Sistemas Distribuídos

Chamada Remota de Procedimento

Referência

- "Sistemas operacionais modernos" Andrew S. TANENBAUM Prentice-Hall, 1995
  - Seção 10.3 pág. 285-304

Conteúdo

- Introdução
- Operação básica
- Objetivos do RPC
- Passagem de parâmetros
- Ligação dinâmica
- Casos de falhas
- Semântica de execução

Conteúdo

- Implementação
- Padronização do RPC
- Identificação de módulos e procedimentos
- XDR – Formatação do RPC
- Portmapper
- RPCgen

Introdução

- Construir um sistema distribuído segundo o modelo cliente/servidor utilizando as primitivas *send* e *receive* (ex.: sockets) pode se tornar complicado: estas primitivas não fazem parte dos conceitos-chave dos sistemas centralizados.

- Deseja-se que sob a ótica do usuário o sistema distribuído se pareça com o centralizado.

Introdução

- Birrell e Nelson (1984): permitir que programas chamassem procedimentos localizados em outras máquinas.

- Ex.: um processo rodando em uma máquina A chama um procedimento em uma máquina B, o processo que chamou é suspenso, e a execução é realizada na máquina B.

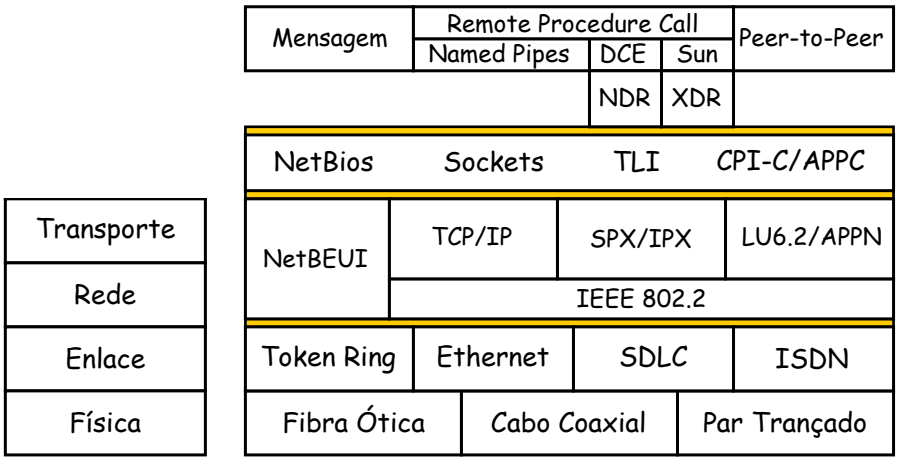
Introdução

- A informação pode ser transportada do processo chamador para o procedimento chamado através de parâmetros, e voltar para o processo como resultado da execução do procedimento.
- Nenhuma operação de troca de mensagem ou de entrada/saída é vista pelo programador.
- Método: Chamada Remota a Procedimento (RPC - Remote Procedure Call)

Introdução

- RPC permite ao projetista de uma aplicação distribuída focalizar na aplicação e não na comunicação entre os vários componentes que compõem a aplicação.
- Exemplo de servidores acessados via RPC:
  - servidores NFS (*Network File System*): permitem acesso transparente à sistemas de arquivos remotos;
  - portmapper: diretório de servidores RPC;
  - servidor de NIS (*Network Information System*):
  - serviço de nomes e *passwords*;
  - servidores de estatísticas.

Introdução

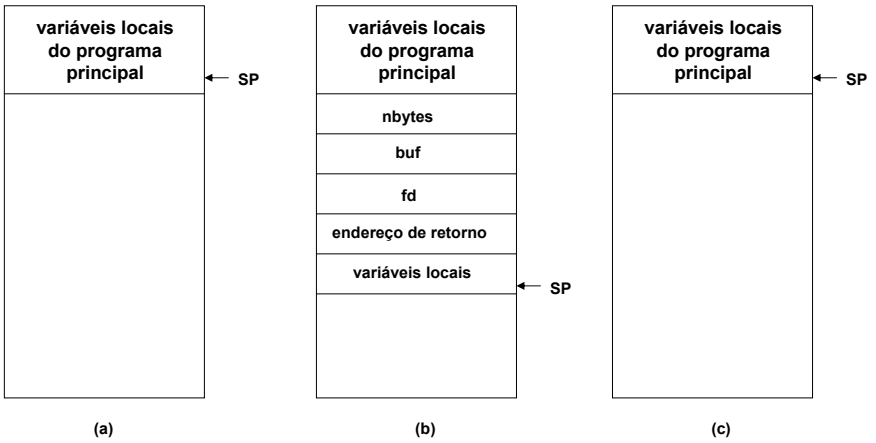


Operação básica

- Considere a chamada

count = read(fd, buf, nbytes)

  - fd: número inteiro
  - buf: vetor de caracteres
  - nbytes: número inteiro



Operação básica

- Em C parâmetros podem ser passados
  - por valor: fd, nbytes - copiados para a pilha, se for modificado não afeta o original
  - por referência: buf - ponteiro para a variável (endereço), se for modificado modifica o original
- Chamada por cópia/restauração: na chamada do procedimento copia variável para a pilha e ao término desta, copia de volta substituindo o valor original

Operação básica

- Chamada local: a rotina *read* é inserida pelo programa ligador no código-objeto do programa.
- Esta rotina coloca os parâmetros em registradores e executa uma chamada de sistema READ, com um TRAP para o kernel.

Operação básica

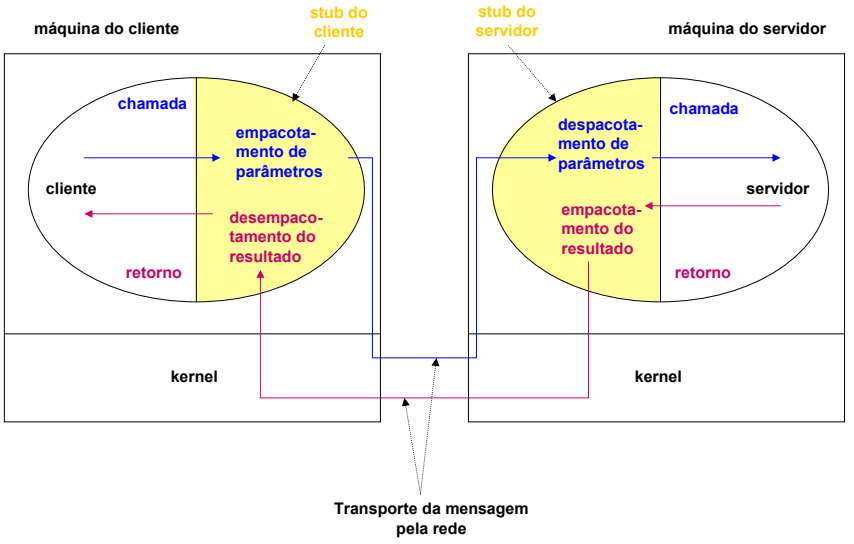
- Em RPC a idéia é fazer com que a chamada remota se pareça com uma chamada local.
- Quando o *read* for um procedimento remoto será inserido uma versão diferente do procedimento, o **stub do cliente**.

Operação básica

- Ele é chamado usando a mesma seqüência do procedimento local e também envia um TRAP para o kernel mas não coloca os parâmetros no registradores.
- Ele empacota os parâmetros em uma mensagem e pede ao kernel para enviar a mensagem ao servidor.

Operação básica

- No servidor a mensagem é transferida para o **stub do servidor**, que é ligado ao servidor real.
- O stub desempacota os parâmetros recebidos da mensagem e chama o procedimento do servidor.
- O retorno do resultado é feito da forma inversa.



Operação básica

- Todos os detalhes da troca de mensagem são escondidos nos stubs.
- Passos realizados em uma chamada remota a procedimento:
  1. O procedimento do cliente chama o stub do cliente da maneira usual.
  2. O stub do cliente constrói uma mensagem e envia um trap ao kernel.
  3. O kernel envia uma mensagem ao kernel remoto.
  4. O kernel remoto entrega a mensagem ao stub do servidor.

Operação básica

5. O stub do servidor desempacota os parâmetros constantes na mensagem e chama o servidor.
6. O servidor realiza seu trabalho e retorna o resultado para um buffer dentro do stub.
7. O stub do servidor empacota tais resultados em uma mensagem e emite um trap para o kernel.
8. O kernel remoto envia a mensagem para o kernel do cliente.
9. O kernel do cliente entrega a mensagem ao stub do cliente.
10. O stub desempacota os resultados e os fornece ao cliente.

Objetivos do RPC

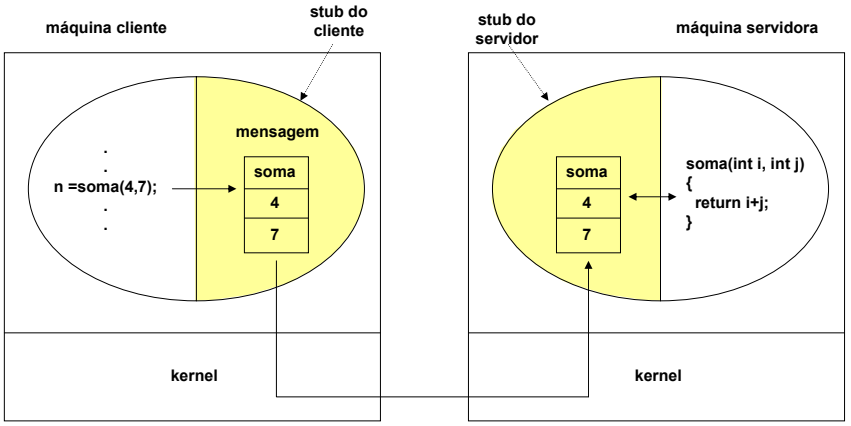
- **Objetivos** de uma implementação de RPC:
  - a chamada de um procedimento deve ser transparente no que diz respeito à localização (local ou remota) do procedimento chamado;
  - a semântica de uma chamada remota deve ser idêntica à de uma chamada local: passa parâmetros e retorna resultados;
  - no caso de erro de implementação, uma chamada local pode não retornar, caso o procedimento chamado entre num laço infinito, porém falhas de comunicação local entre o procedimento chamador e o chamado equivalem a uma falha total do sistema.

Objetivos do RPC

- Esses objetivos foram ***parcialmente conseguidos*** nas implementações atuais:
  - a sintaxe de uma chamada local é idêntica à de uma chamada remota, porém o programador deve codificar uma rotina de interface com o sistema **run-time** do RPC chamada de rotina **stub;**
  - módulos remotos devem ter seus procedimentos "registrados" junto ao ambiente de execução do RPC;
  - a semântica de execução não é transparente a falhas de comunicação, ou seja, uma chamada remota pode retornar um erro do tipo "timeout".

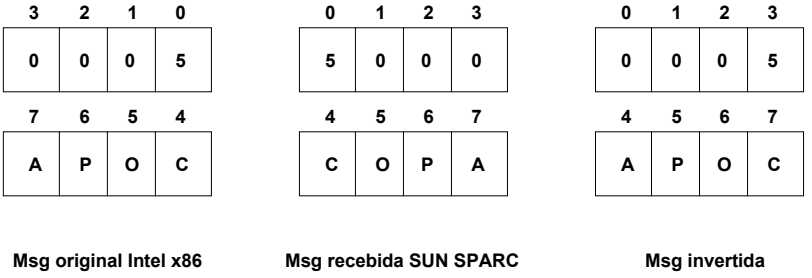
Passagem de parâmetros

- Função do stub do cliente: empacotar os parâmetros em uma mensagem (ordenação de parâmetros) e enviá-los ao stub do servidor.
- Considere o procedimento *soma(i,j)* que faz a soma algébrica de dois inteiros.
- A mensagem deve incluir o nome ou número do procedimento chamado.



Passagem de parâmetros

- Problemas:
  - máquinas diferentes com diferentes representações para números, caracteres e outros dados (ASCII, EBCDIC).
  - representação de números inteiros: (complementos de 1 ou complementos de 2)
  - big endian x little endian:
    - big endian: numera bytes da direita para esquerda (Intel 386)
    - little endian: numera bytes da esquerda para direita (SUN SPARC)



Msg original Intel x86 | Msg recebida SUN SPARC | Msg invertida

Passagem de parâmetros

- Para o mecanismo funcionar é necessária a informação do número e dos tipos dos parâmetros.
  - Cliente e servidor possuem esta informação.

```
exemplo(char x, float y, int z[5])
{
...
}
```

exemplo	
	x
y	
5	
Z[0]	
Z[1]	
Z[2]	
Z[3]	
Z[4]	

Passagem de parâmetros

- Como representar as informações nas mensagens?

Forma canônica

- Ex.: complemento de 2 para inteiros, ASCII para caracteres, 0 (falso) e 1 (verdadeiro) para booleanos, formato IEEE para números em ponto flutuante e little endian.

Passagem de parâmetros

- Problema: ineficiência
  - na comunicação entre um cliente big endian com um servidor big endian são feitas duas conversões (uma para little endian e de volta para big endian)
- Solução: identificar nos primeiros bytes o formato utilizado

Passagem de parâmetros

- Como os stubs são gerados?
  - Dada uma especificação do procedimento do servidor e as regras para codificação, o formato da mensagem é determinado de maneira inequívoca
  - É possível fazer com que o compilador leia a especificação do servidor e gere os stubs do cliente e do servidor

Passagem de parâmetros

- Passagem de ponteiros
  - os ponteiros só tem significado no espaço de endereçamento do processo no qual está sendo usado
- Se o ponteiro aponta para um vetor de caracteres pode-se copiar todo o vetor e enviá-lo ao servidor. Se o vetor for modificado pelo servidor ele é copiado de volta para que o cliente possa atualizá-lo (cópia/restauração).

Passagem de parâmetros

- Otimização do processo:
  - se o vetor for apenas de entrada não é necessária a cópia de volta para o cliente
  - se for de saída não precisa ser copiado do cliente para o servidor
- É necessária uma especificação na linguagem indicando se os parâmetros são de entrada ou saída
- Esta técnica não funciona para ponteiros para estruturas de dados arbitrárias (grafo complexo)

Ligação dinâmica

- Como o cliente localiza o servidor?
  - Gravar no hardware sua localização: inflexível
  - Ligação dinâmica
- Especificação de um servidor de arquivos usando as primitivas send e receive:


```
void main() {
    struct message m1, m2;
    int r;
    while(1) {
        receive(FILE_SERVER, &m1);
        switch(m1.opcode) {
            case CREATE:
                r = do_create(&m1, &m2); break;
            case READ:
                r = do_read(&m1, &m2); break;
            case WRITE:
                r = do_write(&m1, &m2); break;
            case DELETE:
                r = do_delete(&m1, &m2); break;
            default:
                r =E_BAD_OPCODE;
        }
        m2.result = r;
        send( m1.source, &m2);
    }
}
```

Ligação dinâmica

- Especificação formal de um servidor:

```
especificação do servidor de arquivos versão 3.1:
long read(in char name[MAX_PATH], out char buf[BUF_SIZE],
        in long bytes, in long position);
long write(in char name[MAX_PATH], in char buf[BUF_SIZE],
        in long bytes, in long position);
int create(in char name[MAX_PATH], in int mode);
int create(in char name[MAX_PATH]);
end;
```


Ligação dinâmica

- Para cada procedimento é especificado o tipo do parâmetro: in, out ou inout
  - relativo ao servidor: in é enviado do cliente para o servidor
- A partir da especificação formal são criados os stubs do cliente e servidor

Ligação dinâmica

- Ao iniciar a execução do servidor, a chamada de inicialização **exporta** a interface do servidor: o servidor envia uma mensagem ao programa **ligador** (binder), para que este tome conhecimento de sua existência (**registro do servidor**)
- Registro: nome do servidor, número da versão, identificador (número de 32 bits) e **manipulador** (localização - endereço Ethernet ou endereço IP + identificador do processo)

Chamada	Entrada	Saída
Register	**Nome, versão, manipulador, identificação**	
Deregister	**Nome, versão, identificação**	
Lookup	**Nome, versão**	**Manipulador, identificação**

Ligação dinâmica

- Ao chamar um dos procedimentos remotos pela primeira vez o stub verifica que não está ligado ao servidor → envia mensagem ao ligador solicitando a **importação** da versão 3.1 da interface
  - se não houver: erro
  - se houver: retorna o manipulador e identificação (para entrega de mensagens ao destino correto)

Ligação dinâmica

- Vantagem - flexibilidade:
  - vários servidores para uma mesma interface
  - tolerância a falhas: verificar se o servidor ainda está ativo
  - autenticação de cliente: lista de clientes permitidos
- Desvantagem:
  - overhead de importação e exportação
  - sistema distribuído razoável: vários ligadores distribuídos → excessiva troca de mensagens para sincronização

Casos de falhas

- Objetivo de RPC: fazer a chamada remota se parecer com a local. Exceções:
  - variáveis globais
  - parâmetros de entrada e saída
  - ocorrência de erro
- Falhas:
  1. O cliente não é capaz de localizar o servidor
  2. A mensagem do cliente para servidor é perdida
  3. A mensagem resposta do servidor é perdida
  4. O servidor sai do ar após receber solicitação
  5. O cliente sai do ar após ter enviado solicitação

Falha: cliente não localiza servidor

- Servidor está fora do ar
- Cliente solicita versão antiga do servidor
  - usar valor de retorno da função para indicar erro (ex.: -1)
    - se o resultado da função for -1 (ex.: 3 - 4)
  - gerar exceção ou usar sinal (SIGNOSERVER)
    - nem todas as linguagens suportam exceções
    - mecanismo não é transparente

Falha: perda de mensagem de solicitação de serviço

- Uso de temporizador pelo kernel. Se expirar a mensagem é enviada novamente
- Se ocorrer a perda de várias mensagens desistir de enviar mensagens

Falha: perda de mensagem com resposta

- Se a resposta não chegar o que ocorreu?
  - A solicitação se perdeu?
  - A resposta se perdeu?
  - O servidor é lento?
- Operação **idempotente**: pode ser executada quantas vezes for necessário sem gerar efeito colateral. Ex.: ler os primeiros 512 bytes de um arquivo

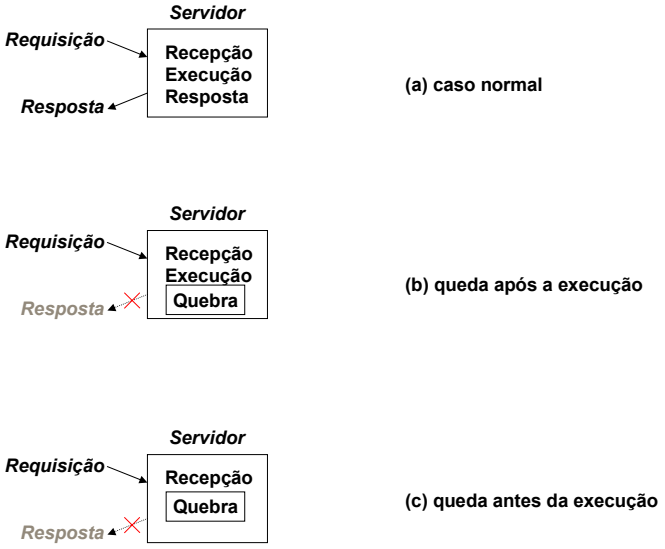
Falha: perda de mensagem com resposta

- Operação não idempotente: ex.: transferir dinheiro entre contas de um banco

- Soluções:
  - estruturar solicitações para que sejam idempotente
  - numerar seqüencialmente as requisições
  - identificar (bit) as mensagens originais e as retransmissões

Falha: queda do servidor

- Eventos em um servidor no caso normal e no caso de queda
  - em (b) o sistema deve reportar falha
  - em (c) o sistema pode retransmitir requisição



Falha: queda do servidor

- O kernel do cliente não tem como saber que é quem. 3 linhas de ação:
  - **Semântica de no mínimo uma vez**: o cliente continua tentando até receber uma resposta → garante que a chamada remota será executada
  - **Semântica de no máximo uma vez**: o kernel deve desistir e reportar problema → garante que a chamada foi executada no máximo uma vez ou nenhuma
  - Não garantir absolutamente nada: a chamada remota pode ter sido executada 0 ou mais vezes
- Ideal: **semântica de exatamente uma vez**

Falha: queda do cliente

- Cliente manda mensagem para servidor e falha antes da resposta do servidor
- Processamento sem que ninguém espere por resposta (processamento **órfão**):
  - gasta tempo de CPU
  - pode bloquear recursos
  - o cliente pode fazer nova chamada gerando confusão
- 4 soluções

Falha: queda do cliente

- **Extermínio**: antes que o stub envie a mensagem deve gravar em arquivo (persistente) a informação que deseja obter. Após o boot da máquina do cliente o arquivo é verificado e os órfãos são assassinados. Problemas:
  - sobrecarga com gravação em arquivo
  - o órfão pode criar outros órfão difíceis de localizar
  - a rede pode ser dividida em 2 (falha em um gateway) tornando impossível eliminar os órfãos

Falha: queda do cliente

- **Reencarnação**: o tempo é dividido em eras numeradas seqüencialmente. Quando o cliente se recupera envia mensagem de broadcast inaugurando nova era. Quando o servidor recebe esta mensagem elimina órfão da geração anterior.

Falha: queda do cliente

- **Reencarnação branda**: quando chega a mensagem de broadcast, cada uma das máquinas deve verificar se existe processamento remoto solicitado. Se seu proprietário não puder ser localizado o processamento é descartado.

Falha: queda do cliente

- **Expiração**: é dada uma fatia de tempo (T) a cada chamada remota. Se neste tempo não terminar processamento deve solicitar explicitamente nova fatia de tempo. Após a queda basta o cliente aguardar o tempo T.
  - Problema: escolha de um T adequado

Semântica de Execução

- Idealmente uma chamada remota deveria garantir a execução do procedimento chamado exatamente uma vez, mas isto é impossível de garantir na presença de erros de comunicação, mesmo se utilizarmos o protocolo TCP .
- Se ocorre *timeout* no chamador após uma chamada ser encaminhada pelo protocolo de comunicação, várias situações podem ocorrer:
  - a comunicação foi interrompida e a chamada se perdeu ou o servidor caiu antes de executá-la;
  - a chamada foi executada mas a comunicação foi interrompida e a resposta se perdeu, ou o servidor da chamada caiu após executar a chamada.

Semântica de Execução

- Mesmo se o "run-time" do RPC for implementado sobre TCP, no 1º caso a chamada remota não foi executada e no 2º caso foi executada uma vez. Não obstante, o chamador não fica sabendo qual das duas possibilidades ocorreu, ou seja, a semântica de execução é: "no máximo uma execução" ("at most once").
- No caso de retorno sem erro, o protocolo TCP garante que a execução ocorreu exatamente uma vez , o que caracteriza situação de normalidade.

Semântica de Execução

- Se o protocolo utilizado for UDP, a situação é bem pior, pois nenhuma garantia pode ser dada:
  - o pacote com o pedido de chamada pode se perder ou ser duplicado, e a resposta pode se perder, de forma que a chamada pode não ser executada, ou ser executada uma ou mais vezes;
  - a semântica de execução é - se há retorno: "**execução uma ou mais vezes**" ("at least once semantics"), se não há retorno: "**execução zero ou mais vezes**".

Padronização do RPC

- Um sistema RPC completo foi especificado e implementado pela Sun (inspirado em implementações pioneiras na Xerox) e posteriormente padronizado pela **ONC - Open Network Computing**.

Padronização do RPC

- Esta padronização da ONC inclui:
  - um protocolo de comunicação incluindo formatos de msgs para chamada remota e retorno de resultados;
  - um protocolo para especificação de representação de dados independente de arquitetura ("big endian *versus* little endian")';
  - regras para identificação de módulos remotos e de seus procedimentos;
  - escolha pela aplicação de 2 protocolos: TCP ou UDP;
  - especificação de semântica de execução compatível com a do protocolo de transporte escolhido pela aplicação.

Identificação de Módulos Remotos e de seus Procedimentos

- Terminologia
  - **programa remoto ≅ módulo remoto ≅ servidor RPC**
  - **chamada remota ≅ cliente**
- como o RPC é um mecanismo genérico, é necessário indicar qual servidor será contactado e qual procedimento dentro deste servidor deverá ser evocado;
- o parâmetro correspondente ao nó remoto é usado pelo RPC para obter o endereço IP do host onde se encontra executando o servidor;

Identificação de Módulos Remotos e de seus Procedimentos

- O parâmetro do nome do programa é, na realidade, um número atribuído por um administrador da rede para designar processos servidores individuais.
- O Padrão RPC especifica que, para cada programa remoto executando em um computador, deve ser associado um par de números de 32 bits escolhidos pelo programador.

(#prog, #versão)

  - **#prog** - definidos pela SUN
  - **#versão** - define a versão de implementação do Sist. RPC, começando por 1.

Identificação de Módulos Remotos e de seus Procedimentos

- Da mesma forma, também é atribuído um inteiro para cada procedimento dentro do programa remoto, identificado univocamente pela tripla:

  (#prog, #versão, #proc)

  - **#prog** identifica o programa remoto
  - **#versão** - define a versão de implementação do Sist. RPC
  - **#proc** identifica um procedimento dentro do programa remoto

- Os procedimentos são numerados seqüencialmente a partir de 1. Por convenção, o número 0 é reservado para um procedimento de ***echo*** que pode ser utilizado para testar se o programa remoto pode ser alcançado.

Identificação de Módulos Remotos e de seus Procedimentos

- Para evitar conflitos entre programas de diferentes organizações, a Sun divide os números dos programas em grupos, como ilustrado a seguir:

de	até	valores atribuídos por
0x00000000	0x1FFFFFFF	Sun Microsystems, Inc.
0x20000000	0x3FFFFFFF	Gerente Local
0x40000000	0x5FFFFFFF	Transiente (temporário)
0x60000000	0x70000000	Reservado
... ...	... ...	Reservado
0xE000000	0xFFFFFFFF	Reservado

Identificação de Módulos Remotos e de seus Procedimentos

Nome	Número Atribuído	Descrição
portmap	100000	portmapper
rstatd	100001	rstat, rup, e perfmeter
rusersd	100002	remote users
nfs	100003	network file system
ypserv	100004	yp (NIS)
mountd	100005	mount, showmount
dbxd	100006	DBXprog (debugger)
ypbind	100007	NIS binder
walld	100008	rwall, shutdown
yppassword	100009	yppassword
etherstatd	100010	ethernet statistics

Identificação de Módulos Remotos e de seus Procedimentos

Nome	Número Atribuído	Descrição
rquotad	100011	rquotaprog, quota, rquota
sprayd	100012	spray
selection_svc	100015	selection service
dbsessionmgr	100016	unify, netdbms, dbms
rexd	100017	rex, remote_exec
office_auto	100018	alice
lockd	100020	kimprog
lockd	100021	nimprog
statd	100024	status monitor
bootparamd	100026	bootstrap
pcnfsd	150001	NFS for PC

Identificação de Módulos Remotos e de seus Procedimentos

- A comunicação do cliente com um programa remoto é feita através de sockets TCP ou UDP.
- É inconveniente prefixar um número de port para o programa remoto ou servidor, pois:
  - não há um mapeamento entre #prog (32 bits) e porta (16 bits);
  - programas remotos são em geral transientes.

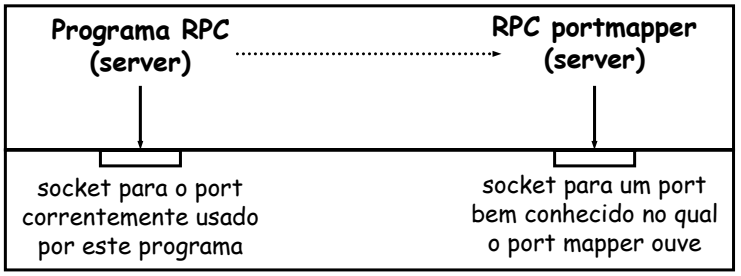
Identificação de Módulos Remotos e de seus Procedimentos

- Solução: mapeamento dinâmico entre **#prog** e **#port**
  - cada máquina que suporta RPC possui um servidor RPC chamado **portmapper** associado à um **port** conhecido 111;
  - quanto um programa remoto é iniciado, ele obtém um *socket* local com um #port transiente e o repassa ao servidor **portmapper** de sua máquina através da mensagem RPC contendo (#prog, #versão, #port transiente);
  - ao receber a mensagem, o servidor **portmapper** registra numa tabela local em memória;
  - quando a aplicação do cliente é iniciada ela envia uma mensagem RPC contendo (#prog, #versão) para o #port 111 do **portmapper**;
  - o **portmapper** consulta sua tabela e devolve ao cliente o número do **port** associado ao programa remoto, ou seja, do servidor;

Identificação de Módulos Remotos e de seus Procedimentos

- a partir desse ponto os procedimentos de execução localizados no cliente utilizarão este número de **port** recebido, para comunicar com o servidor RPC (programa remoto).

- Todos os passos acima são transparentes ao programador da aplicação RPC.



Identificação de Módulos Remotos e de seus Procedimentos

- **Política de Retransmissão** - temporização de duração fixa e um número fixo de retransmissões.

- **Problema da Exclusão Mútua**
  - em princípio, vários clientes poderiam estar chamando simultaneamente procedimentos do mesmo módulo remoto e a execução concorrente desses procedimentos poderia gerar inconsistência nas estruturas de dados globais aos procedimentos.
  - em vez de permitir execução concorrente com mecanismos de exclusão mútua, a especificação de RPC resolveu o problema da forma mais simples possível --- apenas um procedimento de cada vez de um módulo remoto pode estar em execução, assim chamadas pendentes são enfileiradas.

Cliente RPC

- O lado do cliente do protocolo RPC é bastante simples. O cliente ativa **callrpc**, passando-lhe oito parâmetros:
  - nome do nó remoto;
  - nome do servidor;
  - número da versão do servidor;
  - procedimento a ser ativado;
  - tipo do parâmetro de entrada enviado ao procedimento remoto;
  - parâmetro de entrada ou uma estrutura no caso de mais de um parâmetro de entrada;
  - tipo do parâmetro de saída retornado pelo procedimento remoto;
  - parâmetro de saída ou estrutura de dado no caso de mais de um parâmetro de saída.

Cliente RPC

- nó remoto - usado para obter o endereço IP do host onde se encontra executando o servidor;
- nome do servidor - qual servidor será contactado;
  - parâmetro do nome do programa é, na realidade, um número atribuído por um administrador da rede para designar processos servidores individuais.
- versão do servidor - RPC facilita a expansão do servidor ao permitir a atribuição de um número de versão para cada nova versão do servidor, assim versões novas e antigas de um servidor poderão existir;
- procedimento - procedimento dentro deste servidor que deverá ser evocado;

Cliente RPC

- parâmetro de entrada - parâmetros individuais precisam ser combinados em estruturas para que possam ser enviados ao servidor ou programa remoto;
- parâmetro de saída - no caso de mais de um parâmetro de saída, os resultados serão retornados em uma estrutura de dados, e só após serem recebidos serão individualmente extraídos da estrutura.

XDR - Formatação no RPC

- Como máquinas diferentes possuem formatos diferentes, a RPC utiliza um protocolo chamado ***External Data Representation*** (XDR) para traduzir dados de/para um formato de intercâmbio pré-definido:
  - Várias rotinas de serviço são fornecidas com a RPC para conversão de tipos básicos de dado para XDR.
  - Estas incluem **xdr_int**, **xdr_double**, **xdr_float**, **xdr_long**, **xdr_string**, etc. O parâmetro tipo na evocação do procedimento remoto corresponde ao nome da rotina de serviço XDR que o RPC deve chamar para converter o parâmetro correspondente a ser enviado/recebido para/do procedimento.

XDR - Formatação no RPC

- Por exemplo, suponha que um procedimento remoto, chamado **LOOKUP**, converta um nome em endereço (trata-se de uma função típica do servidor de nome).
- O comando **callrpc** poderia ser o seguinte:

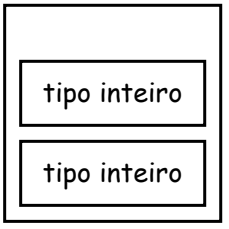
  callrpc(estação, servidor, minha_versão, LOOKUP, xdr_string, nome, xdr_long, endereço)

- ... no caso onde vários parâmetros devem ser transferidos, estes terão que ser colocados em uma única estrutura de dados.

XDR - Formatação no RPC

- Por exemplo, caso dois inteiros tenham que ser enviados a um procedimento remoto, eles podem ser combinados na estrutura mostrada abaixo:

```
struct arg {
   int x;
   int y;
}

struct arg meus_dados;
meus_dados.x = ...;
meus_dados.y = ...;
```



XDR - Formatação no RPC

- Neste caso, o usuário deverá fornecer a sua própria rotina de serviço XDR para o comando **callrpc**.
- Escrever uma rotina deste tipo significa percorrer a estrutura de dados chamando a rotina XDR apropriada para cada elemento na estrutura.
- Seja a rotina callrpc usada na seguinte chamada:

callrpc(ESTAÇÃO, SERVIDOR, VERSÃO, média, xdr_argumentos, meus_dados, xdr_float, resultado)

XDR - Formatação no RPC

- Exemplo de rotina para conversão de estrutura de dados para XDR, para o exemplo da rotina apresentada na página anterior:

```
struct arg *pointer;      /* aponta para "meus_dados" */

XDR *xdr_argumentos;          /* aponta p/ o fluxo de dados XDR */

{
  xdr_int( xdr_sp, &pointer -> x );  /* converte o primeiro elemento */
  xdr_int( xdr_sp, &pointer -> y );  /* converte o segundo elemento */
  return;
}
```

Servidor RPC

- O servidor deve registrar os seus procedimentos que poderão ser acessados remotamente através da rotina **registerrpc** no servidor de **portmapper;**
- Posteriormente, ativa-se a rotina da biblioteca RPC denominada **svc_run** - esta rotina permite ao servidor esperar as chamadas aos procedimentos registrados.

Servidor RPC

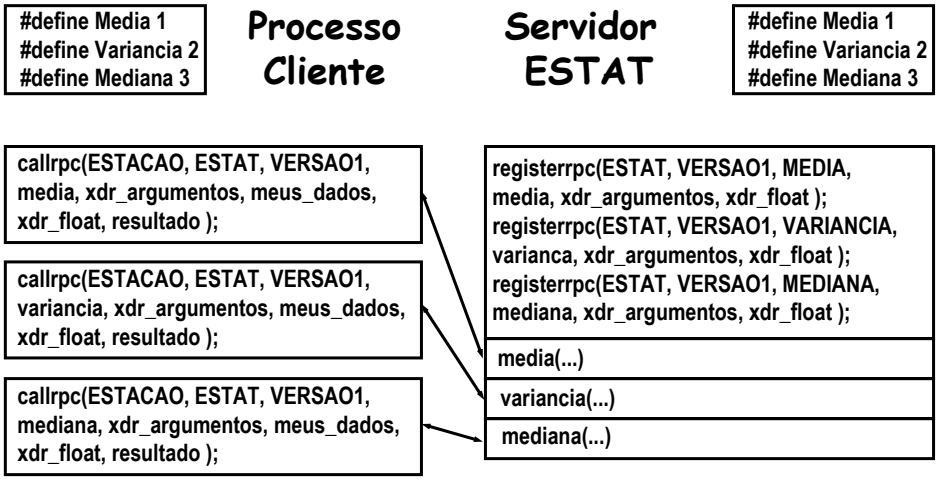
- A rotina **registerrpc** da biblioteca RPC utiliza seis (06) parâmetros discriminados a seguir:
  - número do programa a ser registrado;
  - número da versão;
  - número do procedimento que está sendo registrado;
  - nome do procedimento a ser evocado;
  - rotina de serviço XDR a ser evocada para os parâmetros de entrada;
  - rotina de serviço XDR a ser evocada para os parâmetros de saída.

Servidor RPC

- O diagrama abaixo mostra como registrar o exemplo anterior (conversão de estrutura de dados para XDR):

```
#define UM  1

registerrpc( NÚMERO_SERVIDOR, VERSÃO, UM,
             média, xdr_argumentos, xdr_float );

svc_sun;  /* não retorna */
```


Servidor RPC



Servidor Portmapper

- Conforme mencionado anteriormente, os números de programa RPC são divididos em grupos de 20000000(hex) como mostrado anteriormente.
- A faixa definida pela Sun é usada para processos servidores de interesse geral para usuários da Internet como, por exemplo, Network File System - NFS, Yellow Pages, e outras aplicações.

de	até	valores atribuídos por
0x00000000	0x1FFFFFFF	Sun Microsystems, Inc.

Servidor Portmapper

- A faixa seguinte pode ser usada para servidores específicos da empresa e os números são alocados pelos administrador da rede. A terceira faixa é alocada dinamicamente por rotinas de serviço RPC.

de	até	valores atribuídos por
0x00000000	0x1FFFFFFF	Sun Microsystems, Inc.
0x20000000	0x3FFFFFFF	Gerente Local
0x40000000	0x5FFFFFFF	Transiente (temporário)

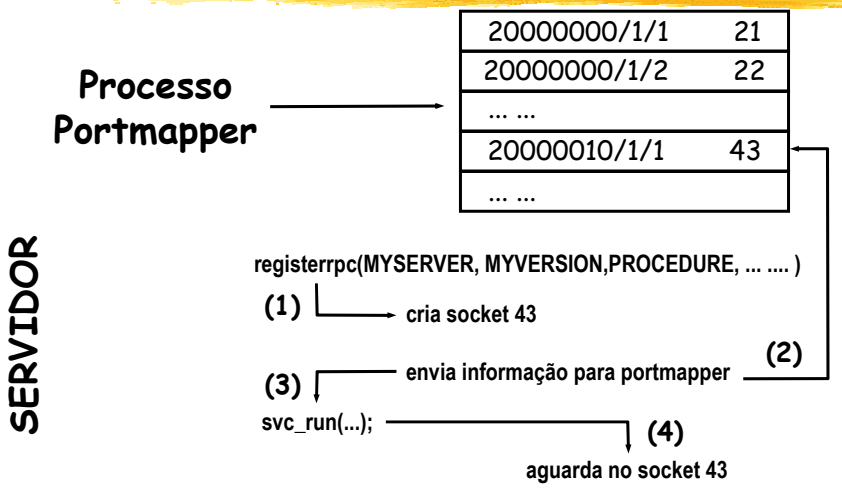
Servidor Portmapper

- Os números de programa são ligados dinamicamente a números de porta UDP e TCP por meio do processo RPC denominado **portmapper**. O processo **portmapper** é executado na porta 111.
- Vejamos agora um exemplo que ilustra o uso do **portmapper** - este exemplo foi extraído do livro Introdução aos Sistemas Cliente/Servidor: Guia prático para profissionais de sistemas. Paul E. Renaud.

Servidor Portmapper

- Quando um servidor evoca **registerrpc (1),** a rotina da biblioteca cria um *socket* e envia uma mensagem para o **portmapper** informando que o *socket* corresponde a um determinado procedimento remoto **(2).**
- O **portmapper** utiliza esta informação para atualizar uma tabela interna contendo entradas constituídas de **programa /versão/procedimento** associadas a descritores de *socket*.
- A chamada registerrpc retorna **(3)** e o servidor chama **svc_run (4)** passando a aguardar uma evocação do procedimento registrado.

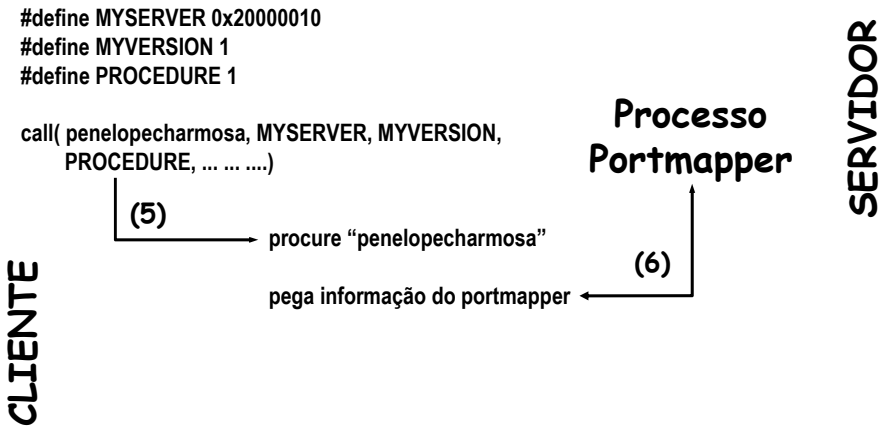
Servidor Portmapper



Servidor Portmapper

- Quando um cliente ativa **callrpc (5),** a rotina da biblioteca RPC usa o parâmetro correspondente ao nome da estação onde se encontra o servidor para obter o endereço IP da estação.
- Em seguida envia uma mensagem para o **portmapper** rodando nesta máquina **(6).** O **portmapper** pesquisa a tabela usando como entrada o número de **programa/ versão/proced**to fornecido pelo cliente e determina o descritor de socket que o servidor está usando.
- O **portmapper** responde à rotina da biblioteca RPC que executa do lado do cliente enviando o descritor de socket que o servidor está usando.

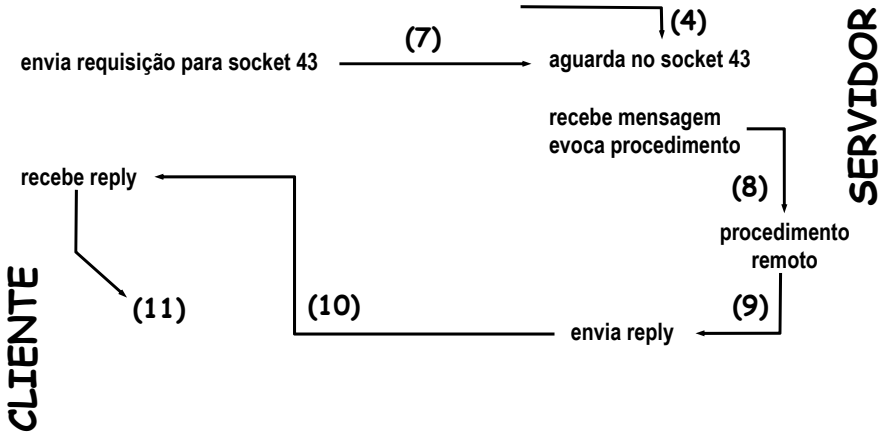
Servidor Portmapper



Servidor Portmapper

- Na seqüência, **callrpc** envia os parâmetros da chamada de procedimento para o processo servidor **(7)** e o procedimento é ativado **(8).**
- Quando o procedimento retorna **(9),** os resultados são enviados de volta a **callrpc (10),** que depois retorna os resultados para o cliente **(11).**

Servidor Portmapper



Servidor Portmapper

- No caso do exemplo anterior, somente um procedimento remoto foi registrado por parte do servidor.
- Caso mais de um procedimento seja registrado, quando da evocação de **svc_run** o servidor ficará aguardando a evocação de um procedimento remoto em um conjunto de sockets.

Exemplo usando RPC

- Envia e recebe um inteiro
- Envia e recebe um float
- Envia e recebe uma string

  - Server - PORTMAPPER deve estar rodando
  - Client - PORTMAPPER e SERVIDOR REMOTO devem estar rodando

Servidor

```
/* Server RPC - SRV-RPC.C File - Receive an integer, a
   float and a string and return them respectively */
/* PORTMAPPER MUST BE RUNNING */

/* Include Files */
#include <rpc/rpc.h>
#include <stdio.h>

/* Defines for INTEGER Function */
#define intrcvprog ((u_long)150000)
#define intvers    ((u_long)1)
#define intrcvproc ((u_long)1)

/* Defines for FLOAT Function */
#define fltrcvprog ((u_long)150102)
#define fltvers    ((u_long)1)
#define fltrcvproc ((u_long)1)
```

... Servidor

```
/* Defines for STRING Function */
#define strrcvprog ((u_long)150204)
#define strvers    ((u_long)1)
#define strrcvproc ((u_long)1)
/* Program Main */
main() {
  /* Declaration of Functions */
  int *intrcv();          /* declaration of "intrcv"
  function */
  float *floatrcv( );     /* declaration of
  "floatrcv" function */
  char **strrcv( char** ); /* declaration of
  "strrcv" function */
```


... Servidor

```
  /* REGISTER PROG, VERS AND PROC WITH THE PORTMAPPER */
  /* Program that sends an INTEGER to the remote host */
  registerrpc( intrcvprog, intvers, intrcvproc,
               intrcv, xdr_int, xdr_int );
  printf("INTRCV Registration with Port Mapper
   COMPLETED.\n\n");

  /* Program that sends an FLOAT to the remote host */
  registerrpc( fltrcvprog, fltvers, fltrcvproc,
               floatrcv, xdr_float, xdr_float);
  printf("FLOATRCV Registration with Port Mapper
   COMPLETED.\n\n");

  /* Program that sends an STRING to the remote host */
  registerrpc( strrcvprog, strvers, strrcvproc,
               strrcv, xdr_wrapstring, xdr_wrapstring );
  printf("STRRCV Registration with Port Mapper
   COMPLETED.\n\n");
```

... Servidor

```
  /* Espera por chamadas aos procedimento
  registrados */
  svc_run();


  /* If "svc_run()" return error, print a
  message and exit */
  printf("Error: SVC_RUN returned!\n");
  exit(1);
}
/* End of Program Main */
```


... Servidor

```
/* Function "intrcv()" */
int *intrcv(int *in)
{
   int *out;

   printf("Integer received: %d   ",*in);
   out = in;
   printf("Integer being returned:
  %d\n\n",*out);
   return (out);
}
```

... Servidor

```
/* Function "floatrcv()" */
float *floatrcv(float *in)
{
   float *out;

   printf("Float received: %e   ",*in);
   out = in;
   printf("Float being returned:
  %e\n\n",*out);
   return(out);
}
```


... Servidor

```
/* Function "strrcv()" */
char **strrcv(char **in)
{
  char **out;    /* char *out = malloc(64); */

  printf("String received: %s   ",*in);
  /*   strcpy(out, *in); */
  out = in;
  printf("String being returned: %s\n\n",
  *out);
  return (out);
}
```

Cliente

```
/* Client RPC - CLT-RPC.C File */
/* Send an integer, a float and a string to the remote host
   and receive the integer, float and string back  */
/* PORTMAPPER AND REMOTE SERVER MUST BE RUNNING */

/* Include Files */
#include <rpc/rpc.h>
#include <stdio.h>

/* Defines for INTEGER Function */
#define intrcvprog ((u_long)150000)
#define intvers    ((u_long)1)
#define intrcvproc ((u_long)1)
/* Defines for FLOAT Function */
#define fltrcvprog ((u_long)150102)
#define fltvers    ((u_long)1)
#define fltrcvproc ((u_long)1)
```


... Cliente

```
/* Defines for STRING Function */
#define strrcvprog ((u_long)150204)
#define strvers    ((u_long)1)
#define strrcvproc ((u_long)1)

/* Main Program */
main(int argc, char *argv[]) {
  int in_int, out_int; /* param for "intrcv" function */
  float in_flt, out_flt;  /* param for "floatrcv" fnct*/

  /* parameters for "strrcv" function */
  char *in_str = malloc(64);
  char *out_str = malloc(64);

  int error;
```

... Cliente

```
 /* how enter the arguments in a command line */
if( argc < 5 ) {
  fprintf(stderr,"Usage: clt-srv <hostname> <integer>
          <float> <string>\n");
  exit (-1);
} /* endif */

/* assign the arguments to the parameters */
in_int = atoi(argv[2]);  /* ascii to integer */
in_flt = atof(argv[3]);  /* ascii to float */
strcpy(in_str, argv[4]);  /* copy argv[4] to in_str */
```


... Cliente

```
/* call for "intrcv" function */
error = callrpc( argv[1], intrcvprog, intvers,
                 intrcvproc, xdr_int, (char *)&in_int,
                 xdr_int, (char *)&out_int);
if (error != 0) {
   fprintf(stderr,"ERROR: callrpc failed: %d
\n",error);
   fprintf(stderr,"Program: %d Version: %d Procedure:
%d", intrcvprog, intvers, intrcvproc);
    exit(1);
 } /* endif */

printf("Integer sent: %d    Integer received: %d\n\n",
       in_int, out_int);
```

... Cliente

```
/* call for "floatrcv" function */
error = callrpc(argv[1], fltrcvprog, fltvers,
                fltrcvproc, xdr_float, (char *)&in_flt,
                xdr_float, (char *)&out_flt);
if (error != 0) {
   fprintf(stderr,"ERROR: callrpc failed: %d
\n",error);
   fprintf(stderr,"Program: %d Version: %d Procedure:
%d", fltrcvprog, fltvers, fltrcvproc);
    exit(1);
 } /* endif */

printf("Float sent: %e   Float received: %e\n\n",
       in_flt, out_flt);
```


... Cliente

```
/* call for "strrcv" function */
error = callrpc( argv[1], strrcvprog, strvers,
          strrcvproc, xdr_wrapstring, (char *)&in_str,
          xdr_wrapstring, (char *)&out_str);
if (error != 0) {
   fprintf(stderr,"ERROR: callrpc failed: %d
\n",error);
   fprintf(stderr,"Program: %d Version: %d Procedure:
%d", strrcvprog, strvers, strrcvproc);
    exit(1);
 } /* endif */

printf("String sent: %s   String received: %s\n\n",
       in_str, out_str);
exit(0);
} /* end of Main Program */
```

A Implementação RPC da Sun Microsystems - Introdução

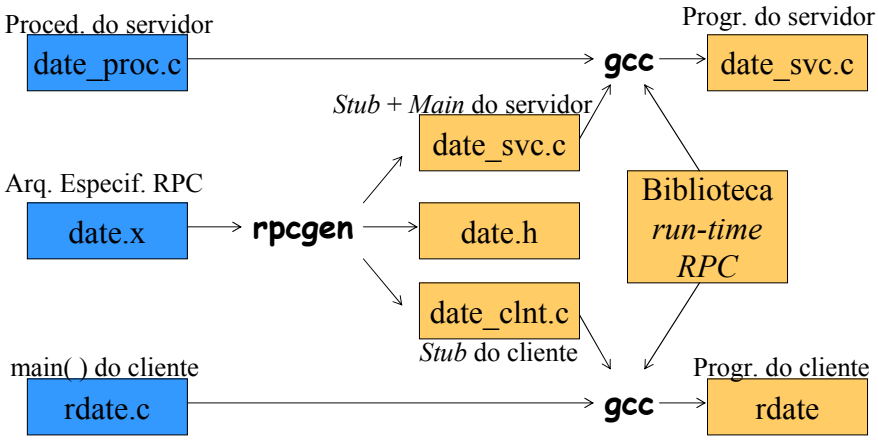
- O RPC da Sun consiste das seguintes partes:
  - **rpcgen**, um compilador que toma a definição de uma interface de procedimento remoto e gera os *stubs* do cliente e do servidor.
  - **XDR** (*eXternal Data Representation*), uma forma padrão de codificação de dados de maneira portável entre diferentes sistemas.
  - Uma biblioteca "*run-time*" para tratar de todos os detalhes.

Geração de um programa RPC

- Usaremos um exemplo simples para mostrar o que está envolvido na escrita de um cliente e de um servidor que utilizam RPC.
- Considere as seguintes funções chamadas por um cliente usando RPC:
  - **bin_date_1**: retorna a hora atual, como o número de segundos passados desde 00:00:00 GMT de 1o. de janeiro de 1970. Esta função não tem argumentos e retorna um inteiro longo (**long int**).
  - **str_date_1**, toma um valor inteiro longo da função anterior e o converte em uma string ASCII, adequada ao "consumo humano".

Geração de um programa RPC

Arquivos envolvidos na geração de um programa RPC da Sun



Geração de um programa RPC

- Para se gerar uma aplicação que use RPC, o que se tem a fazer é prover os 3 arquivos mostrados no lado esquerdo da figura anterior:
  - os procedimentos do servidor, que são chamados remotamente pelo cliente,
  - o arquivo de especificação RPC e
  - a função *main( )* do cliente
- O compilador **rpcgen** toma o arquivo **date.x** e gera dois novos arquivos ".c", com os *stubs* do cliente e do servidor e a função *main( )* do servidor, além de um arquivo cabeçalho, que é incluído em ambos os *stubs*.

Geração de um programa RPC

- O *stub* do cliente é compilado junto com a função *main( )* dele para gerar o programa executável do cliente:

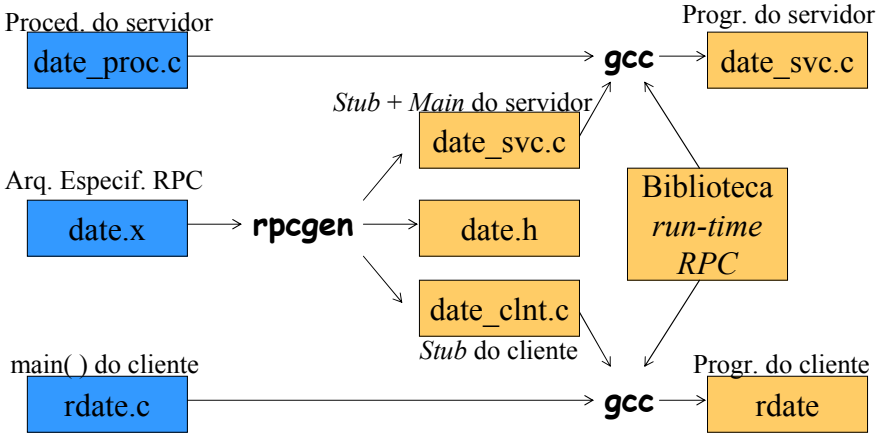
  gcc -o rdate rdate.c date_clnt.c -lrpclib

- De forma análoga, o programa servidor é gerado pela compilação do seu *stub* (que contém a função *main()* para o servidor) junto com o arquivo de procedimentos do servidor:

gcc -o date_svc date_proc.c date_svc.c -lrpclib

Geração de um programa RPC

Arquivos envolvidos na geração de um programa RPC da Sun



Arquivo de Especificação RPC

```
/* date.x - Specif of remote date and time service.
 * Define 2 procedures:
 * bin_date_1(): returns the binary time and date (no
 *               args)
 * str_date_1(): takes a binary time and returns a
 *               human-readable string.
*/
program DATE_PROG {
  version DATE_VERS {
    long   BIN_DATE(void) = 1;   /* proc number = 1 */
    string STR_DATE(long) = 2;   /* proc number = 2 */
  } = 1;                         /* version  num = 1 */
} = 0x31234567;           /* program num = 0x31234567 */
```


Arquivo de Especificação RPC

- Declaram-se todos os procedimentos e se especificam seus argumentos e seus valores de retorno.
- Atribuem-se:
  - um número de procedimento para cada função (1 e 2, no nosso caso),
  - um número de programa (0x31234567) e
  - um número de versão (1, no nosso caso).

Arquivo de Especificação RPC

- Números de procedimentos iniciam em 0 (zero).
- Todo programa e versão remotos devem definir o procedimento número 0 como o "**procedimento nulo**".
  - Ele não requer nenhum argumento e não retorna nada.
  - É automaticamente gerado pelo compilador **rpcgen**.
  - O seu uso é para permitir que um cliente, ao chamá-lo, possa verificar que um programa ou versão particular existe.
  - É útil também ao cliente no cálculo do "*round-trip time*".

Arquivo de Especificação RPC

- O compilador **rpcgen** gera os nomes reais dos procedimentos remotos convertendo os nomes BIN_DATE e STR_DATE para **bin_date_1** e **str_date_1**.

- O que ele faz na verdade é converter para letras minúsculas e acrescentar um caracter de sublinhar (*underscore*) e o número da versão.

Arquivo de Especificação RPC

- Nós definimos o procedimento BIN_DATE sem receber nenhum parâmetro (**void**) e retornando um inteiro longo como resultado (**long**).
- No caso do procedimento STR_DATE, ele toma um inteiro longo como argumento e retorna uma string como resultado.
- O RPC da Sun permite apenas um único argumento e um único resultado.
- Se forem desejados mais argumentos, devemos usar uma estrutura como parâmetro. De forma similar, para retornar mais de um valor, usa-se uma estrutura como valor de retorno.

Arquivo de Cabeçalho RPC

```
#define DATE_PROG    ((u_long) 0x31234567)
#define DATE_VERS    ((u_long) 1)

#define BIN_DATE     ((u_long) 1)
extern  long         *bin_date_1();

#define STR_DATE     ((u_long) 2)
extern  char         **str_date_1(long
  *bintime);
```

Arquivo de Cabeçalho RPC

- O arquivo de cabeçalho define o valor de retorno da função **bin_date_1** como um ponteiro para um inteiro longo e o valor de retorno da função **str_date_1** como um ponteiro para um ponteiro de caracter.
- O RPC da Sun especifica que o procedimento remoto **retorne o endereço do valor retornado**.
- Além disso, é passado ao procedimento remoto o **endereço de seu argumento**, quando este é chamado pelo *stub* do servidor.

Cliente RPC

```
/* rdate.c - client program for remote date service. */

#include    <stdio.h>
#include    <rpc/rpc.h> /* standard RPC include file */
#include    "date.h"    /* file generated by rpcgen  */

main(int argc, char *argv[]) {
  CLIENT *cl;     /* RPC handle */
  char *server;
  long *lresult;  /* return value from bin_date_1() */
  char **sresult; /* return value from str_date_1() */
```

Cliente RPC

```
if (argc != 2) {
    fprintf(stderr, "usage: %s hostname\n", argv[0]);
    exit(1);
}
server = argv[1];

/* Create the client "handle."     */
if ( (cl = clnt_create(server, DATE_PROG, DATE_VERS,
"udp")) == NULL) {
    /* Couldn't establish connection with server. */
    clnt_pcreateerror(server);
    exit(2);
}
```


Cliente RPC

```
  /* First call the remote procedure "bin_date". */
  if ( (lresult = bin_date_1(NULL, cl)) == NULL) {
      clnt_perror(cl, server);   exit(3);
  }
  printf("time on host %s = %ld\n", server, *lresult);

  /* Now call the remote procedure "str_date". */
  if ( (sresult = str_date_1(lresult, cl)) == NULL) {
      clnt_perror(cl, server);   exit(4);
  }
  printf("time on host %s = %s", server, *sresult);

  clnt_destroy(cl);        /* done with the handle */
  exit(0);
}
```

Cliente RPC

- Chamamos **clnt_create** para criar um "identificador" (*handle)* RPC para o programa e versão específico em um *host*.
- Especificamos também o protocolo UDP como o protocolo. Poderíamos ter especificado o argumento final como "**tcp**" para usar o TCP como protocolo de transporte.
- Uma vez que temos o identificador, estamos aptos a chamar qualquer dos procedimentos definidos para aquele programa e versão particular.

Cliente RPC

- Quando chamamos **bin_date_1**, o primeiro argumento da chamada é o mesmo definido no arquivo de especificação (**date.x**).
- O segundo argumento é o identificador do cliente.
- O valor de retorno é um ponteiro para um inteiro longo, como visto no arquivo de cabeçalho **date.h**, gerado pelo compilador **rpcgen**.
- Se um ponteiro para NULL é retornado da chamada de procedimento remoto, é porque ocorreu um erro.
- Quando chamamos a função **str_date_1**, o primeiro argumento é um ponteiro para um inteiro longo.

Servidor RPC

```
/* dateproc.c: remote procedure called by server stub*/
#include    <rpc/rpc.h> /* standard RPC include file */
#include    "date.h"    /* file generated by rpcgen  */

/* Return the binary date and time. */
long *bin_date_1() {
  static long     timeval;    /* must be static */
  long            time();     /* Unix function */

  timeval = time((long *) 0);
  return(&timeval);
}
```

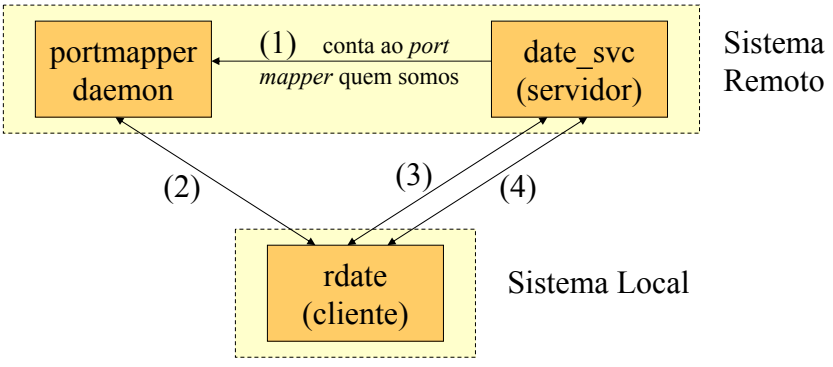

Servidor RPC

```
/* Convert a binary time and return a human readable
   string. */

char **str_date_1(long *bintime) {
  static char *ptr;           /* must be static */
  char        *ctime();       /* Unix function */

  ptr = ctime(bintime); /* convert to local time */

  return(&ptr);   /* return the address of pointer */
}
```

Servidor RPC

- A razão de comentarmos que os valores de retorno devem ser variáveis estáticas (**static**) é porque ambas as funções retornam os endereços destas variáveis.
- Se as variáveis não fossem estáticas (**static**) ou externas (**extern**), isto é, fossem variáveis automáticas, seus valores estariam indefinidos logo após a instrução **return** passasse o controle de volta ao *stub* do servidor, que é quem chama os procedimentos remotos.

Etapas envolvidas em chamadas RPC



1a. Etapa

- Quando disparamos o programa servidor em um sistema remoto, ele cria (**socket**) um *socket* UDP e associa (**bind**) uma porta qualquer a ele.
- Ele então chama uma função da biblioteca RPC, **svc_register**, que contacta o processo ***port mapper***, para registrar seu número de programa e versão.
- O ***port mapper*** mantém as informações do número do programa, número da versão e número da porta.
- O servidor então aguarda por requisições do cliente.
- Note que todas as ações desta etapa são feitas pelo *stub* do servidor, isto é, pela função *main()* gerada pelo compilador **rpcgen**.

2a. Etapa

- Ao iniciarmos o programa cliente, ele chama a função **clnt_create**.
- Esta chamada especifica o nome do sistema remoto, o número do programa, o número da versão e o protocolo.
- Esta função contacta o ***port mapper*** no sistema remoto para encontrar a porta UDP na qual o servidor está "escutando".

3a. Etapa

- O programa cliente chama a função **bin_date_1**.
- Esta função está definida no *stub* do cliente, que foi gerado pelo compilador **rpcgen**.
- Ela envia um datagrama para o servidor, usando o número de porta UDP da etapa anterior.
- A função **bin_date_1** então espera por uma resposta, retransmitindo a requisição uma determinada quantidade de vezes, se uma resposta não for recebida.
- O datagrama é recebido no sistema remoto pelo *stub* associado ao programa servidor.

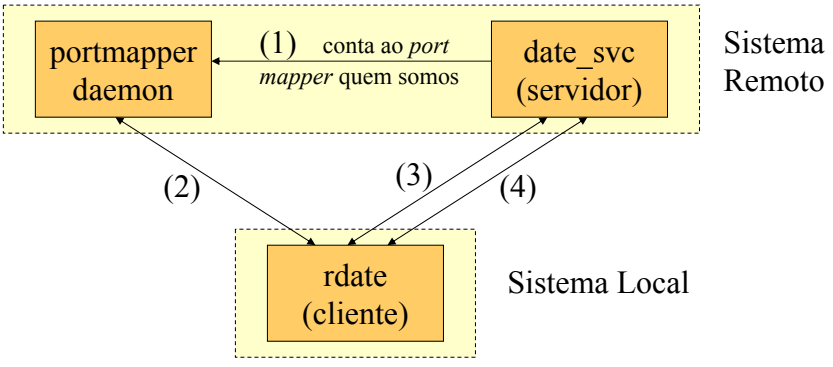
3a. Etapa

- Este *stub* (o do servidor) determina qual procedimento está sendo chamado e chama a função **bin_date_1**.
- Quando esta função retorna ao *stub* do servidor, o *stub* toma o valor de retorno, o converte para o formato padrão XDR e o empacota em um datagrama para transmiti-lo de volta ao cliente.
- Quando a resposta é recebida, o *stub* do cliente pega o valor trazido no datagrama, o converte como requerido e o retorna ao programa cliente.

4a. Etapa

- O cliente chama a função **str_date_1**, que realiza uma seqüência de operações similar às descritas na etapa anterior.
- Uma diferença é que o *stub* do cliente tem que passar um argumento para o procedimento remoto.
- Nota: O processo ***port mapper*** no sistema remoto só precisa ser contactado se um novo identificador de cliente é criado e isto só ocorre se um programa ou uma versão diferente forem chamados.

Etapas envolvidas em chamadas RPC



Exercícios

- "Sistemas operacionais modernos" Andrew S. TANENBAUM Prentice-Hall, 1995
  - Capítulo 10 Exercícios 7-15 (pág. 315)