构建跨平台的通信模型及其基于 RCS 库的实现方法①

廉梦佳 [1,2], 刘荫忠 [1,2], 王俊霖 [3]

[1](中国科学院大学, 北京 100049)
[2](中国科学院 沈阳计算技术研究所, 沈阳 110171)
[3](大连理工大学, 大连 116024)

摘 要: 在控制系统的领域中, 现多数采用的是多处理器或多操作系统,会遇到异构平台间的过程控制等的通信问题, 且不同平台可处理的数据格式不同. 本文提出了一种跨平台的通信模型,该模型中采用中间代理的方法, 将底层通信与上层应用分离, 使底层的通信过程对用户透明. 本文以 RCS 库的 CMS 和 NML 为基础, 根据模型给出了相应的实现方法. CMS 是 RCS 库的底层通信管理系统, 提供消息读写、数据编码/解码以及跨平台的通信功能. NML 是对 CMS 的封装, 提供了更高层次的接口, CMS/NML 均由一系列 C++类实现.

关键词: 跨平台; 通信模型; 中间代理; RCS 库; CMS/NML

Build of Cross-Platform Communication Model and Implementation Method Based on RCS Library

LIAN Meng-Jia[1,2], LIU Yin-Zhong [1,2], WANG Jun-Lin[3]

[1](University of Chinese Academy of Sciences, Beijing 100049, China)
[2](Shenyang Institute of Computing Technology, Chinese Academy of Sciences, Shenyang 110171, China)
[3](Dalian University of Technology, Dalian 116024, China)

Abstract: In the field of the control system, MP or Multiple operating systems is often used, which will encounter the communication problems of process control operations in heterogeneous platforms, because different platforms handle data format is different. This paper proposes a cross-platform communication model. The model uses the method of intermediate proxy, which separates the underlying communication and the upper application and makes the underlying communication process transparent to the user. Based on the CMS and NML of RCS library, according to the model, the paper presents the corresponding implementation methods. CMS is the underlying communication management system of RCS library, providing the functions of reading and writing messages, encoding/decoding data and cross-platform communication. NML is the encapsulation of CMS, providing a higher level interface to make the upper system communication more convenient and to solve the communication between the heterogeneous platforms better. The CMS/NML are realized by a series of C++ classes.

Key words: cross-platform; communication model; intermediate proxy; RCS library; CMS/NML

在现代控制系统领域中, 如专机控制系统、数控系统等, 数据采集与监测控制等已经得到广泛的应用, 大多数是在多种设备间进行数据传输, 有些是基于 windows, 但仍有一部分是基于 Linux、UNIX 等操作系统, 而且不同平台可处理的数据格式不同, 因此双方之间的通信存在很大的差异, 为了屏蔽这些差异, 构建一个跨平台的通信模型就显得尤为重要[1,2].

跨平台的通信主要是解决操作系统和数据处理格式的差异性. 数控系统中现在的跨平台通信多数是采用 socket 套接字、串行接口等实现, 通信功能多数是针对不同的需求进行多次开发, 移植性比较低, 且开发的功能与实现方式基本相相似, 浪费资源, 增长了开发周期, 而且其中处理数据的格式比较单一, 无法满足多样性的选择. 本文中采用能够跨多个平台通信的中间代理思想, 利用对底层系统进行统一封装的方法, 在应用层实现统一的调用接口来屏蔽操作系统的

① 收稿时间:2016-05-30;收到修改稿时间:2016-07-04 [doi:10.15888/j.cnki.csa.005610]

不同; 对于数据的差异性, 采用的是基于中性语言的数据编码/解码的方法[1]. 可确保不同系统平台上的终端软件都可以使用相同的通信库构成连接, 进行通信.

1　构建跨平台通信模型

多操作系统之间的跨平台通信, 实质是指在远程进程与本地进程之间建立消息传输连接后, 通过共享存储缓冲区实现消息交换, 相较于其他进程通信方法, 共享缓冲区最大的优点在于效率高. 基于中间代理的思想构建了图 1 所示的跨平台通信模型.

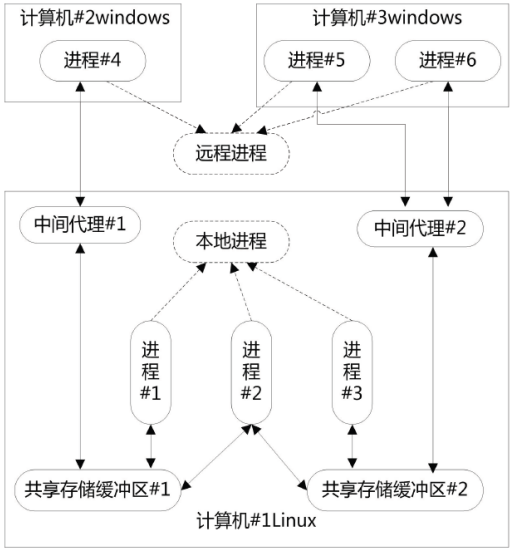

图 1　跨平台通信模型

图中计算机#1、#2 和#3 上运行的可以是相同的操作系统, 也可以是不同的操作系统. 其中进程#1、#2 和#3 通过共享存储缓冲区#1 和#2 直接进行消息的交换实现通信, 无需对消息进行编码/解码. 因为它们处于相同的计算机中, 所以通信属于本地(LOCAL)访问.

进程#4 只有通过中间代理#1, 进程#5 和进程#6 则通过中间代理#2 才可以访问共享存储缓冲区, 进行消息交换. 当进程属于不同的计算机时, 通信则属于远程(REMOTE)访问.

当进程#4 和进程#5 通信时, 它们需要分别通过中间代理#1、#2 向各自所分配的缓冲区中写入要交换给对方的数据, 然后中间代理#1 代替进程#4 访问存储进程#5 数据的缓冲区, 读取数据, 同理中间代理#2 此时可并发地读取进程#4 缓冲区中相应的数据完成进程间的通信.

中间代理主要是位于底层通信系统与应用层之间, 底层通信主要是进行数据的传输和读写、缓冲区管理、系统间建立连接等. 中间代理负责和应用层之间进行交互, 调用底层通信的接口函数进行数据的传输. 中间代理使设计人员不必理解底层通信的具体实现, 只需直接调用接口函数使用, 减轻了其负担[6-9].

2　基于RCS库实现跨平台通信的方法

本文中实现跨平台通信的方法主要是基于RCS库的 CMS 和 NML, RCS(Real-Time Control System)库是由美国国家标准与技术研究院(NIST)研发的实时控制系统, 是一个 C++类库针对多平台的实时分布式应用程序, 具有良好的跨网络、跨平台的通信能力[3,4].

CMS 是RCS 库的底层通信管理软件, 是控制系统中通信的主要部分, 通信过程中 CMS 内置函数可将所有基本的数据类型进行编码/解码, 因此, RCS 库才可以进行跨平台之间的通信. 虽然在不同平台上 CMS 的通信方法不同, 但提供了统一的接口访问共享存储缓冲区.

NML 是 RCS 库的一种独立于计算机控制系统的中性消息语言, 是基于 CMS 的通信语言. NML 是一个高效灵活的通讯机制, 适合嵌入式系统应用, 是有详细的说明的标准接口. 配置文件的使用允许使用者优化选择最高效的通讯协议, 而且没有平台的限制. 基于 RCS 库实现与其他方法的对比, 如表 1 所示.

表 1　跨平台方法实现的比较

	基于 RCS 库	其他方法如: socket、串口等
数据处理格式	中性格式, 可编码/解码所有基本数据类型	格式单一, 有一定的局限性
底层通信	不必考虑底层的实现	开发人员需学习底层硬件的端口及实现通信的方法
传输方式	共享存储缓冲区	通过串口、usb 接口、wifi 等
错误处理	封装了错误机制	需要开发人员自己封装处理
协议	支持多种协议, 不必考虑协议内容	支持协议单一, 需深入学习协议内容

NML 是 CMS 更高层次的应用程序编程接口 API, 是开放的源代码,可以从 NIST 免费获得的. 通过表 1 相对其他方法的对比, RCS 库使开发人员不必考虑底层硬件、协议等的具体实现, 直接在应用层借助 NML 实现异构平台间通信. 可见基于 RCS 库的方法简化了异构平台之间实现通信的工作, 很大程度上降低了系统开发的难度和设计人员的负担, 一定程度上可缩短 3-6 周的开发周期.

2.1 基于 RCS 库实现的通信模块

图 2 显示了利用 RCS 库实现了跨平台通信的各个模块, 主要由 NML 和 CMS 组成, NML 中的服务器相当于图 1 中的中间代理.

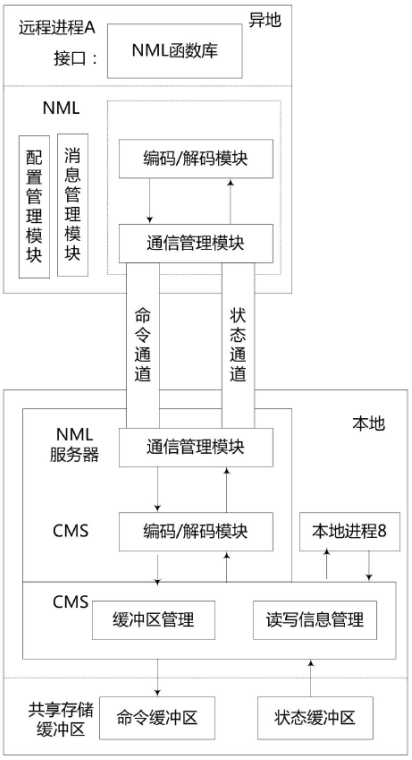

图 2 基于 RCS 库实现跨平台的通信模

接口层: NML 函数库

主要提供建立通道、读写消息等的接口函数, 对底层 CMS 提供高层次的接口函数, 如编码/解码的简化机制.

NML 层

主要是消息管理模块和配置管理模块, 如消息词汇的定义, 消息类型的声明以及 NML 应用程序中配置文件的设置等; 同样包含通信模块等.

CMS 层

给远程进程和本地 NML 服务器都提供了通信模块, 编码/解码模块, 在本地访问中还提供相应的缓冲区管理、读写信息管理[3,4].

2.2 基于 NML 的具体实现过程

通信简化过程大致如图 3 所示: 应用程序的控制模块 control()通过 NML 接口调用自定义的格式化函数 Format(NMLTYPE type, void *buf, CMS *cms)识别消息类型、建立缓冲区连接等, 调用自定义的更新函数 Update(CMS *)更新成员信息, 接着 CMS 更新函数对数据进行编码/解码, 通过命令通道和状态通道传输信息, 最后读写操作调用 CMS 通信函数进行消息的读写.

通信主要是基于 NML 的通信, NML 应用程序的设计主要有以下几步:

① 用 C++类来定义 NML 消息词库, 在头文件中进行声明;

② 创建 NML 配置文件, 指定相应的缓冲区位置以及进程如何与其连接等;

③ 创建 NML 程序中进程与缓冲区的连接;

④ 利用 NML 程序中的读写函数对信息进行传输;

⑤ 用 NML 提供的 NML Code Generator(NML 代码生成器)生成相应的可执行的 C++或者 Java 程序.

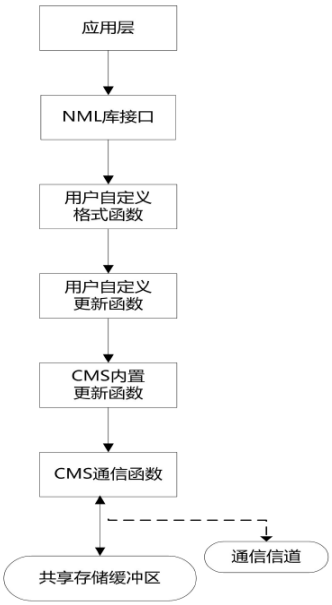

图 3 通信简化过程图

2.2.1 NML 消息定义

NML 消息可看作是, 在写操作期间复制到 NML 缓冲区中, 并在读操作过程中复制出的数据结构. 每个消息有唯一的正整数标识符, 来识别缓冲区中现存储的消息[8-13]. 如在头文件中对数据进行定义的代码:

```
#include "rcs.hh"
enum    exCmdMsgType {
   EX_CONFIG_TYPE=6001,
   EX_GOTO_POINT_TYPE=6002,
```

```
    EX_HALT_TYPE=6003,
    EX_INIT_TYPE=6004
};
enum   exStatMsgType {
    EX_STATUS_TYPE=6000,
    EX_RCS_DONE = 1,
    EX_RCS_EXEC = 2,
    EX_RCS_ERROR = 3
};
```

上述命令消息都继承 RCS_CMD_MSG 类, 例如:

```
class EX_CONFIG : public RCS_CMD_MSG
{
    public:
    //构造函数
    EX_CONFIG();
    CMS 内置的更新函数
    void   update(CMS *);
};
```

状态信息继承 RCS_STAT_MSG 类, 代码如下所示:

```
class EX_STATUS : public RCS_STAT_MSG
{
    public:
    //基本构造函数
    EX_STATUS();
    //继承于类的构造函数
    EX_STATUS(NMLTYPE t, size_t s) :
        RCS_STAT_MSG          (t,s) {};
    //CMS 更新函数
    void   update(CMS *);
    //自定义的变量
    PM_CARTESIAN position;
};
```

另外, 消息类用格式化函数 Format()来建立进程与缓冲区的连接, 识别当前所需处理的消息类型, 使通信过程中 CMS 能正确及时处理缓冲区中的多种类型信息, 如:

```
int the_format(NMLTYPE type,void
  *buf,CMS *cms)
{
    switch(type){
    //可添加多种需处理的消息类型
    case   EX_MSG_TYPE:
      ((EX_MSG *)buf)->update(cms);
      break;
    default:
      return -1;
  }
    return 0;
}
```

2.2.2 NML 配置文件的设置

通信中使用配置文件来设置 NML 缓冲区、进程的相关参数. 开发人员大多数是在配置文件中指定参数选项, NML 调用通信函数时配置文件作为参数读取, 得到相关信息.

配置文件中的 NML 缓冲区是一段有固定最大值的存储空间, 缓冲区行是以大写字母 B 开头作为指示, 必须含有 Name: 进程识别创建或连接缓冲区的名字; 类型 Type: 进程访问缓冲区的方式, 本地有 SHMEM、GLOBMEM、FILEMEM 等方法, 远程则选用 TCP、UDP 等网络协议; Host: 缓冲区所在的主机名或 IP 地址; Size: 缓冲区容量的大小值; Number: 一个 integer 唯一标识缓冲区的序号; MP: 可访问缓冲区的最多进程数; 数据格式 Neutral: 值为 0 是本地格式(native), 为 1 是独立于计算机的中性格式(neutral format)等参数.

配置文件中 P 开头表示进程行, 进程一般有 Name: 进程名; Buffer: 进程连接的缓冲区; Type: 进程类型, 表示进程与缓冲区是否在同一主机上, 关键字是本地 Local 和远程 Remote; Host: 进程坐在计算机的主机名或 IP 地址; OPS: 进程的操作类型, 读 R、写 W 或者读写 RW; Server: 服务器标志, 值为 0 表示不是服务器进程, 为 1 是服务器进程, 为 2 表示不是服务器进程但本身可创建一个服务器进程; Timeout: NML 操作的等待时间; Master: 进程连接饿得缓冲区是否为本进程创建等参数[9], 设置图 1 所示的跨平台通信模型的配置文件如表 2.

表 2 NML 配置文件设置缓冲区

#	Name	Type	Host IP	Size	Numb	Neut	MP
B	Buf1	SHMEM	192.168.1.2	512	1	0	4
B	Buf2	SHMEM	192.168.1.2	1024	2	0	5

表 3 NML 配置文件设置进程

#	Name	Buff	Host IP	Type	OPS	Server
P	Pro1	Buf1	192.168.1.2	Local	W	0
P	Pro2	Buf1	192.168.1.2	Local	RW	0
P	Pro2	Buf2	192.168.1.2	Local	RW	0
P	Pro3	Buf2	192.168.1.2	Local	RW	0
P	Pro4	Buf1	192.168.1.3	Remote	W	0
P	Pro5	Buf2	192.168.1.4	Remote	RW	0
P	Pro6	Buf2	192.168.1.4	Remote	RW	0
P	Buf1 SRV	Buf1	192.168.1.2	Local	RW	1
P	Buf2 SRV	Buf2	192.168.1.2	Local	RW	1

2.2.3 NML 通道的建立

当图 2 中远程进程 A 向本地进程 B 发送命令时, 进程 A 的通信管理模块则需调用 setCmdChannel(new RCS_CMD_CHANNEL(modAFormat,"modA_cmd","modA","test.nml"))函数来建立命令通道,而同样当进程 A 从状态缓冲区中读取进程 B 的状态信息时, 则需调用 setStatChannel(new RCS_STAT_CHANNEL (modAFormat,"modA_sts","modA","test.nml"), modA_status) 函数建立状态通道, 调用函数 modB_sub_num = addSubordinate(new RCS_CMD_CHANNEL(modBFormat,"modB_cmd", "modA", "test.nml"), new RCS_STAT_CHANNEL(modBFormat, "modB_sts", "modA", "test.nml"))与进程 B 进行通道连接. 当通道建立完成以后需要 DECISION_PROCESS()决策函数来调用与当前命令类型合适的命令函数, 发送命令给其他函数. int addSubordinate()返回一个整型数, 当进程 A 调用 sendCommand 发送命令时, 返回值用来识别与其通信的进程.

通信过程中, NML 通道上传输的数据是编码后与平台无关的中性数据格式, 如: XDR、XML 等格式. 消息到达各自平台时再将数据解码成平台可处理的相关格式.

2.2.4 NML 服务器

远程进程需要通过服务器才能连接到本地缓冲区, NML 服务器是一个进程, 运行在本地. 建立服务器连接:

```
#include "nml.hh"
#include "exvocab.hh" //使用 NML 词汇集
void contact(){
    //连接到 NML 缓冲区
NML * command_buffer = new
        NML(ex_format,"command", "server",
        "test.nml");
NML * status_buffer = new NML(ex_format,
    "status", "server","test.nml");
//为两个缓冲区运行 NML 服务器
run_nml_servers();
}
```

2.2.5 NML 读写信息

基于 NML 的通信, 主要是对缓冲区进行数据的读写操作, 对于编码的信息, 还需调用更新函数返回本地数据格式.

NML 常用的是非阻塞写 write(NMLmsg *msg)函数, 返回值为 0 表示成功写入消息, 返回值为-1 表示失败. 如果写消息是队列存储, 则不存在重写覆盖原有消息, 若是非队列存储, 会出现重写覆盖, 因此采用非破坏写函数 write_if_read(NMLmsg *msg), 当缓冲区为空时直接写入, 否则检测信息是否已被读取, 已被读取直接写入新消息, 否则直至消息被读取才可以写入新消息. 写操作如下:

```
#include "nml.hh"
#include "exvocab.hh" //使用 NML 词汇集
//写操作
//连接 NML 命令缓冲区
NML * command_buffer = new
        NML(ex_format,"command",
        "server","test.nml");
EX_MSG   *ex_msg;
ex_msg->d = 12.3456;
ex_msg->c = 'a';
//写信息
if( 0!= command_buffer->write(ex.msg))
{
    printf("error writing:%d\n",
      command_buffer->error.type);}
```

NML 的读函数有非阻塞读 read()、阻塞读 blocking_read()以及监听读 peek()等, 所有读函数的返回值有三类, 返回 0 表示没有新的消息, -1 表示出错, 其他值表示定义消息时的类型值, 则读操作成功, 随后进程对读到的消息进行复制到其缓冲区, 然后可以通过 get_address()函数得到消息所在位置的指针. 读操作如下:

```
#include "nml.hh"
#include "exvocab.hh" //使用 NML 词集
```

```
// 读操作
//连接 NML 状态缓冲区
NML * status_buffer = new NML(ex_format,
   ”status”, ”server”,”test.nml”);
 switch(status_buffer->read())
      {
        case -1:   //NML 出错
          rcs_print( "An error occurred.\n");
          break;
        case 0: //没有新的信息
          rcs_print( "The same message.\n");
          break;
        case EX_MSG_TYPE:
          ex_msg_ptr=(EX_MSG*)
            status_buffer->get_address();
            rcs_print("Have a new   message.\n”);
            rcs_print(" The value of its members
                  are:\n");
            rcs_print("d=%f, c=%c\n",
                  ex_msg_ptr->d,ex_msg_ptr->c);
            break;
      }
```

在读取消息之前可用函数 check_if_read()检测缓冲区消息是否被读取, 对于非队列存储, 返回值为 0 表示有一条未被任何进程读取的新消息, 为 1 表示有一条消息至少已被一个进程读取, 若是队列存储的消息, 为 0 表示该队列中至少有一条信息, 为 1 表示该队列为空. 进程从非队列存储的缓冲区中成功读取消息时, 对其他进程读取该消息不产生影响. 若进程从队列存储缓冲区读取消息成功, 则其他进程已无法读取到该消息.

通过 NML 服务器进行的读写操作则是远程读写操作, 远程读写操作和本地读写操作源代码一样, 只是 NML 配置文件的设置不同.

远程写操作是远程进程将含有消息、目标缓冲区(命令缓冲区)的写操作请求发送给服务器, NML 服务器像本地写操作一样将消息写入目标缓冲区中.

远程读操作是远程进程向服务器发送读请求的目标缓冲区(状态缓冲区), 服务器读取本地状态缓冲区的中信息内容, 然后将结果通过网络发送给远程进程. 远程读操作有轮询(Polling)、订阅(Subscribing)[9].

在进行消息的读写时, 由于缓冲区资源共享, 必然会出现多个读写进程同一时间对同一个缓冲区进行操作, 则可能出现信息的不可靠, 所以设计时应考虑进程之间的同步操作如何解决, RCS 库是一个完善的设计工具, 为此提出了互斥机制 mutex 解决该同步问题, 当读写进程访问共享缓冲区时, 将共享缓冲区锁定为其使用, 在释放之前不允许其他进程访问, 必须等共享的资源释放.

3 结语

随着现代工业控制的发展, 监测控制系统的多样性使数据采集等操作都是在异构平台之间进行, 因此跨平台的通信已然成为一种趋势. 本文中跨平台通信的设计方案, 给异构平台之间互相通信带来了极大地便利. 本文提出了基于 RCS 库中 NML/CMS 的共享缓冲区方法实现不同平台间进程的通信, 各进程可在主流的操作系统, 如 Windows、Linux、UNIX 和 Mac 等上运行, 高效地实现了跨平台的数据通信,而且使通信系统趋于标准化, 使其应用更加地广泛.

参考文献

1 孙涛,王品,李家霁.开放式数控系统中跨平台通信技术的研究.组合机床与自动化加工技术,2010,12:50–53.

2 李校慧,于东,等.数控系统网络通信平台的设计与实现.计算机工程与设计,2013,34(4):1141–1146.

3 田军锋,马跃,等.利用 RCS 库实现数控系统模块间的通信.数控技术,2012,25(7):121–122.

4 王锋.FMS 递阶分布式控制系统建模及通讯机制研究基[硕士学位论文].北京:中国科学院研究生院,2010:24–31.

5 高甜容,于东,秦承刚,胡毅,岳东峰.数控系统中模块间通信方法的设计与实现.计算机工程,2010,12:238–241.

6 李风忠,穆斌.跨平台通信中间件的研究与实现.计算机光盘软件与应用,2011,18:56–57.

7 谢永刚,范琳,王忠民.基于共享内存的进程间通信在嵌入式软件测试中的应用.计算机应用与软件,2011,2:106–108.

8 苏红旗,刘官树.一种基于内存共享的高效进程间通信机制.新型工业化,2014,2:67–73.

9 Shackleford W. Real-time control systems library: Software and documentation. http://www.isd.mel.nist.gv/projects/rcslib. [2014-07-16].

10 周西峰,陆鹏,郭前岗.利用 socket 实现 Windows 与 Linux 平台间的网络通信.微型机与应用,2013,18:49–51.

11 Dellinger E. Cross-platform communication enables continuous process improvement. Control Engineering, 2013, 60(4).

12 Liang HB. A distributed real-time control system based on RCS library. Information & Control, 2011, (5): 581–583.