Extensão do sistema operativo OS9 a um ambiente de multiprocessamento

Ernesto F. V. Martins e António Nunes da Cruz

Resumo **- Este artigo apresenta uma sistema de comunicação inter-processos baseado no modelo de passagem de mensagens que desenvolvemos para um multiprocessador.**
Tomando como base o kernel do sistema operativo OS9 da Microware, residente em cada um dos processadores da máquina, criou-se uma extensão que gere a transferência de mensagens entre processadores e que se apresenta do ponto de vista dos processos de utilizador com um interface de comunicação global através do qual estes podem trocar mensagens duma forma transparente. A implementação baseia-se em canais e circuitos virtuais e as primitivas de comunicação podem funcionar síncrona ou assincronamente, em transferências ponto a ponto ou 'multicast'.

Abstract **- This paper describes an interprocess communication system built around the message passing model which we have developped for our bus-based multiprocessor.**
Starting from an OS9 operating system kernel already up and running in each processor, we designed an extension, made up of an integrated set of system modules, which manages message communication between processors and provides to application processes a global communication interface through which they can exchange messages in a transparent way. The implementation uses channels and virtual circuits and communication primitives support synchronous and asynchronous message transfer and group communication.

I. INTRODUÇÃO

Com o crescente interesse que se tem verificado recentemente nas arquitecturas baseadas em múltiplos processadores, têm surgido no mercado uma variedade de sistemas operativos (SOs) direcionados quer para os sistemas de memória partilhada quer para os sistemas do tipo multicomputador. O SPOX-MP e o Chorus são dois destes exemplos - o primeiro vocacionado para arquitecturas 'tightly-coupled' e o segundo mais adaptado aos sistemas do tipo rede local (LAN) [1].

Capitalizando em produtos já desenvolvidos, outros fabricantes passaram a disponibilizar extensões de multiprocessamento para os seus SOs multitask mono-processador. O VxWorks por exemplo, possui uma extensão para arquitecturas de memória partilhada. A extensão do MQX/MPX permite a sua utilização em sistemas paralelos, principalmente os do tipo 'loosely-coupled'. O PSOS por sua vez, dispõe de múltiplas opções de configuração que permitem a sua utilização numa gama muito variada de arquitecturas multiprocessador [2].

Dependendo da plataforma de desenvolvimento e dos requisitos da aplicação em vista, adoptar uma destas extensões em detrimento dum verdadeiro SO distribuído ou de multiprocessamento, é por vezes a melhor solução apesar das desvantagens relativas. Uma delas é evidentemente o menor desempenho dos mecanismos de comunicação inter-processos. Outra, não menos importante, têm a haver com a transparência e com a coerência dos interfaces típica dos SO distribuídos, mas que é dificíl de conseguir nos SO melhorados com extensões. A principal vantagem das extensões é que constituem a solução mais económica no 'upgrade' dum ambiente convencional (mono-processador) para um multiprocessador. Mesmo que nenhuma extensão do SO utilizado seja adequada para o multiprocessador em questão, é muitas vezes preferível desenvolver uma extensão mínima do que mudar para um novo SO.

Neste artigo apresentamos um exemplo da aplicação prática desta última abordagem, mais concretamente, uma extensão de multiprocessamento que desenvolvemos para o SO OS9 [3]. O multiprocessador 'target' é constituído basicamente por um conjunto de processadores autónomos ligados a um bus paralelo comum, através do qual estes comunicam via mensagens.

A extensão baseia-se também num modelo de comunicação por mensagens e, na presente versão, implementa apenas a função mais importante dum SO multiprocessador: a comunicação transparente entre processos. Esta realiza-se recorrendo a *ligações* (*circuitos virtuais*) e a *canais* de recepção de mensagens. As primitivas colocadas à disposição dos processos funcionam em modo *síncrono* ou *assíncrono* e suportam a *comunicação em grupo*.

A secção seguinte apresenta o multiprocessador (mais rigorosamente, multicomputador) para o qual esta extensão foi desenvolvida do ponto de vista apenas das estruturas físicas de comunicação interprocessador que suportam a implementação da extensão. A secção III inclui uma discussão prévia sobre as opções fundamentais do modelo de passagem de mensagens e descreve a sua implementação. Finalmente na última secção avaliamos o desempenho da extensão de multiprocessamento.

II. Multiprocessador e Sistema de Passagem de Mensagens

A extensão do OS9 aqui apresentada foi desenvolvida para um sistema multiprocessador que pode ter até 16 processadores ligados a um bus Gespac G-96+. Os processadores são baseados no micro MC68020, incluem memória e I/O locais, e um interface, o MPIF (Message Passing Interface), que lhes permite trocar mensagens com outros processadores do sistema. O conjunto de todos os MPIFs ligados através do bus partilhado constitui o Sistema de Passagem de Mensagens, ou MPS, do multiprocessador (Fig.1).

Cada MPIF é constituído por dois circuitos independentes: o Emissor de Mensagens (MS) e o Receptor de Mensagens (MR). Cada um destes inclui uma memória FIFO para armazenamento temporário das mensagens. As mensagens são pacotes de palavras de 16-bits, com um cabeçalho que inclui informação de controlo e endereçamento, e um bloco de dados.

Para enviar uma mensagem o CPU emissor copia-a para o FIFO do MS respectivo. Após receber o comando de envio o MS gere autonomamente todo o processo de transferência da mensagem, nomeadamente a aquisição do bus, a geração do protocolo de passagem de mensagens e, por fim, a sinalização do CPU com a indicação do resultado da transferência.

Os MR's estão permanentemente a 'escutar' o bus em busca de mensagens destinadas aos seus respectivos processadores. O destino das mensagens é identificado por um campo de endereçamento presente no cabeçalho destas. Quando este endereço condiz com o de um dado MR, este estabelece o handshake com o MS do emissor e copia a mensagem integralmente para a memória FIFO que lhe está associada. O CPU do processador receptor é em seguida notificado da chegada da mensagem.

As mensagens são transferidas através do MPS à taxa de 12Mbytes/s.

O controlo de fluxo entre MSs e MRs utiliza um protocolo simples de rejeição de mensagens previsto na especificação do G-96+. Este protocolo é usado nos casos em que os MRs deixam de poder receber mensagens por falta de espaço nos FIFOs respectivos. Quando tal acontece o MR endereçado rejeita todas as mensagens subsequentes até que algum espaço seja libertado no seu FIFO. A rejeição é detectada por sua vez pelo MS emissor, que aborta a transferência, e notifica de imediato o seu CPU o qual poderá mais tarde ordenar uma retransmissão da mensagem.

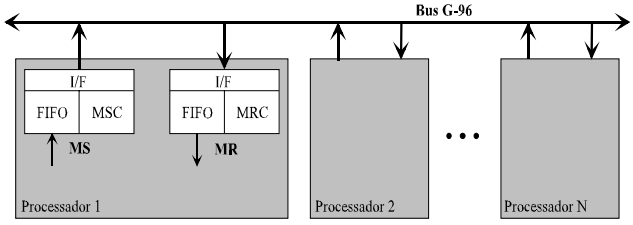

Fig.1 - O Sistema de Passagem de Mensagens (MPS)

Para além da transferência de mensagens ponto-a-ponto o MPS implementa também a comunicação em modo broadcast e multicast. Esta capacidade ao nível do hardware é essencial para uma realização eficiente destes mesmos modos de transferência ao nível da extensão de multiprocessamento.

III. Extensão de Multiprocessamento

O OS9 é um sistema operativo modular baseado num microkernel multitask que se encontra instalado em cada um dos processadores do nosso sistema.

Internamente o OS9 encontra-se funcionalmente repartido numa hierarquia de módulos que implementam, a diferentes níveis de abstracção, funções específicas do sistema operativo. No topo da hierarquia, o microkernel funciona como administrador do sistema e como tal é o responsável pela gestão duma série de recursos e serviços de baixo nível tais como o scheduling dos processos, o serviço às interrupções e o tratamento de excepções, a comunicação entre processos, a gestão de memória, as chamadas ao sistema e o interface com os módulos de I/O.

Para além do kernel a maior parte dos restantes módulos do OS9 têm a haver com a gestão de I/O. A maioria destes são file managers, device drivers e descriptors, módulos portanto mais dependentes da configuração específica dos processadores.

A arquitectura modular é um dos aspectos mais atractivos do OS9. Esta é de facto a característica chave pela qual este SO se presta facilmente a modificações e extensões que podem ser efectuadas de forma organizada através da configuração de módulos já existentes ou através da integração de novos módulos [4]. A abordagem utilizada na implementação de primitivas de comunicação inter-processos que aqui apresentamos, tira partido exactamente desta estrutura modular e consiste efectivamente numa extensão do OS9.

Cada processador aloja um conjunto idêntico de módulos de sistema, que cooperam quer entre si, quer com outros módulos residentes noutros processadores, de tal forma a criar uma infraestrutura virtualmente unificada, que do ponto de vista dos processos se apresenta como um conjunto de funções de comunicação por mensagens.

Na presente versão a extensão de multiprocessamento apenas implementa um método de comunicação inter-processos que é transparente quanto à localização relativa dos processos no sistema. Outras funções normalmente encontradas em SOs distribuídos ou de multiprocessamento, como por exemplo a configuração e a distribuição e balanceamento da carga associada à aplicação, poderão ser implementadas mais tarde recorrendo aos mecanismos de comunicação agora existentes.

No que se segue começamos por apresentar a extensão para o OS9 do ponto de vista da realização do modelo de passagem de mensagens no nosso sistema em concreto.

A. Caracterização do Modelo de Passagem de Mensagens

Uma das primeiras opções que equacionamos na implementação deste modelo de comunicação foi o chamado *método de nomeação*. Este refere-se ao esquema através do qual um processo emissor referencia o destino da mensagem que pretende enviar.

Existem duas soluções básicas. Na primeira o identificador do processo destino é referenciado directamente pelo emissor (*comunicação directa*). Na segunda o processo emissor envia com referência a um repositório de mensagens (*comunicação indirecta*) que por sua vez é acessível ao processo destino. Toda a comunicação inter-processos é realizada através destes depósitos intermédios de mensagens os quais são habitualmente designados por *mailboxes*, *portos* ou *canais* [5], conforme as suas características específicas.

A primeira forma de endereçamento é utilizada no SO distribuído V [6] por referência a identificadores globais de processos. Por sua vez no Mach [7] a comunicação faz-se de forma indirecta via portos.

A nossa extensão de multiprocessamento utiliza uma variante deste segundo método de nomeação em que as mensagens são dirigidas a canais.

A comunicação directa adapta-se a sistemas onde as funções de gestão de processos e comunicação inter-processos se encontram integradas, como acontece nos SOs distribuídos. A nossa extensão, como já referimos, tem apenas por função a comunicação inter-processos e é completamente alheia à gestão destes, o que torna este método bastante mais difícil de implementar tendo em conta a funcionalidade que pretendemos para as primitivas de comunicação.

1. Canais

Os *canais* são estruturas associadas à recepção de mensagens. Para que um processo possa receber mensagens tem que criar pelo menos um canal. O canais tornam os processos que lhe estão associados 'visíveis' de qualquer ponto do sistema.

A comunicação através dum canal é sempre unidirecional. As mensagens são armazenadas no canal à medida que são recebidas e saiem pela mesma ordem à medida que são lidas pelos processos receptores.

A comunicação bidireccional entre dois ou mais processos implica que cada um dos intervenientes defina previamente o seu canal de recepção.

Ao ser criado um canal pode ser definido como *privado* ou *público*. Uma canal privado só pode ser lido pelo processo criador enquanto que um canal declarado como público pode ser partilhado por vários processos receptores. Podemos ter também vários canais afectos a um único processo (fig.2).

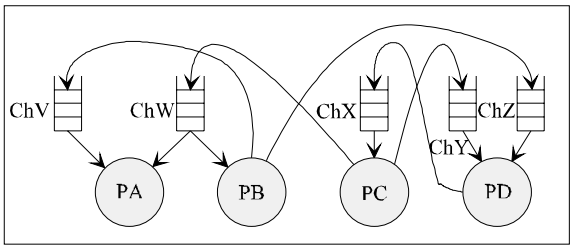

Fig.2 - Comunicação inter-processos através de canais.

Um canal pode ainda ser configurado de forma a sinalizar o processo receptor sempre que uma mensagem dê entrada no buffer do canal. O canal comporta-se neste caso de forma similar a um dispositivo de hardware que gera interrupts (sinais ou eventos, neste caso) e é por isso designado por canal *interrupting*.

2. Primitivas de Comunicação

Do ponto de vista dos processos a nossa extensão de multiprocessamento apresenta um interface constituído por sete primitivas de comunicação (ver caixa "Primitivas de Comunicação").

CREATE e DELCHAN permitem criar e destruir canais, respectivamente. SEND e RECEIVE são as primitivas de transferência de mensagens. As restantes três funções lidam com *ligações*, uma outra entidade a introduzir mais à frente.

As funções SEND e RECEIVE podem funcionar em modo *blocking* (*síncrono*) ou *nonblocking* (*assíncrono*). Embora a maior parte dos SOs distribuídos suportem apenas um tipo de primitiva, nós decidimos incorporar os dois tipos na nossa implementação do modelo de passagem de mensagens. Assim os processos podem escolher dinamicamente o modo mais adequado a cada transferência individual. O modo síncrono será utilizado para sincronizar operações, ou sempre que for necessário garantir a entrega das mensagens. Por sua vez o modo assíncrono é intrinsecamente mais rápido e permite obter um maior grau de paralelismo.

Em qualquer um dos modos de funcionamento a função SEND começa por transferir a mensagem do buffer local do processo chamador, directamente para o hardware do MS ou então para um buffer de sistema. Em modo assíncrono isto permite que o controle seja devolvido rápidamente ao chamador e que este possa reutilizar de imediato o seu buffer local.

Em modo síncrono o processo emissor é bloqueado na função de envio até que a mensagem seja recebida no destino. A sua reactivação é comandada por uma mensagem de confirmação da recepção (mensagem de *acknowledge*) enviada em sentido contrário, ou seja do receptor para o emissor. Esta mensagem é dita de sistema, uma vez que é transferida duma forma não explícita entre os dois processos. A mensagem de acknowledge é gerada no momento em que a mensagem enviada é lida pelo

processo receptor, o que confere a este modo de transferência um elevado grau de fiabilidade em comparação com outras abordagens (por exemplo [8]) onde a confirmação é gerada logo que a mensagem chega ao processador destino, mas antes desta ser entregue ao processo receptor. Neste caso a mensagem de acknowledge não constitui garantia de que a mensagem enviada chegou ao processo destino.

As mensagens são lidas dos canais através da função RECEIVE. Se o canal endereçado não estiver vazio a função retira a mensagem do topo da queue do canal e passa-a ao processo invocante.

Se o canal estiver vazio, em modo blocking a função suspende o processo invocante até que uma nova mensagem dê entrada no canal. Em modo nonblocking o RECEIVE regressa de imediato ao processo invocante com a indicação de que não existem mensagens no canal.

Em ambas as primitivas o modo blocking pode resultar na suspensão indefinida dos processos. Um erro de programação por exemplo, é o suficiente para que uma mensagem enviada em modo síncrono não chegue ao receptor, bloqueando assim indefinidamente o processo emissor. O mesmo pode acontecer com um processo receptor que se encontra bloqueado num canal à espera duma mensagem que nunca é produzida.

Para resolver estes problemas ambas as funções de comunicação foram definidas com um parâmetro de entrada que especifica o intervalo de tempo máximo durante o qual a função deverá ficar bloqueada (*timed-wait*). A função SEND em modo síncrono ficará assim bloqueada até ser recebida a mensagem de acknowledge ou até se esgotar este intervalo de tempo definido. A função de leitura funciona de modo idêntico, retornando ao processo leitor logo que uma mensagem dê entrada no canal, ou então após decorrido o intervalo de time-out.

3. Endereçamento de Canais Remotos

Cada canal é identificado por um nome que deve ser único em todo o sistema. A transferência de mensagens dum processo emissor até a um qualquer canal só pode ser efectivamente realizada após a obtenção do *endereço físico* desse canal. Antes da comunicação propriamente dita o canal deve ser portanto previamente localizado.

Existem duas soluções gerais para este problema [8]. A

Primitivas de Comunicação

SEND (*ConnID, DataPtr, Size, Priority, TimeOut*) - Envia numa mensagem o bloco de dados apontado por *DataPtr*, de comprimento *Size* bytes (max. 256) para a ligação identificada por *ConnID*. *Priority* é um parâmetro binário que indica ao sistema o nível de prioridade da mensagem, Low ou High.O parâmetro *TimeOut* indica o modo de transferência a usar no envio da mensagem. O modo assíncrono é seleccionado para *TimeOut* = 0. Para outros valores de *TimeOut* a primitiva funciona em modo síncrono, bloqueando o chamador até que a recepção da mensagem seja confirmada. O intervalo de tempo máximo durante o qual o chamador está 'disposto' a esperar pelo acknowledge é dado por 2x*TimeOut* ticks, com *TimeOut* entre 1 e 254. Se o valor atribuido a *TimeOut* for 255 este intervalo de tempo é infinito.

***Size* = RECEIVE (*ChanID, DataPtr, TimeOut*)** - Lê uma mensagem do canal correspondente a *ChanID* para o buffer apontado por *DataPtr*. O número de bytes lidos é passado ao chamador em *Size*.
O parâmetro *TimeOut* especifica o intervalo de tempo máximo durante o qual o processo fica bloqueado no caso do canal se encontrar vazio. Este é dado por 2x*TimeOut* ticks, com *TimeOut* entre 0 e 254. Se o valor de *TimeOut* for 255 o tempo de espera é infinito. Se *TimeOut* = 0 o tempo de espera é igualmente zero e o RECEIVE é non-blocking.

***ChanID* = CREATE (*ChNamePtr, Shared, Intrpt*)** - Cria um canal com o nome apontado por *ChNamePtr* (até 4 caracteres). Retorna em *ChanID* o identificador do canal para todas as referências subsequentes (RECEIVE, DELCHAN). Os parâmetros binários *Shared* e *Intrpt* especificam os atributos do canal a criar. O canal é criado como público se *Shared*=1. Um canal público pode ser lido por qualquer processo residente no mesmo processador onde o canal foi criado. Um canal privado (*Shared*=0) só pode ser lido pelo processo que o criou. O canal é criado como 'interrupting' se *Intrpt*=1. Neste caso, por cada mensagem que chegue ao canal, é enviado um sinal para o processo associado (o processo criador). O código do sinal é dado por $200+*ChanID*, identificando univocamente o canal que interrompeu.

DELCHAN (*ChanID*) - Destroi o canal correspondente a *ChanID*. Um canal privado apenas pode ser apagado pelo processo que o criou. Um canal público pode ser apagado por qualquer processo dentro do mesmo processador, desde que não haja nenhum outro processo bloqueado nesse canal. Todas as mensagens existentes no canal ou que a ele cheguem depois desta acção, são ignoradas.

***ConnID* = CONNECT (*ChNamePtr, Shared*)** - Cria uma ligação simples para o canal apontado por *ChNamePtr*. Retorna em *ConnID* o identificador da ligação para todas as referências futuras (SEND, DISCONNECT). O parâmetro *Shared* especifica as características da ligação a criar em termos de pública ou privada, de forma idêntica ao que já foi referido para os canais.

DISCONNECT (*ConnID*) - Destroi a ligação correspondente a *ConnID*. O sucesso desta operação é, tal como nos canais, dependente do atributo privado/público da ligação, tal como descrito na função DELCHAN.

***ConnID* = GROUP (*ConnListPtr, Size*)** - *ConnListPtr* é um ponteiro para uma tabela com *Size*(max. 16) ConnID's. Esta função cria um nova ligação, uma ligação de grupo, que 'aponta' simultaneamente para todos os canais já referenciados individualmente por cada uma das ligações dadas. O *ConnID* retornado é o identificador desta nova ligação. A ligação de grupo só é criada se o processo invocante tiver permissão de utilização de cada uma das ligações simples constituintes.
A ligação de grupo é apenas criada como privada se todas as ligações originais também forem privadas.

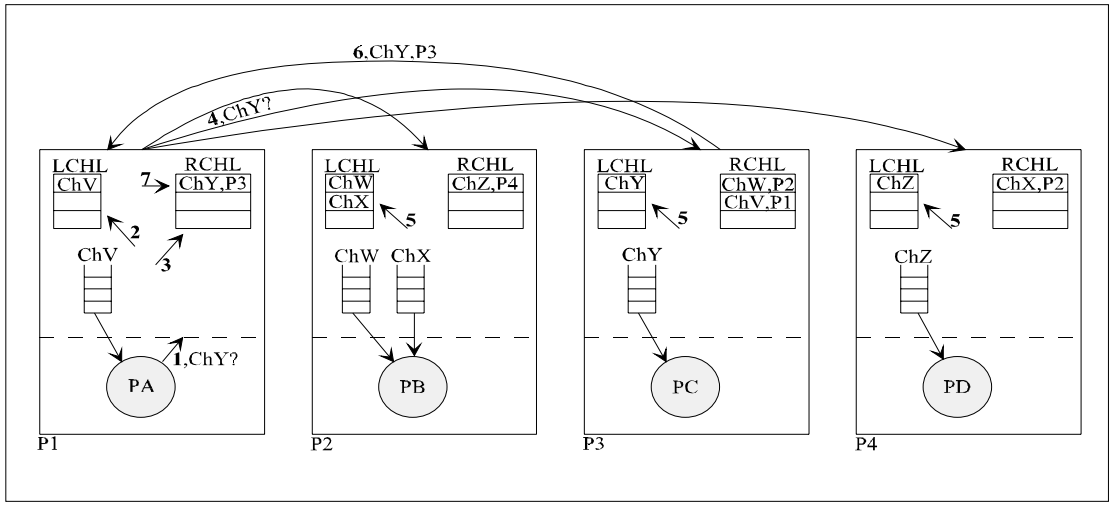

Fig.3 - **1:** Em P1 o processo PA invoca o canal ChY. **2:** Sistema busca ChY na LCHL. **3:** Sistema busca ChY na RCHL. **4:** Mensagem de Inquiry (ChY?) é enviada. **5:** ChY é procurado em todas as LCHL's. **6:** P3 responde com o endereço (P3) de ChY. **7:** RCHL em P1 é actualizada.

primeira passa pela utilização dum *name server* (centralizado ou replicado) que mantém os endereços de todos os canais e que é interrogado sempre que necessário. Na segunda solução é difundida (*broadcast*) uma mensagem de localização do canal por todos os processadores do sistema. Apenas o processador onde o canal procurado está definido responde com o respectivo endereço. O Amoeba [9] é um exemplo típico de SO onde este mecanismo foi adoptado.

Esquemas hibrídos dos dois anteriores são também possíveis. O mais comum consiste em distribuir o name server por todos os processadores. A cada servidor local está associada apenas uma fracção da tabela global de identificação dos canais. Se o endereço físico do canal referenciado não se encontrar nessa fracção da tabela então o servidor local faz o broadcast duma mensagem a interrogar os outros servidores. Esta abordagem é utilizada no SO DIOS [10] (não para localizar canais, mas processos) e no SO do multiprocessador Armstrong [11].

A solução que adoptamos é também basicamente esta última. Cada processador mantém toda a informação respeitante aos seus canais numa tabela local a que chamamos LCHL (*Local Channel List*). Quando é difundida uma mensagem de localização dum canal, a que chamamos mensagem *Inquiry*, todos os processadores 'varrem' a respectiva LCHL em busca do canal procurado. O processador que encontra o canal envia o respectivo endereço físico - numa outra mensagem de sistema designada por *Reply Inquiry* - para o processador interrogante, o qual armazena esta informação numa outra tabela local a que chamamos RCHL (*Remote Channel List*). A figura 3 ilustra detalhadamente este mecanismo.

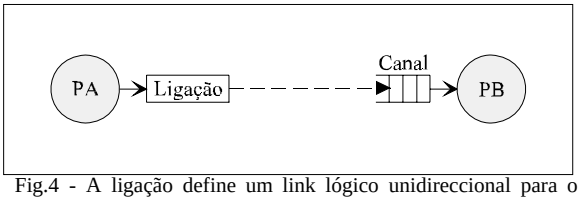

Fig.4 - A ligação define um link lógico unidireccional para o canal.

4. Circuitos Virtuais e Ligações

O procedimento de localização dos canais e de memorização do respectivo endereço físico que acabamos de descrever, constitui o mecanismo fundamental daquilo a que chamamos o protocolo de estabelecimento de ligações. Depois de estabelecida uma ligação, o processo emissor pode finalmente começar a enviar mensagens. O emissor e o canal destino dizem-se ligados através dum *circuito virtual*.

Sistemas de comunicação interprocessos baseados em protocolos orientados para ligações (*connection-oriented protocols*) são comuns a muitos SOs distribuídos, particularmente aqueles em que a comunicação por mensagens se faz de modo explicíto [11,7].

Na prática para inicializar uma ligação, o processo emissor invoca a função CONNECT especificando o nome do canal destino. A extensão localiza o canal de acordo com o procedimento acima referido, e no final cria um estrutura de dados a que chamamos ligação que representa o circuito virtual estabelecido.

As ligações são entidades de comunicação associadas aos processos emissores da mesma forma que os canais se associam aos processos receptores (figura 4). Os processos recebem de canais e enviam com referência a ligações que 'apontam' para os respectivos canais destino. Tal como os canais, também as ligações podem ser privadas ou públicas.

Fundamental nas ligações é a sua função de suporte à sincronização dos processos emissores na comunicação síncrona, em tudo semelhante ao papel dos canais relativamente aos receptores. Depois de enviar uma mensagem em modo síncrono o processo emissor fica suspenso até que seja recebida a correspondente mensagem de acknowledge. Enquanto isso não acontece o

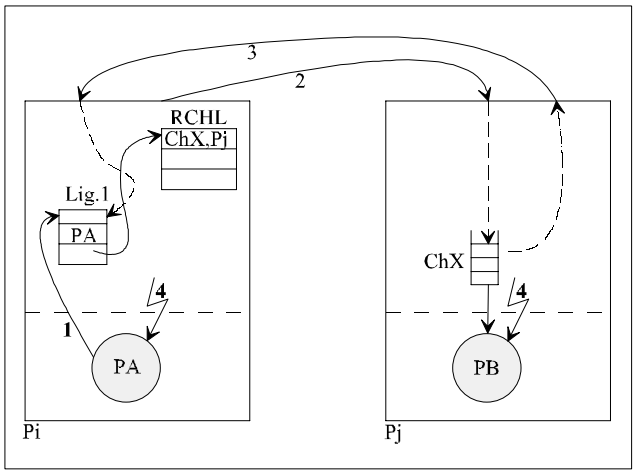

Fig.5 - **1:** PA invoca a função de envio em modo síncrono. O seu identificador é copiado para a ligação e PA bloqueia. **2:** A mensagem é enviada e depositada no canal. **3:** Em Pj é gerada uma mensagem de acknowledge dirigida à ligação Lig.1. **4:** O processo PB é reactivado, se necessário, em Pj. PA é reactivado em Pi.

processo está efectivamente bloqueado na ligação. Quando o acknowledge chega finalmente, transporta o identificador da ligação a sinalizar que por sua vez identifica o processo a reactivar. Este mecanismo está ilustrado na figura 5.

5. Comunicação em Grupo

Até agora falamos apenas em comunicação ponto-a-ponto em que cada ligação aponta apenas para um único canal. A nossa extensão para o SO OS9 suporta também a comunicação em grupo, mais precisamente a transferência de mensagens de um processo emissor para vários canais simultaneamente.

Em alguns SOs com o Amoeba e o V, a comunicação em grupo baseia-se na agregação de processos em grupos que podem ser endereçados como um todo. As mensagens enviadas para um grupo de processos são recebidas individualmente por cada membro do grupo.

No nosso caso este é também o conceito básico subjacente à comunicação de grupo, com a diferença de que não são os processos que se agrupam mas sim as ligações. Um conjunto de *ligações simples* (que referenciam um único canal destino) pode ser agrupado através da função GROUP. A *ligação de grupo* resultante 'aponta' para todos os canais referenciados nas ligações simples originais, pelo que qualquer mensagem enviada através da ligação de grupo é recebida por todos estes canais.

Uma ligação simples pode pertencer simultaneamente a várias ligações de grupo e ao mesmo tempo pode continuar a ser usada isoladamente. Os processos não fazem qualquer distinção entre os dois tipos de ligações, isto é, ambas são referidas com a mesma função de envio de mensagens e com os mesmos parâmetros.

O facto de uma ligação de grupo apontar para vários canais receptores levanta um problema na comunicação síncrona que é o de saber quando é que o processo emissor deve ser reactivado - uma vez que existem vários receptores, irá ser gerado um número igual de mensagens de acknowledge. No SO V o processo emissor é desbloqueado logo após a recepção do primeiro acknowledge. A recepção de acknowledges subsequentes pode ser verificada através duma primitiva especial.

No nosso caso decidimos optar pela reactivação do emissor apenas quando todas as mensagens de acknowledge tiverem sido recebidas. Esta opção foi baseada na premissa de que o protocolo síncrono deve constituir um mecanismo de comunicação fiável, e por isso o emissor só deve prosseguir com a garantia de que a mensagem foi entregue a todos os receptores.

B. Implementação

A figura 6 representa um diagrama da organização dos módulos da extensão de multiprocessamento integrados no SO local de cada processador do sistema. De entre os módulos existentes numa configuração standard do OS9 (file managers, drivers) apenas se representa aqui o kernel por ser o único de interesse na descrição que se segue.

No lado esquerdo da figura estão representados os módulos envolvidos no processamento das mensagens que chegam ao processador. Os módulos do lado direito processam o envio das mensagens para fora do processador. Os módulos MRDrv e MSDrv são respectivamente, os device drivers do Receptor de Mensagens (MR) e do Emissor de Mensagens (MS). Os módulos Mr e Ms contém os parâmetros de inicialização e configuração destes dois dispositivos e designam-se por device descriptors. Um único modulo de dados (por processador) inclui as três estruturas de dados principais da extensão - as tabelas LCHL, RCHL e LCNL.

O elemento central da extensão de multiprocessamento é um file manager denominado *Message Passing Manager*(MPM). É este módulo que implementa as

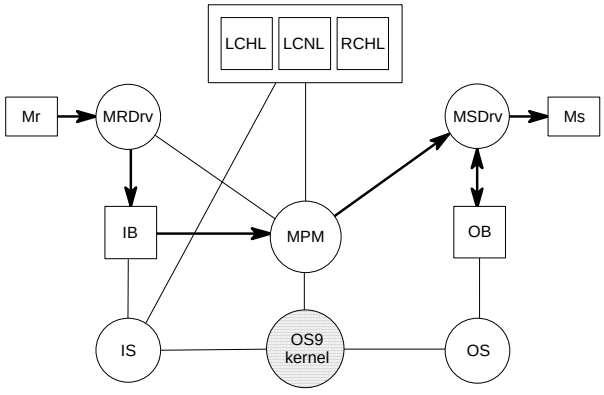

Fig.6 - Módulos da extensão de multiprocessamento. As setas indicam os percursos lógicos das mensagens do ponto de vista da recepção e do envio para o exterior dum processador

funções de comunicação já descritas.

Internamente o MPM consiste num pacote de rotinas, uma por cada primitiva, para onde o kernel transfere o controle ao processar qualquer uma das funções de passagem de mensagens do interface de comunicação. Estas funções são invocadas pelos processos através de chamadas ao kernel (*system calls*), o qual agulha depois para a rotina do MPM correspondente à função pretendida. Algumas destas funções são processadas completamente no MPM, outras, como por exemplo a função Send, requerem eventualmente os serviços dos device drivers.

No que se segue iremos descrever as rotinas do MPM bem como os restantes módulos da extensão seguindo mais ou menos a ordem indicada pelas seta da figura 6, ou seja comecando por onde as mensagens entram em cada processador e terminando onde elas saem.

1. Channel Descriptors

Cada canal é representado por um *Channel Descriptor* (ChD). O ChD (fig. 7a)) é uma estrutura de controle que inclui informação sobre o nome do canal, o processo a que está atribuído, os atributos e o estado do canal, e um par de ponteiros que referem o início e o fim da fila de espera do canal, a *Channel Queue* (ChQ).

Criar um canal envolve a alocação dum ChD na LCHL e a sua inicialização de acordo com os parâmetros fornecidos através da função CREATE. As flags Sh e I memorizam respectivamente os atributos público/privado e interruptting/not interruptting associados ao canal. As flags S e W indicam o estado do canal (status flags). A flag S reflecte o estado da ChQ em termos de vazia/não vazia. A flag W serve para indicar se há algum processo suspenso no canal. Finalmente a flag E indica se o canal está a ser 'apontado' por alguma ligação externa. Esta última informação torna-se necessária quando o canal é terminado através da função DELCHAN. Se a flag E estiver a 1 o sistema gera uma mensagem em modo broadcast a indicar que o canal já não existe. Se E for 0 esta mensagem já não é necessária.

2. Connection Descriptors

Enquanto que a criação de canais envolve apenas actividade local, o estabelecimento de ligações para canais remotos requer a colaboração de todos os processadores do sistema.

Os mecanismos de localização de canais foram já descritos na secção anterior. Todo o processo que descrevemos é normalmente despoletado pela função CONNECT quando o canal procurado não é local.

Se o canal for local o seu ChD está na LCHL. Se não estiver na LCHL a função varre em seguida a RCHL - a lista dos canais remotos conhecidos localmente. Nesta tabela cada canal é representado por um *Remote Channel Descriptor* ou RChD, que inclui os campos indicados na figura 7b).

Se o canal para o qual é requerida uma ligação também não é conhecido na RCHL então o sistema recorre ao mecanismo já descrito para obter o endereço físico do canal. O passo seguinte consiste na alocação dum *Connection Descriptor* (CnD) na LCNL que representa a ligação estabelecida.

A figura 8 representa em a) e b) os dois tipos de CnDs usados respectivamente para as ligações simples e para as ligações de grupo. Alguns dos campos presentes em ambos os CnDs têm exactamente a mesma função que descrevemos para os campos homónimos dos ChDs. CnDs do tipo *G_Connection* são construídos a partir dum grupo de CnDs do tipo *S_Connection* pela função GROUP. O campo N_Connects indica o número de canais para o qual a ligação de grupo aponta. Uma lista no final do CnD identifica cada um dos canais destino através de referências individuais à LCHL (canais locais) ou à

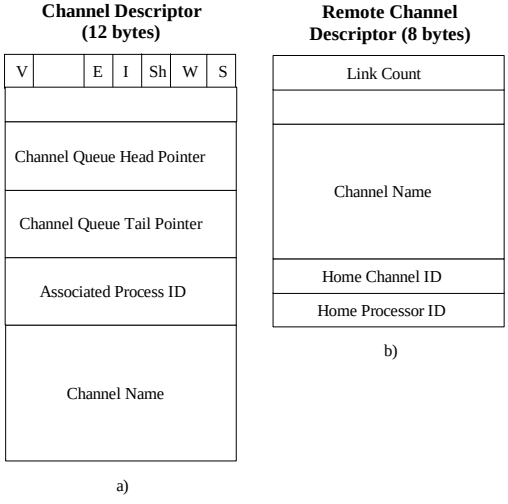

Fig.7 - a) - Channel descriptor(ChD); b) - Remote channel descriptor(RChD).

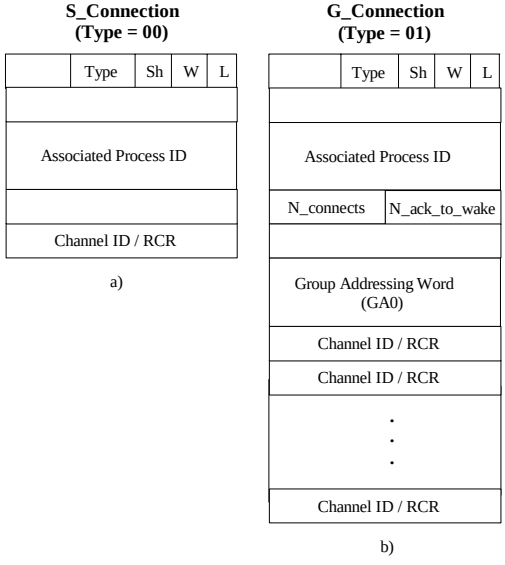

Fig.8 - *Connection descriptors(CnD);*
a) - Ligação simples b) - Ligação de grupo.

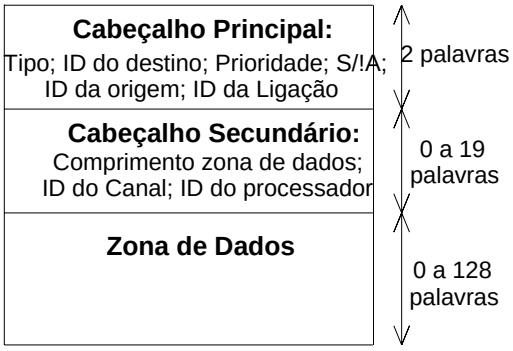

Fig.9 - Formato das mensagens

RCHL (canais remotos).

3. Formato das Mensagens

A figura 9 ilustra o formato geral das mensagens na nossa extensão de multiprocessamento. As mensagens são constituídas pelos três campos principais representados. O *cabeçalho principal* é o mesmo para todas as mensagens. Os outros, isto é, o *cabeçalho secundário* e a *zona de dados* podem variar bastante em termos de conteúdo e de comprimento.

Existem mensagens de *sistema* e de *utilizador*. As mensagens de Acknowledge, Inquiry e Reply Inquiry são as únicas mensagens de sistema definidas até agora. As mensagens de utilizador são as que são usadas na comunicação propriamente dita, entre processos. Dependendo do tipo de ligação de onde originam, estas mensagens são classificadas como *mensagens ponto-a-ponto*, se foram enviadas através de ligações simples, ou *mensagens de grupo*, se foram enviadas através de ligações de grupo.

4. Recepção de Mensagens Remotas

As mensagens recebidas do exterior passam por dois níveis de processamento até chegarem aos canais destino (ver fig.10). Numa primeira fase o MRDrv copia a mensagem do FIFO do MR para um buffer comum de entrada, o *Input Buffer* (IB). A segunda fase é executado pelo *Input Server* (IS), um processo de sistema que lê as mensagens do IB e as processa de acordo com o seu tipo. As mensagens de utilizador são nesta altura introduzidas nos respectivos canais destino, sendo também executados os procedimentos inerentes a cada tipo de mensagem de sistema. Ao receber uma mensagem de acknowledge, por exemplo, o IS procura na LCNL uma ligação com um processo suspenso, e em seguida reactiva esse processo.

Input Buffer - Neste buffer as mensagens estão sequencialmente organizadas segundo uma 'linked-listed' a que chamamos *Input Queue* (IQ). Depois de recebidas, as mensagens são inseridas no topo ou na cauda da IQ conforme a sua prioridade.

Os buffers correspondentes aos canais locais, as ChQs, estão também definidos no IB. Isto permite que as mensagens seram transferidas da IQ para qualquer ChQ (i.e. inseridas nos canais destino), através duma simples manipulação de link pointers sem que ocorra efectivamente uma operação de cópia, ou alocações e libertações de espaço no IB. Assim, embora a figura 10 possa sugerir duas operações de cópia, na realidade exista apenas uma.

Mensagens ponto-a-ponto - Estas mensagens transportam o identificador dum canal destino no cabeçalho secundário. O IS procura esse canal na LCHL e insere a mensagem no fim ou no início da ChQ respectiva de acordo com a sua prioridade.

Se existir algum processo suspenso no canal o IS envia um sinal de *wake-up* para esse processo. Se o canal for do tipo interrupting é também enviado um sinal para o processo registado no ChD. O código do sinal identifica o canal interruptor.

Mensagens de grupo - Estas mensagens transportam no cabeçalho secundário a identificação de todos os canais destino. Estes podem residir todos em processadores distintos mas também podem haver vários canais

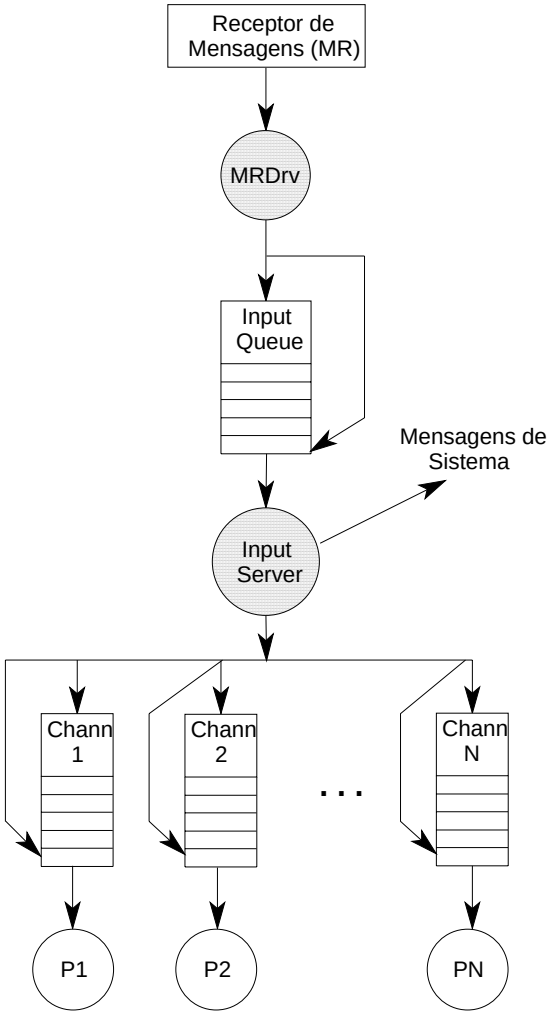

Fig.10 - Percurso das mensagens desde a entrada do processador (FIFO do MR) até aos canais destino.

pertencentes ao mesmo processador.

Perante uma mensagem de grupo a função do IS é distribui-la por todos os canais destino do processador em questão. Para isto a solução mais óbvia seria criar uma réplica da mensagem original por cada canal, uma abordagem que rejeitamos no entanto logo à partida, devido ao overhead e aos problemas de gestão do IB que iria introduzir. Em lugar disso o IS insere em cada canal destino uma mensagem especial, que designamos por *mensagem ponteiro*, que referencia a mensagem de grupo original.

Dado que a mensagem de grupo se destina a vários leitores, é necessário um mecanismo que permita saber quando é que todas as leituras foram já efectuadas. Este mecanismo baseia-se numa variável existente no cabeçalho da mensagem de grupo que é inicializada com o número de destinos. Por cada vez que a mensagem é lida esta variável é decrementada, e quando o seu valor for zero a mensagem de grupo é removida do IB.

5. Leitura dos Canais (RECEIVE)

Até aqui seguimos o caminho das mensagens de utilizador desde o bus do multiprocessador até aos respectivos canais destino. Vejamos agora como se processa a leitura destes canais através da função RECEIVE.

Usando o identificador fornecido pelo chamador a função começa por localizar o canal (o ChD na LCHL). Em seguida a mensagem no topo da ChQ é extraída, o cabeçalho é separado e finalmente o corpo da mensagem é copiado para o buffer do chamador. Se a mensagem é assíncrona a função RECEIVE termina aqui. Caso contrário, se a mensagem for síncrona, a função tem que sinalizar a origem de forma a confirmar a recepção.

A origem da mensagem síncrona determina o mecanismo utilizado para efectuar esta sinalização. O cabeçalho principal da mensagem (ver fig.9) inclui o identificador do processador e o identificador da ligação de onde a mensagem originou. Se a mensagem lida foi enviada a partir de outro processador (i.e., é remota), a função RECEIVE confirma a recepção enviando uma mensagem de acknowledge para a ligação origem. Por outro lado se a mensagem é local, a confirmação traduz-se num sinal de wake-up que é enviado directamente para o processo que se encontra suspenso na ligação origem.

6. Envio de Mensagens através das Ligações (SEND)

Para transferir uma mensagem a função SEND começa por localizar a ligação (o CnD na LCNL) identificada pelo processo chamador. A partir da ligação é obtido o endereço físico do canal destino.

Canais locais - Se a ligação 'aponta' para um canal local então a mensagem é copiada directamente para a ChQ do canal. O parâmetro *Priority* forneido pelo chamador indica se a mensagem deve ser inserida no topo ou na cauda da ChQ. Se existir algum processo suspenso no canal a função envia uma sinal de wake-up para esse processo. O mesmo acontece se o canal for do tipo 'interrupting', só que neste caso o sinal enviado é diferente.

Em modo assíncrono a função SEND termina aqui, regressando ao chamador. Em modo síncrono, no entanto, a função só devolve o controle ao processo depois de ter recebido uma confirmação da recepção da mensagem. Neste caso a função regista o processo chamador num campo específico do CnD e suspende-o de seguida. Mais tarde quando o sinal de confirmação chegar, o processo é reactivado.

Canais remotos - Se o canal referido pela CnD é remoto, então a função SEND chama o driver MSDrv que trata da transferência das mensagens para fora do processador. Ao regressar do driver a função retorna ao chamador ou antes suspende o processo na ligação, conforme se trate do modo assíncrono ou síncrono.

7. Envio de Mensagens Remotas

O driver MSDrv juntamente com um processo de sistema, o *Output Server* (OS), asseguram a gestão de todas as transferências para o exterior de cada processador. Estes módulos estão indicados no diagrama da figura 11 que mostra o percurso lógico das mensagens desde a função SEND até ao hardware do Message Sender (MS).

Driver MSDrv - Normalmente as mensagens são copiadas directamente do buffer do processo para o FIFO do MS, que depois controla automaticamente a transferência ao nível do hardware. Se eventualmente o MS estiver activo (ocupado com outra mensagem) então o driver coloca a mensagem num buffer temporário, o

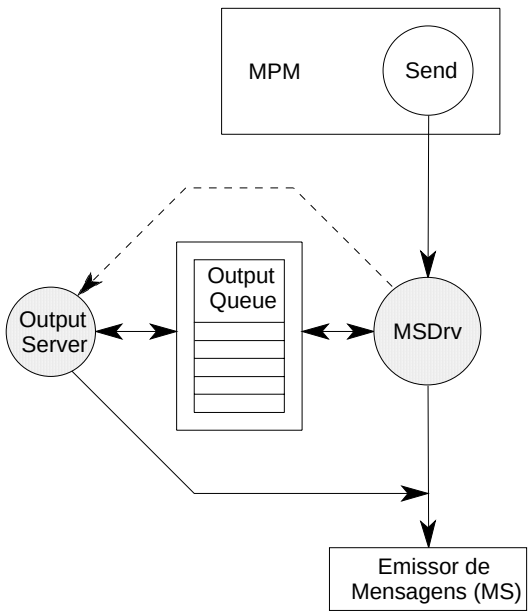

Fig.11 - Percurso das mensagens desde a função SEND até ao Emissor de Mensagens do processador.

Output Buffer (OB), onde esta aguarda até que o MS volte a ficar disponível.

Output Buffer - Tal como no Input Buffer também aqui as mensagens são organizadas segundo uma linked-list a que chamamos *Output Queue* (OQ). Este buffer garante o desacoplamento entre as tarefas de gestão da transferência das mensagens e a função SEND do MPM, permitindo assim a reactivação imediata do processo emissor mesmo que o MS se encontre ocupado nesse instante. Dado que o tempo de ocupação do MS é essencialmente determinado pela taxa de ocupação do bus partilhado, se não houvesse buffering os processos emissores poderiam experimentar periodos de bloqueio provavelmente longos e de duração indeterminada.

Output Server - Uma mensagem pode ser rejeitada pelo processador destino se o seu MR não dispuser de espaço suficiente no FIFO de recepção. Se isto acontecer, a mensagem é re-enviada mais tarde pelo OS.

Na presente versão da extensão o OS assume que a única causa que impede as mensagens de chegarem aos destinos são as rejeições provocadas pelos MR´s. Assim de forma a minimizar o número de retransmissões, o OS nunca tenta retransmitir uma mensagem imediatamente após a sua rejeição - como a mensagem foi previamente rejeitada pelo MR do processador destino por falta de espaço no FIFO de recepção, é provável que a repetição imediata da mesma operação de envio resulte numa nova rejeição.

Se existirem mais mensagens na OQ o OS atrasa a retransmissão e inicia a sequência de envio duma nova mensagem cujo destino seja diferente do da mensagem em falta. Esta última é novamente inserida na OQ e a nova mensagem seleccionada é transferida. Como esta mensagem se destina a outro processador, é pelo menos mais provável que esta transferência seja bem sucedida.

Este procedimento só é no entanto aplicável se existirem outras mensagens na OQ cujo destino (processador) não seja o mesmo da mensagem rejeitada. Para além disso o mesmo procedimento não pode ser também usado se a mensagem em falta foi copiada directamente para o FIFO do MS, e portanto não passou pela OQ. Nestes casos o OS retransmite a mensagem em falta mas apenas algum tempo depois da rejeição ter ocorrido (o OS é desactivado durante um curto periodo), de forma a dar algum tempo ao processador destino para libertar algum espaço no FIFO do seu MR.

C. Desempenho da Extensão

Nesta última secção apresentamos uma estimativa do desempenho da extensão de multiprocessamento.

Seguindo uma prática normalmente usada no teste de outros sistemas do género, efectuamos algumas medidas de tempos de transferência usando um par de processos de teste que trocam entre si uma mesma mensagem. Uma vez que a actividade no sistema se reduz à execução destes processos de teste, as medidas observadas constituem naturalmente apenas uma aproximação muito grosseira da performance real do sistema.

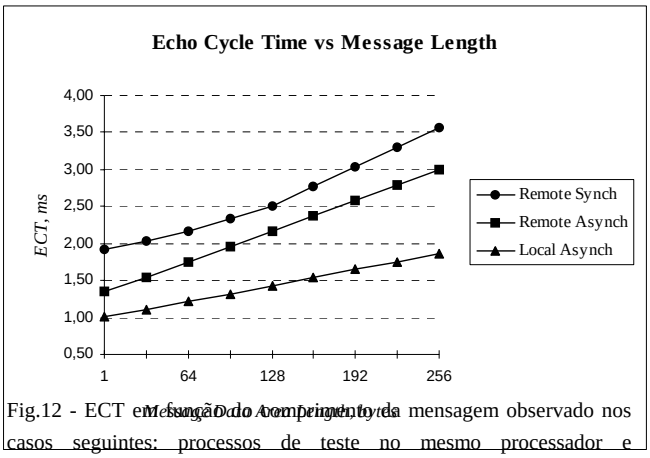

Fig.12 - ECT em função do comprimento da mensagem observado nos casos seguintes: processos de teste no mesmo processador e comunicação assíncrona; processos de teste em processadores distintos e comunicação síncrona e assíncrona.

Assim, medimos o intervalo de tempo desde que um dos processos inicia o SEND até ao instante em que recebe a mesma mensagem do outro processo. Este tempo é o tempo de eco, ou 'Echo Cycle Time'(ECT), e representa portanto duas vezes o tempo de transferência duma mensagem entre processos. De forma a garantir uma precisão razoável nos valores obtidos, as medidas temporais foram efectuadas ao longo dum elevado número de ciclos de emissão e recepção da mensagem. Os resultados obtidos estão graficamente representados na figura 12.

Em modo assíncrono o ECT cresce linearmente com o comprimento da mensagem. Como seria de esperar o tempo de transferência divide-se em duas parcelas, uma constante de overhead e uma componente directamente proporcional ao comprimento da mensagem.

Em modo síncrono esta dependência linear é mantida se os processos forem locais. Contudo se os processos forem remotos (i.e. residirem em diferentes processadores) o ECT em modo síncrono cresce mais rapidamente. Alguns testes mais aprofundados demonstraram que isto acontece porque o padrão de 'scheduling' em cada processador é alterado quando o comprimento das mensagens ultrapassa determinado limite. A transferência de mensagens remotas em modo síncrono é bastante mais sujeita a efeitos de 'scheduling' devido a envolver 4 mensagens por cada ciclo de emissão-recepção, ao passo que em modo assíncrono cada ciclo envolve apenas 2 mensagens.

As medidas reportadas na figura 12 foram obtidas com processadores baseados no MC68020 a funcionar a 12MHz. Dado que o ECT é essencialmente determinado pelo tempo de computação, se aumentarmos a velocidade do CPU o ECT deverá ser reduzido de forma significativa. Para quantificar este aumento de performance efectuamos algumas medidas com CPUs a 16 e a 24MHz que apresentamos no gráfico da figura 13.

Mais uma vez verifica-se uma relação linear entre o ECT e o período de clock do CPU. A inclinação das rectas é determinada pelo comprimento da mensagem.

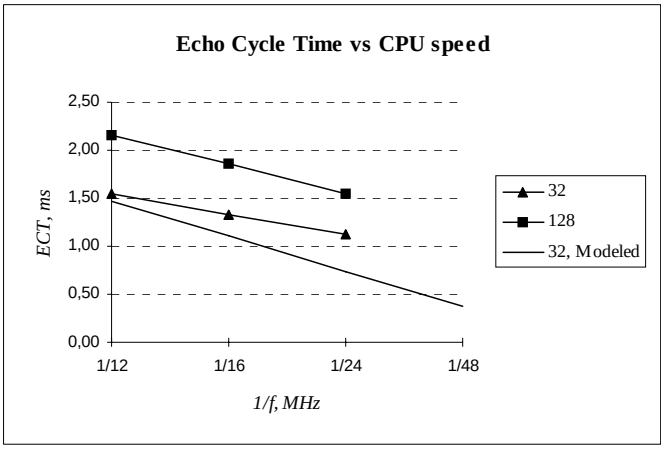

Fig.13 - As duas rectas mais acima referem-se ao ECT medido em função do período de clock dos CPUs, para dois comprimentos de mensagem (32 e 128 bytes - processos de teste comunicam assincronamente).
A recta de baixo refere-se ao ECT estimado, também em função de T_{CLK}, mas considerando sempre CPUs a funcionar sem wait-states.

A velocidade a que o CPU executa instruções é por sua vez também dependente do tempo de acesso da memória. A partir das medidas efecuadas obtivemos a seguinte expressão empírica que modela o ECT em modo assíncrono para processos remotos

$$ECT = (49.5N + 8546)T_{CLK} + 1100T_{ROM} + (15.1N + 2167)T_{RAM}$$

T_{ROM} e T_{RAM} são os tempos de ciclo da ROM e da RAM respectivamente. T_{CLK} é o período de clock dos CPUs e N é o comprimento da mensagem.

Esta expressão verifica os resultados obtidos com um erro muito pequeno e para além disso permite estimar o ECT que teriamos com CPUs e memórias mais rápidos, considerando que o padrão de scheduling se manteria inalterado, tal como observado até aqui para o modo assíncrono.

Os dois traçados superiores da figura 13 (experimentais) foram obtidos variando T_{CLK} e mantendo constantes o comprimento da mensagem e o tempo de ciclo das memórias. A variação do ECT seria obviamente muito mais acentuada se para cada valor menor de T_{CLK} tivessemos utilizado memórias sucessivamente mais rápidas. Para termos uma ideia de como o ECT iria variar nestas condições apresentamos na figura 13 (em baixo) o traçado da expressão anterior em que os tempos T_{ROM} e T_{RAM} foram ajustados de forma a garantir, para cada uma das frequências dos CPUs, acessos sem wait states. O gráfico obtido é na mesma uma recta dado que os tempos de ciclo das memórias são múltiplos inteiros de T_{CLK}. O traçado foi também estendido até 48MHz. A esta frequência o ECT estimado é cerca de 60% do ECT medido a 24MHz.

IV. Conclusões

Um pequeno sistema constituído por vários processadores é muitas vezes a solução ideal para muitas aplicações. Nestes casos o desenvolvimento duma extensão de multiprocessamento a partir dum kernel multitask mono-processador conhecido, pode ser uma alternativa viável à adopção de um novo sistema operativo multiprocessador [2] (assumindo que este existe no mercado).

Neste artigo descrevemos uma extensão de multiprocessamento para o sistema operativo OS9 da Microware. A extensão assenta no paradigma da comunicação por mensagens e implementa um serviço de comunicação transparente entre processos baseado em canais e circuitos virtuais. As primitivas suportam os modelos de comunicação síncrona, assíncrona e em grupo.

Embora o nosso esforço de desenvolvimento tenha sido mais dirigido no sentido de proporcionar funcionalidade do que performance, os resultados obtidos no final revelaram níveis de desempenho que não estão muito longe de sistemas similares [8,11]. Adicionalmente como os tempos de comunicação dependem fortemente da velocidade dos CPUs, podemos ainda melhorar este desempenho apenas à custa dum 'upgrade' dos processadores.

V. Referências

[1] Richard A Quinnel; "Distributed operating systems combine multiple processors into a single machine"; EDN, Sept 28, 1995.

[2] Charles H Small; "Small real-time systems coordinate tasks over tiny local nets"; EDN, Nov. 25, 1993.

[3] Ernesto F.V. Martins and António Nunes da Cruz; "An operating system extension for a multiprocessor"; Submetido a Journal of Systems Architecture.

[4] Richard A Quinnel; "Microkernel and modular OSs"; EDN, April 13, 1995.

[5] Shirley A Williams; "Programming models for parallel systems"; Chichester; John Wiley, 1990.

[6] David R Cheriton; "The V distributed system"; Communications of the ACM, Vol.31, No.3, March 1988.

[7] Andrew S Tanenbaum; "Modern operating systems"; Prentice Hall, 1992.

[8] Miomirka Cvijovic and Mojca Kunc; "An approach to the design of distributed real-time operating systems"; Microprocessors and Microsystems, Vol.16, No.2, 1992.

[9] Andrew S Tanenbaum et al; "Experiences with the Amoeba distributed operating system"; Communications of the ACM, Vol.33, No.12, Dec. 1990.

[10] J. Alves Marques et al; "The distributed operating system of the SMD project"; Software - Practice and Experience Vol.18(9), 859-877, Sept. 1988.

[11] James T Rayfield and Harvey F Silverman; "System and application software for the Armstrong multiprocessor"; Computer, June 1988.