第 33 卷 第 9 期 计 算 机 工 程 2007 年 5 月
Vol.33 No.9 Computer Engineering May 2007
·开发研究与设计技术· 文章编号：1000—3428(2007)09—0266—03 文献标识码：A 中图分类号：TP393

嵌入式环境下 RPC 的设计与实现

袁 菲，陆 阳，海 深
(合肥工业大学计算机与信息学院，合肥 230009)

摘 要：远程过程调用(RPC)是分布式系统中常见的进程间通信手段，与显式的消息传递方法相比，RPC能为多节点进程间通信提供更好的透明性。该文在分析了RPC系统结构的基础上，结合嵌入式环境设计了一个通用的RPC系统，并在ARM7、μC/OS-II、以太网和TCP/IP的基础上实现了该RPC设计。通过测试，比较了RPC方式与显式消息传递的时间损耗。

关键词：远程过程调用；嵌入式；透明性；μC/OS-II

Design and Implementation of RPC for Embedded Environment

YUAN Fei, LU Yang, HAI Shen
(College of Computer and Information, Hefei University of Technology, Hefei 230009)

【Abstract】 RPC is a common method used to inter-process communication of distributed system. RPC can provide more transparence than traditional explicit message passing. This article designs a general RPC system for embedded environment based on analysis of the RPC system architecture. This RPC system is implemented upon ARM7, μC/OS-II, Ethernet, TCP/IP. The tests of RPC and explicit message passing are performed.

【Key words】 Remote procedure call(RPC); Embedded; Transparence; μC/OS-II

1 概述

在嵌入式环境下，节点间的通信往往是通过显式地发送和接收消息进行的，网络的开发者需要了解网络接口及使用方式，通信过程对开发者来说不是透明的。这样做的优点是：通信效率高，可以针对发送的数据类型和语义进行优化，资源占用少，响应速度快。但是它的缺点也很明显，就是不能有效地对程序开发者屏蔽通信过程，增加了开发的复杂性，尤其是在大型系统中。一方面，随着处理器计算能力的不断提高，提升性能、减少资源占用将不是开发中首要考虑的问题；另一方面由于嵌入式应用领域和应用规模的不断扩大，人们对于隐藏通信过程来降低系统的复杂性提出了更高的要求。为了提高节点间通信层次，隐藏通信细节，提高开发效率，本文提出了将远程过程调用(Remote Procedure Call，RPC)应用于嵌入式环境下。

从概念上，节点间的通信可以被划分成 4 种模式[1]：客户机/服务器模式(Client/Server，CS)，远程计算模式(Remote Evaluation，REV)，代码点播模式(Code On Demand，COD)和移动Agent模式(Mobile Agent，MA)。在REV和COD模式中，代码是向资源或数据移动的，这些模式适用于代码较少而数据相对庞大的场合，而且传递代码比传递数据要复杂，显然这 2 种模式不适用于一般的嵌入式环境；而MA模式是上述模式中功能最为强大的，实现难度也是最大的，因此从隐藏通信过程的角度来说，MA模式并不合适。在CS模式中，RPC是最常见、最有效的节点通信方式之一，RPC的概念较早出现于 1981 年B. J. Nelson的博士论文。RPC应用广泛，是许多分布式系统的基础，如Sun RPC，开放软件基金会(OSF)的DCE，CORBA，COM和Java的远程方法调用(RMI)[2]。近年来，RPC与Web技术和XML技术相结合出现了XML-RPC[3]，将RPC的发展推向了新的高度。

RPC 利用过程调用实现多节点的进程间通信，但是又具有与本地过程调用类似的形式，在使用方式上 RPC 调用和普通的过程调用没有区别，所以它可向开发者提供良好的接口。使用 RPC 的过程中，开发者看不到任何显式的消息传递过程。通过对 RPC 的调用参数进行约束和规范，可在不同的平台间实现异构环境的 RPC 调用。另外由于 RPC 需要调用那些封装好的过程，强迫 RPC 设计者提供设计良好的接口，因此在一定程度上降低了模块间的耦合性，提高了模块化程度。

2 RPC 设计

2.1 RPC 原理

从调用方发出 RPC 请求后，根据调用进程是否被挂起可以将 RPC 分成异步 RPC 和同步 RPC。在同步 RPC 中，调用进程会被挂起，异步 RPC 反之，本文中所描述的 RPC 均指同步 RPC。

RPC的基本过程是[4]：节点A调用节点B上的过程时，节点A上的调用进程将被挂起，而B上被调用的过程开始执行。调用方使用参数将信息提供给被调用方，然后被调用方用返回值的形式将结果传回调用方。

如前所述，RPC 中的过程调用与本地的过程调用的形式是类似的，但由于 RPC 在本质上是涉及网络通信的位于不同地址空间的进程间通信，因此实际的 RPC 过程比一般的本地调用要复杂。其基本结构如图 1 所示。在 RPC 的调用方(客户机)中，存在一个为调用方整理参数，并调用操作系统 API 和网络接口进行消息发送的程序，这个程序被称为客户存根(client stub)，而整理参数的过程被称为参数编组(parameter marshaling)；在被调用方(服务器)，RPC 守护进程会解析 RPC

作者简介：袁 菲(1979－)，男，硕士生，主研方向：嵌入式系统应用；陆 阳，教授、博导；海 深，硕士生

收稿日期：2006-05-28 **E-mail：**yuanfei.china@126.com

请求，并为其调用合适的服务器存根，服务器存根(server stub)会将 RPC 请求中的编组参数解编组成适当的参数，然后调用实际的过程，并将结果编组发给调用方。

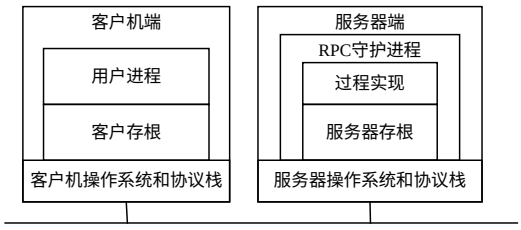

图 1 RPC 系统结构

因此，总的说来，一般的 RPC 过程包括以下的步骤：

(1)客户进程以正常的方式调用客户存根；

(2)客户存根对参数编组并生成消息，然后调用本地的操作系统和网络接口；

(3)客户操作系统将消息发到远程节点的 RPC 守护进程；

(4)RPC 守护进程提取编组后的参数，将其传递给服务器存根；

(5)服务器存根对参数解编组，然后调用实际的过程实现；

(6)执行实际的过程，将返回的结果返回给服务器存根；

(7)服务器存根将结果编组并打包成消息，然后调用本地的操作系统和网络接口；

(8)远程节点的操作系统将结果发送回调用方的操作系统；

(9)调用方的操作系统将消息交给客户存根；

(10)客户存根对编组后的结果进行解编组，并返回给调用它的过程。

2.2 总体设计

在本文所述的设计中，基本遵循如图 1 所示的模型结构。在客户机端 RPC 系统由应用程序和客户存根组成，在服务器端为了处理多个并发的 RPC 请求，在服务器存根和 RPC 守护进程间增加了一个工作进程，这样 RPC 就由 4 部分组成：实际的过程实现，服务器存根，工作进程和 RPC 守护进程。实际的过程实现是客户端存根的远程实现，其在形式上和普通的函数并无区别。其它各个部分的描述见 2.3 节和 2.4 节。

在本文的实现中采用ARM7(Philips LPC2210)为处理器，μC/OS-II为底层操作系统，以太网和TCP/IP协议栈为网络环境。一般来说，同步RPC设计目标主要有 2 个：高吞吐量和低延时，且二者不可兼得[5]。由于在嵌入式环境下，特别是控制环境下，节点间通信的数据量比较小，对实时性有一定的要求，因此在本文所述的设计中是以低延时作为主要目标。

2.3 存根

客户存根和服务器存根是RPC的重要组成部分，担负着屏蔽通信细节和编组(解编组)数据的功能。生成存根有 2 种方法：自动生成或人工编写。自动生成是用接口描述语言(Interface Description Language，IDL)或其它与平台无关的语言描述接口，然后用相应的编译器生成平台相关的存根[6]；人工编写是用某一种语言直接开发与平台相关的存根。

客户端存根最重要的功能是对参数进行编组并将其打包到消息中。尽管 RPC 调用与本地调用十分类似，但执行的过程毕竟是在两地址空间中进行的，因此对传递的参数需要做一些约束，或者说对客户端存根的接口需要做一些约束，这些约束既有利于简化存根的人工编写和自动生成，也是 RPC 协议制定的重要内容。需要考虑的情况有：

(1)参数编组的顺序：各种语言对参数入栈的顺序是不一样的，在 RPC 的调用端和被调用端的程序不一定是用同一种语言编写的，因此必须考虑参数编组的顺序。在设计中规定参数传递的顺序是从左至右，且不允许可变的参数个数，这与 C 语言的从右至左的入栈顺序不同。

(2)数据的表示问题：在不同的机器上对数据可能有不同的解释，如在 8 位机上，可能用 16 位来表示一个 int 类型的数据，而在 32 位机上，int 可能是一个 32 位的数。另外机器还存在字符编码和大小端(endian)等问题。因此通信双方必须就一些最基本的数据类型达成一致。在设计中将基本的数据类型用宏进行(C 语言)重新映射，且在存根中只使用映射后的数据类型，在不同的平台上需要改变映射关系。在 ARM 平台上的对 C 语言基本类型的映射见表 1，且规定小端数据为标准格式。

表 1 ARM 上 C 语言中基本数据类型的映射

基本类型名	映射后的类型名
unsigned char	uint8
signed char	sint8
unsigned short	uint16
signed short	sint16
unsigned int	uint32
signed int	sint32
float	fp32
double	fp64

(3)参数引用的类型：一般说来，主要有 2 种类型的引用：按值调用(call-by-value)和按引用调用(call-by-reference)。对按值调用的参数一般只是简单的复制到堆栈中，被调用过程对复制值的修改不会影响原来调用端的值。按引用调用中传递的可能是指向变量的地址(C/C++)或对变量的引用(C++)，因此解决起来不能像按值调用那样简单，具体情况参见(5)。

(4)输入和输出参数：客户存根和服务器存根除应充分了解调用过程的参数个数和参数的类型外，还应该了解参数的方向是输入还是输出。对输出参数，客户机没必要将其编组进消息；对输入参数，服务器没必要将其返回给调用者；只有对那些既是输入又是输出的参数才需要双向的传递。对输入和输出参数进行区别，可以有效提高数据传输的效率。

(5)指针(引用)参数：指针是一类特殊的参数，指针中保存的是地址，将调用者的地址发送给被调用者是没有意义的，因为 2 个程序运行在不同的地址空间中。最简单的解决办法是禁止使用指针参数，但由于指针是相当重要，某些情况下又不能完全禁止，如 C 语言中数组值的传递，因此一个更符合逻辑的解决办法是对指针指向的内容进行编组，且规定指针的层次，在本文中规定指针只能为一层。如果过程在参数中使用了指针，在客户机端就必须对指针指向的内容进行编组，在服务器端也必须把一个指针传递给实际的实现过程，但这两个指针具有不同的语义。

(6)通信错误的处理：由于 RPC 的通信要通过网络，需要客户机和服务器两方的参与，不能保证在任何时刻远程过程都是可用的。因此需要在存根中加入对通信错误的检测，在本文的设计中，主要检测的是超时错误，且由客户端进行。

综合以上考虑，客户机存根和服务器存根的伪代码见图 2 所示。客户机存根由用户进程调用，而服务器存根由服务器的工作进程调用。从图中服务器存根的伪代码可以看出服务器存根的接口形式相对单一，主要的参数就是编组后的参数列表和编组后的返回值列表。

返回值 客户机存根(参数 1, 参数 2, ..., 参数 n)
{
编组参数(参数 1, 参数 2, …, 参数 n);
将编组后的参数列表打包进 RPC 请求;
调用网络接口发送消息;
等待调用返回;
// 调用进程挂起
if (超时错误) {
出错处理;
返回出错代码;
}
解编组返回值;
将返回值返回给调用者;
}
	服务器存根(编组后的参数列表, 编组后的返回值列表)
	{
	参数 1, 参数 2, …, 参数 n = 解编组参数(编组后的参数列表);
	返回值 = 调用实际的实现过程(参数 1, 参数 2, …, 参数 n);
	编组返回值(返回值);
	}

图 2 客户机和服务器存根的伪代码

2.4 RPC 守护进程及工作进程

当客户机存根将编组后的参数列表发送后，由服务器端的 RPC 守护进程负责接收，并根据 RPC 请求报头中的描述将 RPC 请求传递给正确的存根程序。一个完整的 RPC 请求报由 RPC 请求报头和编组后的参数列表两部分组成。在 ARM 平台上用 C 语言描述的 RPC 请求报头如下：

```
typedef struct {
  uint8  type;          // 表示这是一个 RPC 调用
  uint32 code;          // 调用的存根编号
  uint32 len;           // 参数列表长度
}RPC_REQ_HEAD;
```

服务器存根在能被客户端进程调用前必须向 RPC 守护进程进行注册，在注册时需要向服务器存根提供一个存根编号，只有注册成功的服务器存根才能被调用。当 RPC 请求被服务器接收时，RPC 守护进程用存根编号在 RPC 服务器存根注册列表中查找对应的存根，找到后则进行调用，否则返回一个错误消息，关于服务器存根注册列表的描述如下：

```
typedef void (*server_stub)(void *para, uint32 len, void *resultdata, uint32 resultlen);
  // 服务器存根接口
typedef struct {
  uint32  code;
  // 服务器存根编号
  server_stub stub;
  // 服务器存根的入口地址
}STUB_ITEM;
  // 服务器存根注册项
STUB_ITEM  Stub_List[RPC_SERVER_STUB_NUM];
  // 服务器存根注册表
```

为了提高服务器 RPC 守护进程的工作效率，真正的 RPC 调用过程由工作进程来完成。如果不使用工作进程，直接由 RPC 守护进程调用服务器存根可能会导致新的 RPC 请求无法得到响应。RPC 守护进程用消息队列将服务器存根入口地址、编组参数和网络接口等信息交给工作进程，当工作进程进行 RPC 调用时，RPC 守护进程仍可接收新的 RPC 请求，调用结束后由工作进程将编组后的结果发回调用方。

2.5 RPC 测试

测试主要考察在 RPC 调用过程中，RPC 相关操作对调用时间的影响。测试所使用的过程接口为 uint32 WriteData(uint8 *data, uint32 len); 该过程中将首地址为 data，长度为 len 的数据从客户端发送到服务器端，服务器端对数据进行验证，若正确则认为调用成功，服务器返回接收到的数据长度，否则返回 0。作为对比，测试了直接使用消息完成同样的功能所使用的时间。由于嵌入式环境下数据量比较小，因此规定最大数据长度为 1 024，这个长度可以满足大多数进程通信的要求。测试结果见表 2。

表 2 UDP、TCP 实现的 RPC 调用和直接消息传递的耗时(ms)

数据长度	UDP 消息	RPC(UDP)	TCP 消息	RPC(TCP)
1	22.11	22.47	2 021.89	2 024.60
2	22.41	22.61	2 022.64	2 028.84
4	21.87	21.80	2 022.58	2 054.34
8	20.97	22.42	2 020.79	2 054.45
16	21.46	24.84	2 030.59	2 054.32
32	25.32	25.75	2 039.77	2 034.00
64	25.17	26.16	2 027.31	2 029.53
128	25.39	26.18	2 024.34	2 024.36
256	45.01	45.83	2 034.88	2 036.59
512	50.24	50.60	2 062.33	2 063.75
1 024	80.10	79.89	2 096.28	2 107.00

从测试结果可以看出，RPC 相关操作对调用时间的影响极低，基本实现了低延时的设计目标。RPC 方式与直接使用消息的传递在时间消耗方面相差不大。在嵌入式环境下使用 UDP 协议比使用 TCP 协议有较大的优势，TCP 消耗的时间较长，主要是因为 TCP 协议是面向可靠连接的，在建立连接、管理连接和释放连接等方面都要消耗一定的资源，而且还具有流量控制的功能，这些特性 UDP 协议都没有，因此 TCP 的处理过程比 UDP 要复杂得多。在嵌入式环境下，网络中传输的数据量比较少，对带宽的需求不是很高，因此 UDP 协议更适合用作嵌入式环境下的 RPC 实现基础。分析 UDP 协议的损耗时间，可以发现，当传递的数据大于 256 字节时，消耗的时间有较大幅度的增加，因此对测试所使用的过程在传递小于 256 字节的数据时效率较高。

3 结束语

RPC 是分布式系统中常见的进程间通信手段，但在嵌入式环境下使用 RPC 方式进行通信并不常见。本文分析了 RPC 系统的结构，结合嵌入式环境对 RPC 进行了设计，并在 ARM7、uC/OS-II、以太网和 TCP/IP 的基础上实现了 RPC 的设计。通过测试，验证了本文实现的 RPC 在调用性能上能够与直接使用消息的方式看齐，但 RPC 能为进程间通信提供更好的透明性。尽管 RPC 为节点间的通信提供了一定的透明性，但由于网络通信的过程毕竟不是在一个地址空间中进行的，为了解决更复杂的情况，还需要在 RPC 中做更多深入细致的研究。

参考文献

1 Fuggetta A, Picco G P, Vigna G. Understanding Code Mobility[J]. IEEE Transactions on Software Engineering, 1998, 24(5): 342-361.

2 Vinoski S. RPC Under Fire[J]. IEEE Internet Computing, 2005, 9(5): 93-95.

3 Dissanaike S, Wijkman P, Wijkman M. Utilizing XML-RPC or SOAP on an Embedded System[C]//Proc. of Distributed Computing Systems Workshops. 2004: 438-440.

4 Tanenbaum A S, Van Steen A. 分布式系统：原理与范型[M]. 杨剑峰, 译. 北京: 清华大学出版社, 2004: 54-64.

5 陈良宽, 王雅红. 通用远程过程调用的设计与实现[J]. 小型微型计算机系统, 1996, 17(2): 33-37.

6 毛迪林, 陈向阳, 高传善. RPC 语言自动生成器——rpcgen[J]. 计算机工程, 1994, 20(3): 27-31.