Universidade de São Paulo
Instituto de Ciências Matemáticas de São Carlos
Departamento de Ciências de Computação e Estatística

Aplicações de RPC no Ambiente SUN-OS

Márcio Augusto de Souza
Paulo Sérgio Lopes de Souza
Simone do Rocio Senger de Souza
Marcos José Santana
Regina Helena C. Santana

março/1995

Conteúdo

1. Introdução

Chamadas de procedimentos constituem um dos métodos mais utilizados para a transferência do fluxo de controle dentro de um programa, permitindo assim, sua modularidade. A passagem de parâmetros e o retorno de valores também são comumente utilizados, tornando possível a comunicação entre os procedimentos através de variáveis locais. Usualmente, a chamada de procedimentos é feita dentro do mesmo programa e na mesma máquina e por essa razão é normalmente referenciada como Chamada de Procedimento Local (*Local Procedure Call - LPC*) [STE90].

Chamadas de Procedimentos Remotos (*Remote Procedure Call - RPC*), baseiam-se no mesmo conceito de LPC's, diferindo no fato de que as chamadas são feitas entre programas distintos, não necessariamente situados na mesma máquina. A finalidade do mecanismo de RPC é, portanto, fornecer ao usuário um método de comunicação entre processos (*Interprocess Communication - IPC*) que possua uma estrutura familiar aos programadores e que esconda ao máximo os detalhes de mais baixo nível relacionados com a comunicação, ou seja, permitindo que uma chamada remota seja feita de maneira similar a uma chamada local.

Este trabalho tem como finalidade discutir a implementação de aplicações baseadas no mecanismo de RPC desenvolvido pela *SUN Microsystems, Inc* [SUN88], dentro do ambiente SUN-OS, chamado "Sun RPC System".

Na próxima seção discutem-se aspectos relacionados com o mecanismo Sun RPC, destacando-se as suas características fundamentais. Em seguida são apresentados alguns projetos desenvolvidos utilizando-se desta implementação RPC, servindo como exemplos ilustrativos da utilização dos conceitos apresentados.

Finalmente são discutidas as conclusões obtidas a partir da utilização desses conceitos.

2 . RPC na SUN

O mecanismo de RPC utilizado para a elaboração deste trabalho, segue o modelo definido pela SUN, conhecido como "Sun RPC System". Esse modelo define um grande número de primitivas de controle para o mecanismo RPC, possibilitando ao programador o controle do processo de comunicação em todos os seus detalhes. Todas essas primitivas são oferecidas dentro de um conjunto de bibliotecas, disponíveis no diretório "/usr/include/rpc".

A utilização dessas primitivas, que podem ser chamadas de "baixo nível", trata-se de uma tarefa complicada e demorada. Para isso, a SUN oferece um aplicativo, denominado **rpcgen**, que é um gerador automático de RPC (uma espécie de compilador), isto é, a partir de uma descrição de "alto nível" da interface cliente-servidor, o **rpcgen** cuida de construir toda a estrutura de controle necessária para a comunicação entre cliente-servidor, utilizando as primitivas definidas.

Ao programador é reservada a tarefa de implementar os programas cliente e servidor, e definir os procedimentos e respectivos parâmetros que formarão a interface RPC.

Para a definição da interface RPC, o programador criará um arquivo de especificação, que é escrito utilizando uma linguagem especialmente definida para esse fim (*linguagem RPC)*. Nesse arquivo, são definidas as estruturas que serão transmitidas, ou como parâmetros de entrada ou como resultados das operações remotas, e os procedimentos que definirão essas operações remotas. Esse arquivo de especificação contém apenas uma lista dos procedimentos remotos, sendo que a implementação desses procedimentos é feita no programa servidor (que compreende um conjunto de rotinas remotas).

O aplicativo **rpcgen**, através desse arquivo de especificação, gera os "stubs" e uma pequena biblioteca que deverá ser incluída no cabeçalho dos programas servidor e cliente.

Resumidamente, o processo de geração de uma interface RPC segue o diagrama apresentado na figura 2.1 [STE90].

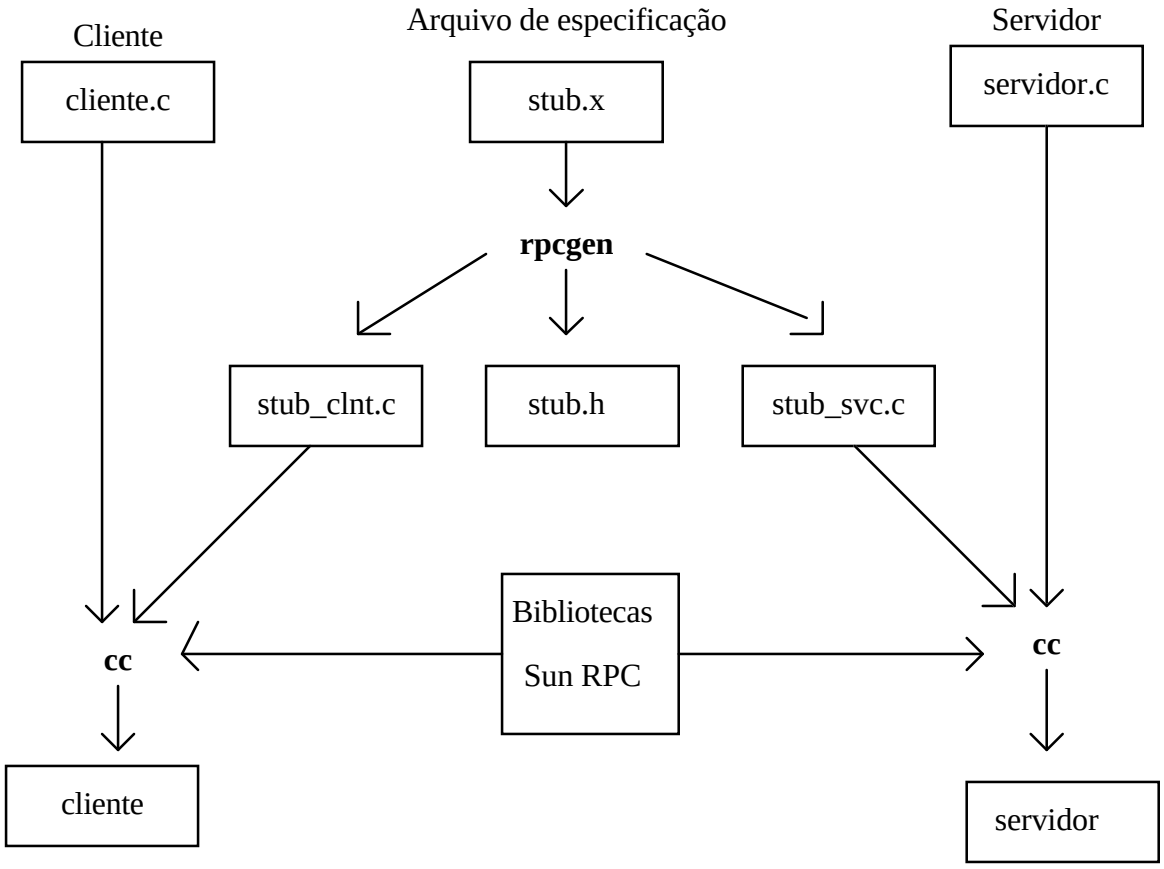

Figura 2.1 - Processo de geração da interface SUN RPC.

O **rpcgen** é aplicado ao arquivo de especificação, aqui denominado *stub.x*. Os arquivos stub_*.c gerados pelo **rpcgen** são os "stubs", que especificam a interface entre cliente e servidor.

Um quarto arquivo pode ser gerado pelo **rpcgen** dependendo do tipo de parâmetros definidos para os procedimentos remotos. Para a transmissão de estruturas de dados (como **structs** ou **unions**) entre programas remotos, é definido um padrão que explicita o formato com a qual a estrutura será transmitida, que é denominado **XDR** *(**EX**ternal **D**ata **R**epresentation)*. Nesse caso, é gerado um arquivo denominado stub_xdr.c que deve ser incluído em ambos os programas cliente e servidor. No caso de transmissão de tipos simples, como inteiros e cadeias de caracteres (*strings)*, esse arquivo não é gerado.

O **rpcgen** cuida também da parte do registro desse serviço no "portmapper", que é um mapeador de portos lógicos em uma determinada máquina, sendo que clientes interessados em determinados serviços fazem acessos aos portos relacionados a esses serviços.

2.1. Geração de RPC's

Para a geração dos códigos executáveis dos programas cliente e servidor, devem-se seguir os seguintes passos:

- utilização do **rpcgen**, indicando o nome do arquivo de especificação.
- compilação e montagem de todos os programas gerados, incluindo a biblioteca RPC (rpclib).

Por exemplo, considere o arquivo "script", gerado para a compilação dos programas descritos na figura 2.1:

rpcgen stub.x
cc -o cliente cliente.c stub_clnt.c stub_xdr.c -Irpclib
cc -o servidor servidor.c stub_svc.c stub_xdr.c -Irpclib

3. Exemplos de Utilização de RPC

3.1. Um servidor de arquivos simples

De modo simples, um servidor de arquivos é uma estação que tem a função de armazenar arquivos de maneira que estes possam ser acessados por quaisquer estações ligadas a ele via uma rede. No nosso caso, esse acesso é feito por intermédio de mecanismos RPC, e uma demonstração simplificada do mecanismo pode ser entendida pela figura 3.1.

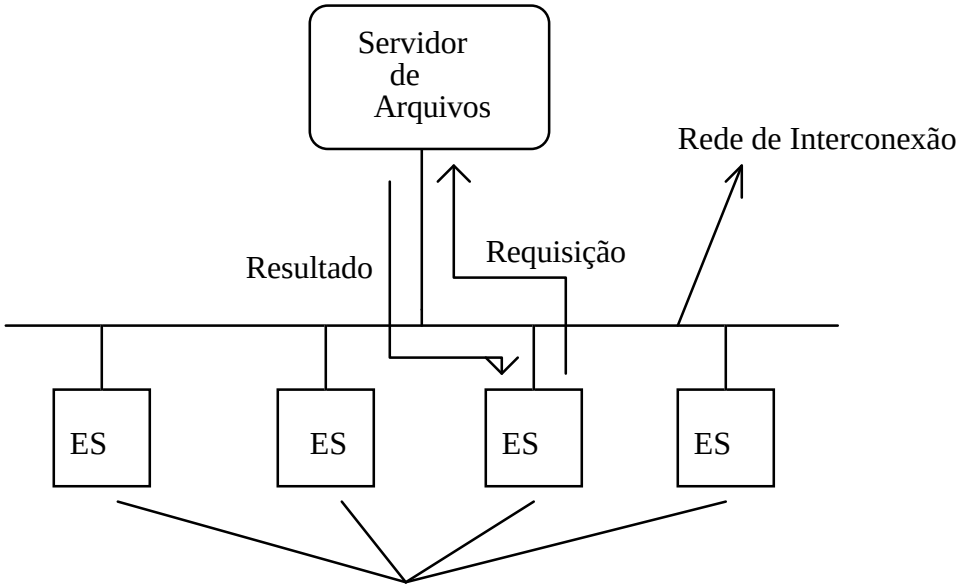

Figura 3.1 Esquema simplificado de um servidor de arquivos.

O cliente remete uma requisição de serviço, que nesse caso trata-se de uma operação qualquer em um arquivo, incluindo nesse pedido os parâmetros necessários para se concretizar a operação remota. O cliente então é bloqueado, até que receba o resultado por parte do serviço de arquivos, que pode ser, entre outros, ou uma determinada informação ou um bit indicando se a operação teve sucesso.

Este projeto trata-se de uma simulação simplificada de um servidor de arquivos, e para tal aplicação, foram definidas as seguintes diretivas:

- Um diretório, chamado "server", foi criado para conter este servidor; o qual representa o diretório "root" do sistema simulado;
- Nesse local se localiza o programa servidor, e tentativas de se mudar para diretórios de níveis hierárquicos inferiores via primitivas remotas não seriam permitidas. Portanto, pelo contexto gerado pelo programa servidor, o sistema de arquivos tem inicio a partir deste diretório;
- Não são consideradas situações de compartilhamento de arquivos.

Definidos os parâmetros básicos da arquitetura desta aplicação, foram definidos as primitivas que são oferecidas remotamente pelo servidor:

* Criação de arquivos;
* Deleção de arquivos;
* Escrita de dados em um arquivo;
* Escrita de dados no final de um arquivo;
* Cálculo do tamanho de um arquivo;
* Leitura de um arquivo;
* Criação de um diretório;
* Cálculo do tamanho de todos os arquivos sob um determinado diretório (toda a árvore abaixo deste diretório é percorrida);
* Deleção de um diretório;
* Cálculo da posição atual na árvore de diretórios (comando *pwd*);
* Troca de diretório corrente (comando *chdir*);
* Cópia de arquivos.
* Informações gerais sobre um determinado arquivo.

Várias outras primitivas foram planejadas, as quais não foram implementadas por motivos de tempo, as quais, porém, podem ser facilmente acrescentadas como por exemplo: comando *mv* (move arquivos entre diretórios) e comando *chown* (muda as permissões de arquivos).

O programa servidor oferece um conjunto de primitivas que podem ser utilizadas por processos clientes de diversos usuários, formando uma espécie de biblioteca para acesso a esse servidor remoto. Para a simulação e teste dessas operações, foi necessário criar um mecanismo que permitisse a avaliação dos resultados conseguidos. Para isso foi necessário implementar um programa cliente que acessasse

essas primitivas de acordo com a vontade do usuário. Para isso, foi definido um programa que recebesse as requisições dos usuários através da linha de comando. Então, o usuário deve teclar o nome do programa cliente e como argumentos deve digitar na frente deste a sequência de caracteres que formam a chamada da primitiva. Então, por exemplo, se o cliente pretende fazer uma operação de criação de um arquivo, então ele deve digitar a seguinte linha de comando:

cliente *cria arquivo* onde:

cliente: nome do programa que requisitas operações ao servidor;
cria: nome dado a primitiva de criação de arquivos;
arquivo: nome do arquivo que se quer criar.

O programa **cliente**, oferece também um menu explicativo de todas as primitivas fornecidas e também uma descrição da sintaxe de cada uma delas.

Uma breve descrição da sintaxe de cada comando é dada a seguir:

cria <arq> <arq>: arquivo	Cria um arquivo
escreve <arq> <pos> <buf> <arq>: arquivo <pos>: posição inicial <buf>: cadeia de caracteres	Escreve uma cadeia de caracteres em uma determinada posição de um arquivo
deleta <arq> <arq> : arquivo	Deleta um arquivo
anexa <arq> <buf> <arq>: arquivo <buffer>: cadeia de caracteres	Anexa uma cadeia de caracteres no final de uma arquivo
le <arq> <pos> <tam> <arq>: arquivo <pos>: posição inicial <tam>: tamanho da cadeia	Lê uma cadeia de caracteres de uma determinada posição de um arquivo

mkdir <dir> <dir>: diretório	Cria um diretório
rmdir <dir> <dir>: diretório	Remove um diretório
pwd	Retorna o diretório corrente
chdir <dir> <dir>: diretório	Muda o diretório corrente
copia <arq1> <arq2> <arq1>: arquivo origem <arq2>: arquivo destino	Copia um arquivo
uso <dir> <dir>: diretório	Retorna a quantidade de bytes total de todos os arquivos abaixo de determinado diretório
informa <arq> <arq>: arquivo	Retorna informações sobre um arquivo

tabela 3.1: tabela de primitivas.

Para a implementação destas primitivas para arquivos, foram utilizados apenas comandos de programação C, o qual apresenta bibliotecas extremamente completas para se tratar tais operações. A referência [SAL87] apresenta uma ótima descrição sobre operações em arquivos.

3.1.1. Implementação

```
/*****************************************************************************
    PROGRAMA: Cliente.c

    FUNCAO: Acessa serviços de um servidor de arquivos remotos via RPC

*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <rpc/rpc.h>
#include <pwd.h>
#include "stub.h"
```

```
/* Alocacao dinamica de memoria */
#define aloca(num,x) (x *) calloc(num,sizeof(x))

 CLIENT *cliente;      /* "handle" para o RPC */
 char   *servidor;     /* identificacao do servidor */
 aux *auxiliar;        /* Estrutura para resultados de operacoes remotas */

/*  Cada um dos seguintes procedimentos e responsavel por ativar os pro-
cedimentos remotos. A descricao dos parametros enviados e dos resultados
retornados pelos rpc serao especificados no programa servidor */


/*************************************************************************
       PROCEDIMENTO: criar_arquivo()

       Cria um arquivo.

        PARAMETROS:
        nome: nome do arquivo a ser criado

*************************************************************************/

void criar_arquivo(nome)
char *nome;
{

  int *fd = NULL;

  fd = aloca(1,int);
  if ( ( fd = cria_arquivo_1(&nome, cliente)) == NULL)
    {
      clnt_perror(cliente, servidor);
      exit(1);
    }
  if (*fd == -1)
    printf(" Fracasso!!! Erro na criacao do arquivo \n");
    else printf(" Sucesso!!! Arquivo %s criado \n", nome);
}


/*************************************************************************
       PROCEDIMENTO: deletar_arquivo()

       Deleta um arquivo.

        PARAMETROS:
        nome: nome do arquivo a ser deletado
```

```
*****************************************************************************/

void deletar_arquivo(nome)
char *nome;
{

  int *resultado = NULL;

  resultado = aloca(1,int);
  if ( ( resultado = deleta_arquivo_1(&nome, cliente)) == NULL)
    {
     clnt_perror(cliente,servidor);
     exit(1);
    }
  if (*resultado == -1)
    printf("Fracasso!!! Arquivo nao pode ser deletado \n");
    else printf(" Sucesso!!! Arquivo %s deletado \n", nome);
}


/*****************************************************************************
        PROCEDIMENTO: escrever_arquivo()

        Escreve uma sequencia de bytes em um arquivo.

        PARAMETROS:
        nome: nome do arquivo a ser atualizado
        comeco: posicao inicial
        dados: string de caracteres a ser escrita

*****************************************************************************/

void escrever_arquivo(nome,comeco,dados)
char *nome;
int comeco;
char *dados;
{

  int *resultado = NULL;

  resultado = aloca(1,int);
  auxiliar = aloca(1,aux);
  strcpy(auxiliar->nome,nome);
  strcpy(auxiliar->buffer,dados);
  auxiliar->inicio = comeco;
  auxiliar->tam_ou_erro = strlen(dados);
  if ( ( resultado = escreve_arquivo_1(auxiliar, cliente)) == NULL)
    {
     clnt_perror(cliente,servidor);
```

```
     exit(1);
    }
  if (*resultado == -1)
    printf("Fracasso!!! Operacao de escrita falha \n");
    else printf(" Sucesso!!! Arquivo %s modificado \n", auxiliar->nome);
}


/*****************************************************************************
       PROCEDIMENTO: append_arquivo()

       Anexa uma sequencia de bytes no final de um arquivo.

        PARAMETROS:
        nome: nome do arquivo a ser atualizado
        dados: string de caracteres a ser escrita

*****************************************************************************/

void append_arquivo(nome,dados)
char *nome;
char *dados;
{

  int *resultado = NULL;

  resultado = aloca(1,int);
  auxiliar = aloca(1,aux);
  strcpy(auxiliar->nome,nome);
  strcpy(auxiliar->buffer,dados);
  if ( ( resultado = append_arquivo_1(auxiliar, cliente)) == NULL)
    {
     clnt_perror(cliente,servidor);
     exit(1);
    }
  if (*resultado == -1)
    printf("Fracasso!!! Operacao de append falha \n");
    else printf(" Sucesso!!! Arquivo %s modificado \n", auxiliar->nome);
}


/*****************************************************************************
       PROCEDIMENTO: tamanho_arquivo()

       Retorna o tamanho de um arquivo.

        PARAMETROS:
        nome: nome do arquivo
```

```
**************************************************************************/

void tamanho_arquivo(nome)
char *nome;
{

  long *resultado = NULL;

  resultado = aloca(1,long);
  if ( ( resultado = tamanho_arquivo_1(&nome, cliente)) == NULL)
    {
     clnt_perror(cliente,servidor);
     exit(1);
    }
  if (*resultado == -1)
    printf("Fracasso!!! Operacao tamanho falha \n");
    else printf(" Sucesso!!! Arquivo %s --> %ld bytes\n", nome, *resultado);
}


/**************************************************************************
        PROCEDIMENTO: ler_arquivo()

        Le uma sequencia de bytes em um arquivo

        PARAMETROS:
        nome: nome do arquivo a ser lido
        comeco: posicao inicial
        tamanho: quantidade de bytes a ser lida

**************************************************************************/

void ler_arquivo(nome,comeco,tamanho)
char *nome;
int comeco;
int tamanho;
{

  aux *auxiliar2 = NULL;    /* resultado da operacao */

  auxiliar = aloca(1,aux);
  auxiliar2 = aloca(1,aux);
  strcpy(auxiliar->nome,nome);
  auxiliar->inicio = comeco;
  auxiliar->tam_ou_erro = tamanho;
  if ( ( auxiliar2 = le_arquivo_1(auxiliar,  cliente)) == NULL)
    {
     clnt_perror(cliente,servidor);
     exit(1);
```

```
    }
  if ((auxiliar2->tam_ou_erro == -1) || (auxiliar2->inicio < auxiliar->tam_ou_erro))
    printf("Fracasso!!! Operacao de leitura falha \n");
    else printf(" Sucesso!!! Arquivo %s lido: %s \n", auxiliar->nome, auxiliar2->buffer);
}


/*****************************************************************************
       PROCEDIMENTO: cria_diretorio()

        Cria um diretorio.

        PARAMETROS:
        nome: nome do diretorio a ser criado

*****************************************************************************/

void cria_diretorio(nome)
char *nome;
{

  int *dd;

  dd = aloca(1,int);
  if ( ( dd = cria_diretorio_1(&nome, cliente)) == NULL)
     {
       clnt_perror(cliente,servidor);
       exit(1);
     }
  if (*dd == -1)
    printf("Fracasso!!! Diretorio nao criado \n");
    else printf("Sucesso!!! Diretorio %s criado \n", nome);
}


/*****************************************************************************
        PROCEDIMENTO: deleta_diretorio()

        Deleta um diretorio.

        PARAMETROS:
        nome: nome do diretorio a ser deletado

*****************************************************************************/

void deleta_diretorio(nome)
char *nome;
{
```

```
  int *dd;                /* resultado da operacao */

  dd = aloca(1,int);
  if ( ( dd = deleta_diretorio_1(&nome, cliente)) == NULL)
     {
       clnt_perror(cliente,servidor);
       exit(1);
     }
  if (*dd == -1)
    printf("Fracasso!!! Diretorio nao deletado \n");
    else printf("Sucesso!!! Diretorio %s removido \n", nome);
}


/*****************************************************************************
        PROCEDIMENTO: meu_pwd()

        Retorna o diretorio corrente.

        PARAMETROS:
        sem parametros

*****************************************************************************/

void meu_pwd()
{

  char **resultado;     /* resultado da operacao */

  if ( ( resultado = meu_pwd_1(NULL, cliente)) == NULL)
     {
       clnt_perror(cliente,servidor);
       exit(1);
     }
  if (*resultado == NULL)
    printf("Fracasso!!! Operacao pwd falha \n");
    else printf("Sucesso!!! Diretorio e %s \n", *resultado);
}


/*****************************************************************************
        PROCEDIMENTO: meu_chdir()

        Muda o diretorio corrente.

        PARAMETROS:
        nome: nome do novo diretorio corrente

*****************************************************************************/
```

```
void meu_chdir(nome)
char *nome;
{

  int *resultado;

  resultado = aloca(1,int);
  if ( ( resultado = meu_chdir_1(&nome, cliente)) == NULL)
     {
       clnt_perror(cliente,servidor);
       exit(1);
     }
  if (*resultado == -1)
    printf("Fracasso!!! Operacao de chdir falha \n");
    else printf("Sucesso!!! Diretorio mudado \n");
}


/*************************************************************************
        PROCEDIMENTO: copia_arquivo()

        Copia um arquivo

        PARAMETROS:
        nome1: nome do arquivo origem
        nome2: nome do arquivo destino

*************************************************************************/

void copia_arquivo(nome1,nome2)
char *nome1;
char *nome2;
{

  int *resultado;
  aux *auxiliar;

  resultado = aloca(1,int);
  auxiliar = aloca(1,aux);
  strcpy(auxiliar->nome,nome1);
  strcpy(auxiliar->buffer,nome2);
  if ( (resultado = copie_arquivo_1(auxiliar, cliente)) == NULL)
    {
      clnt_perror(cliente,servidor);
      exit(1);
    }
  if (*resultado == -1)
    printf("Fracasso!!! Arquivo nao copiado \n");
```

```
   else printf("Sucesso!!! Arquivo copiado \n");
}


/*****************************************************************************
        PROCEDIMENTO: disco_usado()

        Retorna a quantidade de bytes ocupado por um determinado diretorio. A arvore contida
        abaixo deste diretorio e completamente varrida.

        PARAMETROS:
        nome: nome do diretorio percorrido

*****************************************************************************/

void disco_usado(nome)
char *nome;
{

  long *resultado = NULL;

  resultado = aloca(1,long);
  if ( ( resultado = percorre_diretorio_1(&nome, cliente)) == NULL)
     {
       clnt_perror(cliente,servidor);
       exit(1);
     }
  if (*resultado == -1)
    printf("Fracasso!!! Operacao falhou \n");
    else printf("Sucesso!!! Tamanho e -> %ld bytes\n", *resultado);
}


/*****************************************************************************
        PROCEDIMENTO: informa()

        Retorna informacoes sobre um arquivo.

        PARAMETROS:
        nome: nome do arquivo

*****************************************************************************/

void informa(nome)
char *nome;
{

  char **informacao;
```

```
  if ( ( informacao = informa_1(&nome, cliente)) == NULL)
     {
       clnt_perror(cliente,servidor);
       exit(1);
     }
  if (*informacao == NULL)
    printf("Fracasso!!! Operacao informacao falhou\n");
    else printf("Sucesso!!! %s \n",*informacao);
}


/*************************************************************************
        PROCEDIMENTO: ajuda()

        Procedimento que retorna um menu de todos os comandos disponi-
        veis para operacoes remotas.

        PARAMETROS:
        nenhum

*************************************************************************/

void ajuda()
{

  printf("Menu de funcoes:\n\n");
  printf("Este servidor oferece as seguintes funcoes:\n");
  printf("Cria ->     Cria um arquivo \n");
  printf("Deleta ->  Deleta um arquivo \n");
  printf("Escreve -> Escreve uma sequencia de caracteres em um arquivo\n");
  printf("Anexa ->   Escreve uma sequencia de caracteres no final de um arquivo\n");
  printf("Tam ->     Retorna o tamanho de um arquivo\n");
  printf("Le ->      Le uma sequencia de caracteres de um arquivo\n");
  printf("Mkdir ->   Cria um diretorio\n");
  printf("Rmdir ->   Remove um diretorio\n");
  printf("Pwd ->     Retorna diretorio corrente\n");
  printf("Chdir ->   Muda o diretorio corrente\n");
  printf("Copia ->   Copia um arquivo\n");
  printf("Uso ->     Retorna o tamanho de um determinado diretorio\n");
  printf("Informa -> Retorna informacoes sobre um arquivo \n");
}


/*************************************************************************
        PROCEDIMENTO: erro()

        Procedimento que retorna uma mensagem de erro e termina a execucao do programa.

        PARAMETROS:
```

```
        mensagem: mensagem que sera retornada ao usuario.

*************************************************************************/

void erro(mensagem)
char * mensagem;
{

   printf("%s \n ",mensagem);
   printf("\n Digite < cliente -a > para menu de funcoes \n");
   printf("Beijinho, beijinho...\n");
   exit(1);
}



/*************************************************************************

          PROGRAMA PRINCIPAL

          PARAMETROS:
           argc: numero de parametros entrados
          arqv: lista que contem o comando que sera executado e a sua
          respectiva lista de parametros

*************************************************************************/

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

  int valido = 0;     /* variavel auxiliar        */

  servidor = aloca(15,char);
  cliente = aloca(1,CLIENT);
  if (argc == 1)
    erro("Este comando necessita de parametros.");

  if (strcmp(argv[1],"-a") == 0)
    {
     ajuda();
     valido = 1;
     }

  if ((strcmp(argv[1],"cria") == 0) && (argc != 3))
    {
      printf("Uso: cria <arq> \n ");
      printf("onde <arq>: arquivo \n ");
```

```
     erro("");
   }

if ((strcmp(argv[1],"escreve") == 0) && (argc != 5))
   {
    printf("Uso: escreve <arq> <pos> <buf>\n ");
    printf("onde <arq>: arquivo \n ");
    printf("     <pos>: posicao do caracter inicial \n ");
    printf("     <buf>: buffer que contem os caracteres \n ");
    erro("");
   }

if ((strcmp(argv[1],"deleta") == 0) && (argc != 3))
   {
    printf("Uso: deleta <arq> \n ");
    printf("onde <arq>: arquivo \n ");
    erro("");
   }

 if ((strcmp(argv[1],"anexa") == 0) && (argc != 4))
   {
    printf("Uso: anexa <arq> <buf>\n ");
    printf("onde <arq>: arquivo \n ");
    printf("     <buf>: buffer que contem os caracteres \n ");
    erro("");
   }

if ((strcmp(argv[1],"le") == 0) && (argc != 5))
   {
    printf("Uso: le <arq> <pos> <tam>\n ");
    printf("onde <arq>: arquivo \n ");
    printf("     <pos>: posicao do caracter inicial \n ");
    printf("     <tam>: tamanho do bloco lido \n ");
    erro("");
   }

if ((strcmp(argv[1],"tam") == 0) && (argc != 3))
   {
    printf("Uso: tam <arq> \n ");
    printf("onde <arq>: arquivo \n");
    erro("");
   }

if ((strcmp(argv[1],"mkdir") == 0) && (argc != 3))
   {
    printf("Uso: mkdir <dir> \n ");
    printf("onde <dir>: diretorio \n");
    erro("");
   }
```

```
if ((strcmp(argv[1],"rmdir") == 0) && (argc != 3))
   {
    printf("Uso: rmdir <dir> \n ");
    printf("onde <dir>: diretorio \n");
    erro("");
   }

if ((strcmp(argv[1],"pwd") == 0) && (argc != 2))
   {
    printf("Uso: pwd \n ");
    erro("");
   }

if ((strcmp(argv[1],"chdir") == 0) && (argc != 3))
   {
    printf("Uso: chdir <dir> \n ");
    printf("onde <dir>: diretorio \n");
    erro("");
   }

if ((strcmp(argv[1],"copia") == 0) && (argc != 4))
   {
    printf("Uso: copia <arq1> <arq2>\n ");
    printf("onde <arq1>: arquivo origem \n");
    printf("      <arq2>: arquivo destino \n");
    erro("");
   }

if ((strcmp(argv[1],"uso") == 0) && (argc != 3))
   {
    printf("Uso: uso <dir> \n ");
    printf("onde <dir>: diretorio \n");
    erro("");
   }

 strcpy(servidor,"caiua");

 /* Cria o "handle" do cliente */
 if ( (cliente = (CLIENT *)clnt_create(servidor, STUB_PROG, STUB_VERS, "udp")) == NULL)
   {
     clnt_pcreateerror(servidor);
     exit(1);
   }

 printf("Seja bem vindo ao meu Servidor de Arquivos \n\n");

if (strcmp(argv[1],"cria") == 0)
  {
```

```
      criar_arquivo(argv[2]);
      valido = 1;
    }

if (strcmp(argv[1],"escreve") == 0)
    {
      escrever_arquivo(argv[2],atoi(argv[3]),argv[4]);
      valido = 1;
    }

if (strcmp(argv[1],"deleta") == 0)
    {
      deletar_arquivo(argv[2]);
      valido = 1;
    }

if (strcmp(argv[1],"anexa") == 0)
    {
      append_arquivo(argv[2],argv[3]);
      valido = 1;
    }

if (strcmp(argv[1],"le") == 0)
    {
      ler_arquivo(argv[2],atoi(argv[3]),atoi(argv[4]));
      valido = 1;
    }

if (strcmp(argv[1],"tam") == 0)
    {
      tamanho_arquivo(argv[2]);
      valido = 1;
    }

if (strcmp(argv[1],"mkdir") == 0)
    {
      cria_diretorio(argv[2]);
      valido = 1;
    }

if (strcmp(argv[1],"rmdir") == 0)
    {
      deleta_diretorio(argv[2]);
      valido = 1;
    }

if (strcmp(argv[1],"pwd") == 0)
    {
      meu_pwd();
```

```
      valido = 1;
   }

if (strcmp(argv[1],"chdir") == 0)
   {
      meu_chdir(argv[2]);
      valido = 1;
   }

if (strcmp(argv[1],"copia") == 0)
   {
      copia_arquivo(argv[2],argv[3]);
      valido = 1;
   }

if (strcmp(argv[1],"uso") == 0)
   {
      disco_usado(argv[2]);
      valido = 1;
   }

if (strcmp(argv[1],"informa") == 0)
   {
      informa(argv[2]);
      valido =1;
   }

if (!valido)
   erro("Comando desconhecido!");
  clnt_destroy(cliente);  /* elimina "handle " */
exit(0);

}
```

```
/*****************************************************************************
```

PROGRAMA: Servidor.c

FUNCAO: Oferece um servico de arquivos simplificado que trabalha sobre um servidor de arquvivos remoto. Este servico recebe pedidos de servicos via RPC, apresentando um 'port' registrado no 'portmapper' de onde ele pode ser acessado.

```
*****************************************************************************/

/*

    PROCEDIMENTOS REMOTOS
```

```
*/


#include <stdio.h>
#include <rpc/rpc.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <ftw.h>
#include <pwd.h>
#include <sys/dir.h>
#include <time.h>
#include "stub.h"

#define aloca(num,x) (x *) calloc(num,sizeof(x))

static long soma;


/*************************************************************************
        PROCEDIMENTO: cria_arquivo_1()

        Cria um arquivo.

        PARAMETROS:
        nome: nome do arquivo a ser criado

        RETORNO:
        fd: "file descriptor" do arquivo recem criado ou -1 em caso
            de erro

*************************************************************************/

int *cria_arquivo_1(nome)
char **nome;
{

  static int fd;     /* File Descriptor do arquivo */

  if ( (fd = creat(*nome, 0777) ) == -1)
     return(&fd);
  close(fd);
  return(&fd);
}
```

```
/*************************************************************************
        PROCEDIMENTO: deleta_arquivo_1()

        Deleta um arquivo.

        PARAMETROS:
        nome: nome do arquivo a ser criado

        RETORNO:
        erro: -1 em caso de erro

*************************************************************************/

int *deleta_arquivo_1(nome)
char **nome;
{

  static int erro;

  erro = unlink(*nome);
  return(&erro);
}


 /*************************************************************************
        PROCEDIMENTO: escreve_arquivo()

        Escreve uma sequencia de bytes em um arquivo, a partir de  uma posicao inicial

        PARAMETROS:
        auxiliar->nome: nome do arquivo a ser atualizado
        auxiliar->inicio: posicao inicial
        auxiliar->buffer: string de caracteres a ser escrita

        RETORNO
        erro: -1 em caso de erro

*************************************************************************/

int *escreve_arquivo_1(auxiliar)
aux *auxiliar;
{

  static int erro;
  static int fd;

  if ( (fd = open(auxiliar->nome, O_WRONLY) ) == -1)
      return(&fd);
  if ( (erro = lseek(fd, auxiliar->inicio, SEEK_SET)) == -1)
```

```
    return(&erro);
  if ( (erro = write(fd, auxiliar->buffer, auxiliar->tam_ou_erro))  == -1)
    return(&erro);
  close(fd);
  return(&erro);
}


/*****************************************************************************
       PROCEDIMENTO: append_arquivo_1()

       Anexa uma sequencia de bytes no final de um arquivo.

       PARAMETROS:
       auxiliar_nome: nome do arquivo a ser atualizado
       auxiliar_buffer: string de caracteres a ser escrita

       RETORNO:
       erro: -1 em caso de erro.

*****************************************************************************/

int *append_arquivo_1(auxiliar)
aux *auxiliar;
{

  static int erro;
  static int fd;

  if ( (fd = open(auxiliar->nome, O_WRONLY | O_APPEND) ) == -1)
      return(&fd);
  if ( (erro = write(fd, auxiliar->buffer, auxiliar->tam_ou_erro))  == -1)
    return(&erro);
  close(fd);
  return(&erro);
}


/*****************************************************************************
       PROCEDIMENTO: tamanho_arquivo_1()

       Retorna o tamanho de um arquivo.

       PARAMETROS:
       nome: nome do arquivo

       RETORNO:
       resultado: tamanho do arquivo ou -1 em caso de erro
```

```
*****************************************************************************/

long *tamanho_arquivo_1(nome)
char **nome;
{

  static long resultado;
  struct stat informacoes;

  if ( (resultado = stat(*nome, &informacoes)) == -1)
    return(&resultado);
  resultado = informacoes.st_size;
  return(&resultado);
}


/*****************************************************************************
        PROCEDIMENTO: le_arquivo_1()

        Le uma sequencia de bytes em um arquivo

        PARAMETROS:
        auxiliar->nome: nome do arquivo a ser lido
        auxiliar->inicio: posicao inicial
        auxiliar->tam_ou_erro: quantidade de bytes a ser lida

        RETORNO:
        auxiliar2->buffer: cadeia de caracteres lida
        auxiliar2->tam_ou_erro: -1 em caso de erro

*****************************************************************************/

aux *le_arquivo_1(auxiliar)
aux *auxiliar;
{

  static int fd;
  static int quant;
  static aux auxiliar2;

  if ( (fd = open(auxiliar->nome, O_RDONLY) ) == -1)
       {
         auxiliar2.tam_ou_erro = -1;
         return(&auxiliar2);
       }
  if ( lseek(fd, auxiliar->inicio, SEEK_SET) == -1)
       {
         auxiliar2.tam_ou_erro = -1;
         return(&auxiliar2);
```

```
      }
  if ( (quant = read(fd, auxiliar2.buffer , auxiliar->tam_ou_erro))==-1)
     {
       auxiliar2.tam_ou_erro = -1;
       return(&auxiliar2);
     }
  close(fd);
  auxiliar2.tam_ou_erro = 0;
  auxiliar2.inicio = quant;
  return(&auxiliar2);
}


/*****************************************************************************
        PROCEDIMENTO: cria_diretorio_1()

        Cria um diretorio.

        PARAMETROS:
        nome: nome do diretorio a ser criado

        RETORNO:
        dd: descritor do diretorio ou -1 em caso de erro

*****************************************************************************/

int *cria_diretorio_1(nome)
char **nome;
{

  static int dd;     /* File Descriptor do diretorio */

  dd = mkdir(*nome, 0777);
  return(&dd);
}


/*****************************************************************************
        PROCEDIMENTO: deleta_diretorio_1()

        Deleta um diretorio.

        PARAMETROS:
        nome: nome do diretorio a ser deletado

        RETORNO:
        dd: -1 em caso de erro

*****************************************************************************/
```

```
int *deleta_diretorio_1(nome)
char **nome;
{

  static int dd;

  dd = rmdir(*nome);
  return(&dd);
}


/*****************************************************************************
        PROCEDIMENTO: meu_pwd_1()

        Retorna o diretorio corrente, relacionado ao "root" definido
        por esta aplicacao.

        PARAMETROS:
        sem parametros

        RETORNO:
        dd: diretorio corrente

*****************************************************************************/

char **meu_pwd_1()
{
   static char *dd;

   dd = aloca(64,char);
   dd = getcwd((char *)NULL,64);
   dd  =  dd+36;
   if (strcmp(dd,"") == 0)
     strcpy(dd,"/");
   return(&dd);
}


 /*****************************************************************************
        PROCEDIMENTO: meu_chdir_1()

        Muda o diretorio corrente.

        PARAMETROS:
        nome: nome do novo diretorio corrente

        RETORNO:
        erro: -1 em caso de erro
```

```
*************************************************************************/

int *meu_chdir_1(path)
char **path;
{

  static int erro = 0;

  if ( (strcmp(*meu_pwd_1(),"/") ==  0) && (strcmp(*path,"..") == 0) )
    {
     erro = -1;
     return(&erro);
    }
  erro = chdir(*path);
  return(&erro);
}


/*************************************************************************
       PROCEDIMENTO: copia_arquivo_1()

       Copia um arquivo

       PARAMETROS:
       nome1: nome do arquivo origem
       nome2: nome do arquivo destino

       RETORNO:
       destino: fd do arquivo novo ou -1 em caso de erro

*************************************************************************/

int *copie_arquivo_1(auxiliar)
aux *auxiliar;
{

 static int origem, destino, lido;
 char buffer[BUFSIZ];

 if ((origem = open(auxiliar->nome, O_RDONLY) ) < 0)
    return(&origem);
 if ((destino = creat(auxiliar->buffer, 0777)) < 0)
   return(&destino);
 while ( (lido = read(origem, buffer, BUFSIZ) ) > 0)
    {
     if ( write(destino, buffer, lido) < lido)
        {
          close(origem);
```

```
            close(destino);
            destino = -1;
            return(&destino);
          }
      }

  close(origem);
  close(destino);
  return(&destino);
}


/* Procedimento auxiliar para o comando 'ftw' */

long lista_dir(nome,status,tipo)
  char *nome;
  struct stat *status;
  int tipo;
  {

    if (tipo == FTW_NS)
      return(0);
    if (tipo == FTW_F)
         soma = soma + status->st_size;
    return(0);
  }


/*****************************************************************************
        PROCEDIMENTO: percorre_diretorio_1()

        Retorna a quantidade de bytes ocupado por um determinado di-
        retorio. A arvore contida abaixo deste diretorio e completa-
        mente varrida.

        PARAMETROS:
        nome: nome do diretorio a ser percorrido

        RETORNO:
        soma: tamanho em bytes deste diretorio

*****************************************************************************/

long *percorre_diretorio_1(path)
char **path;
{

  static long resultado = 0;
```

```
  soma = 0;
  if ( (resultado = ftw(*path,lista_dir,1)) == -1)
      return(&resultado);
  return(&soma);
}


/*************************************************************************
        PROCEDIMENTO: informa_1()

        Retorna um conjunto de informacoes sobre um arquivo.

        PARAMETROS:
        path: nome do arquivo

        RETORNO:
        descricao: string contendo informacoes sobre um arquivo

*************************************************************************/

char **informa_1(path)
char **path;
{

/* Use octarray para determinar se bits de permissao combinam */
  static short octarray[9] = {
            400,200,100,040,020,010,
            004,002,001 };

/* codigos mnemonicos para permissoes de arquivos  */
  static char perms[] = "rwxrwxrwx";
  struct stat buffer;
  char *descricao;
  char descrip[9];
  int j;
  time_t *tempo;
  struct passwd *usuario;

  descricao = aloca(255,char);
  tempo = aloca(1,time_t);
  usuario = aloca(10,struct passwd);
  if (stat(*path , &buffer) < 0)
     return(NULL);

/* coloca permissoes numa forma legivel */

  for (j=0; j<9; j++)
    {
      if (buffer.st_mode & octarray[j])
```

```
          descrip[j] = perms[j];
        else descrip[j] = '-';
      }
    descrip[9] = '\0';
    *tempo = buffer.st_mtime;
    usuario = getpwuid(buffer.st_uid);
sprintf(descricao," %s - %ld bytes    Prop: %s   Data: %ld   Perm: %s", *path,buffer.st_size, usuario->pw_name, *tempo, descrip);
return(&descricao);
}
```

3.2. O Utilitário Finger

O *Finger* é um utilitário do UNIX, que tem por objetivo mostrar informações sobre os usuários e também sobre a utilização do sistema. Para realizar essa tarefa, o utilitário *Finger* utiliza-se de dois programas, que são: (1) *finger*, que é o responsável por receber as solicitações dos usuários, através da linha de comando, e por requisitar os serviços do (2) *fingerd*, que é o processo responsável por recuperar e retornar as informações de maneira apropriada.

O *finger* desempenha a função de cliente e o *fingerd* a função de servidor, no protocolo que especifica as suas interfaces de comunicação para as requisições e respostas dos serviços desejados. Este protocolo é chamado de *Name/Finger* e é especificado pelo RFC 742 [HAR77].

Modificando os parâmetros que são passados ao *finger*, modifica-se também as respostas por ele obtidas. Quando não se utiliza nenhum parâmetro são exibidos apenas os usuários que estão utilizando a máquina no exato momento em que o *finger* foi executado. Quando um nome é passado como parâmetro, o *finger* irá procurar entre os *login names* e os *user names* de todos os usuários para verificar a existência ou não do nome solicitado. Caso exista é fornecido um conjunto detalhado de informações, não importando se o usuário está em atividade ou não.

As informações obtidas pelo *finger* são:

- *Login Name* e nome completo do usuário (*User Name*);
- Diretório do usuário (*Home Directory*);
- Interpretador de comandos (shell) utilizado;
- Data e hora do último *login*, ou se ele está ativo, data e hora de acesso ao sistema;
- Data e hora da última vez que recebeu e leu um *mail*;
- Um plano (arquivo *.plan*) que geralmente é utilizado para especificar outras informações como, endereço, telefone e outras;
- Um projeto (arquivo *.project*) utilizado para detalhar, por exemplo, as atividades a serem desenvolvidas no dia e onde o usuário poderá ser encontrado. Segundo o manual da SUN, esse ítem do *Finger* apresenta problemas, segundo [SUN90], pois é exibido apenas a primeira linha do arquivo *.project*.

Essas informações são obtidas nos seguintes arquivos:

- *passwd.byname;*
- */etc/utmp;*
- */var/adm/lastlog*;

Esta implementação teve como objetivo desenvolver um utilitário similar ao utilitário *finger*, fazendo com que este desempenhasse todas as suas funções principais. Para a comunicação entre os processos cliente e servidor foi utilizado o mecanismo de comunicação RPC, descrito anteriormente.

O sistema consiste basicamente de: 1) um programa cliente chamado finger_c, o qual solicita as informações necessárias ao servidor, e 2) um programa servidor chamado finger_d, que é o responsável por receber os pedidos dos clientes e devolver as respostas apropriadas.

Para que o finger_c comunique-se com outra máquina, é necessário que o finger_d esteja sendo executado na máquina desejada (supõe-se que ele tenha sido instalado em tempo de carga do sistema).

Para executar o finger_c o usuário precisa apenas chamá-lo na linha de comando e passar os parâmetros desejados. Os três modos de execução disponíveis são:

- finger_c = Mostra quais usuários estão ativos na estação;
- finger_c nome_usu = Mostra informações mais detalhadas sobre os usuários que possuírem no seu *login name* ou no seu nome completo (*user name*) a expressão nome_usu;
- finger_c @estação = Mostra quais usuários estão ativos na estação especificada por @estação;

Na simulação do utilitário *finger* foram acrescentadas duas informações que não constam do utilitário “original”, o número do grupo do usuário e o número do usuário. O problema relacionado ao arquivo *.project* também foi solucionado. Nessa simulação não foram implementados alguns parâmetros, como -l, -x, e outros, pois estes apenas variam a maneira como as informações são exibidas na tela e /ou a quantidade de detalhes dessas informações.

O RPC foi baseado no descrito em Stevens [STE90]. No utilitário *Finger* desenvolvido sua utilização pode ser representada pela figura 3.2.

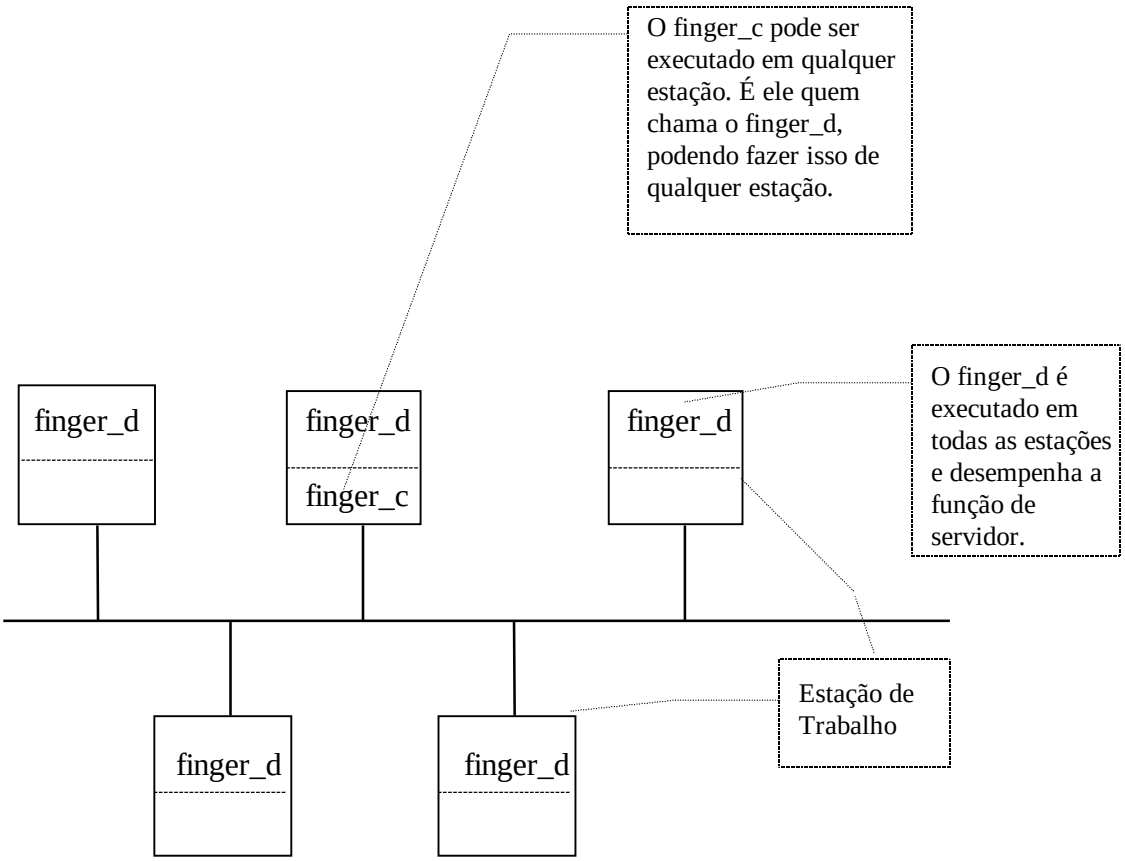

Figura 3.2 - Esquema do utilitário *Finger* desenvolvido.

3.2.1. A Implementação

```
/*****************************************************************************

        PROGRAMA   : stubs.x

        FUNÇÃO: Utilizado para especificar os procedimentos que poderão ser chamados pelo cliente,
bem como os parâmetros de entrada e saída que poderão ser utilizados durante a execução. É através deste
arquivo que o rpcgen, quando executado, gera os procedimentos stubs que serão incluídos no cliente e no
servidor.

*****************************************************************************/

typedef char str_8[8];
typedef char str_16[16];
typedef char str_40[40];
typedef char str_80[80];

/* Estrutura de dados usada para o retorno dos dados lidos no arquivo passwd.byname pelo servidor. Essa
estrutura é usada quando os dados são lidos pela função getpwnam() */
struct Passwd {
        char  name[40];
        int   uid;
        int   gid;
        char  dir[80];
        char  shell[15];
};

/* Estrutura de dados usada para o retorno dos dados lidos no arquivo passwd.byname pelo servidor. Essa
estrutura é usada quando os dados são lidos pela função ypmatch() */
struct Tpasswd {
        int     tot;
        str_8   lname[15];
        str_40  name [15];
        int     uid  [15];
        int     gid  [15];
        str_80  dir  [15];
        str_16  shell[15];
};

/* Estrutura de dados usada para o retorno dos dados lidos no arquivo /etc/utmp pelo servidor.  */
struct Utmp {
        int    tot;
        str_8  term[25];
        str_8  nome[25];
        str_16 onde[25];
        long   hora[25];
};
```

```
/* Estrutura de dados usada para o retorno dos dados lidos no arquivo /var/adm/lastlog  pelo servidor. */
struct Lastlog {
        long hora;
        char term[8];
        char onde[16];
};



/* Estrutura de dados usada para o retorno do calculo feito para a obtenção do idle time  pelo servidor. */
struct IdleTim {
        str_8 term[25];
        int   dia [25];
        int   hor [25];
        int   min [25];
        int   seg [25];
};

/* Estrutura de dados usada para o retorno do cálculo feito para a obtenção da última vez que um mail foi
lido ou recebido, pelo servidor. */
struct Mail {
        long  lido ;
        long  recebido;
};

/*Define quais os procedimentos e quais os parâmetros de entrada e saída que poderão ser utilizados entre
o cliente e o servidor.*/
program DATE_PROG {
        version DATE_VERS {
                Passwd  PEGA_PASSWD(string)       = 1;
                Tpasswd PEGA_TODOS_PASSWD(string) = 2;
                Utmp    PEGA_UTMP(string)         = 3;
                Lastlog PEGA_LASTLOG(int)         = 4;
                IdleTim PEGA_IDLETIM(Utmp)        = 5;
                Mail    PEGA_MAIL(string)         = 6;
        } = 1;              /* número da versão */
} = 0x31234567;          /* número do programa */
```

```
/*****************************************************************************
```

PROGRAMA: finger_c.c

FUNÇÃO: É o programa cliente que solicita as informações para o servidor (finger_d.c). É através deste programa que o usuário tem acesso ao sistema.Todas as informações que ele solicita são feitas através de *RPC*, com exceção dos arquivos .plan e .project que são acessados pelo próprio cliente visto que estes arquivos não pertencem ao NIS *(Network Information Service)*.

```
*****************************************************************************/
#include <stdio.h>
#include <time.h>
#include <pwd.h>
#include <rpc/rpc.h>
#include "stubs.h"          /* gerado pelo rpcgen */

typedef char String[15];

/* Usada para controlar os servidores disponíveis para atendimento das requisições */
struct Reg_CLI {
        char    nom_server[15];
        CLIENT  *cli_id;
};

CLIENT *cl;
struct Reg_CLI vet_hosts[25];
int tot_hosts;
char name_local[15], server[15];


/*****************************************************************************
        PROCEDIMENTO: acha_server()

        Localiza no vetor de servidores o identificados de uma determinada estação para posterior
        chamada remota.

        PARÂMETROS :
        Nome da estação

*****************************************************************************/

CLIENT *acha_server(nom_server)
String nom_server;
{
        int i;
        if( nom_server[0] == '@' )
                nom_server++;
        i = 0;
        while( i < tot_hosts )  {
```

```
                if ( strcmp(vet_hosts[i].nom_server, nom_server) == 0 )
                        return(vet_hosts[i].cli_id);
                i++;
        }
        return (NULL);
}


/*****************************************************************************
        PROCEDIMENTO: carrega_servers()

        Estabelece conexão entre as estações que estão atuando como servidoras (possuem o finger_d).
        Essas estações são conhecidas através de um arquivo texto que possui o nome das estações.

        PARÂMETROS: Nome do arquivo texto com nome das estações.

*****************************************************************************/

void carrega_servers(nome_arq)
char *nome_arq;
{
        FILE *fd_finger;
        if ( (fd_finger = fopen(nome_arq, "r") ) == NULL)          {
                printf("Erro na abertura do %s... \n", nome_arq);
                exit(-1);
        }

        tot_hosts = 0;
        while (fread(vet_hosts[tot_hosts].nom_server, sizeof(vet_hosts[tot_hosts].nom_server), 1,
fd_finger) == 1){
                vet_hosts[tot_hosts].cli_id = (CLIENT *) malloc( sizeof(CLIENT) );
                if( (vet_hosts[tot_hosts].cli_id = clnt_create(vet_hosts[tot_hosts].nom_server,
DATE_PROG, DATE_VERS, "udp") ) == NULL)                    {
                        printf("Cliente nao pode estabelecer conexao com o servidor %s... \n",
vet_hosts[tot_hosts].nom_server);
                        clnt_pcreateerror(vet_hosts[tot_hosts].nom_server);
                }
                else
                        tot_hosts++;
        }

        fclose(fd_finger);
        return;
 }


/*****************************************************************************
        PROCEDIMENTO: mostra_plan ()
```

```
        Mostra o arquivo .plan  caso ele exista.

        PARÂMETROS:
        Caminho a ser percorrido para localizar o arquivo .plan .

*****************************************************************************/

void mostra_plan(nome_dir)
char *nome_dir;
{
        FILE *fd_finger;
        char *nome_arq, cadeia[100];

        nome_arq = (char *) malloc(100*sizeof(char));
        strcat(nome_arq, nome_dir);
        strcat(nome_arq, "/.plan");

        if ( (fd_finger = fopen(nome_arq, "r") ) == NULL)
                printf("Sem .plan \n");
        else    {
                printf("Arquivo .plan: \n");
                while (fgets(cadeia, sizeof(cadeia), fd_finger) != NULL)
                        printf("%s", cadeia);
                fclose(fd_finger);
        }
        return;
}



/*****************************************************************************
        PROCEDIMENTO: mostra_proj()

        Mostra o arquivo .project caso ele exista

        PARÂMETROS:
        Caminho a ser percorrido para localizar o arquivo .project .

*****************************************************************************/

void mostra_proj(nome_dir)
char *nome_dir;
{
        FILE *fd_finger;
        char *nome_arq, cadeia[100];

        nome_arq = (char *) malloc(100*sizeof(char));
        strcat(nome_arq, nome_dir);
        strcat(nome_arq, "/.project");
```

```
        if ( (fd_finger = fopen(nome_arq, "r") ) == NULL)
                printf("Sem .project \n");
        else    {
                printf("Arquivo .project: \n");
                while (fgets(cadeia, sizeof(cadeia), fd_finger) != NULL)
                        printf("%s", cadeia);
                fclose(fd_finger);
        }
        return;
}


/*****************************************************************************
        PROCEDIMENTO: finger_sozinho()

        Solicita e exibe informações resumidas de todos os usuários de uma estação.
        São relacionados todos os usuários ativos no momento. Essa função é executada quando o
         finger_c é chamado sem um nome para o usuário (sozinho).

        PARÂMETROS:
        nenhum.
*****************************************************************************/

void finger_sozinho()
{
        Utmp    *reg_utmp;
        Passwd  *reg_pwd;
        Lastlog *reg_lastlog;
        IdleTim *reg_idle;
        int i, k;
        char *aux, *quando, *nome_usr = "todos";

        aux    = (char *) malloc(9*sizeof(char));
        quando = (char *) malloc(10*sizeof(char));
        reg_lastlog = (Lastlog *) malloc( sizeof(Lastlog) );

        if (( reg_utmp = pega_utmp_1(&nome_usr, cl)) == NULL) {
                printf("pega_utmp nao funcionou no servidor...\n");
                clnt_perror(cl, server);
                exit(3);
        }


        if (( reg_idle = pega_idletim_1(reg_utmp, cl)) == NULL) {
                printf("pega_idletim nao funcionou no servidor...\n");
                clnt_perror(cl, server);
                exit(3);
        }
```

```
if ( strcmp(server, name_local) != 0)
        printf("[%s] \n", server);
printf("  Login  - ");
printf("    Nome Completo    - ");
printf("tty - ");
printf(" Inativo - ");
printf("  Quando  - ");
printf("      Onde      \n");

if(reg_utmp->tot < 0)     {
        printf("Nao ha ninguem... \n");
        return;
}
for(i = 0; i <= reg_utmp->tot; i++)          {
        strcpy(nome_usr, reg_utmp->nome[i]);
        nome_usr[8] = '\0';
        if (( reg_pwd = pega_passwd_1(&nome_usr, cl)) == NULL) {
                printf("pega_passwd nao funcionou no servidor...\n");
                clnt_perror(cl, server);
                exit(3);
        }
        if (( reg_lastlog = pega_lastlog_1(&(reg_pwd->uid), cl)) == NULL)          {
                printf("pega_lastlog nao funcionou no servidor...\n");
                clnt_perror(cl, server);
                exit(3);
        }
        if (reg_lastlog->hora == reg_utmp->hora[i])        {
                printf("%8s   ", nome_usr);
                strcpy(aux, reg_pwd->name);
                aux[20] = '\0';
                printf("%20s   ", aux);

                if (strcmp(reg_utmp->term[i], "console") == 0)    {
                        strcpy(aux, reg_utmp->term[i]);
                        aux[2] = '\0';
                        printf("%3s   ", aux);
                        printf("           ");
                }
                else      {
                        strcpy(aux, reg_utmp->term[i]);
                        aux++; aux++; aux++;
                        printf("%3s   ", aux);
                        if (reg_idle->dia[i] > 0)
                                printf("%2d:", reg_idle->dia[i]);
                        else
                                printf("   ");
                        if (reg_idle->hor[i] > 0)
                                printf("%2d:", reg_idle->hor[i]);
```

```
                                else
                                        printf("   ");
                                if (reg_idle->min[i] > 0)
                                        printf("%2d   ", reg_idle->min[i]);
                                else
                                        printf("     ");
                        }
                        quando = ctime(&(reg_utmp->hora[i]));
                        strncpy(aux, quando, 4);
                        aux[4] = '\0';
                        quando = ctime(&(reg_utmp->hora[i]));
                        for(k = 0; k < 11; k++, quando++);
                        strncat(aux, quando, 5);
                        printf("%10s   ", aux);
                        printf("%s \n", reg_utmp->onde[i]);
                }
        }
        free(aux);
}


/*****************************************************************************
        PROCEDIMENTO: finger_name()

        Solicita e exibe informações detalhadas sobre um usuário, especificado por nome_usr.

        PARÂMETROS:
        Nome do usuário.

*****************************************************************************/

void finger_name(nome_usr)
char *nome_usr;
{
        Tpasswd *reg_tpwd;
        Utmp    *reg_utmp;
        Lastlog *reg_lastlog;
        IdleTim *reg_idle;
        Mail    *reg_mail;
        int i, k, mostrou_mail, mostrou_plan, mostrou_proj;
        char *aux, *quando;

        aux    = (char *) malloc(9*sizeof(char));
        quando = (char *) malloc(10*sizeof(char));
        reg_lastlog = (Lastlog *) malloc( sizeof(Lastlog) );

        if (( reg_utmp = pega_utmp_1(&nome_usr, cl)) == NULL) {
                printf("pega_utmp nao funcionou no servidor...\n");
                clnt_perror(cl, server);
```

```
        exit(3);
}
if (( reg_idle = pega_idletim_1(reg_utmp, cl)) == NULL) {
        printf("pega_idletim nao funcionou no servidor...\n");
        clnt_perror(cl, server);
        exit(3);
}
if (( reg_tpwd = pega_todos_passwd_1(&nome_usr, cl)) == NULL) {
        printf("pega_todos_passwd nao funcionou no servidor...\n");
        clnt_perror(cl, server);
        exit(3);
}
if( reg_tpwd->tot < 0 )   {
        printf("\nNome nao encontrado...\n");
        return;
}
for(i = 0; i <= reg_tpwd->tot; i++)          {
        strcpy(nome_usr,reg_tpwd->lname[i]);
        nome_usr[8] = '\0';
        if (( reg_lastlog = pega_lastlog_1(&(reg_tpwd->uid[i]), cl)) == NULL)     {
                printf("pega_lastlog nao funcionou no servidor...\n");
                clnt_perror(cl, server);
                exit(3);
        }
        if (( reg_mail = pega_mail_1(&nome_usr, cl)) == NULL)            {
                printf("pega_mail nao funcionou no servidor...\n");
                clnt_perror(cl, server);
                exit(3);
        }
        if(reg_utmp->tot < 0)
                reg_utmp->tot = 0;

        mostrou_mail = 0;
        mostrou_plan = 0;
        mostrou_proj = 0;
        for( k = 0; k <= reg_utmp->tot; k++)     {
                if( (strcmp(nome_usr, reg_utmp->nome[k]) == 0) ||
                   (k == reg_utmp->tot) )           {
                        printf("\n");
                        printf("Login.......: %s \n", reg_tpwd->lname[i]);
                        printf("Na vida real: %s  \n", reg_tpwd->name [i]);
                        printf("Diretorio...: %s  \n", reg_tpwd->dir[i]);
                        printf("Shell.......: %s  \n", reg_tpwd->shell[i]);
                        printf("Uid.........: %4d     ", reg_tpwd->uid  [i]);
                        printf("Gid: %3d \n", reg_tpwd->gid  [i]);

                        if( strcmp(nome_usr, reg_utmp->nome[k]) == 0) {
                                aux = ctime(&(reg_utmp->hora[k]));
                                aux[strlen(aux)-1] = '\0';
```

```
                                        printf("Login desde.: %s  ", aux);
                                        printf("no %s  ", reg_utmp->term[k]);
                                        if( strlen(reg_utmp->onde[k]) > 0 )
                                                printf("da %s \n", reg_utmp->onde[k]);
                                        else
                                                printf("\n");
                                        if (strcmp(reg_utmp->term[k], "console") != 0 )
                                                printf("Inativo.....: %2d dias, %2d horas, %2d minutos e
%2d segundos\n",
                                                        reg_idle->dia[k], reg_idle->hor[k], reg_idle-
>min[k], reg_idle->seg[k] );
                                }
                                else    {
                                        aux = ctime(&(reg_lastlog->hora));
                                        aux[strlen(aux)-1] = '\0';
                                        printf("Ultimo login: %s;  ", aux );
                                        printf("no %s  ", reg_lastlog->term);
                                        if( strlen(reg_lastlog->onde) > 0 )
                                                printf("da %s\n", reg_lastlog->onde);
                                        else
                                                printf("\n");
                                }

                                if (!mostrou_mail)          {
                                  printf("Mail.....: ");
                                  mostrou_mail = 1;
                                  if( reg_mail->recebido > reg_mail->lido)  {
                                            printf("Ultimo rec.: %s", ctime(&(reg_mail->recebido)));
                                            printf("              ");
                                            printf("Ultimo lido: %s", ctime(&(reg_mail->lido)) );
                                  }
                                  else
                                            printf("Ultimo lido: %s", ctime(&(reg_mail->lido)) );
                                }
                                if (!mostrou_proj)          {
                                        mostrou_proj = 1;
                                        mostra_proj(reg_tpwd->dir[i]);
                                }
                                if (!mostrou_plan)          {
                                        mostrou_plan = 1;
                                        mostra_plan(reg_tpwd->dir[i]);
                                }
                        }
                }
        }
}


/*****************************************************************************
```

PROCEDIMENTO: main ()

Determina se a busca das informações será feita para um nome de usuário específico ou para todos de usuários ativos em uma determinada estação. Esta escolha é feita através dos parâmetros de entrada.

PARÂMETROS: Pode variar entre:
o nome de um usuário;
o nome de uma estação precedido do caracter @;
nenhum parâmetro (default:pega nome da estação do cliente).

**********/

```
main(argc, argv)
int  argc;
char *argv[];
{
        char *nome_usr;
        int i;

        nome_usr = (char *) malloc(20*sizeof(char));
        gethostname(name_local, sizeof(name_local) );
        carrega_servers("fingerd.tmp");
        if ( argc < 2 )     {
                strcpy(nome_usr, "todos");
                cl = acha_server(name_local);
                strcpy(server, name_local);
        }
        else      {
                strcpy(nome_usr, argv[1]);
                if (nome_usr[0] == '@')  {
                        cl = acha_server(nome_usr);
                        strcpy(server, ++nome_usr);
                        strcpy(nome_usr, "todos");
                }
                else      {
                        cl = acha_server(name_local);
                        strcpy(server, name_local);
                }
                if( cl == NULL){
                        printf("Maquina nao localizada... \n");
                        exit(-1);
                }
        }
        if (strcmp(nome_usr, "todos") == 0)
                finger_sozinho();
        else
                finger_name(nome_usr);
```

```
        clnt_destroy(cl);
        for (i = 0; i < tot_hosts; i++)
                clnt_destroy(vet_hosts[i].cli_id);
        exit(0);
}
```

/***

PROGRAMA: finger_d.c

FUNÇÃO: É o procedimento servidor, que recebe os pedidos do cliente, recupera as informações solicitadas e retorna, ou o que foi requisitado, ou um código de erro. As informações são recuperadas de arquivos específicos do *NIS*, de acordo com o que é solicitado pelo cliente. Para a recuperação destas informações foram utilizadas funções de mais baixo nível, sendo que não foi utilizado nenhum comando do *shell* do *Unix*.

***/

```
#include <pwd.h>
#include <rpc/rpc.h>
#include "stubs.h"
#include <stdio.h>
#include <rpcsvc/ypclnt.h>
#include <utmp.h>
#include "lastlog.h"
#include <ttyent.h>
#include <sys/types.h>
#include <sys/stat.h>

struct hora_convert {
        int dia;
        int hor;
        int min;
        int seg;
        };


/*****************************************************************************
        PROCEDIMENTO: abre_arq()

        Utilizada para abrir os arquivos que precisarão ser acessados.

        PARÂMETROS:
        Nome do arquivo e o modo de abertura.

*****************************************************************************/

FILE * abre_arq(nome_arq, modo)
char *nome_arq;
char modo[2];
{
        FILE *fd_arq;

        if( (fd_arq = fopen(nome_arq,modo)) == NULL) {
                printf("ERRO: Nao conseguiu abrir o arquivo %s. \n", nome_arq);
                exit(0);
```

```
        }
        return fd_arq;
}

/*****************************************************************************
        PROCEDIMENTO: dif_tempo()

        Retorna a diferença entre os long int tempo1 e tempo2  (tempo1 - tempo2) em dias, horas,
        minutos e segundos.

        PARÂMETROS:
        Dois horários como inteiros.

*****************************************************************************/

struct hora_convert  dif_tempo(tempo1, tempo2)
time_t tempo1, tempo2;
{
        long seg = 0, min = 0, hor = 0, dia = 0;
        struct hora_convert hc;

        if ( (tempo1 - tempo2) > 60)       {
                min = (long) ( (tempo1 - tempo2) / 60 );
                seg = (long) ( (tempo1 - tempo2) - (min * 60) );
                if ( min > 60 )    {
                        hor = (long) ( min / 60 );
                        min = (long) ( min - (hor * 60) );
                        if ( hor > 23 )    {
                                dia = (long) ( hor / 24 );
                                hor = (long) ( hor - (dia * 24) );
                        }
                }
        }
        else
                seg = (tempo1 - tempo2);

        hc.dia = (int) dia;
        hc.hor = (int) hor;
        hc.min = (int) min;
        hc.seg = (int) seg;
        return hc;
}


/*****************************************************************************
        PROCEDIMENTO: lower()

        Converte uma string para minúscula.
```

```
        PARÂMETROS:
        Ponteiro para string maiúscula.

*****************************************************************************/

char *lower(s)
char *s;
{
        char *b= s;

        while (*s) {
                *s = tolower(*s);
                ++s;
        }
        return b;
}


/*****************************************************************************
        PROCEDIMENTO: pega_passwd_1()

        Obtém as informações do passwd.byname (NIS), utilizando para isto a função getpwnam().
        A busca é feita pelo *login name* do usuário e é interrompida na primeira ocorrência de *nome*.

        PARÂMETROS:
        Nome do usuário.

*****************************************************************************/

Passwd *pega_passwd_1(nome)
char **nome;
{
        static Passwd ret_pass;
        struct passwd *reg_pass;

        char *indomain, *outval, nom_completo[40];
        int  outvallen, i, pos1, pos2;

        yp_get_default_domain(&indomain);
        yp_bind(indomain);
        yp_match(indomain,"passwd.byname", *nome, strlen((*nome)), &outval, &outvallen);
        if ( (reg_pass = getpwnam(*nome)) != NULL )    {
                i = 0;
                pos1 = 0;
                while ( (i < 4) && (pos1 <= outvallen) )  {
                        if (outval[pos1] == ':')
                                i++;
                        pos1++;
                }
```

```
                pos2 = 0;
                while ( (outval[pos1] != ':') && (pos1 <= outvallen) )       {
                        nom_completo[pos2] = outval[pos1];
                        pos2++;
                        pos1++;
                }
                nom_completo[--pos2] = '\0';
                strcpy(ret_pass.name, nom_completo);
                ret_pass.uid   = reg_pass->pw_uid;
                ret_pass.gid   = reg_pass->pw_gid;
                strcpy(ret_pass.dir, reg_pass->pw_dir);
                strcpy(ret_pass.shell, reg_pass->pw_shell);
        }
        else      {
                strcpy(ret_pass.name, "");
                ret_pass.uid   = 0;
                ret_pass.gid   = 0;
                strcpy(ret_pass.dir, "");
                strcpy(ret_pass.shell, "");
        }
        return (&ret_pass);
}


/*****************************************************************************
        PROCEDIMENTO: pega_todos_passwd_1()

        Recupera todos os usuários que possuam no seu nome completo ou no seu login name a
        string passada como parâmetro Essa função retorna um vetor com os usuário encontrados.

        PARÂMETROS:
        Nome para ser procura (pode ser parcial ou total).

*****************************************************************************/

Tpasswd *pega_todos_passwd_1(nome)
char **nome;
{
        static Tpasswd ret_pass;

        char *indomain, inkey[100], *outval, *outkey, *inval;
        char login_name[8], nom_completo[40], home_dir[80], shell[16], captura[81];
        char *m_login_name, *m_nom_completo, *m_nome;

        int  uid, gid, aux;
        int  outvallen, i, pos1, pos2, result, outkeylen, inkeylen, invallen, cont;

        yp_get_default_domain(&indomain);
        yp_bind(indomain);
```

```
yp_first(indomain,"passwd.byname",&outkey,&outkeylen,&outval,&outvallen);
aux = 0;
cont = 0;
do {
        i = 0;
        pos1 = 0;
        while ( (pos1 <= outvallen) )       {
                pos2 = 0;
                while ( (outval[pos1] != ':') && (pos1 <= outvallen) )       {
                        captura[pos2] = outval[pos1];
                        pos2++;
                        pos1++;
                }
                captura[pos2] = '\0';
                switch(i) {
                        case 0:
                                strcpy(login_name, captura);
                                break;
                        case 2:
                                uid = atoi(captura);
                                break;
                        case 3:
                                gid = atoi(captura);
                                break;
                        case 4:
                                captura[--pos2] = '\0';
                                strcpy(nom_completo, captura);
                                break;
                        case 5:
                                strcpy(home_dir, captura);
                                break;
                        case 6:
                                captura[--pos2] = '\0';
                                strcpy(shell, captura);
                                break;
                }
                i++;
                pos1++;
        }
        m_login_name   = lower(login_name);
        m_nom_completo = lower(nom_completo);
        m_nome         = lower(*nome);
        if ( (strstr(m_login_name,   m_nome) != NULL) ||
            (strstr(m_nom_completo, m_nome) != NULL) )     {
                ret_pass.tot = cont;
                strcpy(ret_pass.lname[cont], login_name);
                strcpy(ret_pass.name [cont], nom_completo);
                ret_pass.uid[cont] = uid;
                ret_pass.gid[cont] = gid;
```

```
                        strcpy(ret_pass.dir  [cont], home_dir);
                        strcpy(ret_pass.shell[cont], shell);
                        cont++;
                }
                strcpy(inkey, outkey);
                inkeylen = outkeylen;
                result = yp_next(indomain,"passwd.byname",inkey,inkeylen,
                        &outkey,&outkeylen,&outval,&outvallen);

        } while (result != YPERR_NOMORE);

        if ( cont == 0 )
                ret_pass.tot = -1;
        return(&ret_pass);
}


/*****************************************************************************
        PROCEDIMENTO: pega_utmp_1()

        Recupera as informações sobre os terminais que estão sendo utilizados no momento, em uma
        determinada estação. Isso permite descobrir quem está ativo no momento.
        Retorna um vetor contendo os dados de cada terminal ativo.

        PARÂMETROS:
        Nome de um usuário ou a string “todos”.

*****************************************************************************/

Utmp *pega_utmp_1(nome)
char **nome;
{
        int i;
        FILE *fd_utmp = NULL;
        struct utmp    rec_ut;

        static Utmp reg_utmp;

        fd_utmp = abre_arq("/etc/utmp", "r");

        i = 0;
        while (fread(&rec_ut, sizeof(rec_ut), 1, fd_utmp) != 0)   {
                if (strcmp(rec_ut.ut_name, *nome) == 0 ||
                  (strcmp(*nome, "todos") == 0 && strcmp(rec_ut.ut_name,"") != 0) )     {
                        reg_utmp.tot = i;
                        strcpy(reg_utmp.term[i], rec_ut.ut_line);
                        strcpy(reg_utmp.nome[i], rec_ut.ut_name);
                        strcpy(reg_utmp.onde[i], rec_ut.ut_host);
                        reg_utmp.hora[i] = rec_ut.ut_time;
```

```
                        i++;
                }
        }
        if( i == 0)
                reg_utmp.tot = -1;
        fclose(fd_utmp);
        return(&reg_utmp);
}


/*****************************************************************************
        PROCEDIMENTO: pega_lastlog_1()

        Caso o usuário não esteja ativo, verifica qual foi seu último *login*; caso esteja ativo verifica
        desde quando ele está ativo.

        PARÂMETROS: Número do identificador do usuário.

*****************************************************************************/

Lastlog *pega_lastlog_1(num_uid)
int *num_uid;
{
        FILE *fd_lastlog;
        struct lastlog rec_ll;
        static Lastlog reg_lastlog;

        fd_lastlog = abre_arq("/var/adm/lastlog", "r");
        fseek(fd_lastlog, (long)( (*num_uid)*sizeof(rec_ll) ), 0);
        fread(&rec_ll, sizeof(rec_ll), 1, fd_lastlog);
        fclose(fd_lastlog);

        reg_lastlog.hora = rec_ll.ll_time;
        strcpy(reg_lastlog.term, rec_ll.ll_line);
        strcpy(reg_lastlog.onde, rec_ll.ll_host);

        return(&reg_lastlog);
}


/*****************************************************************************
        PROCEDIMENTO: pega_idletim_1()

        Calcula em segundos, minutos, horas e dias, o tempo que os terminais ativos estão sem operação.

        PARÂMETROS:
        Estrutura contendo todos os terminais ativos na estação.

*****************************************************************************/
```

```
IdleTim *pega_idletim_1(reg_utmp)
Utmp *reg_utmp;
{
        static IdleTim reg_idle;
        struct stat *buf = NULL;
        struct hora_convert hc;

        time_t time_hoje;
        int i;
        char nom_term[10];

        buf = (struct stat *) malloc(sizeof(struct stat) );
        for(i = 0; i <= reg_utmp->tot; i++)         {
                strcpy(nom_term, "/dev/");
                strcat(nom_term, reg_utmp->term[i]);
                stat(nom_term, buf);
                time_hoje = time(NULL);
                hc = dif_tempo(time_hoje, (*buf).st_atime);
                strcpy(reg_idle.term[i], reg_utmp->term[i]);
                reg_idle.dia[i] = hc.dia;
                reg_idle.hor[i] = hc.hor;
                reg_idle.min[i] = hc.min;
                reg_idle.seg[i] = hc.seg;
        }
        return(&reg_idle);
}


/*****************************************************************************
        PROCEDIMENTO: pega_mail_1()

        Retorna a hora da última vez que um *mail* foi lido ou recebido

        PARÂMETROS:
        Nome do usuário.

*****************************************************************************/

Mail *pega_mail_1(nome)
char **nome;
{
        static Mail reg_mail;
        struct stat *buf = NULL;
        struct hora_convert hc;

        time_t time_hoje;
        char nom_mail[25];
```

```
        buf = (struct stat *) malloc(sizeof(struct stat) );
        strcpy(nom_mail, "/var/spool/mail/");
        strcat(nom_mail, *nome);
        stat(nom_mail, buf);

        reg_mail.lido     = (*buf).st_atime;
        reg_mail.recebido = (*buf).st_mtime;

        return(&reg_mail);
}
```

4. Conclusões

O estudo de RPC revela que esse mecanismo de comunicação não é de difícil compreensão, pois, aliando-se a semelhança que esse apresenta em relação a LPC com uma bibliografia farta e bem definida, consegue-se obter uma rápida assimilação.

Para a implementação dos exemplos de RPC mostrados nesse trabalho, entretanto, houve dificuldades que poderiam ser sanadas rapidamente, caso houvesse, ***na bibliografia disponível***, mais exemplos ilustrativos dos detalhes contidos no modelo Sun RPC.

Para a definição de parâmetros formados por apenas um tipo básico (como *strings* e inteiros), pode-se usar [STE90] como referência, e ter-se-á uma explicação suscinta e um exemplo simples. Porém, desejando-se transmitir estruturas de dados derivadas (como ***structs*** ou ***unions***), as referência citada apresenta-se insuficiente, pois, apenas são citados o modelo de transmissão **XDR** e a chamada linguagem RPC, não sendo apresentado nenhum exemplo. Uma descrição mais elaborada sobre esses dois tópicos é apresentado em [SUN90], porém é importante ressaltar que tais conceitos apresentaram relativa dificuldade para serem assimilados.

O modelo SUN RPC é fortemente baseado em ponteiros, o que gera a necessidade adicional da implementação ser feita com extremo cuidado a fim de se evitar problemas de alocação de memória, erros esses que não são fáceis de se encontrar.

Para a implementação dos exemplos mostrados, o modelo RPC mostrou ser uma ferramenta eficiente, a qual se encaixou perfeitamente nos requisitos que eram necessários. Pode-se afirmar, que sua implementação é simples comparando-se com outros mecanismo de comunicação, levando-se em conta a quantidade de tarefas que esta interface oferece, como o registro no "portmapper", tratamento de erros, etc. É importante ressaltar que tais afirmações podem não ser verdadeiras para outros modelos RPC, que não o Sun RPC.

5. Referências Bibliográficas

[1] Stevens, W. R. **UNIX Network Programming.** Prentice Hall, Englewood Cliffs, 1990.

[2] Salama, B. & Haviland, K. **UNIX System Programming**. Addison-Wesley, 1987.

[3] **SunOS Reference Manuals. Programming Utilies and Libraries. Network Programming.** 1990.

6. Bibliografia auxilar

[1] Tanenbaum, A. S. **Modern Operating Systems**, Prentice-Hall, 1991.

[2] Colouris, G. F. & Dollimore, J. **Distributed Systems**, Addison-Wesley Publishing Company, 1988.

[3] Mullender, S. (ed) **Distributed Systems**. ACM PRESS Frontier Series, Addison-Wesley Publishing Company, 1989.