

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](mailto:ouyang@china-pub.com)

译者: spacelu (spacelu [wuchun\\_lu@163.net](mailto:wuchun_lu@163.net))

译文发布时间: 2001-8-14

版权: 本中文翻译文档版权归中国互动出版网所有。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版权信息。

Network Working Group

Sun Microsystems, Inc.

Request For Comments: 1057

June 1988

Obsoletes: RFC 1050

## RFC:远程过程调用协议说明第二版

(RPC: Remote Procedure Call Protocol Specification Version 2)

### 备忘录状态

该备忘录表书了 Sun 为系统和其它系统用应用的标准, 和我们期望因特网应用的一个考虑. 现在该备忘录还不是一个因特网标准. 该备忘录的发布不受限制.

1. 简介 .....	2
2. 术语学 .....	2
3. RPC 模式 .....	2
4. 传送和语义 .....	3
5. 绑定和集合独立 .....	3
6. 鉴定 .....	4
7. RPC 协议要求 .....	4
7.1 RPC 程序和过程 .....	4
7.2 鉴定 .....	5
7.3 程序号委派 .....	5
7.4 RPC 协议的其它使用 .....	6
8. RPC 信息协议 .....	6
9. 鉴定协议 .....	9
9.1 空鉴定 .....	9
9.2 Unix 鉴定 .....	9
9.3 DES 鉴定 .....	10
10. 记录记号标准 .....	13
11. RPC 语言 .....	13
11.1 RPC 语言描述的例子: .....	14

11.2 RPC 语言说明 .....	14
11.3 语法注意事项: .....	15
附录: 程序协议端口映射.....	15
A.1 端口映射协议说明 (用 RPC 语言) .....	15
A.2 端口映射操作: .....	17
参考资料: .....	17

## 1.简介

文档详细说明了用在 Sun 远程过程调用 (RPC) 包的信息协议第二版. 这个信息协议用外部数据表示 (XDR) 语言来说明. 该文档假设读者对 XDR 熟悉. 它不尽力证明远程过程调用系统或描述它们的应用. 由 Birrell 和 Nelson [1] 写的文档推荐成为远程过程调用概念的最好背景知识.

## 2.术语学

文档讨论了客户, 调用, 服务器, 应答, 服务, 程序, 过程和版本. 每一个远程调用有两方: 积极的客户方发送调用请求到服务器, 服务器发回应答信息. 一个网络服务是一个或多个远程程序的集合. 一个远程程序执行一个或多个远程过程; 过程, 参数和结果都在特殊程序协议说明 (看附录 A 的例子) 里记录. 为了和改变的协议兼容, 一个服务器可能支持多个版本的远程程序.

例如, 一个网络文件服务由两部分组成. 一个程序可能处理高层应用 (如文件系统访问控制和锁控. 另外有些处理低层如文件输入输出和象读和写过程. 网络文件服务的客户将调用代表该客户与对应服务的两类程序相关的过程.

术语客户和服务器只是适用于特殊的传输; 一个特定硬件实体 (主机) 或软件实体 (过程或程序) 能够在不同时间执行两种角色. 例如, 提供远程执行服务的程序可能也是一个网络文件服务的客户端. 另外, 它可能把软件根据服务器和客户端功能成分开分的库或程序.

## 3. RPC 模式

Sun RPC 协议基于远程过程调用模式, 它类似于本地过程调用模型. 在本地调用方式中, 调用者把参数放在公众指定地点 (如注册窗口), 然后发送控制到过程, 最后重新获得控制. 接着, 从指定地点取出过程结果, 调用者继续执行.

远程过程调用相类似. 控制线程在两个过程中逻辑转换: 调用过程和服务过程. 调用过程首先发送一个调用信息到服务过程然后等待应答信息. 调用信息包括过程参数, 应答信息包括过程结果. 一旦接收到应答信息, 就取得过程结果, 然后调用执行继续进行.

在服务器端, 过程保持睡眠状态到调用信息的到达. 当一个调用信息到达, 服务器获得过程参数, 计算结果, 发送应答信息, 然后等待下一个调用信息.

在这种模型中, 任何时间里两个过程只有一个激活. 但是, 该模型只是作为一个例子. Sun RPC 协议对并行模型执行没有限制, 但是其它的有可能不一样. 例如, 一个应用程序可能选择 RPC 调用为异步的, 因此客户端只有等到服务器端的应答才做有效工作. 另外一个可能是使服

务器端生成一个新的任务来处理进来的调用,因此最初的服务器可以处理其他请求.远程调用和本地过程调用有几个重要区别:

1. 错误处理:在远程过程调用中,网络或远程服务器的失败必须处理.
2. 全局变量和副作用:因为服务器没法访问客户地址空间,隐藏的参数不能用全局变量传递或返回副作用.
3. 表现:远程过程操作比本地过程调用慢一到几个数量级.
4. 鉴定:因为远程过程可以在不安全的网络中传输,必须采用鉴定.

结论是即使有工具自动为给定服务产生客户或服务库,仍然必须仔细设计协议.

## 4. 传送和语义

RPC 协议能够执行在几种不同传输协议上. RPC 协议除了信息的规定和解释外,不关心信息是如何从一个过程到另外一个过程,另外,应用想通过文档中没有指定的接口来获得传输层的信息(可能是控制层).例如,传输协议可能对 RPC 信息的尺寸大小进行限制,或可能是基于流的无大小限制的如 TCP. 客户或服务库通过在附录 A 中的机制,必须在传输协议达成一致.

RPC 不会执行任何可靠性和应用应该注意在 RPC 下层的传输协议类型是很重要的. 如知道运行在可靠传输协议如 TCP 上面,大部分工作 TCP 已经替做了.

另外,如果它运行在不可靠传输如 UDP[7]上,它必须执行自己的时间检测,重传,和复制检测,因为 RPC 层没有提供这些服务.

因为传输独立,所以 RPC 协议没有捆绑特殊的语义到远程过程或它们的执行要求上. 可以从下层传输协议中推得语义(但是得明确指定). 例如,考虑 RPC 运行在不可靠传输如 UDP 上. 如果一个应用再时间终止后重传 RPC 调用信息而没有收到应答,那么它不能从过程执行的时间数量推出任何信息. 如果它没有收到应答,它能够推出这个过程至少执行了一次.

服务器尽可能记住前面同意客户端请求而不必重新批准,为了保证首次执行语义. 服务器可以利用通过传输装载每一个 RPC 信息 ID 来完成这项任务. 这个传输的主要应用是通过客户 RPC 层使应答和调用相符. 但是,当重传调用时,一个客户应用可能选择重用原来的传输 ID. 为了获得一次执行语义,在执行了一个调用后,服务器选择记住这个 ID 而不执行有相同 ID 的调用. 除了检测是否相等外,服务器不允许用任何其它方式检查这个 ID.

另外,如果用可靠传输如 TCP,应用可以从应答信息推算出每个过程恰好执行一次,但是如果它没有收到应答信息,则不能假设远程过程没有执行. 注意即使使用一个基于连接的协议如 TCP,应用仍然需要超时和处理服务器崩溃的重新连接操作.

对于传输除了数据报或面向连接协议还有其它很多可能. 例如,请求-应答协议如 VMTP[2]可能是 RPC 的自然传输. 现在 Sun RPC 数据报用 TCP 和 UDP 两种传输协议,还有现在还在实验中的其它协议如 ISP IP4 和 IP0.

## 5. 绑定和集合独立

绑定一个特定客户到特定服务器和传输参数动作不是 RPC 协议说明的一部分. 这个重要的和必要功能是为更高层软件预留.(软件用 RPC 自身;看附录 A)

执行者把 RPC 协议想成网络的跳跃子程序指令(“JSR”);装货人(绑定者)使 JSR 有用,绑定软件使 RFC 游泳,用 RPC 来实现这个任务.

## 6. 鉴定

在每个调用和应答信息中，RPC 协议为客户端提供必须的向服务端验证域，和反之亦然。安全和访问控制机制能够在该信息鉴定上建立。支持多个不同的鉴定协议。在 RPC 报头上的鉴定域表明用哪一个协议。关于特殊鉴定协议的更多信息看第 9 部分：“鉴定协议”。

## 7. RPC 协议要求

RPC 协议必须提供下面条件：

- (1) 调用过程的唯一描述
- (2) 为请求信息提供一致应答信息
- (3) 为服务提供鉴定调用者和反之亦然。

除了这些要求，因为滚动错误，执行错误，用户错误和网络管理等，所以检测这些特性也应该支持。

- (1). RPC 协议不匹配.
- (2). 远程过程协议版本不一致.
- (3). 协议错误(如过程参数的错误配置).
- (4). 远程鉴定失败原因.
- (5). 其它所要过程没有调用的任何原因.

### 7.1 RPC 程序和过程

RPC 调用信息有 3 个无符号正数域——远程程序号, 远程程序版本号和远程过程号——它们唯一的指明了调用的过程. 程序数量由某个中央认证机构管理(象 SUN). 一旦执行者有一个程序号, 它们就可以执行远程程序; 第一个执行程序一般具有版本号 1. 因为大部分新协议的发展, 调用协议的版本号指明了调用者正在使用哪个版本的协议. 版本号使相同服务处理分辨新旧协议成为可能.

过程号标志调用过程. 这些数字在特定程序的协议规范中记录. 例如, 文件服务协议说明可能表明它的过程号 5 表示“读”和过程号 12 表示写.

正如远程程序协议可能改变好几个版本, 实际的 RPC 信息协议也可能改变. 因此, 调用信息也有它自己的 RPC 版本号, 对于在这描述的 RPC 的值总是为 2.

请求的应答信息有足够信息来分辨下面的错误情况:

- (1)RPC 的远程执行不分辨协议版本.
  - 2 返回支持 RPC 的最低和最高版本号.
  - (2)远程程序在远程系统中无效.
  - (3)远程程序不支持请求版本号.
- 支持最程序的最低和最高版本号返回.

(4) 请求过程号不存在. (这个一般是客户端协议或程序错误).

(5) 远程过程参数对服务器端来说是不认参数. (再有, 这个经常由于客户端和服务器端协议的不一致性引起).

## 7.2 鉴定

提供调用者到服务器的鉴定和反之亦然, 这是 RPC 协议的一部分. 调用信息有两个鉴定域, 信任域和校验域. 应答信息有一个鉴定域, 应答校验域. RPC 协议规范按下面定义所有 3 个不透明类型 (用外部表示语言 (XDR) [9])

```
enum auth_flavor {
    AUTH_NULL      = 0,
    AUTH_UNIX      = 1,
    AUTH_SHORT     = 2,
    AUTH_DES       = 3
    /* 和其它类型定义 */
};

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

也就是说, 任何“opaque\_auth”结构是一个“auth\_flavor”枚举类型加上 RPC 协议执行的不透明类型。

鉴定域里的数据解释和语义是个人设定, 独立于鉴定协议规范。(第 9 部分定义了不同鉴定协议)。如果鉴定参数被拒绝, 应答信息包含拒绝原因信息。

## 7.3 程序号委派

程序号根据下列表给成 16 进制 20000000 (十进制 536870912):

0 - 1fffffff	Sun 定义
20000000 - 3fffffff	用户定义
40000000 - 5fffffff	暂时
60000000 - 7fffffff	预留
80000000 - 9fffffff	预留
a0000000 - bfffffff	预留
c0000000 - dfffffff	预留
e0000000 - ffffffff	预留

第一组是由 sun 微系统管理的数字范围, 所有站点应该一样. 第二组是对特殊站点的特定应用. 当一个站点开发大众感兴趣的应用, 那么该应用应该分配一个在第一个区域的号码

值。第三组是动态分配给应用的程序号。最后几组为将来使用预留，应该还没有用上。

## 7.4 RPC 协议的其它使用

该协议的扩展使用是为调用远程过程。通常，每个调用信息和一个应答信息匹配。但是，协议自己是其它协议（非过程调用）能够执行的信息传输协议。sun 当前用的，或可能滥用的，批处理的（或流水线的）RPC 信息协议和广播远程过程调用。

### 7.4.1 批处理

当客户想发送任意数量的调用信息给服务器，可以用批处理方式。典型的批处理用可靠类型流协议（象 TCP）来传输。在批处理中，客户端从来不等待服务器的应答，服务器也不给批调用发送应答。为了疏通通路和让正常调用获得正常确认，一系列的批处理调用通常被合法远程过程调用操作终止。

### 7.4.2 远程过程调用广播

在广播协议中，客户发送广播调用到网络中然后等待无数应答。这种方法要求基于数据报传输方式（如 UDP）作为它的传输协议。当调用成功到达时，支持广播协议的服务器方给以应答，错误时保持它的状态。广播调用用 RPC 服务端口来获得它们的语义。更多信息看附录 A。

## 8. RPC 信息协议

这个部分定义了用 XDR 数据描述语言的 RPC 信息协议。

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

调用信息应答采用两种方式：信息或者被接受或者被拒绝。

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

假设调用信息被接受，下面就是调用远程过程的状态。

```
enum accept_stat {
    SUCCESS = 0, /* RPC 成功执行 */
    PROG_UNAVAIL = 1, /* 远程没有输出过程 */
};
```

```

    PROG_MISMATCH = 2, /* 不支持远程版本 # */
    PROC_UNAVAIL   = 3, /* 程序不支持远远程过程 */
    GARBAGE_ARGS  = 4  /* 过程不能解参数 */
};

```

调用信息被拒绝的原因:

```

enum reject_stat {
    RPC_MISMATCH = 0, /* RPC 版本!=2 */
    AUTH_ERROR   = 1  /* Remote 不能鉴定调用者 */
};

```

为什么鉴定失败:

```

enum auth_stat {
    AUTH_BADCRED      = 1, /* 坏信任书 (坏的签名) */
    AUTH_REJECTEDCRED = 2, /* 客户必须重新调用 */
    AUTH_BADVERF      = 3, /* 错误校验 (签名破坏) */
    AUTH_REJECTEDVERF = 4, /* 验证口令期满或破坏 */
    AUTH_TOOWEAK     = 5  /* 安全原因拒绝 */
};

```

所有 RPC 信息以事物标志-XID 开始,接着是两个区别域.联合的判别式是 msg\_type 类型,在信息的两种类型中进行交换.应答信息的 xid 总是和初始化调用信息相符. NB: xid 域只是用作客户匹配调用信息的应答或为服务器检测重传;服务器方不能把这个 ID 看作任何类型的系列号.

```

struct rpc_msg {
    unsigned int xid;
    union switch (msg_type mtype) {
        case CALL:
            call_body cbody;
        case REPLY:
            reply_body rbody;
    } body;
};

```

RPC 调用内容:

在第二版的 RPC 协议说明中, rpcvers 必须等于 2. prog, vers 和 proc 域指定了远程程序,版本号,和远程程序调用的过程.这些域后是两个鉴定参数: cred(鉴定信任书)和 verf(鉴定校验),然后是远程过程参数,在特定的程序协议中规定.

```

struct call_body {
    unsigned int rpcvers; /* 必须等于 2 (2) */
    unsigned int prog;
    unsigned int vers;
    unsigned int proc;
    opaque_auth cred;
    opaque_auth verf;
    /* 过程指定参数从这开始 */
};

```

```
};
```

RPC 调用应答内容:

```
union reply_body switch (reply_stat stat) {
case MSG_ACCEPTED:
    accepted_reply areply;
case MSG_DENIED:
    rejected_reply rreply;
} reply;
```

服务器接受的 RPC 调用应答:

即使调用被接受,也有可能存在错误.第一个域是服务器产生的用来使它对客户端有效的鉴定校验域.紧接着是成员是枚举类型 `accept_stat` 的联合.该联合的 `SUCCESS` 项是协议规定的.`PROG_UNAVAIL`, `PROC_UNAVAIL` 和 `GARBAGE_ARGS` 为空.`PROG_MISMATCH` 项指定服务器支持的远程过程调用最低和最高版本号.

```
struct accepted_reply {
    opaque_auth verf;
    union switch (accept_stat stat) {
case SUCCESS:
    opaque results[0];
    /*
     * 指定过程结果从这开始
     */
case PROG_MISMATCH:
    struct {
        unsigned int low;
        unsigned int high;
    } mismatch_info;
default:
    /*
     * Void. Cases include PROG_UNAVAIL, PROC_UNAVAIL,
     * and GARBAGE_ARGS.
     */
    void;
    } reply_data;
};
```

被服务器端拒绝的 RPC 调用应答:

调用被拒绝的原因有两个:或是服务器没有运行 RPC 协议 (`RPC_MISMATCH`) 兼容版本,或是服务器拒绝调用 (`AUTH_ERROR`) 鉴定.当 RPC 版本不符时,服务器返回 RPC 支持的最低和最高版本号.当拒绝鉴定时,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. 鉴定协议

如前面所述, 鉴定参数是不透明的, 但是对其它 RPC 协议开放. 这个部分定义在 Sun 运行程序的一些内容. 其它地方免费开发新的鉴定类型, 内容号的委派和程序号的委派规则相同.

### 9.1 空鉴定

当客户端不知道自己的 ID 或服务器不关心客户端是谁, 必须进行调用. 在这种情况下, RPC 信息的信任, 校验和应答校验值 (opaque\_auth 联合的判别式) 似乎 "AUTH\_NULL".

opaque\_auth

实体字节数没有定义. 建议把它的长度设置为 0.

### 9.2 Unix 鉴定

当客户在 UNIX 系统中识别时, 客户想在自身识别. RPC 调用信息信任判别式值是 "AUTH\_UNIX". 信任不透明体内容为下:

```

struct auth_unix {
    unsigned int stamp;
    string machinename<255>;
    unsigned int uid;
    unsigned int gid;
    unsigned int gids<16>;
};

```

"stamp" 域是调用机器产生的任意 ID. "machinename" 是调用机器名 (象 "krypton"). "uid" 是调用者有效的用户 ID. "gid" 是调用有效的组 ID. "gids" 是调用者所在组的记数数组. 检验和信任应该为 "AUTH\_NULL" (上面定义). 注意这些信任域在机器名, uid, gid 等特定域里是唯一的. 域内名字的讨论不在这个文档范围.

从服务器收到的回答校验判别式的值应该是 "AUTH\_NULL" 或 "AUTH\_SHORT". 当值为 "AUTH\_NULL" 时, 回答校验字符串编码成不透明结构. 现在这种新的不透明结构取代 "AUTH\_UNIX" 传送给服务器. 服务器保持一个缓冲, 对应短期不透明结构 (到调用者的初始信任书. 调用者通过新的信任书能够保存网络带宽和服务器 cpu 周期.

服务器在任何时候都可能刷新短期不透明结构。如果如此发生，远程过程调用将由于认证错误被拒绝。失败的原因为：“AUTH\_REJECTEDCRED”。在此看来，客户想利用初始“AUTH\_UNIX”信任书。

## 9.3 DES 鉴定

UNIX 鉴定主要有下面三个主要问题：

- (1) 名字太 UNIX 导向
- (2) 没有通用的地址，UID 和 GID 空间。
- (3) 有校验，因此认证书容易被伪造。

DES 鉴定将解决这些问题。

### 9.3.1 命名

第一个问题就是用一个简短的字符串来表示客户端而不用操作系统指定的整数。这个字符串称为“网络名字”或客户端网络名字。除了指定客户端，服务器端不允许用任何方式翻译客户端名字内容。这样，netnames 在因特网上对任何客户应该是唯一的。

需要操作系统执行 DES 认证来为用户产生保证调用远程服务器时唯一的 netnames。OS 已经知道如何辨别它们系统的用户。扩展这个机制到网络是简单可行的。例如，一个 sun unix 用户有一个用户 ID 为 515 将被分派网络名为：[unix.515@sun.com](mailto:unix.515@sun.com)。这个网络名包含三个部分来保证其唯一。分析其，在因特网中有且仅有一个名字域为：“sun.com”。在该域里头，有且仅有一个 unix 用户有 ID515。但是，要考虑的是，在该域里有可能使用另外一种操作系统的用户，如 VMS 有相同的域名。所以为了保证这两个拥护能够由操作系统区别开来。一个用户为[unix.515@sun.com](mailto:unix.515@sun.com)而另外一个为“vms.515@sun.com”。

第一个域实际上是命名方式而不是操作系统名字。但是碰巧的是，命名方式和操作系统之间存在一一对应关系。如果这个标准为世界所同意，第一个域是名字标准而不是操作系统名字。

### 9.3.2 DES 鉴定校验

不祥 UNIX 鉴定，DES 鉴定有校验功能，这样服务器能够验证客户的认证书（反之也是）。校验的内容主要是加密的时间戳。服务器解密该时间戳，如果这个时间值和实际时间靠近，那么客户肯定已经正确加密。客户能够正确加密的唯一方式就是知道 RPC 任务的交谈密钥。如果客户知道交谈密钥，它一定是实际客户。

交谈密钥是客户用 DES 加密产生的并在第一次 RPC 调用时传给服务器。交换密钥在第一次事物中用公共密钥来加密。用 DES 鉴定的特殊的公共密钥是有 192 位的 Diffie-Hellman [3]。加密方式的详细内容在下面进行描述：

为了保证所有这些事物有效，客户端和服务端应该具有相同的时间值，可以通过网络时间协议。如果网络时间同步不能得到保证，那么客户端可以在开始传送前用简单时间要求协议来确定服务器的时间。

服务器决定客户端时间戳是否有效的方式是有些复杂。对任何事物除了第一次，服务器需要检查两件事：

1. 从相同客户端发来的时间戳应该比前一次的大
2. 时间戳没有期满。

如果服务器的时间比客户端时间戳加上客户的窗口还晚，那么时间戳失效。窗口就是第一次任务中客户传给服务器的数量。可以认为是信任书的生存时间。

除了首次外，这里解释每样事情。在首次任务中，服务器只有检查没有过期的时间戳。如果所有这些都可以成功，那么对客户端发送任意数据代替时间戳更有可能成功。作为附加检查，客户端在第一次传送过程中发送加密部分，它必须等于窗口大小减一，否则服务器将拒绝该信任书。

客户也必须检查从服务器端返回的校验来保证其合法性。服务器返回它从客户端收到的时间戳减去一秒。如果客户端收到和这不同的数据，它将拒绝之。

### 9.3.3 别名和时钟同步

第一次传送之后，服务器 DES 认证子系统返回给客户端一个整数别名，这个整数别名是在后来事物中代替网络名，加密 DES 密钥和窗口用的。别名是想服务器表中的一个索引，该索引查找服务器表中所对应的网络名，解密 DES 密钥和窗口。

虽然开始时时钟是同步的，但是随后客户端和服务器端可能又失去同步。当不同步发生，客户端 RPC 子系统必须获得“RPC\_AUTHERROR”来重新获得同步。

即使客户保持和服务器同步，它也要获得“RPC\_AUTHERROR”。其原因是服务器的别名表大小有限，无论随时需要它都刷新输入。在这种情况下，客户端重新发送初始信任书，然后服务器重新发配一个别名。如果服务器崩溃，那么全部别名表都刷新，这样所有客户端都得重新发送它们色初始信任书。

## 9.3.4 DES 信任协议说明书

两种信任书为：一种是客户端用它的全网络名，另外一种是用服务器发布给它的别名（无符号整数）。在第一次传送中，客户端必须发送它的全名到服务器，然后服务器返回给客户端别名。客户端将可以用该别名在后续的事物中和服务器传送。没有特别要求要用别名，但是由于其表现良好而用它。

```
enum authdes_namekind {
    ADN_FULLNAME = 0,
    ADN_NICKNAME = 1
};
```

64 位加密数据：

```
typedef opaque des_block[8];
```

网络用户名的最大长度：

```
const MAXNETNAMELEN = 255;
```

完整的用户名包括客户端网络名，加密的对话密钥和窗口。窗口实际上是信任书的生存时间。如果该时间表示校验时间戳加上窗口已经过期，那么服务器将作废该请求并不同意之。为了保证请求不重犯，除了首次传送服务器坚持认为时间戳大雨先前的那个。在首次传送中，服务器检查窗口校验是否小于窗口。

```
struct authdes_fullname {
    string name<MAXNETNAMELEN>; /* 客户端名          */
    des_block key;              /* PK 加密谈话密钥  */
    opaque window[4];          /* 加密窗口          */
};
```

```
};
```

信任书或者是全名或者是别名:

```
union authdes_cred switch (authdes_namekind adc_namekind) {
case ADN_FULLNAME:
    authdes_fullname adc_fullname;
case ADN_NICKNAME:
    int adc_nickname;
};
```

时间戳把从 1970 年 1 月一日 0 时起的时间值编码:

```
struct timestamp {
    unsigned int seconds;    /* 秒值 */
    unsigned int useconds;  /* 微妙值 */
};
```

校验: 客户变量

窗口校验只用在首次会话中。和一个信任书相关联, 这些项在加密前封装在下面结构中:

```
struct {
    adv_timestamp;          -- one DES block
    adc_fullname.window;   -- one half DES block
    adv_winverf;           -- one half DES block
}
```

该结构用 CBC 模式和 0 输入向量来加密。所有其它时间戳用 ECB 模式加密。

```
struct authdes_verf_clnt {
    des_block adv_timestamp; /* 加密时间戳*/
    opaque adv_winverf[4];  /* 加密窗口校验 */
};
```

校验: 服务器变量

服务器返回收到客户端的时间戳减去一秒。同时它也告知客户端在后续的事务中用别名来传送。

```
struct authdes_verf_svr {
    des_block adv_timeverf; /* 加密校验 */
    int adv_nickname;      /* 客户端的新别名 */
};
```

### 9.3.5 Diffie-Hellman 加密

在该主题中, 有两个常量“BASE”和“MODULUS” [3]。Sun 为 DES 认证协议选择的特别值为:

```
const BASE = 3;
const MODULUS = "d4a0ba0250b6fd2ec626e7efd637df76c716e22d0944b88b"
```

该方法工作的方式用例子很好得到阐述。假设有两个人“A”和“B”, 他们想互相发送

加密信息。所以 A 和 B 都用随机数生成自己的密钥。假设这些密钥表示为 SK (A) 和 SK (B)。他们都发布他们的共钥。共钥的计算方法为：

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

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

符号“\*\*”在这表示求幂。现在 A 和 B 都可以求的公共密钥，用 CK (A, B) 表示，但是不用显示出密钥：

A 的计算方法为：

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

B 的计算方法为：

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

这两个值其实是相等的：

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

去掉“mod MODULUS”部分假设去模运算为简单运算：

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

因此，用前面 B 计算的值替代 PK (B)，对 PK (A) 采取类似计算：

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

那么将：

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

在协议中该共钥 CK (A, B) 不用来加密时间戳。而是用来加密用来加密时间戳的密钥。这么做的原因是尽量少用共钥，以为被破译。破译交谈密钥的严重性小多了，因为谈话时间相对小。

对话密钥用 56 位的 DES 密钥加密，但是共钥是 192 位。为了减少数量，从共钥中按照下面选取 56 位。中间 8 字节从共钥中提取，然后在每个字节的低位加上奇偶校验，生成一个有 8 位校验位的 56 位密钥。

## 10. 记录记号标准

当 RPC 信息传送到上层传送协议（如 TCP），为了检查和从协议错误中恢复，必须界定信息。这个就是记录标志（RM）。Sun 用这种 RM/TCP/IP 传送方式在 TCP 流上传送 RPC 信息。一个 RPC 信息装入一个 RM 记录。

一个记录由一个或多个记录碎片组成。一个记录片是 4 字节头后面跟上 0 到  $(2**31) - 1$  字节片数据。这些字节编码成无符号二进制数；和 XDR 整数一样，字节顺序是从高位到低位。数字编码两个值——布尔值表示该片是否是记录的最后一片（位值 1 表示是最后一片）和 31-位的二进制只表示片数据的字节长度。布尔值是报头的高位；长度是 31 位低位。（注意记录说明不是 XDR 标准格式）。

## 11. RPC 语言

正如有必要描述正式语言的 XDR 数据类型，有必要描述操作这些数据类型的过程。RPC 语言是 XDR 语言的扩展，增加了“程序”，“过程”和“版本”说明。下面的劳资用来描述语

言部分。

## 11.1 RPC 语言描述的例子:

```

这是简单 ping 程序的说明例子。      program PING_PROG {
    /*
     * 最新最完整版本
     */
    version PING_VERS_PINGBACK {
        void
        PINGPROC_NULL(void) = 0;

        /*
         * Ping the client, return the round-trip time
         * (in microseconds). Returns -1 if the operation
         * timed out.
         */
        int
        PINGPROC_PINGBACK(void) = 1;
    } = 2;

    /*
     * 原始版本          */
    version PING_VERS_ORIG {
        void
        PINGPROC_NULL(void) = 0;
    } = 1;
} = 1;

const PING_VERS = 2;      /* latest version */

```

第一个版本 PING\_VERS\_PINGBACK 有两个过程, PINGPROC\_NULL 和 PINGPROC\_PINGBACK. PINGPROC\_NULL 没有参数和返回值,但是它用于计算从客户端到服务器的往返时间。一般,任何 RPC 协议的过程 0 应该有相同的意义,不需要任何认证。第二个过程用作服务器反向 ping 客户机操作,然后返回所用时间(微妙计)。第二版 PING\_VERS\_ORIG 是协议的初始版本,它不包含 PINGPROC\_PINGBACK 过程,当这个程序完成后,它可能从整个协议退下来。

## 11.2 RPC 语言说明

除了增加下面描述的“program-def”外, RPC 语言和 RRC1014 定义的 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
    ("," type-specifier)* ")" "=" constant ";"
```

### 11.3 语法注意事项:

1. 增加了下面的关键字，它们不能用作标志符。“program”和“version”。
- 2 在一个程序定义域里一个版本名和版本数量不能出现两次。
- 3 在一个版本定义里一个过程名不能出现多次。同样适用于过程数量。
- 4 在相同的空间里，程序标志符作为常量和类型标志符。
- 5 只有无符号常量可以赋给程序，版本和过程。

## 附录：程序协议端口映射

端口映射程序映射 RPC 程序和版本号到特定传输端口号的关系。该程序动态绑定远程过程。

因为保留端口数量非常少和可能远程过程调用非常多，因此这样处理非常必要。通过对保留端口运行端口映射程序，其它远程过程的端口数可以查询到。

端口映射同样可以辅助广播 RPC. 一个 RPC 程序通常在不同机器上用不同的端口数，所以不可能直接广播这些程序。但是，端口映射程序有一些固定端口号。所以如果要广播一程序，客户可以发送信息到端口映射程序寻找广播地址。每个广播影射接收广播然后调用客户端的本地服务。当端口映射程序获得从本地服务的答案，将它发送给客户端。

### A.1 端口映射协议说明（用 RPC 语言）

```
const PMAP_PORT = 111; /* portmapper port number */
```

从程序，版本和协议到端口号的映射：

```
struct mapping {
    unsigned int prog;
    unsigned int vers;
    unsigned int prot;
```

```
    unsigned int port;
};
```

“prot”域支持的值:

```
const IPPROTO_TCP = 6;    /* protocol number for TCP/IP */
const IPPROTO_UDP = 17;   /* protocol number for UDP/IP */
```

映射列表:

```
struct *pmaplist {
    mapping map;
    pmaplist next;
};
```

Arguments to callit:

```
struct call_args {
    unsigned int prog;
    unsigned int vers;
    unsigned int proc;
    opaque args<>;
};
```

Results of callit:

```
struct call_result {
    unsigned int port;
    opaque res<>;
};
```

端口映射过程:

```
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 端口映射操作:

现在端口映射程序支持两种协议 (UDP 和 TCP)。在每一个程序上端口映射程序工作在分派的端口 111 (SUNRPC)。

下面对每个端口映射过程进行描述:

PMAPPROC\_NULL:

这个过程不工作。一般, 任何协议的过程 0 都是没有参数和返回值。

PMAPPROC\_SET:

在一台机器上当一个程序首次有效, 它在相同的机器上登记端口映射程序。该程序传送程序号 "prot" 和在其上等待服务请求的端口 "port"。过程返回布尔值: 如果该过程成功建立映射, 该布尔值显示谁的值是对, 反之亦然。如果已经存在三元组 (程序, 版本, 程序号), 那么程序拒绝建立映射。

PMAPPROC\_UNSET:

当一个程序失效, 它从机器上把该端口程序去掉。参数和结果跟函数 "PMAPPROC\_SET" 的一样。参数的协议和端口域省去。

PMAPPROC\_GETPORT:

给定一个程序号 "prog", 版本号 "vers" 和传输协议号 "prot", 该过程返回该程序等待调用请求的端口号。端口值 0 表示该程序没有登记。参数的 "port" 忽略。

PMAPPROC\_DUMP:

该程序列举在端口映射数据库中的所有输入。它没有参数, 返回一系列程序, 版本, 协议和端口值。

PMAPPROC\_CALLIT:

该过程允许客户端在相同的机器上不用知道远程过程调用端口号调用另外一个远程过程。它也可以通过众所周知的端口映射程序用来扩展支持广播到任意远程过程。参数 "prog", "vers", "proc" 和 "args" 字节书是程序号, 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, 那么该过程只有发送一个应答。
2. 端口映射程序和远程过程只用 UDP 通讯。

该过程返回远程过程端口数, 和远程过程应答。

## 参考资料:

[1] Birrell, A. D. & Nelson, B. J., "Implementing Remote Procedure Calls", XEROX CSL-83-7, October 1983.

[2] Cheriton, D., "VMTP: Versatile Message Transaction Protocol", Preliminary Version 0.3, Stanford University, January 1987.

- [3] Diffie & Hellman, "New Directions in Cryptography", IEEE Transactions on Information Theory IT-22, November 1976.
- [4] Mills, D., "Network Time Protocol", RFC-958, M/A-COM Linkabit, September 1985.
- [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, Information Sciences Institute, September 1981.
- [7] Postel, J., "User Datagram Protocol", RFC-768, Information Sciences Institute, August 1980.
- [8] Reynolds, J., and Postel, J., "Assigned Numbers", RFC-1010, Information Sciences Institute, May 1987.
- [9] Sun Microsystems, "XDR: External Data Representation Standard", RFC-1014, June 1987.