基于 Docker 技术的 GUI 应用的在线迁移研究①

房锦章[1,2], 武延军[1]

[1](中国科学院软件研究所 基础软件国家工程研究中心, 北京 100190)
[2](中国科学院大学, 北京 100190)

摘 要: GUI 应用在桌面环境中占据主流, 在线迁移 GUI 应用能够提升用户体验. 然而, 当今广泛使用的 X 窗口系统所定义的协议十分复杂, 不利于 GUI 进程的在线迁移. 另外, 如果两台机器的运行时环境不一致, 将会导致迁移失败. 因此至今未有实现 GUI 应用的在线迁移. 近年来, Wayland 作为新的图形接口协议面世, 其中规定软件图形渲染由 GUI 客户端负责, 这为迁移工作带来极大的便利. 而当今热门的 Docker 容器技术则能把 Wayland GUI 应用及其运行时库打包, 确保运行时环境的一致性. 当迁移发生时, 整个容器都会被迁移至目标机器继续运行. 本文开发了相应的处理 Wayland 协议模块, 并在 CRIU 工具的基础上实现了重建 Wayland 状态. 通过实验证明, 本文方案是可行的, 容易推广到其他的 GUI 应用.

关键词: GUI 应用; 在线迁移; 容器; 桌面

Research on the Live Migration of GUI Applications Based on the Docker technology

FANG Jin-Zhang[1,2], WU Yan-Jun[1]

[1](National Engineering Research Center for Fundamental Software, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)
[2](University of Chinese Academy of Science, Beijing 100190, China)

Abstract: As GUI applications are dominant in desktop environment, online migration GUI applications can improve user experience. However, the X Window System protocol which is in widespread use today, is very complex and not conducive to the live migration of GUI applications. In addition, if the runtime environments of the two machines are not consistent, it will lead to the migration failure. Therefore the live migration of GUI applications has not been realized so far. In recent years, Wayland emerges as a new graphical interface protocol, in which rendering is left to GUI clients. This brings great convenience for the migration. And today's popular Docker container technology is able to package the Wayland GUI application and its runtime library, which ensures the consistency of the runtime environments. When migration happens, the whole container will be migrated to the target machine so that the GUI application can continue its running status. In this paper, the corresponding processing Wayland protocol module is developed. And on the basis of the CRIU tool, it realizes the reconstruction of Wayland state. The experiments show that our scheme is feasible, and can be extended to most GUI applications with minor modification.

Key words: GUI applications; live migration; container; desktop

进程在线迁移是一门比较成熟的技术, 研究成果不少, 但主要针对的是服务器后台进程, 主要应用在分布式的集群环境[1,2,3,4,5]. 而关于 GUI 进程的在线迁移在中外文献上却几乎是一纸空白. 究其原因, 主要有两点.

其一, 难以实现. 无用户交互的服务器后台进程的在线迁移, 和 GUI 应用进程的在线迁移, 两者区别在于后者需要处理图形接口协议. 图形接口协议, 或称软件窗口系统协议, 是规定显示服务器与 GUI 客户端通信方式的协议. 这类协议都是有状态协议, 不易

① 基金项目:中国科学院先导专项(XDA06010600)
收稿时间:2016-01-29;收到修改稿时间:2016-02-29 [doi: 10.15888/j.cnki.csa.005351]

迁移. 有状态协议是指下一个状态依赖于上一个状态, 客户端的每次请求都不是独立的, 而是相关的. 以X11协议来举例说明, X11协议是目前Unix/Linux广泛使用的图形接口协议. GUI 应用进程给 X11 服务端发送请求, 会更改服务端的状态. 这些状态信息存在服务端, 如果服务端是不被迁移的部分, 那么唯一的办法是记录保存这些状态信息, 然后在迁移目标机器的X11 服务端上还原这些状态. 可见, 这个过程的难易程度完全取决于图形接口协议的复杂程度. X 协议在上世纪 80 年代就被定义了, 而后并无大的修改. 研究表明, 该协议的设计并不利于 GUI 进程的在线迁移[6].

其二, 被认为是没有太大必要的. 无论是虚拟机的在线迁移, 抑或进程的在线迁移, 主要都是应用在集群环境当中, 被拿来实现高可用性或者负载均衡, 针对的都是后台服务器进程. GUI 应用一般都是个人的桌面应用, 在线迁移显得并不是那么的迫切. 如果用户在一台机器上运行某个 GUI 应用遇到了问题, 大部分用户都可以接受在另外的机器上重新运行这个GUI 应用.

近年来, 探索 GUI 在线迁移的解决方案出现了转机. 一方面, 很多新的技术层出不穷, 给探索的道路扫清了诸多障碍. 另一方面, 由于硬件技术的发展, 个人拥有多台设备来办公或娱乐已经是常态. 如果这些办公娱乐的 GUI 应用支持在线迁移, 显而会带来更加友好出色的用户体验. 因此, GUI应用支持在线迁移在当今现实生活中显得必要且实用.

1 技术背景

1.1 用户空间下的进程迁移技术

进程在线迁移是一个由来已久的研究课题, 有很多的解决方案和研究成果, 其中不少在内核和应用源代码上做文章, 可扩展性不好. OpenVZ 开源社区曾经希望在 Linux 内核增加设置检查点和还原(Checkpoint and Restore, 简称 CR)功能, 但由于种种原因最终被拒. 后来这部分功能改成在用户空间实现, 作为用户可选的一个工具. 这就是现在的 CRIU 工具(Checkpoint/Restore In Userspace), 即用户空间下对进程设置检查点并能够还原的工具. 这个功能实际上是应用在线迁移的重要步骤[4].

1.2 容器技术

虚拟机的在线迁移和进程的在线迁移很不一样, 进程在线迁移需要更多地考虑资源冲突的问题. 比如进程号冲突, 源主机和目标主机很有可能使用同一进程号. 又比如源主机和目标主机运行环境不太一致, 可能是目标主机缺失某一动态库或者软件版本不兼容等. 这些问题都会导致进程无法在目标主机还原. 而使用容器技术则能很好地解决这个问题.

使用容器能够在一个宿主机上运行多个隔离的系统, 共用一个内核, 使得系统更加高效. Docker 是当今最热的容器技术, 支持将应用打包, 实质上是封装一个完整的应用和完整的运行环境的过程. 打包后的容器可以放到任意能够运行 Docker 容器的机器上, 都能运行该容器的应用, 而不依赖主机的运行环境. 所以对于在线迁移 GUI 应用, 使用 Docker 容器比普通进程更具实际意义. 目前最新 Docker 官方版本 1.9 还没有直接支持在线迁移, 社区正在积极地集成 CRIU 来完成这个功能. 而非官方版本当中, 网上有热心开源人士在 Docker 1.5 和 Docker 1.7 的基础上集成了CRIU[8,9].

1.3 图形接口协议

Wayland 是新兴的图形接口协议, 用以取代传统的X窗口系统. Wayland是异步的面向对象的消息协议, 容易解析处理. 另外, 与 X 规定的由服务端负责完成软件图形渲染工作不同, Wayland改成由GUI客户端负责[7]. 渲染数据是图形接口协议当中最难记录保存和还原的状态, 而这个状态改成了保存在GUI进程空间. 也就是说这个状态不用特殊处理, 随进程内存一起迁移即可, 这无疑给 GUI 应用迁移带来了极大的便利.

1.4 寄生代码

假设在源主机中, GUI 应用进程 A 被进程 B 设置检查点, 迁移到目标主机后, GUI应用是进程C被进程D 还原. 那么如果进程 B(或 D)希望进程 A(或 C)执行特定的额外操作, 就需要寄生代码(Parasite Code)技术[10-13]. 寄生代码是被编译成 PIE(position-independent executable)特殊格式的代码, 有自己的栈空间, 但不能单独执行, 必须寄托在某一已存在的进程空间. CRIU工具在对进程进行CR时, 使用到寄生代码. 而本文中还原显示服务器的状态, 也需要使用到寄生代码.

2 系统整体架构

图形接口协议都是有状态的协议. GUI 应用的在线迁移, 不仅仅是 GUI 应用进程本身的迁移, 还必须

包括图形接口服务器端关于该GUI应用进程的相关状态的迁移. 因此有两种可选的解决方案, 一种是迁移整个有状态的服务器端, 一种是把该 GUI 应用相关的状态单独进行迁移. 显然, 前者是笨重的方案, 后者才是真正的解决之道.

而要迁移状态, 就需要把状态记录保存下来. 唯一的办法是在客户端和服务器端中间, 添加本文开发的 Monitor 模块截获相关的数据并进行处理. 系统的整体架构图如下所示.

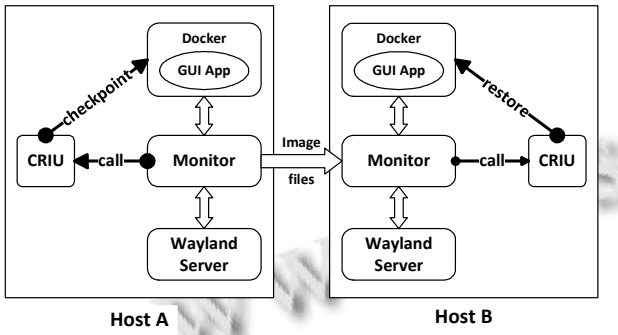

图 1 GUI 应用在线迁移系统整体架构

如图 1 所示, Monitor 模块截获了 GUI 应用与图形接口服务器的交互消息. 截获消息后, 除了转发, 还要进行分析、记录和保存. 也就是说, Monitor 模块也实时地保存着一份和服务器端一样的状态信息. 这些状态信息将会在迁移的时候被发送到目标主机上.

当需要进行 GUI 进程迁移时, 在 Host A 中 Monitor 调用 CRIU 模块, 对 GUI 进程设置检查点(checkpoint). 然后把 GUI 进程的检查点镜像文件和实时保存的关于该 GUI 进程的状态文件, 一起发送到目标主机 Host B. 由 Host B 的 Monitor 模块调用 CRIU 的还原(restore)功能, 把该 GUI 进程重启起来. 本文增强了 CRIU 的功能, 还负责把 GUI 进程的状态文件进行分析后, 利用寄生代码技术在 Host B 的显示服务器重建该 GUI 应用的状态.

使用 Docker 对 GUI 应用进行封装, 能够很好地屏蔽运行环境的差异性. Docker 是当今最流行的容器技术, 开发者可以打包他们的应用和依赖包到一个可移植的 Docker 容器当中, 然后发布到任何流行的 Linux 机器上. 两台主机的运行环境有差异是普遍存在的, 而使用 Docker 来封装, 则保证了运行环境的一致性.

3 系统实现

3.1 解析 Wayland 协议

解析 Wayland 协议是 Monitor 模块记录保存状态信息的基础和前提. Wayland 协议是基于面向对象思想设计的异步消息通信协议. 与 X11 协议不同, Wayland 协议的通信方式只能使用 Unix 域套接字(也称本地套接字). 客户端发送给服务端的消息称之为 request, 从服务端发送给客户端的消息有 event 或者 error 两种.

request 消息是对某个对象所进行的操作请求, event 消息往往是由于某个对象状态发生了变化, 服务端主动发送给客户端通知, error 消息是出错报告. 消息的格式简单而统一, 由消息头和消息体构成. 消息头长度固定, 为 8 个字节. 前面 4 个字节存放的是对象的 ID, 紧接着分别是两个字节的操作码和消息总长度. 消息体是操作参数, 长度不固定.

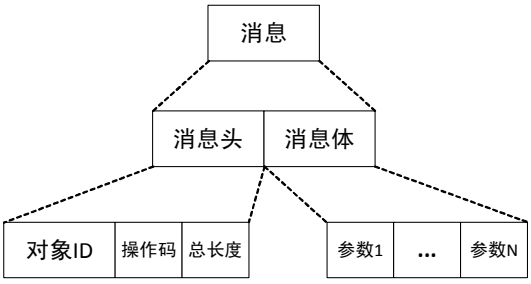

图 2 Wayland 消息格式[14]

参数的个数由 Wayland 协议所定义的. Wayland 官方定义了最核心的协议, 开发者可以扩展协议. 协议都是使用 XML 文件来描述的, 涉及到的最主要概念及其含义见下表.

表 1 Wayland 协议主要的概念

属性	含义
interface	对象模板, 类似于面向对象中 class 的概念
request	客户端请求消息, 类似于面向对象中方法的概念
event	服务端返回事件通知, 与 request 的区别是方向不同
arg	request 和 event 的参数

下面摘抄 wayland.xml 当中的关键内容.

```
<?xml version="1.0" encoding="UTF-8"?>
<protocol name="wayland">
  <interface name="wl_compositor" version="3">
    <description summary="the compositor singleton">
A compositor.  This object is a singleton global.  The
```

```
compositor is in charge of combining the contents of multiple surfaces into one displayable output.
    </description>
    <request name="create_surface">
      <description summary="create new surface">
      Ask the compositor to create a new surface.
      </description>
      <arg name="id" type="new_id" interface="wl_surface"/>
    </request>
    <request name="create_region">
      <description summary="create new region">
      Ask the compositor to create a new region.
      </description>
<arg name="id" type="new_id" interface="wl_region"/>
    </request>
  </interface>
</protocol>
```

解析 Wayland 协议，得先扫描定义 Wayland 协议的 XML 文件，然后根据这些信息去解析 Wayland 消息. 对象每次创建的时候，得记录它对应的 interface. 而 request 或 error 的操作码则是与其在 XML 里出现的顺序相关，第一个操作码是 0，依次递增. 如 wl_compositor 中的 create_surface 操作码为 0，create_region 操作码为 1.

3.2 记录、保存、还原对象

Wayland 协议是有状态的面向对象的协议，所有的消息都是关于某个对象的. 所以迁移 Wayland 的状态，其实就是迁移 Wayland 的对象. 所有的对象都有唯一的 ID 值来标志. 因此, Monitor 模块需要按照时间先后记录创建了哪些对象，对象的 ID 值是多少，设置了什么样的属性. 这些信息在设置检查点时会被序列化保存成状态文件，发送给目标主机.

GUI 进程在目标主机重启，而迁移过程对其来说是透明的，也就是 GUI 进程不知此时所在主机的显示服务器没有其相关状态. 本文使用寄生代码来在显示服务器上重建 GUI 进程的对象. 具体步骤如下:

① CRIU 模块使用 ptrace 系统调用暂停 GUI 进程(即进入 PTRACE_SEIZE 状态)，保存寄存器当前状态，现场保护;

② 在 GUI 进程当中插入 mmap 系统调用，分配一段可执行的内存段;

③ 把寄生代码复制到刚分配的内存，设置 IP 寄存器，使得 GUI 进程的执行点更改为寄生代码的起始处;

④ 寄生代码执行，发出重建对象的 request 消息，Monitor 模块接收后，处理再转发给显示服务器;

⑤ 寄生代码完成后，进程重新进入 PTRACE_SEIZE 状态. 把分配的内存释放，还原寄存器初始状态，还原现场.

对象 ID 值是由客户端来指定的，这个可能是 Wayland 并无深意的设计，却给 GUI 进程迁移带来了极大的方便. 试想如果是由服务端来指定的，那么在重启 GUI 进程前，需要在服务端创建所需的对象. 此时对象的 ID 和迁移前的对象的 ID 是不一致的，那么就需要 Monitor 模块在每个来往的消息中提取对象 ID，并做转换，修改后再转发. 可见这个做法既降低了性能又增加了系统复杂性.

Wayland 对象根据其创建方式可以分为两种. 第一种是普通对象，由客户端自行创建，并指定 ID 值. 还原此类对象最为简单，在重启的 GUI 进程使用寄生代码创建同样 ID 值的对象即可. 另外一种全局对象，即服务端早已存在的对象. 在客户端建立起连接后，服务端会告知客户端存在哪些全局对象. 客户端给自己感兴趣的全局对象赋予 ID 值，方能使用. 这个过程称之为绑定对象. 显然还原此类对象也较为简单，只需给相同的全局对象绑定同样的 ID 值即可.

3.3 共享内存的处理

Wayland 协议当中有个别的特殊对象的创建过程，需要文件描述符这个特殊的参数，用以实现共享内存. 下面就以 wl_shm_pool 对象的创建来说明.

Wayland 协议虽然规定了软件渲染的工作交由客户端完成，但是硬件渲染还是由服务端和内核联手完成的，也就是说软件渲染的结果需告知服务端. Wayland 规定客户端和服务端的渲染数据是通过共享内存的方式来提高性能的. 在客户端与服务端建立起连接后，客户端会创建一个临时文件，获得文件描述符 fd，然后通过系统调用 mmap 进行内存映射. 当把 fd 通过 Unix 域套接字传送给服务端的时候，服务端再通过 fd 进行一次系统调用 mmap，则客户端和服务端即可操作同一段内存，实现了共享内存的操作.

mmap 系统调用有两个重要参数 addr 和 length.

addr 参数用来指定内存映射的起始地址, length 指定映射的长度. addr 参数可以为空, 这样内核会自主选择合适的一段虚拟内存来映射. 迁移前, GUI 进程创建 wl_shm_pool 对象时, 并不指定这个起始地址. 迁移至目标主机后, CRIU 模块重启了 GUI 进程, 接下来使用寄生代码重建 wl_shm_pool 对象时, 需要指定这个起始地址. 这个起始地址必须和之前的一致, GUI 进程才能识别出迁移前的渲染数据.

4 实验与验证

本节主要通过实验来验证本文提出的 GUI 应用迁移方案的可行性.

4.1 实验环境

实验需要两台型号一样的 PC 物理机, 基本环境是: Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz, 8 GB 内存, 操作系统是 Linux Mint 17.

由于 CRIU 和 Docker 都对内核有较高的要求, 所以需要升级内核. 下载 Linux kernel 4.3, 按照 CRIU 官网上的说明[15], 使能各项所需的内核配置. 由于本文使用 Docker 的存储引擎是 AUFS, 而 AUFS 已经被最新内核所抛弃, 所以需要给内核打 AUFS 的补丁[16], 编译内核时使能 CONFIG_AUFS_FS 选项.

使用 CRIU 来 CR 一个 Docker 容器的方法分为两种, 分别是外部 CR(External C/R)和本地 CR(Native C/R)[17]. 两者的区别是 Docker 守护进程是否能够感知 CR 操作. Docker 官方最新版本还没有提供本地 CR, 本文实验使用的是外部 CR[17].

CRIU 可以对多个进程(进程树)以及其内部的资源设置检查点, 但如果某个资源关联了外部就需要特殊处理. 如管道的两端, CRIU 可以同时对管道两端的进程设置检查点, 但如果只设置其中一端则是失败的. 原因是还原的时候, 无法确认另一端是否存在. 对于 Unix 域套接字同样存在这个问题. 由于 GUI 应用 Unix 域套接字另一端(显示服务器)是肯定存在的, 所以本实验修改了 CRIU 1.7 版本的源代码, 注释了对这个问题的报错, 使之忽略这个问题.

4.2 实验过程

4.2.1 Docker 容器运行 Wayland 应用

Docker 需要一些特别的设置才能支持运行 GUI 应用. 已经有人提出在 Docker 当中运行 X GUI 应用的方法[18]. 在 Docker 中运行 Wayland GUI 应用也十分的类似.

Wayland 服务器在系统某个目录下(由环境变量 XDG_RUNTIME_DIR 来指定)创建本地套接字文件(由环境变量 WAYLAND_DISPLAY 来指定)来监听客户端. 例如本文使用 Wayland 服务器是 weston 1.3, 它在/run/user/1000 目录下创建了 wayland-0 这个套接字. 在 Docker 容器里需要设置这两个环境变量, Wayland 客户端才能连接上这个套接字.

本文选择 Ubuntu 镜像来创建 Docker 容器. 在容器内必须安装的软件依赖包是 libwayland-client0, 包含了 Wayland 客户端运行时的依赖库. 为简化操作, 实验采用的 Wayland 客户端是一个“hello world”的 Wayland 程序[19].

Docker 容器内部和外部是两个独立的文件系统, 要想让 Docker 内部的 Wayland 客户端访问外部的 Wayland 服务端, Docker 提供了相关的支持. 可以挂载本地目录/run/user/1000 到 Docker 内部的文件系统.

完成以上相关设置, 在 Docker 容器内就可以运行 Wayland 应用.

4.2.2 对 Wayland 应用设置检查点并还原

在 Wayland 应用运行过程当中, Monitor 模块一直记录相关的状态信息. 当向 Monitor 发出迁移的命令时, Monitor 就会向调用 CRIU 命令, 对 Docker 容器执行设置检查点操作, 同时 Monitor 会把记录的状态信息保存成状态文件, 和设置检查点产生的镜像文件, 一起发给目标主机. 目标主机接收完成后, Monitor 先调用 CRIU 重启 Docker 容器来运行这个应用. CRIU 此时再利用寄生代码技术, 通过分析状态文件, 重建 Wayland 之前的对象. 这样就能在目标主机上还原这个 Wayland 应用.

4.3 实验结果

用于本次实验的 Wayland 应用十分简单, 没有复杂的交互过程, 仅有一个简单的响应鼠标移动事件. 通过实验可以看到, Wayland 应用迁移到目标主机后, 能够继续响应鼠标移动事件. 说明在 Wayland 服务端成功地迁移了 Wayland 的对象, 说明了该方案实现了 GUI 应用的在线迁移功能.

5 总结与展望

本文对 GUI 应用在线迁移技术的实现进行了一个前沿探索. 结果表明, 采用 Docker, CRIU 和 Wayland

等新技术, 可以初步实现 GUI 应用的在线迁移. 但是, 还有很多细致的工作需要继续探索. 把 GUI 应用的在线迁移做到和文件传输一样便捷, 在未来可能是一个重要发展方向. 这需要进一步的努力, 包括容器标准的确立, 进程迁移技术的改进, 图形接口协议的定义, 甚至操作系统内核的支持等. 本文工作只是迈出了图形应用迁移技术的第一步.

参考文献

1 Duell J. The design and implementation of berkeley lab's linux checkpoint/restart. Lawrence Berkeley National Laboratory, 2005.

2 Gerofi B, Fujita H, Ishikawa Y. An efficient process live migration mechanism for load balanced distributed virtual environments. 2010 IEEE International Conference on Cluster Computing (CLUSTER). IEEE. 2010. 197–206.

3 Milojičić DS, Douglis F, Paindaveine Y, et al. Process migration. ACM Computing Surveys (CSUR), 2000, 32(3): 241-299.

4 Bozyigit M, Wasiq M. User-level process checkpoint and restore for migration. ACM SIGOPS Operating Systems Review, 2001, 35(2): 86–96.

5 Laadan O, Hallyn SE. Linux-CR: Transparent application checkpoint-restart in Linux. Linux Symposium, 2010: 159.

6 Nye A. X Protocol Reference Manual for X11, Release 6. O'Reilly Media, Inc., 1995.

7 http://wayland.freedesktop.org/docs/html.

8 https://github.com/SaiedKazemi/docker.

9 https://github.com/boucher/docker/tree/cr-combined.

10 http://criu.org/Code_blobs.

11 http://criu.org/Parasite_code.

12 https://code.google.com/p/ptrace-parasite.

13 https://github.com/xemul/compel.

14 刘琳.Wayland 协议分析.科技视界,2014,(34):93–95.

15 http://criu.org/Installation.

16 https://github.com/sfjro/aufs4-standalone.

17 http://criu.org/Docker.

18 http://fabiorehm.com/blog/2014/09/11/running-gui-apps-with-docker/.

19 https://github.com/hdante/hello_wayland.