CuriOS: Improving Reliability through Operating System Structure 阅读报告

江梦　P14206009

本文介绍了一种新型的 OS 设计——CuriOS，一种通过控制操作系统的结构来提升系统可靠性的设计方法。发生在微内核操作系统服务上的错误可能无意识地导致操作系统的状态冲突和服务崩溃，而面对这种冲突或者服务崩溃，单纯地通过简单的系统重启并不是一个可靠性很高的解决方案。举例来说，一个含有用户状态的服务（比如回话信息）如果对它盲目地进行重启，这将导致状态丢失并且会影响所有使用这个服务的用户。而 CuriOS 就是为了避免这种问题而提出的一种解决方案。它代表一类新型的 OS 设计，它使用轻量级的分布性、独立性以及 OS 服务状态的持久性来减轻重启时状态丢失带来的问题。这种设计也大大减少了包含用户相关状态的 OS 服务的错误传播，通过只有在需要请求处理时将服务压缩到分离开来的保护域以及授权访问客户相关状态来实现相关功能。作者通过进行一些错误注入的实验得出结论：从 87%-100%的人工 OS 服务错误中恢复是有可能的，比如文件系统、网络、定时器和调度器，并且所需要的性能开销很低。

本文提出的主要问题是如何提高操作系统的可靠性。关于操作系统可靠性的研究已经有几十年的历史，但至今仍是大家关注的首要问题之一。操作系统的错误有可能是软件故障或者硬件故障，硬件故障主要由于温度、老化、辐射等原因造成，而软件故障主要是由操作系统的复杂性带来的。而无论是硬件故障还是软件故障，对于这些故障造成的错误在故障恢复方面都已经有所研究，也有一些相应的解决方法。如文章先后介绍的 redundancy 故障恢复方法和微内核设计方法，同时作者也论述了这两个方法各自的缺陷：前者可能存在未检测到的内部错误，而后者通常只是修复了瞬时的故障，并不是一个长久有效的办法。重新启动服务器，正如文中所述，会丢失客户的相关状态。而检查服务器和客户端，当遇到故障的时候进行回滚，回滚到以前已知的良好情况下的状态，这是一个比较可靠的办法，但是在开销上的话费是相当昂贵的，而且还需要客户去处理所谓“时光倒流”而带来了一系列问题。使用工程性客户端去随时监听处理服务器崩溃的问题也是一种可行方案，然而这种方案存在价格昂贵和代码复杂等问题，同时它还想客户端推送了一系列的故障检测逻辑从而会影响客户端性能。通过这些问题的提出，引出本文所要介绍的 CuriOS。CuriOS 的设计理念在于通过控制操作系统的结构来提升操作系统可靠性，并尽可能避免客户端的复杂性以及各种情况带来的开

销问题。

CuriOS 是一个面向对象的操作系统，它主要针对减轻服务器崩溃的问题带来的故障影响来设计的，其背后的思想也比较简单。客户端的状态不在和服务器一起存储，而是和客户端本身。然而，它映射到一个客户端本身不能访问到的存储区域，这样做的理由是可以避免造成客户端可以编辑他们如何出现在服务器上的情况从而带来的安全问题。当一个函数被调用到相应的服务器时，调用这个函数的客户端的状态就会被映射到服务器的地址空间，而且只有该客户端的状态。这样做有效地防止错误在所有客户中传播，因为没有一种方法可以使一个客户端在一次同时破坏两台客户端的状态。

总的来说，上述内容正是本文的创新点，新贡献。这是一个非常简单但是却十分完整，干净利落的想法。正如文中所说，把状态推送到一个客户端上并不是一个很新的想法（NFS 就是这样的一个例子），但是把这个状态进行隔离（甚至是对该客户端本身）才是最关键的一点。

本文在介绍 CuriOS 的核心思想之后就是一系列的实验和测试来进行评估。首先是一个关于“服务器故障之后一个流行的微内核是如何设计（如 Minix3,L4,Chorus 以及 EROS 和一些其他的操作系统）”的一个总结。其中有一个非常值得关注的部分有这样几个重要点，它包含了一些关于微内核错误恢复和故障隔离设计相关内容的原则和意见。内容如下:第一，确保地址（如文件句柄）能够在服务器正常重新启动，以确保透明度。第二，要阻止客户端超时或者在错误恢复期间发行新的呼叫，从而阻止错误传播回客户端。第三，客户端的状态应该确保一点在服务器端重新启动。最后一点，客户端状态应确保隔离。

之后本文进入到 CuriOS 的设计环节。首先从结构上进行阐释，CuriOS 是由一系列交互对象的集合构成的。一个对象可以被限制在一个孤立的存储器保护区域，以减少错误传播。并把这种对象命名为“被保护的对象（protected object）”。所有的“被保护的对象（protected object）”的方法将在减少权限并且有硬件强制内存保护的情况下执行。在 CuriOS 中，所谓“被保护的对象（protected object）”类似于一个在传统微内核系统中的服务器。“被保护的对象（protected object）”有一个小的内核进行协同工作，这个内核称为 Cuik，用来提供标准的 OS 服务。Cuik 通过对受保护的方法(protected method)的调用来操作“被保护的对象（protected object）”，每种“被保护的对象（protected object）”被分配一个私有堆栈。私有堆栈保留着“被保护的对象（protected object）”的每个线程。此堆栈分配并控制受保护的方法（protected method）的第一次调用，提供了一个小延迟用来处理第一次调用“被保护的对象（protected object）”。在 CuriOS 中，线程由 Cuik 管理，使用定义好的借口，一个在执

行 CuriOS 的线程可以跨越用户空间，应用程序，内核和“被保护的对象（protected object）”的边界。CuriOS 采用 C++进行编写，并且使用面向对象技术，尽量避免了代码的重复。同时在 CuriOS 机制中，C++的异常处理被用作错误信号来进行标记。接着本文介绍了关于服务器的状态管理，一个服务器通过实现一个“被保护的对象（protected object）”来支持一个操作系统。一个服务器状态区（SSR）是一个存储 OS 服务器的客户端相关信息的一块被分配的区域。在 CuriOS 中所有的 SSR 是有一个单独的对象进行管理，这个对象叫做 SSRManager。这个 SSRManager 提供用来注册一个新的服务器的功能，绑定客户端到（产生 SSR 的）服务器，撤销客户端-服务端的结合（删除相关联的 SSR）以及枚举所有与服务器相关联的固态继电器。接下来文章主要讲解操作系统的服务器的构建以及可恢复性的错误。

在介绍完整体设计之后，本文对 CuriOS 的整体服务端进行阐述。在进行服务端介绍的时候，主要通过定时器管理，调度，网络，文件系统以及设备驱动几个方面分别进行解释。一个由 CuriOS 提供的定时器管理服务允许用户应用程序访问定时器功能。客户端可以通过发送一个周期性请求启动一个定时器。一旦发生故障引起重启的时候，定时器管理服务可以通过最开始分布在 SSR 中的时间信息重新创建一个完整的内部链表。CuriOS 是一个进程容器的模型，这个容器通过提供一种方法来选择下一个进程运行，并提供调度策略过程。当调度失败重新启动时，它会查询 SSRManageer 的所有客户，并且重新创建它的内部列表。在 CuriOS 的恢复机制中，允许一个可靠的网络协议栈的建设。每个 SSR 与客户端 socket 对象相关联，并且在被映射到 TCP 的 PO 地址空间时与它交互。当有一个将要到来的包时，相应的 SSR 就在相应位置出现并且在通过堆栈发送之前完成映射。CuriOS 目前支持两种不同的文件系统。包括 CramFS 文件系统和 Linux ext2 文件系统。有一个很重要的需要注意的一点，当我们谈及文件系统恢复的时候，我们并不是指恢复错误的文件系统在磁盘上的状态。有许多其他的方法来处理磁盘上的错误数据。最后一个部分是设备驱动，CuriOS 中连续端口的驱动由 PO 来实现，这个 PO 有一个完整的连续端口注册地址映射到内存区域的路径。这个 PO 没有所属的服务并且只有一个客户端：就是 CuriOS 控制对象。当对连续端口进行读写操作时发生的错误，可以通过重启这个 PO 来实现。这个 PO 在创建时就富有对物理内存的读写权限。硬件计时器的错误修复也依赖于它们，上述的定时器管理服务也同样依赖于这些服务的正确功能。

接下来一章进行对 CuriOS 的评估过程。在这一部分文章通过以下几个方面对 CuriOS 进行了一系列评估:错误恢复，性能已经内存开销这三个方面。这里同时进行了一个关于构建 CuriOS 过程中重构工作的分析。在错误恢复方面，针对上一章提到的定时器管理，系统调度，

文件系统等方面进行了详细的评估验证。随后通过受保护的方法调用来对性能方面进行评估。受保护的方法相比于普通的 C++方法调用会带来额外的处理开销，并得出结论认为这些开销可以通过代码优化等方法进一步降低。除了额外的代码实现，开销的另一个主要来源是在进行页码表选择时需要刷新 TLB 页面。而 ARM 架构允许选择性的刷新 TLB 入口，而文章目前的实现不支持此功能。最后得出了在性能评估方面，从错误检测到恢复系统正常通常是在几百微秒。然后进行的是内存管理开销的评估。最后分析了重构工作。

本文所进行的评估包含测量 CuriOS 的执行时的内存消耗以及 CPU 开销。CuriOS 提供了一种叫做“被保护的对象（protected object）”，它的背后是能够照应到客户端状态的映射和失败时能够重新启动的一种包装服务（通过一种 C++异常来标记信号）。CPU 的开销并不是像平常那样，几乎每次呼叫的时间是双倍状态，然而仍然保持在微秒的级别。造成这种状态的主要原因被认定是这些时间花费在刷新 TLB 的交换机上页表之间切换到受保护的对象之间了。恢复时间并没有超过所谓的定量时间，说是只有几百微秒而已，然而并没有什么定量来决定这个时间。内存问题似乎同样也是一种假象。 大量的篇幅专门用于解释为什么状态必须至少是 1KB - 它必须能够达到最接近的页数。 然而，这里给出的最好的数据是对于“小规模数量的客户端”可以以 10kb 数量级进行排序。然而文中并没有说清楚这个所谓的“小规模”是如何来判断或者定义的，或者说它是否是现实的。 同时这种结论也分不清这样的结果多少是由于 CuriOS 的设计，多少是由于微内核本身的作用。

从错误中恢复的能力是可以被测量出来的，这种结果看似会有些奇怪。有两种故障注入到服务器（分别用于每个测试）。这些错误可能会被忽视，但如果它们被检测然到之后，CuriOS 会重启服务器。 第二类故障是一个内存错误，它只是返回了一种“bad read”代码。 这些错误总是能被检测到，并且 CuriOS 每一次都能够进行成功恢复。

尽管这些故障都是存在的，本文有一个看似并不是说的通的地方，文章将 CuriOS 与现有的系统进行比较，在这里只是简单的说“所有的这些故障会导致现有的操作系统服务或者系统崩溃”，如果这是真的，文章需要验证在设计了 CuriOS 的情况下可以承受相比于传统微内核系统来说多少更多的故障错误。作者在文章也评估了故障恢复的时间，不过是用的百分比的形式，这并没有一个直观的数据，而且也不能准确地描述所谓的“恢复”，而只是说明这个方法是可用的。

虽然这些声音故障注入，有一个奇怪的不情愿的文件进行比较的现有系统古玩的行为。本文简单地说，“所有这些故障会导致在大多数现有的操作系统服务或系统故障” - 如果是真的，这将加强古玩的情况下，以显示它多少缺点是免疫比传统微内核。作者还测量时间

的故障恢复，从百分比 - 但不准确描述他们通过'回收'的意思，只是说'可用'。 有时客户会断开连接 - 我认为是的，他们试图避免的条件之一 - 而是他们的指望这些作为成功。

在最后的部分本文针对 CuriOS 进行了一系列的讨论工作。这个讨论从以下五点进行展开：安全，容错能力，对其他系统的适用性，一些其他的优势以及它的缺点。这一部分的讨论并不是文章的总结，而是对前文可能没提到的内容进行了一些扩展，包括了优点之后分析了系统的缺点，其他系统的适应性等都可以算是对整个文章缺口的补充。有时客户端失去连接，这种情况应该是系统试图避免的情况，但他们把这种情况算作成功的情况来进行。这一点上还是感觉有些不妥。

在最后一章进行了简短的小总结，并提供了源代码的网站。总体来讲本文所提出的想法比较新颖，从理论上可以得到很有效的分析和证明，很清晰地说明了 CuriOS 对于提高操作系统可靠性所进行的设计的独到之处，这个想法是一个非常好的思路，有很强的说服力。然而在评估方面感觉本文并不是十分完美，有很多内容似乎一带而过，不置可否。如果让我来写这篇文章，我认为可以在实验和评估方面再进行一些具体的研究，用一些实际的数据来进行评估验证，从而使 CuriOS 在提高系统可靠性，针对错误恢复，系统性能和内存开销方面得到更全面更直观的结论，这样文章的结论更加完美。