# 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。

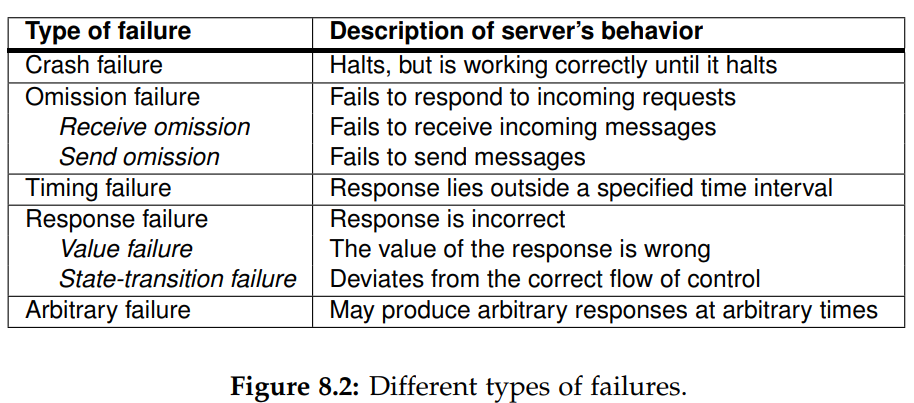
**永久性故障**是指在替换错误组件之前一直存在的故障。烧坏的芯片、软件bug和磁盘头崩溃都是永久性故障的例子。

**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**

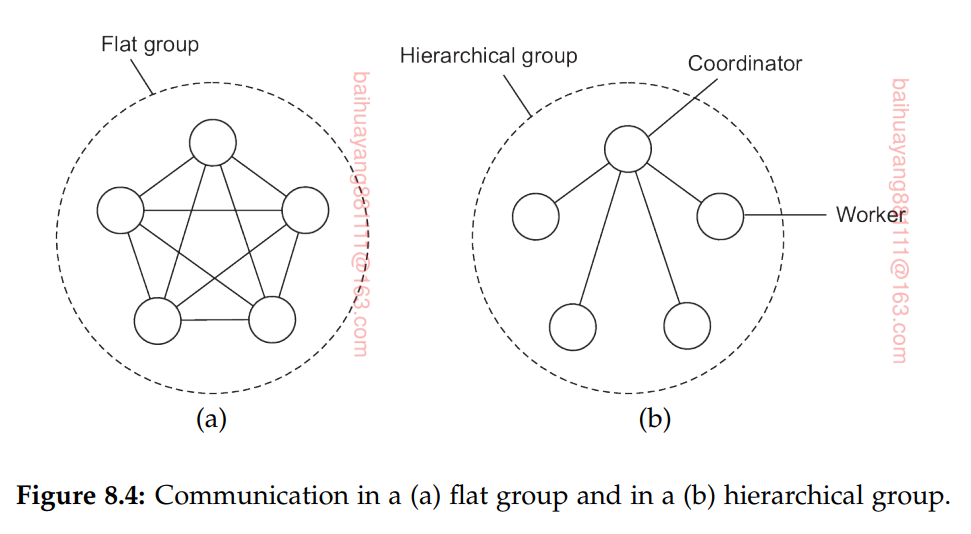
容忍错误过程的关键方法是将几个相同的进程组织成一个组。所有组都具有的关键属性是，当消息发送到组本身时，组的所有成员都会接收它。这样，如果一个组中的一个进程失败了，希望其他进程可以接管它[Guerraoui and Schiper, 1997]。

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:

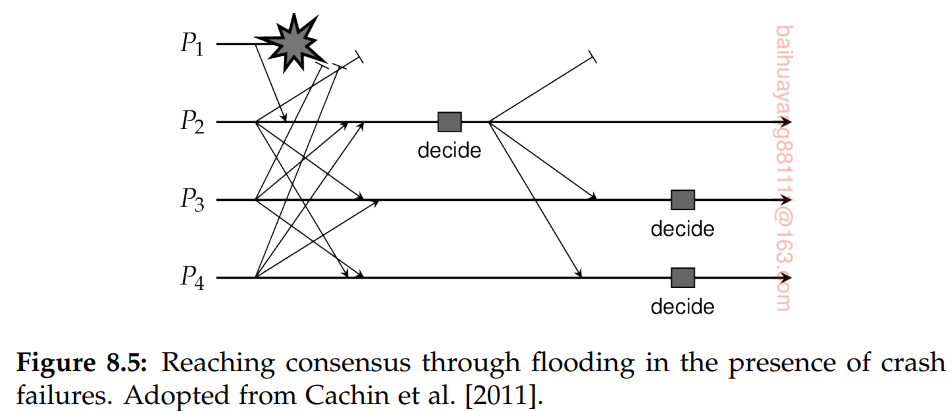
*在容错进程组中，每个非故障进程与其他非故障进程以相同的顺序执行相同的命令。*

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。

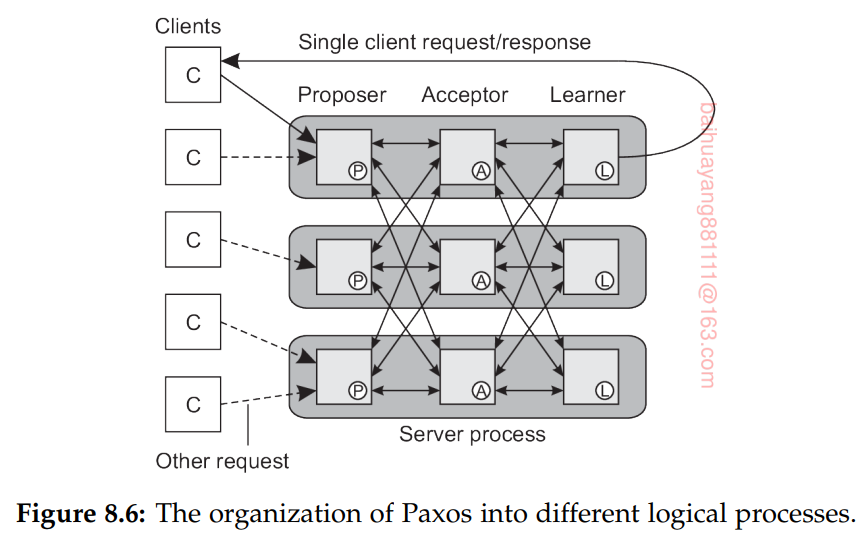
•进程可能会出现崩溃故障，但不会出现任意故障，也不会串通一气。

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。