|  |  |
| --- | --- |
| 学号 | 2017202110118 |
|  |

# 冗余并不意味着容错：分析分布式存储对单个错误和损坏的响应

|  |  |
| --- | --- |
| 院（系）名 称 ： | 计算机学院 |
| 专 业 名 称 ： | 网络空间安全 |
| 学 生 姓 名 ： | 苏成伟 |
| 指 导 教 师 ： | 何水兵 教授 |

### 摘要

我们分析了现代分布式存储系统在存储文件系统故障（如数据损坏和读写错误）时的表现。 我们评估了八个流行的分布式存储系统，并发现了许多与文件系统容错有关的漏洞。 我们发现，现代分布式系统并不总是使用冗余来从文件系统故障中恢复：单个文件系统故障可能导致灾难性后果，如数据丢失，损坏和不可用性。 我们的研究结果对下一代容错分布式和云存储系统的设计具有启发意义。

### 1引言

基于云的应用，诸如互联网搜索，照片和视频服务[19,65,67]，社交网络[90,93]，运输服务[91,92]和电子商务[52]等都依赖于现代分布式存储系统来管理他们的数据。这个重要的系统类包括键值存储（例如Redis），配置存储（例如ZooKeeper），文档存储（例如MongoDB），列存储（例如Cassandra），消息队列（例如Kafka）和数据库（例如，RethinkDB）。

现代分布式存储系统以复制的方式存储数据以提高可靠性。每个副本在单独硬件上的本地文件系统上工作，以存储和管理关键用户数据。在大多数情况下，复制可以屏蔽故障，例如系统崩溃，电源故障以及磁盘或网络故障[22,24,30,31,40,80]。不幸的是，诸如磁盘和闪存驱动器的存储设备表现出更复杂的故障模式，其中某些数据块可能变得不可访问（读取和写入错误）或者更糟糕，数据可能是默默地损坏[8，60，85]。这些复杂的故障被称为部分存储故障[63]。

先前的研究[10,63,98]已经显示了文件系统如ext3，NTFS和ZFS是如何处理部分存储故障的。在某些情况下，文件系统只是简单地将故障传播到应用程序;例如，如果底层设备块损坏，ext4将原样返回损坏的数据给应用程序。在其他情况下，文件系统会在错误传递给应用程序之前对错误做出反应并将其转换为另一种错误。例如，btrfs将底层的块损坏转换为读取错误。无论哪种情况，我们都将文件系统引发的故障称为文件系统故障。

现代分布式存储系统在处理文件系统故障时的表现至关重要，并极大地影响基于云的服务。尽管如此，对于现代分布式存储系统如何对文件系统故障作出反应的知之甚少。

一个普遍而广泛的预期是，在更高层（即跨越副本）的冗余能够从本地文件系统故障中恢复数据[12,22,35,41,81]。例如，理论上，分布式存储系统的一个节点中的不可访问的数据块将不会导致用户可见的数据丢失，因为相同的数据被冗余地存储在许多节点上。基于这个期望，在本文中，我们将回答以下问题：现代分布式存储系统在处理本地文件系统故障时的表现如何？他们使用冗余从文件系统故障中恢复吗？

为研究现代分布式存储系统如何应对本地文件系统故障，我们构建了一个名为CORDS的故障注入框架，其中包括以下关键部分：errfs，一个用户级的FUSE文件系统，可以注入文件系统故障，以及errbench，一套系统特定的工作负载，它驱动系统与本地存储进行交互。对于每个注入的故障，CORDS将自动观察系统的处理结果。我们研究了使用CORDS的8个广泛使用的系统：Redis [66]，ZooKeeper [6]，Cassandra [4]，Kafka [5]，RethinkDB [70]，MongoDB [51]，LogCabin [45]和CockroachDB [14]。

我们研究中最重要的教训是：在大多数现代分布式存储系统中，单个文件系统故障会导致灾难性后果。尽管在分布式存储中普遍存在校验和，冗余和其他弹性方法，但单个不合时宜的文件系统故障可能会导致数据丢失，损坏，不可用，并且在某些情况下将损坏扩散到其他完整的副本。

我们系统研究的好处是双重的。首先，我们的研究帮助我们描述了八个系统对文件系统故障的处理行为，并且还发现了这些广泛使用的系统中的许多错误。我们发现这些系统可以静默地将损坏的数据返回给用户，丢失数据，将损坏的数据传播到完整的副本，变得不可用或查询时返回不期待的错误。例如，日志初始化期间的单个写入错误可能会导致ZooKeeper中写入不可用。同样，Redis和Cassandra中的一个节点中的损坏数据可以传播到其他完整的副本。在Kafka和RethinkDB中，一个节点上的损坏会导致用户可见的数据丢失。

其次，我们的研究使我们能够观察在所有系统中对文件系统故障进行的处理。具体来说，我们首先发现系统采用了不同的数据完整性策略;一些系统小心地使用校验和，而另一些则完全信任堆栈中的较低层来检测和处理损坏。其次，故障在本地通常是不被发现的，即使被发现，崩溃也是最常见的反应。一个节点上未检测到的故障可能导致有害的全局效应，如用户可见的数据损坏。第三，如上所述，单一故障可能会造成灾难性的集群范围效应。尽管分布式存储系统跨多个节点复制数据和功能，但单个节点上的单个文件系统故障可能会导致有害的集群范围影响;令人惊讶的是，许多分布式存储系统并不总是使用冗余作为恢复的来源。第四，崩溃和损坏处理纠缠在一起;系统通常会从崩溃和损坏的混合状态中恢复过来，意外地调用错误的恢复子系统来处理错误，最终导致糟糕的结果。最后，通常使用的分布式协议中的细微差别可能会传播损坏或数据丢失;例如，我们发现执行分布式协议（如领导选举，读取修复和重新同步）的细微之处可能会传播损坏或数据丢失。

本文包含三个主要贡献。首先，我们建立一个故障注入框架（CORDS），将文件系统故障注入应用程序（§3）。其次，我们介绍了八种广泛使用的现代分布式存储系统的行为研究，以了解它们如何对文件系统故障作出反应，并揭示这些存储系统中的许多错误（§4.1）。我们已经联系了七个系统的开发者，其中五个已经认识到我们发现的问题。虽然应用级别的修复可以容忍一些问题，但容忍许多其他问题需要进行基本的设计更改。第三，我们从所有系统中得出一系列观察结果，显示了一些常见的数据完整性和错误处理问题（§4.2）。我们的测试框架和我们报告的错误是公开的[1]。我们希望我们的结果能够引发讨论和未来研究，以提高下一代云存储系统的弹性。本文的其余部分安排如下。 首先，我们提供文件系统故障的背景，并表明处理文件系统故障在现代分布式存储系统（§2）中的重要性。 然后，我们描述我们的错误模型，以及我们的框架如何注入错误和观察行为（§3）。 接下来，我们介绍我们的行为分析和系统观察（§4）。 最后，我们讨论相关的工作（§5）并结束（§6）。

### 2背景和动机

我们首先介绍为什么在文件系统上运行的应用程序在读取和写入等操作期间可能遇到错误。接下来，我们表明为什么这样的文件系统故障在分布式存储系统的环境中很重要，以及这些系统的端到端的数据完整性和错误处理的必要性。

#### 2.1文件系统故障

文件系统下的存储堆栈中的层由许多复杂的硬件和软件组件组成[2]。栈的底部是媒体介质（磁盘或闪存设备）。媒体上方的固件控制媒体的功能。固件的命令由设备驱动程序提供。文件系统可能会遇到多种潜在原因的错误，包括介质错误，磁盘中的机械和电气问题，固件中的错误以及总线控制器中的问题[8,9,48,54,63,79,81]。有时，由于操作系统[13]，设备驱动程序[88]的其他部分中的软件错误，有时甚至由于文件系统自身的错误[26]，可能会导致错误。

由于这些原因，文件系统出现两个问题：块错误，某些块不可访问（也称为潜在扇区错误）和块损坏，其中某些块不包含预期的数据。

文件系统可以在磁盘返回一个明确的错误时，检测到被访问的块的某些问题（例如磁盘中的ECC警告该块有一个位错误），从而观察到块错误[9,79]。以前在32个月内超过100万个磁盘驱动器的研究表明，8.5％的近线磁盘和1.9％的企业级磁盘产生了一个或多个潜在扇区错误。最近的结果显示，在基于闪存的SSD中也会出现类似的错误[48,54,81]。

由于驱动器固件中的错误导致写入错误或丢失，文件系统可能会收到损坏的数据[8,60]，或者磁盘内ECC未检测到位错误。由于块以磁盘本身检测不到的方式损坏，因此块的损坏是悄无声息的。在许多情况下，文件系统会不知不觉地访问这些损坏的块，并将其无声地返回给应用程序。 Bairavasundaram等人在对超过41个月的151万个磁盘驱动器的研究中发现，超过40万个块有校验和不匹配[8]。传闻证据表明存储错误和损坏现象普遍存在[18,37,75]。考虑到存储损坏和错误的频率，文件系统遇到这种错误的概率是不可忽视的。

在许多情况下，当文件系统遇到来自其底层的故障时，它只是将它原样传递到应用程序[63]。例如，默认的Linux文件系统ext4只是简单地将错误或损坏的数据返回给应用程序，当底层的块不可访问或损坏时。在另一些情况下，文件系统可能会将潜在的故障转换为另一种故障。例如，btrfs和ZFS将基础损坏转换为错误 - 当访问底层损坏的磁盘块时，应用程序将收到错误而不是损坏的数据[98]。无论哪种情况，我们都将文件系统引发的这些故障称为文件系统故障。

#### 2.2为什么要用分布式存储系统？

鉴于本地文件系统可能会返回损坏的数据或错误，数据完整性和适当的错误处理的责任就落在了应用程序身上，因为他们关注的是安全存储和管理关键用户数据。大多数单机应用程序（如独立数据库和非复制键值存储系统）完全依赖本地文件系统来可靠地存储用户数据;他们很少有办法从本地文件系统故障中恢复。例如，在读取时，如果本地文件系统返回错误或损坏的数据，则应用程序无法恢复该段数据。他们最好的行动方式是可靠地检测到这些故障，并向用户发送适当的错误信息。

与单机应用程序类似，现代分布式存储系统也依靠本地文件系统来安全地管理关键用户数据。但是，与单机应用不同，分布式存储系统以复制的方式存储数据。精心设计的分布式存储系统可以使用冗余来从错误和损坏中恢复，而不管其本地文件系统提供的支持。理想情况下，即使一个副本损坏，整个分布式存储系统也不应受到影响，因为其他副本上存在相同数据的完整副本。同样，一个节点中的错误不应该影响系统的全局可用性，因为功能（应用程序代码）也在许多节点上复制。

端到端数据完整性和错误处理的情况可以在系统设计中的经典端到端参数中找到[78]。 Ghemawat等人也描述了在Google文件系统中需要这种基于端到端校验和的检测和恢复，因为底层便宜的IDE磁盘通常会破坏块服务器中的数据[29]。 同样，Google [22]在构建大型互联网服务方面的经验教训强调了高层软件应该如何提供可靠性。 考虑到分布式系统的端到端数据完整性和错误处理的可能性，我们研究了现代分布式存储系统是否采用端到端技术以及如何从本地文件系统故障中恢复。

### 3测试分布式系统

正如我们在前面部分讨论的那样，文件系统可能会抛出错误或将损坏的数据返回到在其上运行的应用程序; 强大的应用程序需要能够处理这样的文件系统故障。 在本节中，我们首先讨论我们的文件系统故障模型。 然后，我们描述我们的方法，注入由我们的模型定义的故障，并观察注入故障的影响。

#### 3.1故障模型

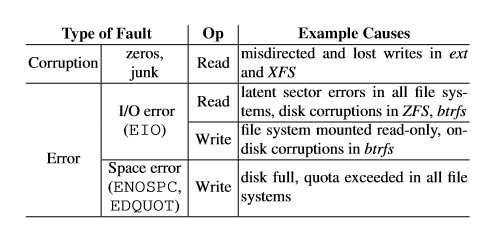
我们的故障模型定义了应用程序可能遇到的文件系统故障条件。 我们的模型的目标是注入当前和未来文件系统中代表故障条件的故障，并将分布式存储系统驱动到错误情况中。

我们的错误模型有两个重要特征。 首先，我们的模型考虑在单个节点中一次只向单个文件系统块注入单一故障。尽管多个相关的文件系统故障[8,9]是有趣的，但我们关注的是在单个节点中注入单个故障的最基本的情况，因为我们的故障模型旨在为应用程序提供最大的恢复余地。另一方面，多个相关的错误可能排除这样的余地。

其次，我们的模型仅将错误注入到应用程序级别的磁盘结构中，而不是文件系统元数据。文件系统可能能够保护自己的（元）数据[27];但是，如果用户数据损坏或无法访问，则应用程序将收到损坏的块或者可能收到错误（如果文件系统具有用户数据的校验和）。因此，应用程序处理这种情况是至关重要的。

表1显示了我们的模型在读取和写入操作期间可能出现的故障，以及最常用的文件系统中可能导致特定故障的根本原因的一些示例。对于所有进一步的讨论，我们使用术语块来表示文件系统块。

表1：可能的故障和例子原因



如果以前对该块的写入丢失或某个不相关的写入被误导到该块，则应用程序可能会读取已损坏的块（带有零或垃圾）。例如，在文件系统和XFS的ext系列中，没有用户数据的校验和，因此应用程序可以读取这种损坏的数据，而不会出现任何错误。我们的模型通过破坏读取中的零或垃圾块来捕获这种情况。

即使在对用户数据进行校验和的文件系统（如btrfs和ZFS）上，也可能检测到损坏，但不能恢复（除非在ZFS中使用诸如copies = 2的特殊选项进行安装）。尽管btrfs和ZFS使用的用户数据校验和可以防止应用程序访问损坏的数据，但是当应用程序访问损坏的数据块时，它们会返回错误。我们的模型通过在读取上返回类似的错误来捕获这种情况。而且，当读取的数据存在潜在的扇区错误时，应用程序可以在读取时接收EIO。这种情况在所有常用的文件系统上是可能的，包括ext4，XFS，ZFS和btrfs。

如果底层磁盘扇区不可写，并且磁盘没有重新映射扇区，文件系统以只读模式挂载，或者正在写入的文件已经在btrfs中损坏，则应用程序可以接收来自文件系统写入的EIO。对于需要额外空间的写入（例如，将新块添加到文件中），如果底层磁盘已满或者用户的块配额已耗尽，则应用程序可以在任何文件系统上分别接收ENOSPC和EDQUOT。

我们的错误模型以我们认为是现实的方式注入错误。例如，如果一个块标记为损坏被写入，随后读取该块将看到最后写入的数据，而不是损坏的数据。 同样，当一个块被标记为读取或写入错误，并且如果该文件被删除并重新创建（可能分配新的数据块），我们不会为随后读取或写入该块而返回错误。 同样，当空间错误返回时，所有需要额外空间的后续操作将遇到相同的空间错误。

#### 3.2方法

我们现在描述我们的方法来研究分布式系统如何对本地文件系统故障做出反应。我们构建了CORDS，一个包含errfs，FUSE [28]文件系统和errbench的故障注入框架，一组工作负载和每个系统的行为推理脚本。

##### 3.2.1系统工作负载

为了研究分布式存储系统如何对本地文件系统故障做出反应，我们需要运行其代码，以便与本地文件系统进行交互。我们为此制作了工作量套件errbench;我们的套件包含每个系统的两个工作负载：读取现有的数据项，插入或更新数据项。

##### 3.2.2故障注入

我们通过插入一些数据项将正在研究的系统初始化为已知状态，并确保它们被安全地复制并保存在磁盘上。我们的工作负载读取或更新插入的项目作为初始化的一部分。接下来，我们将应用程序配置为通过指定其挂载点作为应用程序的数据目录在errfs上运行。因此，应用程序执行的所有读写操作都会通过errfs进行，从而可以注入故障。我们多次运行应用程序工作负载，每次通过errfs为单个文件系统块注入单个故障。 errfs可以注入两种类型的损坏：用零或垃圾损坏。对于损坏，errfs执行读取操作，并在返回到应用程序之前更改标记为损坏的块的内容。 errfs可能会注入三种类型的错误：EIO读取（读取错误），EIO写入（写入错误）或ENOSPC和EDQUOT写入，需要额外的空间（空间错误）。为了模拟错误，errfs不执行操作，而是简单地返回一个适当的错误代码。

##### 3.2.3行为推断

对于注入单个故障的每个工作负载，我们观察系统运行的行为。我们的系统特定行为推理脚本从系统的日志文件和客户端可见输出（如服务器状态，返回码，错误（stderr）和输出消息（stdout））中收集系统行为。一旦已知注入故障的系统行为，我们将观察到的行为与预期的行为进行比较。以下是我们测试的预期行为：

•承诺的数据不应该丢失

•查询不应该默默地返回损坏的数据

•集群应该可用于读取和写入

•重试后查询不应该失败

我们相信我们的预期是合理的，因为在分布式系统的单个节点中的单个故障理想情况下不会导致任何不良行为。如果我们发现观察到的行为与预期不符，我们会将特定的运行（工作负载和注入的错误的组合）标记为错误，分析相关的应用程序代码，联系开发人员和文件错误。

局部行为和全球效应。在分布式系统中，多个节点使用本地文件系统来存储用户数据。当一个节点注入故障时，需要注意两点：注入故障的节点的局部行为和故障的全局效应。

在大多数情况下，节点在本地对注入的故障作出反应。如图1的图例所示，由于注入故障，节点可能会崩溃或部分崩溃（只有少数线程的进程被终止）。在某些情况下，节点可以通过重试任何失败的操作或通过使用内部冗余数据（复制副本中的相同冗余数据的情况）来解决问题。或者，节点可以检测并忽略损坏的数据或只记录错误消息。最后，节点甚至可能不能检测或采取任何措施来防止故障。

故障的全局效应是外部可见的结果。全局效应取决于分布式协议（如领导者选举，共识，恢复，修复）如何响应故障节点的本地行为。例如，即使一个节点可以在本地忽略损坏的数据并丢弃它，全局恢复协议也可以解决这个问题，从而导致一个正确的外部可观察的行为。有时，由于分布式协议如何反应，全局损坏，数据丢失，读取不可用性，写入不可用性，不可用性或查询失败是可能的。当一个节点作为局部反应简单地崩溃时，系统以减少的冗余运行，直到手动干预。

对于给定的工作量和故障，这些本地行为和全局效应可能会因故障注入节点所扮演的角色（领导者或追随者）而异。为了简单，我们统一使用术语领导者和追随者，而不是主从。

我们在这里注意到，我们的工作负载套件和模型并不完整。首先，我们的套件只包含简单的读取和写入工作负载，而更复杂的工作负载可能会产生额外的见解。其次，我们的模型不会注入所有可能的文件系统错误;相反，它只注入了诸如损坏，读取，写入和空间错误等一系列错误。但是，即使我们简单的工作负载和故障模型也会导致系统陷入极端情况，导致有趣的行为。我们的框架可以扩展到包含更复杂的错误，我们的工作负载套件可以增加更复杂的工作负载;我们把它作为未来工作的一个途径。

### 4结果和观察

我们研究了八个广泛使用的分布式存储系统：Redis（v3.0.4），ZooKeeper（v3.4.8），Cassandra（v3.7），Kafka（v0.9），RethinkDB（v2.3.4），MongoDB（v3.2.0），LogCabin（v1.0）和CockroachDB（beta20160714）。我们配置所有系统以提供最高的安全保证。我们启用了校验和，同步复制和同步磁盘写入。我们将所有系统配置为形成三个节点的集群，并将复制因子设置为三。

我们将结果分为四部分。首先，我们提供我们详细的行为分析和每个系统的定性总结（§4.1）。其次，我们推导并提出了一套与所有八个系统（§4.2）中的数据完整性和错误处理相关的观察结果。接下来，我们讨论当前文件系统的特性，这些特性可以影响我们发现的问题（§4.3）。最后，我们讨论为什么现代分布式存储系统不能容忍单个文件系统故障，并描述我们与开发人员交互的经验（§4.4）。

#### 4.1系统行为分析

图1显示了当故障注入不同的磁盘结构时，所有系统的行为。右侧显示的ondisk结构，名称采用以下格式：file name.logical entity。我们从我们对文件的磁盘格式的理解中推导出逻辑实体名称。如果一个文件可以包含在一个文件系统块中，我们不会显示逻辑实体名称。解读图1：我们引导读者阅读一个系统的几个结构（Redis）的图的相关部分。当appendonly文件中的元数据结构中存在损坏或访问错误时，该节点会简单地崩溃（Redis中两个工作负载的第一行本地行为框）。如果领导者崩溃，则集群变得不可用，并且如果追随者崩溃，则集群以减少的冗余（两个工作负载的第一行全局效应）运行。未检测到appendonly文件中用户数据的损坏（两个工作负载的第二行本地行为）。如果领导者被破坏，会导致全局的用户可见损坏，如果追随者损坏，则没有有害的全局效应（读取工作负载的第二行全局效应）。相比之下，附加文件用户数据中的错误导致崩溃（两个工作负载的本地行为的第二行）;领导者和追随者的崩溃分别导致集群不可用性和冗余度降低（两种工作负载的全局效应的第二行）。

我们接下来定性总结图1中每个系统的结果。

##### 4.1.1 Redis

Redis是一个流行的数据结构存储，用作数据库，缓存和消息代理。 Redis使用一个简单的appendonly文件（aof）来记录用户数据。从aof获取定期快照以创建redis数据库文件（rdb）。启动期间，追随者重新同步领导者的rdb文件。当前领导失败时，Redis不会自动选举领导。

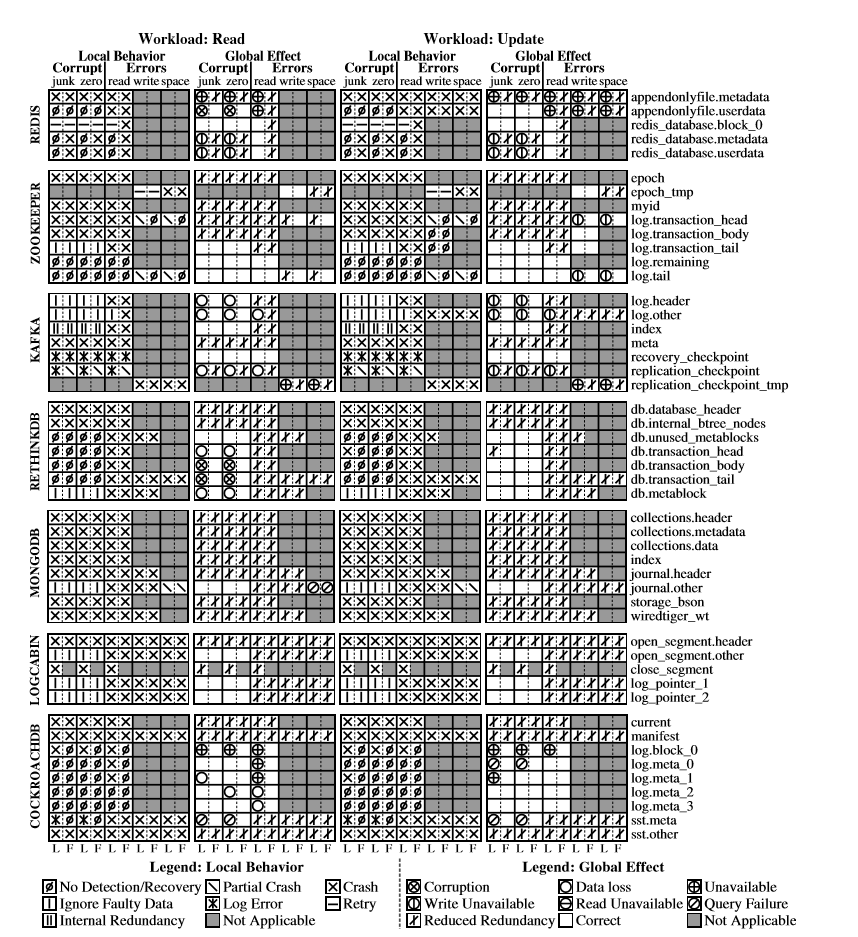
总结和错误：Redis不针对aof用户数据使用校验和;因此，它不检测损坏。图2（a）显示了重新同步协议如何在领导者和追随者之间传播损坏的用户数据，导致全局用户可见的损坏。如果追随者被破坏，同样的协议无意中通过从领导者获取数据来修复损坏。 aof中的元数据结构中的损坏和aof中的错误导致其崩溃，从而使集群不可用。由于领导者在重新同步期间发送rdb文件，同样的损坏会导致追随者崩溃。这些崩溃最终使集群不可用于写入。

##### 4.1.2 ZooKeeper

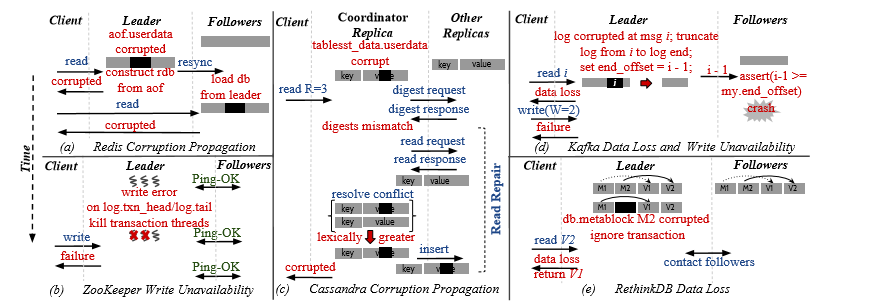
ZooKeeper是一种用于存储配置信息，命名和分布式同步的流行服务。它使用日志文件来附加用户数据;日志的第一个块包含一个头，第二个包含事务正文，第三个包含事务尾以及ACL和其他信息。

总结和错误：ZooKeeper可以使用校验和检测日志中的损坏，但是只是通过崩溃来作出反应。同样，它在大多数错误情况下崩溃，导致冗余减少。在所有的崩溃情况下，ZooKeeper都可以可靠地选出新的领导者，从而确保可用性。 ZooKeeper在尾部损坏时会在本地忽略事务;领导选举协议防止该节点成为领导者，避免不良行为。最终，受损的节点通过联系领导来修复其日志，从而导致正确的行为。

不幸的是，ZooKeeper不能将写入错误恢复到事务头和日志尾（图1 - ZooKeeper中的第4和第8行）。图2（b）描述了这种情况。在日志初始化期间发生写入错误时，错误处理



图一：系统行为



图二：错误实例

代码尝试正常关闭节点，但仅杀死事务处理线程;仲裁线程保持活动（部分崩溃）。因此，其他节点认为领导者是健康的，不会选出新的领导者。但是，由于领导者已经部分崩溃，所以

不能提出任何交易，导致无限期的写入不可用。

##### 4.1.3 Cassandra

Cassandra是Dynamo-like [23] NoSQL商店。用户数据表（tablesst）和系统模式（schemasst）都使用日志结构化合并树[59]的变体来存储。与我们研究的其他系统不同，Cassandra没有领导者和追随者;相反，节点形成一个环。因此，我们在图3中分别显示它的行为。

总结和错误：Cassandra仅对用户数据启用校验和验证，作为启用压缩的副作用。关闭压缩时，不会在用户数据（表格数据）上检测到损坏。在读取查询中，协调器节点收集并比较来自R副本[20]的数据的摘要（哈希）。如果摘要不匹配，则使用最新的时间戳获取策略解决值中的冲突。如果时间戳之间存在联系，则选择词汇最大值并将其安装在其他副本上[38]。如图2（c）所示，在R = 3时，如果损坏值在词汇上大于原始值，则将损坏的值返回给用户，并将损坏传播到其他完整的副本。另一方面，如果损坏的值在词汇上较少，则修复损坏的节点。读取R = 1的受损节点总是返回损坏的数据。

tablesst索引中的故障导致查询失败。模式数据和模式索引中的故障会导致节点崩溃，使得R = 3和W = 3的读取和写入不可用。其他架构文件中的错误导致查询失败。在大多数情况下，即使在R = 3的情况下运行，在R = 1配置中观察到的用户可见问题也不会被修复。

##### 4.1.4 Kafka

Kafka是一个分布式持久消息队列，客户可以在其中发布和订阅消息。它使用日志来追加新消息，每个消息都进行校验和。它维护一个索引文件，将消息索引到日志中的字节偏移量。复制检查点和恢复检查点分别指示到目前为止向追随者复制了多少个消息以及到目前为止将多少消息刷新到磁盘。

总结和错误：在读写错误时，Kafka大多数是崩溃的。图2（d）显示了Kafka可能会丢失数据并且无法写入的情况。当一个日志条目在领导上被破坏（图1 - Kafka中的第一行和第二行）时，它会在本地忽略该条目以及日志中的所有后续条目。领导者然后指示追随者也这样做。在接到这个指示后，追随者发出了一个致命的宣言，接着就崩溃了。一旦追随者崩溃，集群变得不可用于写入，数据也会丢失。指数损坏是使用内部冗余来确定的。领导者的复制检查点中的故障导致数据丢失，因为领导者无法记录追随者的复制偏移。当领导者分别不能读取或写入复制检查点和复制检查点tmp时，Kafka变得不可用

##### 4.1.5 RethinkDB

RethinkDB是一个分布式数据库，适合将查询结果推送到实时Web应用程序。它使用一个持久的B-树来存储所有的数据。 B树中的元块指向构成数据库的当前版本和先前版本的数据块。在更新期间，首先仔细地写入新的数据块，然后将具有校验和的元块更新为指向新的块，从而实现原子更新。

总结和错误：在数据库头和内部B树节点的任何错误，RethinkDB只是崩溃。如果领导崩溃，则自动选出新领导。 RethinkDB依靠文件系统来确保数据块的完整性;因此，它不检测事务主体和尾部中的损坏（图1 - RethinkDB中的第5和第6行）。当领导者的这些块被损坏时，RethinkDB将自动返回损坏的数据。图2（e）描述了当指向事务的事务头或元块在领导者上被破坏时，数据是如何丢失的。即使追随者有相同数据的完整副本，但领导者并不修复其损坏或丢失的数据，即使我们执行多数选项的读取。当追随者受到损坏时，他们并没有通过联系领导来解决。虽然这不会导致用户可见的直接损坏或损失（因为领导者的数据是最终返回的数据），但是当损坏的追随者成为未来的领导者时，这种做法就是如此。

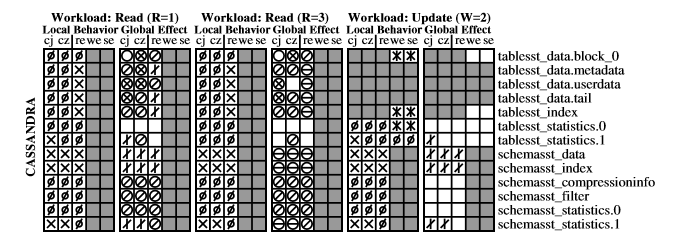


图3：系统行为：Cassandra

##### 4.1.6 MongoDB

MongoDB是一个流行的复制文档存储，使用下面的WiredTiger [53]进行存储。 当一个项目被插入或更新时，它首先被添加到日志中; 然后，它被检查点集合文件。

总结和错误：MongoDB在大多数错误上崩溃，导致冗余度降低。 如果当前的领导人崩溃，新的领导者会自动选出。 MongoDB使用所有文件的校验和; 任何文件的任何块中的损坏都会导致校验和不匹配，最终导致崩溃。 上述情况的一个例外是当日志头以外的块被破坏时（图1 - MongoDB中的第六行）。 在这种情况下，MongoDB检测并忽略损坏的块; 然后，损坏的节点截断其损坏的日记，下降成为一个追随者，并最终通过联系领导修理其日记。 在附加到日志中的情况下存在空间错误的情况下，查询失败。

##### 4.1.7 LogCabin

LogCabin使用Raft共识协议[56]为其他系统提供复制和一致的数据存储来存储其核心元数据。它实现了一个分段日志[77]，每个段是文件系统上的一个文件。当前的开放段被充分利用时，它被关闭，并且新的段被打开。两个指针文件指向日志的最新两个版本。它们交替更新;当指针文件被部分更新时，LogCabin使用指向日志的略微旧但一致的版本的另一个指针文件。

总结和错误：LogCabin在所有读取，写入和空间错误中崩溃。同样，如果一个封闭段中的一个开放段文件头或块被破坏，LogCabin就会崩溃。 LogCabin使用校验和识别打开段中的任何其他块中的损坏，并通过简单地放弃和忽略损坏的条目以及该段中的所有后续条目（图1 - LogCabin中的第二行）作出反应。如果日志指针文件损坏，LogCabin会忽略该指针文件并使用其他指针文件。

在上述两种情况下，领导选举协议确保了受损节点不成为领导;损坏的节点成为追随者，并通过联系新的领导者来修复其日志。这确保在任何故障情况下，LogCabin不会全局损坏或丢失用户数据。

##### 4.1.8 CockroachDB

CockroachDB是一个SQL数据库，用于解决磁盘，机器和数据中心的故障。它使用下面的RocksDB的调整版本进行存储;存储引擎是将传入数据附加到持久日志的LSM树;然后定期压缩内存中的数据以创建sst文件。清单和当前文件指向数据库的当前版本。

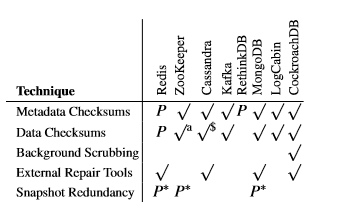
总结和错误：大多数情况下，CockroachDB只是在任何数据结构上的损坏崩溃和错误崩溃，导致冗余减少。领导者日志文件中的错误有时会导致整个集群不可用，因为一些追随者在领导者崩溃之后也会崩溃。其他一些日志元数据中的损坏和错误可能导致数据丢失，CockroachDB默默返回零行。 sst文件中的损坏和几块日志元数据的损坏导致查询失败，并显示错误消息，例如表不存在或者数据库不存在。总的来说，我们发现CockroachDB在故障处理方面有很多问题。然而，由于CockroachDB仍处于积极的发展阶段，未来可靠性可能会提高。

#### 4.2系统观察

我们现在提出一组关于所有八个系统的数据完整性和错误处理的观察结果。

＃1：系统采用不同的数据完整性策略。表2显示了现代分布式存储系统为确保数据完整性所采用的不同策略。如图所示，系统使用一系列技术来检测并从损坏中恢复。该表还显示了系统之间的差异。在频谱的一端，有一些系统试图通过使用校验和（例如ZooKeeper，MongoDB，CockroachDB）防止存储堆栈中的数据损坏，而另一端则包括完全信任并依赖较低层的系统在存储堆栈中处理数据完整性问题（例如，RethinkDB和Redis）。尽管采用了大量的数据完整性策略，但是所有系统都表现出不希望的行为

表2：数据完整性策略



P - 仅适用于某些磁盘结构; - Adler32校验和\* - 仅限于时间; $ - 未使用时的压缩关闭

有时，看起来不相关的配置设置会影响数据的完整性。例如，在Cassandra中，校验和仅被验证为启用压缩的副作用。由于这种行为，关闭压缩时无法检测或修复损坏，导致用户可见的损坏。

我们也发现一些系统使用不合适的校验和算法。例如，ZooKeeper使用Adler32，它仅适用于解压缩后的错误检测，并且可能会碰撞非常短的字符串[47]。在我们的实验中，我们能够注入造成校验和碰撞的损坏，使ZooKeeper能够处理损坏的数据。我们相信像ZooKeeper这样的元数据存储可靠地存储配置设置等小实体并不罕见。一般来说，我们认为需要更多的关注来了解可能的校验和选择的稳健性。

＃2：本地行为：故障往往未被发现;即使检测到，崩溃是最常见的局部反应。我们发现故障通常在本地不被发现。有时候，这会导致直接有害的全球效应。例如，在Redis，领导人附件中的损坏现象未被发现，导致全局性的损坏。此外，领导者的rdb中的损坏也未被发现，并且当被发送给追随者时，导致他们崩溃，导致无法使用。类似地，在Cassandra中，表格数据的损坏未被检测到，这导致向用户返回损坏的数据，并且有时将其传播到完整的副本。同样，RethinkDB也不会在领导者的事务头上检测到损坏，从而导致全球用户可见的数据丢失。同样，交易机构的损坏现象也未被发现，导致全局性的损坏。追随者也没有发现同样的错误;如果一个损坏的追随者将来成为领导者，那么全局数据丢失或损坏是可能的。

尽管有些系统有目的地检测到故障并作出反应，但有些系统只是作为副作用对故障做出反应。例如，ZooKeeper，MongoDB和LogCabin仔细检测和处理损坏。另一方面，Redis，Kafka和RethinkDB有时只是作为失败的反序列化的副作用而对损坏作出反应。

我们观察到碰撞是最常见的故障反应。当系统检测到错误或遇到错误时，它们就会崩溃，从图1的本地行为中大量的崩溃符号可以看出。虽然单个节点的崩溃并不会立即影响集群可用性，但总体不可用性即将到来，因为其他节点也可能随后失败。另外，即使一个节点崩溃，需要写入或读取所有副本的工作负载也不会成功。在崩溃之后，如果故障是粘滞的，简单地重新启动并不会起作用;该节点将重复崩溃，直到手动干预修复了潜在的问题。我们也观察到节点比错误更容易崩溃。

我们观察到，失败的操作很少重新尝试。尽管在使用重试的几种情况下重试有帮助，但我们观察到，有时无限期地重试操作可能导致更多的问题。例如，当ZooKeeper由于空间错误而无法写入新纪元信息（纪元tmp）时，它会删除并创建一个新文件，保持旧文件描述符的打开状态。由于ZooKeeper盲目地重试这个序列，并且由于空间错误是粘性的，节点很快就会耗尽描述符并崩溃，从而降低可用性。

＃3：冗余未充分利用：单一故障会造成灾难性的集群范围影响。与广泛的分布式系统冗余可以帮助从单个故障中恢复的预期相反，我们观察到即使是单个错误或损坏也会导致不利于集群的问题，例如总体不可用性，无声的损坏以及过多的数据丢失或无法访问。在很多情况下，几乎所有的系统都不使用冗余作为恢复的来源，并且错过了使用其他完整副本进行恢复的机会。请注意，我们在研究中发现的所有错误和不良行为都是由于一次只在单个节点中注入单个错误。鉴于数据和功能被复制，理想情况下，不应该表现出任何不良行为。

一些系统（MongoDB和LogCabin）通过利用其他副本自动从一些（不是全部）数据损坏中恢复。这种恢复涉及本地和分布式恢复行动之间的协同作用。具体而言，在遇到损坏的条目时，这些系统在本地忽略错误数据（本地恢复策略）。然后，领导者选举算法确保数据项已被损坏，因此被忽略的节点不成为领导者（全局恢复策略）。结果，损坏的节点最终通过从当前领导者获取损坏的数据来恢复损坏的数据。在很多情况下，即使这些系统也不能通过冗余来自动恢复。例如，LogCabin和MongoDB分别在闭合段或集合被破坏时简单地崩溃。

我们还发现，只有一小部分数据出错时，会影响过多的数据。表3显示了当一小部分数据发生故障时受到影响的不同范围。受影响的部分可能会无声无息或无法访问。例如，在Redis中，当appendonly文件中的元数据发生错误或者在appendonly文件数据中存在读写错误时，所有用户数据都可能无法访问。同样，在Cassandra中，当小部分数据出错时，整个表格可能无法访问。Kafka有时可能会丢失整个日志或从损坏的条目开始直到日志结束的所有条目。当RethinkDB的一小部分损坏或指向该事务的元区块损坏时，RethinkDB将丢失所有作为事务更新的数据。

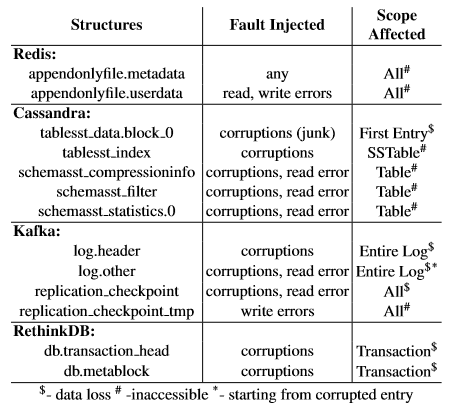
总之，我们发现冗余没有被有效地用作恢复的来源，并且冗余可以帮助实现功能和数据的一般期望没有实现。

＃4：崩溃和损坏处理纠缠在一起。我们发现许多系统的检测和恢复代码经常无意中尝试检测和修复两个根本不同的问题：崩溃和数据损坏。

存储系统实现崩溃一致的更新协议（即使在更新期间崩溃的情况下，数据应该总是可恢复的，不应该损坏或丢失）[7,61,62]。为此，系统谨慎地命令写入并使用校验和来检测部分更新的数据或由于崩溃而可能发生的损坏。在检测到由于损坏导致的校验和不匹配时，所有系统总是运行崩溃恢复代码（即使损坏实际上不是由于崩溃而是由于存储堆栈中的实际损坏），最终导致不希望的效应，诸如数据丢失。

这个问题的一个典型例子是RethinkDB。 RethinkDB不使用应用程序级校验和来处理损坏。但是，它为其元区块使用校验和来从崩溃中恢复。每当元区块损坏时，RethinkDB检测元区块校验和不匹配并调用其崩溃恢复代码。崩溃恢复代码认为系统在最后一个事务提交时崩溃。因此，它会回滚承诺和已经确认的交易，导致数据丢失。同样，当Kafka中的日志被损坏时，恢复代码将损坏视为崩溃的信号;因此，它会截断并丢失日志中的所有其他数据，而不是仅修复损坏的条目。这个问题的根本原因是无法区分由于实际存储堆栈损坏导致的崩溃。

表3 受影响的范围



LogCabin尝试使用以下逻辑将崩溃与损坏区分开来：如果封闭段（已满的段）中的块已损坏，则会正确地将该问题标记为损坏，并通过简单的崩溃作出反应。另一方面，如果开放段中的块（仍然用于持久化事务）已损坏，则将其检测为崩溃并调用其通常的崩溃恢复过程。 MongoDB也以类似的方式区分了日志损坏中的收益损坏。即使是试图辨别损坏事件的系统也并不总是这样做。

纠结检测和恢复崩溃和损坏的重要后果。在发生损坏（崩溃）恢复期间，有些系统会提取过多的数据来解决问题。例如，当日志条目在LogCabin和MongoDB中损坏时，他们可以通过联系其他副本来修复损坏的日志。不幸的是，他们通过忽略损坏的条目和所有后续条目直到日志结束并随后获取所有被忽略的数据而​​不是仅仅获取损坏的条目来这样做。由于在上次提交事务期间发生了崩溃，因此这些系统会假定损坏的条目是日志中的最后一个条目。同样，Kafka的追随者也从领导者获取额外的数据，而不是只有损坏的条目。

＃5：常用分布式协议的细微差别可能会导致数据损坏或数据丢失。我们发现，在执行常用的分布式协议，如领导选举，readrepair [23]和重新同步的细微之处，可以传播损坏或数据丢失。

例如，在Kafka，一个节点的本地数据丢失可能导致全局数据丢失，这是由于其领导选举协议中的细微不同之处。 Kafka维护一套insync-replicas（ISR），并且这个集合中的任何节点都可以成为领导者。当日志条目在Kafka节点上损坏时，它会忽略日志中的当前和所有后续条目，并截断日志直到最后一个正确的条目。从逻辑上讲，现在这个节点不应该成为ISR的一部分，因为它已经丢失了一些日志条目。然而，这个节点并没有从ISR中移除，因此最终仍然可以成为领导者，并且静静地丢失数据。这种行为与ZooKeeper，MongoDB和LogCabin的领导选举协议相比，忽略日志条目的节点不会成为领导者。

在Dynamo风格的仲裁系统中使用了修复协议来修复任何具有陈旧数据的副本。在读取请求上，协调器从配置数量的副本中收集正在读取的数据的摘要。如果所有摘要匹配，则简单地返回协调器的本地数据。如果摘要不匹配，则应用内部冲突解决策略，并将解析的值安装在副本上。在Cassandra，实施读修复，冲突决议解决词汇更大的价值;如果注入的损坏字节在词汇上大于原始值，则损坏的值将传播到所有其他完整的副本。

类似地，在Redis中，当数据项在领导者上被破坏时，它不被检测到。随后，再同步协议将损坏的数据传播给领导者的追随者，覆盖追随者上存在的正确版本的数据。

#### 4.3文件系统的影响

我们发现的所有错误都可能发生在XFS和所有的ext文件系统上，包括缺省的Linux文件系统ext4。鉴于这些文件系统通常用作大型分布式存储部署副本中的本地文件系统，并由开发人员推荐[50,55,64,76]，我们的研究结果对这种真实世界的部署具有重要意义。

文件系统（如btrfs和ZFS）使用用户数据的校验和;在检测到损坏时，它们返回一个错误，而不是让应用程序静静地访问损坏的数据。因此，由于注入块损坏而发生的错误不会在这些文件系统上显示。我们还发现，在这样的文件系统上部署端到端校验和的应用程序出乎意料地导致了不良的交互。具体而言，应用程序由于错误比损坏更频繁地崩溃。在损坏的情况下，一些应用程序（例如LogCabin，ZooKeeper）可以使用校验和冗余来恢复，从而导致正确的行为;但是，当损坏转化为错误时，这些应用程序崩溃，导致可用性降低。

#### 4.4讨论

我们现在考虑为什么分布式存储系统不能容忍单个文件系统故障。在一些系统中（例如RethinkDB和Redis），我们发现主要的原因是他们期望底层的存储栈层可以可靠地存储数据。随着越来越多的部署迁移到可靠的存储硬件，固件和软件可能不成为现实的云，存储系统需要开始采用端到端的完整性策略。

接下来，我们认为分布式系统中的恢复代码没有经过严格测试，导致不良行为。尽管许多系统使用校验和和其他技术，但是执行此类机器的恢复代码却没有经过仔细的测试。我们主张未来的分布式系统需要使用像我们这样的故障注入框架来严格地测试故障恢复代码。

第三，虽然一系列的研究工作[25,79,83，84,94]和企业存储系统[49,57,58]提供了解决部分故障的软件指导方针，但这种智慧并没有被过滤到商品化的分布式存储系统。我们的研究结果为分布式系统提供了动力，以现有的研究工作为基础，实际上可以容忍崩溃以外的故障[17,44,97]。最后，虽然冗余被有效地用来提供改进的可用性，但是作为从文件系统和其他部分故障中恢复的来源，它仍然没有被充分利用。为了有效地使用冗余，首先，必须仔细设计虚拟磁盘数据结构，以便可以识别损坏的或不可访问的数据部分。接下来，损坏恢复必须从崩溃恢复中解耦，以仅修复数据的损坏或无法访问的部分。有时，如果完整的副本不可达，恢复损坏的数据可能是不可能的。在这种情况下，结果应该由设计来定义，而不是作为实现细节。

我们联系了系统开发人员，了解我们发现的行为。 RethinkDB和Redis依靠底层存储层来确保数据的完整性[68,69]。 RethinkDB打算将设计更改为将来包含应用程序级校验和，并更新文档以反映我们报告的错误[71,72]，直到此问题得到解决。他们也证实了损坏和崩溃处理纠缠[73]。

由CORDS发现的ZooKeeper中的写入不可用性错误是由真实世界的用户遇到的，并且最近已经修复[99,101]。 ZooKeeper开发人员提到，检测到损坏并不是一个有意识的设计决策[100]。 LogCabin的开发者也证实了在开放领域损坏和崩溃处理的纠缠;他们补充说，很难区分部分写作和开放段的损坏[46]。 CockroachDB和Kafka的开发者也回应了我们的错误报告[15,16,39]。

### 5相关工作

我们的工作建立在四个相关工作的基础上。存储堆栈中的损坏和错误：正如§2所讨论的，关于存储错误和损坏的详细研究[8,9,48,54,79,81]激发了我们的工作。故障注入：我们的工作与将故障注入系统并测试它们的稳健性有关[11,32,82,89]。有几项工作为分布式系统构建了通用故障注入器[21,36,86]。一些研究表明文件系统[10,63,98]和运行在它们上面的应用程序[87,97]如何对存储和内存故障做出特别的反应。我们的工作来自于两个工作机构，但是专注于测试分布式系统对存储故障的行为。我们相信我们的工作是第一个全面检查存储故障在多个分布式存储系统中的影响的。测试分布式系统：几个分布式模型检查器成功发现了分布式系统中的错误[34,43,95]。 CORDS暴露了模型检查程序无法发现的错误。模型检查器通常重新排列网络消息并注入崩溃来查找错误;他们不会注入存储相关的故障。类似于模型检测器，像Jepsen [42]这样的测试错误网络下的分布式系统的工具是对CORDS的补充。我们以前的工作[3]研究文件系统崩溃行为如何影响分布式系统。但是，这些故障只发生在与CORDS引入的块损坏和错误不同的故障。错误研究：最近的一些错误研究[33,96]已经对分布式系统中发现的常见问题提供了见解。袁等人，在他们的研究中，34％的灾难性失败是由于意料之外的错误状况。我们的结果也表明系统不能很好地处理读写错误;这种糟糕的错误处理在很多情况下会导致有害的全局效应。我们相信，错误研究和错误注入研究是相辅相成的。而错误研究表明通过检查导致在外遇到错误的事件序列来构建测试用例，像我们这样的错误注入研究集中于注入一种类型的错误，发现新的错误和设计缺陷。

### 6 结论

我们表明，对文件系统故障的容忍并不是根深蒂固在现代分布式存储系统中。这些系统没有配备有效地使用复制副本上的冗余来从本地文件系统故障中恢复;由于单个本地文件系统故障，用户可见的问题（如数据丢失，损坏和不可用）可能会出现。随着分布式存储系统成为存储关键用户数据的主要选择，仔细测试所有类型的故障非常重要。我们的研究是朝着这个方向迈出的一步，我们希望我们的工作将会带来更多的工作来构建下一代的容错分布式系统。

### 参考文献

1. Cords Tool and Results. [http://research.cs. wisc.edu/adsl/Software/cords/.](http://research.cs.wisc.edu/adsl/Software/cords/)
2. Ramnatthan Alagappan, Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Aws Albarghouthi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Beyond Storage APIs: Provable Semantics for Storage Stacks. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems (HOTOS’15)*, Kartause Ittingen, Switzerland, May 2015.
3. Ramnatthan Alagappan, Aishwarya Ganesan, Yuvraj Patel, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. ArpaciDusseau. Correlated Crash Vulnerabilities. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI ’16)*, Savannah, GA, November 2016.
4. Apache. Cassandra. [http://cassandra. apache.org/.](http://cassandra.apache.org/)
5. Apache. Kakfa. [http://kafka.apache.org/.](http://kafka.apache.org/)
6. Apache. ZooKeeper. [https://zookeeper. apache.org/.](https://zookeeper.apache.org/)
7. Remzi H. Arpaci-Dusseau and Andrea C. ArpaciDusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.91 edition, May 2015.
8. Lakshmi N. Bairavasundaram, Andrea C. ArpaciDusseau, Remzi H. Arpaci-Dusseau, Garth R. Goodson, and Bianca Schroeder. An Analysis of Data Corruption in the Storage Stack. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST ’08)*, San Jose, CA, February 2008.
9. Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *Proceedings of the 2007 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS ’07)*, San Diego, CA, June 2007.
10. Lakshmi N. Bairavasundaram, Meenali Rungta, Nitin Agrawal, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Analyzing the Effects of Disk-Pointer Corruption. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN ’08)*, Anchorage, Alaska, June 2008.