

PSDDE

Programação em Sistemas Distribuídos

RPC

Remote Procedure Call

Implementação passando dados primitivos como parâmetros

Daniel Cintra Cugler
danielcugler@ifsp.edu.br

Conteúdo dos slides retirados do livro:

TANENBAUM, A. S; STEEN, M. V Sistemas Distribuídos: Princípios e Paradigmas. 2 Ed, São Paulo: Ed. Prentice Hall, 2007.

O que veremos...

- Implementação de um exemplo de RPC passando como parâmetro dados primitivos
 - RPCGEN
 - IDL
 - RPCL

Introdução

- Implementaremos um exemplo de chamada de procedimento remoto através de programa implementado na linguagem C, que rodará no sistema operacional Linux.
- Disponibilizaremos dois métodos que podem ser invocados remotamente
 - `int soma(int)`: recebe um inteiro ***n*** e retorna ***n+1***
 - `int subtrai(int)`: recebe um inteiro ***n*** e retorna ***n-1***

Implementação utilizada

- Existem muitas implementações de RPC. Utilizaremos a implementação disponibilizada na biblioteca "libc6-dev" do Linux.
- Esta biblioteca possui a implementação da Sun Open Network Computing RPC (referenciada na literatura como *"Sun ONC RPC"* ou apenas *"Sun RPC"*)

Preparando o ambiente para criar e rodar o programa

- `sudo apt-get install rpcbind`
- `sudo apt-get install build-essential`

RPCGEN

- Utilizaremos o programa RPCGEN para gerar os arquivos necessários para nosso programa.
- RPCGEN é uma ferramenta que gera automaticamente código que implementa o protocolo RPC. O arquivo de entrada para o RPCGEN é um arquivo com extensão .x, escrito em RPC IDL (veremos a seguir)

Criando o programa IDL

- Para definir os tipos de dados que serão trocados e os métodos disponibilizados em nossa aplicação de RPC, utilizaremos uma linguagem de descrição de interface (Interface Description Language IDL).
- A especificação IDL foi criada para permitir a descrição de interfaces de forma não vinculada a outras linguagens de programação. Isto permite a comunicação entre diferentes aplicações que são desenvolvidas em linguagens diferentes.
- No nosso caso, utilizaremos a RPC IDL, conhecida como RPCL (Remote Procedure Call Language).

Criando o programa RPCL

- RPCL é similar à linguagem C
- O primeiro passo é criar um arquivo .x, que contém a assinatura dos métodos e a definição dos dados que serão acessados via RPC.
- O arquivo .x será a entrada para o compilador RPCGEN.

Mais informações sobre RPCL:

https://docs.oracle.com/cd/E23824_01/html/821-1671/rpcproto-24229.html#scrolltoc

Criar calculadora.x em RPCL

```
program NOME_DO_PROGRAMA {  
    version VERSAO_DO_PROGRAMA {  
        int soma(int) = 1;  
        int subtrai(int) = 2;  
    } = 1;  
} = 0x3012227;
```

Criar calculadora.x em RPCL

```
program NOME DO PROGRAMA {  
    version VERSAO DO PROGRAMA {  
        int soma(int) = 1;  
        int subtrai(int) = 2;  
    } = 1;  
} = 0x3012227;
```

Qualquer nome que o programador deseje utilizar para identificar seu programa e sua versão

Criar calculadora.x em RPCL

```
program NOME_DO_PROGRAMA {  
    version VERSAO_DO_PROGRAMA {  
        int soma(int) = 1;  
        int subtrai(int) = 2;  
    } = 1;  
} = 0x3012227;
```

Os IDs são utilizados para identificar diferentes programas e suas diferentes versões de implementação. Por ex., é possível ter mais de uma tag “version VERSÃO_DO_PROGRAMA”, cada uma com seu ID e conjunto de métodos. Então, é possível clientes invocarem diferentes versões de implementação dos métodos de um programa P (não entraremos nesses detalhes)

Criar calculadora.x em RPCL

```
program NOME_DO_PROGRAMA {  
    version VERSAO_DO_PROGRAMA {  
        int soma(int) = 1;  
        int subtrai(int) = 2;  
    } = 1;  
} = 0x3012227;
```

Métodos que serão disponibilizados para acesso via RPC

Criar calculadora.x em RPCL

```
program NOME_DO_PROGRAMA {  
    version VERSAO_DO_PROGRAMA {  
        int soma(int) = 1;  
        int subtrai(int) = 2;  
    } = 1;  
} = 0x3012227;
```

É permitido passar apenas um parâmetro por vez.

*Se for desejado passar mais de um parâmetro, deveremos utilizar **structs** (veremos posteriormente)*

Criar calculadora.x em RPCL

```
program NOME_DO_PROGRAMA {  
    version VERSAO_DO_PROGRAMA {  
        int soma(int) = 1;  
        int subtrai(int) = 2;  
    } = 1;  
} = 0x3012227;
```

IDs para cada método. O valor deve ser sequencial, e definido pelo programador

Rodando RPCGEN

- Em seguida, rodar na linha de comando:
 - `rpcgen calculadora.x`
- A execução do *rpcgen* vai gerar, ***neste exemplo***, 3 arquivos:
 - **`calculadora.h`** : contêm a definição dos métodos a serem implementados no cliente e servidor - comece lendo este arquivo!
 - **`calculadora_clnt.c`**: contém a implementação do *stub* do cliente
 - **`calculadora_svc.c`**: contém a implementação do *stub* do servidor

Rodando RPCGEN

- Em seguida, rodar na linha de comando:
 - `rpcgen calculadora.x`

- A execução do `rpcgen` gera os seguintes arquivos:
 - **calcula.xdr**: arquivo de definição de tipos (XDR) a serem utilizados.
 - **calcula_client.c**: código do cliente.
 - **calcula_server.c**: código do *stub* do servidor.

Veremos nas próximas aulas que o *rpcgen* pode gerar até 4 arquivos (arquivo XDR, não gerado em nosso exemplo).

Rodando RPCGEN

- Para o cliente, RPCGEN gera automaticamente as funções para lidar com RPC (*no arquivo calculadora_clnt.c*), mas você tem que implementar o *main()*.
- Para o servidor, RPCGEN gera automaticamente o *main()* (*no arquivo calculadora_svc.c*), mas você tem que implementar as funções que serão disponibilizadas para acesso via RPC.
- Portanto, será necessário criar mais dois arquivos .c.
 - (a) *cliente.c*, que vai implementar as funções desejadas para o cliente;
 - (b) *servidor.c*, que vai implementar os métodos disponibilizados para acesso via RPC.

calculadora.h

- Analisando o conteúdo de calculadora.h temos, entre outras coisas, a definição dos métodos SOMA e SUBTRAI que definimos em calculadora.x.

```
#define SOMA 1
extern int * soma_1(int *, CLIENT *);
extern int * soma_1_svc(int *, struct svc_req *);
#define SUBTRAI 2
extern int * subtrai_1(int *, CLIENT *);
extern int * subtrai_1_svc(int *, struct svc_req *);
```

calculadora.h

Notem que os retornos das funções e os envios de parâmetros são SEMPRE feitos através de ponteiros!

```
#define SOMA 1
extern int * soma_1(int *, CLIENT *);
extern int * soma_1_svc(int *, struct svc_req *);
#define SUBTRAI 2
extern int * subtrai_1(int *, CLIENT *);
extern int * subtrai_1_svc(int *, struct svc_req *);
```

Analizando os métodos em calculadora.h

- ***soma_1*** e ***subtrai_1*** (note que o "1" é a versão do programa) são os *stubs* do cliente. Estes *stubs* ***já foram implementados*** pelo RPCGEN e contêm as funções que o cliente pode chamar via RPC.
- ***soma_1_svc*** e ***subtrai_1_svc*** são protótipos para as funções do servidor que ***ainda teremos que implementar***.
- O tipo de dado "***CLIENT***" especifica uma conexão para um servidor RPC (*próximo slide*)
- A estrutura "***svc_req***" é utilizada para enviar ao servidor algumas informações sobre a requisição de entrada (*não exploraremos isso*)

Criar cliente.c

página 1 de 2

Utilize o editor de sua preferência, por ex., gedit, vi, Dev C++.

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "calculadora.h"

int main (int argc, char *argv[]) {

    CLIENT *meu_cliente;
    int *resposta_servidor;
    char *ip_do_servidor = argv[1];

    if (argc != 2) {
        printf("Sintaxe incorreta (Utilize: ./cliente <IP_DO_SERVIDOR>)\n");
        exit(1);
    }

    meu_cliente =
    clnt_create(ip_do_servidor, NOME_DO_PROGRAMA , VERSAO_DO_PROGRAMA , "tcp");

    if (meu_cliente == NULL) {
        clnt_pcreateerror(ip_do_servidor);
        exit(1);
    }
}
```

Criar cliente.c

página 2 de 2

```
int n;
printf("Digite n: ");
scanf("%d", &n);

int operacao;
printf("Escolha a operação (0)Subtrai um e (1) Soma um: ");
scanf("%d", &operacao);

switch(operacao) {
    case(0):
        resposta_servidor = subtrai_1(&n, meu_cliente);
        break;
    case(1):
        resposta_servidor = soma_1(&n, meu_cliente);
}

if (resposta_servidor == NULL) {
    clnt_perror(meu_cliente, argv[1]);
    exit(1);
}

printf("A resposta do servidor foi=%d\n", *resposta_servidor);
return 0;
```

```
}
```

Criar servidor.c

```
#include <stdio.h>
#include "calculadora.h"

int * soma_1_svc(int *n, struct svc_req *attr) {
    printf("Função SOMA - Recebido número: %d\n", *n);

    *n = *n + 1;
    return(n);
}

int * subtrai_1_svc(int *n, struct svc_req *attr) {
    printf("Função SUBTRAI - Recebido número: %d\n", *n);

    *n = *n - 1;
    return(n);
}
```

Criar servidor.c

```
#include <stdio.h>
#include "calculadora.h"

int * soma_1_svc(int *n, struct svc_req *attr) {
    printf("Função SOMA - Recebido número: %d\n", *n);

    *n = *n + 1;
    return(n);
}
```

int

Se fosse desejado não reutilizar a variável *n*, recebida como parâmetro, seria necessário criar uma variável global para armazenar o valor de resposta.

É necessário que seja global, pois quando é finalizada a execução da função, todas as variáveis locais deixam de existir – e o endereço que foi retornado vai ser de uma variável que não existe mais.

}

Criar servidor.c

```
#include <stdio.h>
#include "calculadora.h"
```

```
int * soma_1_svc(int *n, struct svc_req *attr) {
    printf("Função SOMA - Recebido número: %d\n", *n);
```

ATENÇÃO!!

No programa do servidor, toda linha que utiliza o printf **deve** terminar com **\n**, caso contrário, o texto ficará *preso* no buffer e resultará em comportamento inadequado.

```
    struct svc_req *attr) {
        do número: %d\n", *n);
```

```
}
```

Compilando os fontes

- Pronto!
- Acabamos de gerar todos os arquivos necessários para nossa aplicação RPC
- Agora, precisamos compilar todos os programas. Para isso, geraremos um script.

Criar compilar.sh

```
#!/bin/bash
```

```
gcc -c calculadora_clnt.c
```

```
gcc -c calculadora_svc.c
```

```
gcc -c servidor.c
```

```
gcc -c cliente.c
```

```
gcc -o client calculadora_clnt.o cliente.o
```

```
gcc -o server calculadora_svc.o servidor.o
```

Compilando os fontes

- Dar permissão de execução para o script

```
chmod +x compilar.sh
```

- Executar no console `./compilar.sh`. Depois disso, o que interessa são apenas os binários gerados ("client" e "server"). Se desejado, copie esses arquivos para pastas separadas.

Executando cliente e servidor no mesmo computador

Abra dois consoles no mesmo computador. Em um execute primeiramente `"./server"`. No outro, execute `"./client localhost"`. Insira os dados solicitados e veja o resultado.

Executando cliente e servidor em computadores diferentes

Se desejar rodar o servidor em outro computador, copie o arquivo "server" para outro computador e execute-o ("./server"). Em seguida, execute o cliente passando como parâmetro o IP do servidor. Por exemplo: `"./client 192.168.0.2"`.

O que foi visto...

- Implementação de um exemplo de RPC passando como parâmetro dados primitivos
 - RPCGEN
 - IDL
 - RPCL

Exercício 1

Crie um programa que, via RPC, disponibilize uma função chamada *fatorial*, que receberá como parâmetro um valor do tipo int e retornará o fatorial desse número. O fatorial deve ser calculado recursivamente no servidor.

Exercício 2

Crie um programa que, via RPC, disponibilize uma função chamada *round_trip_time*, que não recebe nem retorna parâmetros. Essa função vai ser utilizada pelo cliente apenas para calcular o round trip time (RTT) em milissegundos.

O resultado esperado a ser retornado por este programa é a quantidade de tempo (em milissegundos) que foi gasto desde a chamada à função no servidor e o recebimento da resposta.

Exercício 3

Dado o conteúdo de um arquivo .x escrito em RPCL, implemente um programa que acesse o método descrito em um servidor remoto, a ser passado pelo professor *(ainda não vimos como passar strings como parâmetro – analise com calma o arquivo .h gerado automaticamente antes de iniciar a implementação)*.

```
program NOME_DO_PROGRAMA {  
    version VERSAO_DO_PROGRAMA {  
        void postar_no_mural(string) = 1;  
    } = 1;  
} = 0x3012231;
```