当我们谈论分布式时我们在谈论什么

我尝试学习分布式系统已经有相当一段时间了，可以说一旦你开始挖掘，似乎就没有尽头了，兔子洞一个接着一个。分布式系统中的文献相当广泛，许多论文来自不同的大学，另外还有许多书籍可供选择。对于像我这样的初学者来说，要决定读什么资料，买什么书是很困难的。

与此同时，我发现一些博主推荐这篇或那篇论文，要想成为一名分布式系统工程师(不管这意味着什么)，必须了解这些内容。因此，需要读取的东西的列表不断增长:FLP, Zab, Time, Clocks and the Ordering of Events in a Distributed Systems, Viewstamped Replication, Paxos, Chubby 等等。我的问题是，很多时候我都找不到理由来解释我为什么要读这篇或那篇论文。我喜欢为了满足好奇心而学习知识这一点，但同时，一个人需要优先阅读什么，因为一天只有24小时。

如上所述，除了丰富的论文和研究材料，还有很多书。我买了好几本书，到处看了几章，开始发现一本书名很有前途的书与我要找的东西完全没有关系，或者内容没有直接针对我想要解决的问题。因此，我想回顾一下我认为分布式系统的主要概念，引用一些论文、书籍或资源，从中可以了解它们。

当我在写这些单词的时候，我在不断地学习，请有一些耐心，允许一些错误，并意识到我将尝试扩展我在这里写的任何东西。

在我们开始之前，我必须告诉你，我已经在各种会议上发表了这篇博客文章，如果你感兴趣，这里有一些幻灯片:

这是我在斯德哥尔摩Erlang用户大会上演讲的视频

让我们从这篇文章开始:

主要概念

分布式系统算法可以根据属性的不同进行分类。分为：时序模型;使用的进程间通信类型;该算法的故障模式假设;我们会看到很多其他的。

以下是我们将看到的主要概念：

- Timing Model（时序模型）
- Interprocess Communication（进程间通信）
- Failure Modes（故障模式）
- Failure Detectors（故障探测）
- Leader Election（领导选举）
- Consensus（一致性）
- Quorums（仲裁集）
- Time In Distributed Systems（分布式系统中的时间）
- A Quick Look At FLP（瞥一下FLP）
- Conclusion（结论）
- References（附录）

时序模型（Timing Model）

这里我们有同步模型、异步模型和部分同步模型。

同步模型是最容易使用的模型;在这里，组件同时采取步骤，即所谓的同步轮。消息传递的时间通常是已知的，我们也可以假定每个进程的速度，例如：一个进程执行算法的一个步骤需要多长时间。这个模型的问题是，它不能很好地反映现实，更不用说在分布式系统上下文中了，在分布式系统上下文中，一个进程可以向另一个进程发送消息，并希望对方给我们一致的信号，所以才可确定消息到达该进程。好的方面是，使用这个模型可以实现理论结果，稍后可以转换到其他模型。例如，由于这个模型对时间的保证，我们可以看到，如果一个问题不能在这些时间保证下解决，那么一旦我们放松它们，它也许就不可能解决(例如，考虑一个完美的故障检测器)。

summary：在同步模型下有问题的模型，在其他模型下，这些问题不可能被解决。同步模型保证了时间。

异步模型变得有点复杂。在这里，组件可以按照它们选择的任何顺序执行步骤，并且它们不能保证执行这些步骤的速度。这个模型的一个问题是，虽然它可以简单地描述和更接近现实，但它仍然不能正确地反映它。例如，一个进程可能需要无限长的时间来响应一个请求，但是在一个真正的项目中，我们可能会对该请求施加超时，一旦超时过期，我们将中止该请求。该模型的一个难点是如何确保流程的活动状态。最著名的不可能的结果之一，“与一个错误的进程达成一致是不可能的”属于这个时序模型，在这个模型中，不可能检测到流程是否已经崩溃，或者流程是否只是花费了无限长的时间来回复消息。

summary：通信的进程间无法达成一致的状态，如果某个进程状态错误，其它进程无法探测到。

在**部分同步模型**中，组件有一些关于定时的信息，可以访问几乎同步的时钟，或者它们可以近似估计消息传递的时间，或者一个进程执行一个步骤需要多长时间。

summary：根据现实，对耗时做出合理的假设。

进程间通信（Interprocess Communication）

这里我们需要考虑系统中的进程如何交换信息。它们可以通过在**消息传递模型**中互相发送消息，或者使用**共享内存模型**(通过访问共享变量共享数据)来做到这一点。

需要记住的一点是，我们可以使用消息传递算法来构建分布式共享内存对象。书中一个常见的例子是读/写寄存器的实现。我们也有队列和堆栈，一些作者使用它们来描述一致性属性，比如linearizabilty。我们不应该将共享内存（通过访问共享变量在进程间共享数据）与构建在消息传递之上的共享内存抽象(就像刚刚提到的那些)混淆。

回到消息传递模型，在尝试理解算法时，我们还需要考虑另一个抽象：进程之间使用的链接类型(想象成用于在进程之间来回发送消息的通道)。这些链接将为使用它们的算法提供一定的保证。例如，有一个完美的链接抽象，具有可靠的交付和发送没有重复；这种抽象还可以确保一次交付。我们可以很容易地看到，这种抽象并没有反映真实的世界，因此，算法设计者在尝试设计更接近真实系统的模型时，还使用了其他类型的链接抽象。记住即使完美的连接抽象是不真实的，它仍然可以很有用，例如，假设连接是完美的，如果我们可以证明一个问题是不可能解决的，那么我们知道很多相关问题可能也不能解决。在链接主题上，作者通常考虑或假设FIFO消息排序，如在zab（找一下论文）中。

故障模式（Failure Modes）

我已经写了一篇关于分布式系统中的故障模式的文章，但是在这里值得重申。分布式系统模型的一个特性是假定了什么样的进程故障。在崩溃停止模式（crash-stop）下，一个进程被认为是正确的，直到它崩溃。一旦崩溃，就永远无法恢复。还有崩溃恢复模型（crash-recovery），其中进程可以在错误之后恢复，在这种情况下，一些算法还包括一种让进程恢复崩溃前的状态的方式。这可以通过从持久存储读取数据或与组中的其他进程通信来实现。值得注意的是，对于一些组成员算法，崩溃然后恢复的进程不能被认为是与以前存在的进程相同的进程。这通常取决于是动态组还是固定组。

summary：故障模式的选择取决于系统的目标，选择崩溃恢复模式是否面临复杂度和成本问题？分析故障并针对故障选择何种恢复方式本身很复杂，如果需要使用日志、持久化存储等外围对象来恢复，又引入了性能、存储成本等问题，是否得不偿失？

也有进程接收或发送消息失败的故障模式，这些称为遗漏故障模式。还有不同种类的遗漏，进程可能无法接收消息或发送消息。为什么这很重要?设想一个场景，一组进程实现了一个分布式缓存。如果一个进程能够接收到来自组上其它进程的请求，但未能响应这些进程的请求，该进程仍将保持其最新状态，这意味着它可以响应客户端的读请求。

summary：这种模式很新鲜，看似组内通信异常，实则组的状态是一致的（最新的，对于缓存而言），提供的服务也是可靠的，我们能看到组内发生了故障，但组提供的某些服务看似没有故障，因此我们不知道组内是否还发生其他故障，导致已知故障的影响被覆盖或者抵消，会不会存在其他影响未被发现呢？组内无故障，对外提供服务可能不正常；组内有故障，对外提供服务可能正常，这就是分布式系统的不可预知性。

更复杂的故障模式称为拜占庭故障模式或任意故障模式，在这种模式中，流程可以向其对等方发送错误信息；它们可以模拟流程；用正确的数据回复其他进程，但是混淆它的本地数据库内容等等。

在考虑系统设计时，我们应该考虑要处理哪种类型的进程故障。Birman(参见可靠分布式系统指南)认为，通常我们不需要处理拜占庭故障。他引用了雅虎所做的工作。他们得出的结论是崩溃故障比拜占庭故障更常见。

summary：故障模式用来描述进程故障的表现，以及处理它们的方式。

故障	处理方式	故障模式
进程奔溃	重启	crash-stop，发生故障就停止服务，通过重启恢复服务
进程奔溃	重启并恢复状态	crash-recovery，发生故障依旧需要重启，重启后需要恢复到故障前的状态
发送消息失败	重发？不处理？	分布式缓存系统，某个故障进程可以接受其它进程的请求，但无法响应它们，该进程虽然故障，但数据是最新的，可以响应客户端的读请求，不需要其他恢复方式。

进程奔溃故障出现的概率通常是最大的，设计系统时要考虑哪些故障是必须处理的？太复杂的故障没必要纠结，一是出现概率低，二是出现时如果使用恢复模式，会陷入递归问题。

故障探测器（Failure Detectors）

根据进程故障模式和时间假设，我们可以构造抽象，以便在流程已崩溃或怀疑已崩溃时向系统报告。有完美的故障检测器，永远不会给出错误的阳性结果。有一个紧急停止故障模式加上一个同步系统，我们可以通过使用超时来实现这个算法。如果我们要求进程定期ping回故障检测器进程，我们就确切地知道ping应该在什么时候到达故障检测器(基于同步模型的保证)。如果ping在某个可配置的超时之后没有到达，那么我们可以假设另一个节点已经崩溃。

在更真实的系统上，也许不可能总是假定消息到达目的地所需的时间，或者假定进程执行步骤需要多长时间。在这种情况下，我们可以有一个故障检测器p，如果q在超时N毫秒后没有响应，它将报告进程q疑似故障。如果q晚一点回复了，那么p将从疑似故障列表中移除q，它将增加N，由于它不知道自己和q之间的实际网络延迟，但它想停止怀疑q奔溃，所以可以说q过去是活跃的，尽管它曾花费比N大的时间完成ping 回复。如果在某个时候q崩溃了，那么p首先会怀疑它已经崩溃了，并且它永远不会修改它的判断（因为q永远不会ping回来）。在可靠和安全的分布式编程介绍这本书的“最终完美故障检测器”这一章下可以找到这种算法更好的描述。

故障检测器通常具有两种特性：完整性和准确性。对于最终完美的故障检测器类型，我们有以下几点：

- **强完整性：**最终，每一个崩溃的进程都会被每一个正确的进程永远怀疑。
- **最终强精确性：**最终，任何正确的进程都不会怀疑正确的进程。

故障检测器在解决异步模型中的一致性问题中起着至关重要的作用。在前面提到的FLP论文中有一个非常著名的不可能性结论。这篇论文讨论了当一个进程可能失败时，异步分布式系统中达成一致的不可能性。解决这个不可能性结论的一种方法是引入一个可以绕过这个问题的故障检测器。

领袖选举（Leader Election）

与故障检测器相关的问题是，实际上做相反的事情，以确定哪个进程没有崩溃，因此可以正常工作。然后这个进程将被网络中的其他对等点信任，它将被认为是能够协调一些分布式操作的领袖。这是像Raft或Zab这样依赖于领袖来协调行动的协议。

在协议中有一个leader会引入节点之间的不对称，因为非leader节点将成为追随者。这将导致leader节点最终成为许多操作的瓶颈，因此根据我们试图解决的问题，使用需要进行领袖选举的协议可能不是我们想要的。注意，大多数通过某种共识实现一致性的协议都使用一个领袖进程和一组追随者。请参阅Paxos、zab（找一下论文）或Raft中的一些例子。

一致性（Consensus）

一致性或者协调问题最早是在Pease，Shostak和Lamport的论文“在故障存在的情况下达成一致”中介绍的。他们是这样介绍这个问题的：

容错系统通常需要一种方法，通过这种方法，独立的处理器或进程可以达成某种确切的共同协议。例如，冗余系统的处理器可能需要周期性地同步它们的内部时钟。或者，他们可能必须确定一个时变输入传感器的值，这个值给他们每个人一个稍微不同的读数。

因此，共识是一个在独立进程之间达成一致的问题。这些进程会针对某个问题提出一些值，比如它们传感器当前的读数，然后根据这些值商定一个共同的动作。例如，一辆汽车可能有不同的传感器，为它提供关于刹车温度水平的信息。这些读数可能有一些变化，取决于每个传感器的精度等等，但ABS计算机需要就应该在刹车上施加多大的压力达成一致。这是我们日常生活中需要解决的一个共识问题。这本书容错实时系统解释了在汽车工业的背景下分布式系统中的共识和其他问题。

实现某种形式的共识的进程通过暴露具有建议和决定功能的API来工作。当共识开始时，进程将提出一个特定的值，然后它将根据系统中提出的值来决定一个值。这些算法必须满足终止、有效性、完整性和一致性。我们对常识举例：

- **终止：**每个正确的进程最终都会决定一些值。
- **有效性：**如果一个进程决定了v，那么某些进程提出了v。
- **完整性：**没有进程决定两次。
- **一致：**不存在两个正确的进程有不同地决定。

关于共识的更多细节，请参考上面提到的原始论文。下面的书也是一个很好的参考：

- Introduction to Reliable and Secure Distributed Programming, Chapter 5.
- Fault-tolerant Agreement in Synchronous Message-passing Systems.
- Communication and Agreement Abstractions for Fault-tolerant Asynchronous Distributed Systems.

仲裁集（Quorums）

Quorums是一种用于设计容错分布式系统的工具。Quorums是指在某些进程可能失败时可以用来理解系统特性的交叉进程集。

例如，如果我们有一个算法，其中N个具有崩溃-失效模式的进程，那么当我们有大多数进程对系统申请某种操作时（例如对数据库进行写操作），我们有一个合法进程数（约束）。如果少数进程可能崩溃，即N/2 - 1进程崩溃，我们仍然有大多数进程知道向系统申请的最后一个操作。例如，Raft在向系统提交日志时使用多数。一旦集群中的半数服务器响应了leader的日志复制请求，leader就会将一个条目应用到它的状态机。leader加上一半的服务器构成了大多数。这样做的优点是Raft不需要等待整个集群响应日志复制RPC请求。

另一个例子是：假设我们希望限制一个时刻只有一个进程对共享资源进行访问。该资源由一组进程S保护。每当进程p想要访问资源时，它首先需要请求对保护资源的大多数进程S的授权。S中的大多数进程将对资源的访问权授予p。现在进程q进入系统并试图访问共享资源。无论它在S中接触哪个进程，q永远不会达到授予它访问共享资源权限的大多数进程，直到该资源被p释放。有关更多细节，请参见The Load,Capacity,and Availability of Quorum Systems。

Quorums并不总是指大多数进程。有时，他们甚至需要更多的进程来组成一个使操作成功的仲裁程序，就像在N组进程可能遭受拜占庭故障的情况下。在这种情况下，如果f是可容忍的进程故障数量，那么quorum将是一组(N + f) / 2以上的进程（请参阅“可靠和安全的分布式编程介绍”）。

summary：少数服从多数，多数是超过一半以上的一个值。有时不仅是限制成功的最少进程数，还可能同时限制失败的最大进程数。

如果你对这个话题感兴趣，有一本书专门介绍分布式系统中的quorums：仲裁系统：用于存储和协商一致的应用程序

分布式系统的时间

理解时间及其后果是分布式系统中最大的问题之一。我们已经习惯了生活中事件一个接一个发生的概念，一个完美定义的happen-before顺序，但是当我们有一系列分布式进程、交换消息、并发访问资源等等时，我们如何判断哪个进程事件之前发生过？为了能够回答这类问题，进程需要共享一个同步时钟，并确切地知道电子在网络中移动、在cpu调度任务等等情况下需要多长时间。显然，这在现实系统中是不太可能的。

讨论这些问题的开创性论文是时间、时钟和分布式系统中事件的顺序，它引入了逻辑时钟的概念。逻辑时钟是一种为系统中的事件分配数字的方法；上述数字与实际时间的流逝无关，而是与分布式系统中节点对事件的处理有关。

有许多种逻辑时钟算法，如矢量时钟或间隔树时钟。

关于分布式系统中时间的一个非常有趣的讨论，我建议阅读Justin Sheehy的文章There is No Now。

我认为时间及其在分布式系统中的问题是需要理解的关键概念之一。**同时性的概念是我们必须放弃的**。这与“绝对知识”的旧观念有关，我们过去认为绝对知识是可以得到的。物理定律告诉我们，即使是光也需要一些时间才能从一个地方到达另一个地方，所以无论什么时候它到达我们的眼睛，它被我们的大脑处理，无论光在交流什么，它都是世界的一个旧视图。翁贝托·艾柯(Umberto Eco)在创造敌人一书中的“绝对与相对”(Absolute and Relative)章节中讨论了这个想法。

summary：时序相关的事件，通过逻辑时钟给事件编序，达到处理时判断先后次序的目的，将事件处理顺序与时间发生时点（即真实世界的时间流逝）分隔开来。

FLP一警

为了完成本文，让我们快速地看一看“**与1个错误进程达成分布式一致性是不可能的**”论文，从而尝试将我们刚刚学习到的关于分布式系统的概念联系起来。

摘要的开头是：

共识问题涉及到一个包含许多进程的异步系统，其中一些进程可能是不可靠的。

所以我们有一个异步系统，没有做出时间假设，不管是处理速度或者消息传递速度。我们也知道这些进程可能崩溃。

这里的问题是，在通常的技术术语 ，异步可能引用处理请求的一种方式，例如RPC，在进程p发送异步请求给进程q，当q处理请求时，p继续做其他事情，那代表：p不阻塞等待一个回复。我们可以看到，这个定义与分布式系统文献中使用的定义完全不同，因此，如果没有这些知识，就很难完全理解FLP论文的第一句话的含义。

论文紧接着说：

在本文中，我们展示了一个令人惊讶的结果，即没有一个完全异步的协商一致协议能够容忍一个进程的突然死亡。我们不考虑错综复杂的故障，我们假设消息系统是可靠的——它正确且准确地交付了所有消息一次

所以论文仅考虑了crash-stop故障模式（有时叫fail-stop）。我们还可以看到，由于消息系统是可靠的，所以没有遗漏故障。

最后，他们还加上了这个约束条件：

最后，我们没有假设检测进程死亡的能力，因此一个进程不可能判断另一个进程是否已经死亡(完全停止)或只是运行得很慢。

所以我们也不能使用故障探测器。

简而言之，这意味着“FLP不可能”适用于使用故障停止模式、基于可靠消息系统以及无法检测进程死亡的异步系统。如果不了解与分布式系统的不同模型相关的理论，我们可能会遗漏其中的许多细节，或者我们只是用与作者的意思完全不同的方式来解释它们。

有关FLP的更详细概述，请参阅本文:FLP不可能的简要介绍

此外，阅读Marcos Aguilera的论文被共识研究绊倒:误解与问题也很有趣，该论文讨论了FLP成为分布式系统中不可能结果意味着什么(剧外警告:不可能的程度与停机问题不同)。

结论

正如您所看到的，学习分布式系统需要时间。这是一个非常广泛的主题，每个子领域都有大量的研究。同时，实现和验证分布式系统也是相当复杂的。在许多微妙的地方，犯错误会使我们的实现在意想不到的情况下完全崩溃。

如果我们选择了错误的仲裁，然后我们新的花哨的复制算法丢失了关键数据怎么办?或者我们选择了一个非常保守的仲裁，它在不需要的情况下减慢了应用程序的速度，使我们与客户打破了SLAs（高可用约定）?如果我们试图解决的问题根本不需要共识，我们可以最终保持一致，那该怎么办?也许我们的系统有错误的时间假设?或者它使用不适合底层系统属性的故障检测器?如果我们决定优化像Raft这样的算法，通过避免这里或那里的一小步，最终破坏了它的安全保证，会怎么样呢?如果我们不理解分布式系统的基本理论，所有这些以及更多的事情都可能发生。

好的，我明白了，我不会重新发明分布式系统的轮子，但是有了这么多的文献和问题，从哪里开始呢?正如本文开头所述，我认为随意阅读论文不会给您带来任何好处，正如FLP论文所示，理解第一句话需要了解各种计时模型。因此，我推荐以下书籍，以便开始:

Nancy Lynch的分布式算法。这本书可以说是分布式系统的圣经。它涵盖了上面所引用的各种模型，每个模型都有相应的算法。

Christian Cachin等人对可靠、安全的分布式编程的介绍。它不仅是一个非常好的介绍，而且涵盖了很多种类的协商一致算法。这本书充满了解释算法的伪代码，这是一件好事。

当然还有更多的书，但我认为这两本书是一个好的开始。如果你觉得你需要潜得更深一些，这里列出了本文中用到的资源：

附录：

- Marcos K. Aguilera. 2010. Stumbling over consensus research: misunderstandings and issues. In Replication, Bernadette Charron-Bost, Fernando Pedone, and André Schiper (Eds.). Springer-Verlag, Berlin, Heidelberg 59-72.
- Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte. 2008. Interval Tree Clocks. In Proceedings of the 12th International Conference on Principles of Distributed Systems (OPODIS '08), Theodore P. Baker, Alain Bui, and Sébastien Tixeuil (Eds.). Springer-Verlag, Berlin, Heidelberg, 259-274.
- Kenneth P. Birman. 2012. Guide to Reliable Distributed Systems: Building High-Assurance Applications and Cloud-Hosted Services. Springer Publishing Company, Incorporated.
- Mike Burrows. 2006. The Chubby lock service for loosely-coupled distributed systems. In Proceedings of the 7th symposium on Operating systems design and implementation (OSDI '06). USENIX Association, Berkeley, CA, USA, 335-350.
- Christian Cachin, Rachid Guerraoui, and Luis Rodrigues. 2014. Introduction to Reliable and Secure Distributed Programming (2nd ed.). Springer Publishing Company, Incorporated.
- Tushar Deepak Chandra and Sam Toueg. 1996. Unreliable failure detectors for reliable distributed systems. J. ACM 43, 2 (March 1996), 225-267.
- Umberto Eco. 2013. Inventing the Enemy: Essays. Mariner Books.
- Colin J. Fidge. 1988. Timestamps in message-passing systems that preserve the partial ordering. Proceedings of the 11th Australian Computer Science Conference 10 (1) , 56–66.
- Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1983. Impossibility of distributed consensus with one faulty process. In Proceedings of the 2nd ACM SIGACT-SIGMOD symposium on Principles of database systems (PODS '83). ACM, New York, NY, USA, 1-7.
- Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12, 3 (July 1990), 463-492.
- Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21, 7 (July 1978), 558-565.
- Leslie Lamport. 1998. The part-time parliament. ACM Trans. Comput. Syst. 16, 2 (May 1998), 133-169.
- Nancy A. Lynch. 1996. Distributed Algorithms. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
- Moni Naor and Avishai Wool. 1998. The Load, Capacity, and Availability of Quorum Systems. SIAM J. Comput. 27, 2 (April 1998), 423-447.
- Brian M. Oki and Barbara H. Liskov. 1988. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In Proceedings of the seventh annual ACM Symposium on Principles of distributed computing (PODC '88). ACM, New York, NY, USA, 8-17.
- Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference (USENIX ATC' 14), Garth Gibson and Nickolai Zeldovich (Eds.). USENIX Association, Berkeley, CA, USA, 305-320.
- M. Pease, R. Shostak, and L. Lamport. 1980. Reaching Agreement in the Presence of Faults. J. ACM 27, 2 (April 1980), 228-234.
- Stefan Poledna. 1996. Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism. Kluwer Academic Publishers, Norwell, MA, USA.
- Michel Raynal. 2010. Communication and Agreement Abstractions for Fault-Tolerant Asynchronous Distributed Systems (1st ed.). Morgan and Claypool Publishers.
- Michel Raynal. 2010. Fault-tolerant Agreement in Synchronous Message-passing Systems (1st ed.). Morgan and Claypool Publishers.
- Benjamin Reed and Flavio P. Junqueira. 2008. A simple totally ordered broadcast protocol. In Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware (LADIS '08). ACM, New York, NY, USA, , Article 2 , 6 pages.
- Justin Sheehy. 2015. There Is No Now. ACM Queue
- Marko Vukolic. 2012. Quorum Systems: With Applications to Storage and Consensus. Morgan and Claypool Publishers.