Electrónica e Telecomunicações

universidade de aveiro

theoria poiesis praxis

AVEIRO • SET. • 95 • VOL. 1 • Nº 4
Revista do Departamento de Electrónica e Telecomunicações da Universidade de Aveiro

Modelo de Integração de Aplicações Distribuídas e não Colaborantes

Rui Pedro Lopes, António Miguel Esteves, José Luís Oliveira, Joaquim Arnaldo Martins

***Resumo*- Este artigo descreve um modelo, para o desenvolvimento de aplicações distribuídas, que permite a divisão de uma aplicação em diversos módulos de processamento. Serão igualmente apresentadas, as potencialidades deste modelo para integrar aplicações autónomas e não colaborantes. Apesar da generalidade do sistema proposto a sua aplicabilidade será discutida no âmbito de um projecto de um Sistema de Gestão de Redes.**

***Abstract*- This paper describes a model that enables the development of distributed applications and provides an interface for a transparent integration of standalone packages. This model is presented in the context of a Network Management System scenario.**

I. INTRODUÇÃO*

O desenvolvimento de aplicações é, cada vez mais, condicionado por um conjunto crescente de requisitos, relacionados com o aumento de funcionalidade, com a necessidade de servir uma maior leque de sistemas e, ainda, com o aumento de desempenho. Estas solicitações, embora nem sempre sejam conciliáveis (desempenho *versus* funcionalidade), são, normalmente, prioritárias na estratégia de desenvolvimento, pelo que é importante encontrar soluções que permitam a sua optimização. A divisão da aplicação em módulos e a sua distribuição por diferentes processos concorrentes, é uma das técnicas que permite atingir este objectivo.

Esta sub-divisão pode resultar de uma política explícita no planeamento do sistema ou, pelo contrário, ser consequência da utilização de aplicações autónomas, desenvolvidas sem qualquer mecanismo que permita a partilha directa com outras aplicações. A proliferação, em domínio público, de aplicações que fornecem soluções parciais para determinados problemas, aumenta significativamente a quota deste segundo tipo de distribuição. A integração e a reutilização, numa aplicação, da informação fornecida por estes programas/módulos, assume-se como uma tarefa de especial importância.

O trabalho apresentado ao longo deste artigo foi polarizado segundo estes dois objectivos: a distribuição de processos e aplicações; e a integração de aplicações não colaborantes. Como primeira abordagem são apresentadas algumas técnicas de comunicação entre processos (locais ou remotos), especialmente o método de passagem de mensagens (*IPC message queues*) e o mecanismo de RPC (*Remote Procedure Call*). Será apresentado um modelo que permite utilizar, de uma forma transparente para o utilizador, os dois sistemas em simultâneo e que se baseia no conceito de comutador de aplicações. Será enquadrado, igualmente, um cenário que rentabiliza e justifica a utilização deste modelo (gestão de redes de dados).

II. TÉCNICAS GERAIS DE COMUNICAÇÃO ENTRE PROCESSOS

A partilha de dados entre diferentes procedimentos (*functions*), de uma dada aplicação, pode ser feita utilizando variáveis globais. Este método é, no entanto, pouco "elegante" pois conduz a códigos menos claros e mais sujeito a erros. Uma alternativa a esta solução é dada pelas chamadas a funções, ao permitirem a passagem de argumentos e o retorno de resultados.

Estas soluções, embora funcionais para ambientes monoprocesso, tornam-se ineficazes sempre que se pretende trocar informação entre diferentes processos (cenários multiprocesso).

Devido a esta limitação, foram sendo desenvolvidos, e integrados em alguns sistemas operativos, vários métodos de comunicação entre processos, ou IPC (*InterProcess Communication*). Alguns dos métodos são típicos do sistema UNIX, como por exemplo: *pipes*, *fifos* (ou *named pipes*), passagem de mensagens (*message queues*) e memória partilhada (*shared memory*), semáforos (*semaphores*) [1,2,3]. Todos estas técnicas utilizam o *Kernel* do sistema operativo como coordenador de informação (Fig. 1), apesar de não ser obrigatório para a comunicação entre processos.

Estes métodos apresentam algumas similaridades e são resultado, essencialmente, de diferentes soluções técnicas. Contudo, alguns estão limitados a relações tipo *parent-child* (*pipes*, *fifos*) enquanto que outros são aplicáveis

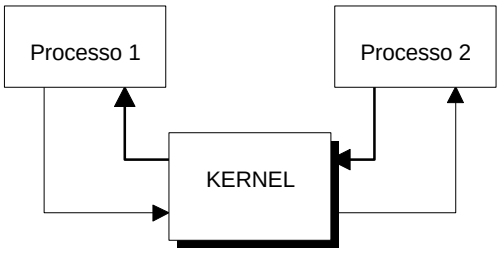

Fig. 1 - IPC entre dois processos num mesmo sistema.

* Trabalho realizado no âmbito da disciplina de Projecto.

sobre processos independentes (*message queues*, *shared memory*, *semaphores*).

Qualquer um destes métodos soluciona o problema de comunicação entre processos num mesmo sistema. A comunicação entre processos em diferentes sistemas coloca, no entanto, outro tipo de condicionalismos que não são resolvidos por estes. O mecanismo de RPC (*Remote Procedure Call*) é um dos métodos que permitem solucionar este tipo de problemas.

Na descrição seguinte serão apresentados os métodos de passagem de mensagens e de RPC.

A. IPC - Passagem de Mensagens

A passagem de mensagens constitui, juntamente com a memória partilhada e os semáforos, o *System V IPC*. O conceito principal deste sistema é a troca de mensagens (sob a forma de uma estrutura de dados) colocadas numa pilha, que é identificada por uma chave própria. A chave (tipo *key_t*, inteiro de 32 bits) pode ser obtida por três métodos:

- usar uma chave privada (IPC_PRIVATE) obrigando à troca do identificador, gerado pela chave, através de um ficheiro;
- pré-acordada entre o cliente e o servidor (estática e acessível no código);
- criada (*ftok*) a partir de um *pathname* e um valor entre 0 e 255.

Uma vez na posse da chave, que irá servir para gerar o espaço comum de passagem de mensagens, os processos podem inicializar as estruturas necessárias e desencadear o processo de comunicação.

O sistema de desenvolvimento disponibiliza um conjunto de primitivas para operação sobre mensagens (Tabela 1).

Uma pilha de mensagens é criada, ou uma já existente é acedida, usando a primitiva *msgget*:

```
int msgget(key_t key, int flag);
```

O valor *flag* indica a acessibilidade da pilha (semelhante ao sistema de protecção de ficheiros, no sistema UNIX).

Quando uma pilha de mensagens é criada, é seguida uma sequência de regras, tendo em conta o estado do sistema e as *flags* disponibilizadas à primitiva de criação. Estas regras podem ser resumidas (Fig. 2):

- Especificando o bit de IPC_CREAT da flag é criado um novo canal de passagem de mensagens para a chave especificada, caso ainda não exista. Se já se encontrar um com a chave especificada, este é referenciado.
- Especificando os bits de IPC_CREAT e de IPC_EXCL na flag, é criado um canal de passagem de mensagens baseado na chave especificada, apenas se ainda não existir. Se já existir uma entrada com a chave indicada há ocorrência de erro.
- Especificando IPC_EXCL sem IPC_CREAT não tem qualquer sentido.

Tabela 1 - Resumo de primitivas utilizadas na passagem de mensagens.

Primitivas	Descrição
msgget	primitiva de criação ou abertura
msgctl	primitiva de controlo de operações
msgsnd	primitiva de operação de escrita
msgrcv	primitiva de operação de leitura

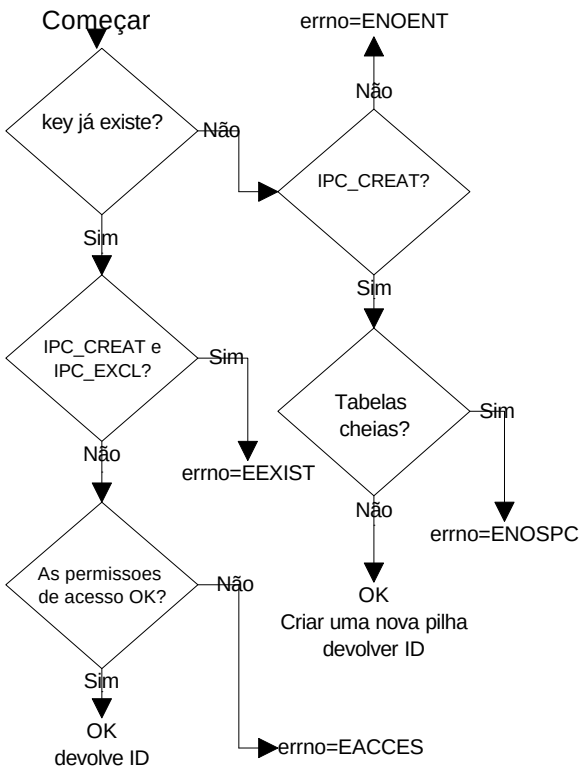

Fig. 2 - Fluxograma para o funcionamento de *msgget*.

Todas as mensagens são armazenadas pelo *Kernel* que utilizam um identificador, *msqid*, para referenciar uma determinada pilha. A forma de armazenamento assenta numa lista ligada, com uma estrutura de dados semelhante à seguinte, para cada pilha:

```
struct msqid_ds {
    struct ipc_perm msg_perm;
       /* Estrutura de permissões */
    struct msg *msg_first;
       /* Apontador para a 1. mensagem */
    struct msg *msg_last;
       /* Apontador para a última mensagem */
    ushort msg_cbytes;
       /* # de bytes existentes na pilha */
    ushort msg_qnum;
       /* # de mensagens na pilha */
    ushort msg_qbytes;
       /* # máximo de bytes da pilha */
    ushort msg_lspid;
       /* pid do último msgsnd */
    ushort msg_lrpid;
       /* pid do último msgrcv */
```

```
    time_t msg_stime;
        /* Hora do último msgsnd */
    time_t msg_rtime;
        /* Hora do último msgrcv */
    time_t msg_ctime;
        /* Hora do último msgctl */
};
```

Cada mensagem da pilha (Fig. 3) possui os atributos seguintes:

- *type* - inteiro tipo *long*;
- *length* - comprimento do conteúdo de dados da mensagem em bytes (pode ser zero);
- *data* - dados transmitidos (se *length* for maior que zero).

Uma vez aberto o canal de comunicação (criada a lista ligada com zero entradas), as mensagens podem ser colocadas na pilha utilizando a primitiva *msgsnd*:

```
int msgsnd(int msqid, struct msgbuf *ptr,
           int length, int flag);
```

O argumento *ptr* é um ponteiro para uma estrutura (definida em `<sys/msg.h>`) com a seguinte base:

```
struct msgbuf {
    long type;     /* Tipo da mensagem (>0) */
    char mtext[ ]; /* Dados */
};
```

O único campo com significado para o gestor de mensagens do *Kernel* e, portanto, de existência obrigatória, é o tipo da mensagem (*type*). O outro campo não tem qualquer tipo de restrição.

O argumento *length* (de *msgsnd*) especifica o comprimento da mensagem, excluindo o campo *type*.

O argumento *flag* pode ser IPC_NOWAIT ou zero. O valor IPC_NOWAIT permite que esta primitiva se torne não bloqueante, com devolução imediata se não houver espaço na pilha para a nova mensagem. Se não for indicado o valor IPC_NOWAIT a primitiva obriga o processo a bloquear até conseguir colocar a mensagem na lista ligada.

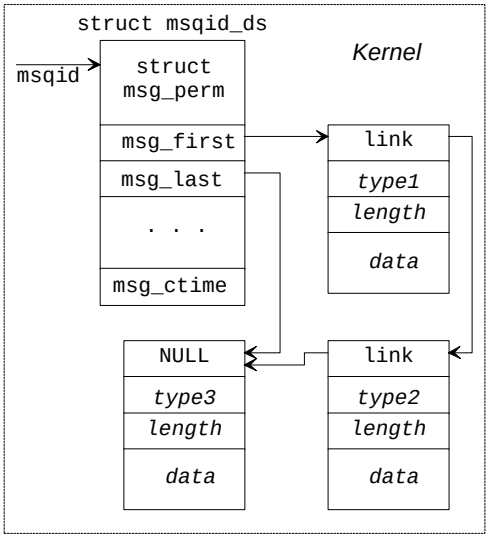

Fig. 3 - Pilha de mensagens existentes no *Kernel*.

Uma mensagem é retirada da lista usando a primitiva *msgrcv*:

```
int msgrcv(int msqid, struct msgbuf *ptr,
           int length, long msgtype, int flag);
```

Para além de alguns elementos comuns à operação de escrita, o argumento *msgtype* permite especificar qual a mensagem da pilha que se pretende retirar. Se for igual a zero é adoptada a filosofia FIFO (*First In, First Out*).

O valor de retorno de *msgrcv* representa o número de *bytes* de dados na mensagem (`ptr->mtext`).

Para remover a pilha de mensagens (canal de comunicação) do *Kernel* é utilizada a primitiva *msgctl*:

```
int msgctl(int msqid, int cmd,
           struct msqid_ds *buff);
```

Para o efeito, *cmd* terá de ser indicado como IPC_RMID.

B. Remote Procedure Call (RPC)

O mecanismo RPC [4] é, basicamente, um processo que permite a uma aplicação em execução num determinado computador, a utilização de procedimentos que são executados noutro computador, interligado fisicamente com o primeiro. A execução desse procedimento remoto processa-se como se ele fosse local, através da passagem de parâmetros e da recolha dos valores de retorno.

O conceito de mecanismo RPC é a base para um outro conceito mais alargado, que designaremos de sistema RPC, que compreende o conjunto das ferramentas e dispositivos que constituem a base de trabalho para a construção de aplicações baseadas em mecanismos RPC. Um sistema RPC, inclui, vulgarmente, um conjunto de funções para definição do protocolo de comunicação, um compilador de protocolo e ainda funções que permitem a gestão da comunicação.

A principal vantagem deste tipo de sistema é o facto de proporcionar o desenvolvimento de aplicações distribuídas, num conjunto de computadores, sem que seja necessário considerar os aspectos respeitantes à utilização da rede e outros relacionados com a incompatibilidade entre sistemas.

Os RPC baseiam-se no modelo cliente-servidor, em que o programa que faz o pedido de execução do procedimento remoto é o cliente e o programa que proporciona a execução desse procedimento é o servidor. O cliente e o servidor não comunicam directamente entre si, mas através de processos intermediários que gerem a comunicação. A estes processos intermediários dá-se o nome de *stubs* e têm por função gerir a interacção dos sistemas ou seja, a construção, a formatação e a troca das mensagens, que circulam entre o cliente e o servidor através do protocolo RPC (Fig. 4). Os *stubs* utilizam filtros que convertem os tipos de dados usados em cada sistema, em tipos reconhecíveis por ambos, resolvendo eventuais incompatibilidades na sua representação.

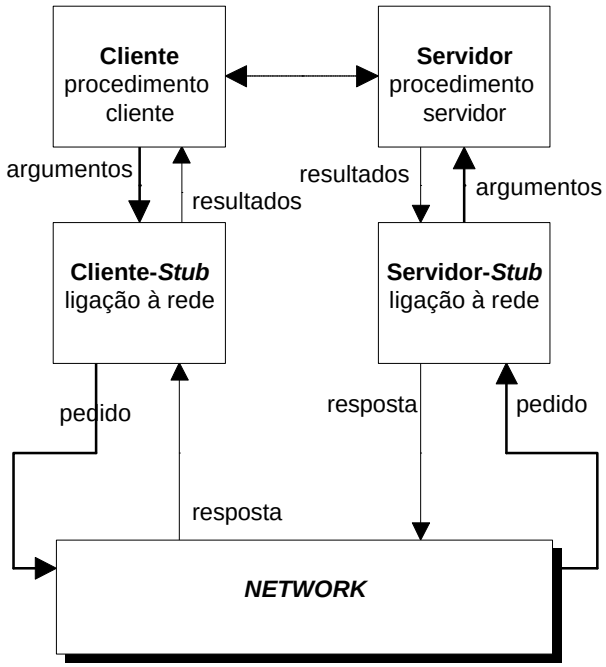

Fig. 4 - Chamada remota a um procedimento.

Consegue-se, portanto, que, do ponto de vista do programador, a utilização de procedimentos locais ou remotos seja virtualmente semelhante.

Presentemente existem dois sistemas predominantes de RPC, denominados, respectivamente, *Open Network Computing* (ONC) e *Network Computing System* (NCS). A breve apresentação de um sistema de RPC, que se segue, foca apenas o sistema ONC [4,5,6] (versão 4.0), visto que este é o mais divulgado e o único disponível em domínio público. No entanto, a filosofia e muitos dos métodos a seguir descritos, são também aplicáveis ao sistema NCS.

O sistema ONC permite a construção de aplicações distribuídas em dois níveis de desenvolvimento distintos, que denominaremos, genericamente, por alto e baixo nível. Do nível alto para o nível baixo, aumenta a complexidade da programação, aumentando simultaneamente, o grau de controlo e eficiência obtidos na aplicação. Convém realçar que estes dois níveis de programação, apesar de distintos, não são, de forma alguma, estanques. Pelo contrário, é muitas vezes desejável, que aplicações produzidas com primitivas de alto nível sejam posteriormente "afinadas" pelo programador, com primitivas de baixo nível, visando o aumento de eficiência da aplicação ou a introdução de funcionalidade que a alto nível não é possível obter.

O sistema ONC inclui um compilador de protocolo (RPCGEN) que, a partir da definição da interface entre o cliente e o servidor, gera automaticamente os *stubs* do cliente e do servidor, tornando assim mais expedito o desenvolvimento da aplicação. O programador pode refinar, posteriormente, o código gerado por este compilador.

Normalmente, o desenvolvimento de uma aplicação baseada num sistema RPC inicia-se pela definição da interface a usar na comunicação entre o cliente e o servidor. Esta definição consiste na identificação do procedimento remoto e na definição dos tipos de dados usados na comunicação.

Identificação do procedimento remoto

A identificação da função remota baseia-se na atribuição de identificações numéricas ao programa servidor a utilizar (identificações únicas em cada computador e limitadas a valores entre 0x20000000 e 0x3FFFFFFF), do procedimento propriamente dito e ainda da versão do procedimento, visto que o mesmo servidor pode manter várias versões distintas, da mesma função. Esta identificação será, posteriormente, utilizada tanto pelo cliente como pelo servidor para a troca de mensagens.

A definição seguinte é um exemplo típico:

```
#define MY_PROG    ((u_long) 0x20000000)
    /* Id do servidor */
#define MY_VERS    ((u_long) 1)
    /* Id da versão */
#define MY_PROC    ((u_long) 1)
    /* Id do procedimento */
```

Definição dos tipos de dados

Na definição da interface é essencial definir que tipos de dados devem circular entre o cliente e o servidor. Atendendo a que a representação interna dos diferentes tipos de dados varia de computador para computador, torna-se essencial definir os tipos dos dados de uma forma que seja compreensível por todos os computadores. Para atingir este fim o sistema RPC utiliza uma forma de representação dos tipos de dados, a que se dá o nome de *External Data Representation* (XDR) [6], que consiste num conjunto de rotinas que codificam e descodificam os dados usados localmente, segundo representações independentes.

Como exemplo apresenta-se um filtro que permite a utilização de *strings* como argumento ou valor de retorno de um procedimento:

```
#include <rpc/rpc.h>
bool_t xdr_filter(xdrs,objp)
XDR *xdrs;
char *objp;
{  return
   (xdr_string(xdrs,&objp,MAX_STRING_SIZE));
}
```

Neste exemplo, o argumento *xdrs* é utilizado para indicar se é uma codificação ou uma descodificação, o argumento *objp* é um ponteiro para a *string* de entrada ou de saída, conforme o valor de *xdrs*.

Para a transmissão de dados de tipos complexos como, por exemplo, as estruturas, é necessário construir filtros bidireccionais que procedem à sua decomposição e

recomposição, em tipos simples, e á respectiva codificação e descodificação. A construção deste tipo de filtros pode ser efectuado pelo compilador de protocolo (RPCGEN).

O passo seguinte, no desenvolvimento da aplicação, é a construção do procedimento de serviço remoto. Este procedimento deve ser construído como se de um procedimento local se tratasse, apenas seguindo a restrição de usar como parâmetros de entrada e valores de retorno, dados dos tipos definidos anteriormente.

A ultima tarefa a executar é a construção do cliente, do servidor e dos respectivos *stubs*, através de primitivas de alto ou de baixo nível. De seguida apresentam-se, de uma forma simplificada, algumas destas primitivas.

Implementação de alto nível

As primitivas de alto nível, fornecidas pelo sistema, são as seguintes:

```
registerrpc(...);
svc_run(...);
int callrpc(...);
```

A primitiva *registerrpc* informa o computador servidor que o procedimento especificado, existente no programa servidor especificado, está pronto a ser utilizado por aplicações remotas (ou locais). Os seus argumentos incluem a identificação do programa servidor, do procedimento e da versão a utilizar, bem como os filtros XDR.

A função *svc_run* coloca o programa servidor num ciclo de espera por solicitações de outras aplicações.

A primitiva *callrpc* é utilizada pelo programa cliente para executar um procedimento existente num programa servidor. Como argumentos inclui, para além dos utilizados pela função *registerrpc*, a identificação do sistema servidor e os parâmetros de passagem à função remota. A utilização desta primitiva pressupõe que o programa servidor já procedeu ao seu registo (*registerrpc*) e se encontra á espera de pedidos de execução de um procedimento (*svc_run*).

A construção de um sistema RPC com primitivas de alto nível implica, como já foi referido, limitações relativamente ao controlo que é possível obter sobre o sistema. Uma das limitações mais importantes resulta da utilização do protocolo UDP para estabelecer a comunicação entre o cliente e o servidor. Existem restrições ao nível da quantidade de informação que é possível transmitir em cada ligação e não garante fiabilidade na comunicação [7].

Implementação de baixo nível

O compilador RPCGEN [4] é normalmente utilizado para produzir aplicações baseadas em primitivas de baixo nível.

O RPCGEN necessita que o programador especifique a interface a utilizar entre o cliente e o servidor. Esta definição é realizada produzindo um ficheiro em que se definem os nomes dos procedimentos remotos, que se pretende que o servidor reconheça, e os tipos de dados, utilizados nos seus parâmetros e nos valores de retorno. Baseado nesta especificação o compilador gera os s*tubs* do cliente e do servidor (Fig. 5).

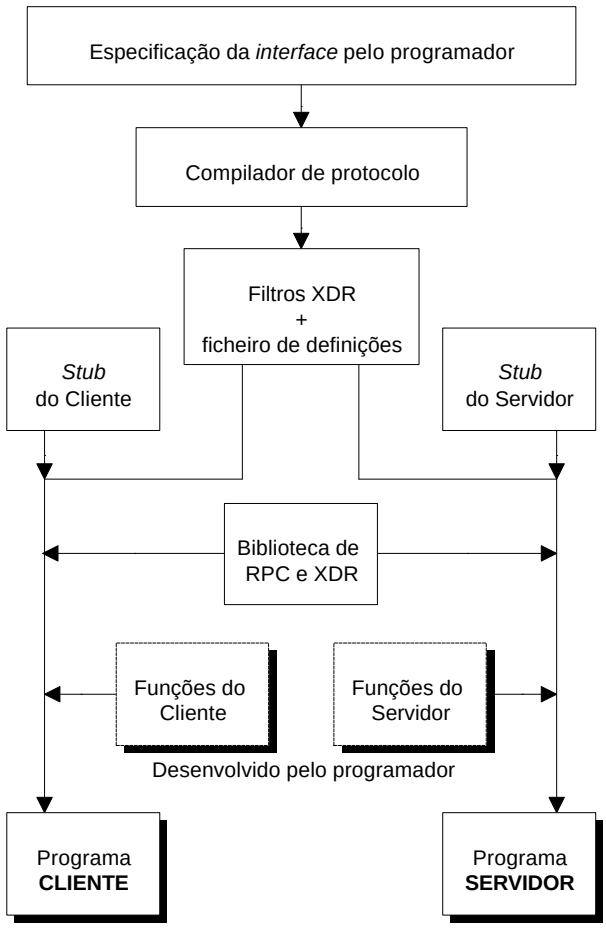

Fig. 5 - Desenvolvimento de aplicações utilizando o RPCGEN.

O compilador reconhece uma linguagem de descrição da interface (RPCL), organizada de modo similar à linguagem C e utilizando directivas de pré-processamento iguais às do *cpp*.

O compilador aceita seis tipos, diferentes, de definições:

- Constantes
- Enumerações
- Estruturas
- Uniões
- Tipos definidos pelo utilizador (*typedefs*)
- Programas

As enumerações, estruturas e *typedefs* são semelhantes às utilizadas em C, as constantes definem constantes, inteiras e simbólicas. As uniões são análogas aos registos variantes existentes em Pascal e finalmente as definições do tipo programa especificam os procedimentos que devem ser reconhecidos pelo servidor.

O RPCGEN reconhece ainda quatro tipos de declarações:

- Simples
- *Arrays* de tamanho fixo

- *Arrays* de tamanho variável
- Ponteiros

A sintaxe das declarações é, também, semelhante à utilizada em C.

Para terminar a apresentação do RPCGEN, é importante referir que a sua utilização implica que a interacção entre o cliente e o servidor não se faz por pedidos e respostas directas, mas sim através dos *stubs* produzidos. Ao interpretar a definição dos procedimentos disponíveis no servidor, descritos em RPCL, o RPCGEN atribui um nome para o procedimento (segundo uma regra bem definida) baseado no nome fornecido pelo programador na especificação da interface. Este identificador é utilizado simultaneamente para disponibilizar o procedimento (no servidor) e para invocar o respectivo serviço a partir do cliente.

As primitivas de baixo nível permitem um controlo mais eficaz dos diferentes aspectos do sistema, nomeadamente a escolha do protocolo de transporte (TCP ou UDP), a obtenção de informação respeitante ao cliente e ao servidor, a definição de temporizadores (*timeouts*) e dos processos de autenticação e encriptação [4].

Na construção de um sistema RPC, baseado neste tipo de primitivas, assume especial importância a existência de uma interface simples entre os *stubs* e os programas cliente e servidor. A importância desta simplicidade, tem por base o respeito pela filosofia deste tipo de sistema, no qual a chamada de um procedimento remoto deve ser virtualmente idêntica à chamada de um procedimento local.

Enumeram-se em seguida algumas das primitivas de utilização mais comum:

```
clnt_create(...);
clnt_destroy(...);
clnt_control(...);
clnt_call(...);
svctcp_create(...);
svcudp_create(...);
svc_register(...);
```

As primeiras três funções estão relacionadas com o desenvolvimento do cliente RPC.

A função *clnt_create* cria e inicializa as estruturas de dados respeitantes ao cliente e estabelece, ainda, a comunicação como o servidor. Os argumentos permitem especificar o sistema (*hostname*) e a aplicação servidores, o procedimento remoto, e a sua versão, e ainda o tipo de protocolo a utilizar na comunicação (UDP ou TCP). Esta primitiva deve ser chamada antes de qualquer outra acção que envolva o cliente.

A função *clnt_destroy* destroi a estrutura de dados criada para o cliente (*clnt_create*).

A primitiva *clnt_control* permite obter informações sobre as características do cliente ou actuar sobre estas de forma a modificá-las.

A função *clnt_call* faz a chamada efectiva do procedimento remoto. É possível explicitar o valor do *timeout* da solicitação. O uso desta primitiva implica, no mínimo, que já se tenha efectuado o *clnt_create* e que o servidor já esteja registado e pronto a aceitar pedidos.

Do lado do servidor existe um conjunto de serviços semelhantes.

A primitiva *svctcp_create* é utilizada para criar um serviço de transporte baseado no protocolo TCP e que será usado no servidor. Este tipo de serviço baseia-se *buffered I/O* pelo que permite definir o tamanho do *buffer* de recepção e de transmissão. Se estes valores não forem especificados serão usados valores por omissão que dependem do sistema.

A função *svcudp_create* é semelhante à primitiva *svctcp_create*, mas criando um serviço de transporte baseado no protocolo UDP.

A primitiva *svc_register* tem por função, registar a aplicação servidora no serviço *portmap*, do sistema servidor. Após o registo, o programa servidor passa a encontrar-se acessível a partir de outros computadores.

O sistema de RPC é, presentemente, um dos métodos de construção de sistemas distribuídos mais poderosos e flexíveis. Existem, actualmente, versões do sistema ONC que apresentam alguns melhoramentos relativamente á versão aqui apresentada, nomeadamente, um compilador de protocolo mais poderoso, melhores sistemas de segurança e ainda a possibilidade de construir aplicações independentes do protocolo de transporte.

III. MODELO GLOBAL DO SISTEMA DESENVOLVIDO

A apresentação anterior constitui um referencial das diferentes técnicas de programação de aplicações distribuídas, privilegiando os métodos RPC e passagem de parâmetros utilizados neste trabalho.

O modelo para partilha de informação entre processos distribuídos baseia-se em dois módulos cooperativos: um de comunicação local, assente em mensagens, e um outro para comunicações remotas, baseado em RPC.

A. Módulo de comunicação local

A arquitectura deste módulo de comunicação (Fig. 6), tem por base o método de passagem de mensagens apresentado anteriormente.

O elemento central deste modelo é um *router* que recebe e encaminha todas as mensagens. A transmissão e a recepção destas mensagens é da responsabilidade da unidade *MESG* que providencia os serviços de comunicação. Este objecto serve como uma *interface* entre qualquer processo comunicante e o sistema de passagem de mensagens. Os argumentos fornecidos a esta unidade (objecto) são o endereço do destinatário (sob a forma de um número inteiro) e os dados a enviar. Estes

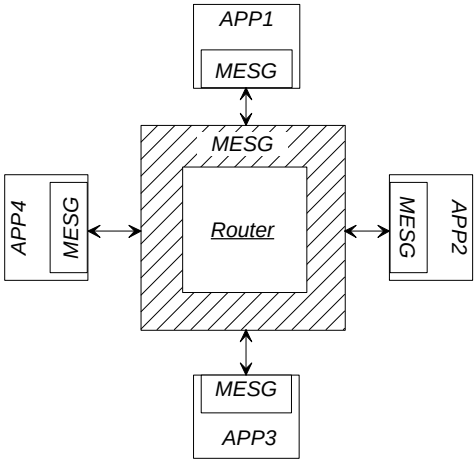

Fig. 6 - Esquema simplificado da arquitectura de comunicação.

argumentos são codificados numa trama (Fig. 7) e colocados na pilha de mensagens.

A trama corresponde à estrutura de dados seguinte:

```
typedef struct MSGTYPE
{
    int len;   //#bytes em data (0 ou > 0)
    long type; // tipo da mensagem (> 0)
    struct
    {
        Address dest;
        char data[MAXMESGDATA];
    } frame;
} Mesg;
```

Cada processo que necessite de enviar dados a um outro processo, deve informar o *Router* da sua presença na rede virtual. Do mesmo modo, deve indicar o seu abandono no momento de fecho da sua actividade. O *Router* poderá, no entanto, confirmar a presença dos diversos intervenientes em intervalos de tempo regulares.

Um exemplo de funcionamento do sistema com dois processos, é apresentado na figura 8.

B. Módulo de Comunicação Remota

O módulo de comunicações baseado em RPC tem por objectivo permitir a uma aplicação, em execução num determinado computador, a utilização de serviços prestados por aplicações executadas noutros computadores. Pretendia-se que este módulo fornecesse uma interface independente (do ponto de vista das aplicações locais e remotas) para os pedidos e para as respostas obtidas, tornando-se o mais geral possível, de modo a permitir a sua utilização por aplicações dos mais

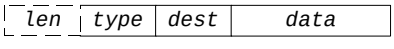

Fig. 7 - Formato da trama codificada por *MESG*.

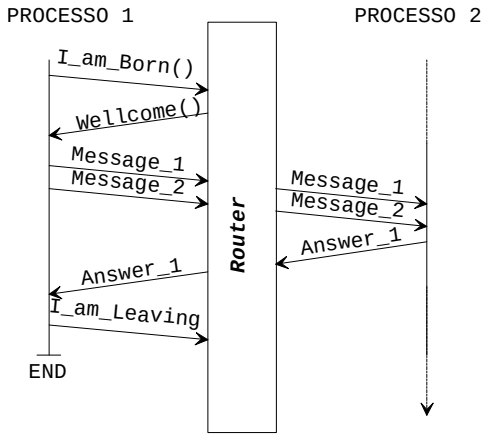

Fig. 8 - Exemplo de comunicação típico entre dois processos.

diversos tipos. Para obter esta generalidade, a resposta a um pedido é efectuada na forma de uma lista ligada de caracteres. Esta forma de representar a resposta, permite, quando necessário, o encapsulamento de qualquer tipo de dados, assim como permite, com facilidade, a recepção de resultados sob a forma de ficheiros. Esta última possibilidade revela-se extremamente útil devido ao facto de existirem muitas aplicações, que seriam, neste caso, as fornecedoras do serviço pedido, cuja execução resulta na criação de um ficheiro de dados.

No desenvolvimento deste sistema foi seguido um modelo que baseia a sua operação em dois canais distintos entre a máquina cliente e a máquina servidora. Um dos canais é utilizado na execução de pedidos e o outro para assinalar que os resultados já se encontram disponíveis (Fig. 9).

Quando os resultados das aplicações não colaborantes são disponibilizados sob a forma de ficheiros, o sistema funciona do seguinte modo:

1. A aplicação faz um pedido de execução de uma aplicação remota ao cliente local.
2. O cliente local transmite o nome dessa aplicação ao servidor existente na máquina remota. Antes de executar a transmissão do nome da aplicação, o cliente consulta uma base de dados local, para identificar a máquina a quem deve dirigir o pedido.
3. O servidor remoto inicia a execução do módulo de controlo de execução de aplicações, indicando-lhe qual a aplicação pretendida, colocando-se de seguida á espera de novos pedidos.
4. O módulo de controlo de execução de aplicações, procede á execução da aplicação pretendida, aguarda pelos resultados e coloca-os num ficheiro, se tal não for feito pela própria aplicação.
5. O módulo de controlo de execução de aplicações transmite ao cliente remoto o nome do ficheiro de resultados.
6. O cliente remoto transmite o nome do ficheiro de resultados ao servidor local.
7. O servidor local transmite ao cliente local o nome do ficheiro que contem os resultados.

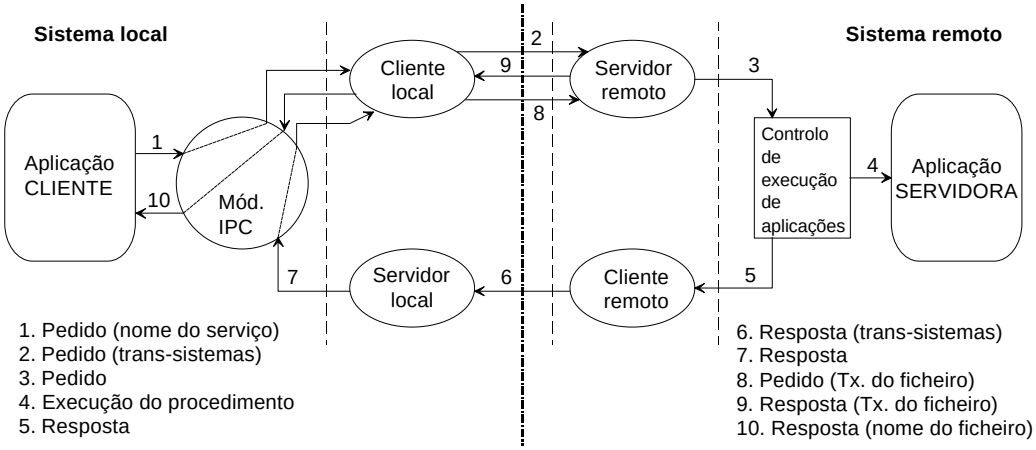

Fig. 9 - Implementação do sistema de comunicação remota.

8. O cliente local pede ao servidor remoto a transmissão do ficheiro de resultados.
9. O servidor remoto transmite o ficheiro de resultados ao cliente local.
10. O cliente local cria um ficheiro local de resultados, e transmite o nome desse ficheiro á aplicação que fez o pedido inicial.

A implementação apresentada não obedece ás regras normalmente seguidas num sistema deste tipo, visto que vulgarmente os pares cliente/servidor locais e remotos não se encontram sob a forma de programas distintos, mas sim integrados num único programa (Fig. 10).

A escolha deste sistema pouco usual, prende-se com a necessidade de fornecer um sistema que seja, simultaneamente, assíncrono e multi-tarefa. A primeira exigência está relacionada com o facto de a aplicação remota poder ter um tempo de execução longo, o que implica que o sistema remoto deveria receber o pedido, providenciar a sua satisfação e só após a sua conclusão enviar a resposta ao sistema que fez o pedido. A segunda exigência prende-se com a necessidade de garantir que um sistema possa facultar vários serviços simultaneamente, ou seja, após receber um pedido e antes de enviar o resultado, a máquina deve estar disponível para receber outros pedidos.

A principal vantagem da integração dos pares cliente/servidor num só módulo (Fig. 10-a), consiste no facto de a comunicação entre o sistema local e o remoto ser efectuado pela mesma entidade, quer para fazer os pedidos quer para receber as respostas o que, normalmente, é vantajoso, visto que tendo, pelo contrário, um servidor e um cliente seria necessário que o cliente estivesse permanentemente a verificar se a resposta já tinha chegado ao servidor. No entanto, neste caso específico essa vantagem deixa de existir, devido ao facto de ser utilizado, a nível local, um módulo de comunicação entre processos que permite ao próprio servidor informar o cliente da recepção da resposta esperada, evitando assim as verificações por parte do cliente. O sistema integrado tem também a vantagem de aproveitar melhor os recursos do sistema em que é desenvolvido, se bem que esta vantagem é relativamente reduzida.

As principais vantagens do modelo apresentado (Fig. 10-b), que levaram à sua escolha, baseiam-se na sua simplicidade, modularidade e facilidade de modificação, se tal for requerido no futuro. Em contrapartida o sistema integrado, implica uma complexidade no desenho dos módulos de comunicação varias vezes superior, o que implica que não possui nenhuma das características apontadas ao método de separação do pares cliente/servidor.

Pela avaliação feita das vantagens de cada um dos métodos concluiu-se que a poupança de recursos, no modelo integrado, era largamente excedida pelas vantagens do modelo de separação clientes e servidores.

C. Arquitectura global

A arquitectura proposta foi desenhada de modo a fornecer uma interface genérica para aplicações distribuídas, sejam elas locais ou remotas, interdependentes ou autónomas.

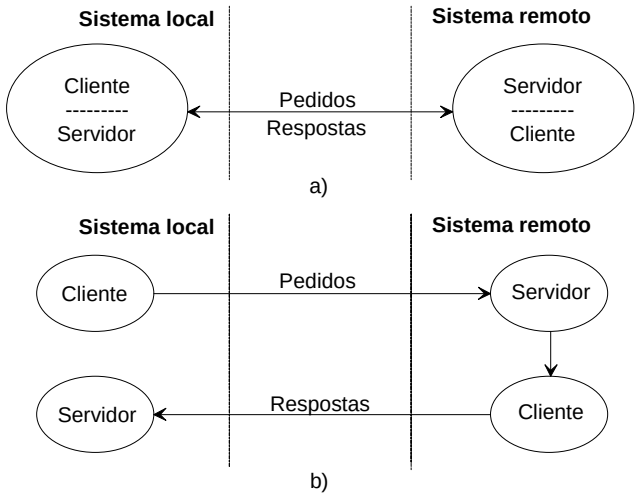

Fig. 10 - Ligação por RPC:
a) sistema clássico; b) sistema desenvolvido.

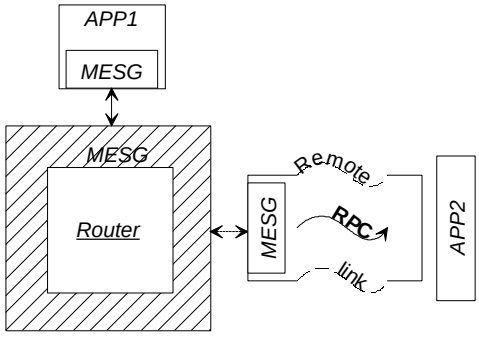

Fig. 11 - Arquitectura global de interligação entre processos.

A comunicação entre processos residentes em sistemas diferentes implica a utilização conjunta dos dois métodos apresentados (Fig. 11). Esta interligação de processos não requer o conhecimento, por parte de nenhum deles, do sistema remoto, mas apenas das funções de comunicação disponibilizadas pelos módulos de IPC e de RPC.

O processo cliente será, em princípio, desenvolvido na aplicação principal que distribui tarefas e recolhe informação. O processo servidor poderá assumir diversas formas (Fig. 12):

- local, construído de raiz e adaptado ao sistema (Fig. 12-a);
- local e modificado (por intermédio de um processo adaptador) de modo a ser compatível com o sistema (Fig. 12-b);
- remoto e modificado (por intermédio de um processo adaptador) de modo a ser compatível com o sistema (Fig. 12-c).

Para qualquer destas transacções, a comunicação é feita de uma forma completamente transparente para o utilizador, como de uma *system call* se tratasse. Na figura não é representado o *Router*, pois, do ponto de vista dos processos comunicantes, ele não interfere na comunicação.

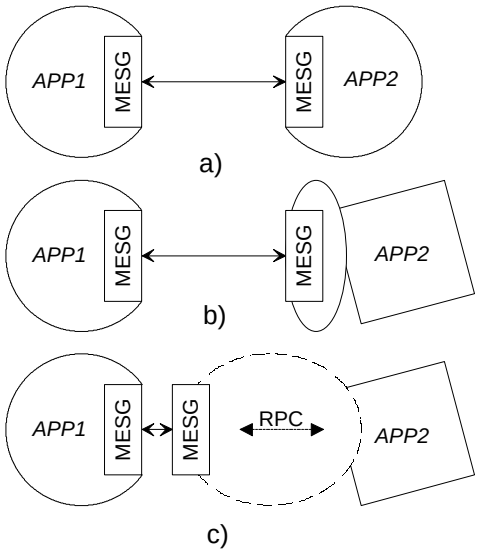

Fig. 12 - Comunicação entre diversos tipos de processos.

IV. DESENVOLVIMENTO DO SISTEMA

O sistema apresentado foi desenvolvido em sistemas UNIX e SunOS, segundo o paradigma da programação orientada para objectos (C++). Descrevem-se em seguida alguns destes objectos relacionados com algumas das unidades anteriormente apresentadas.

A. Unidade "Router"

O processo *Router* é um processo autónomo que faz a comutação de mensagens entre todos os processos registados. O *Router* é baseado num objecto tipo `Core`, classe que é especificada segundo o protótipo:

```
class Core
{
private:
  Mesg mesg;
  int id;
  Address up[MAXPROCESS];
  int n_process;
  void ToMe(int);
  void Route(int);
  void mesg_send();
  int mesg_recv();
  void erro(char*);
  int execute(char*);
  void write_up(char*);
  void delete_up(char*);
  int gone(char*);
  int get_out(char*);
public:
  Core();
  ~Core();
  void Exec(char*);
  void ReadBus();
};
```

Em termos de operação, este objecto principia por ler uma mensagem da pilha, descodifica-a e efectua o seu processamento, que varia consoante a informação sobre o destinatário:

- mensagem para ele próprio - interpretação através do método *ToMe*;
- mensagem vazia - despreza-a e passa à próxima;
- mensagem de difusão - envia uma cópia para cada processo registado;
- mensagem dirigida a um processo registado - a mensagem é retransmitida (método *Route*).

B. Unidade "MESG"

A integração de uma aplicação no modelo de processamento distribuído proposto, deve ser efectuada à custa da unidade *MESG*, responsável pelo encapsulamento dos mecanismos de comunicação. O

objecto que representa esta unidade conceptual é especificado segundo a classe `Bus`.

```
class Bus
{
private:
  int id;
  char* buffer;
  Mesg mesg;
  Address my_add;
  int total_size;
  void mesg_send();
  int mesg_recv();
  void erro(char*);
  int read_data(char*, char*);
  int write_data(char*, char*);
public:
  Bus(Address);
  ~Bus();
  void Send(Address, char*);
  char* Receive();
  int Execute(char*,char*);
  void I_am_Leaving();
  void I_am_Born();
  Address getDestAdd();
};
```

A classe fornece métodos para o registo de cada aplicação (*I_am_Born*), para receber e enviar mensagens para os processos registados (*Receive* e *Send*), para executar localmente uma determinada aplicação (*Execute*) e para registar o abandono de um determinado cliente.

V. CENÁRIOS DE APLICAÇÃO

O modelo de comunicação entre processos apresentado, foi desenvolvido no âmbito de um sistema de gestão de redes de dados. Este tipo de sistema tem uma necessidade particular de ferramentas de interligação, visto que a sua operação se baseia essencialmente num conjunto de aplicações autónomas distribuídas por diversos computadores ao longo da rede.

O sistema construído baseia-se numa aplicação central de processamento e gestão que coordena a operação de diversas aplicações, residentes em computadores distintos. Estas aplicações são autónomas e fornecem funcionalidades especificas, que no seu conjunto constituem o sistema de gestão. Estas funcionalidades abrangem áreas tão diversas como a monitorização de tráfego, detecção automática de topologia, contabilização e taxação de recursos e sistemas de controlo de segurança (Fig. 13).

A distribuição das aplicações autónomas tem por base, por um lado, a exigência de localização em pontos estratégicos da rede, e por outro lado, a falta de portabilidade de algumas aplicações desenhadas para arquitecturas computacionais particulares.

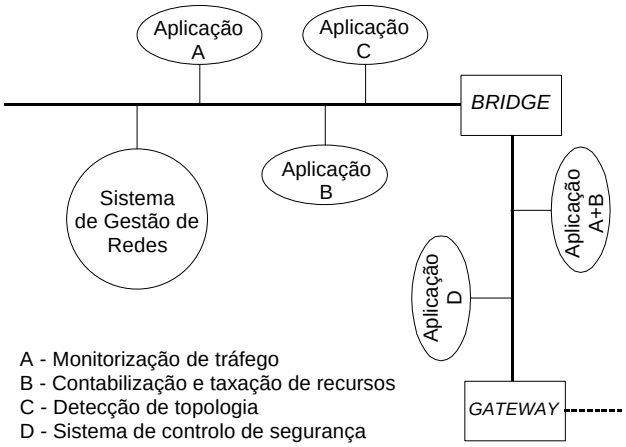

Fig. 13 - Comunicação entre diversos tipos de processos

VI. CONCLUSÕES

Foram apresentadas diversas técnicas para partilhar processos, distribuídos local ou remotamente em sistemas heterógeneos. Foi dado particular destaque ao método de passagem de mensagens (*IPC message queues*) como solução de distribuição local, e ao mecanismo de RPC (ONC RPC) pela suas potencialidades para distribuição de tarefas em diferentes sistemas.

Com base nestas duas técnicas foi proposto um modelo que permite o desenvolvimento de aplicações colaborantes e a integração de aplicações independentes e não colaborantes. Esta última facilidade explora a interface de entrada e saída de dados da aplicação para construir uma interface que adapta esta informação, segundo o modelo proposto.

Como cenário de utilização, foi apresentado um projecto de um Sistema de Gestão de Redes onde o modelo permite integrar aplicações específicas e autónomas, que enriquecem a informação disponibilizada, centralmente, pelo sistema de gestão

VII. REFERÊNCIAS

[1] W. Richard Stevens, *Advanced programming in the UNIX Environment*, Addison Wesley, 1992.

[2] W. Richard Stevens, *UNIX Network Programing*, Prentice-Hall, Englewood Cliffs, N. J., 1990

[3] Rochkind, M. J., Advanced UNIX Programming, Prentice-Hall, Englewood Cliffs, N. J., 1985.

[4] J. Bloomer, *Power Programming with RPC*, O'Reilly & Associates, Inc., 1992.

[5] Sun Microsystems, Inc, "RPC: Remote Procedure Call, Protocol Specification, version 2", *RFC 1014*, 1987.

[6] Sun Microsystems, Inc, "XDR: External Data Representation standard", *RFC 1057*, 1988.

[7] Douglas Comer, *Internetworking with TCP/IP - Principles, Protocol and Architecture*, Prentice-Hall, 1987.