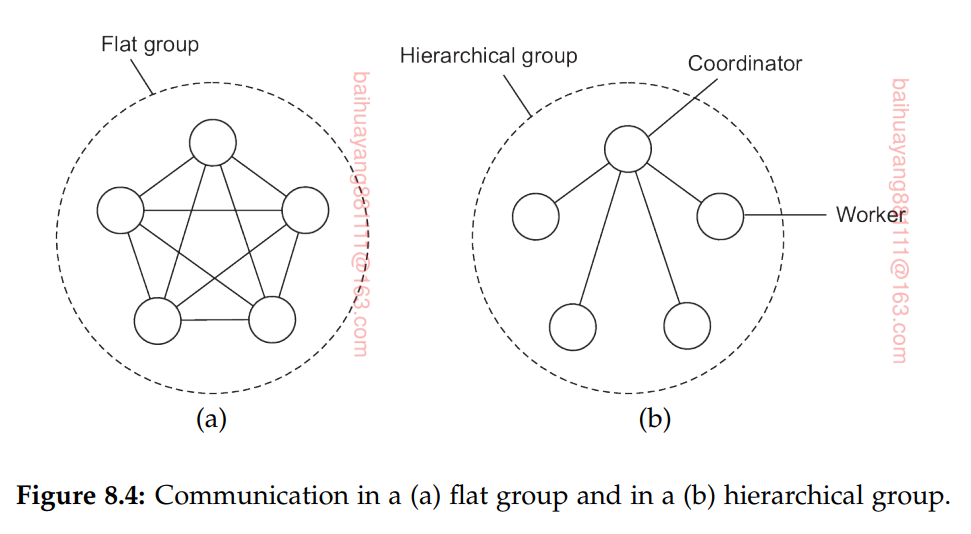
容忍错误过程的关键方法是将几个相同的进程组织成一个组。所有组都具有的关键属性是，当消息发送到组本身时，组的所有成员都会接收它。这样，如果一个组中的一个进程失败了，希望其他进程可以接管它[Guerraoui and Schiper, 1997]。

进程组可能是动态的。可以新创建进程组，也可以删除旧的进程组。一个进程可以在系统运行期间加入一个组，也可以离开一个组。进程可以同时是多个组的成员。因此，需要一些机制来管理组和组成员。

引入组的目的是允许进程将其他进程的集合作为单个抽象来处理。因此，进程P可以向组Q = {Q1，…， Qn}的服务器，而不需要知道他们是谁，有多少，或他们在哪里，这可能会改变调用。对于P，Q就是单一的逻辑进程。

**Group organization**

不同组之间的一个重要区别与他们的内部结构有关。在某些组中，所有进程都是相等的。没有一个独特的领导者，所有的决定都是集体做出的。在其他组中，存在某种层次结构。例如，一个进程是协调器，其他所有进程都是工作者。在这个模型中，当工作请求被生成时，不管是由外部客户机还是由某个工作人员生成，它都会被发送到协调器。然后协调器决定哪个worker最适合执行它，并将其转发到那里。当然，更复杂的层次结构也是可能的。这些通信模式如图8.4所示。



每个组织都有自己的优点和缺点。平面组是对称的，没有单点故障。如果其中一个进程崩溃，组就会变得更小，但是还可以继续。缺点是做决定比较复杂。例如，要决定任何事情，通常必须进行投票，这会导致一些延迟和开销。

层次组具有相反的属性。失去协调器会使整个组陷入停顿，但是只要它还在运行，就可以在不影响其他人的情况下做出决策。在实践中，当分层组中的协调器失败时，需要接管它的角色，并选出其中一个工作人员作为新的协调器。

**会员管理**

当存在组通信时，需要一些方法来创建和删除组，以及允许进程加入和离开组。一种可能的方法是拥有一个组服务器，所有这些请求都可以发送到该服务器。然后，组服务器可以维护所有组及其确切成员的完整数据库。这种方法简单、有效，而且很容易实现。不幸的是，它与所有集中技术都有一个主要缺点:单点故障。如果组服务器崩溃，则组管理将不复存在。可能大多数或所有的组都必须从头开始重建，可能终止正在进行的任何工作。

相反的方法是以分布式的方式管理组成员。例如，如果(可靠的)多播可用，外部人员可以向所有组成员发送消息，宣布希望加入组。

理想情况下，要离开一个组，成员只需向每个人发送一个告别消息。在容错上下文中，假定故障-停止故障语义通常是不合适的。问题是，没有礼貌的声明说进程会像自愿离开时那样崩溃。问题是，无法在进程因崩溃而给出一个声明，从而使它退出组。其他成员必须通过注意到崩溃的成员不再响应任何东西来发现这一点。一旦确定崩溃的成员确实停机(而不仅仅是速度变慢)，就可以从组中删除它。

另一个棘手的问题是离开和加入必须与发送的数据消息同步。换句话说，从进程加入组的那一刻开始，它必须接收发送到该组的所有消息。类似地，进程一旦离开组，就不能从组接收更多消息，其他成员也不能从该进程接收更多消息。确保将join或leave集成到正确位置的消息流的一种方法是将该操作转换为发送到整个组的消息序列。

与组成员关系有关的最后一个问题是，如果太多进程宕机，以至于组无法正常工作，该怎么办。需要一些协议来重建组。总是有一些过程必须主动开始，但是如果同时开始两三个会发生什么呢。协议必须能够承受这种压力。同样，可能需要通过协调，例如，领导人选举算法。

**故障屏蔽和复制**

进程组是构建容错系统的解决方案的一部分。特别是，拥有一组相同的进程允许我们屏蔽该组中的一个或多个错误进程。换句话说，我们可以复制进程并将它们组织成一个组，用一个(容错的)组替换单个(脆弱的)进程。如前一章所讨论的，有两种方法可以实现这种复制:通过基于基本的协议，或者通过复制写协议。

在容错情况下，基于主备份的复制通常以主备份协议的形式出现。在本例中，一组进程以分层方式组织，其中主进程协调所有写操作。实际上，主服务器是固定的，但如果需要，它的角色可以由备份之一接管。实际上，当主服务器崩溃时，备份会执行一些选择算法来选择一个新的主服务器。

复制写协议以活动复制的形式使用，也可以通过基于quorum的协议使用。这些解决方案对应于将一组相同的流程组织成一个平面组。其主要优点是，这些组没有以分布式协调为代价的单点故障。

使用进程组来容忍错误的一个重要问题是需要多少复制。为了简化我们的讨论，让我们只考虑复制写系统。如果一个系统能够在k个部件的故障中存活下来，并且仍然能够满足它的规格，那么这个系统就被称为k容错系统。如果组件(比如进程)安静的失败了，那么拥有k + 1个组件就足以提供k容错能力。如果k个进程简单地停止，那么另一个的答案就可以用了。

另一方面，如果进程出现任意故障，在出现故障时继续运行，并发出错误或随机响应，则至少需要2k + 1个进程才能实现k容错（why？？？）。在最坏的情况下，k个失败的进程可能意外地(甚至有意地)生成相同的响应。然而，剩下的k + 1也会得到相同的答案，所以客户或投票人可以相信大多数人。

现在假设在一个容错k组中有一个进程失败。该组织作为一个整体，仍然符合它的规范，即它能够容忍多达k个成员的失败(其中一个成员刚刚失败)。但是如果超过k个成员失败了会发生什么？在这种情况下，所有的赌注都打了水落石出，无论该组织做什么，其结果(如果有的话)都不可信。另一种看待这个问题的方法是，进程组在模仿单个健壮进程的行为时失败了。

**故障系统与崩溃故障的共识**

正如前面提到的，在客户机和服务器方面，我们采用了一个模型，在这个模型中，一个潜在的非常大的客户机集合现在向一组进程发送命令，这些进程作为一个单独的、高度健壮的进程共同工作。为了使这一设想成为现实，我们需要做一个重要的假设:

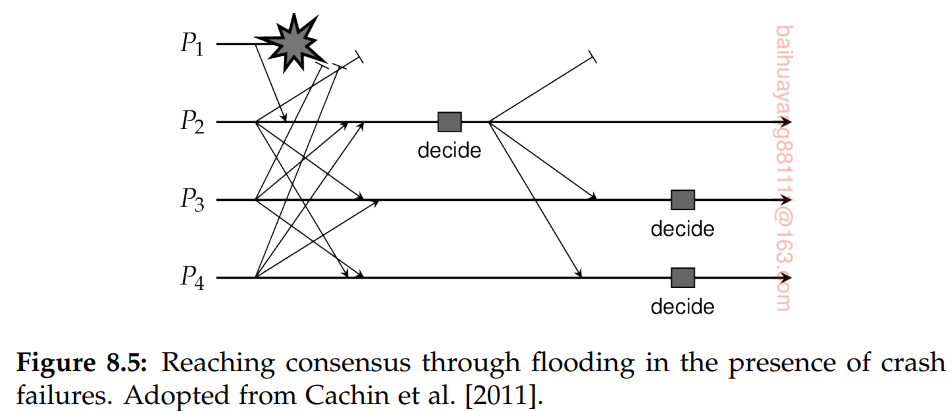
*在容错进程组中，每个非故障进程与其他非故障进程以相同的顺序执行相同的命令。*

在形式上，这意味着组成员需要就执行哪个命令达成一致。如果失败不能发生，达成共识是容易的。例如，我们可以使用第6.2节中描述的Lamport的完全有序多播。或者，为了保持简单，使用一个集中的排序器，它向需要执行的每个命令分发一个序列号，也可以完成这项工作。不幸的是，生活中并非没有失败，在一组过程中，在更现实的假设下达成共识是很困难的。

为了说明问题，让我们假设有一组进程P = {P1，…， Pn}在故障-停止故障语义下运行。换句话说，我们假设在组成员之间可以可靠地检测到崩溃故障。通常，客户端与组成员联系，请求组成员执行命令。每个组成员都维护一个建议命令列表:其中一些命令是直接从客户端接收的;其他它从它的小组成员收到的。们可以通过以下方法达成共识，这一方法来自Cachin等人[2011]，被称为洪泛共识。

从概念上讲，该算法是循环运行的。在每一轮中，Pi都向P中的其他进程发送它迄今为止看到的建议命令列表。在一轮结束时，每个进程将所有接收到的建议命令合并到一个新列表中，然后，如果可能，它将从这个列表中确定要执行的命令。重要的是要认识到选择算法对所有进程都是相同的。换句话说，如果所有进程都具有完全相同的列表，那么它们都将选择要执行的相同命令(并从列表中删除该命令)。

不难看出，只要进程没有失败，这种方法就可以工作。当进程Pi在轮r中检测到进程Pk崩溃时，问题就开始了。为了具体说明这一点，假设我们有一个由四个进程组成的进程组{P1，…， P4}，并且P1在轮r中崩溃。同样，假设P2在崩溃前从P1接收到建议的命令列表，但是P3和P4没有(换句话说，P1在有机会将列表发送到P3和P4之前就崩溃了)。图8.5概述了这种情况。



假设所有进程在轮r开始时都知道谁是组成员，P2准备在接收到其他成员的相应列表时决定执行哪个命令：到目前为止，它已经提出了所有的命令。P3和P4则不然。例如，P3可能检测到P1崩溃，但它不知道P2或P4是否已经收到P1的列表。从P3的角度来看，如果有另一个进程确实接收了P1提出的命令，那么该进程可能会做出与自身不同的决策。因此，P3所能做的最好的决定就是推迟到下一轮。本例中的P4也是如此。当流程收到来自每个非故障流程的消息时，它将决定进入下一轮。这假定每个进程都能够可靠地检测到另一个进程的崩溃，否则它将无法确定谁是没有错误的进程。

因为进程P2接收了所有命令，所以它确实可以做出决策，并随后将该决策广播给其他进程。后，在下一轮r + 1中，进程P3和P4也将能够做出决策:它们将决定执行P2选择的相同命令。要理解为什么这个算法是正确的，重要的是要认识到，只有当一个进程检测到另一个进程失败时，它才会在没有做出决定的情况下进入下一轮。最后，这意味着在最坏的情况下，最多只剩下一个没有错误的进程，这个进程可以简单地决定要执行什么命令。再次注意，我们假设可靠的故障检测。

但是，当进程P2发送给P3的决策丢失时会发生什么?在这种情况下，P3仍然无法做出决定。更糟的是，我们需要确保它做出的决策与P2和P4相同。如果P2没有崩溃，我们可以假设重新传输它的决策可以挽救局面。如果P2确实崩溃了，P4也会检测到这一点，然后P4会重新广播它的决定。同时，P3已经进入下一轮，在收到P4的决策后，将终止算法的执行。

**举例：Paxos参考wikipedia（-p463）**

基于泛洪的共识算法是不太现实的，因为它只是依赖于一个故障停止故障模型的事实。更实际的方法是假设一个故障噪声故障模型，其中一个进程最终将可靠地检测到另一个进程已经崩溃。在下面的文章中，我们将描述一个被广泛采用的协商一致算法的简化版本，**Paxos**。它最初是1989年Leslie Lamport以技术报告的形式发表的，但是过了大约10年，才有人认为通过常规的科学渠道传播它可能不是一个坏主意[Lamport, 1998]。最初的出版物不容易理解，例如其他旨在解释它的出版物[兰普森，1996;Prisco等，1997;Lamport, 2001;van Renesse and Altinbuken, 2015]。

**Essential Paxos**

Paxos运行的假设相当**薄弱**：

•分布式系统是部分同步的(事实上，它甚至可能是异步的)。

•进程之间的通信可能不可靠，这意味着消息可能丢失、复制或重新排序。

•被破坏的消息可以被检测出来(因此随后被忽略)。

•所有操作都是确定性的:一旦执行开始，就知道它将做什么。

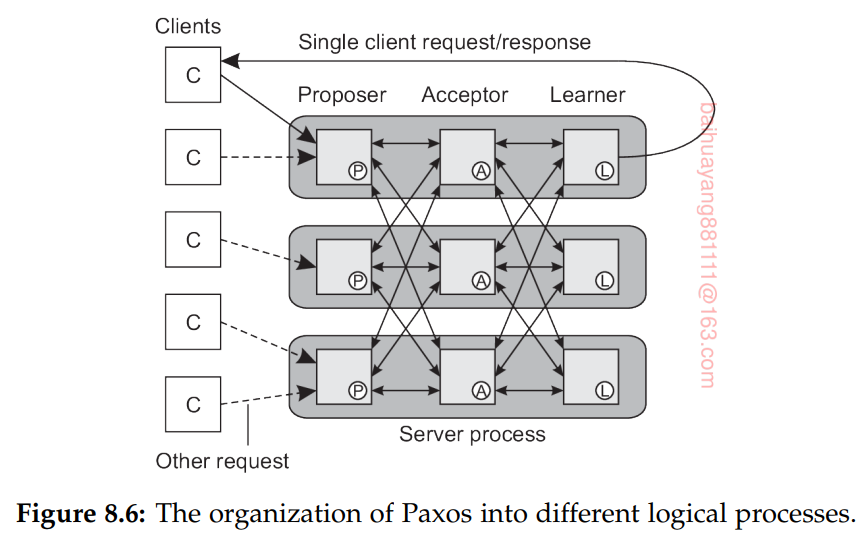
•进程可能会出现崩溃故障，但不会出现任意故障，也不会串通一气。

总的来说，对于许多实际的分布式系统来说，这些都是现实的假设。

我们大致遵循Lamport[2001]、Kirsch和Amir[2008]给出的解释。该算法是由不同类型的逻辑过程组成的网络。首先，有些客户机请求执行特定的操作。在服务器端，每个客户机由一个**提议者**表示，该提议者试图接受客户机的请求。通常，只有一个提议者被指定为**领导者**，并推动协议达成共识。

我们需要确定的是提议的操作被**接受方**接受。如果大多数接受人接受同一提案，则该提案被称为被选中。然而，选择什么仍然需要学习。为此，我们将有许多**学习进程**，一旦它被大多数接受方告知，每一个学习进程都将执行选定的建议。

需要注意的是，一个提议者、接受者和学习者组成了一个物理过程，运行在客户机与之通信的一台机器上，如图8.6所示。因此，我们假设，例如，如果一个提议者崩溃，那么它所属的物理进程将崩溃。通过复制此服务器，我们的目标是在出现崩溃故障时获得容错能力。



基本模型是，主要的提议者接收来自客户的请求，一次一个。非领导提议者将任何客户请求转发给领导。主要的提议者将它的提议发送给所有的接受者，告诉每个人接受请求的操作。每个接受者随后将广播一条学习消息。如果学习者从大多数接受者那里收到了相同的学习信息，那么他对于要执行哪个操作就达成了共识，并将执行它。

至少有两个具体问题需要进一步注意。首先，服务器不仅需要就执行哪个操作达成一致意见，还需要确保每个服务器都实际执行该操作。换句话说，我们如何确定大多数没有故障的服务器将执行该操作？从本质上说，只有一种方法可以解决这个问题:让学习信息被重新传输。然而，要使此工作正常，接受器必须记录其决策(反过来需要一种清除日志的机制)。因为我们假设全局有序的提案时间戳(稍后将进行解释)可以很容易地检测到缺失的消息，而且所有学习者总是以相同的顺序执行已接受的操作。

作为一般规则，承载主要提议者的服务器也将在其请求的操作执行完毕时通知客户机。如果另一个进程接管了主导，那么它也将处理对客户机的响应。

这就引出了第二个重要问题:一个失败的领导者。如果一个领导者的失败能够被可靠地检测出来，那么生活就会变得容易，之后就会选出一个新的领导者，之后，恢复中的领导者会立即注意到周围的世界已经发生了变化。不幸的是，生活并不那么容易。Paxos被设计用来容忍那些仍然认为自己处于领先地位的提议者。其结果是，提案可以由不同的提案人(每个人都认为自己是领导者)同时发出。因此，我们需要确保这些建议能够彼此区别开来，以便确保接受方只处理来自当前领导人的建议。

请注意，依赖一个主要的提议者意味着任何Paxos的实际实现都需要伴随一个领导人选举算法。原则上，该算法可以独立于Paxos运行，但通常是Paxos的一部分。为了区分并发提案和不同的提案，每个提案p都有一个惟一关联的(逻辑的)时间戳ts(p)。

为了区分不同提案者的不同提案，每个提案p都有一个惟一关联的(逻辑的)时间戳ts(p)。

如何实现惟一性留给实现，但是我们将很快描述一些细节。令oper(p)表示与提案p相关的操作。诀窍是允许多个建议被接受，但是每个被接受的建议都具有相同的关联操作。这可以通过确保如果选择了提案p，那么任何具有较高时间戳的提案也将具有相同的关联操作来实现。换句话说，我们需要它满足：



当然，要选择p，它需要被接受。这意味着，当我们保证如果选择p，那么任何接收方接受的稿时间戳的提案，都保证有对P相同的操作。但是，这是不够的，因为假设在某个时刻，一个提议者简单地向一个之前没有收到任何提议的接受者a发送一个新的提议p0(具有迄今为止最高的时间戳)。请注意，根据我们对消息丢失和多个提议者的假设，这可能确实会发生，每个提议者都认为自己处于领先地位。如无其他建议，A只接受p0。为了防止这种情况的发生，我们必须对此作出保证。

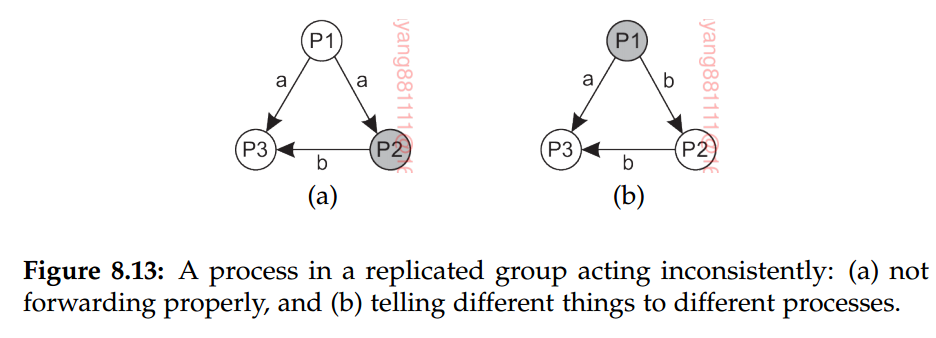
*如果选择了提案p，那么由提案者发出的任何时间戳较高的提案都具有与p相同的关联操作*

在解释下面的Paxos算法时，我们确实会看到提议者可能需要采用来自接受器的操作来支持自己的操作。这将在一个主要的提议者失败后发生，但其提议的操作已经得到大多数接受人的同意。

（-463/596）都是讲paxos,略过，直接阅读参考文献。

**在有故障的系统中与任意故障达成一致**

到目前为止，我们假设副本只受到崩溃故障的影响，在这种情况下，一个进程组需要由2k + 1服务器组成，以使k个崩溃的成员存活下来。在这些情况下，一个重要的假设是一个进程不会与另一个进程串通，或者更具体地说，在它向其他进程发送的消息中是一致的。图8.13所示的情况不应该发生。在第一种情况下，我们看到进程P2转发的值或操作与它应该转发的值或操作不同。对于Paxos，这可能意味着主操作告诉备份没有接受操作o，而是传播一个不同的操作o'。在第二种情况下，P1向不同的进程讲述不同的内容，例如让一个leader向一些备份发送操作o，同时向其他进程发送操作o'。同样，我们注意到这并不一定是恶意行为，而仅仅是疏忽或委托失败。



在本节中，我们将研究如何在容错过程组中达成共识，其中k个成员可以在假定任意故障的情况下失败。特别地，我们将表明，在这些失败假设下，我们至少需要3k + 1个成员达成共识。考虑一个由n个成员组成的进程组，其中一个成员被指定为主进程P，其余的n - 1被指定为备份B1，…Bn−1。我们做出如下假设:

客户机向主节点发送一个值v∈{T, F}，其中v表示真或假。

消息可能丢失，但是可以检测到。

如果没有检测到这些信息，消息就不能被破坏(因此随后被忽略)。

消息的接收者可以可靠地检测到它的发送者。

为了达成所谓的**拜占庭协议**，我们需要满足以下两项要求:

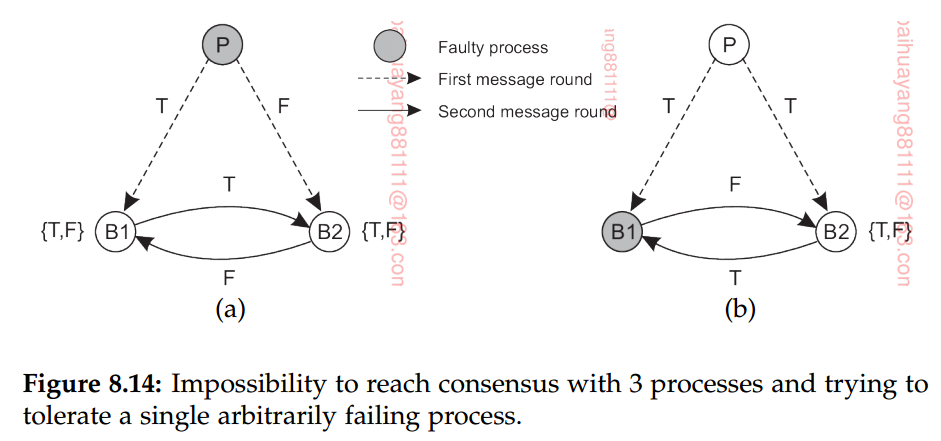
BA1:每个非故障备份进程都存储相同的值

BA2:如果主进程没有错误，那么每个没有错误的备份进程都会准确地存储主进程发送的内容。

注意，如果主服务器出错，备份机器会存储相同的但是错误的值。此外，应该清楚的是，如果主节点没有故障，满足BA2意味着满足BA1。

**为什么有3k个进程是不够的**

为了了解为什么只有3k个进程不足以达成共识，让我们考虑一下我们希望容忍单个进程失败的情况，即k = 1。考虑图8.14，它本质上是图8.13的扩展。



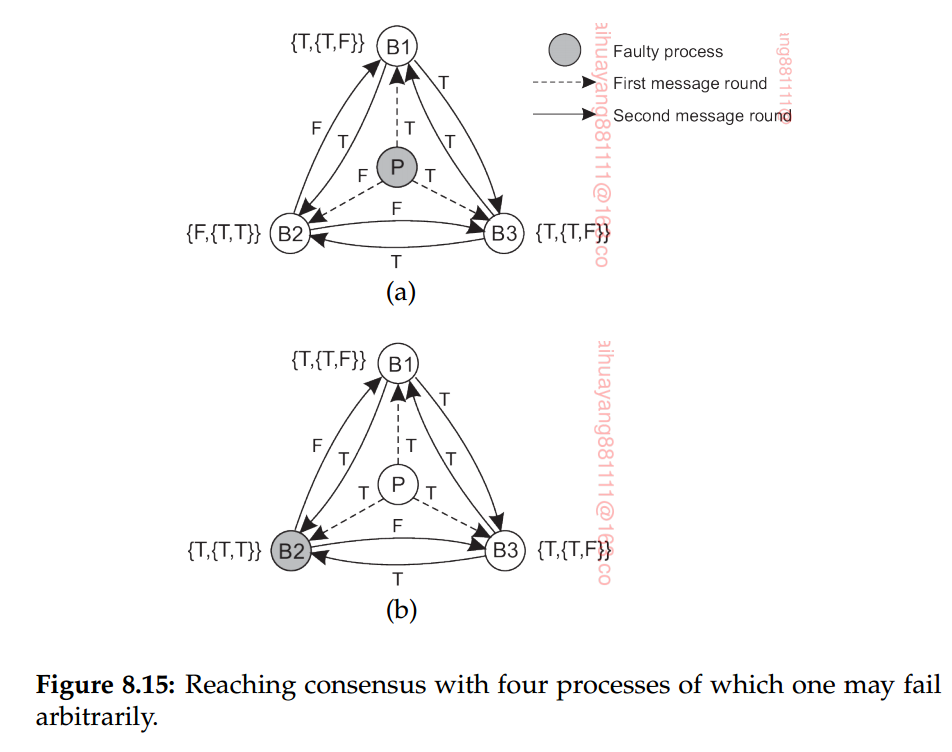
在图8.14(a)中，我们看到错误的主P分别向备份B1和B2发送了两个不同的值。为了达成共识，两个备份进程都将接收到的值转发给另一个进程，从而导致第二轮消息交换。到那时，B1和B2各自收到了一组值{T, F}，从中不可能得出结论。同样，当错误的值被提出时，我们也无法达成共识。在图8.14(b)中，主P和备份B2操作正确，但是B1不正确。它不是将值T转发给进程B2，而是发送错误的值F。结果是，B2现在将看到一组值{T, F}，它不能从中得出任何结论。换句话说，P和B2无法达成共识。更具体地说，B2无法决定存储什么，因此我们无法满足需求BA2。

**为什么有3k + 1个进程就足够了**

现在让我们关注一组3k + 1进程的情况。我们的目标是表明，我们可以建立一个解决方案，其中k组成员可能遭受任意故障，而其余的非故障进程仍将达成共识。同样，我们首先关注n = 4 k = 1的情况。考虑图8.15，它显示了一个主P和三个备份进程B1、B2和B3的情况。

在图8.15(a)中，我们大致描述了主P出错并向其备份提供不一致信息的情况。在我们的解决方案中，进程将把接收到的内容转发给其他进程。在第一轮中，P分别向B1发送T，向B2发送F，向B3发送T。然后，每个备份将它们所拥有的发送给其他备份。只有主故障，这意味着在两轮备份之后，每个备份都将收到一组值{T, T, F}，这意味着它们可以就值T达成一致。

当我们考虑其中一个备份失败的情况时，我们得到了图8.15(b)所示的情况。假设(非错误的)主节点将T发送给所有备份，但是B2是错误的。其中B1和B3在第二轮中将T发送给其他备份，B2最坏的情况可能是发送F，如图所示。尽管如此，B1和B3会得到相同的结论，即P发出了T，从而满足了我们之前所说的BA2的要求。



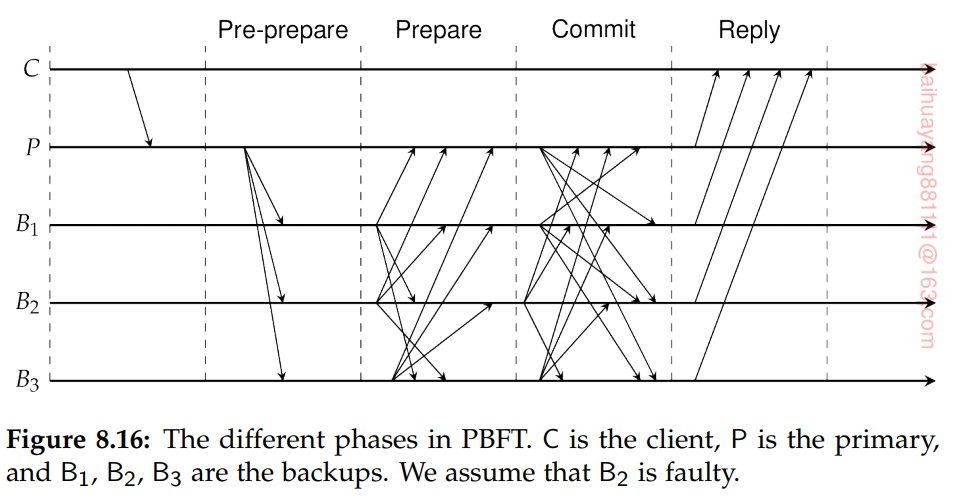
**实例:拜占庭式的实际容错能力**

长期以来，拜占庭式的容错或多或少都是一个奇特的话题，部分原因是将安全性、活动性和实际性能结合起来是很难实现的。大约是在2000年，Barbara Liskov和她当时的学生Miguel Castro设法提出了一个实用的协议实现，用于复制能够处理任意故障的服务器。让我们简要地看一下他们的解决方案，这个解决方案被称为**实用拜占庭容错**(或简称**PBFT**)。

与Paxos一样，PBFT只对其环境做了一些假设。它对复制服务器的行为不做任何假设:假定有故障的服务器可以表现出任意的行为。同样，消息可能丢失、延迟和接收顺序错误。但是，假定消息的发送者是可识别的。在这些假设下，只要不超过k个服务器失败，就可以证明PBFT是安全的，这意味着客户机将始终收到正确的答案。如果我们还可以假设同步，这意味着消息延迟和响应时间是有限的，那么它还提供了活动性。在实践中，这意味着PBFT假设一个部分同步模型，其中没有界限的限延迟是一个异常，例如由攻击引起的。

为了理解这个算法，让我们后退一步，回顾一下我们到目前为止讨论的关于建立一个k容错进程组的内容。一个基本的问题是，这样一个组的行为就像一个单一的中央服务器。因此，在只有崩溃失败的假设下，当客户机发送请求时，它应该期望得到k + 1相同的答案。如果服务器崩溃，返回的响应会更少，但它们是相同的。

我们需要解决的第一个问题是，并发请求都是以相同的顺序处理的。为此，PBFT采用了主备份模型，共有3k + 1个副本服务器。为了简单起见，让我们暂时假设主程序没有错误。在这种情况下，客户机C向主服务器发送执行操作o的请求(图8.16中表示为P)。主服务器具有当前非故障副本服务器集合的概念，用视图v表示，它只是一个数字。主节点将时间戳t分配给o，然后对其进行递增，以用于后续请求。随后，主服务器向备份发送一条(签名的)预准备(t、v、o)消息。



如果备份在v中，且之前从未接受过时间戳为t的操作，则(非故障)备份将接受预准备。接受预准备的每个备份都向其他备份(包括主备份)发送一条(再次签名)消息prepare(t, v, o)。一个关键的观察结果是，当一个没有故障的复制服务器S接收到2k个消息prepare(t, v, o)时，这些消息都与主服务器接收到的pre-prepare消息S相匹配时，在没有故障的服务器之间，对于先执行哪个操作的顺序存在共识。

（这段看不懂到p473）

**实现容错的一些局限性**

将复制的过程组织到一个组中有助于提高容错性。然而，现在应该清楚的是，这是要付出代价的，即潜在的性能损失。在目前讨论的解决方案中，容错组中的进程可能需要交换大量消息才能做出决定。拜占庭协议协议很好地说明了进程可能是多么紧密耦合。我想到的问题是，实现特定形式的容错，比如能够承受任意故障，是否总是可能的。

**On reaching consensus**

正如我们所提到的，如果客户可以通过投票机制做出决定，那么我们可以容忍2k + 1进程中的k个进程在结果上出错。然而，我们所做的假设是，进程不会联合起来产生错误的结果。

一般来说，如果我们要求过程组达成共识，事情就会变得更加复杂，而这在许多情况下是需要的。达成协商一致意见有三个条件。

•进程产生相同的输出值

•每个输出值必须是有效的

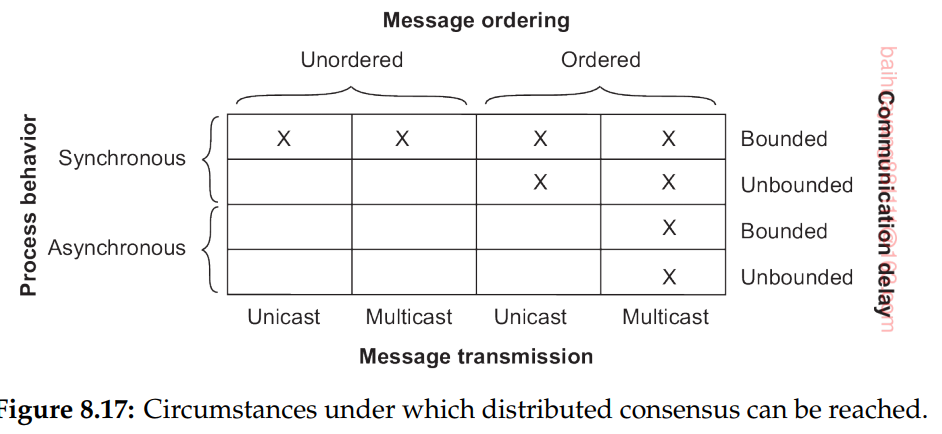
•每个进程最终都必须提供输出

需要达成一致意见的一些例子包括选择一个协调器、决定是否提交事务以及在工作人员之间分配任务。当沟通和进程都很完美时，达成共识往往很简单，但如果不是这样，就会出现问题。

分布式协商一致算法的一般目标是使所有的非故障过程在某些问题上达成一致，并在有限的步骤内建立一致。对于底层系统的不同假设需要不同的解决方案，即使存在解决方案，这一事实也使问题变得复杂。Turek和Shasha[1992]区分了以下情况：

1. 同步和异步系统。在某种程度上重新表述我们的描述，当且仅当已知进程以锁步模式运行时，系统是同步的。形式上，这意味着应该有某个常数c≥1，这样，如果任何进程已经执行了c + 1步，那么其他每个进程都至少执行了1步。
2. 通信延迟是否有界。延迟是有界的，当且仅当我们知道每条消息都以全局的和预先确定的最大时间交付时。
3. 消息交付是否有序(实时)。换句话说，我们将来自不同发送方的消息按照它们在实际全局时间中发送的顺序交付的情况与我们没有此类保证的情况区分开来。
4. 消息传输通过单播或多播完成。

事实证明，只有在图8.17所示的情况下才有可能达成协商一致意见。在所有其他情况下，都可以表明不存在解决方案。注意，在实践中，大多数分布式系统都假定进程的行为是异步的，消息传输是单播的，通信延迟是无界的。因此，我们需要使用有序(可靠)的消息传递，比如TCP提供的消息传递。同样，在实际情况中，我们假设同步行为是默认的，但是也要考虑到可能存在无界延迟。图8.17说明了当进程可能失败时，分布式协商一致的重要性。



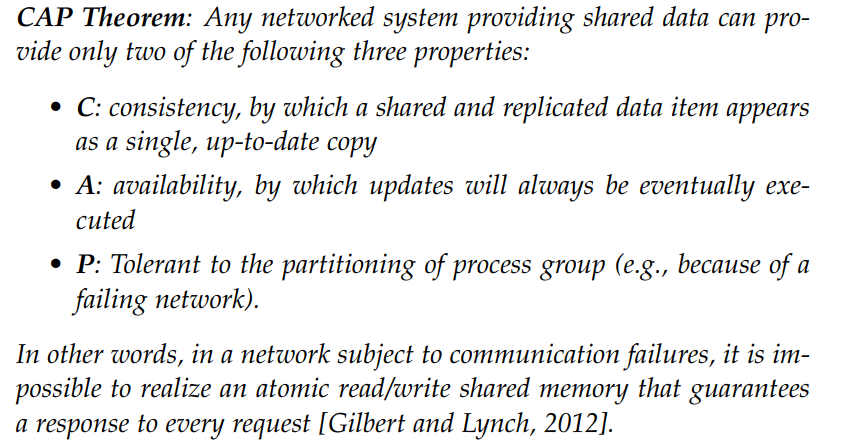
达成共识可能是不可能的。Fischer等人[1985]证明，如果不能保证在已知的有限时间内传递消息，即使有一个进程出错(尽管该进程无声地失败)，也不可能达成一致意见。这类系统的问题是，任意慢的进程与崩溃的进程是无法区分的。，你分不清死者和生者。

**一致性、可用性和分区**

与能否达成协商一致意见的条件密切相关的是能够达成一致意见的时间。在本例中，一致性意味着当我们有一个进程组，客户端向该组发送请求时，返回给该客户端的响应是正确的。 我们正在处理一种**安全属性**:一种声称不会发生坏事的属性。对于我们的目的，我们考虑的操作类型是那些似乎由一个集中式服务器以明确定义的顺序执行的操作。现在，我们更清楚了:这些操作是由进程组执行的，以承受k组成员的失败。

我们引入过程组来改进容错性，更具体地说，是为了提高可用性。可用性通常是一个**活性属性：**最终，一些好事会发生。就我们的进程组而言，我们的目标是最终获得对客户机发出的每个请求的(正确的)响应。对于作为分布式系统一部分的服务来说，在响应上保持一致同时又具有高可用性并不是不合理的要求。不幸的是，我们可能要求太多了。

在实际情况中，我们认为组中的进程确实可以彼此通信的基本假设可能是错误的。信息可能丢失;组可能由于网络故障而被分区。（分区是啥意思？在不同的网段？）。2000年，Eric Brewer提出了一个重要的定理，后来Gilbert和Lynch[2002]证明了这个定理的正确性。



这就是**CAP理论**。正如Brewer[2012]所解释的，理解该定理的一种方法是考虑两个进程由于网络故障而无法通信。允许一个进程接受更新会导致不一致，因此我们只能使用属性{C P}。如果要在两个进程不能通信时提供一致性的假象，那么其中一个进程就必须假装不可用，这意味着只有{A, P}。然而，只有当两个进程能够通信时，才有可能进行通信保持一致性和高可用性，这意味着我们只有{C,A} （个人理解，首先满足P的情况下，代表网络可以出错，即两个进程之间无法发现对方出错，那么要么将出错进程停用 即满足C 而抛弃A， 要么继续使用出错进程 即满足A 抛弃C。 在不满足P的情况下， 由于进程之间可以发现出错，可以同时满足C和A）。

还要注意与达成协商一致意见的关系;事实上，如果共识要求证明进程产生相同的输出，那么提供一致性就比较弱。 这也意味着，如果不可能实现上限，那么达成共识也是不可能的。

CAP定理是关于在安全性和活性之间达成平衡的，基于这样一个观察，即在一个本质上不可靠的系统中，无法同时获得安全性和活性。实际的分布式系统本质上是不可靠的。布鲁尔和他的同事观察到的是，在实际的分布式系统中，尽管无法到达另一个进程，但是必须选择继续前进。换句话说，当一个分区通过高延迟表现出来时，我们需要做一些事情。

当似乎正在进行分区时，底线是继续进行(容忍有利于一致性或可用性的分区)，同时启动一个可以减轻潜在不一致性影响的恢复过程。确切地决定如何进行依赖于应用程序:在许多情况下，数据库中有重复的键可以很容易地修复(这意味着我们应该容忍不一致)，而重复的大笔资金转移可能不会修复(这意味着我们应该决定容忍较低的可用性)。有人可能会说，CAP定理从本质上推动分布式系统的设计者从理论解决方案转向工程解决方案。感兴趣的读者可以参考[Brewer, 2012]了解如何做出这样的举动。

**故障检测**从我们目前的讨论中可以清楚地看出，为了适当地掩盖失败，我们通常还需要检测它们。故障检测是分布式系统容错的基础之一。归根结底，对于一组进程来说，非故障成员应该能够决定谁仍然是成员，谁不是。换句话说，我们需要能够检测到某个成员何时失败。

当涉及到检测进程故障时，基本上只有两种机制。 两个进程都主动发送“您还活着吗?”相互发送消息(显然他们希望得到答案)，或者被动地等待来自不同进程的消息。后一种方法只有在能够保证有足够的通信时才有意义。

关于故障检测器有大量的理论工作要做。归根结底，超时机制用于检查进程是否失败。 如果一个进程P探测另一个进程Q以查看是否失败，那么如果Q在一段时间内没有响应，则P怀疑Q已经崩溃。

在实际设置中，使用探测和超时存在一些问题。例如，由于网络不可靠，简单地说进程失败是因为它没有返回探测消息的答案可能是错误的。换句话说，很容易产生误报。假如因为误报让一个完全正常的进程从我们的成员列表中删除，那么很明显是我们做错了什么。另一个严重的问题是，暂停只是简单的原油。正如Birman[2012]所注意到的那样，除了缺少对单个消息的响应之外，几乎没有任何构建适当的故障检测子系统的工作。在查看行业部署的分布式系统时，这种说法更加明显。

在设计故障检测子系统时，需要考虑各种各样的问题(参见庄等[2005])。例如，故障检测可以通过闲聊进行，其中每个节点定期向其邻居宣布它仍在运行。如前所述，另一种方法是让节点彼此主动探测。

故障检测还可以作为定期与邻居交换信息的副作用进行，就像基于gossip的信息传播一样(我们在第4章中讨论过)。Obduro [Vogels, 2003]基本上也采用了这种方法:进程定期讨论它们的服务可用性。这些信息通过网络逐渐传播。最终，每个进程都将了解其他进程，但更重要的是，将有足够的本地可用信息来决定一个进程是否失败。可用性信息陈旧的成员可能已经失败。

另一个重要的问题是，故障检测子系统在理想情况下应该能够区分网络故障和节点故障。 处理这个问题的一种方法是不让一个节点决定它的一个邻居是否崩溃。相反，当注意到探测消息超时时，节点请求其他邻居查看它们是否能够到达假定的故障节点。当然，积极的信息也可以共享:如果节点仍然活着，则可以将该信息转发给其他相关方(这些方可能正在检测可疑节点的链接失败)。

这就引出了另一个关键问题:当检测到一个成员失败时，应该如何通知其他没有错误的进程。下面是一种简单但有些激进的方法。在FUSE [Dunagan et al.， 2004]中，进程可以加入一个跨广域网络的组。组成员创建一个生成树，用于监视成员故障。成员向邻居发送ping消息。当一个邻居没有响应时，ping节点立即切换到它也不再响应来自其他节点的ping的状态。通过递归，可以看到将单个节点故障快速提升为组故障通知。

8.3可靠的客户端-服务器通信

在许多情况下，分布式系统中的容错主要集中在有故障的进程上。然而，我们还需要考虑通信失败。前面讨论的大多数故障模型同样适用于通信通道。特别是，通信通道可能会出现崩溃、遗漏、定时和任意故障。实际上，在构建可靠的通信通道时，重点是掩盖崩溃和遗漏故障。任意故障可能以重复消息的形式发生，这是由于在计算机网络中，消息可能会被缓冲一段相对较长的时间，并且在原始发送方已经发出重传之后重新注入网络。

**点对点通信**

在许多分布式系统中，可靠的点对点通信是通过使用可靠的传输协议(如TCP)来建立的。TCP通过使用确认和重传来掩盖遗漏故障(以丢失消息的形式出现)。这样的故障对TCP客户机是完全隐藏的。

然而，连接的崩溃故障并没有被掩盖。当(无论出于什么原因)TCP连接突然中断，从而无法通过通道传输更多消息时，可能会发生崩溃故障。在大多数情况下，客户端通过引发异常被告知通道已经崩溃。掩盖此类故障的唯一方法是让分布式系统尝试自动建立新连接，只需重新发送连接请求。基本的假设是，另一方仍然或再次响应此类请求。

**失败时的RPC语义**

现在，让我们在使用远程过程调用(Remote Procedure call, rpc)等高级通信工具时更仔细地研究客户机-服务器通信。RPC的目标是通过使远程过程调用看起来像本地调用来隐藏通信。除了少数例外，到目前为止我们已经相当接近了。实际上，只要客户机和服务器都运行良好，RPC就能很好地完成它的工作。当错误发生时，问题就出现了。因此，本地调用和远程调用之间的差异并不总是容易掩盖的。为了组织我们的讨论，让我们区分RPC系统中可能发生的五种不同类型的故障，如下所示:

1. 客户端无法定位服务器。

2. 从客户机到服务器的请求消息丢失。

3.服务器在收到请求后崩溃。

4. 从服务器到客户机的应答消息丢失。

5. 客户机在发送请求后崩溃。

这些类别中的每一个都有不同的问题，需要不同的解决方案。

**客户端无法定位服务器**

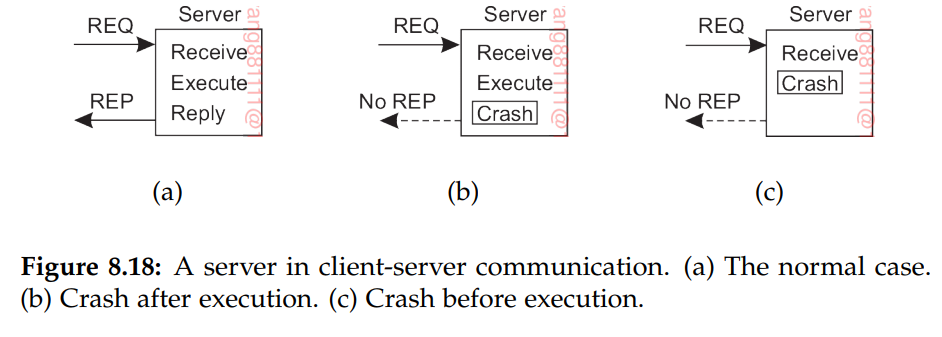
首先，客户端可能无法找到合适的服务器。例如，所有服务器都可能停机。或者，假设客户机是使用客户机存根的特定版本编译的，并且在相当长的一段时间内没有使用二进制文件。 与此同时，服务器不断发展，安装了新版本的接口;生成新的存根并投入使用。当客户机最终运行时，绑定器将无法将其与服务器匹配，并将报告失败。虽然此机制用于保护客户机避免意外地尝试与服务器通信，而服务器可能在需要什么参数或应该做什么方面与客户机不一致，但问题仍然是如何处理此故障。

一种可能的解决方案是让错误引发**异常**。在某些语言中(例如Java)，程序员可以编写特殊的过程，在出现特定错误时调用这些过程，比如除零。在C语言中，可以为此目的使用信号处理程序。换句话说，我们可以定义一个新的信号类型SIGNOSERVER，并允许以与其他信号相同的方式处理它。

这种方法也有缺点。首先，并非每种语言都有例外或信号。另一点是，必须编写异常或信号处理程序会破坏我们一直试图实现的透明性。假设您是一名程序员，您的老板让您编写append过程。你微笑着告诉她，它将在五分钟内被编写、测试和记录下来。然后她提到，您还必须编写一个异常处理程序，以防今天没有这个过程。在这一点上，很难保持远程过程与本地过程没有什么不同的错觉，因为在非分布式系统中，为“无法定位服务器”编写异常处理程序将是一个非常不寻常的请求。透明度就讲到这里。

**失去了请求消息**

列表中的第二项是处理丢失的请求消息。这是最容易处理的:在发送请求时，让操作系统或客户机存根启动计时器。如果计时器在回复或确认返回之前过期，则再次发送消息。如果消息确实丢失了，服务器将无法区分重新传输和原始消息之间的区别，一切都会正常工作。当然，除非丢失了如此多的请求消息，以致客户机放弃并错误地断定服务器已停机，否则我们将回到“无法定位服务器”。如果请求没有丢失，我们唯一需要做的就是让服务器能够检测到它正在处理重传。不幸的是，这样做并不像我们在讨论丢失回复时所解释的那样简单。



**服务器崩溃**

列表上的下一个失败是服务器崩溃。服务器上的正常事件序列如图8.18(a)所示。请求到达，执行，然后发送应答。然后考虑图8.18(b)。与之前一样，请求到达并执行，但是服务器在发送响应之前崩溃。最后，看看图8.18(c)。请求再次到达，但这一次服务器在执行之前就崩溃了。当然，也没有回复。

图8.18中令人讨厌的部分是(b)和(c)的正确处理不同。在(b)中，系统必须向客户端报告故障(例如，引发异常)，而在(c)中，系统只能重新传输请求。问题是客户机的操作系统无法分辨哪个是哪个。它只知道它的计时器已经过期。

关于在这里做什么，存在三种学派[Spector, 1982]。一种方法是等到服务器重新引导(或者让客户机的中间件透明地重新绑定到新服务器)并再次尝试该操作。这样做的目的是不断尝试，直到收到回复，然后将其发送给客户。这种技术称为**至少一次语义**，并保证RPC至少执行一次，但可能更多。

第二种方式是立即放弃，并报告失败。这种方法称为**最多一次语义**，并保证RPC最多一次被执行，但可能根本没有执行。

第三种哲学是什么都不保证。当服务器崩溃时，客户端得不到任何帮助，也得不到关于发生了什么的承诺。RPC可能在任何地方执行过，从零到很多次。该方案的主要优点是易于实现。

这些都不是特别有吸引力。我们想要的是精确的**一次性语义**，但是一般来说，没有办法安排它。假设远程操作包括处理一个文档，例如从LATEX和其他源生成一些PDF文件。当文档被完全处理后，服务器向客户机发送一条完成消息。还假定当客户机发出请求时，它接收到一个确认请求已被发送到服务器的消息。服务器可以遵循两种策略。它既可以在实际告诉文档处理器执行其工作之前发送完成消息，也可以在文档处理之后发送。

假设服务器崩溃并随后恢复。它向所有客户端宣布，它刚刚崩溃，但现在又重新运行起来。问题是客户端不知道它处理文档的请求是否已经实际执行。客户可以遵循四种策略。首先，客户端可以决定永远不重新发出请求，这样做的风险是文档将不会被处理。其次，它可以决定总是重新发出请求，但这可能导致文档被处理两次。 第三，只有在尚未收到请求已被发送到服务器的确认信息时，才可以决定重新发出请求。在这种情况下，客户端依赖于服务器在发送请求之前崩溃的事实。 第四种也是最后一种策略是，只有在收到请求的确认后才重新发出请求。对于服务器有两种策略，对于客户机有四种策略，总共需要考虑8种组合。

不幸的是，没有任何组合是令人满意的:可以表明，对于任何组合，要么请求永远丢失，要么执行两次。

**失去应答消息**

丢失的回复也很难处理。显而易见的解决方案是再次依赖于客户机操作系统设置的计时器。 如果在合理的时间内没有收到回复，请再次发送请求。这个解决方案的问题在于客户并不确定为什么没有答案。是请求或响应丢失了，还是服务器只是速度太慢。这可能会有所不同。

特别是，一些操作可以安全地重复，只要有必要，没有造成任何损害。像请求文件的前1024字节这样的请求没有副作用，并且可以根据需要经常执行，而不会造成任何危害。具有此属性的请求称为**幂等请求**。

现在考虑对银行服务器的一个请求，该请求要求将钱从一个帐户转移到另一个帐户。如果请求到达并被执行，但是应答丢失，客户机将不知道这一点，并将重新传输消息。银行服务器将把这个请求解释为一个新的请求，并将执行它。两倍的钱将被转移。转账不是幂等的。

解决此问题的一种方法是尝试以幂等的方式构造所有请求。然而，在实践中，许多请求(例如，转账)本质上是非幂等的，因此需要其他东西。另一种方法是让客户机为每个请求分配一个序列号。通过让服务器跟踪每个使用它的客户机最近接收到的序列号，服务器可以区分原始请求和重新传输之间的区别，并可以拒绝第二次执行任何请求。但是，服务器仍然必须向客户机发送响应。注意，这种方法确实需要服务器在每个客户机上维护管理。此外，维持的时间还是个未知数。另外一个安全措施是在消息头中添加一个比特，用于区分初始请求和重新传输(其思想是，执行初始请求总是安全的;重新传输可能需要更多的关注)。

**客户端崩溃**

失败列表上的最后一项是客户机崩溃。如果客户机向服务器发送一个请求来做一些工作，然后在服务器响应之前崩溃，会发生什么情况。此时计算是活动的，没有父进程在等待结果。 这种不需要的计算称为孤立(计算)。

孤立计算可能会导致各种各样的问题，干扰系统的正常运行。作为最低限度，它们浪费了处理能力。它们还可以锁定文件或以其他方式占用有价值的资源。最后，如果客户机重新引导并再次执行RPC，但是孤儿的响应随后立即返回，可能会导致混淆。

我们能对孤立计算做些什么呢？提出了四种解决办法。首先，在客户机存根发送RPC消息之前，它生成一个日志条目，告诉它将要做什么。日志保存在磁盘或其他媒介上，以避免崩溃。重新引导之后，检查日志，并显式地终止孤立。这种解决方案称为孤立灭绝。

这种方案的缺点是为每个RPC编写磁盘记录的开销非常大。此外，它甚至可能不起作用，因为孤儿本身也可能执行rpc，从而产生很难或不可能找到的孤儿或后代。最后，网络可能被分区，例如，由于网关失败，即使能够找到它们，也不可能杀死它们。总而言之，这不是一个有希望的方法。

第二个解决方案，称为转世，所有这些问题都可以在不需要写磁盘记录的情况下得到解决。它的工作方式是将时间划分为按顺序编号的时代。当客户机重新启动时，它向所有机器广播一条消息，宣布新纪元的开始。当这样的广播出现时，所有远程计算都被终止。当然，如果网络被分割，一些孤立可能会存活下来。然而，幸运的是，当他们报告回来时，他们的回复将包含一个过时的纪元号，这使得他们很容易被发现。

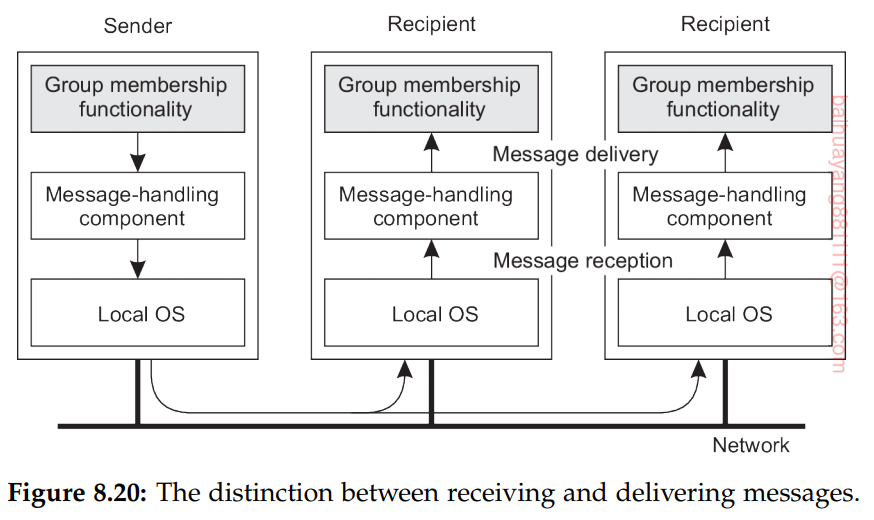
第三种解决方案是对这一想法的一种变体，但不那么严厉。这叫做**温柔的轮回**。当时代广播出现时，每台机器都检查是否有远程计算在本地运行，如果有，则尽力找到它们的所有者。只有当所有者无法找到任何位置时，计算才会终止。

在第四个解决方案(称为过期)中，每个RPC都有一个标准时间T来完成这项工作。如果它不能完成，它必须显式地请求另一个量子。当然，这是相当麻烦的。另一方面，如果在崩溃之后客户端在重启之前等待一个时间T，那么所有的孤儿都肯定会消失。这里要解决的问题是在不同需求的rpc面前选择一个合理的T值。

在实践中，所有这些方法都是粗糙和不可取的。更糟的是，杀害孤儿可能会产生无法预料的后果。例如，假设孤儿获得了一个或多个文件或数据库记录的锁。如果孤儿突然被杀，这些锁可能会永远保留。此外，孤立可能已经在各种远程队列中创建了条目，以便在将来某个时候启动其他进程，因此即使杀死孤立也可能无法删除它的所有痕迹。可以想象，它甚至可能重新开始，带来无法预料的后果。

8.4可靠的群通信

考虑到复制的过程弹性有多重要，因此可靠的多播服务也很重要就不足为奇了。这样的服务保证将消息传递给进程组中的所有成员。不幸的是，可靠的多播结果出人意料地棘手。在本节中，我们将进一步研究可靠地向进程组交付消息所涉及的问题。让我们首先定义什么是可靠的组通信。直观地说，它意味着发送给进程组的消息应该传递给该组的每个成员。如果将处理消息的逻辑与组成员的核心功能分离开来，就可以方便地区分接收消息和交付消息，如图8.20所示。消息由消息处理组件接收，消息处理组件将消息传递给包含组成员核心功能的组件。非正式地，进程P接收到的消息也将由P传递。

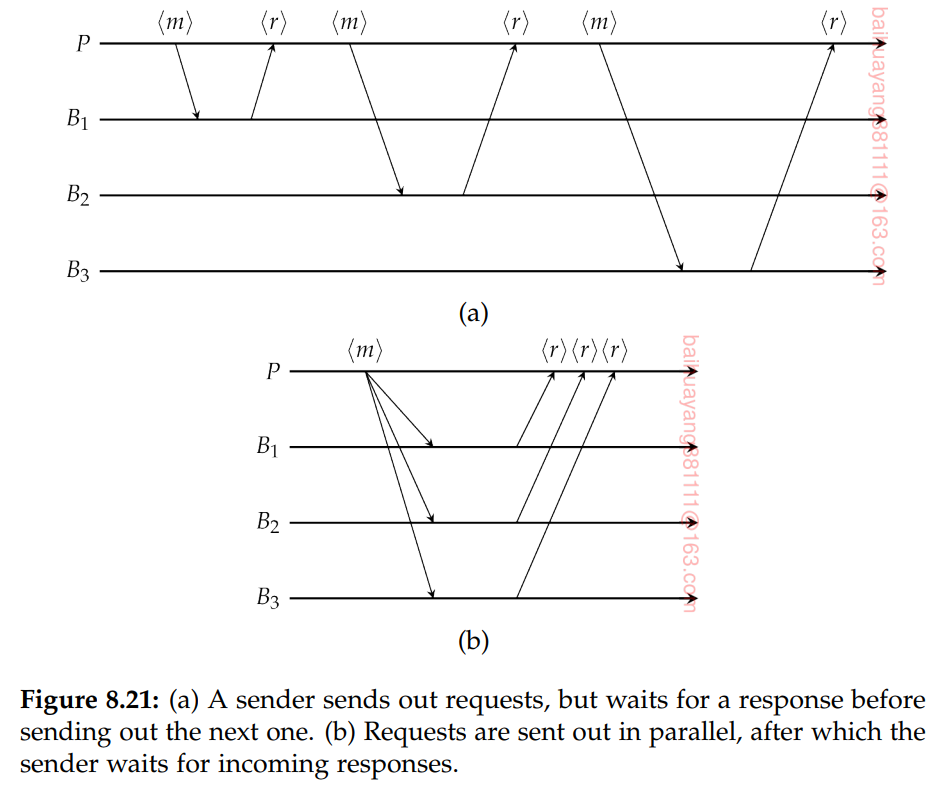


例如，确保来自相同发送方的消息按照发送时的相同顺序交付，通常由messagehandling组件负责。同样，提供可靠的消息传递是一个可以而且应该与组成员的核心功能分离的特性，通常由消息处理组件实现(如果不是由底层操作系统实现)。

有了接收和传递消息之间的这种分离，我们就可以更精确地了解可靠的组通信意味着什么。让我们区分存在错误进程时的可靠通信和假定进程正确运行时的可靠通信。在第一种情况下，当可以确保消息被所有非故障组成员接收并随后交付时，组通信被认为是可靠的。

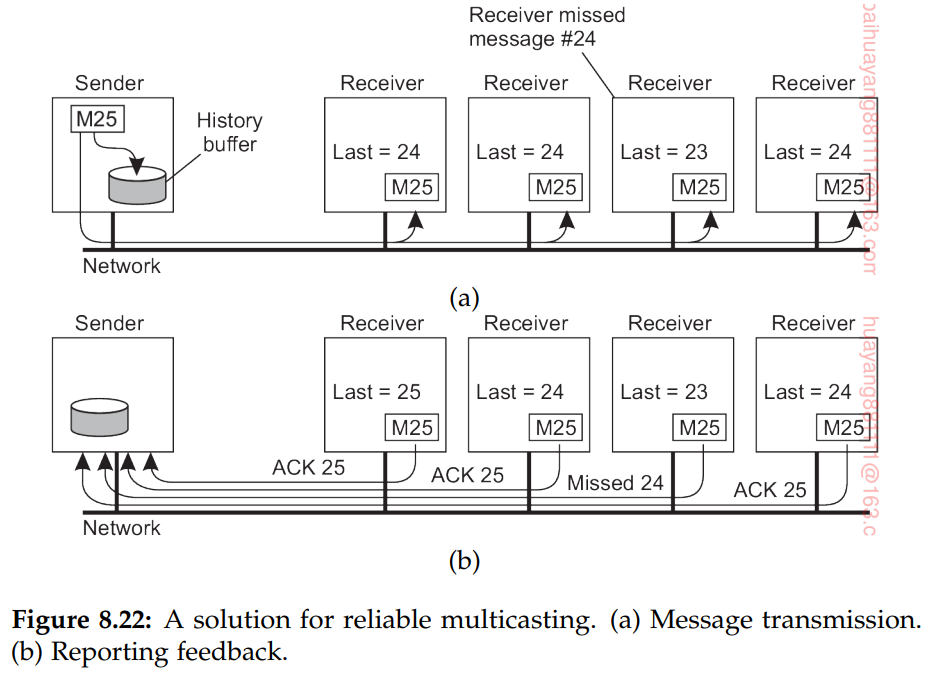
这里棘手的部分是，在传递消息之前，应该就团队的实际情况达成一致。如果发送方希望G组的每个成员都发送一条消息，但是无论出于什么原因，在发送时我们实际上有另一个G' != G组，我们应该问问自己是否可以发送该消息。

如果我们能忽视关于群体成员的共识，情况就会变得简单。特别地，让我们首先假设一个发送进程有一个目标收件人列表。在这种情况下，它可以简单地部署可靠的传输层协议，比如TCP，然后一个接一个地向每个接收者发送消息。如果接收进程失败，稍后当进程恢复时可能重新发送该消息，或者完全忽略该消息(例如，因为发送方离开了组)。如果期望组成员发送响应，即使它只是一个确认，也可以通过将发送请求和接收响应分开来加速通信，如图8.21中的消息序列图所示。



大多数传输层提供可靠的点对点通道;它们很少向一组进程提供可靠的通信。它们提供的最佳功能是让进程彼此建立点对点连接，以便与之通信。当进程组相对较小时，这种建立可靠性的方法是一种直接而实用的解决方案。另一方面，我们通常可以假设底层通信系统确实提供不可靠的多播，这意味着多播消息可能会部分丢失，并由一些(但不是所有)预期的接收者传递。

图8.22显示了一个实现可靠组通信的简单解决方案。发送进程为其多播的每个消息分配一个序列号，并将消息本地存储在历史缓冲区中。假设发送方知道接收方，则发送方只需将消息保存在其历史缓冲区中，直到每个接收方返回确认。当接收到序列号大于s的消息时，接收方可以怀疑丢失了序列号为s的消息m。在这种情况下，它向发送方返回一个否定的确认，请求重新传输m。



有各种各样的设计需要权衡。例如，为了减少返回给发送方的消息数量，确认可以与其他消息一起使用。此外，可以使用点对点通信将消息转发到每个请求进程，或者使用发送到所有进程的单个多播消息来完成消息的重传。Popescu等人[2007]讨论了可靠组播的一般问题。发布/订阅系统上下文中可靠的多播的调查和概述，这也是相关的[Esposito et al.， 2013]。

**原子多播**

现在让我们回到在出现进程故障时需要实现可靠的多播的情况。特别是，在分布式系统中经常需要的是保证消息要么传递给所有组成员，要么完全不传递。这也被称为**原子组播**问题。

要了解原子性为何如此重要，请考虑在分布式系统上作为应用程序构建的复制数据库。分布式系统提供可靠的多播设备。特别是，它允许构造消息可以可靠地发送到的进程组。因此，复制的数据库被构造为一组进程，每个副本对应一个进程。更新操作总是多播到所有副本，然后在本地执行。因此，我们假设正在使用activereplication协议。

为了保持简单，假设客户机与一个副本P联系，并请求它执行更新。复制通过将更新多播给其他组成员来实现这一点。不幸的是，在组播完成之前，P崩溃了，使组中的其他成员陷入困境:一些组成员将收到更新请求;而另一些则不会。如果已经接收到请求的成员将其交付给数据库，那么显然我们将拥有一个不一致的复制数据库。有些副本已经处理了更新，有些则没有。这种情况需要避免，我们应该让更新交付给所有没有错误的成员，或者完全不交付。前者反映P在完成多播后崩溃，而后者表示P在请求更新之前就崩溃了。

这两种情况都很好，并且与客户机与允许崩溃的单个服务器通信的情况相对应。如果一些组成员执行更新，而其他成员不执行更新，则会影响分发的透明性，但更糟的是，客户端将不知道如何处理这种情况。

**虚拟同步**

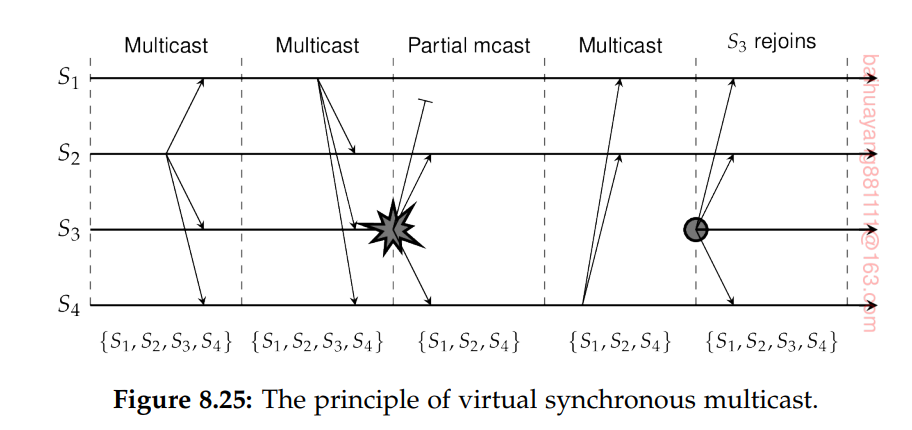
在存在进程故障的情况下，可以根据进程组和组成员的更改准确定义可靠的多播。正如我们前面所做的，我们对接收和传递消息进行了区分。特别地，我们再次采用了一个模型，其中分布式系统由消息处理组件组成，如图8.20所示。接收到的消息在此组件中被本地缓存，直到可以将其交付给应用程序，应用程序在逻辑上被作为组成员放置在更高的层。

原子组播的整个思想是，组播消息m与应该交付它的进程列表惟一关联。这个交付列表对应于一个组视图，即组中包含的一组进程的视图，发送方在消息m进行多播时拥有该视图（这个视图是发送的目标进程吗？似乎不是，应该是发送方进程）。一个重要的观察是，列表中的每个进程都具有相同的视图。换句话说，它们都应该同意m应该由它们各自交付，而不是由其他进程交付。

现在假设消息m在其发送者(比如P)具有组视图G时是多播的。此外，假设在进行多播时，另一个进程Q加入或离开组。组成员的这种更改自然会向G中的所有进程声明。换句话说，**视图更改**是通过多播一条消息vc来实现的，该消息宣布Q的加入或退出。我们现在有两个同时传输的多播消息:m和vc。我们需要保证的是m要么由G中的所有进程在任何进程执行vc指定的视图更改之前交付，要么根本不交付。注意，这一要求可与我们在第6章中讨论的全顺序多播相比较。

一个很快浮现在脑海中的问题是，如果m不是由任何进程传递的，我们怎么能说一个可靠的组播协议呢？原则上，只有一种情况允许m的交付失败:当组成员变更时，发送m。 在这种情况下，要么G中所有剩余的(没有错误的)成员在同意P不再属于组之前交付m，要么没有人交付m。如前所述，后者对应的情况是P在有机会发送m之前被认为已经崩溃。

这种更强的可靠组播形式保证了组播到组视图G的消息由G中的每个非故障进程传递。 如果消息的发送方在多播期间崩溃，则消息要么被发送到所有剩余进程，要么被每个进程忽略。这种可靠的多播据说是几乎同步的。



为了说明这些问题，请考虑图8.25中所示的四个进程。在某个时间点，我们有一个由S1、S2、S3和S4组成的组。在一些消息被多播之后，S3崩溃了。然而，在崩溃之前，它成功地向进程S2和S4多播了一条消息，但没有向S1多播。然而，在本例中，虚拟同步确保根本不发送消息，从而有效地建立了消息在S3崩溃之前从未发送过的情况。

S3从组中删除后，其余组成员之间继续通信。稍后，当S3恢复时，它可以在其状态更新到最新之后再次加入组。虚拟同步的原理来自于所有的多播都发生在视图更改之间。换句话说，视图更改充当了一个屏障，没有多播可以跨越这个屏障。在某种意义上，它可以与上一章讨论的在分布式数据存储中使用同步变量进行比较。视图更改发生时正在传输的所有多播都在视图更改生效之前完成。虚拟同步的实现并不像我们下面讨论的那样简单。

**消息顺序**

虚拟同步允许应用程序开发人员将多播看作发生在由组成员更改分隔的时代。然而，关于多播的顺序还没有说过。一般来说，有四种不同的顺序:

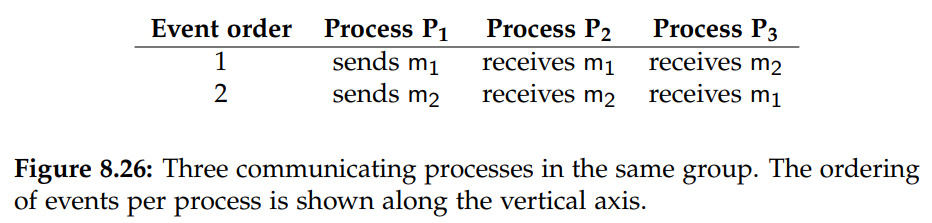
1. 无序多播

2. FIFO-ordered多播

3.有原因地下令多播

4.完全有序的多播

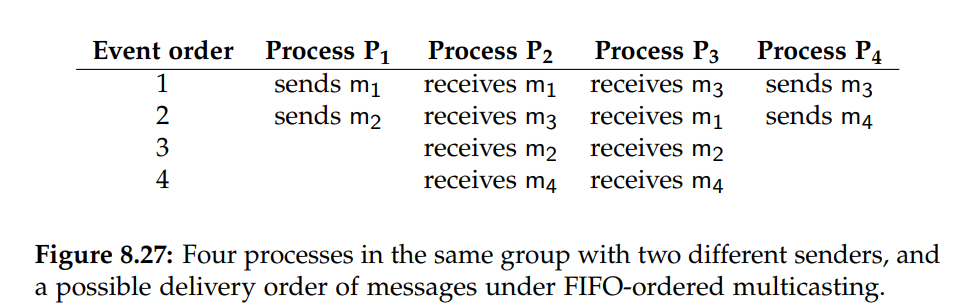
一个**可靠的、无序的组播**实际上是同步的组播，它不保证接收到的消息由不同进程交付的顺序。要解释这一点，假设提供发送和接收原语的库支持可靠的多播。receive操作将阻塞调用者，直到消息可以传递为止。



现在假设发送者P1向一个组多播两条消息，而该组中的其他两个进程正在等待消息到达，如图8.26所示。假设进程在这些多播期间不会崩溃或离开组，那么P2的消息处理组件可能首先接收到消息m1，然后接收到消息m2。因为没有messageorder约束，所以可以按照接收消息的顺序交付消息。相反，P3中的消息处理组件可能首先接收消息m2，然后接收消息m1，并以相同的顺序将这两个消息传递给P3的高级应用程序。

在**可靠的fifo有序多播**的情况下，消息处理组件层被迫按照已发送消息的相同顺序交付来自相同进程的传入消息。考虑一组四个进程中的通信，如图8.27所示。对于FIFO排序，惟一重要的是消息m1总是在m2之前交付，同样，消息m3也总是在m4之前交付。这个规则必须被组中的所有进程遵守。换句话说，当P3的通信层首先接收到m2时，它将与P3一起等待，直到它接收并交付m1为止。

但是，对于不同进程发送的消息的交付没有限制。换句话说，如果进程P2在m3之前接收到m1，那么它可能按照这个顺序传递两条消息。同时，进程P3可能在接收m1之前已经接收到了m3。FIFO顺序表明P3可以在m1之前交付m3，尽管这个交付顺序与P2不同。

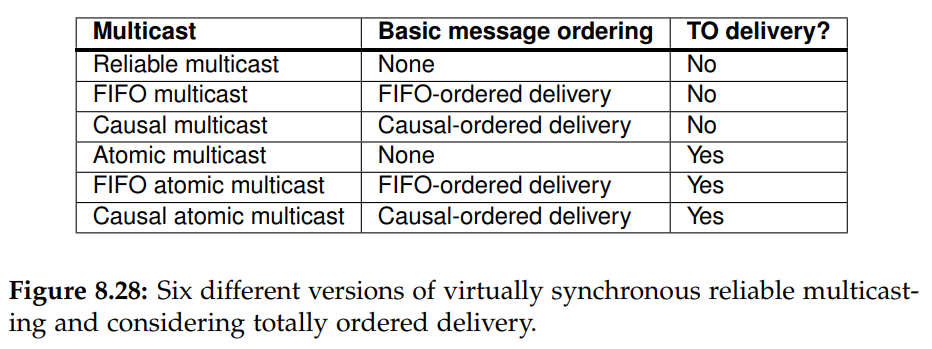


最后，**可靠的因果顺序组播**传递消息，从而保留了不同消息之间的潜在因果关系。换句话说，如果一个消息m1恰好在另一个消息m2之前，不管它们是否由同一个发送方多播，那么每个接收方的通信层总是在接收和交付m1之后交付m2。注意，因果顺序多播可以使用向量时间戳来实现，如第6章所述。

除了这三种排序之外，还可能存在一个额外的约束，即消息交付也必须是完全有序的。 **Total-ordered传递**意味着，无论消息传递是无序的、FIFO有序的还是因果有序的，都需要在传递消息时以相同的顺序传递给所有组成员。

例如，使用FIFO和完全有序组播的组合，图8.27中的进程P2和P3可能都首先传递消息m3，然后传递消息m1。但是，如果P2在m3之前交付m1，而P3在交付m1之前交付m3，则会违反总顺序约束。请注意FIFO顺序仍然应该得到遵守。换句话说，m2应该在m1之后交付，因此m4应该在m3之后交付。

提供消息完全有序传递的可靠同步称为**原子多播**。通过上面讨论的三种不同的消息排序约束，可以得到六种可靠的多播形式，如图8.28所示。



8.5分布式提交（两阶段，三阶段 跳过）

<https://zhuanlan.zhihu.com/p/35298019>

上一节讨论的原子多播问题是一个更常见的问题，称为分布式提交。分布式提交问题涉及由进程组的每个成员执行操作，或者根本不执行操作。在可靠的多播情况下，操作是消息的传递。对于分布式事务，操作可能是在参与事务的单个站点上提交事务。Tanisch[2000]讨论了分布式提交的其他例子，以及如何解决这个问题。

分布式提交通常通过协调器来建立。在一个简单的方案中，这个协调器告诉所有其他也涉及到的被称为参与者的进程，是否(本地)执行有问题的操作。该方案被称为**单阶段提交协议**。它有一个明显的缺点，如果其中一个参与者不能实际执行操作，那么就没有办法告诉协调器。例如，在分布式事务的情况下，本地提交可能是不可能的，因为这会违反并发控制约束。

实际上，需要更复杂的方案，最常见的方案是两阶段提交协议，我们将在下面详细讨论。该协议的主要缺点是它通常不能有效地处理协调器的故障。为此，开发了一个三相协议，我们将在Note 8.13中单独讨论。

原始的两阶段提交协议(2PC)是由Gray提供的。在不失一般性的情况下，考虑一个分布式事务，其中包含多个进程的参与，每个进程运行在不同的机器上。假设没有发生故障，协议由以下两个阶段组成，每个阶段由两个步骤组成：

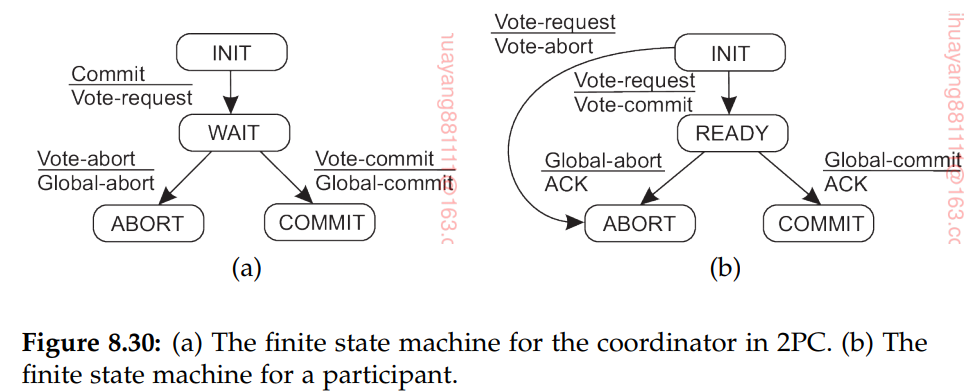
1. 协调器向所有参与者发送一个投票请求消息。

2. 当参与者接收到投票请求消息时，它将向协调器返回一条投票提交消息，告诉协调器它准备在本地提交事务的一部分，或者以其他方式返回一条投票中止消息。

3.协调器收集参与者的所有投票。如果所有参与者都投票提交事务，那么协调器也会投票。在这种情况下，它向所有参与者发送一个全局提交消息。但是，如果一个参与者投票中止事务，协调器也将决定中止事务并多播一个全局中止消息。

4. 每个投票赞成提交的参与者都等待协调器的最终反应。如果参与者接收到全局提交消息，它将在本地提交事务。否则，当接收到全局中止消息时，事务也将在本地中止

第一阶段是投票阶段，由步骤1和步骤2组成。第二阶段是决策阶段，由步骤3和步骤4组成。这四个步骤如图8.30所示为有限状态图。



当在发生故障的系统中使用这个基本的2PC协议时，会出现几个问题。首先，请注意协调器和参与者都具有阻止等待传入消息的状态。

跳过这段，网上有更好的资料讲解 两阶段 三阶段。

<https://zhuanlan.zhihu.com/p/35298019>

8.6恢复

到目前为止，我们主要关注允许我们容忍错误的算法。然而，一旦发生了故障，发生故障的流程必须能够恢复到正确的状态。接下来，我们首先关注恢复到正确状态的实际含义，然后通过检查点和消息日志记录，记录和恢复分布式系统的状态的时间和方式。

**简介**

容错的基础是从错误中恢复。回想一下，错误是系统中可能导致故障的那一部分。错误恢复的整个思想就是用无错误状态替换错误状态。基本上有两种形式的错误恢复。

在**backward recovery（向后恢复）**中，主要问题是使系统从当前的错误状态恢复到以前的正确状态。 为此，需要不时记录系统的状态，并在出现问题时恢复这样的记录状态。每次(部分)记录系统的当前状态时，都要做一个**检查点**。

另一种形式的错误恢复是**forward recovery（向前恢复）**。在本例中，当系统进入错误状态时，将尝试使系统处于正确的新状态，而不是回到以前的检查点状态，从而可以继续执行。正向错误恢复机制的主要问题是必须预先知道哪些错误可能发生。只有在这种情况下，才有可能纠正这些错误并移动到一个新的状态。

考虑到可靠通信的实现，向后和向前错误恢复之间的区别很容易解释。从丢失的包中恢复的常用方法是让发送方重新传输该包。实际上，包重传建立了我们试图回到以前的正确状态，即丢失的包正在发送的状态。因此，通过包重传的可靠通信是应用向后错误恢复技术的一个例子。

另一种方法是使用一种称为**擦除校正**的方法。在这种方法中，丢失的包是由其他成功交付的包构造的。例如，在(n, k)块擦除代码中，一组k个源包被编码成一组n个编码包，这样，任何一组k个编码包都足以重构原始的k个源包。典型值为k = 16或k = 32, k < n≤2k(如[Rizzo, 1997])。如果没有发送足够多的包，发送方将不得不继续发送包，直到可以构造以前丢失的包为止。擦除校正是一种典型的正向错误恢复方法。

在分布式系统中，向后错误恢复技术作为一种通用的故障恢复机制得到了广泛的应用。 向后错误恢复的主要好处是它是一种普遍适用的方法，独立于任何特定的系统或过程。换句话说，它可以作为通用服务集成到分布式系统的(中间件层)中。

然而，向后错误恢复也带来了一些问题[Singhal and Shivaratri, 1994]。首先，就性能而言，将系统或进程恢复到以前的状态通常是一个相对昂贵的操作。正如后面几节将讨论的，通常需要做很多工作才能从进程崩溃或站点故障中恢复。解决这个问题的一个潜在方法是设计非常便宜的机制，通过这种机制可以简单地重新启动组件。

其次，由于向后错误恢复机制独立于实际使用它们的分布式应用程序，因此不能保证一旦恢复发生，相同或类似的故障就不会再次发生。如果需要这样的保证，处理错误通常需要应用程序进入恢复循环。换句话说，完全的故障透明性通常不能由向后错误恢复机制提供。

最后，尽管向后错误恢复需要检查点，但有些状态永远无法回滚。例如，一旦一个(可能是恶意的)人拿走了突然从功能不正常的自动柜员机中滚出来的1000美元，那么钱就只有很小的可能被塞回自动柜员机中。同样地，在大多数Unix系统中，在热情地输入/bin/rm -fr \*但从错误的工作目录中恢复到以前的状态，可能会使一些人脸色苍白（**删库跑路**）。有些事情是不可逆转的。

检查点允许恢复到以前的正确状态。然而，采用检查点通常是一个昂贵的操作，并且可能会带来严重的性能损失。因此，许多容错分布式系统将检查点和**消息日志**结合起来。在本例中，在采取检查点之后，进程在发送消息之前记录其消息(称为基于**发送器的日志记录**)。 另一种解决方案是，让接收进程在将传入消息发送到正在执行的应用程序之前首先记录它。 此方案也称为**基于接收方的日志记录**。当接收进程崩溃时，有必要恢复最近的检查点状态，然后重播已发送的消息。因此，将检查点与消息日志记录相结合可以恢复位于最近检查点之外的状态，而不需要检查点。

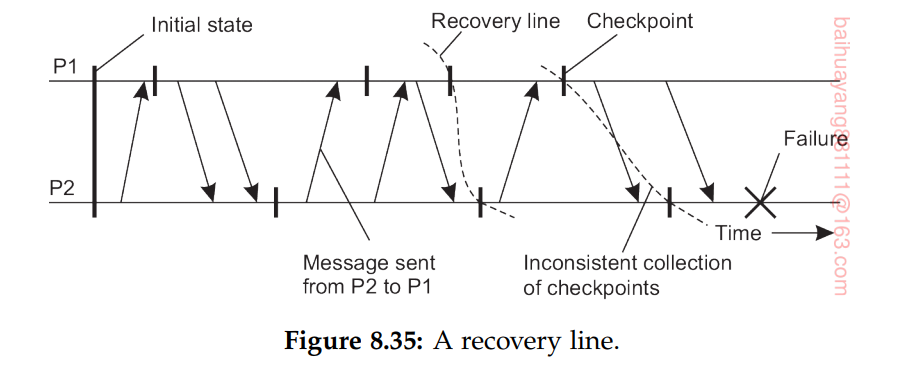
在只使用检查点的系统中，进程将恢复到检查点状态。从那时起，他们的行为可能会与失败发生前有所不同。例如，由于通信时间是不确定的，因此现在可能以不同的顺序传递消息，从而导致接收者做出不同的反应。但是，如果发生了消息日志记录，则会实际重播上次检查点之后发生的事件。这样的重放使得与外部世界的互动更加容易。

例如，考虑由于用户提供错误输入而发生故障的情况。如果只使用检查点，系统在接受用户输入之前必须回退到检查点，以便恢复到完全相同的状态。使用消息日志记录，可以使用较旧的检查点，在此之后可以重播事件，直到用户应该提供输入为止。在实践中，拥有更少的检查点和消息日志记录的组合比使用多个检查点更有效。

Elnozahy等人[2002]对分布式系统中的检查点和日志记录进行了调查。各种算法细节可以在Chow和Johnson[1997]中找到。

**检查点（这小节pass 看不懂）**

在容错分布式系统中，向后错误恢复要求系统定期保存状态。特别是，我们需要记录一致的全局状态，也称为**分布式快照**。在分布式快照中，如果进程P记录了接收到的消息，那么还应该有一个进程Q记录了发送消息。



要在流程或系统故障后恢复，需要从每个进程保存的本地状态构造一致的全局状态。特别是，最好恢复到最近的分布式快照，也就是指最近的分布式快照。

**协调检查点**

在**协调检查点**中，所有进程同步以共同将它们的状态写入本地存储。协调检查点的主要优点是保存的状态自动全局一致。一个简单的解决方案是使用两阶段阻塞协议。协调器首先将检查点请求消息多播给所有进程。当进程接收到这样的消息时，它接受一个本地检查点，对它正在执行的应用程序传递给它的任何后续消息进行排队，并向协调器确认它已经接受了检查点。当协调器收到来自所有进程的确认时，它会多播检查点完成消息，以允许(阻塞的)进程继续运行。

很容易看出，这种方法还将导致全局一致的状态，因为不会将任何传入消息注册为检查点的一部分。这样做的原因是，任何与接受检查点请求相关的消息都不被认为是本地检查点的一部分。同时，传出消息(由正在运行的应用程序传递给检查点进程)在本地排队，直到收到检查点完成的消息。

该算法的一个改进是只向那些依赖于协调器恢复的进程发送检查点请求，而忽略其他进程。如果进程接收到的消息与协调器自上次检查点以来发送的消息直接或间接相关，则该进程依赖于协调器。这就引出了**增量快照**的概念。

要获取增量快照，协调器只向自上次获取检查点以来发送消息的进程发送检查点请求。 当进程P接收到这样的请求时，它将请求转发给自上次检查点以来P本身向其发送消息的所有进程，以此类推。进程只转发一次请求。当所有进程都被标识出来后，第二个多播实际上用来触发检查点，并让进程在它们停止的地方继续运行。

**独立的检查点**

（这章后面的pass）