

组织：中国互动出版网 (<http://www.china-pub.com/>)

RFC 文档中文翻译计划 (<http://www.china-pub.com/compters/emook/aboutemook.htm>)

E-mail: ouyang@china-pub.com

译者：马东辉 (eaststone ma_donghui@263.net)

译文发布时间：2001-3-28

版权：本翻译文档可以用于非商业用途自由转载，但必须保留本文档的翻译及组织信息。

Network Working Group

Sun Microsystems, Inc.

Request for Comments: 1050

April 1988

远程过程调用协议规范

(RFC1050---Remote Procedure Call Protocol Specification)

摘要

远程过程调用 (Remote Procedure Call) 可以使程序调用远方节点上的过程象调用本地过程一样方便。本文就是远程过程调用协议规范的中文版。

目录

1. 简介.....	1
2. 术语.....	2
3. RPC 的模型.....	2
4. 传输和语义.....	2
5. 绑定与集合独立性.....	3
6. 认证.....	4
7. RPC 协议的要求.....	4
8. RPC 消息协议.....	7
9. 认证协议.....	10
10. 记录标记的标准.....	16
11. RPC 语言.....	16
附录 A: 端口映射器程序协议.....	18

1. 简介

此文档详细说明了一个使用在实现 Sun 公司的远程过程调用(RPC)包中的消息协议。此消息协议是由外部数据描述(XDR)语言[9]来定义的。这篇文档假定读者对 XDR 非常熟悉，它并不试图去证明使用 RPC 的好处。这里推荐由 Birrell and Nelson [1]所写的文章，作为了解 RPC 的背景。

2. 术语

此文档讨论了服务器，服务，程序，过程，客户和版本这些术语。服务器就是实现网络服务的软件。网络服务是一个或多个远程程序的集合。一个远程程序实现了一个或多个远程过程；这些过程的参数和结果已经存档在特定的程序协议规范中（见附录 A 中的例子）。网络客户是向服务发出远程过程调用的软件。一个服务器可能支持不止一种版本的远程程序，这样以便于与改变的协议兼容。

例如，一个网络文件服务可能由两个程序组成。一个程序处理诸如文件系统访问控制和锁定这样的高层应用。另一个程序处理低层的文件 I/O，拥有象“read”和“write”这样的过程。网络文件服务的客户机根据自己用户的需要将会调用服务中与这两个程序关联的过程。

3. RPC 的模型

远程过程调用模型与本地过程调用模型非常相似。在本地过程调用中，调用者把要传给过程的参数放在明确定义好的位置上（例如一个结果记录）。然后调用者将把控制转交到被调用的过程中，最后得到返回的控制。在返回点上，被调用的过程的结果从明确定义好的位置上取出，调用者继续执行。

远程过程调用也是相似的，通过两个进程逻辑地运行在一起构成一条控制主线。一个是调用者进程，另一个是服务器进程。也就是说，调用者进程发送给服务器进程一条调用消息，并等待（阻塞）直到收到响应的消息。调用消息包含被调用的过程的参数等。响应消息包含着被调用过程的结果等。一旦收到响应消息，被调用过程的结果就会被取出，调用者将恢复执行。

在服务器一侧，一个进程处于休眠状态来等待调用消息的到达。当一条调用消息到达后，服务器进程从消息中取出被调用过程的参数，计算出结果，发送响应消息，然后等待下一条调用消息。

注意：这种模式中，在任一个给定的时间上，两个进程中只有一个是激活的。但是，这种模式只是作为一个例子。RPC 协议在并发模型的实现上不作限制，也可能存在着其它的实现并发模型的方法。例如，一种实现可能选择 RPC 调用是异步进行的，这样客户机就可以在等待服务器的响应中，做其它有用的工作。另一种可能性是使服务器创建一个新的任务来处理输入的请求，这样服务器就可以自由地接收其它的请求。

4. 传输和语义

RPC 协议不依赖于传输协议。也就是说，RPC 不关心消息是怎样从一个进程传递到另一个进程中去的。这个协议仅仅处理协议的规范和消息的解释。

还有必要指出 RPC 并不去试图实现任何一种可靠性，所以应用程序必须考虑在 RPC 下层的传输层协议。如果应用程序知道自己运行在象 TCP/IP[6]这样的可靠传输层的上层的时候，它就知道保证可靠性的大部分工作已经做好了。在另一方面，如果应用程序运行在象 UDP/IP[7]这样的不可靠传输层的上层，那么它必须实现自己的重传和超时策略，而这些服务 RPC 层是不提供的。

因为独立于传输层，RPC 协议并不把特殊的语义附加到远程过程上。可以从下面的传输层推断出来（但是应该有明确的定义）。例如，考虑 RPC 运行在不可靠传输层 UDP/IP 的上层。如果应用程序在很短的超时时重传 RPC 消息，当没有接收到响应的时候，它能够判断的唯一的的事情就是过程没有执行或者执行了一次以上。当收到响应的时候，它可以推断出过程至少执行了一次。

服务器可能希望记住以前准许的从客户端发来的请求。为了在某种程度上确保至多只执行一次的语义，服务器不再重新批准这些请求。服务器通过利用打包在 RPC 请求中的事务 ID 来实现这个功能。事务的主要用处就是客户端的 RPC 层用它来匹配对请求的响应。但是，客户应用程序当重传一个请求的时候可以选择再使用它以前的事务 ID。服务器应用程序在知道了这个事实后，可以选择在准许了一个请求后记住这个事务 ID，为了获得在某种程度上至多只执行一次的语义，服务器对于具有相同 ID 的请求不再重新批准。除了可以进行检验相等的操作之外，不允许服务器使用其它的方法来检查这个 ID。

另一方面，如果使用了一个象 TCP/IP 这样的可靠传输，应用程序能够从一条响应消息推断出过程已经正确地执行了一次。但是，如果它没有接收到响应消息，则不能假设远程过程没有执行。注意：即使使用了象 TCP 这样的面向连接的协议，应用程序仍然需要超时和重新连接的功能来处理服务器崩溃的情况。

除了数据报或者面向连接的协议之外，还存在其它的传输层协议的可能性。例如，一种象 VMTP[2]这样的请求响应协议对 RPC 来说也许是最自然的传输层协议。

注意：在 Sun 中，RPC 当前实现在 TCP/IP 以及 UDP/IP 的上层。

5. 绑定与集合独立性

绑定一个客户端到一个服务上的行为并不是远程过程调用规范的一部分。这个重要且必要的功能留给了一些上层的软件（这个软件可能本身就使用 RPC，见附录 A）

执行者应该把 RPC 协议看作是网络上的跳转到子程序指令（“JSR”）；装载程序（绑定者）使 JSR 可用，装载程序本身使用 JSR 来完成它自己的任务。同样，网络使 RPC 可用，使用 RPC 来完成它的任务。

6. 认证

RPC 协议提供了一些必要的字段使客户能向服务标识它自己，反过来服务也要用一些字段来向客户标识自己。安全和访问控制机制建立在消息认证之上。一些不同的认证协议能够得到支持。在 RPC 报头中有一个字段用来指出使用哪一种认证协议。关于更详细的认证协议的信息在第九节“认证协议”中讨论。

7. RPC 协议的要求

RPC 协议必须要能提供以下的功能：

- (1) 一个调用过程的唯一规范
- (2) 匹配响应消息和请求消息的规则
- (3) 服务鉴别调用者和调用者鉴别服务的规则

除了以上的要求，还需要能检测出以下由于协议，实现，用户和网络管理中所产生的错误：

- (1) RPC 协议不匹配
- (2) 远程程序协议版本不匹配
- (3) 协议错误（诸如错误指定了过程参数）
- (4) 远程认证失败的原因
- (5) 不能调用想要的过程的其它原因

7.1 RPC 程序和过程

RPC 调用消息有三个无符号的字段：远程程序号，远程程序版本号和远程过程号。这三个字段唯一地标识了被调用的过程。程序号由一些主要的权威来管理（就象 Sun 公司）。一旦执行者有了一个程序号，他就可以执行他的远程程序；第一次执行很可能有一个版本号 1。因为大多数新的协议都发展成更好，更稳定，更成熟的协议，所以调用消息中的版本字段标识了调用者使用了协议的哪一个版本。版本号使得通过相同的服务器进程运行旧协议和新协议成为可能。

过程号标识着被调用的过程。这些号码已经归档在特定的程序的协议规范中。例如，文件服务协议规范可能把它的过程号 5 定义为“read”，过程号 12 定义为“write”。

就像远程程序协议在一些版本上可能有一些变化一样。实际的 RPC 消息协议也可能有所改变。因此，调用消息也有它自己的 RPC 版本号。对于在这里描述的 RPC 来说，这个版本号总是等于 2。

请求消息的响应消息具有足够的信息来区分下面的错误情况：

- (1) RPC 的远程实现应该使用协议的第二版，RPC 协议支持的最低和最高版本号将返回。
- (2) 远程程序在远程系统中不可用。
- (3) 远程程序不支持请求的版本号。远程程序支持的最低和最高版本号将返回。
- (4) 请求的过程号不存在。（这总是因为调用者一方的协议或者程序出错）
- (5) 传给远程过程的参数在服务器看来是无用的（这实际上是由客户端和服务器之间的协议没有达成一致造成的）。

7.2 认证

服务器对调用者的认证和调用者对服务器的认证的规则是 RPC 协议的一部分。调用消息有两个认证字段，证书和校验符响应消息有一个认证字段，即响应校验符。RPC 协议规范把这三个字段都定义成不透明的类型。

```
enum auth_flavor {
    AUTH_NULL      = 0,
    AUTH_UNIX      = 1,
    AUTH_SHORT      = 2,
    AUTH_DES        = 3
    /* 更多的定义 */
};

struct opaque_auth {
    auth_flavor flavor;
    opaque body<400>;
};
```

用简单的英语表示，任何一个"opaque_auth"结构就是由一个"auth_flavor"枚举类型后跟一些对 RPC 协议实现来说是不透明的字节组成的。

在认证字段中包含的数据的语义和解释是个别指定的，独立于认证协议规范。（第九节定义了不同的认证协议）

如果认证参数被拒绝，响应消息将包含说明为什么会被拒绝的信息。

7.3 程序号分配

程序号根据下表以十六进制 20000000（十进制 536870912）为一组来进行分发。

0 - 1ffffff	由 Sun 公司定义
20000000 - 3ffffff	由 Sun 公司定义
40000000 - 5ffffff	暂时的
60000000 - 7ffffff	保留
80000000 - 9ffffff	保留
a0000000 - bffffff	保留
c0000000 - dffffff	保留
e0000000 - fffffff	保留

第一组是属于升阳微系统公司（Sun 公司）管理的号码范围。在所有场所都应该是一致的。第二组号码范围对应用程序来说对特定场所是特有的。这个范围主要是用来调试新程序的。当在一个场所中开发的一个应用程序要引起公众的注意，这个应用程序就应该在第一组号码范围中分配一个号码。第三组号码范围对应用程序来讲是动态地产生程序号。最后的那些号码范围组是为将来使用保留的，目前还没有使用。

7.4 RPC 协议的其它使用

使用 RPC 协议的主要目的就是为了解调用远程的过程。也就是说，每一个调用消息都与一个响应消息匹配。但是，这个协议本身是一个消息传送协议，其它协议（非 RPC 协议）也可以通过这个协议来实现。Sun 当前正在为下面两种非 RPC 协议使用 RPC 消息协议，这两种协议是批处理（或者管道）和广播 RPC，下面将讨论，但是不详细说明。

7.4.1 批处理

批处理允许客户发送一个任意大的调用消息序列给服务器；它典型地使用可靠字节流协议（象 TCP/IP）来作为它的传输层。在批处理的时候，客户从来不等待服务器的响应，服务器也不发送对批处理请求的响应。一个批处理调用序列总是由合法的 RPC 终止，这是为了刷新管道（以肯定确认）。

7.4.2 广播 RPC

在基于广播 RPC 的协议中，客户向网络中发送一个广播数据包，然后等待响应。广播 RPC 使用不可靠的数据报协议（象 UDP/IP）作为它的传输层。支持广播协议的服务器只有在请求被成功执行后才进行响应。在出错的时候服务器将保持沉默。广播 RPC 使用端口映射器 RPC 服务来获得它的语义。（要获得更多的信息请参见附录 A）

8. RPC 消息协议

这一节用 XDR 数据定义语言来定义 RPC 消息协议。这些消息是用一种严谨的风格来定义的。

```
enum msg_type {
    CALL    = 0,
    REPLY   = 1
};

/*
 *调用消息的响应可以呈现出两种形式；
 *消息要么被接受要么被拒绝
 */

enum reply_stat {
    MSG_ACCEPTED = 0,
    MSG_DENIED   = 1
};

/*
 *如果一条调用消息被接受，下面是试图调用远程过程的状态。
 */

enum accept_stat {
    SUCCESS      = 0,    /* RPC 执行成功 */
    PROG_UNAVAIL = 1,    /* 远程没有输出程序 */
    PROG_MISMATCH = 2,   /* 远程不支持版本号 */
    PROC_UNAVAIL = 3,    /* 程序不支持过程 */
    GARBAGE_ARGS = 4     /* 过程不能解释参数 */
};

/*
 * 调用消息被拒绝的原因
 */

enum reject_stat {
    RPC_MISMATCH = 0, /* RPC 版本号不等于 2 */
    AUTH_ERROR   = 1  /* 远程不能认证调用者 */
};

/*
 * 认证失败的原因
 */

enum auth_stat {
    AUTH_BADCRED = 1, /* 错误的证书 (封装被打破) */
};
```

```
AUTH_REJECTEDCRED = 2, /* 客户必须开始新会话 */
AUTH_BADVERF      = 3, /* 错误的校验符 (封装被打破) */
AUTH_REJECTEDVERF = 4, /* 校验符过期或者重放 */
AUTH_TOOWEAK     = 5  /* 因为安全原因拒绝 */
};

/*
 * RPC 消息
 *所有消息以一个事务标识符 xid 开始, 接下来是有判别的两个分支的联合。
 *这个联合的判别式是 msg_type 等于两类消息的一种。
 *响应消息的 xid 总是匹配开始调用消息的 xid。
 *客户端使用这个字段把收到的响应消息与调用消息配备,
 *服务器端使用这个字段来检测重传。
 *服务器端不能把这个 id 看作为任何一种类型的序列号。
 */
struct rpc_msg {
    unsigned int xid;
    union switch (msg_type mtype) {
        case CALL:
            call_body cbody;
        case REPLY:
            reply_body rbody;
    } body;
};

/*
 * RPC 请求调用的实体:
 *在 RPC 协议规范的版本 2 中, rpcvers 字段必须等于 2。
 *字段 prog, vers,和 proc 确定了远程程序,以及程序的版本号
 *和在远程程序中被调用的过程。在后面是两个认证参数
 *cred (认证证书) 和 verf (认证校验符)。认证参数的下面是远程过程的参数,
 *这些参数是由具体的程序协议来指定的。
 */
struct call_body {
    unsigned int rpcvers;          /* 必须等于 2 */
    unsigned int prog;
    unsigned int vers;
    unsigned int proc;
    opaque_auth cred;
    opaque_auth verf;
    /* 过程的特定参数从这里开始 */
};

/*
```

```
* RPC 请求的响应实体;
* 调用消息要么被接受要么被拒绝
*/
union reply_body switch (reply_stat stat) {
case MSG_ACCEPTED:
    accepted_reply areply;
case MSG_DENIED:
    rejected_reply rreply;
} reply;

/*
* 当服务器接受 RPC 请求时的响应:
*即使请求被接受, 也可能会有错误。
* 第一个字段是服务器产生的认证校验符, 为了使调用者能够识别服务器。
*接下来是一个判别式是枚举类型 accept_stat 的联合。
*联合中 SUCCESS 的一个分支是特定的协议。
*联合中 The PROG_UNAVAIL, PROC_UNAVAIL 和 GARBAGE_ARGS
*的分支为空。PROG_MISMATCH 分支确定了服务器支持的远程
*程序的最低和最高版本号。
*/
struct accepted_reply {
    opaque_auth verf;
    union switch (accept_stat stat) {
    case SUCCESS:
        opaque results[0];
        /*
        *具体的过程结果从这里开始
        */
    case PROG_MISMATCH:
        struct {
            unsigned int low;
            unsigned int high;
        } mismatch_info;
    default:
        /*
        *空。这些情况包括 PROG_UNAVAIL,
        * PROC_UNAVAIL 和 GARBAGE_ARGS.
        */
        void;
    } reply_data;
};

/*
*当服务器拒绝 RPC 请求时的响应:
```

*请求可能由于两种原因被拒绝：一种是服务器没有运行 RPC 协议的
*兼容版本(RPC_MISMATCH),另一种是服务器不认证调用者(AUTH_ERROR)。
*在 RPC 版本不匹配的情况下，服务器返回所支持的最低和最高版本号。
*在不认证的情况下，返回失败状态。
*/

```
union rejected_reply switch (reject_stat stat) {
case RPC_MISMATCH:
    struct {
        unsigned int low;
        unsigned int high;
    } mismatch_info;
case AUTH_ERROR:
    auth_stat stat;
};
```

9. 认证协议

象前面所说的那样，认证参数是不透明的，但是对于 RPC 协议的其余部分是可调整的。这一节说明了在 Sun 中实现（或是由 Sun 支持的）的认证的“特征值”。在其它的场所中可以自由得创造新的认证类型。特征值的分配的规则和程序号的分配规则是一样的。

9.1 不认证

通常过程会在调用者不知道它是谁或者服务器不关心调用者是谁的情况下调用。在这种情况下 RPC 消息中的证书，校验符和响应校验符的“特征值”（opaque_auth 的联合的判别式）为"AUTH_NULL"。opaque_auth 的实体中的字节没有定义。建议这些 opaque 长度为 0。

9.2 UNIX 认证

远程过程的调用者可能希望象在 UNIX 系统中那样标识它自己。RPC 调用消息中的证书判别式的值是"AUTH_UNIX"。这些证书的 opaque 实体编码成下面这样的结构：

```
struct auth_unix {
    unsigned int stamp;
    string machinename<255>;
    unsigned int uid;
    unsigned int gid;
    unsigned int gids<10>;
};
```

“stamp”是一个调用者机器产生的二进制标识符（ID）。“machinename”是调用者机器的名

字（例如 “krypton”）。“uid”是调用者的有效用户标识符 ID。“gid”是调用者的有效组标识符 ID。“gids”是一个包含调用者作为成员的组的计数数组。伴随着证书的校验符应该是“AUTH_NULL”（上面定义的）

从服务器收到的响应消息中的响应校验符的判别式的值可能是“AUTH_NULL”，或者是“AUTH_SHORT”。在“AUTH_SHORT”的情况下，响应校验符的字符串字节编码成不透明的结构。这个新的不透明结构现在可以传递给服务器以代替原来的“AUTH_UNIX”特征值证书。服务器保留一个缓冲区用来把这个速记下来的不透明结构（通过一个“AUTH_SHORT”类型响应校验符传回）映射到调用者原来的证书上。调用者可以使用新的证书节省网络带宽和服务器上的cpu周期。

服务器可能在任一时间刷新这个速记下来的不透明结构。如果这种情况发生，远程过程调用消息将由于认证错误被拒绝。失败的原因将是“AUTH_REJECTEDCRED”。在这时，调用者可能希望试试原来的“AUTH_UNIX”证书类型。

9.3 DES 认证

UNIX 认证遇到了下面两个主要问题：

- (1) 命名太面向于 UNIX
- (2) 没有校验符，所以证书很容易伪造

DES 认证努力来解决这两个问题。

9.3.1 命名

处理第一个问题的方法是通过使用一个简单的字符串代替操作系统特定的整数来定位调用者。这个字符串就作为“netname”或者调用者的网络名。除了在鉴别调用者之外，不允许服务器在任何其它的方面解释调用者名字中的内容。因此，网络名对于在 Internet 中的每一个调用者来说都应该是唯一的。

由每一个操作系统上的 DES 认证实现来为它的用户产生网络名，以确保当用户造访远程服务器的时候，他的网络名是唯一的。操作系统也知道怎样区分本地系统中的用户。把这种机制扩展到网络上通常存在着一些小问题。例如，在 Sun 上的具有用户 ID 号为 515 的 UNIX 用户可能被分配给下面的网络名：“unix.515@sun.com”。这个网络名包含三项以使服务器确认它是唯一的。在 Internet 网上叫做“sun.com”的命名域只有一个。在这个域中用户 ID 是 515 的 UNIX 用户也仅有一个。但是，在其它的操作系统上可能有另一个用户，例如在 VMS 上，也在同样的命名域中，碰巧也有同样的用户 ID。为了确保两个用户被区分，我们加入操作系统名。所以一个用户是“unix.515@sun.com”，另一个是“vms.515@sun.com”。

第一个字段实际上是一种命名方法，而不是一个操作系统名。现在仅仅是碰巧在命名方法和操作系统之间存在一对一对应。如果全世界在命名标准上取得一致，第一个字段应该是

标准的名字，而不是一个操作系统名。

9.3.2 DES 认证校验

不像 UNIX 认证，DES 认证有一个校验符，这可以使服务器验证客户的证书（反之客户也可以验证服务器的证书）。校验符的内容主要是一个加密的时间戳。服务器可以解密这个时间戳。如果它接近真实的时间，那么客户一定已经正确的加密了它。客户端能加密它的唯一方法是知道这个 RPC 会话的“会话密钥”。如果客户端知道这个会话密钥，那么它就是真正的客户端。

会话密钥是由一个由客户端产生的 DES 密钥[5]，在第一次 RPC 调用时通知服务器。会话密钥在第一次事务中由一个公钥模式加密。使用在 DES 认证中的特别的公钥模式是 Diffie-Hellman [3],它具有 128 比特位的密钥长度。这种加密方法的细节以后讨论。

为了完成工作，客户端和服务端需要相同的时间概念。如果网络时间同步不能保证，那么客户端可以在开始会话前与服务端建立同步。也许要与 Internet 时间服务器协商(TIME [4])。

服务器判断一个客户端时间戳是否有效的方法有点复杂。对于除了第一次事务之外的其它事务，服务器只检查两件事。

- (1) 这个时间戳比来自同一个客户端的前一个时间戳大
- (2) 这个时间戳没有过期。

如果服务器时间比客户端的时间戳与客户端的窗口值（window）相加后的总和还要晚，这个时间戳过期。“window”是客户端在它的第一次事务中传递（已加密过）给服务器的一个数字。你可以把它看作是证书的生存期。

以上说明了除第一次事务的每一次事务。在第一次事务中，服务器仅检查时间戳没有过期。如果就是所作的全部工作的话，对于客户端来说，发送随机的数据代替时间戳将会非常容易，这会有很大的成功机会。作为一个附加的检查，客户端在第一次事务中发送一个叫“窗口校验符”的加密项，它必须等于窗口值减 1，否则服务器将拒绝这个证书。

客户端也必须检查从服务器返回的校验符，以确定服务器是合法的。服务器把它从客户端收到的加密时间戳减一秒后再送回给客户端。如果客户端得到数据与此不同，它将拒绝。

9.3.3 昵称和时钟同步

在第一次事务之后，服务器 DES 认证子系统在它返回给客户端的校验符中有一个“昵称”整数，客户端可以在以后的每次事务中使用这个昵称来代替它的网络名，加密的 DES 密钥和窗口。昵称更像一个在服务器上表的索引，在这张表中存储着每一个客户端的网络名，DES 密钥译文和窗口。

尽管客户端和服务器的时钟在开始时被同步，但是他们可能再次变得不同步。当这种情况发生后，客户端 RPC 子系统可能在应该需要再次同步的时候得到"RPC_AUTHERROR"。

客户端即使在与服务器同步的情况下仍然可能得到"RPC_AUTHERROR"错误。这个原因就是服务器的昵称表大小有限，它可以在它需要的时候随时刷新表中的条目。在这种情况下，客户端应该重新发送它的原来的证书，服务器将给它一个新的昵称。如果服务器崩溃，昵称表中的条目得到刷新，所有的客户端都将必须重新发送它的原来的证书。

9.3.4 DES 认证协议规范（用 XDR 语言描述）

```
/*
 *有两种证书:一种是客户端使用它的全称网络名，另一种是客户端使用
 *由服务器发给它的昵称（仅仅是一个无符号整数）。在第一次与服务器的事务中，
 *客户端必须使用它的全称名。服务器将返回给客户端它的昵称。
 *客户端可以在以后的与服务器的使用它的昵称。并不是一定要使用昵称，
 *但是使用昵称将是一个明智的选择。
 */
enum authdes_namekind {
    ADN_FULLNAME = 0,
    ADN_NICKNAME = 1
};

/*
 *一个由 DES 加密的 64 比特数据块。
 */
typedef opaque des_block[8];

/*
 * 网络用户名的最大长度
 */
const MAXNETNAMELEN = 255;

/*
 * 全称包括客户端的网络名，一个加密的会话密钥和窗口。窗口实际上就是
 * 证书的生存期。如果在校验符中的时间戳指示的时间加上窗口后过期，
 *那么服务器应该使请求过期，而且不承认它。为了确保请求不会重放，
 *服务器应该保持时间戳比前一个看到的时间戳大,除非这是第一个事务。
 *在第一个事务中，服务器只检查窗口校验符是一个小于窗口的值。
 */
struct authdes_fullname {
    string name<MAXNETNAMELEN>; /* 客户端名字 */
    des_block key; /* 公用密钥加密过的会话密钥*/
    unsigned int window; /* 加密过的窗口 */
};
```

```
};

/*
 * 一个证书要么是全称要么是昵称
 */
union authdes_cred switch (authdes_namekind adc_namekind) {
case ADN_FULLNAME:
    authdes_fullname adc_fullname;
case ADN_NICKNAME:
    unsigned int adc_nickname;
};

/*
 * 时间戳从 1970 年 1 月 1 日午夜开始给时间编码
 */
struct timestamp {
    unsigned int seconds;    /* 秒 */
    unsigned int useconds;  /* 微秒 */
};

/*
 * 校验符：客户端的种类
 * 窗口的校验符仅使用在第一次事务中，在与全称证书的关联中，
 * 这些项目在加密前打包在下面的结构中。
 *
 * struct {
 *     adv_timestamp;        -- one DES block
 *     adc_fullname.window; -- one half DES block
 *     adv_winverf;         -- one half DES block
 * }
 * 这个结构以一个输入的 0 向量使用 CBC 加密模式加密
 * 所有的其它时间戳加密是使用 ECB 加密模式加密
 */
struct authdes_verf_clnt {
    timestamp adv_timestamp; /* 加密的时间戳 */
    unsigned int adv_winverf; /* 加密的窗口校验符 */
};

/*
 * 校验符：服务器种类
 * 服务器把这个加密的时间戳减一秒后，返回给客户端。
 * 服务器也告诉客户端它的昵称（未加密），已备在将来的事务中使用。
 */
struct authdes_verf_svr {
```

```

timestamp adv_timeverf;    /* 加密的校验符 */
unsigned int adv_nickname; /* 客户端新的昵称*/
};

```

9.3.5 Diffie-Hellman 加密方法

在这种模式中，有两个常数"PROOT" 和"MODULUS"。Sun 为 DES 认证协议选择了特定的值：

```

const PROOT = 2;
const MODULUS = "b520985fb31fcdf75036701e37d8b857"; /* 十六进制 */

```

这种模式工作的过程最好用一个例子来说明。假设有两个人“A”和“B”想互相发送加密的消息。所以，A 和 B 都随机产生一个“秘密”密钥，这个密钥他们并不想让其它人知道。把这两个密钥表示为 SK(A) 和 SK(B)。他们也在一个公开的姓名地址录中发布他们的“公用”密钥。公用密钥由以下方式计算出。

$$PK(A) = (PROOT ** SK(A)) \text{ mod } MODULUS$$

$$PK(B) = (PROOT ** SK(B)) \text{ mod } MODULUS$$

“**”符号在这里表示求幂。现在 A 和 B 都能得到在他们之间的“公共”密钥，而不需要暴露他们的秘密密钥。公共密钥在这里描述为 CK(A, B),

A 电脑：

$$CK(A, B) = (PK(B) ** SK(A)) \text{ mod } MODULUS$$

B 电脑：

$$CK(A, B) = (PK(A) ** SK(B)) \text{ mod } MODULUS$$

这两个值是相等的：

$$(PK(B) ** SK(A)) \text{ mod } MODULUS = (PK(A) ** SK(B)) \text{ mod } MODULUS$$

我们去掉"mod MODULUS"的取模算法部分来进行简化。

$$PK(B) ** SK(A) = PK(A) ** SK(B)$$

然后，用以前 B 计算得到的结果来替换 PK(B)，对 PK(A)也进行这样的步骤

$$((PROOT ** SK(B)) ** SK(A) = (PROOT ** SK(A)) ** SK(B))$$

这将导致：

$$\text{PROOT}^{**}(\text{SK}(\text{A}) * \text{SK}(\text{B})) = \text{PROOT}^{**}(\text{SK}(\text{A}) * \text{SK}(\text{B}))$$

不使用公共密钥 $\text{CK}(\text{A}, \text{B})$ 来加密在协议中使用的时戳。它只用于加密会话密钥，而使用这个会话密钥来加密时戳。这样做的原因是因为应该尽可能少的使用公共密钥，以防公共密钥被破译。会话密钥即使被破译，造成的损失也较小，因为会话的时间总是相对较短。

会话密钥是使用 56 比特 DES 密钥加密的。而这个公共密钥是 128 比特位。为了减少比特的数量，象下面这样从公共密钥中选择 56 比特。从公共密钥中选择最中间的 8 个字节，然后把奇偶校验加在每一个字节的最低比特位。这样就产生了一个带有 8 比特奇偶校验位的 56 比特密钥。

10. 记录标记的标准

当 RPC 消息在一个字节流协议（象 TCP/IP）上层传送的时候，有必要在一个消息和另一个消息之间划定界线，这样是为了检测出用户协议的错误，并可能对错误进行恢复。这就叫做记录标记（RM）。Sun 使用 RM/TCP/IP 来在 TCP 流上传送 RPC 消息。一个 RPC 消息适配一个 RM 记录。

一个记录是由一个或者多个记录片断组成。一个记录片断是 4 字节的头，后跟 0 至 $(2^{**}31)-1$ 字节的片断数据。片断头的字节编码成一个无符号的二进制数；象是 XDR 中的整数一样，字节的顺序是从高到底的。这个数字编码成两个部分的值：一个部分是布尔值，指示这个片断是否是记录的最后一个记录片断（值 1 表明此片断是最后一个片断），另一部分是一个 31 比特的无符号二进制值，它是这个片断数据用字节计数时的长度。布尔值是这个片断头的最高位比特；长度是低位的 31 比特。（注意这个记录规范并不是 XDR 的标准形式！）

11. RPC 语言

就象在一个正式的语言中需要定义 XDR 数据类型一样。也需要在正式的语言中定义操作 XDR 数据类型的过程。为了这个目的，我们使用 RPC 语言。它是 XDR 语言的扩展。使用下面的例子来描述这种语言的精髓。

用 RPC 语言描述的服务的例子

这里有一个简单 ping 程序的例子说明。

```
/*
 *简单的 ping 程序
 */
program PING_PROG {
    /*
     * 最近和最好的版本
```

```

    */
    version PING_VERS_PINGBACK {
        void          PINGPROC_NULL(void) = 0;

        /*
         * Ping 这个调用者，返回往返时间（用微秒来表示）。如果操作超时，
         * 返回-1。
         */
        int          PINGPROC_PINGBACK(void) = 1;
    } = 2;

    /*
     * 原来的版本
     */
    version PING_VERS_ORIG {
        void          PINGPROC_NULL(void) = 0;
    } = 1;
} = 1;

const PING_VERS = 2;    /* 最近的版本*/

```

在第一个版本中，PING_VERS_PINGBACK 有两个过程 PINGPROC_NULL 和 PINGPROC_PINGBACK。PINGPROC_NULL 不需要参数，也不返回结果，但是它对计算从客户端到服务器再回到客户端的往返时间很有用。在会话中，任何 RPC 协议的过程 0 都应该有同样的语义，不需要任何种类的认证。客户端用第二个过程来使服务器对客户端进行一个反向的 ping 操作，并且它返回使用这个操作的时间值（用微秒表示）。下一个版本 PING_VERS_ORIG 是这个协议的原来的版本，它不包含 PINGPROC_PINGBACK 过程。这对兼容旧的客户端程序很有用，随着程序的升级，它可能从整个协议中删去。

11.1 RPC 语言规范

RPC 语言除了加入了"program-def"的定义，与 XDR 语言一样。

```

program-def:
    "program" identifier "{"
        version-def
        version-def *
    }" "=" constant ";"

version-def:
    "version" identifier "{"
        procedure-def
        procedure-def *

```

```
"}" "=" constant ";"
```

```
procedure-def:  
type-specifier identifier "(" type-specifier ")"  
    "=" constant ";"
```

11.2 语法的注意事项

- (1) 下面的关键字不能用作标识符: "program" and "version";
- (2) 版本名不能在一个程序定义的范围内出现超过一次。
同样, 版本号在这个程序定义的范围里也不能出现超过一次。
- (3) 过程名不能在一个版本定义的范围内出现超过一次。
同样, 过程号在这个版本定义的范围里也不能出现超过一次。
- (4) 程序的标识符有象常数标识符和类型标识符同样的名字空间。
- (5) 只有无符号的常数可以分配给程序, 版本和过程。

附录 A: 端口映射器程序协议

端口映射器程序把 RPC 程序和版本号映射到特定的传输端口号上。这个程序可以对远程的程序进行动态绑定。

这种方法是符合需要的, 因为保留的端口号的范围是有限的, 而潜在的远程程序是很多的。在一个保留端口号上运行端口映射器, 只要查询这个端口映射器就可以确定其它远程程序的端口号。

端口映射器也在广播 RPC 中使用。一个给定的 RPC 程序在不同的机器上经常绑定到不同的端口号上, 所以没有方法直接广播到所有的这些程序。而端口映射器有固定的端口号。要向给定的程序发送广播, 客户端实际上把消息发送到广播地址上的端口映射器。获得广播的每一个端口映射器调用由客户端指定的本地服务。当端口映射器收到本地服务的响应, 它再把这个响应发送给客户端。

A.1 端口映射器协议规范 (用 RPC 语言)

```
const PMAP_PORT = 111;      /* 端口映射器的端口号*/
```

```
/*
 * (程序, 版本, 协议) 到端口的映射
 */
struct mapping {
    unsigned int prog;
    unsigned int vers;
    unsigned int prot;
    unsigned int port;
};

/*
 * “prot”字段支持的值
 */
const IPPROTO_TCP = 6;      /* TCP/IP 的协议号 */
const IPPROTO_UDP = 17;    /* UDP/IP 的协议号*/

/*
 * 一个映射列表
 */
struct *pmaplist {
    mapping map;
    pmaplist next;
};

/*
 * 调用的参数
 */
struct call_args {
    unsigned int prog;
    unsigned int vers;
    unsigned int proc;
    opaque args<>;
};

/*
 * 调用的结果
 */
struct call_result {
    unsigned int port;
    opaque res<>;
};

/*
 * 端口映射器的过程
 */
program PMAP_PROG {
```

```
version PMAP_VERS {
    void          PMAPPROC_NULL(void)          = 0;
    bool          PMAPPROC_SET(mapping)        = 1;

    bool          PMAPPROC_UNSET(mapping)      = 2;

    unsigned int  PMAPPROC_GETPORT(mapping)    = 3;

    pmaplist      PMAPPROC_DUMP(void)         = 4;

    call_result   PMAPPROC_CALLIT(call_args)  = 5;
} = 2;
} = 100000;
```

A.2 端口映射器操作

端口映射器程序当前支持两种协议（UDP/IP 和 TCP/IP）。端口映射器在这两种协议中的任一种中都分配在端口 111 上(SUNRPC [8])。下面是每一个端口映射器过程的描述：

PMAPPROC_NULL:

这个过程不工作。按照习惯，任何协议的 0 号过程不接收参数，也不返回结果。

PMAPPROC_SET:

当一个程序在一个机器上可用时，它要向同一台机器上的端口映射器注册它自己。

这个程序传递它的程序号 “prog”，版本号 “vers”，传输协议号 “prot”和端口号 “port”。在 “port”这个端口号上，程序等待服务的请求。这个过程返回一个布尔值，如果过程成功建立了映射，这个值为“TRUE”。否则，这个值为“FALSE”。

如果已经存在“(prog, vers, prot)”元组，这个过程拒绝建立映射。

PMAPPROC_UNSET:

当程序变为不可用的时候，它将在同一台机器上的端口映射器中注销它自己。它的参数和结果和“PMAPPROC_SET”有同样的含义。参数中的协议和端口号字段将被忽略。

PMAPPROC_GETPORT:

当被给出了程序号 “prog”，版本号 “vers”和传输协议号 “prot”，这个过程将返回此程序等待调用请求的端口号。返回一个 0 的端口值意味着这个程序没有注册。参数中的 “port”字段将被忽略。

PMAPPROC_DUMP:

这个过程列举了端口映射器数据库中的所有条目。

此过程不需要参数，它返回一个程序，版本，协议，和端口值的列表。

PMAPPROC_CALLIT:

这个过程允许调用者调用在同一台机器上的另一个远程过程，而不需要知道这个远程过程的端口号。通过众所周知的端口映射器的端口号，它支持二进制远程程序的广播。参数"prog", "vers", "proc", "args"分别是程序号，版本号，过程号和远程过程的参数。

注意：

- (1) 如果过程成功执行，过程仅发送一个响应。否则保持沉默（不响应）。
- (2) 端口映射器只使用 UDP/IP 协议与远程程序通信。

这个过程返回远程程序的端口号和远程过程执行的结果。

参考书目

- [1] Birrel, A. D., and Nelson, B. J., "Implementing Remote Procedure Calls", XEROX CSL-83-7, October 1983.
- [2] Cheriton, D., "VMTP: Versatile Message Transaction Protocol", Version 0.7, RFC-1045, Stanford University, February 1988.
- [3] Diffie & Hellman, "Net Directions in Cryptography", IEEE Transactions on Information Theory IT-22, November 1976.
- [4] Postel, J., and Harrenstien, K., "Time Protocol", RFC-868, Network Information Center, SRI, May 1983.
- [5] National Bureau of Standards, "Data Encryption Standard", Federal Information Processing Standards Publication 46, January 1977.
- [6] Postel, J., "Transmission Control Protocol - DARPA Internet Program Protocol Specification", RFC-793; Network Information Center, SRI, September 1981.
- [7] Postel, J., "User Datagram Protocol", RFC-768, Network Information Center, SRI, August 1980.

- [8] Reynolds, J. and Postel, J.; "Assigned Numbers", RFC-1010, Network Information Center, SRI, May 1987.

- [9] Sun Microsystems; "XDR: External Data Representation Standard", RFC-1014; Sun Microsystems, June 1987.