Uma Experiência de Portabilidade do ANSAware para Ambiente Novell

Manoel Camillo Penna
Marcelo Ehalt
Mauro Sérgio Fonseca
Sergies Batista

Departamento de Informática
Centro Federal de Educação Tecnológica do Paraná
Av. Sete de Stembro, 3165 - 80230-901 - Curitiba, PR
E-mail: penna@dainf.cefetpr.br

Resumo

Este artigo apresenta o desenvolvimento de um sistema de passagem de mensagem *(Message Passing System* - MPS), baseado na interface de programação de aplicação *(Application Programming Interface* - API) da rede Novell 3.11. O MPS permite incorporar esta API à plataforma de sistemas distribuídos ANSAware, que fornece uma realização dos conceitos de processamento distribuído aberto (*Open Distributed Processing* - ODP). Inicialmente este trabalho descreve a plataforma ANSAware e as APIs UNIX 4.3 BSD e Novell 3.11. Em seguida, o projeto do MPS e detalhes de sua implementação são apresentados.

Abstract

This paper presents the development of a message passing system (MPS), based on the application programming interface (API) of Novell 3.11 network. The MPS alows this API to be incorporated to the ANSAware distributed computing platform, which provides an implementation of Open Distributed Processing (ODP) concepts. This work describes the ANSAware platform, as well as UNIX 4.3 BSD and Novell 3.11 APIs. Then, the design of the MPS and details of its implementation are presented.

1. INTRODUÇÃO

A tecnologia de sistemas informáticos distribuídos começa atingir sua maturação com o surgimento dos primeiros sistemas comerciais. Uma questão fundamental que se coloca neste contexto é a capacidade de inter-operabilidade destes sistemas. De fato, sistemas distribuídos devem ser, pela própria natureza, necessariamente inter-operantes. Neste sentido duas linhas de ação são identificadas, a formação de consórcio de fabricantes, com propostas de sistemas comuns ou regras de inter-operabilidade e a definição de padrões. Na primeira linha destacam-se o consórcio *Open Software Foundation* (OSF) com a linha de produtos "*Distributed Computing Environment*" (DCE) [1] e o consórcio *Object Management Group* (OMG) com as especificações "*The Common Object Request Broker Arquitecture*" (CORBA) [2]. No que tange padrões para sistemas distribuídos abertos, destacam-se os trabalhos realizados pela

International Standards Organization (ISO) para definição de um modelo de referência para o processamento distribuído aberto (*Open Distributed Processing* - ODP) [3, 4].

Paralelo ao trabalho de padronização, ressaltamos aquele desenvolvido pelo grupo ANSA *(Advanced Networked Systems Arquitecture)* [5], que se iniciaram no âmbito de um projeto do programa *Alvey* na Inglaterra, tendo se desdobrado a nível europeu no projeto *Integrated System Architecture* (ISA) do programa ESPRIT. Este trabalho tem contribuído de maneira decisiva para o desenvolvimento das normas ODP, sendo atualmente conduzido pela APM (*Ansa Project Management*). Um dos seus principais resultados é o lançamento de um produto denominado ANSAware [6], uma implementação de um produto em conformidade com as normas ODP. Observa-se que não existem ainda requisitos de conformidade completamente definidos para ODP, entretanto este software disponibiliza a maioria dos conceitos até então definidos no modelo de referência, constituindo-se em um banco de testes para sistemas ODP.

O presente trabalho foi realizado com base no sistema ANSAware. O objetivo é descrever a implementação de uma camada de software para portar este sistema para os protocolos usados em redes Novell. Como veremos, o ANSAware foi projetado de maneira bastante modular, com uma camada específica para isolar os diversos protocolos e suas **interfaces de programação de aplicação** (*Application Program Interface* - API) do núcleo do ANSAware. Isto indica a previsão deste tipo de desenvolvimento, o que era de se esperar, já que a missão fundamental do ANSAware é permitir o desenvolvimento de aplicações distribuídas em ambientes heterogêneos. O ambiente objetivo (Novell 3.11) foi escolhido devido a sua grande penetração no mercado e consequente disponibilização em um grande número de instalações.

A próxima seção apresenta o *software* ANSAware, objeto de motivação deste projeto, sendo focalizado os aspectos de suporte à comunicação, que interessam mais diretamente a este trabalho. As seções 3 e 4 descrevem resumidamente as duas APIs envolvidas, as sockets do UNIX 4.3 BSD e a API da Novell 3.1. O objetivo desta descrição é salientar as principais diferenças entre elas considerando-se os aspectos necessários ao desenvolvimento pretendido. Estas diferenças são discutidas na seção 5. Na seção 6 apresentamos a nossa solução. Particularmente são descritos os **sistemas de passagem de mensagem** (*Messge Passing System* - MPS) para dois protocolos de transporte, um do tipo datagrama e o outro do tipo circuito virtual. É apresentada ainda a solução para a compatibilização dos mecanismos de recepção assíncrona de mensagem, já que o escalonamento de tarefas na plataforma ANSAware é baseado em recepção assíncrona de mensagem. Finalmente, a seção 7 avalia a implementação realizada e apresenta as conclusões.

2. O SOFTWARE ANSAware

2.1. Introdução

Uma aplicação distribuída consiste em diversos programas, fisicamente espalhados em vários computadores, e que interagem entre si. Estes programas, no ambiente ANSAware, são codificados na linguagem C, seguindo o paradigma da programação orientada a objeto. Eles interagem segundo o modelo de invocação remota, no qual um objeto cliente solicita a realização de uma operação por um objeto servidor.

Para que o comportamento dos participantes, numa dada interação, produza os resultados desejados, há necessidade da definição de todas as possibilidades de cooperação entre estes participantes. O ANSAware provê, para esta finalidade, uma **linguagem de definição de interface** (*Interface Definition Language* - IDL). Esta linguagem permite a definição de uma interface, com todas as operações possíveis de serem realizadas através da mesma. Um **compilador de stubs** (*stub compiler* - stubc) também é fornecido, a fim de gerar o código que viabiliza a interação, quando os componentes estão distribuídos. Este código é chamado de *stub*. Somente dois tipos de invocação são permitidos: (i) A interrogação, na qual o objeto cliente espera até que o objeto servidor realize a operação e lhe retorne os resultados. (ii) O anúncio, no qual o objeto cliente prossegue com seu processamento, não aguardando a execução da operação pelo servidor.

Os componentes cliente e servidor são codificados na linguagem C, utilizam a interface definida e podem interagir por meio desta interface. A programação é realizada de modo declarativo, isto é, o programador possui uma maneira declarativa para definir as invocações. O ambiente ANSAware provê uma **linguagem incorporada à linguagem C**, a qual possui a sintaxe das declarações, e um **preprocessador** (PREPC), o qual converte os programas C com as declarações PREPC em programas C compiláveis.

Esta é, basicamente, a seqüência para geração de aplicativos no ANSAware: (i) Especifica-se a interface com todas as operações permitidas por ela. A especificação é compilada pelo stubc, gerando um stub para o cliente e outro para o servidor. (ii) Codificam-se os componentes cliente e servidor da aplicação, os quais são pré-processados pelo PREPC, gerando os códigos do cliente e do servidor. (iii) O stub do cliente mais o código do cliente e as funções de bibliotecas necessárias são compilados e ligados, gerando a cápsula do cliente. O mesmo procedimento é seguido para geração da cápsula do servidor. O termo cápsula é explicado na seção 2.3. A figura 1, extraída de [6], esclarece este procedimento.

2.2. O Modelo Computacional

Definindo-se a terminologia própria deste modelo, pode-se visualizar a idéia básica do mesmo. Um **serviço** é uma função que manipula informações, podendo processá-la, armazená-la ou transferí-la. Os componentes do aplicativo que utilizam serviços são chamados de **clientes**. Componentes que fornecem serviços são chamados **servidores**. Estes componentes são genericamente chamados de **objetos computacionais**. Um objeto pode ser cliente de um serviço e servidor de um outro ao mesmo tempo. Os serviços somente são providos em uma **interface**, embora os objetos possam ter mais de uma interface.

Alguns serviços são úteis a grande parte das aplicações, tais como os que oferecem a facilidade de gerência de nomes e de localização de servidores. Tais serviços são ditos arquiteturais. Neste ambiente, os objetos ***trader***, **fábrica (*factory*)** e o **gerente de nós (*node manager*)** são os provedores de tais serviços. O *trader* funciona como um diretório onde são registrados os serviços disponíveis. Os servidores exportam as referências a interfaces correspondentes a seus serviços, para o *trader*. Os clientes que necessitam de um serviço, importam do *trader* a referência a interface compatível com sua necessidade. Dada a necessidade de definição de mais termos, a fábrica e o gerente de nós são detalhados na seção 2.3. A maneira como os objetos cooperam é através das referências a interface. Estas referências são instâncias de uma determinada interface. Somente quando um cliente está de

posse de uma referência a interface é que lhe é permitido invocar operações fornecidas pelo servidor nesta interface.

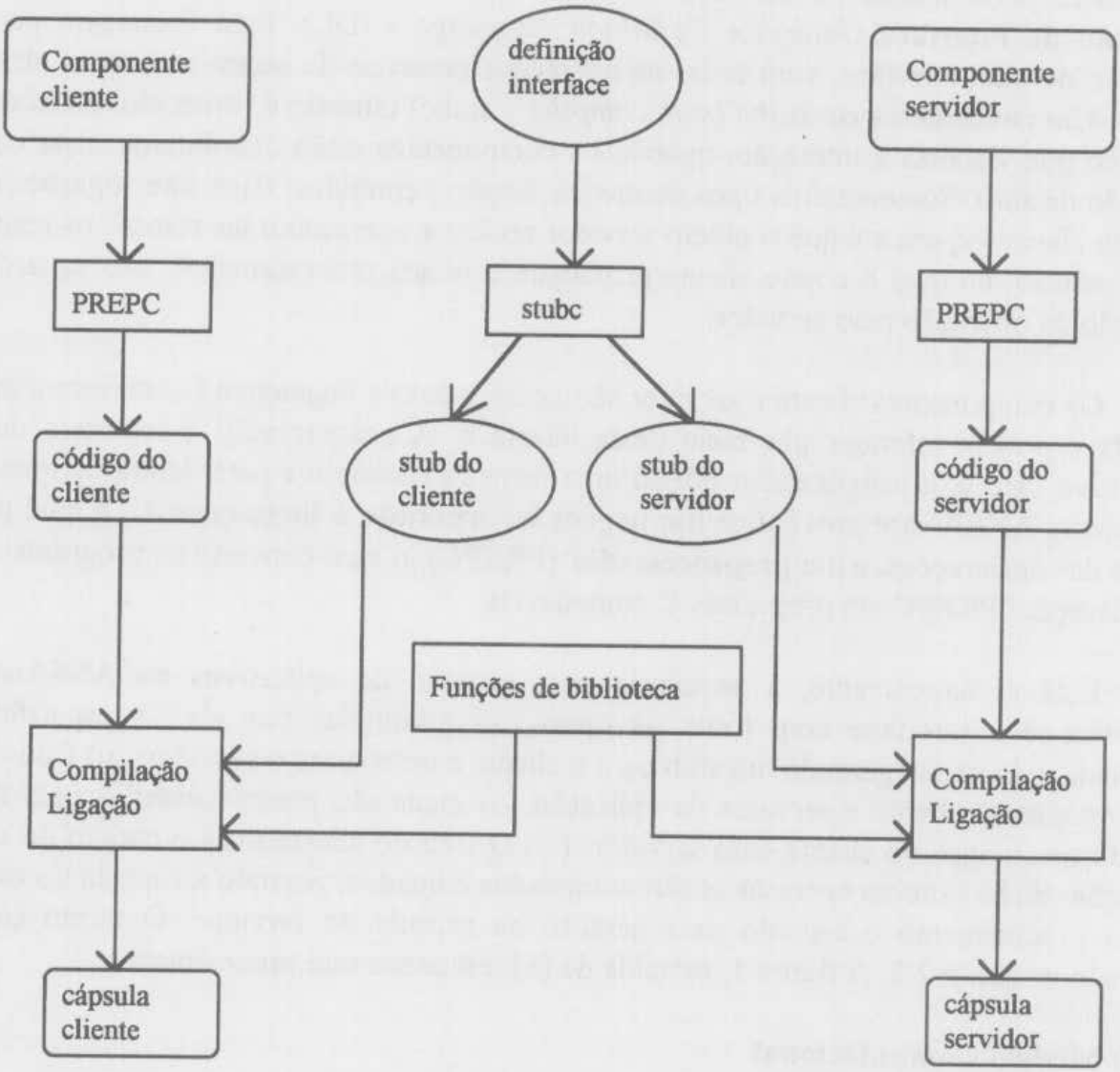

Figura 1 - Processo de geração de aplicativos no ANSAware

No tocante a quesitos de transparência, a versão 4.1 do ANSAware implementa apenas as transparências de acesso (que permite uma estilo de interação uniforme, a despeito do objeto ser local ou remoto) e de localização (que permite a interação com um objeto sem referência a sua localização física).

2.3. O Modelo de Engenharia

O modelo de engenharia realiza um mapeamento entre os componentes do modelo computacional e as estruturas de engenharia capazes de prover-lhe suporte.

A noção de **nó** (***node***) corresponde a um conjunto visto como uma unidade. Por exemplo, um nó pode ser um computador pessoal ou uma rede de computadores gerenciada por um sistema operacional distribuído. Cada nó possui recursos (*tasks*, *threads*, contadores de eventos (*eventcounts*), seqüenciadores (*sequencers*), *sockets*, *plugs*, canais e referências a interfaces). O **núcleo** é o objeto de engenharia responsável pela gestão destes recursos. Uma

cápsula é um objeto de engenharia cujo conceito corresponde a um processo, a um espaço de endereçamento. O programador constrói cápsulas que se comunicam. Os recursos são atribuídos às cápsulas para que estas possam realizar o processamento.

Um conjunto de objetos computacionais compilados é chamado de **objeto de engenharia**. Objetos de engenharia não possuem atributos de transparência. São adicionadas operações aos objetos computacionais, quando compilados, de modo que estes possam interagir com **serviços de transparência**, os quais proporcionam o grau de transparência desejado. Este trabalho é realizado pelo PREPC. A figura 2, extraída de [7], detalha estes conceitos.

Um objeto de engenharia corresponde à menor unidade que pode ser manipulada, isto é, distribuída, ativada, desativada ou enviada para outra localização (migração). Objetos de engenharia comunicam-se entre si através do núcleo.

Os recursos providos pelo núcleo são definidos, de forma sucinta, abaixo:
(i) Uma ***thread*** corresponde a uma unidade de atividade numa cápsula. *Threads* podem compartilhar estruturas de dados e podem se sincronizar;
(ii) Uma ***task*** é um processador virtual que provê os recursos para a execução da *thread;* portanto, os recursos são alocados às *tasks*. A *thread*, para ser executada, necessita estar ligada a uma *task*. O número de *tasks* em uma cápsula determina o número de *threads* que podem ser executadas simultaneamente;
(iii) **Contadores de eventos** e **seqüenciadores** são os mecanismos providos para sincronizar *threads*;
(iv) Uma ***socket*** corresponde ao endereço para invocação de operações em um servidor. A *socket* é encapsulada juntamente com a referência a uma interface. O ***plug*** corresponde ao ponto de acesso à interface, pelo cliente. O caminho formado pelo conjunto *plug* + *socket* é chamado de **canal;**
(v) **referências a interfaces** são as estruturas de dados que identificam uma instância de uma interface. Todas as cápsulas possuem uma referência a interface conhecida, a qual indica o *trader*. Deste modo, as cápsulas podem obter referências para as outras interfaces, para execução de operações.

Alguns serviços são providos pelo ambiente, conforme já mencionado. A função da **fábrica** é criar (e destruir), dinamicamente, objetos de engenharia. Existe uma fábrica para criação de cápsulas. Outra fábrica, na cápsula, permite a criação de objetos dentro da cápsula. Cada objeto, por sua vez, conta com uma fábrica para criação de instâncias de interfaces. De modo simplificado, uma cápsula é composta por zero ou mais objetos; os objetos contêm zero ou mais interfaces; as interfaces possibilitam a execução de zero ou mais operações. O **gerente de nós** administra os serviços presentes em cada nó. Necessita da fábrica para criação de objetos, os quais proverão os serviços. Portanto, cada nó deve contar com uma fábrica e um gerente de nós.

Salienta-se ainda que o ANSAware pode ser construído tendo por base vários sistemas operacionais, a saber: UNIX, VMS e MS-DOS/Windows. Cada ambiente destes representa um modelo tecnológico diverso.

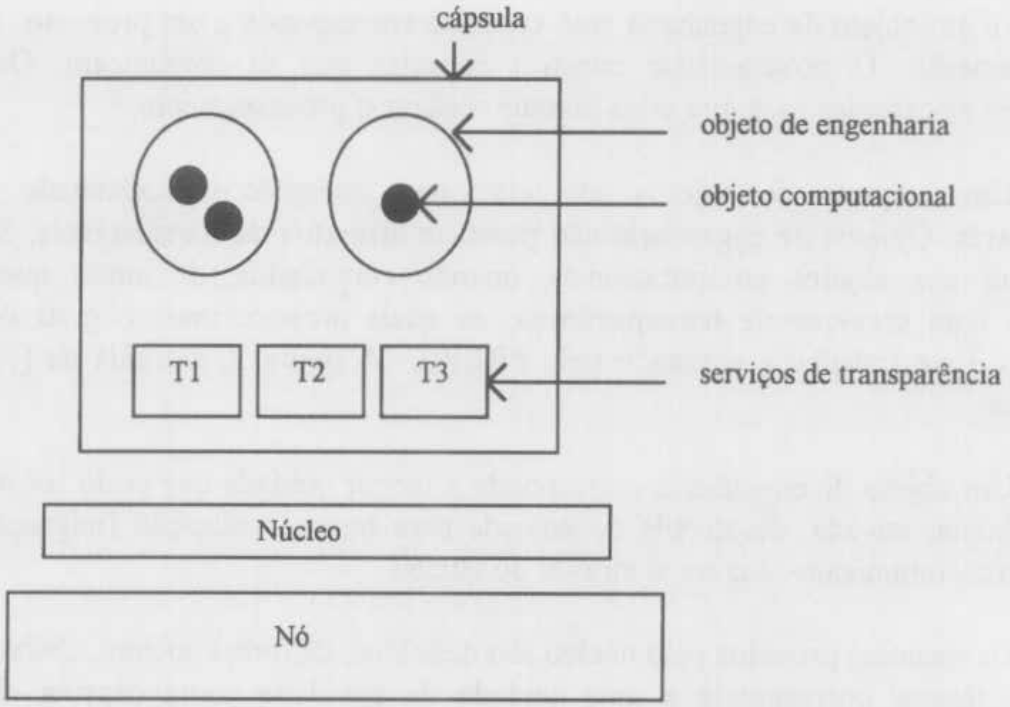

Figura 2 - Detalhamento do modelo de engenharia

A decomposição do ANSAware visando a incorporação de múltiplas APIs inclui uma camada de software denominada **sistema de passagem de mensagem** (*Message Passing System* - MPS), que implementa uma interface única para os protocolos de chamada de procedimentos remotos (*Remote Procedure Call* - RPC) do ANSAware[1]. Assim sendo, são encapsulados juntamente com o núcleo, os protocolos de RPC mais um conjunto de funções de acesso aos protocolos subjacentes, isto é, os MPSs. Somente os MPSs suportados são compilados e ligados ao núcleo (através de compilação condicional). Os endereços das funções que compõem um MPS são carregados em estruturas, que por sua vez são armazenadas em um vetor. Os protocolos de RPC acessam a MPS de modo transparente através deste vetor, tornando-se assim independente da implementação subjacente. A figura 3 ilustra a organização dos componentes de comunicação do ANSAware.

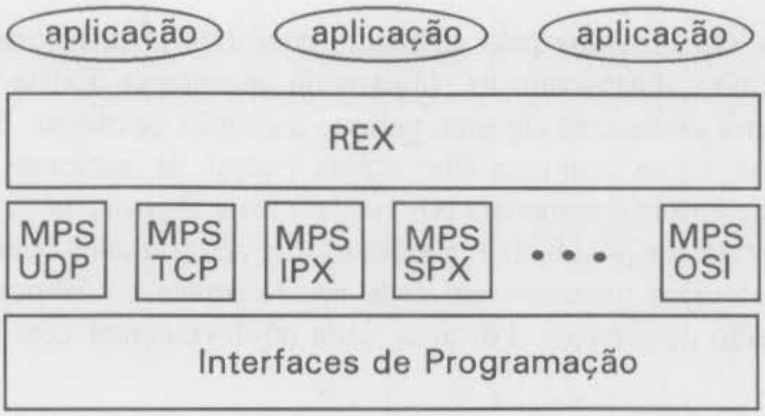

Figura 3 - Componentes de comunicação do ANSAware

[1] Atualmente estão implementados dois protocolos: O protocolo REX [6] (*Remote Execution Protocol*) implementa um protocolo clássico de execução remota e o protocolo GEX [7] (*Group Execution Protocol*) implementa um protocolo de execução remota para grupos. Este projeto aplica-se somente ao primeiro.

A implementação ANSAware apresenta vários dos conceitos introduzidos nas regras ODP, o que a torna um excelente modelo para novas pesquisas, dada sua compatibilidade com a padronização.

3. API UNIX 4.3 BSD

3.1. Introdução

Neste trabalho, denominamos API a interface disponível ao programador que pretende usar protocolos de comunicação. A API do UNIX 4.3 BSD [8], oferece ao programador uma abstração do canal de comunicação e respectivos protocolos, que é conhecida como *socket*. As *sockets* provêm o suporte para a comunicação entre processos que executam em um mesmo sistema e/ou em sistemas diferentes, tendo aparecido pela primeira vez na versão 4.1 BSD para o VAX, em 1982. É possível utilizar *sockets* para interfacear aplicativos com pilhas de protocolos diferentes, embora neste trabalho, nos interessemos apenas pelos protocolos da arquitetura Internet.

Os dados que trafegam pela rede de comunicação devem, ao chegar ao destino, ser entregue ao aplicativos corretamente. No caso da arquitetura Internet, o sistema deve ser capaz de multiplexar dados de aplicativos diferentes para um dos dois protocolos de transporte disponíveis, UDP ou TCP. Do outro lado, é realizado o processo contrário de demultiplexação. O protocolo da camada de rede (IP) usa então um identificador numérico denominado número do protocolo (*protocol number*) para identificar o protocolo de transporte. Os protocolos de transporte usam outro identificador numérico denomidado número de porta (*port number*) para identificar aplicações. A combinação de um endereço IP e um número de porta é a materialização de uma *socket*, que identifica univocamente um processo dentro da rede Internet. Um par de *sockets*, um para a máquina que recebe e um para a máquina que transmite, define um canal de comunicação.

3.2. Estruturas de Dados

Uma API é caracterizada pelas estruturas de dados que devem ser manipuladas e pelo conjunto de funções disponíveis para a programação da troca de informação. A estrutura de dados fundamental na API UNIX 4.3 BSD, é uma estrutura que define um endereço de processo para uma *socket*. As definições desta estrutura estão no arquivo <sys/socket.h>:

```
struct sockaddr{      u_short sa_family;      /* address family: AF_xxx  value */
                      char sa_data[14];       /* up to 14 bytes of protocol specific address */
              };
```

O conteúdo dos 14 bytes do endereco são interpretados de acordo com o tipo de endereço. Para a arquitetura Internet, as estruturas estão definidas em <netinet/in.h> como segue:

```
struct sockaddr _in {  short sin_family;          /*AF_INET  */
                       u_short sin_port;          /* 16-bit port number */
                       struct in_addr sin_addr;   /* 32-bit netid/hostid */
                       char sin_zero[8];          /* unused */
                    };
```

Esta estrutura de endereço, depois de corretamente processada por uma sequência de chamadas de sistema, retorna ao programador um descritor de arquivo (*file descriptor*), que será manipulado no programa, de maneira análoga a qualquer outro descritor convencional.

3.3. Recepção Assíncrona de Mensagens

A recepção assíncrona de mensagens é um ponto de particular relevância para este trabalho, pois como veremos, influencia o escalonamento de tarefas no ANSAware. Uma das maneiras de implementá-la, quando usamos a API UNIX 4.3 BSD, é através da chamada de sistema *select*. O mecanismo de escalonamento de *threads* do ANSAware é baseado nesta chamada de sistema, implementando o que se denomina de um mecanismo de *upcall* [9]. O escalonador interroga o sistema (através do *select*) para determinar quais *sockets* têm dados a receber. Para aquelas *sockets* que têm dados a receber, o escalonador aciona a primitiva de recepção do protocolo de mais alto nível (REX neste caso), que termina por acionar a primitiva de recepção do MPS correspondente, que por sua vez lerá os dados na *socket*. Este mecanismo evita o bloqueio no momento de recepção, já que a leitura será acionada apenas quando existirem dados a serem recebidos. A chamada de sistema *select* manipula um conjunto de descritores de arquivo (uma *socket* também é um descritor de arquivo), conforme descrito a seguir.

O primeiro argumento especifica a faixa dos descritores de arquivos que serão examinados. Os três argumentos seguintes são respectivamente, um ponteiro para o conjunto de descritores de arquivo a serem lidos pelo chamador, outro ponteiro para o conjunto de descritores para os quais tenham sido solicitada uma gravação e outro para condições excepcionais. Caso o usuário não esteja interessado em uma destas funções basta passar um ponteiro nulo no argumento correspondente. Cada conjunto é uma estrutura contendo um vetor de inteiros longos, que corresponde a uma máscara de bits onde os bits ligados indicam os descritores de arquivo que devem ser examinados.

O tamanho do vetor é indicado pelo ponteiro **FD_SETSIZE***. As macros **FD_SET**(*fd,&mask*)* e **FD_CLR**(*fd,&mask*)* servem para incluir e remover um descritor de arquivo do conjunto *mask*. O conjunto *mask* deve ser zerado através da macro **FD_ZERO**(*&mask*). O último argumento (*timeout*) também é um ponteiro para uma estrutura do tipo *timeval*, que especifica o valor de um intervalo de tempo máximo, antes do qual a chamada ao *select* retornar. Se a estrutura apontada por *timeout* estiver zerada, significa esta deve retornar imediatamente. Se o argumento for um ponteiro nulo, significa que a chamada deve bloquear até que algum descritor contenha dados a serem lidos, que tenham concluído uma gravação, ou que estejam em condição de pendência especial.

A função *select* normalmente retorna o número de descritores de arquivos selecionados. O valor retornado é zero, quando ela termina por estouro do temporizador. O valor retornado é -1, quando ela termina devido a um erro, sendo que a máscara de descritores de arquivos permanece inalterada. Se a função termina com sucesso, os três conjuntos indicarão quais descritores de arquivos estão prontos para serem lidos, escritos, ou tenham condições excepcionais pendentes. O estado de um descritor de arquivos dentro de uma máscara selecionada pode ser testado com a macro **FD_ISSET**(*fd, &mask)**, que retorna não zero se o descritor é um membro do conjunto *mask*, e zero, caso contrário.

A descrição detalhada do funcionamento das demais funcionalidades da API UNIX 4.3 BSD está fora do escopo deste trabalho. O texto desta seção discutiu apenas os tópicos necessários à compreensão do desenvolvimento realizado.

4. API da NOVELL 3.11

4.1. Introdução

As funções do serviço de comunicação da interface de programação de aplicação (API) da Novell [10], habilitam um aplicativo executando em uma estação de trabalho, a transmitir ou receber mensagens, para ou de outras estações de trabalho, através dos protocolos de comunicacão IPX ou SPX. Nesta API, todos os eventos, como recebimento ou transmissão de uma mensagem, são controlados por uma estrutura de dados chamada *Event Control Block* (ECB). Os diversos campos da ECB definem completamante a operação de comunicação a ser realizada. Por exemplo, antes de uma aplicação transmitir uma mensagem, campos apropriados da ECB apontam para áreas de memória contendo o cabeçalho do pacote a ser transmitido e para o *buffer* contendo a informação. Igualmente, quando um aplicativo deseja receber uma mensagem, o aplicativo deve preparar uma ECB que contenha o endereço de um *buffer* para armazenar o seu conteúdo, incluindo o cabeçalho mais a informação. O preenchimento de alguns campos do cabeçalho deve ser feito pelo programador que usa as funções da API.

Tipicamente, uma aplicação passa o endereço da ECB como argumento para a função de comunicação IPX ou SPX. Quando a ECB é submetida para processamento, o campo *InUseFlag* da ECB é posicionado para um valor não nulo, indicando que a ECB não se encontra disponível para outra operacão. Depois de realizar a operação, um campo com código de retorno é posicionado para um valor apropriado e o *flag* de uso é reposicionado para zero, indicando que a ECB está novamente disponível para o uso.

4.2. Estruturas de Dados

A construção de programas usando a API da Novell requer ao programador o conhecimento da estrutura dos cabeçalhos do protocolo subjacente, tendo que fornecer uma área de memória onde este será montado. É necessário ainda, o preenchimento de determinados campos do cabeçalho e da ECB. Segue portanto uma descrição resumida dos mesmos.

4.2.1. O Protocolo IPX

O IPX é um protocolo definido pela Xerox e utilizado nas redes Novell para a comunicação em redes com baixa taxa de erros de comunicação. O IPX é um protocolo do tipo datagrama, isto é, não necessita de estabelecimento de conexões para realizar a comunicação. Cada pacote é considerado como uma entidade individual, não havendo garantia de sequenciamento entre pacotes distintos, ou garantia de entrega de um pacote. Como não existe o estabelecimento de conexão, cada pacote IPX deve conter os endereços de origem e de destino.

A estrutura de um pacote IPX é idêntica a estrutura de um pacote *Xerox Network Standard* (XNS). O pacote consiste de 30 octetos de cabeçalho e seguidos de 0 a 546 octetos

dados. O tamanho mínimo e máximo de um pacote são 30 e 576 octetos respectivamente. O conteúdo dos octetos de dados são de responsabilidade do aplicativo podendo assumir qualquer formato. O arquivo NXT.H contém as definições das estruturas em C necessárias para o uso da API. Os campos do cabeçalho são definidos na estrutura IPXHeader que consiste dos seguintes campos:

```
WORD        CheckSum          /* high-low */
WORD        Lenght            /* high-low */
BYTE        TransportControl
BYTE        PacketType        /* set on send */
IPXAdress   Destination       /* set on send* /
IPXAdress   Source
```

Os seguintes campos devem ser preenchidos pela aplicação: *Packet Type*, com o valor 4 (*Datagram Protocol Packet*) e *Destination*, quando a operação desejada for o a transmissão de um pacote. A estrutura IPXAdress é definida no arquivo NTX.H como segue:

```
BYTE        network[4]        /*high-low*/
BYTE        node[6]           /*high-low*/
BYTE        socket[2]         /*high-low*/
```

O campo *network* identifica o endereço da rede de destino ou de origem, dependendo do caso. O campo *node* identifica o endereço da estação (por exemplo, o endereço da placa ethernet), e o campo *socket*[2], o número da porta de comunicação.

O protocolo IPX executa as tarefas da camada de rede (Nível 3) do modelo OSI. Estas tarefas incluem endereçamento, roteamento e encaminhamento de pacotes de uma rede para outra. A API da Novell oferece os serviços do IPX a nível de transporte, isto é, endereçamento fim-a-fim das portas de comunicação.

4.2.2. O Protocolo SPX

O SPX é um protocolo definido pela Xerox e utilizado nas redes Novell para a comunicação em redes com alta taxa de erros de comunicação O SPX é um protocolo de comunicação orientado a conexão, isto é, antes que um pacote SPX possa ser transmitido, uma conexão entre o remetente e o receptor deve ser estabelecida. O SPX executa as tarefas necessárias para garantir a entrega e a sequência de pacotes, detectando e corrigindo erros e eliminando a duplicação de pacotes. Em contra partida a estas garantias, o SPX tem menor performance do que o IPX.

O pacote SPX é semelhante ao pacote IPX, contendo 12 octetos adicionais no cabeçalho. O pacote consiste de duas partes: o cabeçalho com 42 octetos e a parte dos dados que pode ter de 0 até 534 octetos. O tamanho mínimo do pacote é 42 octetos (o cabeçalho) e

[2] O termo *socket* nas duas APIs referem-se a conceitos distintos. Uma *socket* do UNIX 4.3 BSD corresponderia a um ECB da API da Novell, enquanto que uma *socket* da API da Novell corresponderia a um número de porta na API UNIX 4.3 BSD.

o tamanho máximo é 576 octetos. Os campos do pacote SPX, desde o campo *CheckSum* até o campo *Source* têm exatamente o mesmo significado que os campos do pacote IPX.

Os campos preenchidos são os mesmos que para o caso do IPX: *Packet type*, com o valor 5 (*Sequenced Packet*) e *Destination*, quando a operação desejada for a transmissão de um pacote. O endereço de destino só precisa ser fornecido para o estabelecimento da conexão.

4.2.3. Event Control Block

Uma ECB é uma estrutura de dados cujos campos contêm informações usadas para transmitir ou receber pacotes IPX ou SPX. Visto que uma ECB pode controlar a transmissão ou o recebimento de pacotes, ela é as vezes referida como uma ECB de transmissão ou uma ECB de recepção, dependendo da sua função. A ECB de transmissão e a ECB de recepção correspondem à mesma estrutura, a diferença é que a ECB de transmissão precisa de um endereço destino no campo *Immediate Address* enquanto a ECB de recepção não. Uma ECB é composta de 2 partes: uma parte de tamanho fixo de 36 octetos e uma parte de dados que varia com o número de *buffers* associados a ela. A seguinte tabela mostra os campos de uma ECB, seguida da descrição resumida dos principais campos envolvidos na programação:

Deslocamento	Conteúdo	Tipo	Ordenação
0	Link Address	BYTE[4]	offset-segment
4	ESR Address	BYTE[4]	offset-segment
8	In Use Flag	BYTE	
9	Completion Code	BYTE	
10	Socket Number	WORD	high-low
12	IPX Workspace	BYTE[4]	
16	Driver Workspace	BYTE[12]	
28	Immediate Address	BYTE[6]	
34	Fragment Count	WORD	low-high
36	Fragment Address 1	BYTE[4]	offset-segment
40	Fragment Size 1	BYTE[2]	
42	Fragment Address 2	BYTE[4]	
46	Fragment Size 2	BYTE[2]	

ESR Address: Contém o endereço da rotina ESR a ser chamada quando um evento de transmissão ou de recepção se completa (ver ítem 4.3).

In Use Flag: Contém um valor diferente de zero enquanto a API estiver usando a ECB. Quando uma solicitação se completa, a API reposiciona este campo em zero.

Completion Code: Indica o estado final da operação solicitada. Não pode ser considerado válido até que a API reposicione o campo *In Use Flag* para zero.

Socket Number: Contém o número da *socket* com a qual esta ECB é associada.

Immediate Address: Contém o endereço do nó parceiro, sendo o endereço do primeiro/último roteador da rede local, se o pacote não foi transmitido/recebido de um nó na mesma sub-rede.

Fragment Count: Contém o número de fragmentos que compõem a mensagem.

Fragment Descriptor: É um vetor de ponteiros para as áreas de memória que compõem a mensagem a ser enviada ou recebida. É dividido em dois campos: *Address*, contendo o endereço do *buffer* que contém o fragmento e *Size*, contendo o tamanho do fragmento.

4.3 Recepção Assíncrona de Mensagens

O processamento assíncrono de um evento é realizado na API da Novell por um procedimento denominado *Event Service Routine* (ESR). Um ESR é um procedimento escrito pelo programador e identificado na ECB, para ser chamado após o término de uma operação. Os ESRs são sempre invocados após a conclusão de um evento. Neste momento, o campo *In Use Flag* já foi reposicionado para zero, o campo *Completion Code* já contém o código de retorno relativo ao evento, e todas as informações associadas ao evento já estão disponíveis. Uma ECB aponta para o ESR que deve ser chamado após a ocorrência de um evento. Quando o controle passa para o ESR, este recebe um ponteiro para a ECB que corresponde ao evento.

Um ESR é um procedimento de interrupção e como tal é chamado com o *flag* de interrupção desabilitado. Portanto, um ESR deve ser escrito para executar o mais rápido possível, não realizando tarefas que poderiam ser realizadas pela estrutura principal do aplicativo. As seguintes condições são verdadeiras quando um ESR é invocado: o ponteiro da ECB associada está contido no par de registradores ES:SI; todos os registradores exceto SS e SP ja foram salvos na pilha do usuário; e as interrupções estão desabilitadas. O ESR não deve retornar nada.

5. COMPARAÇÃO DAS DUAS APIs

Nesta seção é realizada uma breve comparação entre as duas APIs relativa aos aspectos relevantes aos objetivos deste trabalho. Não pretendemos realizar um estudo comparativo para indicar os melhores aspectos desta ou daquela API, mas indicar as diferenças, e as implicações no desenvolvimento deste projeto. Ambas APIs disponibilizam um conjunto de funções para realizar as tarefas de comunicação, onde as principais diferenças são:

- mesmas primitivas para comunicação com conexão e sem conexão (UNIX) versus primitivas diferenciadas (Novell);
- estruturas de dados para o controle das operações de comunicação;
- estrutura de endereçamento;
- visibilidade dos campos do protocolo;
- o tratamento assíncrono.

As primitivas diferenciadas não afetaram de modo apreciável o desenvolvimento dos MPSs, pois sempre foi possível realizar um mapeamento entre as primitivas correspondentes. O conjunto de primitivas das duas APIs pode ser considerado equivalente. As estruturas de dados para o controle das comunicações são bastante diferentes, como pôde ser visto nas seções 3.2 e 4.2. Entretanto a compatibilização também não foi complexa a este nível, pois ambas APIs oferecem estas estruturas como tipos de dados pré-definidos, que podem ser fácilmente incluídos no texto dos programas.

A estrutura de endereçamento das duas arquiteturas subjacentes também é diferente. Na arquitetura Internet, o endereço de rede é completamente desvinculado do endereçamento das camadas inferiores, enquanto que na arquitetura XNS, o endereço de rede é uma combinação do endereço da sub-rede mais o endereço do nodo, que nada mais é que o endereço da placa acoplada à estação. A segunda opção "fere" os princípios da independência entre as camadas, podendo entretanto ser mais eficiente. Analogamente, na API da Novell, o cabeçalho dos pacotes IPX e SPX é visível a nível de interface de programação, enquanto que na API UNIX 4.3 BSD, este aspecto é completamente transparente. Isto indica claramente que a API da Novell provê uma interface de mais baixo nível, onde o programador tem que estar consciente de detalhes do processo de comunicação. Estas dificuldades de mapeamento foram superadas, no entanto, com relativa facilidade.

O último ítem de diferenciação foi aquele que mais trouxe dificuldade no desenvolvimento deste trabalho, já que a diferença entre ambos é bastante significativa, inclusive na abordagem conceitual. A maneira usual de se implementar o processamento assíncrono na API UNIX 4.3 BSD é através da chamada de sistema do UNIX que monitora o estado de descritores de arquivo (*select*). Isto reflete a uma propriedade interessante do sistema UNIX, onde toda entrada e saída tem um tratamento bastante semelhante para o programador, sendo manipulada por um descritor de arquivo. A API transforma o canal de comunicação (um par de *sockets*) em um descritor de arquivo, que pode ser monitorado pela chamada de sistema *select*. Este procedimento esta calcado na execução de uma monitoração por parte do programa e seria adequadamente classificado como uma técnica de *polling*. Por outro lado, o processamento assíncrono oferecido na API da Novell é totalmente baseado em um esquema de interrupções.

A grande dificuldade relativa a esta diferenciação ocorre porque o recebimento de mensagens no ANSAware é realizado com base no esquema assíncrono do UNIX. Em outras palavras, o escalonador do ANSAware toma decisões baseado no retorno da chamada de sistema *select*. Foi necessário o desenvolvimento de um mecanismo equivalente para a API da Novell, descrito na seção 6.2.3.

6. ESPECIFICAÇÃO DO SOFTWARE

6.1. Descrição

O ANSAware implementa dois protocolos de execução remota, um para interações RPC convencionais e outro para interações em grupo. O protocolo convencional é denominado REX, o qual é construído sobre uma interface de programação bem definida, composta por um conjunto de funções. Para cada interface de comunicação diferente, é definido um MPS, que fornece uma interface padrão entre o protocolo REX e uma API particular. O MPS realiza o tratamento da heterogeneidade da comunicação no ANSAware, isto é, um MPS deve construir, sobre uma interface qualquer, uma interface padrão para o REX. Um mapa indica quais MPSs estão carregados. Cada entrada do mapa contém o endereço dos pontos de entrada de cada função MPS requerida. Todo módulo MPS, para qualquer API, é construído pelo conjunto das cinco funções que seguem:

MPS_cleanup() - desabilita o MPS, devendo realizar todas as tarefas necessárias para o término da mesma, por exemplo, fechar as *sockets* ou cancelar as solicitações em andamento.

MPS_tick - é chamada periodicamente pelo REX iniciando uma pesquisa para verificar a existência de conexões abertas ociosas, que devem então ser liberadas, evitando que uma porta de comunicação fique ociosa por um grande período, enquanto outras esperam por conexão.

MPS_startup - inicializa o MPS, sendo chamada durante a inicialização do programa. Suas principais funções são garantir que o MPS esteja funcionando; adquirir um único endereço para o par processo/MPS e alocar os recursos locais necessários.

MPS_sendMsg() - transmite uma mensagem usando a API subjacente.

MPS_receiveMsg() - recebe uma mensagem usando a API subjacente.

O esqueleto programa abaixo mostra o vetor que contém os endereços da tabela dos pontos de entrada para cada função MPS requerida. O REX usa este vetor para acionar uma função particular de uma MPS específica.

```
typedef struct{ansa_Status (*mpsStartup) (  ansa_CapsuleAdr *self,
                                            ansa_Cardinal *extra
                                          );
            ansa_Status (*mpsSendMsg)( char *pkt,
                                       ansa_Cardinal psize,
                                       ansa_CapsuleAdr *remcap
                                     );
            ansa_Cardinal (*mpsReceiveMsg)(   ansa_BufferLink *blp,
                                              ansa_Cardinal psize,
                                              ansa_CapsuleAdr *remcap
                                           );
            void (*mpsTick)(void);
            void (*mpsCleanup)(void);
            ansa_Cardinal maxsendUnit;
            ansa_Cardinal maxReceiveUnit;
      } mpsTable;
mpsTable *mpsMap[];
```

6.2. Detalhamento

6.2.1. O MPS IPX

A seguinte estrutura foi definida para implementar o MPS para o protocolo IPX:

```
struct estrutura {      char flag;
                        char bufferIn[MAXRECEIVEUNIT];
                        char receive[12];
                        unsigned int tamanho;
} structIn[MAXBUFFERIN];
```

A estrutura *structIn* é um vetor de *buffers* utilizado para receber os pacotes IPX. O campo *flag* indica se o *buffer* está em uso; o campo *bufferIn* armazena os dados recebidos; o

campo *receive* contém o endereço de quem mandou a mensagem; e o campo *tamanho* contém o tamanho da mensagem recebida. Esta estrutura é manipulada por um procedimento ESR denominado *ProcessReceivedData*, o qual, quando acionado, armazena os pacotes IPX recém recebidos em uma entrada disponível. Quando o vetor de *buffers* está cheio, a mensagem recebida é descartada pelo MPS, cabendo ao protocolo superior (REX) a retransmissão.

Na recepção de mensagens, o mecanismo de *upcall* (ver seção 3.3) atua do seguinte modo. O procedimento ESR recebe a mensagem salvando-a no vetor *structIn*, marcando o flag respectivo para indicar que a mensagem foi recebida mas ainda não processada. O escalonador utiliza a função *select* para determinar se existem mensagens recebidas nas portas de comunicação. Implementamos uma versão particular da função *select* para este MPS, que baseia-se no campo *flag* de *structIn* para retornar as portas de comunicação que têm mensagens de entrada pendente (ver seção 6.2.3). O REX é acionado, invocando por sua vez a função de interface *MPS_receive*, que recupera a mensagem apropriada no vetor *structIn*, repassando-a então ao REX.

Foram reescritas as funções de interface com o REX: *MPS_startup*, *MPS_send*, *MPS_receive* e *MPS_cleanup*. O motivo destas alterações é que todas elas usam alguma funcionalidade específica do MPS subjacente.

6.2.2. O MPS SPX

As seguintes estruturas são utilizadas em um MPS para um protocolo com conexão:

Pod - A estrutura pod é usada como um *overlay* para endereços de processo. No campo p_adr fica o endereço propriamente dito. O vetor de octetos definido no campo p_addr é suficientemente longo para conter um endereço Internet ou XNS.

```
struct pod {    ansa_Cardinal p_len;
                Byte          p_adr[(int) POD_ADR_LEN];
                PortID        p_prt;
                MsgLen        p_eml;
} Pod;
```

CATable - É uma tabela *hash*, cuja chave é o número da porta de comunicação e cujo dado é o endereço do processo (uma estrutura do tipo pod).

VCCache - Esta estrutura implementa um *cache* de circuitos virtuais, mantido em uma lista duplamente encadeada. Cada entrada da lista contém um circuito virtual aberto cujos dados ficam armazenados no campo vc_pod, que tem o tipo da estrutura definida acima. O vetor VCList contém as entradas propriamenta ditas. VCList[0] é a cabeça da lista. O ponteiro vc_lru, dentro de cada elemento do vetor, aponta para a próxima entrada menos recentemente utilizada, enquanto que o ponteiro vc_mru, aponta para a mais recentemente utilizada. A única singularidade é que na cabeça da lista, o ponteiro vc_mru aponta para a entrada menos recentemente utilizada (LRU), e o ponteiro vc_lru, aponta para a entrada mais recentemente utilizada. São definidas as variáveis TheLRUEntry e TheMRUEntry, para se acessar diretamente estes dois elementos.

```
struct vccache {
        struct vccache *VC_lru;
        struct vccache *vc_mru
        struct pod vc_pod;
        asnsa_Cardinal vc_num;  /* número de pacotes enviados ou recebidos */
        ansa_Cardinal vc_use;   /* flag para indicar se a entrada foi usada no último tick */
        ....                    /* campo específico à MPS para descrever a conexão */
} VCCache;

VCCache VCList[MAX_ENTRIES];
```

As seguintes funções internas manipulam o vetor VCList, de modo a implementar o algorítmo LRU na gestão das conexões em andamento:

- mps_VCCacheInitialize( ) - inicializa o *cache*;
- mps_CATableInitialize( ) - inicializa a tabela *hash*;
- mps_unlinkEntry( ) - retira uma entrada do *cache*;
- mps_linkMRUEntry( ) - inclui uma entrada como a mais recentemente utilizada;
- mps_linkLRUEntry( ) - inclui uma entrada como a menos recentemente usada;
- mps_getFreeEntry( ) - localiza uma entrada não utilizada;
- mps_getEntryByPin( ) - pesquisa otimizada (*hash*) de uma entrada baseada na porta de comunicação;
- mps_getEntryByAddress( ) - localiza uma entrada pelo endereço;

Estas funções foram quase totalmente reaproveitadas porque as funções são opacas às estruturas de informação que elas manipulam. A função getEntryByPin necessitou de pequena modificação, já que implementa no MPS-TCP, uma pesquisa onde a chave é o descritor de arquivo relativo à conexão. Na versão MP-SPX, a chave é a ECB que descreve a conexão. A função getFreeEntry foi parcialmente reescrita porque ao se procurar uma entrada livre, se ela não existir, esta função termina a conexão da entrada menos recentemente usada. O término de uma conexão é evidentemente diferente nas duas APIs.

As seguintes funções de apoio foram totalmente reescritas: mps_connect, mps_accept, mps_close, mps_disconnect, mps_drain. Também foram reescritas as funções de interface com o REX: MPS_startup, MPS_send, MPS_receive e MPS_cleanup. O motivo destas alterações é que todas elas usam alguma funcionalidade específica do MPS subjacente.

6.2.3. Recepção Assíncrona de Mensagens

Para se realizar a integração foi necessário compilar os fontes do ANSAware com as bibliotecas MPS-IPX e MPS-SPX, tendo sido necessárias algumas modificações internas no ANSAware. Inicialmente, foi feito um estudo para verificar se o endereçamento não afetaria o funcionamento das camadas superiores ao MPS, já que os endereços dos protocolos IPX/SPX são mais longos que os endereços dos protocolos UDP/TCP. Após o estudo concluimos que a estrutura de endereço do REX era superdimencionada, e que o REX não manipulava diretamente os dados ali colocados. Assim sendo, o único local que precisava efetivamente ser alterado era o mecanismo que realiza o escalonamento de processos a partir das mensagens recebidas. Como já vimos, este mecanismo é baseado na chamada de sistema *select* do UNIX, não é disponivel no ambiente Novell.

Foi implementado então um mecanismo para simular o efeito da função *select*, através de um procedimento ESR, associado a cada ECB de recepção. Este procedimento armazena as mensagens recebidas em uma lista circular. Cada entrada na lista tem um *flag* associado à existência ou não de uma mensagem disponível para leitura. No escalonador do ANSAware, a invocação de *select* é substituída por uma função que examina estes *flags*. Também foi implementada uma versão da macro **FD_ISSET** (ver seção 3.3). Ambas retornam um número positivo quando existe uma mensagem disponível para leitura.

Outra alteração relevante diz respeito à maneira como o processo, ao qual é solicitado o estabelecimento de uma conexão, completa esta solicitação. O mecanismo de *upcall* descrito em 3.3 é usado, no MPS-TCP, da seguinte forma. Quando um pedido de conexão chega ao processo, evidentemente o descritor de arquivo que está em escuta (*listen*), é marcado pelo *select* como tendo dados a receber. Isto terminará por acionar o protocolo REX que vai preparar um pacote para completar o processo de ligação (*binding*) do mecanismo RPC, acionando então a operação MPS_send no MPS apropriado. Neste momento a conexão ainda não está estabelecida, isto é, o MPS-TCP ainda não executou a função *accept* da API. O MPS-TCP server-se então de um *flag* que indica que esta é a primeira mensagem neste descritor, para saber que precisa invocar o *accept* antes de transmitir a menssagem. A chamada da função *accept* retorna um descritor novo que será usado a partir de então para aquela conexão. No MPS-SPX este procedimento foi implementado através de um procedimento ESR, associado à ECB que está em escuta (*listen*). Quando o pedido de estabelecimento de conexão chega ao processo, o procediment ESR é acionado, realizando as tarefas anteriores, dispensando o artifício do *flag* para a primeira mensagem usado no MPS-TCP.

7. AVALIAÇÃO E CONCLUSÃO

Este trabalho apresentou um projeto de desenvolvimento de sistemas de passagem de mensagem (MPSs) visando acoplar os protocolos IPX e SPX da API da Novell 3.11 à plataforma de desenvolvimento de aplicações distribuídas ANSAware. Ambos MPSs apresentados nas seções anteriores foram implementados, conforme descrito.

Alguns experimentos de avaliação foram realizados, não tendo sido detectadas variações de performance entre as chamadas de procedimentos remotos, quando comparamos os MPSs desenvolvidos com aqueles já existentes para os protocolos TCP e UDP. Acreditamos que a razão para isso é que, embora o desenvolvimento tenha sido bastante complexo devido à tecnologia envolvida, o número de linhas de código de um MPS é tipicamente pequeno (1500 linhas de código C no nosso caso para os dois MPSs). A continuação natural deste projeto se dará com a investigação de como a inclusão de outro protocolo de execução remota, perticularmente o GEX, afetaria os MPSs desenvolvidos para, então, e adequá-los caso adaptações venham a se fazer necessárias.

Este artigo descreveu um projeto de desenvolvimento inédito (já existiam MPSs para UDP, TCP e OSI), cuja relevância se justifica pela importância do tema e pela ampla penetração dos ambientes envolvidos. A implantação destes MPSs no ANSAware amplia o domínio de heterogeneidade abrangidos por esta plataforma, e além disso, o processo de desenvolvimento nos possibilitou o domínio de uma técnica importante para o tratamento de heterogeneidade em ambientes distribuídos.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Rosenberry, W.; Kenney, D.; Fisher, G.
"Understanding DCE", O'Reilly & Associates, Inc., 1993.

[2] Object Management Group
"The Common Object Request Broker: Architecture and Specification", OMG Document Number 91.12.1, December 1991.

[3] International Standrds Organization
"Open Distributed Processing: Basic Reference Model", ISO 10746, 1994.

[4] Tschammer et all.
"Processamento Distribuído Aberto e o Modelo RM-ODP-ISO", 11º-Simpósio Brasileiro de Redes de Computadores, 10 a 13 de maio de 1993, Universidade Estadual de Campinas, Campinas, SP.

[5] Advanced Networked Systems Architecture
"ANSA Reference Model", Release 01.00, Vol. A, March 1989.

[6] Architecture Projects Management
"ANSAware 4.1: Application Programming in ANSAware", Document RM.102.02, February 1993.

[7] Architecture Projects Management
"ANSAware 4.1: System Programming in ANSAware", Document RM.102.02, February 1993.

[8] Stevens, R.
"Unix Network Programming", Prentice-Hall, 1990.

[9] Clark, D.
"The Structuring of Systems Using Upcalls", ACM Operating Systems Review, Vol. 19, No.5, December 1985, pp. 171-180.

[10] Novell Inc. "Novell C Interface for DOS", Volumes I & II, 1990.