Why Do Computers Stop and What Can be Done About It

作者：Jim Gray 1985
原文：http://www.hpl.hp.com/techreports/tandem/TR-85.7.pdf
译者：phylips@bmy 2013-04-30
译文：http://duanple.blog.163.com/blog/static/709717672013418948 1819/

[序：早在互联网出现以前，Tandem Computers 就已经构建了具有高度容错性和可用性的系统。Tandem Computers 是最早从事容错服务器制造的厂商，它制造的机器广泛应用在银行证券等在线处理交易领域。本文即是 Jim Gray 在 Tandem Computers 工作期间所撰写的，文中揭示了 Tandem Computers 的“NonStop”神话所依赖的那些重要技术：isolation、failing fast、transactional updates、process pairs、supervision。同时提出了容错领域的很多重要概念，诸如：Availability 、Reliability 、MTBF、MTTR。虽然这篇文章是写在 1985 年，距今已 30 年，但是影响深远，其中的很多内容即使在今天看来依然非常有意义。

虽然在之前的一些文章中已经对 Jim Gray 进行了一些介绍，但是在这里还是继续深入介绍一下吧，顺带也说一说 Tandem Computers 。

Jim Gray(1944-2007)，98 年图灵奖得主，“For seminal contributions to database and transaction processing research and technical leadership in system implementation.”。生于加利福尼亚的旧金山，由他的母亲扶养成人。1961 年高中毕业后到 1971 年的这十年，都是在大学里度过的。Gray 最初计划主攻物理学。在参与一个与航空公司为期两年的合作项目后，更加加深了他对学术研究的向往。1966 年毕业拿到数学和工程学学士学位后，他选择继续攻读研究生课程并开展研究工作。之后一年，一方面在新泽西的贝尔实验室进行工作，另一方面还参与了纽约大学科朗研究所一些课程学习。1967 年，又回到了 Berkeley，加入了新成立的 CS 系。1969 年取得了博士学位，研究课题是上下文无关文法和形式语言理论。之后在 IBM 的资助下，又在 Berkeley 做了两年博士后。在此期间，他领导了名为 CAL 的分时系统(Timesharing System)研究项目，该项目旨在建立一个安全可靠的操作系统。

1971 年，Jim Gray 加入 IBM 位于纽约的沃森研究中心的通用科学部门(General Science Department)。在此期间他结识了 John Cocke(因在编译器理论和设计、大规模系统架构及 RISC 方面的突出贡献，而获得 1987 年图灵奖)。在这里度过了一个冬天之后，Gray 决定返回加利福尼亚，加入了 IBM 的圣何塞研究实验室(现今的 IBM Almaden)，该实验室主要与 IBM 的通用产品部门相配合，负责设计和制造计算机磁盘驱动器。那时数据库管理已成为研究热点，而正是作为该实验室一员的 Edgar F. Codd，此前发表了一篇深具影响的论文，在论文中提出了一种管理组织数据库系统的新方式—关系模型。在 IBM 以及其他地方已经有好几个项目开始尝试构建以关系模型为基础的实际系统。1973 年，IBM 决定将来自沃森和圣何塞实验室的人们合并到圣何塞的一个项目中。Gray 不久也加入了这个项目，该项目就是著名的 System R。该项目持续了 5 年—与 Berkeley 的 Ingres 一同—被认为是关系模型工业化的基石。Ray Boyce 和 Don Chamberlin 为 System R 设计了

被广泛使用的 SQL 查询语言。Gray 是 System R 中的关键人物，通过结合他自身在系统和理论方面的丰富经验，提出了解决并发控制和崩溃恢复等相关问题的标准方法，提出了事务的概念。开发了能够在维护数据库一致性的情况下，支持多个事务并发执行以及崩溃后重启的技术。也正是因这些工作而使他获得了 1998 年的图灵奖。当 System R 的研究工作接近尾声的时候，Gray 又开始投入到将这些技术产品化的工作中，并开始思考如何将这些技术扩展到分布式的场景中。

1980 年，Gray 跳槽到了 Tandem Computers，在这里度过了接下来的十年。Tandem 是在将容错性硬件和软件应用于商业系统方面的领军者。它们通过使用高速互联网络将独立计算机连接起来提供高扩展性。在 Tandem Computers 期间，Jim Gray 对该公司的主要数据库产品 ENCOM PASS 进行了改进与扩充，并参与了系统字典、并行排序、分布式 SQL、NonStop SQL 等项目。

Tandem Computers 成立于 1974 年，由四个惠普公司的工程师创立(James Treybig、Mike Green、Jim Katzman、Jack Loustaunou)。他们的商业计划要求系统不会因为“单点故障”而宕机，同时与同类非容错系统相比，仅仅略微贵一些。Tandem 公司认为，这是非常重要的商业模式，限制了额外的费用很重要，因为客户往往在制定程序解决故障时，容错硬件的成本太高。Tandem Computers 公司的第一个系统是 Tandem/16 或称为 T/16 (后来成为 NonStop I 以及其后推出的继任者，NonStop II) 。该系统的设计是在 1975 年完成，并在 1976 年出售给花旗银行。Tandem/16(或称为 NonStop I)由 2 至 16 个处理器组成，每个有着约 0.7 MIPS 的运算能力，在内存、IO 总线，定制的跨 CPU 的计算总线与 Dynabus 四个部分 实现双链接。这个创造了双链接的模块，使任何单一的总线故障(包括 IO 和 Dynabus)并不会影响系统，仍然可以自由的调用的其他模块。1981 年 NonStop II 发布。1983 年的 NonStop TXP 系统，增加了一倍多的速度为 2.0 MIPS，并增加了物理内存至 8MB。这些系统同样的使用了 Dynabus，源于 NonStopI 的超越性设计使 NonStop 在未来得以延续。1986 年进行重大升级的 NonStop VLX 系统发布，VLX 使用新一代的 Dynabus ，速率从 13 Mbit/s 增为至 40 Mbit/s(共计 20 兆位/ s 的独立总线)。他们还推出了 FOX II ，使网络规模从 1 公里增加到 4 公里。1986 年 Tandem 还推出了第一款容错 SQL 数据库—— NonStop SQL 。NonStop SQL 包括了一些基于“守护者”的新功能 ，以确保跨越节点数据的有效性。 在节点增长数量与性能的线性指标上，NonStop SQL 是相当领先的，而当时，大多数数据库往往在两个处理器时，性能已经趋向平稳。后继的版本发布于 1989 年，用于增值交易，在节点增加发方面，仍然在当时可圈可点。1997 年，Tandem 被康柏(Compaq)公司收购，2002 年康柏又被惠普收购，现在 NonStop 成为了惠普的产品。

Gray 认为关系型数据库模型和 SQL 数据访问语言也同样是在线应用的可靠基础，同时他也开始关注客户在对比来自各个软硬件提供商的产品时所遇到的困难。设计了面向终端用户的性能 benchmarks，同时协助成立了中立于厂商的组织—TPCC(Transaction Processing Performance Council)。这也激励厂商们不断地为改进产品做出努力。

Gray 在容错方面的兴趣促使他与 Tandem 的客户们一起研究故障的起因。发表了来自于产品型容错系统相关统计早期论文(即本文)，指出大部分系统故障主要源

自于系统管理操作和软件 bug。随着用户对地理上分布式的计算机系统和终端网络需求的增加，Gray 研究了分布式情况下的可用性、一致性以及其他重要的系统属性。他在 Tandem 期间发表的很多技术报告和论文都是用来帮助客户规划他们的应用，帮助 Tandem 工程师规划加强他们的产品，涉及了性能、可靠性、可用性以及易用性等多个主题。

在 Tandem 呆了 10 年后，1990 年 Gray 转到了 DEC，在接下来的四年他指导了 RDB 关系型数据库管理系统以及 ACMS 事务处理监控器的产品开发。此外，他还与 Andreas Reuter 完成了 “*Transaction Processing: Concepts and Techniques*”的写作。这本书最初源于 1986 年 Gray 为为期一周的研讨会准备的讲稿，到 1992 年写完，已经是 800 多页了。这本书的完成对 Gray 来说是一个重要转折点的标志，此后他的工作开始转向事务处理之外的其他领域。

1994 年，Gray 离开了 DEC，回到了 Berkeley。1994 到 1995 年间，Gray 和 Gordon Bell 提议微软在旧金山建立一个专注于服务器和可扩展性的前瞻性开发实验室。微软接受了这个提议，在接下来的 12 年里，Gray 为他自己设定了这样的目标：“to put all the world’s scientific data online, along with tools to analyze the data”。他与他在微软的同事以及几所大学一起构建了一系列系统，为针对大规模科学数据的访问、搜索和计算提供支持。

2007 年 1 月 28，Gray 驾船出海后失踪。

说明：
Failure：当系统行为与所期望的不一致时我们就说系统发生了故障。
Error：error 是一个系统状态，不进行纠正可能导致 failure。一个 error 是否一定引发一个 failure 是由 3 个主要因素决定的：1.系统是否有冗余，有冗余，error 就不一定导致 failure；2.error 持续时间，可能在产生破坏性后果之前 error 就消失了；3.用户可接受的某种行为底线，譬如有的系统每天都会有一定量的错误发生，在用户可接受的前提下，并不需要做任何纠正动作。
Fault：一个 fault 是一个 error 被证实的或者假定的原因。
这三者之间的关系是：***faults produce errors which lead to failures.***

摘要

针对商业级容错系统的故障统计分析表明软件和管理操作是故障产生的主要来源。后面我们将会讨论各种软件容错技术—尤其是 process-pairs，事务和可靠性存储。需要指出的是产品软件中的 fault 通常都是 soft 的(transient，暂时性的，如果程序状态被重新初始化，并且操作被重新执行的话，通常都不会再次发生)，同时通常是通过将事务机制与持久化的 process-pairs 相结合来提供容错保证—软件容错性的关键。

导引

像病人监护，进程控制，在线事务处理和电子邮件服务这样的一些计算机应用都需要高可用性。

分析下典型大规模系统的故障实际上是非常有意思的：假设现在因操作或软件失误(fault)导致了系统的不可用，图 1 展示了整个的时间线。通常要人们要花几分钟才能意识到出现了问题，同时重启通常是最显而易见的解决方案。操作人员首先花五分钟备份下系统状态以用于后面的分析，之后才能进行重启。对于一个大规模系统来说，操作系统要花几分钟才能启动好，之后是数据库和数据通信系统开始进行重启。数据库可能几分钟内就能完成重启，但是要重启一个大规模终端网络可能要花上一个小时。一旦网络恢复正常，用户可能还要花些时间在那些之前正在执行的任务上。重启后，对于系统来说会堆积了一堆任务需要执行—这样重启之后的瞬时负载将会产生一个峰值。这会影响系统的 sizing。

传统的管理良好的事务处理系统大概每两周发生一次故障[Mourad]，[Burman]。对于这样的系统来说，90 分钟的不可用意味着大概 99.6%{!（14*24*60-90）/（14*24*60）= 99.6%}的可用性，这听起来已经很不错了。但是对于医院的病人，钢厂和 email 用户来说—每 10 天就有 1.5 小时不可用根本是无法接受的。尤其是不可用通常是发生在需求达到峰值的时候(越是需要时越是不可用) [Mourad]。

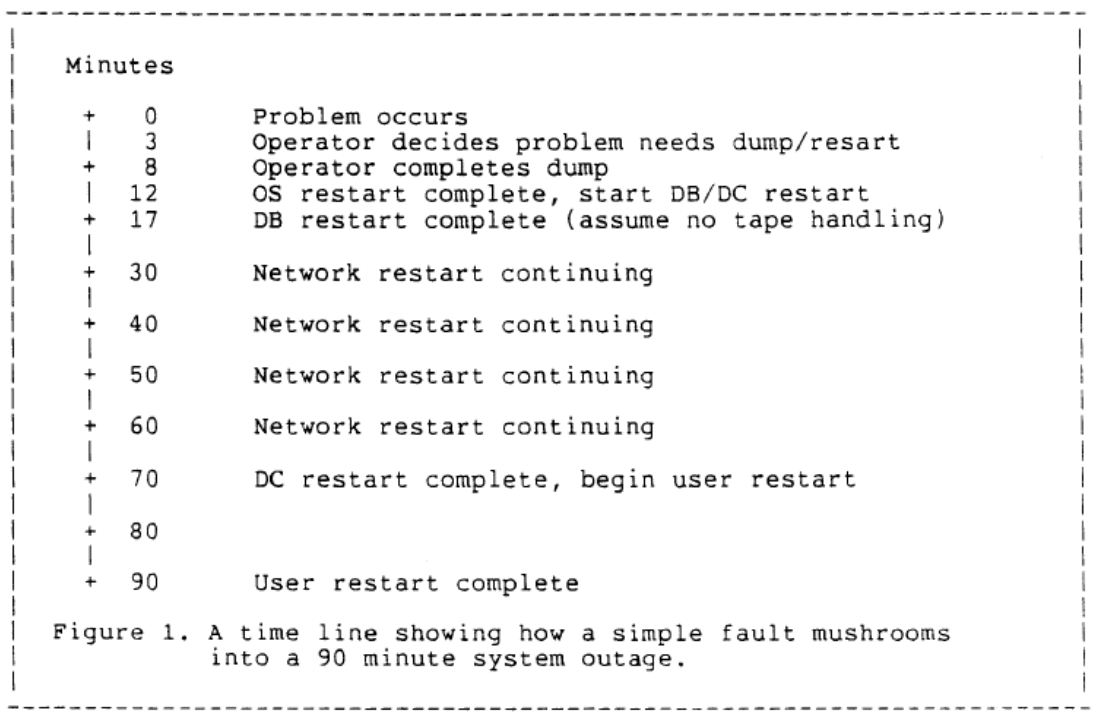

Figure 1. A time line showing how a simple fault mushrooms into a 90 minute system outage.

这些应用需要系统至少从表面上是持续可用的—系统的某些部分可能发生故障，但是剩余部分可以容错并持续提供服务。本文就描述了这样的结构以及这种的成功系统—Tandem NonStop System。它的 MTBF 是以年为单位计的—比传统设计好了两个数量级。

可靠性和可用性是两个不同的概念：可用性是指在给定的响应时间内完成正确的事，可靠性是指不做错事。可靠性可以用 MTBF(Mean Time Between Failures)来度量。故障有一个 MTTR(Mean Time To Repair，平均修复时间)属性。可用性可以用系统可用的概率来进行表达：

$$\text{Availability} = \frac{\text{MTBF}}{\text{MTBF+MTTR}}$$

在分布式系统中，可能是某些部分可用某些部分不可用。提供高可用性的关键是将系统模块化，使得每个模块是一个独立的故障单元并且是可替换的。可用通过配置备用模块来实现快速修复，如果 MTTR 够小，从外面看来故障可能就只是一个延迟而不是故障。

模块化和冗余的结合是在组件发生故障的时候提供持续服务能力的关键。使用冗余技术，在不可靠组件的基础上构建高可靠的系统，是由冯诺依曼首先提出的[Von Neumann]。在他的模型中，如果要让系统的 MTBF 达到 100 年，冗余度需要是 20,000。当然，他考虑的组件的可靠性要远低于晶体管，他当时思考的是人体神经系统和电子管。但是，仍然不太好理解，为什么冯诺依曼需要 20,000 的冗余度，而当前的电子系统只需要 2 就能达到很高的可用性。关键在于冯诺依曼的模型缺乏模块化，线路中的一点故障，通常会导致整个系统发生故障。

冯诺依曼的模型考虑了冗余但是缺乏模块化。但是，现代计算机系统具有鲜明的模块化特征—模块内的故障只会影响到模块本身。此外，每个模块都是 fail-fast 的—要么正常工作要么停止工作[Schlichting]。将冗余与模块化相结合，就使得人们只需要 2 的冗余度，而不再是 20,000。已经相当经济了。

比如，现代磁盘的 MTBF 已经可以达到 10,000 小时之上了—大概一年一次故障。很多系统会采用两个这样的磁盘，在上面存储相同信息，同时为它们使用独立的路径和控制器。假设 MTTR 为 24 小时，同时故障的发生也是独立的，那么整体的 MTBF(两个磁盘的故障发生时间相差在 24 小时之内的平均时间)将会大于 1000 年。实际中，故障并不是那么独立的，但是 MTTR 通常也是要远小于 24 小时的，因此可用性基本上还是这么高。

扩展一下，容错性硬件可以按照如下方式构建：

- 层次化地将系统分解为多个模块
- 将模块设计得使得它们的 MTBF 超过一年
- 使得每个模块都是 fail-fast 的—要么正确完成工作要么不工作
- 通过让模块产生故障信号或者周期性地发送”I AM ALIVE”消息或者设置 watchdog 来迅速检测模块故障
- 配置额外的模块来承接故障模块的工作负载。花费的时间，包括模块故障检测的时间，需要是秒级的。这样就使得模块的 MTBF 可以以千年计。

最终系统的硬件 MTBF 可以以十年或百年计。

这样就实现了容错性硬件。不幸的是，目前还没有说到如何对那些故障的主要来源进行容错：软件和管理操作。后面我们会展示如何将这些同样的想法用于实现软件容错性。

容错系统故障分析

目前已经有很多关于计算机系统故障方面的研究了。据我所知，目前还没有人关注商业性容错系统。针对容错系统的统计数据与传统大型机非常不同[Mourad]。简单来说，硬件软件以及管理操作的 MTBF 是传统计算系统的 500 倍—这就是容错性的力量。另一方面，故障来源中的各类型比率与传统系统基本一致，主要是软件和管理操作，硬件和环境占了一小部分。

Tandem Computers Inc.生产了一系列容错系统。我分析了 7 个月的时间里报告给 Tandem 的系统故障原因，样本集合覆盖 2000 多个系统，系统运行时间超过 10,000,000 个机时(或者说是超过 1300 个系统工作年)。基于与部分客户的交流，这些报告大概覆盖了所有系统故障的 50%。还有一些因为客户和环境引起的故障未被报告。那些供应商方面的故障基本上都被报告了。

整个故障测量期间，总共有 166 个故障被报告，其中包括一次火灾和一次洪水。根据报告计算出的整个系统的 MTBF 是 7.8 年，如果再将那些未被报告的故障考虑进去，MTBF 大概是 3.8 年。都要远远高于传统设计中的 1 周的 MTBF。

通过与 4 个对系统故障具有良好记录的大客户的交流，我得到了一个更精确的数据。他们的 MTBF 平均大概是 4 年(与 7.8 年，50%的报告率相一致)。此外它们的统计数据中，还包含了那些未被报告的部分，我会详细进行分析。

大概 1/3 的故障属于“infant mortality”故障{!即早期失效，处于浴盆曲线的前半段，多为制造缺陷，除此之外还有偶然失效，老化失效}—具有复发问题的产品。如果将这些故障去掉，那么剩下的故障还有 107 个，对于这些故障我们进行了一些有趣的分析(见 table1)。

首先系统的 MTBF 从 7.8 年变成了超过 11 年。

系统管理，包括操作员动作，系统配置和系统维护，是故障的主要来源—42%。软硬件维护又是其中最大的一个类别。高可用系统允许用户在系统工作期间增加软硬件，以及进行一些防护工作。通过在线性的维护，将系统的可用性提高了两个数量级。但是，有时也可能会出错，根据我们的测量数据大概每 52 年一次。这个数字有些牵强—因为只要是在系统进行在线维护或者增添软硬件期间发生的故障，我们都将其归为维护性故障。有时候实际上可能是因为维护人员敲错了命令或者是卸载了错误的模块。有些让人吃惊的是大概每几十年才会发生一个人为导致的故障—很明显工作人员都非常细心，同时设计也容忍了很多人为失误。系统操作是人为故障的第二个主要来源。我怀疑还有很多这样的故障未被报告，因为如果系统故障是因为操作人员导致的，他们很少会报告给我们。系统配置，

保证软件、代码和硬件集合的正确性，是对于可靠的系统管理来说第三大头疼的地方。

软件故障也是系统失效的主要原因—占整体的 25%。Tandem 为客户提供了大概 400 万行代码，尽管已经非常认真，但是软件中仍然免不了会有 bug。此外，客户也写了一些软件。应用方软件故障也很可能不会被报告。我猜想估计只有 30%被报告了。如果这一点成立的话，应用程序导致的系统失效将会是 12%，整个因软件引起的故障将达到 30%。

下一个要说的是环境方面的故障。完全的通信故障发生过三次，此外还有一次火灾，一次洪水。未接到由于制冷或空气条件引起的故障报告。对于那些没有紧急备用电源的客户来说，电力失效是故障的主要来源(北美城市供电系统的 MTBF 通常为 2 个月)。Tandem 系统可以在保证不丢失数据和通信状态的情况下，容忍 4 个小时以上的电力失效(MTTR 几乎为 0)，因此客户通常不会将小于 1 小时的电力故障报告给我们。

由于电力故障并未被报告，因此实际上系统失效的最小来源是硬件，主要是磁盘和通信控制器。{!从下表中看，表面上硬件占了 18%，环境占了 14%，环境是比例最小的那个原因，但是由于电力故障实际上很多并未被报告，如果将这些未被报告的电力故障考虑在内，实际上环境的比例将会增大，这样硬件实际上就变成了比例最小的那个了}。该测量集合包含了超过 20,000 个磁盘—超过 100,000,000 个磁盘工作时。我们观察到了 19 次磁盘故障，但是如果将“infant mortality”故障排除在外，那么就只剩下 7 个了。此外，MTBF 达到了 500,000,000 小时。这与前面计算出来的 1000 年的 MTBF 相吻合。

System Failure Mode	Probability	MTBF in years
Administration	42%	31 years
Maintenance:	25%	
Operations	9% (?)	
Configuration	8%	
Software	25%	50 years
Application	4% (?)	
Vendor	21%	
Hardware	18%	73 years
Central	1%	
Disc	7%	
Tape	2%	
Comm Controllers	6%	
Power supply	2%	
Environment	14%	87 years
Power	9% (?)	
Communications	3%	
Facilities	2%	
Unknown	3%	
Total	103%	11 years

Table 1. Contributors to Tandem System outages reported to the vendor. As explained in the text, infant failures (30%) are subtracted from this sample set. Items marked by "?" are probably under-reported because the customer does not generally complain to the vendor about them. Power outages below 4 hours are tolerated by the NonStop system and hence are under-reported. We estimate 50% total under-reporting.

Implications of the Analysis of MTBF

上面的统计信息已经揭示了一个非常明显的结论：高可用性的关键在于容忍软件和人为操作方面的错误。

根据测量结果，商业化容错系统具有 73 年的硬件 MTBF(table1)。我认为因硬件问题导致的系统失效能达到 75%的报告率。结合设备的 MTBF，可以计算出在采样集合中大概产生过 50,000 次硬件错误(fault)。其中只有不到千分之一的导致了双重故障或服务中断。硬件容错性已经非常好了。

未来，伴随着更好的设计、集成水平的提升和连接器数目的降低，硬件将会变得更加可靠。

与之相比，软件和系统管理方面的趋势则不乐观。系统变得越来越复杂。在我们的研究中，在超过 1300 年的操作时长内，管理员报告了 41 个关键性的操作失误。这样，操作的 MTBF 就是 31 年。实际上操作员应该犯了更多错误，只是大部分错误并不致命。而且这是在管理员都已经是非常细致，实践经验非常丰富的情况下。

提高系统可用性最好的办法是让系统可以自配置，尽量减少人为的维护工作来降低人为的失误。此外系统提供给操作员的接口应当尽可能的简单、保持一致并且容错。

系统维护也是这样。维护接口应当尽量简化。新设备的安装必须具有容错过程，维护接口必须简化甚至消除。比如，Tandem 的最新磁盘设备已经不需要对客户工程师进行专门培训(安装过程已经是“傻瓜”式的)，同时也不需要专门对它们进行维护。

上面的统计信息的第二个暗示揭示了一种矛盾：

- 新的和变化中的系统具有更高的故障率。新出厂的产品占据了所有失效的 1/3。维护工作又占据了剩余失效的 1/3。提高可用性的一种方式是只安装那些经过验证的硬件和软件，然后就不再管它们。正如老人们常说的“没有坏，就不要去修它”。
- 另一方面，Tandem 的相关研究发现相当一部分的失效是由一些已知的硬件和软件 bug 引起的，尽管它们已经被 fix 过了，但是这些 fix 还未更新到这些系统中。这也告诉我们应该尽快安装软硬件的 fix。

这里有两个极端：从不进行变更和尽可能快地进行变更。普遍认为，变更的风险是非常高的。大部分的安装版本都怠于安装更新，而是依赖于容错性拖到下一个主版本发布。毕竟，如果它昨天还能工作，那么明天也应该可以继续工作。

这里我们需要将软件和硬件维护区分开来。软件 fix 的个数要高于硬件 fix 几个数量级。正因此才导致了软件和硬件维护策略上的差异。人们不能摒弃那些预防性硬件维护工作—我们的研究显示短期内可能看起来没问题但是长期看来不做这些工作可能会导致灾难。一定要及时安装硬件 fix。同时还要尽可能地进行预防性维护工作以最小化错误产生的影响。而软件则不同，同一个研究表明应该只在软件 bug 会导致系统失效的情况下安装 bug fix。否则，应该等待下一个主版本的发布，或者是在安装到目标环境之前进行严格的测试。Adams 也得出过类似结论[Adams]，他指出对于绝大多数 bug 来说，重现的概率实际上是非常低的。

统计结果也表明如果可用性是一个主要目标，那么应当尽量避免使用那些不成熟的或者是经常遭受infant mortality故障的产品。处于技术前沿是好事，但是也要注意避免新技术带来的损失。

统计结果的最后一个暗示是：软件容错性是非常重要的。软件容错性是本文后面要讲述的主题。

Fault-tolerant Execution

通过上面的分析可以看出，软件大概占了整个失效的超过25%。50年的MTBF已经是非常优秀的了。Tandem的软件系统有400万行代码，同时每年还以20%的速度在增长。虽然代码实践和代码测试方面的改进工作一直在进行，但是要清除所有

软件中的所有bug基本上是不可能的。就算程序在历经设计review、质量保证、beta测试之后，大概每千行代码仍会有一个bug。这表明系统中仍存在数千个bug。只是这些bug很少会导致系统故障，因为系统可以容忍软件错误。

这种软件容错性的关键在于：

- 通过进程和消息实现软件模块化
- 通过fail-fast的软件模块实现故障包容
- 通过进程对(process-pairs)来容忍硬件和临时的软件错误
- 通过事务机制来保证数据和消息的完整性
- 通过将事务机制与进程对(process-pairs)相结合来简化异常处理和容忍软件错误

本节将会对上述几点展开进行讨论。

通过进程和消息实现软件模块化

与硬件类似，软件容错性的关键在于要将一个大系统分解成多个模块，每个模块是一个独立的服务和故障单元。单个模块的故障影响不会超出这个模块的范围。

关于如何进行软件模块化有一些值得关注的争论。从Burrough的 Esbol开始直到后续的像Mesa和Ada这样的语言，编译器作者一直都假设硬件是完美的，他们认为通过静态的编译时类型检查就可以很好地将错误隔离。与之相比，操作系统设计者们强调进行运行时检查，并将进程作为进行保护和故障的单元。

尽管由编程语言提供的编译器检查和异常处理的确非常有价值，但是历史似乎更青睐于运行时检查以及采用进程进行错误包容。它的好处在于简单性—如果进程或处理器工作不正常，直接停掉它就好了。进程提供了一个用来进行模块化、服务、错误容纳及故障处理的非常清晰的单元。

通过 fail-fast 的软件模块实现故障包容

用于进行错误隔离的进程策略要求进程软件模块必须是fail-fast的，要么它能够正确工作，要么能够检测出错误、报告故障以及停止运行。

可以通过防御式编程使得进程是fail-fast的。这种方法会检查所有的输入，中间结果，输出以及相关数据结构。当检测到错误后，它会产生一个故障信号然后停止。Fail-fast软件通常具有很小的错误检测延迟。

通过不与其他进程共享状态，进程可以实现错误包容；但是它就只能通过内核消息系统中的消息与其他进程进行通信。

软件错误是 soft 的—Bohrbug/Heisenbug 定理

在进一步解释容错、process-pairs 之前，我们需要有一个软件故障模型。众所周知，绝大多数的硬件错误都是 soft 的—也就是说，大部分硬件错误是暂时性的。内存错误纠正和校验，以及通信系统中的重传机制是用来解决暂时性硬件错误的标准方式。这些技术可以将硬件的 MTBF 提升 5-100 倍。

实际上软件也与之类似—大部分软件错误都是 soft 的。如果程序状态被重新初始化，并且操作被重新执行的话，通常都不会再次失败。

对于一个已经经历了结构设计、设计 review、质量保证、alpha 测试、beta 测试和数月乃至数年的产品化的软件系统来说，绝大多数“hard”(即使重试也总是会失败的)的软件错误都已经没有了。剩下的 bug 都是很少情况下才会被触发的，比如在特殊的硬件条件(罕见的或瞬时的硬件错误)，极端条件(内存耗尽、计数器溢出、中断丢失等)或者是竞态条件(忘记申请信号量)。

在这些情况下，将程序重启然后再重新执行，它很可能就又可以工作了，因为环境有了轻微的差别。而且毕竟它一分钟之前还可以工作的。

大部分产品软件的 bug 都是 soft 的—Heisenbug(不确定的 bug，当你盯着它们的时候它们又消失不见了)—这一点对于系统程序员来说已经是众所周知的了。Bohrbugs,类似于玻尔原子，是实实在在的，很容易通过标准技术检测出来，因此会比较令人困扰。但是 Heisenbugs 可能会在 bugcatcher 眼皮底下潜藏数年而不被发现。实际上 bugcatcher 只要能让 Heisenbug 消失就足够了。这非常类似于物理学中的不确定性原理。

我已经尝试过对可以通过重新执行来容忍 Heisenbug 的几率进行定量研究。这是非常困难的。并没有得到什么有价值的东西。我是这样来做这个实验的：查看几十台机器上的 Spooler 的错误日志。Spooler 是由一组 fail-fast 的进程集合组成。当其中一个进程检测到错误时，它会停止运行，然后让它的兄弟继续进行相关操作。兄弟进程会进行一次软件重试。如果兄弟进程也运行失败，那么这个 bug 就属于 Bohrbug 而不是 Heisenbug。在整个测量周期内，在 132 个软件错误中，只有一个是 Bohrbug，剩余的都是 Heisenbug。{! Heisenbug 这一术语出自 Jim Gray 或 Bruce Lindsey，其定义就是那些不易查找且不可复制的漏洞。相比之下，Bohrbug 是另一种硬性漏洞，只要在特定输入环境中运行某段代码，这些漏洞就一定会出现。这两个术语的出处为 Niels Bohr(尼尔斯·玻尔)的原子桌球模型和 Werner Heisenberg(沃纳·海森堡) 的观察者效应模型}

相关研究也曾经在[Mourad]中被报告过。在 MVS/XA 中，有专门的恢复过程负责尝试从软硬件错误中进行恢复。如果软件错误是可恢复的，那么它就是一个 Heisenbug。在该研究中，系统软件中大概有 90%的软件错误会触发恢复过程，在这些恢复中，又有 76%的成功率。这意味着，MVS 恢复过程大概将整个系统的 MTBF 扩展了 4 倍。

如果能够对该定理进行更深入的定量研究是非常有益的。这样，系统设计者就可以从实际经验出发对 Heisenbug 定理进行进一步的挖掘以提高软件容错性。

Process-pairs for fault-tolerant execution

人们可能会认为 fail-fast 模块将会产生可靠的但是不可用的系统—模块始终处在停止过程中。但是，就像容错性硬件那样，配置额外的软件模块，可以让在出现硬件故障或软件 Heisenbug 的情况下的 MTTR 达到毫秒级。如果模块的 MTBF 为 1 年，那么双进程就可以提供十分令人满意的 MTBF。将进程数变为 3 倍，无法再对 MTBF 进行改进，因为系统其他部分的(比如操作人员)MTBF 与它的差距是多个数量级的(即单纯提高某个方面的冗余度，对于系统整体 MTBF 的影响会变小，因为系统是有短板的)。因此，在实践中容错性进程组通常被称为 process-pairs。有如下几种测量来设计 process-pairs：

- Lockstep：在该方案中，主进程和副本进程分别在不同的处理器上同步地执行相同的指令流[Kim]。如果其中一个处理器发生故障，另一个会继续执行计算。这种方法提供了很好的硬件容错性，但是无法容忍 Heisenbug。因为这两个流始终执行相同的程序，最终也会以相同的方式失败。
- State Checkpointing：在该模式中，会通过通信会话在请求者和 process-pairs 之间建立联系。Pair 中的主进程负责进行计算，然后将状态改变和返回信息发送给它的备份。如果主进程发生故障，会话会切换到备份进程继续与请求者进行对话。使用会话序列号来检测重复和丢失的消息，同时在重复请求到达时重发之前的响应[Bartlett]。实践证明 checkpointing process-pairs 可以提供出色的容错性(见 table1)。但是它的编程实现比较困难。目前的趋势是转而采用下面要讲的 Delta 或 Persistent 策略。
- Automatic Checkpointing：这种模式非常类似于 state checkpointing，只是在这里由内核自动化地管理 checkpointing，将编程人员从繁重的工作中解放出来。正如[Borg]中描述的，所有发送给和来源于进程的消息都会由消息内核为备份进程保存好。此后，这些消息会被备份进程进行 replay 以恢复到主进程的状态。当需要在备份处进行大量的计算和存储时，可以将主进程状态直接拷贝到备份处，这样就可以忽略消息日志并不需进行 replay。这种模式看起来比 state checkpointing 发送了更多数据，也产生了更多的执行开销。
- Delta Checkpointing：这是 state checkpointing 的一种演化。它是将逻辑的而不再是物理的更新发送给备份[Borg]。Tandem 通过使用这种模式将成功地消息流量减半，并将总的消息字节数降低了 3 倍[Enright]。除了性能上的优势外，它还将主副本之间的耦合从物理上变成了逻辑上的。这意味着主本中的 bug 基本上不会影响副本的状态。
- Persistence：在 persistent process-pair 中，如果主进程发生故障，副本进程会以空白状态进行启动，并且对发生故障时主进程的状态一无所知。只有那些正在打开和关闭的会话会被同步到备份中。这些在[Lampson]中被称为 stable process。进程持久化是最简单的方式，同时开销也很低。进程持久化的唯一问题在于它们无法隐藏故障。如果主进程发生故障，它所管理的数据库和设备将会处于混乱状态，同时请求者会观察到备份进程就像失忆了似的。我们需要一种简单的方式来对这些进程重新进行同步，使得它们处于相同状

态。正如下面要解释的，事务就提供了这样的重同步机制。

上面这些策略的优点和缺点总结如下：

- Lockstep 进程无法容忍 Heisenbug
- State Checkpoints 提供了容错性，但是很难编程实现
- Automatic Checkpoints 看起来比较低效—向备份发送了大量数据
- Delta Checkpoint 性能很好但是很难编程实现
- Persistent Processes 在故障发生时会丢失状态

下面我们要指出的是，通过将事务与 persistent processes 结合，简化了编程同时提供了出色的容错性。

Transactions for data integrity

事务由一组操作组成，它们可能是数据库更新操作、消息、外部计算机的动作，这些操作共同组成了一个一致性的状态转换。

事务具有 ACID 属性[Haeder]：
原子性(Atomicity)：事务的动作要么全部完成要么一个都不做。事务要么 abort 要么 commit。
一致性(Consistency)：每个事务应当看到的都是正确的状态视图，即使同时有多个事务正在进行状态更新。
完整性(Integrity)：事务应该是一个合法的状态转换。
持久性(Durability)：事务一旦提交，所有的后果都必须被保留，即使发生了故障。
{!此处应该是Jim Gray的误记，就算是在Haeder 的"Principals of Database Recovery" 中，ACID代表的也是：Atomicity , Consistency ,Isolation, Durability ，上面的描述中，Consistency 实际上相当于Isolation ， Integrity 相当于Consistency }

事务提供给程序员的编程接口是非常简单的：通过 BeginTransaction 发起一个事务，通过 EndTransaction 或 AbortTransaction 结束事务。剩下都是由系统负责完成。

经典的事务实现是采用锁来保证一致性，同时使用日志或 audit trail(审计跟踪，系统活动的流水记录)来保证原子性和持久性。Borr 展示了如何将这些概念扩展到分布式的容错系统中[Borr81,84]。

事务使得应用程序员从需要考虑处理大量错误的窘境中解脱出来。在事情太复杂的情况下，程序员可以通过调用 AbortTransaction 将状态进行重置使其回到事务起始状态。

Transactions for simple fault-tolerant execution

事务提供了可靠的执行过程和数据可用性(reliability means not doing the wrong

thing, availability means doing the right thing and on time)。事务本身并未直接提供高可用性。如果硬件发生了故障或者产生了软件错误，绝大多数的事务处理系统将会停止并进行重启—导引中所提到的90分钟的不可用。

是有可能通过将事务与process-pairs相结合来提供容错性，从而避免绝大多数这样的不可用。

如前所述，process-pairs可以容忍硬件故障和软件Heisenbug。但是大多数类型的process-pairs都很难实现。而简单点的process-pairs，像persistent process-pairs，在主进程出错副本进程替换它时会产生失忆症状。在副本进程接任后，网络和数据库会处于未知状态。

关键在于事务机制知道如何UNDO掉未完成事务的所有变更。因此，我们可以简单地abort掉与发生故障的persistent process相关的所有未提交的事务，然后再根据输入消息将这些事务重启。这样就清空了数据库和系统的状态，并将它们重置为事务开始时的状态。

这样persistent process-pairs加上事务就提供了一个即使是在发生硬件故障或Heisenbug情况下仍然可以继续执行的简单执行模型。这就是Encompass数据管理系统容错性的关键[Borr81]。程序员通过传统语言(Cobol, Pascal, Fortran)编写fail-fast模块，同时事务机制加上persistent process-pairs保证了编写出来的程序的健壮性。

不幸的是，那些编写操作系统内核、事务机制本身以及某些设备驱动器的人们仍然需要编写传统的process-pairs，但是应用开发者已经不需要了。Tandem将事务机制与操作系统集成的原因之一就是让事务机制对尽可能多的软件可用[Borr81]。

Fault-tolerant Communication

通信链路是分布式系统中最不可靠的部分。部分是因为它们数量众多，部分是因为它们的 MTBF 比较差。对它们进行管理、故障诊断和维修过程的跟踪等方面所需的操作是非常令人头疼的[Gary]。

在硬件层面，通过采用具有不同故障模型的多个数据路径来实现通信容错性。

在软件层面，引入会话概念。会话具有如下简单语义：通过会话来完成一系列消息的传输。如果通信路径发生故障，则会尝试其他路径。如果所有路径都不通，会话就会以失败告终。采用超时和消息序列号机制来检测消息的丢失和重复。所有这些在会话层之上都是透明的。

通过会话使得 process-pairs 可以工作。在主进程发生故障时，会话会切换到备份进程[Bartlett]。通过会话序列号(Bartlett 称之为 SyncIDs)对接收者和发送者间的通信状态进行同步，使得请求/响应是幂等的。

事务与会话的配合如下：如果事务 abort 了，会话序列号就会从逻辑上地被重设为事务开始时的序列号，同时所有中间产生的消息将会被取消。如果事务成功提交，该会话的消息将保证会且仅会被传输一次[Spector]。

Fault-tolerant Storage

容错性存储的基本形式就是将一个文件存储在两个具有独立故障特征的存储媒介上—比如，两个不同的磁盘驱动器上，或者更好一点一个磁盘、一个磁带。如果单个文件的 MTBF 是一年，那么两个文件就有一千年的 MTBF，三个拷贝也只能达到与两个文件相同的 MTBF，这是因为 Tandem 系统上的相关统计显示，在这种情况下，其他因素会成为瓶颈。

远程备份可以突破这一点。如果可以负担的起，能够将副本存储在一个远程位置上，那么是可以很好地提高可用性的。远程副本将具有不同的管理员，不同的硬件，以及不同的环境。只有软件是相同的。基于 Table1 的分析，这将可以防止 75%的故障(即所有的非软件故障)。由于这种方式对 Heisenbug 也具有很好的防护能力，因此远程备份也可以有效的避免大多数的软件错误。

有很多方式可以对数据进行远程备份，我们可以让副本完全一致，也可以是尽可能快地将更新应用到多个副本上，甚至是定期地对它们进行更新。[Gray]描述了一些采用不同策略实现远程副本的代表性系统。

事务为存储系统提供了 ACID 属性—Atomicity、Consistency、Integrity 和 Durability [Haeder]。事务日志再加上数据的归档拷贝就提供了一种存在于具有独立故障模型的媒介上的数据副本。如果主副本出现问题，那么还可以通过归档拷贝以及将归档之后提交的那些更新应用在该拷贝上来进行重构。这就是数据持久性。

此外，事务还会对更新进行协调控制，以保证 all-or-nothing 属性。这就允许我们对复杂数据结构进行更新时，不用担心故障的发生。如果出现问题，事务机制可以将更新操作取消。这就是原子性。

用于实现容错性存储的第三种技术是将数据划分到多个磁盘或节点上，进而限制故障的影响范围。如果数据根据地理位置进行了划分，那么即使是通信网络或者是远程节点无法工作，本地用户还是可以访问本地数据。此外，[Gray]中也给出了一些通过数据分区来获取更高可用性的系统实例。

Summary

计算机系统会因为各种原因而失败。采用传统设计方案的大规模计算系统每隔几周就会因为软件、操作失误、硬件等方面的原因发生一次故障。与之相比，大规模容错系统的 MTBF 要高几个数量级—通常是以年为单位而不是周。

用于容错性硬件的技术都已经经过良好验证，非常成功。即使在高可用系统中，硬件也只是造成系统不可用的很次要的因素。

通过将实现容错性硬件的这些概念应用到软件构建过程中，可以将软件的 MTBF 提升多个数量级。这些概念包括：模块化、防御式编程、process-pairs、容忍软件错误—Heisenbugs。

事务+持久化的 process-pairs 提供了可容错的执行过程。事务+可恢复的通信会话提供了可容错的通信。事务+数据备份提供了容错性存储。此外，事务的原子性还会对数据库变更，通信网络以及执行进程进行协调。这都简化了高可用软件的构建过程。

但是系统的配置、管理以及维护仍然是一个未解决的问题。目前管理维护人员实际要比我们想象中的做地好很多了。但是我们不应该期望可以找到更好的人，而是应该简化和降低在系统交互中所需的人为因素。

Acknowledgments

The following people helped in the analysis of the Tandem system failure statistics: Robert Bradley, Jim Enright, Cathy Fitzgerald,Sheryl Hamlin, Pat Helland, Dean Judd, Steve Logsdon, Franco Putzolu,Carl Niehaus, Harald Sammer, and Duane Wolfe. In presenting the analysis, I had to make several outrageous assumptions and "integrate" contradictory stories from different observers of the same events.For that, I must take full responsibility. Robert Bradley, Gary Gilbert, Bob Horst, Dave Kinkade, Carl Niehaus, Carol Minor, Franco Putzolu, and Bob White made several comments that clarified the presentation. Special thanks are due to Joel Bartlett and especially Flaviu Cristian who tried very hard to make me be more accurate and precise.

References

[Adams] Adams,Products" ,E., "Optimizing Preventative Service of Software IBM J. Res. and Dev., Vol. 28, No.1, Jan. 1984.Production Online High Performance
[Bartlett] Bartlett, J.,"A NonStop Kernel," Proceedings of the Eighth Symposium on Operating System Principles, pp. 22-29, Dec. 1981.
[Borg] Borg, A., Baumbach, J., Glazer, S., "A Message System Supporting Fault-tolerance", ACM OS Review, Vol. 17, No.5, 1984.
[Borr 81] Borr, A., "Transaction Monitoring in ENCOMPASS," Proc. 7Th VLDB, September 1981. Also Tandem Computers TR 81.2.
[Borr 84] Borr, A., "Robustness to Crash in a Distributed Database: A Non Shared-Memory Multi-processor Approach," Proc. 9th VLDB,Sept. 1984. Also Tandem

Computers TR 84.2.
[Burman] Burman, M. "Aspects of a High Volume Banking System", Proc. Int. Workshop on Transaction Systems, Asilomar, Sept. 1985.
[Cristian] Cristian, F., "Exception Handling and Software Fault Tolerance", IEEE Trans. on Computers, Vol. c-31, No.6, 1982.
[Enright] Enright, J. "DP2 Performance Analysis", Tandem memo, 1985.
[Gray] Gray, J., Anderton, M., "Distributed Database Systems Four Case Studies", to appear in IEEE TODS, also Tandem TR 85.5.
[Haeder] Haeder, T., Reuter, A., "Principals of Database Recovery", ACM Computing Surveys,1983.Transaction-Oriented Vol. 15, No.4. Dec.
[Kim] Kim, W., "Highly Available Systems for Database Applications",ACM Computing Surveys, Vol. 16, No.1, March 1984
[Lampson] Lampson, B.W. ed, Lecture Notes in Computer Science Vol.106, Chapter 11, Springer Verlag, 1982.
[Mourad] Mourad, S. and Andrews, D., "The Reliability of the Operating System", Digest of 15th Annual Int. Sym. On Tolerant Computing, June 1985. IEEE Computer Society Press.IBM/XA
[Schlichting] Schlichting, R.D., Schneider, F.B.,Processors, an Approach to Designing Fault-Tolerant Systems", ACM TOCS, Vol. 1, No.3, Aug. 1983."Fail-Stop Computing
[Spector] "Multiprocessing Architectures for Local Computer Networks",PhD Thesis, STAN-CS-81-874, Stanford 1981.34
[von Neumann] von Neumann, **J.** "Probabilistic Logics and the Synthesis of Reliable Organisms From Unreliable Components", Automata Studies, Princeton University Press, 1956.

译考文献

JAMES ("JIM") NICHOLAS GRAY，英文版
防止应用程序出现Heisenbug漏洞，英文版，中文版
面对软件错误构建可靠的分布式系统.pdf，中文版
Tandem，中文版
How Complex Systems Fail，英文版
Interesting bits from “Why Do Computers Stop and What Can Be Done About It?”，英文版
Paper Discussion” Why Do Computers Stop and What Can Be Done About It”，英文版
Fault Tolerance，英文版