海量存储技术结课报告

###### 0感想

我读的这篇论文是关于分布式系统测试的，作者是来自伦敦帝国学院，MIT，微软的研究人员。这篇论文讲的是关于测试分布式系统。作者开篇就讲到，测试分布式系统十分具有挑战性，因为测试时会有许多不确定的故障源。传统的测试技术诸如单元测试，集成测试和压力测试，这些测试也很有效果，但是他们针对那些大错误效果比较好，对于那些细微但是很严重的错误，就显得无能为力。作者正是基于这个问题，提出了一种新的测试方法。

新的方法是基于P#，它是C#的扩展，为Microsoft.NET框架中编写的分布式系统建模，规范和系统测试提供支持。要使用P#进行测试，程序员必须用三个构件来扩充原始系统：系统的非确定性执行环境的模型;一个将系统驱动到感兴趣行为的测试工具，和安全性或属性规范。然后使用P#系统地运行测试工具并根据提供的规范验证程序是否出错。

这篇论文的主要贡献如下：

1.提出了一个新的建模方法，指定正确的属性，并用P#测试真正的分布式系统。  
2.作者分享了使用P#测试构建在Microsoft Azure之上的三个分布式存储系统的经验，发现了传统测试技术无法发现的细微的错误。  
3.作者还对新方法的成本和收益进行评估。

下面谈谈我对这篇文章的理解。作者提出的方法相比传统的测试方法的确有许多改进，并且取得了很多进步，发现并修复了很多错误。

但是我认为这篇论文也还有一些缺点。首先作者只是在测试环境中找错误，并不是在真实的生产环境中找错误。如果能在实际的生产环境中发现，定位，修复错误，那么这个方法无疑会更加有效。其次就是作者采用的P#语言，这一点无疑会加重其它平台的负担，如测试基于其他技术开发的分布式系统。对于其它平台分布式系统的测试，这个方法是否还有效，作者尚未提到。如果能给出一个通用的，语言通用的测试方法，那么就更加完美。

下面的内容是对这些论文的简单翻译。着重关注作者的思路和研究方法，对于2-5节作者给出的三个案例，属于对方法的应用，过于专业和具体就没有翻译。

揭露测试环境中分布式存储系统中的错误

###### 摘要

由于多种不确定性的来源，测试分布式系统是具有挑战性的。传统的测试技术，例如单元测试，集成测试和压力测试，对于防止生产时遇到的严重但细小的错误是无效的。诸如TLA+之类的形式化技术只能在基于逻辑的模型层次上验证系统的高级规范，而不能检查实际的可执行代码。在本文中，我们提出了一种测试分布式系统的新方法。我们的方法采用先进的系统测试技术，彻底检查可执行代码是否符合其高级规范，大大提高了重要测试的覆盖率。我们的方法已经应用于Microsoft Azure云计算平台上的三个分布式存储系统。在这个过程中，许多错误被识别，重现，确认和修复。这些错误需要并发性和失败的微妙组合，使得使用传统测试技术难以找到这些错误。我们的方法的一个重要优势是在小的环境中发现了一个错误，并且有完整的系统跟踪，这大大提高了调试的效率。

###### 1引言

分布式系统是非常难以设计，实施和测试的。这个挑战是由于许多非确定性的原因，例如意外的节点故障，系统组件之间的异步交互，由于不可靠的通信通道造成的数据丢失，使用多线程代码来利用多核机器以及与客户机交互。所有这些不确定性的来源都可以很容易地制造出一些难以检测，诊断和修复的bug。这些错误可能隐藏在代码路径中，只能由特定的并发事件触发，并且只能在极其罕见的情况下显示，但后果可能是灾难性的。

生产分布式系统的开发人员使用许多测试技术，如单元测试，集成测试，压力测试和故障注入。尽管大量使用了这些测试方法，但由于并发性和失败事件的微妙组合而产生的许多错误在测试期间被遗漏，并且仅在生产中才被暴露。生产时出现严重的错误，可能会给公司造成巨大的损失，并导致客户的不满。

我们采访了Microsoft Azure的技术领导和高级管理人员，分析了分布式系统开发中的主要问题。目前最为关键的问题之一是如何提高测试覆盖率，以便在测试过程中发现错误，而不是在生产过程中发现错误。对更好的测试技术的需求不是特定于微软的;亚马逊和谷歌等其他公司已经承认，测试方法必须得到改进，才能推动生产中使用的日益复杂的分布式系统的正确性。

最近，亚马逊网络服务(AWS)团队使用正式的方法防止生产中出现严重但微妙的错误。他们的方法是从生产系统中提取高级逻辑，用表达式TLA+语言把这个逻辑表示为规范，最后用模型检查器验证规范。虽然这种方法非常有效，但是在AWS中的使用却证明了这种方法不足以“验证可执行代码是否正确实现了高级规范”，并且AWS团队承认它不知道任何工具可以处理像亚马逊这样庞大复杂的系统“。

我们发现，由于规范和可执行代码之间的差距，检查高级规范是必要的，但是还不够。我们的目标是弥合这一差距。我们提出了一种新的方法，直接在可执行代码上验证高级规范。我们的方法不同于以前的方法，要求开发人员切换到不熟悉的领域特定的语言，或手动注释和编写代码。相反，我们允许开发人员通过使用主流编程语言C#编写测试工具来系统地测试生产代码。这显著降低了微软Azure团队接受的障碍。

我们的测试方法基于P#，它是C#的扩展，为Microsoft。NET框架中编写的分布式系统的建模，规范和系统测试提供支持。要使用P#进行测试，程序员必须用三个构件来扩充原始系统：系统的非确定性执行环境的模型;一个将系统驱动到感兴趣行为的测试工具，和安全性或属性规范。然后P#系统地运行测试工具并根据提供的规范验证程序行为。

最初的P#论文讨论了用P#编写的程序的语言设计问题和数据竞争检测，而这项工作主要集中在使用P#来测试Microsoft内部的三个分布式存储系统：Azure Storage vNext;实时表迁移;以及Azure服务结构。我们在这些系统中发现了很多错误，包括一个微妙的属性错误，它只在压力测试中间歇性地表现出来，并不是固定的。我们的测试方法在一个非常小的环境中发现了这个错误，这使得开发人员可以轻松地跟踪错误痕迹，识别并修复问题。

总而言之，我们的贡献如下：  
•我们提出了一个新的建模方法，指定正确的属性，并系统地用P#测试真正的分布式系统。  
•我们讨论了我们使用P#测试构建在Microsoft Azure之上的三个分布式存储系统的经验，发现了传统测试技术无法发现的细微的错误。  
•我们评估使用我们方法的成本和收益，并显示P#可以在小的环境中检测到错误，并且易于追踪错误。

###### 2用P#测试分布式系统

我们工作的目标是在投入生产环境之前，在分布式系统中发现错误。典型的分布式系统由多个通过消息传递相互作用的组件组成。如果消息或意外的失败和超时处理不当，他们可能会导致微妙的错误。为了揭示这些错误，我们使用了P#，这是C#语言的一个扩展，它提供了：(i)语言支持，用于指定正确性的属性，以及用.NET编写的分布式系统的环境建模。(ii)系统化的测试引擎，可以探索分布式事件之间的交织，比如消息传递的非确定性顺序，客户请求，失败和超时。

使用P#建模涉及三个核心活动。首先，开发人员必须修改原始系统，以便消息不通过真实网络发送，而是通过PSharp.Send(…)方法。这种修改不是侵入式的，因为它可以使用虚拟方法调度来执行，这是一种广泛用于测试的C#语言特性。其次，开发者必须编写一个P#测试工具，通过非确定性地触发各种事件将系统驱动到感兴趣的行为。前两个活动的目的是使用P#明确声明系统中所有不确定性的来源。最后，开发人员必须指定系统执行的正确标准。P#中的规范可以对安全性或属性进行编码。

在测试过程中，P#运行时知道在建模过程中声明的所有非确定性的来源，并利用这些知识创建一个调度点，每次必须进行非确定性选择。P#测试引擎将序列化(在一个单独的盒子里)系统，并且从开始到结束重复执行它，每次探索一个可能不同的非确定性选择集合，直到它达到用户提供的界限，或者违反安全或属性设置。这个测试过程是全自动的，没有误报(假定一个准确的测试装置)。发现错误后，P#将生成一个代表错误路径的跟踪，然后可以重现错误。与通常在生产期间生成的日志相反，P#跟踪提供了所有通信事件的全局顺序，因此更易于调试。

由于分布式系统的高度异步性，这些系统可能达到的状态数目呈指数级增长。诸如MODIST和dBug之类的工具专注于测试未经修改的分布式系统，但当试图彻底探索生产规模的分布式存储系统(如Azure Storage vNext)的整个状态空间时，这很容易导致状态空间爆炸。另一方面，TLA+等技术已成功用于工业领域，对复杂分布式系统的规范进行验证，但无法验证实际代码的执行。

在这项工作中，我们提出了上述两个极端之间的解决方案：根据P#环境中的模型测试一个或多个组件的真正实现。我们的方法的好处是它可以通过探索一个大大减少的状态空间来检测实际实现中的错误。请注意，使用P#进行测试并不是免费的;开发人员不得不花费精力和时间来使用P#构建测试工具。但是，开发人员在部署之前已经花费大量时间为分布式系统构建测试套件。P#方法增加了这种努力，通过投入时间对环境进行建模，通过发现更多的错误来提供红利(参见§6)。原则上，我们的方法不是特定于P#和.NET框架，可以与任何其他具有相同功能的编程框架结合使用。

###### 2.1 P#编程模型

P#程序由多个状态机组成，这些状态机通过交换事件来异步通信。在分布式系统的情况下，可以使用P#事件来模拟系统组件之间的常规消息，失败或超时。P#机声明类似于C#类声明，但是一台机还包含一个事件队列和一个或多个状态。每个可以注册行动来处理传入的事件。

P#机彼此并发运行，每个机器执行一个事件处理循环，从队列中取出下一个事件，并通过调用已注册的动作来处理它。一个动作可能会将机器转换到新的状态，创建一台新机器，向机器发送一个事件，访问一个字段或者调用一个方法。在P#中，发送操作是非阻塞的;该事件被简单地排入目标机器的队列，这将同时出列和处理该事件。所有这些功能都是在轻量级运行时库中提供的，它建立在顶层微软的任务并行库。

###### 2.2 一个分布式系统的例子

图1显示了一个简单的分布式存储系统的伪代码，这个系统是为解释我们的测试方法而设计的。该系统由一个客户端，一个服务器和三个存储节点(SN)组成。客户端向服务器发送包含要复制的数据的ClientReq消息，然后在发送下一个请求之前等待获得确认。当服务器接收到ClientReq时，它首先在本地存储数据(在数据字段中)，然后向所有SN广播ReplReq消息。当SN接收到ReplReq时，它通过在本地存储接收到的数据来处理消息(使用最佳方法)。每个安装程序都会定期发送超时消息。在收到超时后，SN将向包含存储日志的服务器发送同步消息。服务器通过调用isUpToDate方法来处理Sync，以检查SN日志是否是最新的。如果不是，则服务器向过期的SN发送重复的ReplReq消息。如果SN日志是最新的，则服务器将副本计数器加1。最后，当有三个副本可用时，服务器向客户端发送确认消息。

在上面的例子中有两个错误。第一个错误是服务器不跟踪唯一的副本。即使同步SN已经被认为是副本，副本计数器也会在每个最新的同步时递增。这意味着，当存在少于三个副本时，服务器可能会发送确认，这是错误的行为。第二个错误是服务器在发送Ack时不会将副本计数器重置为0。这意味着当客户端发送另一个ClientReq时，它将永远不会收到Ack，从而无限期地阻塞。

###### 2.3 建模示例系统

要系统地测试图1的示例，开发人员必须首先创建一个P#测试工具，然后指定系统的正确性属性。图2说明了一个可以找到§2。2中讨论的两个错误的测试工具。图中的每个框代表一个并发运行的P#机器，而箭头表示从一台机器发送到另一台机器的事件。我们使用三种类型的盒子：(i)圆角和厚的盒子  
边界表示包装在P#机器中的真实组件;(ii)具有薄边框的框表示P#中的建模组件;(iii)带有虚线框的框表示用于安全性或属性检查的特殊P#机。

我们不建模服务器组件，因为我们想测试它的实际实现。服务器被封装在P#机器中，负责：(i)通过PSharp.Send(…)方法发送事件，而不是真实的网络; (ii)将收到的事件传递给被包装的组件。我们对SN进行建模，以便将数据存储在内存中，而不是存储在磁盘上(测试过程中可能效率低下)。我们还对客户端进行建模，以便通过反复发送非确定性生成的ClientReq来驱动系统，然后等待Ack消息。最后，我们对定时器进行建模，以便P#控制系统中所有与时间有关的不确定性。这使得P#运行时可以控制测试期间何时将超时事件发送到SN，并系统地探索不同的时间表。

P#使用面向对象的语言功能，如接口和虚拟方法调度，将实际代码与建模代码连接起来。工业界的开发人员习惯于使用这些功能，并大量使用它们来测试生产系统。根据我们的经验，这显着降低了微软内部工程团队接受P#测试的时间。

在§2。4和§2。5中，我们讨论如何在P#中表示安全性和属性规范，以检查示例系统是否正确。第3节，第4节和第5节介绍了如何使用P#来模拟和测试Microsoft的实际分布式存储系统的细节。有兴趣的读者还可以参考P#GitHub仓库来查找手册和样本(例如Paxos和Raft)。

###### 2.4在P#中指定安全属性

安全属性规范概括了源代码断言的概念;安全违规是有踪迹的。P#支持用于指定P#机器本地的安全属性的常用断言，还提供了通过使用安全监视器(一种可以接收但不发送事件的特殊P#机器)指定全局断言的方法。

安全监视器保持本地状态，该状态被修改以响应从普通(非监视器)机器接收到的事件。这个本地状态用于维护与指定属性相关的计算的历史记录。通过对安全监视器的私人状态的断言来标记错误的全局行为。因此，监视器将规范(监视器内)所需的监视状态与监视器外部的程序状态完全分开。§2。2例子中的第一个bug是一个安全漏洞。为了找到它，开发人员可以编写一个安全监视器(见图2)，该监视器包含一个从唯一SNID到布尔值的映射，表示SN是否是复制或不复制。每次SN复制最新的数据，它都会通知监视器更新地图。每次服务器发出一个确认，它也通知监视器。如果监视器检测到没有实际存在的三个副本发送确认，则触发安全违规。

###### 2.5在P#中指定属性

属性规范概括了非终止，违规的痕迹，表现为缺乏进展。典型地，通过逻辑公式来指定特性。我们采取不同的方法，并允许开发人员编写活动监视器。与安全监视器类似，活动监视器可以接收但不发送事件。

生存监视器包含两种特殊的状态：热和冷。热状态表示执行中需要进展但尚未发生的点;例如一个节点失败了，但是一个新节点还没有启动。活动监视器在通知系统必须进行时进入热状态。生命力监视器在通知系统已经进行时离开热状态并进入冷态。如果活动监视器在无限长的时间内处于热状态，则无限执行是错误的。我们的活跃监视器可以编码任意时间逻辑属性。

生存冲突由无限执行见证，其中所有并行执行的P#机器都相当安排。由于不可能通过在有限的时间内执行一个程序来产生一个无限的执行，所以我们在P#中的属性检查的实现使用几种启发式来近似无限的执行。在这项工作中，我们认为一个执行比一个大的用户提供的界限更长，是一个“无限”的执行。请注意，使用这种启发式时，检查公平性是不相关的，因为我们务实地使用了一个大的界限。§2.2的例子中的第二个bug是属性错误。为了检测它，开发人员可以编写一个活跃状态监视器(见图2)，该活动状态监视器表示客户端发送了一个ClientReq并等待一个Ack的热状态转换为冷态，这表示服务器已经发送了一个响应最终客户端请求的响应。每一次一个服务器收到一个ClientReq，它通知监视器转换到热状态。服务器每发出一次确认，它都会通知监视器转换到冷态。如果监视器在有限的无限执行终止时处于热状态，则会触发违例。

###### 6量化使用P#的成本

我们报告了我们在本文讨论的三个案例研究中应用P#的经验。我们的目标是回答以下两个问题：

1．使用P#对分布式系统的环境进行建模花费了多少人力？  
2.使用P#系统地测试分布式系统花费了多少计算时间？

###### 6.1环境建模的成本

环境建模是使用P#的核心活动。关闭一个系统以使其能够进行系统测试是必需的。表1列出了三个案例研究的计划统计。“系统”下的列是指真实的待测系统，而“P#测试工具”下的列是指用P#编写的测试。我们报告：被测系统的代码行(#LoC);在被测系统(#B)中发现的错误数量;用于测试工具(#LoC)的P#代码行;机器数量(#M);状态转换的数量(#ST);以及动作处理程序的数量(#AH)。  
对Azure存储vNext系统中ExtentManager的环境进行建模需要大约两个星期的兼职开发。这个系统的P#测试工具是三个案例研究中最小的(代码行)。这是因为这个建模练习的目的是重现那些困扰着vNext开发者的特殊缺陷。  
同时开发MigratingTable和P#测试工具大约需要五个星期。测试工具是与实际系统并行开发的。这与另外案例研究不同，在这两个案例研究中，建模活动是独立进行的，在开发过程之后。

建模Fabric需要大约五个月的时间，这是P#作者的努力。相比之下，本文讨论的其他两个系统则由其相应的开发人员进行建模和测试。虽然建模Fabric需要大量的时间，但是这是一次性的工作，每次发布只需要逐步完善。

###### 6.2系统测试的成本

使用P#我们在案例研究中发现了8个严重的错误。正如本文前面所讨论的那样，这些bug在传统的测试技术中很难找到，但是P#设法在一个小的环境中发现和重现它们。据开发人员说，P#的痕迹是有用的，因为它允许他们了解错误，并及时修复它们。在所有发现的错误都被修复之后，我们添加了标志来允许它们被单独重新引入，以用于评估。

表2列出了在每个案例研究中运行P#系统测试引擎并重新引入错误的结果。CS列显示哪个案例研究对应于每个bug：“1”用于Azure Storage vNext;“2”表示MigratingTable。我们不包括Fabric案例研究，因为我们正在等待确认  
发现的错误(见§5)。我们使用运行Windows10Pro64位的8GBRAM在2。50GHzIntelCorei5-4300UCPU上执行所有实验。我们配置P#系统测试引擎执行100,000次执行。所有报告的时间都是以秒为单位的。我们实现了两个不同的调度器，负责选择在每个调度点执行的下一个P#机器：一个随机调度器，它随机从启用的4台机器列表中选择一台机器;和一个随机的基于优先级的调度器，它总是调度启用了最高优先级的机器(这些优先级在执行期间的随机点根据随机分布而改变)。我们决定使用这两个调度器，因为随机调度被证明对于发现并发性错误是有效的。调度程序的随机种子是使用当前系统时间生成的。基于优先级的调度器配置有每次执行2个随机机器优先级改变开关的预算。

对于vNext案例研究，两个调度程序都能够在11秒内重现Extent Node Liveness Violation错误。在bug执行中做出的非确定性选择的数量远高于其他错误的原因是Extent Node Liveness Violation是一个属性错误：如§2.5所讨论的，我们让程序运行很长时间，然后检查属性财产持有。

对于MigratingTable案例研究，我们评估了11个错误的§4的P#测试工具，包括实际发生在开发中的8个有机错误和三个假想错误(用\*表示)，这些错误代码是我们认为有趣的方法使系统不正确。该工具发现了7个有机错误，这些错误在表2中列出。其余四个错误在100,000次执行中没有被我们的默认测试工具捕获。我们认为这是因为触发它们的输入和时间表在使用的发行版中太稀少了。为了证实这一点，我们为每个bug编写了一个自定义的测试用例，用一个特定的输入来触发它，并能够快速地重现错误;表格显示了这些运行的结果。请注意，随机调度程序只能触发7个MigratingTable错误;我们不得不使用基于优先级的调度器来触发剩余的四个错误。

使用P#找到的MigratingTable中的QueryStreamedBackUpNewStream错误很突出，因为它反映了随着系统发展，很容易发生的一种监督。这个错误是在从虚拟表中实现流读取，它应该返回表中所有按键排序的行的流。基本的实现思想是从后端表执行流式读取并合并结果。根据IChainTable规范，每一行从流中读取可以反映在流开始和读取行之间的任何时间的表的状态。开发人员勾画了一个证明，只要迁移者只是将行从旧表复制到新表，合并过程将保留此属性。但是，当迁移者添加支持以在复制之后从旧表中删除行时，后端流可能会看到从旧表中删除行，但不能将其插入到新表中，即使插入首先发生，行将被错过。

P#测试在几秒钟内发现了上述错误。MigratingTable开发人员花了10分钟时间来分析跟踪信息，以便诊断发生了什么事情，尽管当然这是在他们添加了MigratingTable特定的跟踪信息之后，并且有了几天的分析经验。开箱即用，P#踪迹只包含机器级和事件级的信息，但很容易添加特定于应用程序的信息，而且我们在所有案例研究中都这样做了。

###### 7相关工作

与我们的工作最相关的是模型检查和系统并发性测试，这是过去被广泛用于在分布式系统的实际实现中发现Heisenbugs的两个强大的技术。

如MODIST和dBug之类的先进模型检查器通常专注于测试整个未经修改的分布式系统，这种方法很容易导致状态空间爆炸。DEMETER建立在MODIST之上，旨在探索未修改的分布式系统时减少状态空间。DEMETER将大型系统的各个组件单独进行探索，然后动态地提取组件之间的接口行为以执行全局探索。相比之下，我们试图提供一个更实用的处理状态空间爆炸的方法。我们首先使用P#对分布式系统进行部分建模。然后，我们系统地测试每个系统组件针对其P#测试用例的实际执行情况。我们的方法旨在加强单元和集成测试，广泛应用于生产的技术，每次只有单个或少量的组件进行测试。

SAMC提供了一种在系统测试中整合特定于应用程序的信息的方法，以减少工具必须探索的交织集。这种基于偏序降阶的技术是我们的方法的补充：P#可以用它们来减少勘探状态空间。同样，其他工具可以使用像P#这样的语言技术来编写模型，并降低被测系统的复杂性。

MACEMC是用MACE语言编写的分布式系统的模型检查器。MACEMC的焦点是使用基于有界随机游走的算法并结合启发式来发现属性属性违规。由于MACEMC只能测试以MACE编写的系统，因此在工业环境中不易使用。相比之下，P#可以应用于传统代码用C#编写的一种主流语言。

形式化方法已经成功应用于工业领域，以验证分布式协议的正确性。一个值得注意的例子是亚马逊网络服务团队使用TLA+。TLA+是一种表达形式规范语言，可用于通过模型检查来设计和验证并发程序。TLA+以及其他类似的规范语言的局限在于它们被应用于系统的模型而不是实际的系统。即使对模型进行验证，真实实现与已验证模型之间的差距仍然很大，所以实现的缺陷仍然是一个现实的问题。

最近的正式方法包括Verdi和IronFleet框架。在Verdi中，开发人员可以编写和验证Coq中的分布式系统。在系统成功验证之后，Verdi将Coq代码转换为OCaml，然后编译执行。在IronFleet中，开发人员可以使用Dafny语言和程序验证程序构建分布式系统。Dafny使用Z3SMT验证系统正确性，最后将验证的系统行为。相比之下，P#对已经用.NET编写的系统进行了很多的测试，根据我们的经验，这降低了工程团队采用的标准。

###### 8结论

我们提出了一个测试分布式系统的新方法。我们的方法涉及使用P#，这是C#语言的扩展，提供了先进的建模，规范和系统测试功能。我们报告了在Microsoft内部的三个分布式存储系统上应用P#的经验。使用P#，这些系统的开发者发现，重现，确认并修复了大量的错误。

###### 9致谢

我们感谢我们的HaryadiGunawi和匿名审稿人的意见。我们也要感谢伯克利大学的Ankush Desai，和来自微软的Rich Draves、David Goebel，David Nichols 、SW Worth。感谢他们在各个研究阶段的反馈和讨论。最后感谢英特尔公司对研究的支持。