Elanix 内核对象服务通信机制的设计与实现[1]

苏杭，高峭，陈志成，王小鸽
（清华大学计算机科学与技术系，北京 100084）
Email: su-h03@mails.tsinghua.edu.cn

摘　要：Elanix 是和欣操作系统(Elastos)在 Linux 上的构件化虚拟操作系统，Elanix Server 是其中的内核对象服务模块。文章针对 Elanix 中内核对象的构件化特性，提出了 Elanix Server 的通信机制，设计并实现了其通信层次，包括：实现各内核对象的主体层、构件化设计的接口层、使用元数据的列集层、基于设备文件的传输层。文中分析了此通信机制的优点，测试表明 Elanix Server 对应用程序请求的响应时间明显少于 Wine Server 的响应时间，这为 Elanix 内核对象与应用程序之间的通信提供了有效途径。

关键词：和欣操作系统, 内核对象服务, 通信机制, CAR 构件技术, Linux 模块
中图分类号：TP302.1　　文献标识码：A

Design and Implementation of Communication Mechanism in Elanix Server

SU Hang, GAO Zheng, CHEN Zhicheng, WANG Xiaoge
(Department of Computer Science and Technology, Tsinghua University, Beijing 100084)

Abstract: Elanix is the component-based virtual operation system of Elastos on Linux. Elanix Server provides kernel object services for applications. In this paper, a communication mechanism of Elanix Server is proposed for the component-based kernel objects. The design and Implementation of the communication mechanism are presented in detail. It contains four layers including body layer, interface layer, marshal layer and transfer layer. This paper also analyzes the advantages of the communication mechanism. A test result shows Elanix Server has a better performance than Wine Server, and proves the mechanism is an efficient approach for communication between Elanix kernel objects and applications.

Key words: Elastos, Kernel Object Service, Communication Mechanism, CAR, Linux Module.

1 引言

“和欣”操作系统（Elastos）是基于构件技术的下一代网络操作系统。Elastos 提供的功能模块全部基于 CAR（Component Assembly Run-Time）构件技术 [1]。按照 CAR 构件标准开发的应用程序不仅可以在 Elastos 上运行，而且可以在各种操作系统上的“和欣”虚拟操作系统上运行。

虚拟操作系统是虚拟机技术的一种。它对操作系统进行抽象，在一个操作系统环境中虚拟出另一个操作系统的环境，使得应用程序在不用改动代码的情况下，在虚拟操作系统上运行[2]。Wine 和 Linux 上的“和欣”虚拟操作系统（Elanix）都是这类虚拟机。以 Wine 为代表的虚拟操作系统，经过十几年的发展，技术已经日臻成熟。目前在 Wine 上已经可以直

[1]**基金项目：**国家高技术研究发展计划（863 计划）项目（编号 2003AA1Z2090） 。
作者简介：苏杭（1979-），男，安徽马鞍山人，硕士，主要研究方向是虚拟操作系统；高峭（1973-）， 男，北京人，硕士，主要研究方向为 WEB 环境下构件运行机制；陈志成（1973-），清华大学计算机科学与技术系，博士后；王小鸽（1957-），清华大学操作系统与中间件技术研究中心，中心副主任，教授。

接运行部分 Windows 95/98/NT 程序[3]。但是 Wine 仍有不足之处，目前 Wine 遇到的最大问题是 Wine Server 对应用程序的请求响应缓慢[4]。

类似与 Wine，Elanix 可以在 Linux 上虚拟出 Elastos 操作系统环境，让 Elastos 应用程序和 CAR 构件无需重新编译，跨平台运行在 Linux 操作系统上。Elastos 和 Windows 有很多相似之处，比如都使用 PE 可执行文件格式，都有构件规范（CAR 与 COM）等。同时 Elastos 也有着自身的特点，比如 Elastos 内核提供基于 CAR 构件标准的内核对象服务，所以笔者在借鉴了 Wine 的成熟技术基础上加以改进，设计并实现了 Elanix Server 的通信机制，克服了 Wine Server 响应缓慢的不足。

2 Elanix Server

2.1 Elanix 系统结构

如图 1 所示，为了向 Elastos 应用程序和 CAR 构件提供与 Elastos 操作系统相同的运行环境。Elanix 虚拟操作系统由三部分组成，各部分功能描述如下：

（1）PE Loader 负责将 Elastos 应用程序和 CAR 构件加载到 Linux 进程空间中执行。

（2）动态链接库 Elastos.so 向 Elastos 应用程序和 CAR 构件提供了 Elastos API 函数。

（3）Elanix Server 模拟 Elastos 操作系统内核的功能，提供内核对象服务。

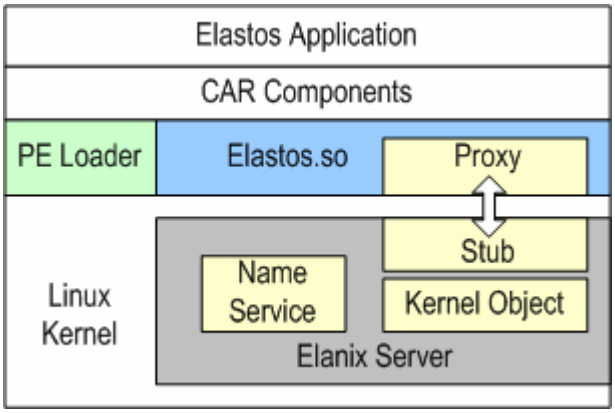

图 1 Elanix 虚拟操作系统结构图

2.2 内核对象服务

与传统操作系统不同，Elastos 内核的各个功能模块按照 CAR 构件规范建立，都是 CAR 构件对象。Elastos 内核向 Elastos 应用程序提供的大部分系统服务是内核对象的接口，只有少量系统服务是直接以 API 函数形式提供。如表 1 所示，Elanix Server 作为 Elastos 内核在 Linux 上的模拟实现，同样也向 Elastos 应用程序提供内核对象服务。

表 1 Elanix Server 中主要内核对象及其方法

对 象	说 明	接口中的方法
进程对象	代表 Elanix 进程，包含多个线程对象	Start, Kill, GetThreads, ...
线程对象	代表 Elanix 线程，执行单位	Start, Suspend, Resume, Abort, ...
共享内存对象	代表可以被多个进程共享的一段内存	Attach, Detach
互斥对象	代表互斥信号量	Lock, TryLock, Unlock
模块对象	代表内存中载入的一段代码或数据	GetEntryPoint, GetName, …

与传统通过 API 函数提供操作系统内核功能相比，内核对象服务有以下优点：

(1) 符合面向对象和面向构件的程序设计思想。应用程序编写更方便、结构更清晰。

(2) 都有完备的元数据，实现自描述，可以动态加载/卸载，这包括硬件驱动即插即用。

(3) 具有动态可配置性，当环境发生变化时，可以对内核对象服务做相关调整

3 Elanix Server 通信机制的设计

3.1 Linux 模块与设备文件

Elanix Server 中的内核对象主要负责进程、线程管理、共享内存、进程间通信等工作。这些内核对象中的数据和函数需要被多个进程共享，所以 Elanix Server 及其与用户程序之间的通信方式可以考虑以下四种设计方案：

(1) 申请一块所有进程共享的内存，将内核对象放入其中。各个进程可以直接调用共享内存中的内核对象。

(2) 在一个单独的 Linux 进程中实现内核对象，其它进程通过 IPC 或 Socket 与其通信。

(3) 修改 Linux 内核源码，在 Linux 内核中实现内核对象，重新编译 Linux 内核。用户程序可以通过系统调用得到内核对象服务。

(4) Elanix Server 使用 Linux 模块作为载体，实现内核对象。用户程序可以通过设备文件得到内核对象服务。

经过比较，笔者选择用第四种方案设计并实现了 Elanix Server 通信机制。Linux 模块是一种目标对象文件，它们是 Linux 内核的一部分，但是并没有编译到内核里面去。模块可以在系统启动时加载到系统中，也可以在系统运行的任何时刻加载；在不需要时，可以将模块动态卸载[5,6]。与其它三种方案相比，第四种方案有以下优点：

(1) Linux 模块运行在 Linux 内核空间，由于用户空间的程序不可以直接访问内核空间的程序和数据，这样提高了 Elanix Server 的安全性和可靠性。共享内存虽然使用方便，但这种方式不可靠，一个应用程序的误操作就有可能导致整个 Elanix 系统崩溃。

(2) 基于设备文件的通信方式，其通信效率要高于 IPC 和 Socket 等进程间通信方式。

(3) 与修改 Linux 内核源码方法相比，使用 Linux 模块作为载体，不需要重新编译 Linux 内核，便于 Elanix Server 的发布和升级，提高了 Elanix Server 灵活性和在不同版本 Linux 内核上的可移植性。

3.2 Elanix Server 通信层次

因为用户空间中的程序不可以直接访问内核空间中的内核对象，如同程序不可以直接访问处于不同进程空间的 CAR 构件对象，所以笔者将构件跨进程通信机制引入内核对象服务，使得用户程序可以通过代理透明地调用内核对象，如同内核对象处于同一用户空间一样。

表 2　Elanix Server 通信各层含义

空间 / 层次	用户空间	内核空间
第四层 主体层	用户程序主体，请求调用内核对象服务。	内核对象主体，实现内核对象服务。
第三层 接口层	以 CAR 构件接口的形式向用户程序提供内核对象服务。	所有的内核对象接口定义以及一个总的 IKernel 接口定义。
第二层 列集层	根据元数据生成标准代理对象，将对象函数调用信息打包传递。接收存根对象返回结果。	内核对象的存根解包代理传来的信息，调用对应的内核对象服务。发送结果。
第一层 传输层	将请求从用户空间传递到内核空间。	分发请求，将结果从内核空间传递到用户空间。

如表 2 所示，为了保证 Elanix Server 可以向 Elastos 应用程序提供与 Elastos 内核无差别的内核对象服务，定义了 Elanix 中用户程序与内核对象的通信层次。每个层次提供服务帮

助上个层次之间通信，且每个层次对于上个层次来说都是透明的，可替换的。

4 Elanix Server 通信机制的实现

4.1 主体层与接口层

Elanix Server 的接口层定义了各个内核对象的接口包括 IKernel、IProcess、IThread、IMutex、IShareMemory、IModule 等等。主体层则是各个内核对象的具体实现。如图 2 所示，Elanix Server 调用 Linux 内核函数实现了与 Elastos 相同功能的内核对象，最后这些对象都被汇总到一个 Kernel 对象中。

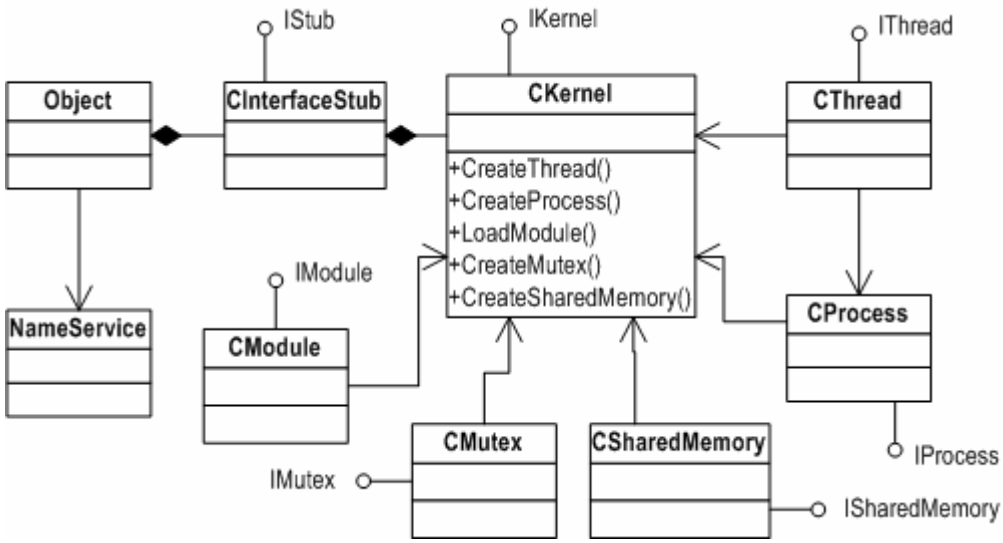

图 2 主要内核对象及名字服务结构图

在 Elanix Sever 初始化过程中，Kernel 对象将在名字服务（Name Service）上注册为“ElastosKernel”服务。当用户程序调用内核对象服务时候，先用 EzFindService 函数查找“ElastosKernel”服务得到 IKernel 接口指针，再得到各个内核对象的接口指针，然后就可以调用各个内核对象的方法。例如，启动新进程执行 Example 程序的代码如下：

```
EzFindService(EZCSTR("ElastosKernel"), (IUnknown **)&pIKernel);   //得到 IKernel 指针
pIKernel->CreateProcess(&pIProcess);     //创建新进程对象，得到 IProcess 指针
pIProcess->Start(EZCSTR(“Example”), NULL);     //在新进程中启动程序 Example
```

4.2 使用元数据的列集层

列集层主要负责自动生成内核对象的代理和存根、函数调用的列集散集。Elanix Server 中的内核对象都是按照 CAR 构件规范编写的，都有完备的元数据。下面给出了与列集层相关元数据的 C 语言定义。该部分元数据描述各个内核对象类、接口、方法的各自属性，以及它们之间的关系。在元数据的支持下，列集层在运行过程中动态自动生成内核对象的代理和存根，摒弃了通过静态编译生成存根、代理的方式，避免了静态编译生成存根、代理常常造成代码量增大，代码逻辑显得极其复杂等问题。

图 3 中显示了接口代理与接口存根之间的联系。用户程序调用内核对象服务的列集散集过程描述如下：

(1) 当新建一个内核对象的时候，会实例化 CInterfaceStub 类为该内核对象实现的每一个接口创建一个接口存根，同时为该内核对象创建用 oid 唯一标识的 Object 对象，通过 Object

对象可以索引各个接口存根。

(2) 当用户程序请求某个内核对象接口时，将会在用户空间中实例化 CInterfaceProxy 类，为该内核对象接口创建接口代理，并将此接口代理的指针作为内核对象接口指针返回给用户程序。

(3) 当用户程序调用内核对象接口中的方法时，因为此时用户程序得到是接口代理的指针，而且在 C++对象模型中，接口即是纯虚基类，纯虚基类由一张虚函数表和指向该虚函数表的指针组成，所以用户程序中对虚函数表指针的操作，就变成对接口代理成员变量 m_pvVptr 的操作（当且仅当 m_pvVptr 是接口代理的第一个成员变量）；用户程序对虚函数的调用，就变成对 mtable 表中对应序号的 method 函数的调用。接着 method 函数调用该接口代理中的 ProxyEntry 方法，同时把自己在 mtable 中的序号做为参数也传递给 ProxyEntry。

```
//与列集层相关的元数据结构定义
typedef struct _MethodEntry {
    UINT8       paramNum;
    CIBaseType  *params;
}MethodEntry;
typedef struct _InterfaceEntry {
    IID         iid;
    UINT16      methodNum;
    MethodEntry *methods;
}InterfaceEntry;
typedef struct  _ClassEntry {
    CLSID       clsid;
    UINT16      interfaceNum;
    InterfaceEntry  *interfaces;
}ClassEntry;
```

(4) ProxyEntry 根据该方法的元数据将 Oid 标识、接口序号 uIndex、方法序号和以及输入参数打包。然后将打包好的数据通过 SysInvoke 函数发送给对应接口存根，并等待 SysInvoke 函数返回结果。

(5) SysInvoke 能够从用户空间穿越到内核空间，根据 oid 找到对应的 Object 对象，然后，根据接口序号 uIndex 找到对应的接口存根，调用该接口存根中的 Invoke 方法。

(6) 接口存根中的 Invoke 根据元数据信息解包接口代理发来的数据，然后调用真正内核对象接口中对应序号的方法，得到返回结果。再使用元数据将返回结果打包。

(7) SysInvoke 返回后，接口代理中的 ProxyEntry 解包传来的结果，返回给用户程序。

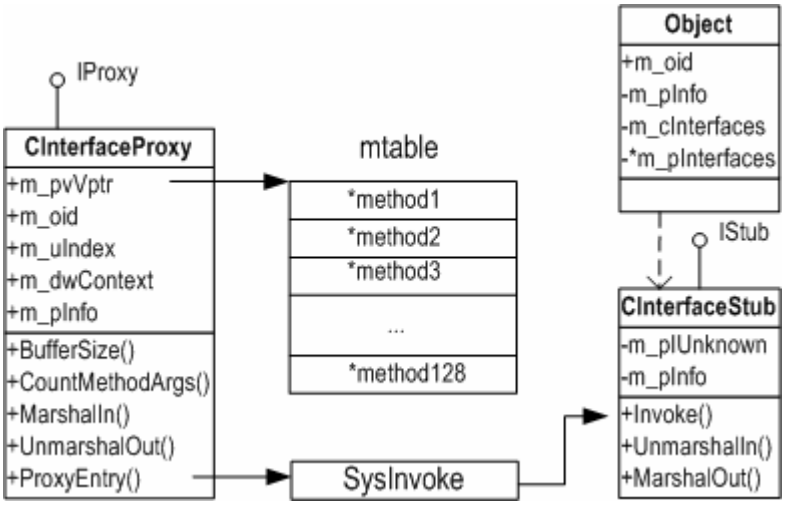

图 3 接口代理与接口存根联系图

4.3 基于设备文件的传输层

传输层的功能是为列集层之间通信服务。列集层之间的接口是定义好的一组 SysCall 函数，处于用户空间列集层程序可以透明的调用内核空间的 SysCall 函数而不必关心传输层是如何实现函数请求、参数和结果传递的。

Elanix Server 是一个 Linux 模块，在传输层它使用设备文件与用户程序相互通信。其关

键部分代码如下所示：

Protocal.h 头文件中定义了通信双方共用的数据结构包括函数请求号 SysNo 枚举类型，函数输入 Request 联合体、输出参数 Reply 联合体等等。在 Elanix Server 中定义了 SysCall 函数指针数组，注意 SysCallTable 中函数指针的排列顺序必须与 SysNo 中定义的一致。

```
//Protocal.h 中定义 SysNo
enum SysNo{
        RegisterService,
        UnRegister,
        Invoke,
        …
        NB_SYSNO,
};
```

```
//Elanix Server 中定义 SysCall 函数表
typedef void (*handler) (union request* preq,
                         union reply* prep);
static const handler SysCallTable[NB_SYSNO] ={
        (handler) SysRegisterService,
        (handler) SysUnRegister,
        (handler) SysInvoke,
        ….
};
```

数据结构定义完毕后，对于不同 SysCall 调用，Request 联合体和 Reply 联合体会被赋予不同结构类型和值。然后用户程序中就可以通过读写设备文件发送函数请求、得到函数结果。Elanix Server 会接受传入的数据，通过已经定义好的函数指针数组 SysCallTable 解析并分发，调用对应的 SysCall 函数完成请求。最后，当用户程序调用 read 函数的时候，Elanix Server 会将 Reply 中的结果传给用户程序。

```
//用户程序中读写设备文件
if(open("/dev/server", O_RDWR) != -1){
        write(fd, pRequest, sizeof(*pRequest));
        read(fd, pReply, sizeof(*pReply));
        fclose(fd);
}
```

```
//Elanix Server 中分发 SysCall 请求
enum SysNo sysno = request_ptr->sysno;
if(sysno < NB_SYSNO)
        SysCallTable[sysno](request_ptr,
                            reply_ptr);
```

5 Elanix Server 与 Wine Server 性能比较

5.1 测试设计与结果

Wine 是 Linux 上的 Windows 虚拟操作系统[3]。与 Elanix Server 一样，Wine Server 模拟了部分 Windows 内核功能，提供进程和线程管理、进程间通信等系统调用[5]。此次测试专门针对 Wine Server 和 Elanix Server 对用户程序请求的响应时间。测试前在 Wine Server 和 Elanix Server 中各定义了一个新的空系统调用 NoOperation，该系统调用请求不做任何处理，直接返回空值。

测试硬件环境为：CPU 是 AMD 1.5G、内存 256M，硬盘 80G。

测试软件环境为：操作系统是 Red Hat Linux 9.0，内核版本是 2.4.20-8，编译器是 gcc 3.2.2，Wine 的版本为 20041019。

分别测试 Wine Server 和 Elanix Server 对用户程序一百万次、二百万次、三百万次请求响应时间。得到测试结果如表 3 所示。

表 3　Elanix Server 和 Wine Server 的性能测试结果

测试次数	1 百万次	2 百万次	3 百万次	平均单次
Elanix Server 响应时间	0.579352 (ms)	1.158849 (ms)	1.737585 (ms)	0.579324 (μs)
Wine Server 响应时间	10.28146 (ms)	21.01983 (ms)	31.63047 (ms)	10.44497 (μs)

5.2 分析讨论

结果显示 Wine Server 的响应时间明显要大于 Elanix Server。前者平均单次响应时间是 10.44497 微秒，后者平均单次响应时间是 0.579324 微秒，仅为前者的 1/18。原因如下：

（1）Elanix Server 是一个处于 Linux 内核空间的内核模块，响应用户程序请求不需要切换进程上下文；Wine Server 作为一个独立的用户进程，每次响应用户程序的请求，都必须在两个进程上下文间来回切换[3]。每次进程上下文切换包括保存当前进程环境、刷新 CPU TLB、运行调度算法、设置 MMU、恢复下一个将运行的进程环境。这大大增加了 Wine Server 的响应时间。

（2）用户程序与 Elanix Server 之间是通过设备文件互连，这种连接方式要快于 Wine Server 与用户程序之间的 Socket 连接方式[3]。

但是由于 Elanix Server 是 Linux 模块，与 Linux 内核耦合性较强，所以 Elanix 在其它“类 Unix”操作系统上的可移植性不如 Wine。鉴于目前 Elanix 只考虑在 Linux 上实现，对应用程序请求的快速响应是最重要的需求，当前的 Elanix Server 的通信机制是最佳选择

6 结束语

Elanix 虚拟操作系统中 Server 模块负责向 Elastos 应用程序提供内核对象服务。内核对象服务不同于一般操作系统内核提供的系统调用。它是由符合 CAR 构件规范的内核对象提供的方法组成。Elanix Server 通信机制定义了四个通信层次，使用代理、存根和列集、散集技术，满足了 Elastos 应用程序调用内核对象服务的要求。此外，Elastos Server 采用了 Linux 模块为载体，Linux 设备文件为通信方式。与同类型虚拟操作系统 Wine 相比，Elastos Server 对用户程序请求的响应时间明显少于 Wine Server 的响应时间，在 Elanix 虚拟系统的实际应用中体现了良好的性能。

参考文献

[1] 科泰世纪有限公司. 《和欣操作系统 2.0 与 CAR 构件技术》资料大全 [EB/OL], http://www.elastos.com.cn/ download.php? DownLoadID=3, 2004, 11.

[2] J. E. Smith, Ravi Nair. Virtual Machines: Architectures, Implementations and Applications[M]. Morgan Kaufmann Publishers, 2004.

[3] Uwe Bonnes, Jonathan Buzzard, Zoran Dzelajlija et al. Wine Developer’s Guide[EB/OL], http://www.winehq.org/site/docs/wine-devel/index, 2004, 11.

[4] Wine Weekly Newsletter. http://www.winehq.com/site?news=archive[EB/OL], 2005, 1.

[5] De Goyeneche, J.-M. De Sousa. Loadable Kernel Modules, E.A.F. Software[J], IEEE.Volume 16, Issue 1, Jan.-Feb, 1999, P65 -71.

[6] Peter Jay Salzman, Ori Pomerantz. The Linux Kernel Module Programming Guide[EB/OL]. http://tldp.org/LDP/lkmpg/2.6/html/, 2005,1.