Implementação e Testes do Sistema de Comunicação do Ambiente RIO

Alexandre Sztajnberg
alexszt@gsc.ele.puc-rio.br

Orlando Loques
loques@gsc.ele.puc-rio.br

Julius C. B. Leite
julius@gsc.ele.puc-rio.br

Grupo de Sistemas de Computação
Departamento de Engenharia Elétrica
Pontifícia Universidade Católica, Rio de Janeiro
Cx. Postal 38063 - CEP 22452-920 - RJ

Resumo

A demanda por serviços diversificados em aplicações distribuídas impõe a necessidade de um modelo de comunicação flexível e modular, que possa acomodar continuamente novas funções e requisitos. Neste trabalho apresenta-se o modelo de comunicação adotado no ambiente RIO[1], que inclui uma metodologia para a construção de aplicações distribuídas e permite selecionar a forma mais adequada para comunicação entre seus módulos. Apresenta-se, também, o quadro de mensagem do sistema e como interagem cada um de seus campos com os múltiplos protocolos suportados pelo ambiente. Em seguida, são discutidos pontos relevantes ao desempenho do sistema e propostas de otimização.

Abstract

The distributed applications demand for diversified services leads to the necessity of a modular and flexible communication model, that can host, continuously, new communications functions and requirements. In this paper the communication model adopted by the RIO environment is presented, wich includes a distributed application construction methodology and permits the selection of the most adequate form of communication between it's modules. It is also presented the system's message frame and how their fields interacts with the multiple protocols supported. Performance aspects and otimization proposals are discussed.

1. Introdução

O modelo de comunicação do ambiente RIO está fortemente estruturado segundo uma metodologia de software que distingue a construção de módulos individuais da composição efetiva de um sistema através da interligação dos mesmos [12]. A metodologia também conceitua dispositivos de interconexão, que possibilitam a descrição conveniente do relacionamento dos componentes de um sistema através de trocas de mensagens. A implementação deste modelo, além de modular, deve ser configurável, de modo a possibilitar que versões diferentes da mesma aplicação, utilizando protocolos diferentes, sejam compostas.

[1]RIO - Reconfigurable Interconnectable Objects. O nome RIO será usado como sinônimo de ambiente, metodologia e sistema, onde for aplicável.

2. Conceitos Básicos

A estrutura básica para a construção de qualquer sistema é o **módulo** (fig. 1). Cada **tipo** de módulo é uma entidade que contém um trecho de código, definições de variáveis, dados internos e um ou mais pontos de interconexão externa, por onde se realizam as interações com outros módulos.

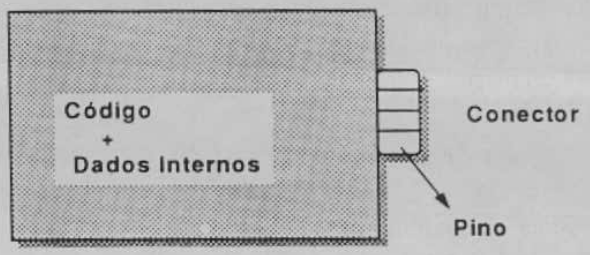

Figura 1: esquema de um módulo

Esta interação é realizada através de interfaces definidas por **conectores**, cuja função é concentrar as atividades de comunicação de um módulo de forma disciplinada. Os conectores são definidos separadamente dos módulos. Portanto, qualquer módulo pode, na fase de programação, *encaixar* um conector já definido e, posteriormente, durante a operação do sistema, usá-lo para trocar **mensagens** com outros módulos. Neste contexto, módulo é um conceito abstrato a partir do qual todas as construções do sistema são feitas. O tipo de módulo pode ser considerado como uma *fôrma* básica para a criação de unidades operacionais. No momento em que um tipo de módulo precisa ser ativado, uma cópia deste módulo é **instanciada** (recebendo uma identidade e área de dados próprias). Em seguida, seus conectores são ligados a outros conectores e, isto feito, é iniciada sua operação efetiva.

A interconexão de módulos se dá através da **ligação** entre conectores. Cada conector contém um ou mais **pinos de ligação**, que são os elementos básicos por onde se realiza a comunicação. Toda a informação enviada ou recebida por meio de um pino é composta por estruturas de dados, chamadas **mensagens**. Assim, cada pino é caracterizado pelos **tipos de mensagens** que por ele trafegam, pelo **sentido** das mensagens (entrando ou saindo do módulo) e pela estilo de troca de mensagens associado ao mesmo. Desta forma, a consistência da ligação entre dois ou mais pinos poderá ser verificada, permitindo-se somente ligações entre pinos compatíveis.

2.1. Transação

A **transação** é uma abstração utilizada pelo programador na fase de projeto, para

definir a semântica da troca de mensagens entre os módulos da aplicação. Isto permite ao programador que se concentre nos detalhes da aplicação, tendo apenas que especificar a forma pela qual os módulos vão interagir. O sistema fornece funções primitivas para a programação das transações dentro dos módulos [12] e sintaxes adequadas podem ser definidas para diferentes linguagens de programação. Os detalhes de como estas interações são processadas, que protocolos irão utilizar, ou se existem dependências quanto à tecnologia utilizada são, em princípio, irrelevantes para a aplicação. Tais detalhes receberão atenção em outro contexto.

Podemos definir, a *priori*, três tipos básicos de transação: **síncrono, assíncrono** e **multicast.** Cada tipo de transação pode possuir variantes através do acréscimo de temporizadores ou pela seleção de um protocolo específico para o transporte das mensagens. Além disso, estilos de transação diferentes podem ser introduzidos no sistema. A seguir descreve-se cada um dos tipos básicos de transação.

- **Assíncrona**

A transação assíncrona admite apenas uma mensagem enviada por determinado módulo, sem o compromisso de esperar uma resposta por parte do módulo receptor (fig. 2). O módulo segue o seu processamento tão logo tenha enviado a mensagem.

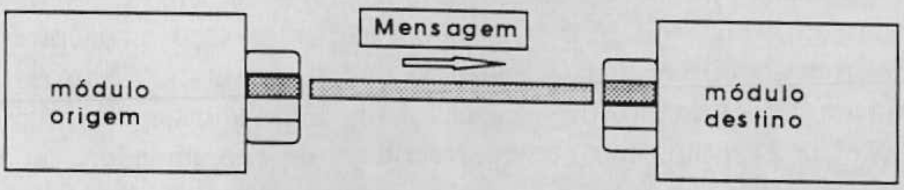

Figura 2: transação assíncrona entre dois módulos conectados

A escolha mais imediata para concretizar o transporte da mensagem seria um protocolo baseado em datagrama, como o UDP. Outra possibilidade seria a utilização de um protocolo mais confiável, ainda baseado em datagrama, que chamamos de **datagrama confiável.**. Neste protocolo, uma mensagem é enviada de forma assíncrona, trabalhando o sistema de comunicação no sentido de garantir o transporte da mensagem de forma confiável. Neste caso, o sistema de comunicação da estação originária da mensagem fica aguardando o reconhecimento explícito por parte da estação receptora. Pelo lado da estação receptora, o sistema de comunicação passa o conteúdo da mensagem para o módulo de destino e, paralelamente, envia uma mensagem de reconhecimento para a estação originária, completando-se assim uma transação. Devemos deixar bem claro que não foi criada aqui uma nova transação. A seleção de um protocolo específico para a transação assíncrona apenas lhe proporcionou um atributo adicional, sem qualquer alteração nos módulos envolvidos.

- **Síncrona**

Na transação síncrona existe, em princípio, a troca de duas mensagens: um **pedido** e uma **resposta** a este pedido (fig. 3), sendo que o módulo que transmitiu o pedido ficará bloqueado aguardando a chegada da resposta do módulo receptor (estilo RPC - *Remote Procedure Call* ou Pedido/Resposta). O tempo máximo que o módulo originador da mensagem ficará aguardando a resposta é programável (parâmetro de ***time out***) e pode variar de zero (comportando-se como uma transação assíncrona) até infinito (ficando bloqueado até receber a resposta).

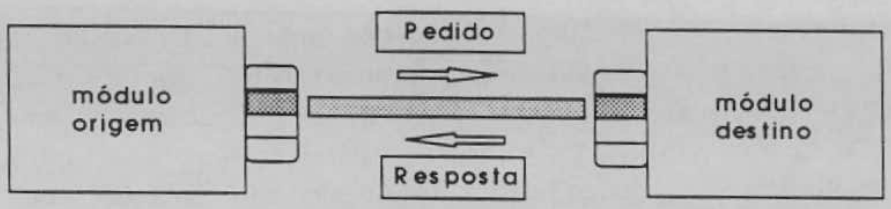

Figura 3: transação síncrona entre dois módulos conectados

Uma série de variações imediatas sobre a transação síncrona básica pode ser feita utilizando-se um **identificador** de transação. Esta identidade atravessa os limites de uma estação para outra, associada às mensagens. O identificador contém informações a respeito da transação, pinos envolvidos, temporizadores, número de ordem, etc. Sendo assim, um único módulo poderá trabalhar com várias transações concorrentemente e tratá-las de formas diferenciadas, em uma ordem arbitrária. Existe, ainda, um mecanismo para o **tratamento de exceções**. Uma anormalidade no sistema irá gerar uma condição de exceção, que poderá, a critério do programador, ser tratada ou ignorada. As exceções são mapeadas em mensagens com alta prioridade e enviadas pelo sistema de comunicação para o módulo afetado (ver subseção 4.4.4).

Uma opção para o suporte desta transação seria usar qualquer protocolo já disponível no ambiente nativo, e.g., TCP. Uma alternativa seria implementar um protocolo de transporte especial internamente ao RIO, similarmente ao adotado nas propostas de suporte a RPC. Em princípio, as possibilidades para o transporte de mensagens são amplas. Por exemplo, pode-se suportar um esquema de **memória virtual distribuída**, que possibilite o remapeamento de uma página de memória entre quaisquer dois espaços de endereçamento; o tratamento de falta de página (*page fault*) é feito de forma automática. Neste esquema a troca de mensagens é feita por referência e o conteúdo só é transportado quando necessário. Este tipo de transporte também é chamado de **copy-on-reference**, pois a porção da mensagem é efetivamente transferida apenas quando requerida. Esta técnica pode ser bastante eficiente para mensagens muito longas.

- **Multicast**

Uma outra abstração associada aos conceitos de conexão e transação é o **grupo**. Um conjunto de módulos poderá formar um grupo, fazendo com que pelo menos um de seus pinos esteja ligado ao grupo (fig. 4). Este tipo de ligação suporta comunicação de módulos via disseminação de mensagens, apropriado para transações tipo **multicast**.

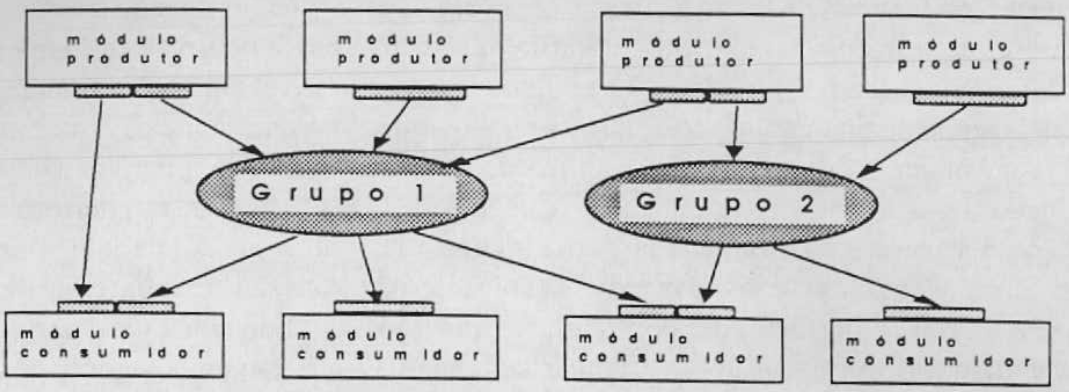

Figura 4: estrutura de dois grupos

Os grupos podem ser criados e terminados dinamicamente através de um serviço de nomes (os grupos são nomeados explicitamente), facilitando a associação de transações e protocolos específicos a cada grupo, de forma independente. De imediato, duas semânticas podem ser associadas à transação multicast: síncrona e assíncrona. No **multicast assíncrono** um módulo difunde uma mensagem para todos os membros do grupo e prossegue a sua execução normalmente. No caso do **multicast síncrono**, o módulo que difundiu a mensagem fica aguardando a resposta de pelo menos um módulo receptor, antes de prosseguir o seu processamento. As outras mensagens de resposta para esta mesma transação são descartadas pelo sistema de comunicação. Outras semânticas podem ser implementadas utilizando-se, por exemplo, mecanismos baseados em votação das respostas dos módulos receptores.

Uma especialização implementada para a transação multicast é a sua versão confiável (*Reliable Multicast*). A transação **multicast confiável** tem o objetivo de difundir mensagens para um grupo de módulos de forma atômica, ordenada e, dependendo do protocolo utilizado, garantindo um tempo máximo de entrega. Para a aplicação funcionar desta forma é bastante que o operador selecione um **protocolo de difusão confiável** adequado como a forma de conexão entre os módulos do grupo. O ambiente RIO suporta atualmente a proposta de Cristian [3,11], estando em fase de implementação a proposta de Kaachoek [7].

3. O Modelo de Comunicação do Ambiente RIO

Os protocolos de comunicação, em geral, permitem que alguns de seus parâmetros sejam negociados antes de se estabelecer comunicação. Parâmetros como o tipo de controle de fluxo, tamanho de janelas, tamanho de buffers, etc., podem ser negociados entre as partes envolvidas na comunicação. O modelo do ambiente RIO propõe que, além destes parâmetros, as transações e os protocolos a eles associados, possam, também, ser negociados e que, sobretudo, esta negociação possa ocorrer externamente aos módulos que serão envolvidos. O código do módulo é, então, programado de forma independente dos parâmetros e protocolos que irão prover a conexão entre eles. Somente no momento de configurar a aplicação os protocolos e seus parâmetros são selecionados e as conexões efetuadas. A faculdade de configurar e interferir na operação do sistema é possibilitada por uma interface [12] através da qual indica-se as ligações entre os módulos e os respectivos estilos de interconexão. Esta interface está associada a uma linguagem de configuração que possui comandos para ativar e desativar módulos, formar grupos e indicar conexões. O programador, agora, possui meios para testar uma aplicação em várias situações, utilizando diferentes protocolos e parâmetros para obter o desempenho mais adequado, de forma simples e concisa.

3.1. Conexão: Envolvendo Protocolos e Transações

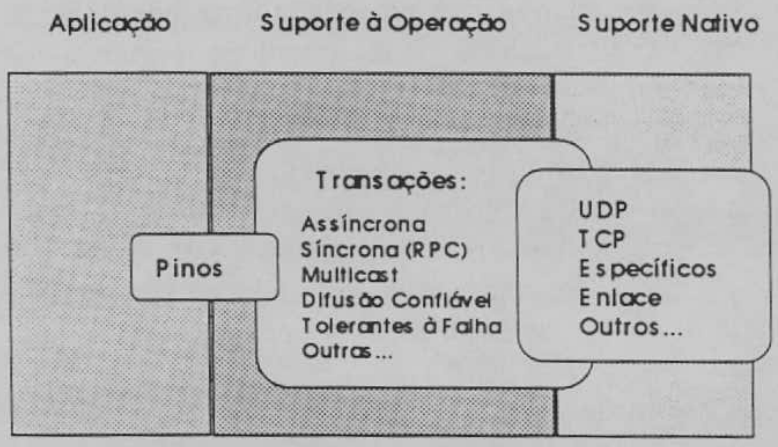

Figura 5: interação entre transação e protocolo

A conexão é um conceito que especifica completamente como dois pinos de um par de módulos irão se comunicar. Ela abrange o tipo de transação, os protocolos utilizados e outros atributos, como tolerância a falhas. Mesmo atuando em partes tão diversas do sistema a conexão é bastante consistente: o programador trabalha com transações ao prever a comunicação entre os módulos e o projetista da aplicação, no domínio da

configuração, seleciona os outros atributos. O esquema da fig. 5 ilustra a relação entre estes conceitos e sugere que existe uma superposição entre eles.

Para melhor ilustrar as vantagens deste modelo tomemos dois exemplos:

a) Determinada aplicação que utiliza transações assíncronas será executada em uma rede local de alta confiabilidade. Um protocolo baseado em datagrama seria a indicação natural para o transporte. Suponhamos, agora, que a mesma aplicação seja executada em uma rede que apresenta uma taxa de erros muito alta ou que esteja constantemente congestionada. Neste caso, um protocolo baseado em conexão (circuito virtual) seria mais adequado, devido aos controles de erros e fluxo nele embutidos.

b) Determinada aplicação precisa fazer uso de transações multicast. Se o sistema estiver fisicamente conectado por uma rede local, é bastante apropriada a utilização do recurso de *Broadcast* da própria rede local para o transporte de mensagens. Suponha, agora, que o sistema esteja conectado por uma rede geograficamente distribuída e que uma mensagem tenha que atravessar vários roteadores, com enlaces os mais variados. A utilização do mesmo mecanismo de transporte da situação anterior seria, neste caso, até mesmo prejudicial. Uma conexão ponto-a-ponto com cada módulo do grupo seria mais indicada. Em uma terceira hipótese, admitamos que esta mesma aplicação precise possuir propriedades de atomicidade e ordenação em suas transações. Um protocolo de difusão confiável seria o mais adequado para este caso.

Observe-se nas situações anteriores que, embora protocolos diferentes tenham sido selecionados para adequar o ambiente físico às aplicações, estas não são alteradas.

4. Sistema de Suporte e Comunicação

A arquitetura do sistema RIO é baseada em uma estrutura de camadas e no conceito de **micronúcleo (*microkernel*)** configurável. Uma unidade de execução completa é chamada de estação (fig. 6a). Cada estação contém uma instância do micronúcleo, o suporte à operação e módulos de aplicação. Um conjunto compacto de **primitivas**, para o suporte dos conceitos de módulo, conexão e troca de mensagens, está contido no micronúcleo, que encapsula as dependências do hardware ou sistema operacional, facilitando o transporte do sistema para outras plataformas. Os outros serviços e os módulos da aplicação são completamente suportados pelo micronúcleo, que também permite a execução concorrente de instâncias de módulos. São ainda incluídas no micronúcleo a interface com serviços de transporte, serviços de temporização e o

suporte à instanciação e comunicação entre instâncias locais.

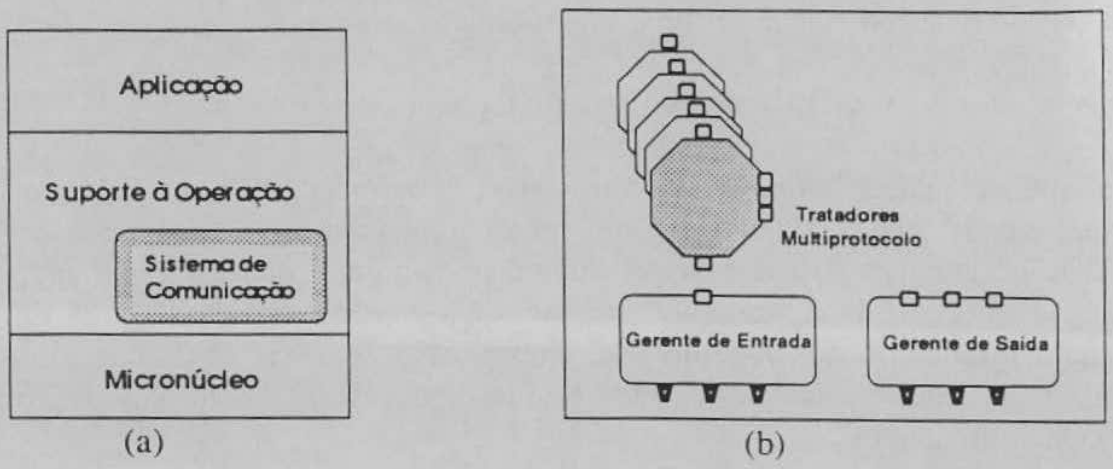

Figura 6: (a) arquitetura de uma estação, (b) sistema de comunicação.

A camada de suporte à operação contém o serviço de configuração local e os módulos do sistema de comunicação encarregados do transporte das mensagens no ambiente distribuído (fig. 6b). O serviço de configuração local provê a capacidade de criação de instâncias de módulos e ligação entre seus conectores. No sistema pode-se ainda identificar um serviço de configuração global que processa descrições abstratas de sistemas e gera os comandos de configuração local. Ainda nesta camada funciona o serviço de nomes, que permite identificar unicamente os tipos de módulos disponíveis e as instâncias de um sistema em operação. O sistema de comunicação implementa o suporte às diversas transações e protocolos disponíveis para uso na construção dos módulos das aplicações. Nas próximas subseções são descritos os módulos integrantes do sistema de comunicação.

4.1. O Tratador Multiprotocolo

O Tratador Multiprotocolo é encarregado de armazenar e entregar mensagens que vêm do sistema de comunicação para os módulos de destino e, no sentido inverso, mensagens de módulos de aplicação, que estão respondendo a um pedido inicial, para o sistema de comunicação. O Tratador Multiprotocolo se conecta dinamicamente a vários módulos do sistema de comunicação através de seus pinos (fig. 7), e é também responsável pela **demultiplexação** das mensagens recebidas, através de sua replicação em várias instâncias que executam concorrentemente. Também é função do Tratador Multiprotocolo implementar parte da semântica das transações, usando informações contidas nas próprias mensagens, para conduzi-las adequadamente. O Tratador Multiprotocolo é de especial importância na implementação de novas transações, como

a Difusão Confiável Síncrona [11], mecanismos de Tolerância a Falhas e Gerência de Configuração, como descrito em [1].

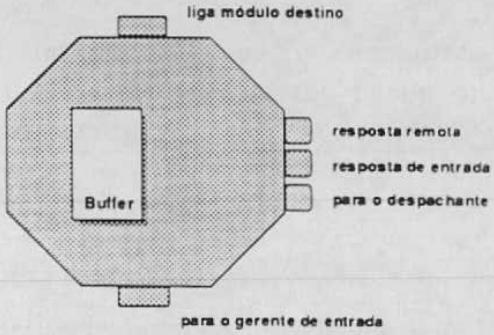

Figura 7: o Tratador Multiprotocolo

4.2. O Gerente de Entrada

O sistema de comunicação deve fornecer serviços uniformes, independentemente da plataforma na qual esteja implementado. O Gerente de Entrada padroniza o recebimento das mensagens vindas de camadas de Transporte ou de interfaces de rede. Cada mensagem recebida é repassada para um Tratador Multiprotocolo que a processa adequadamente. Nesta implementação, o Gerente de Entrada permite acesso a protocolos (atualmente UDP - *Unreliable Datagram Protocol* e TCP - *Transmission Control Protocol*) através de *sockets* dedicados. Além disso, possui canal dedicado para recebimento de mensagens de difusão da rede. Com alguns acréscimos, o Gerente de Entrada poderia ser preparado para receber mensagens de outros protocolos, como, por exemplo, CLNP, TP4 ou VMTP [5] (fig.8).

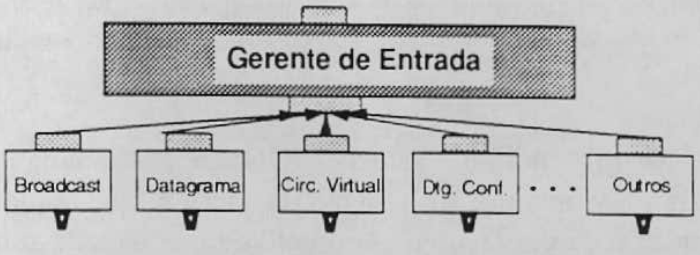

Figura 8: o Gerente de Entrada

Da mesma forma, em um sistema onde um serviço de transporte não seja disponível ou adequado, podem ser introduzidos tratadores de interrupção associados a diversas tecnologias de rede, e.g., Ethernet, Token-Ring, ou utilizar-se diretamente mecanismos de Enlace. Neste caso, os protocolos de transporte seriam implementados internamente ao sistema de comunicação, similarmente ao que foi proposto em [6]. Este mesmo

esquema permite a concepção de novos protocolos de Transporte, atendendo a necessidades específicas de determinadas aplicações, a exemplo do datagrama confiável abordado anteriormente.

O Gerente de Entrada é composto por um módulo principal e vários *daemons* dedicados ao processamento inicial das mensagens recebidas. A existência de um *daemon* para cada tipo de transporte não é obrigatória. Contudo, esta estrutura foi escolhida por razões de modularidade.

4.3. O Gerente de Saída

O Gerente de Saída faz o papel inverso ao do Gerente de Entrada. Ele é encarregado de receber mensagens de módulos de aplicação que estejam iniciando uma transação, ou de um Tratador Multiprotocolo quando a transação estiver em curso, e despachá-las pela rede através do protocolo adequado. Além de oferecer serviços diferenciados, dependendo da porta de entrada pela qual a mensagem chega, o Gerente de Saída obtém, da estrutura de dados relacionada à conexão, informações sobre a forma pela qual a mensagem deve ser transmitida. Informações como o tipo de protocolo de transporte, se é uma mensagem para um destinatário único ou se é para ser difundida, entre outras, vão desencadear ações específicas. Estas informações são também inseridas na mensagem e posteriormente usadas, no lado receptor, para guiar seu tratamento.

Figura 9: o Gerente de Saída

O esquema da fig. 10 apresenta duas estações RIO realizando uma transação síncrona, sob o ponto de vista dos módulos do sistema de suporte e comunicação. Isto permite que se tenha uma noção da operação do conjunto de módulos de suporte à operação e aplicação. Inicialmente, o módulo produtor na estação *Draco* envia uma mensagem diretamente para o Gerente de Saída (1), que é transportada via rede Ethernet para a estação *Tucana* (2). Na estação *Tucana* o Gerente de Entrada recebe a mensagem e encarrega (3) um Tratador Multiprotocolo da entrega desta ao módulo consumidor (4). O módulo consumidor, por sua vez, recebe a mensagem e gera uma resposta (5). O próprio Tratador Multiprotocolo leva a resposta para o Gerente de Saída (6) para

que a mesma seja enviada para a estação *Draco* (7). Após ser recebida pelo Gerente de Entrada da estação *Draco* (8), a mensagem de resposta é levada por um Tratador Multiprotocolo para o módulo produtor (9), que aguardava a chegada da resposta. Assim a transação se completa.

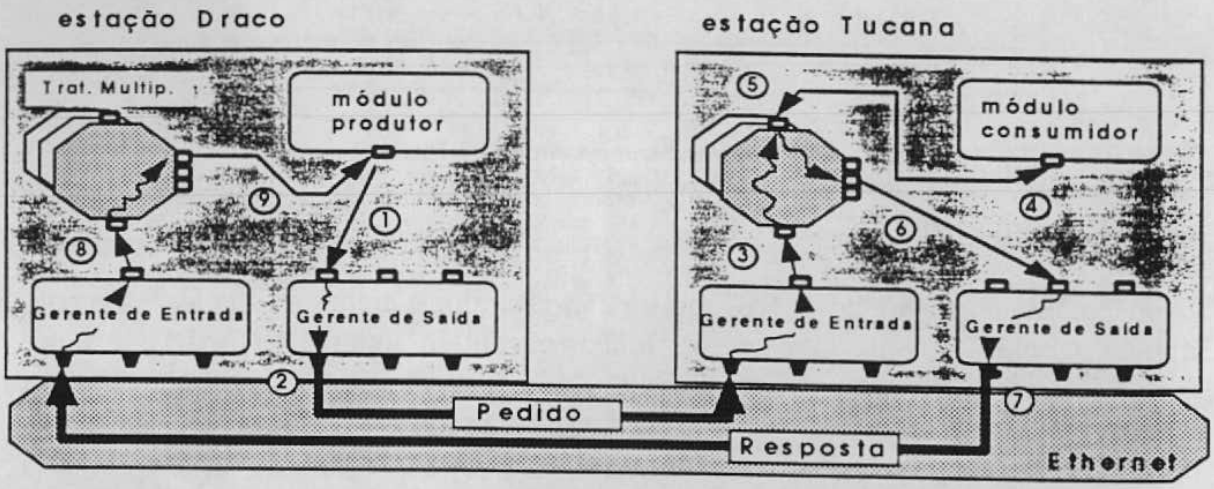

Figura 10: transação síncrona e o Sistema de Comunicação

4.4. O Quadro da Mensagem do Ambiente RIO

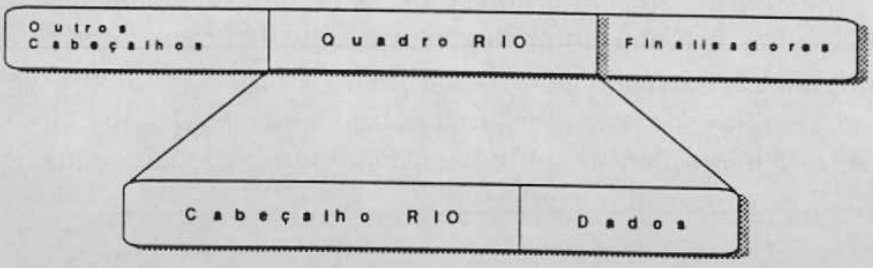

Figura 11: o quadro do ambiente RIO em relação ao quadro básico

Em um ambiente distribuído, baseado em troca de mensagens, é necessário que o quadro da mensagem carregue informações relativas aos protocolos que gerenciam o seu transporte. Normalmente estas informações estão dispostas em um cabeçalho, antes da mensagem e em um finalizador da mensagem. Quando um quadro chega ao seu ponto de destino cada nível de protocolo retira as informações necessárias e este nível e passa o quadro para a camada superior. Este processo continua até que a apenas mensagem original chegue ao processo de destino. A seguir serão examinados os campos do quadro RIO (fig. 12) e a relação destes com os serviços de comunicação.

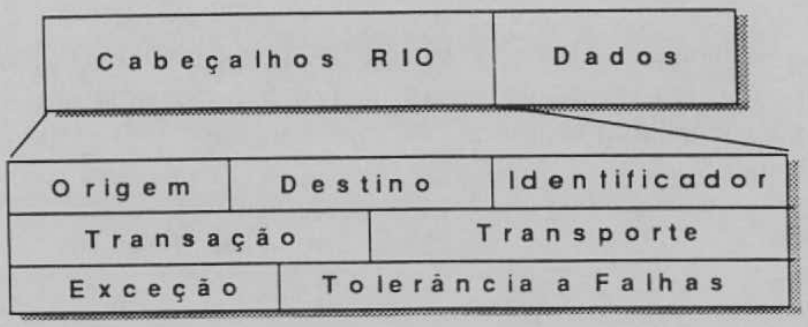

Figura 12: quadro do ambiente RIO

4.4.1. Endereçamento

O endereçamento no ambiente RIO é feito de forma transparente à tecnologia de rede adotada. Como o sistema de teste está implementado em uma rede TCP/IP existe um serviço interno ao Sistema de Comunicação que mapeia os endereços IP relevantes ao sistema, isto é, de máquinas físicas que possuem uma estação RIO ativa, nos endereços das portas de transporte (*sockets* ou equivalente) onde os serviços de comunicação são oferecidos. Os endereços no sistema são hierárquicos e possuem 3 campos **<Estação.Módulo.Pino>**. Desta forma, a ligação entre dois módulos requer o par de endereços completos pino-a-pino. O endereço composto pelo endereço básico (relativo à tecnologia da rede) e o endereço lógico (referente ao RIO) é armazenado por um serviço de endereçamento e nomes [12], tornando rapidamente disponível o endereço completo de determinado módulo. No quadro são transportados o endereço do módulo originador da mensagem, em **origem,** e do destinatário, em **destino.** Ambos são necessários no caso de transações síncronas, casos de exceção e retransmissões. O campo **identificador** contém informações pertinentes à transação em curso, como foi apresentado na subseção 2.1.

4.4.2. Transação

Mesmo tendo características atômicas, uma transação passa por vários estados, devido à natureza distribuída do ambiente. Por outro lado, a modularidade do ambiente impõe que estados não sejam armazenados e que servidores não fiquem dedicados para determinada transação. Por isso, as mensagens também carregam consigo informações sobre o estado da transação. Desta forma, os módulos de suporte à operação retiram do próprio quadro da mensagem informações a respeito do tipo de transação e em que parte da transação a mensagem se encontra. As alternativas de seqüência da máquina de estados a serem seguidas pela mensagem são selecionadas baseadas nestas informações.

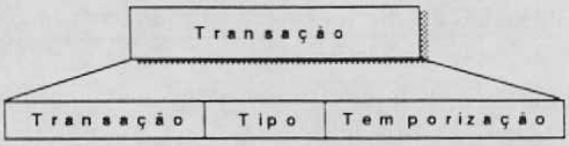

Figura 13: campos utilizados na transação

O campo **tipo** identifica o tipo básico da transação em que a mensagem está envolvida. Os tipos básicos disponíveis são o **síncrono** e o **assíncrono**. O campo **transação** identifica qual o estado atual em que se encontra a transação. Por exemplo, a transação **síncrona** pode estar em três estados primitivos, identificados por **M_Request**, **M_Reply** e **M_Exception**, indicando uma mensagem de requisição, de resposta ou de ocorrência de exceção, respectivamente. O campo **temporização** permite que se associe um prazo máximo para a duração de uma transação. Outras formas de interpretação para os campos tipo e transação são usadas na identificação de diferentes tipos de transações, como, por exemplo, o multicast confiável.

4.4.3. Transporte

O quadro contém um campo com a informação do protocolo utilizado para o transporte da mensagem. Normalmente, este protocolo será efetivamente de Transporte, mas poderá ser utilizado diretamente um protocolo de Enlace ou um protocolo específico implementado especialmente para determinada situação. O campo **transporte** identifica, então, o protocolo pelo qual esta sendo transportada a mensagem. Atualmente, **UDP**, **TCP** e **BCAST** representam transporte baseados em datagrama, circuito virtual e *broadcast*, respectivamente.

4.4.4. Tratamento de Exceções

O tratamento de exceções possui um campo particular no quadro da mensagem, que descreve uma exceção específica ocorrida. Existem alguns tipos de exceção *hard-coded*, já previstos no nível desuporte à operação. O programador poderá, também, criar seus próprios códigos de exceção, para tratar erros definidos e detectados no nível da aplicação.

No esquema da fig. 14 podemos visualizar algumas situações de exceção. Em uma transação sem erros, existe uma mensagem de pedido e eventualmente uma de resposta. A primeira situação de erro ocorre no suporte à operação da própria estação do módulo originador, antes mesmo da mensagem chegar a outra estação (por exemplo, a tentativa de enviar uma mensagem para um pino não conectado ou um circuito virtual que não conseguiu ser estabelecido). Na estação remota, também,

podem ocorrer situações de exceção no suporte à operação (por exemplo, a mensagem destina-se a um módulo que não existe ou que foi desativado) e na Aplicação (um dado incorreto ou não esperado, por exemplo), sendo esta última prevista e tratada pelo programador da aplicação. A última situação está associada a erros não previstos diretamente pelo sistema, e resultarão em um *time out*.

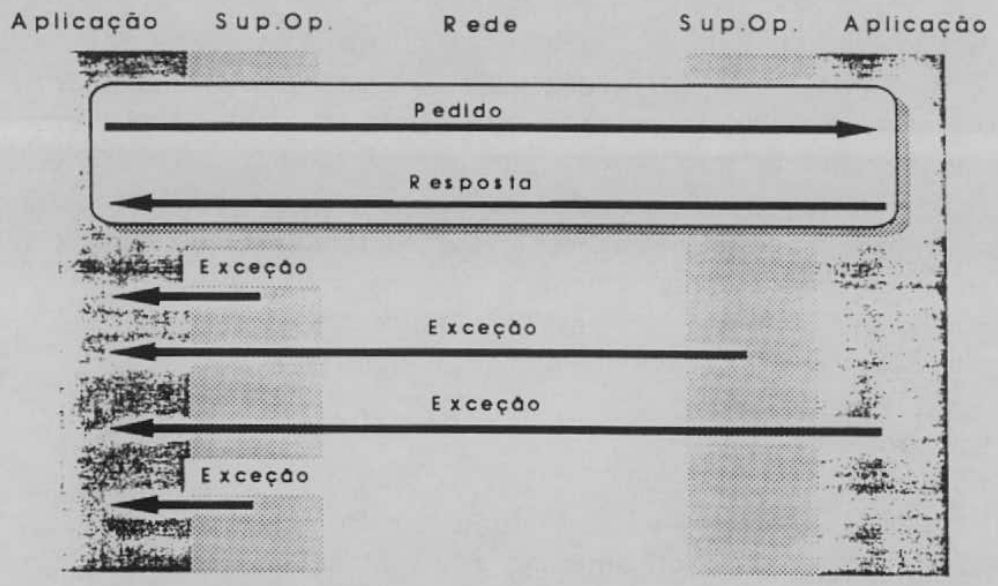

Figura 14: situações de exceção

4.4.5. Tolerância a Falhas

O sistema permite que a aplicação selecione o nível de tolerância a falhas para a operação de determinados módulos. Várias técnicas podem coexistir no ambiente, proporcionando um ajuste fino no nível de segurança de funcionamento de cada aplicação ativa. Desta forma, a mensagem leva um código relativo ao tipo de tolerância a falhas dos módulos envolvidos na aplicação a qual ela pertence. Estão disponíveis, atualmente, algumas técnicas baseadas em Replicação Ativa e Passiva. Maiores detalhes sobre tolerância a falhas no sistema podem ser vistos em [1].

5. Questões de Desempenho no Ambiente RIO

Uma das preocupações ao se construir um sistema modular é quanto à uma possível queda no desempenho devido aos *overheads* inerentes à modularidade (cópia de dados entre espaços de endereçamento diferentes, chaveamentos de contexto, etc.). Em consequência, a validade do conceito de micronúcleo é questionada com frequência [8]. Devido à natureza experimental e à proposta de permitir flexibilidade nos serviços

do ambiente RIO, a modularidade é imperiosa. Daí a escolha, entre outras razões, de uma estrutura com micronúcleo para o sistema.

A principal motivação para a obtenção de dados sobre desempenho é a comparação entre resultados de testes do ambiente RIO e resultados de sistemas com propostas semelhantes (Mach e Chorus, por exemplo), publicados em relatórios técnicos [4,8]. O objetivo é medir o *overhead* do ambiente RIO em relação ao sistema nativo, UNIX, bem como identificar quais são os módulos ou partes de código que estão causando este *overhead*. Experimentos iniciais demonstraram que, em alguns casos, o ambiente RIO obtém resultados superiores ou comparáveis aos outros sistemas. Contudo, na maioria dos testes, sistemas utilizando RPC e *sockets* diretamente oferecem desempenho superior.

5.1. Threads de Controle

O sistema foi concebido segundo o conceito de processos ***LightWeight,*** onde múltiplas ***threads de controle*** (processos *LightWeight*) executam concorrentemente, com um chavemento que exige pouco esforço computacional se comparado com processos UNIX padrão. Cada instância de módulo do sistema é mapeada em uma *thread de controle*. Até o presente momento este mecanismo é oferecido apenas como bibliotecas que devem ser ligadas aos programas, executando em modo usuário, com desempenho limitado. Por exemplo, isto ocorre no sistema operacional SunOS e sua biblioteca LightWeight Processes [9], sobre os quais foi implementado o RIO.

5.2. A Avaliação

Os testes, descritos nesta subseção, foram baseados em transações assíncronas programadas em uma aplicação modelo **produtor/consumidor** com protocolo UDP. A duração média de 10 laços de 100 transações completas (*round trip*) foi medida para cada sistema a ser avaliado. O ambiente físico constou de uma rede Ethernet, interligando estações de trabalho Sun (SPARC SLC). A rede foi mantida com tráfego praticamente exclusivo ao teste, mas o com ambiente de trabalho o mais próximo possível do usual em aplicações típicas. Além do *round trip* também foi medido o tempo de cópia de memória para memória e o tempo de chaveamento, quando executado explicitamente, entre duas *threads*, para determinar com que parcela estes dois fatores contribuem para o tempo total de *round trip*. As tabelas, a seguir, apresentam alguns dos resultados obtidos nos testes.

Na primeira coluna da Tabela 1 temos os sistemas alvo, sendo que **Socket** representa um processo UNIX que utiliza apenas as funções de *sockets*; **LWP** um sistema que

utiliza a biblioteca LightWeight Processes e *sockets*; **RPC** é um sistema implementado apenas com o RPC do UNIX e **RIO** representa o ambiente RIO. As duas outras colunas contém as medidas realizadas em uma mesma estação e em estações diferentes, respectivamente. Observa-se que, devido a uma otimização do sistema, as transações que ocorrem na mesma estação RIO são bastante rápidas, mas que, por outro lado, quando realizadas entre estações distintas, o *round trip* do RIO é alto, comparado com os dos outros sistemas.

Sistema Alvo	Draco / Draco (ms)	Draco / Tucana (ms)
Socket X Socket	2,5	2,5
RPC	2,9	2,9
LWP X LWP	5,7	4,7
RIO X RIO	0,78	10,5
LWP X Socket	3,7	3,7
Socket X LWP	4,0	3,7

Tabela 1: medidas de round trip

A tabela 2 apresenta o tempo medido para a cópia de uma seqüência de bytes entre duas áreas de memória distintas dentro da mesma estação, utilizando-se a função *memcpy*. A Tabela 3 apresenta o tempo de duração da troca de contexto entre duas *threads* (a primitiva *lwp_yield*, da biblioteca *LightWeight Processes*, força o chaveamento entre a *thread* corrente e a próxima *thread* pronta para executar).

memcpy (S1,S2,n)	0 byte	1 byte	10 bytes	100 bytes	500 bytes	1000 bytes
Draco (µ seg)	620	624	624	640	700	775

Tabela 2: tempo de cópia de memória para memória

Tempo de Chaveamento da LWP	Draco (µ seg)
lwp_yield	100

Tabela 3: tempo de chaveamento

Alguns dados relevantes podem ser levantados imediatamente:

- o tempo de cópia de memória-para-memória e o tempo de chaveamento entre threads não parecem ser os principais consumidores de tempo (640 e 100 µs, respectivamente em relação a 10 ms de *round trip*);
- o tempo de transporte da mensagem pela rede, em torno de 2,5 ms representa aproximadamente 25% do tempo total de *round trip* (retirado da Tabela 1, comparando-se a 1ª linha com a 4ª);

- existe uma variação muito grande na ordem de grandeza do tempo de *round trip* de LWP x LWP (3ª linha da Tabela 1) e RIO x RIO (4ª linha da Tabela 1), embora os dois sistemas utilizem os mesmos recursos;
- pode-se diagnosticar, baseado nos pontos anteriores, que não são os recursos utilizados, individualmente, os responsáveis pelo *overhead* do sistema, mas a interação entre estes parece estar, de alguma forma, atrasando o sistema como um todo.

As próximas tabelas são retiradas de artigos publicados a respeito do sistema Mach [8] e permitem uma comparação de desempenho com RIO. Podemos verificar que na situação em que os dois sistemas mais se aproximam (que seria o UNIX - UX29 - emulado sobre o sistema Mach na versão 2.5), as medidas de *round trip* são da mesma ordem de grandeza (Tabela 4). Nota-se também uma diferença significativa entre as medidas que envolvem apenas o sistema de suporte e as medidas que abrangem o usuário final (linhas 3 e 4 da Tabela 5), resultado similar ao do RIO:

Unix server (IPC, no VM)	14.3 ms
Unix server (syscall, no IPC, no VM)	13.6 ms
Unix server (mapped driver)	13.1 ms

Tabela 4: *round trip* para UDP, executando-se o Mach 3.0 com uma versão experimental do UX29, na primeira linha, ou com extensões como o mapeamento do suporte a Ethernet para o espaço de endereçamento do usuário [6].

New UDP Server (user-to-user)	4.2 ms
New UDP Server (server-to-server)	4.0 ms
Mach 2.5 kernel	4.8 ms
Unix server (IPC, VM)	19.5 ms

Tabela 5: *round trip* para UDP: na 1ª linha com servidores especiais executando em cada máquina; na 2ª linha é eliminado o IPC do sistema Mach, que não envolve o usário final, apenas o sistema de comunicação; na 4ª linha temos a medida para o servidor Unix do Mach versão 2.5, que poderia ser comparado ao ambiente RIO em termos de utilização dos recursos do sistema.

Após serem concluídas as medidas e a análise comparativa com outros sistemas, os mesmos testes foram executados habilitando-se o sistema de *tracing*. O sistema de *tracing* estende o tempo de execução das transações, embora não prejudique a proporcionalidade dos tempos amostrados [2]. Prosseguindo a avaliação, serão apresentadas algumas amostragens destes testes, em forma gráfica, representando a atividade dos módulos do sistema de suporte à operação e de aplicação. Na amostragem da fig. 15 observa-se, de forma macroscópica, a evolução das transações produtor/consumidor, vista do lado produtor. Verifica-se com clareza o momento do chaveamento entre os módulos, que estão identificados do lado esquerdo do gráfico.

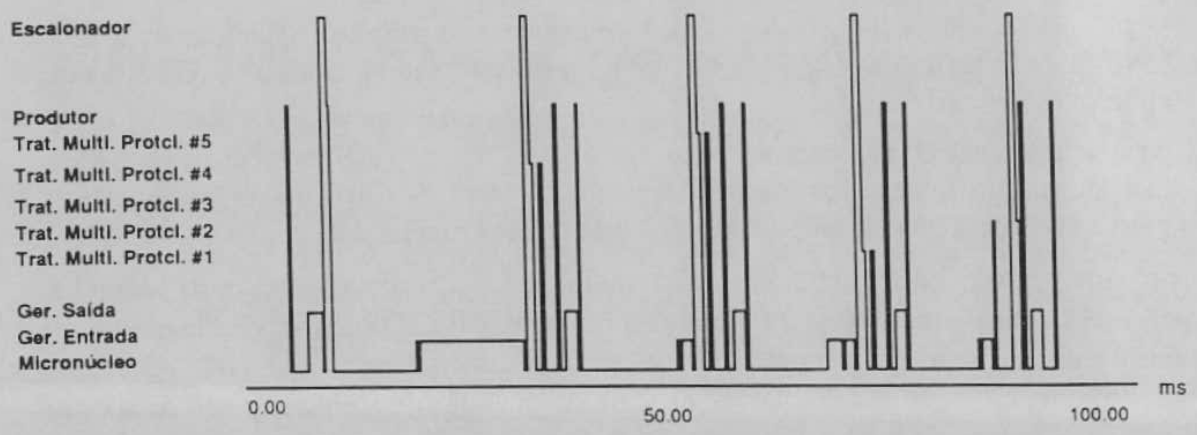

Figura 15: sequência de transações (lado produtor)

A análise do gráfico de uma única transação, vista do lado consumidor (fig. 16), permite extrair outras informações sobre o desempenho do sistema. Podemos constatar que a ordem em que os módulos são escalonados apresenta a coerência esperada (como visto na subseção 4.1, vários Tratadores Multiprotocolos são previstos; a alocação dos mesmos é realizada arbitrariamente). Por outro lado, identificam-se trechos onde o sistema apresenta um tempo de execução dilatado em proporção à duração total de cada transação. Os pontos assinalados com *1* possuem chamadas a *socket* para enviar uma mensagem pela rede, e os assinalados com *3* chamadas a *socket* para receber uma mensagem pela rede. Nota-se que o tempo gasto em cada uma destas chamadas é da ordem de 1 a 2 ms, o que é comparável ao tempo de duração de um *round trip* obtido na medida **Socket x Socket**, listada na Tabela 1. Isto mostra que chamadas a *sockets* dentro de processos LightWeight, gastam mais tempo do que se realizadas dentro de processos UNIX normais, prejudicando o desempenho das transações RIO.

Verifica-se, também, uma duração excessiva e bastante variável quando o **Micronúcleo** (pontos assinalados com *2*) e o **Escalonador** (pontos assinalados com *4*) estão executando. Nestes trechos o sistema de suporte efetua chamadas a funções da biblioteca LightWeight Processes para implementar o escalonamento de *threads*[2] e primitivas de temporização. Analisando-se o comportamento do Micronúcleo e do Escalonador, concluiu-se que estes atrasos são causados pela inabilidade da biblioteca LightWeight Processes passar o controle imediatamente para a próxima *thread*, pronta para executar.

---

[2]Devido a uma particularidade, foi utilizada a função lpw_suspend para implementar o escalonamento, ao invés da função lwp_yield. A função lpw_suspend bloqueia a thread, mas não passa o controle imediatamente, como seria esperado, para a próxima thread pronta para executar.

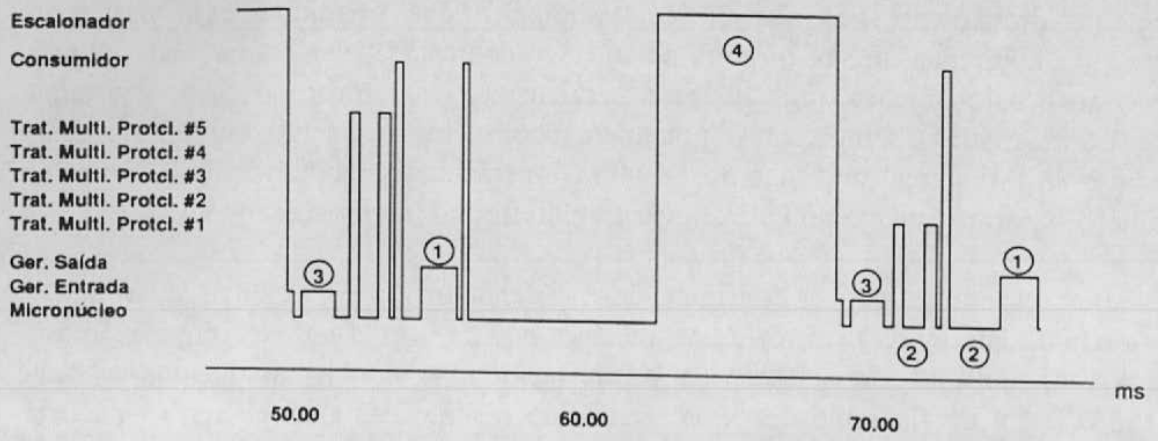

Figura 16: amostra de uma transação (lado consumidor)

A reunião destes dados permite a inferência de que grande parte do desempenho atual é limitado por características do próprio sistema operacional (biblioteca de *threads* + UNIX), que não delega à aplicação muito controle sobre o escalonamento (que é executado pelo executivo da própria biblioteca). Sempre que é feita alguma chamada a recursos do UNIX, o RIO demora a reassumir o controle. Além disso, o sistema é demasiadamente impactado pelo fato de não haver suporte a *threads* dentro do núcleo do UNIX, pois o sistema é executado como um processo de usuário, sem maiores privilégios e dentro de um mesmo processo UNIX. Sempre que outro processo UNIX é escalonado para executar, todo o RIO é suspenso. Sem estas restrições o sistema RIO teria o tempo de *round trip* reduzido, sem maiores esforços.

6. Conclusão

No decorrer de nosso projeto ficamos cada vez mais convencidos da utilidade dos conceitos associados à metodologia de construção de software. A possibilidade de selecionar transações e protocolos no domínio da configuração facilita o atendimento dos requisitos das aplicações distribuidas. O projetista da aplicação pode realizar sua tarefa sem se preocupar com detalhes da comunicação, em baixo nível, que implicariam em modificações na codificação interna dos módulos. Além disso, abstrações poderosas não comumente oferecidas em ambientes convencionais podem ser incluidas no nível de suporte de operação e utilizadas transparentemente pelas aplicações. Como exemplos, temos protocolos de disseminação confiável e replicação de módulos para tolerância a falhas.

O desempenho do sistema de comunicação poderá ser otimizado se o sistema operacional for atualizado para uma versão mais recente, Solaris SunOS 5.0, que possui uma arquitetura especial para o suporte de *threads* [10]. Além de serem

suportadas diretamente pelo *kernel* (do Solaris), o que permite acesso direto as funções de comunicação, as *threads* podem ser controladas de forma mais efetiva quanto ao escalonamento, prioridades e paralelismo, se comparado com o sistema atual. Características semelhantes, juntamente com escalonamento em tempo real, tendem a se tornar padronizadas nos sistemas operacionais modernos, o que deverá facilitar o suporte eficiente do RIO em várias plataformas de processamento.

Uma interface gráfica para a configuração e gerenciamento de sistemas distribuídos está sendo desenvolvida. Através desta interface poderá ser definida a composição dos módulos da aplicação e selecionada a transação e protocolo associados a cada conexão. Além da flexibilidade de configuração a aplicação poderá ser visualizada quando em operação, facilitando avaliações de desempenho e correções ao nível de cada conexão.

7. Agradecimentos

Os autores gostariam de agradecer o incentivo e a colaboração recebidas dos colegas da PUC/RJ e ao apoio financeiro recebido do CNPq e do MCT.

8. Bibliografia

[1] Brito, O. F. G, Malucelli, V. V. e Loques, O. G, "Tolerância a Falhas no Ambiente RIO", V Simpósio em Computadores Tolerantes a Falhas, São José dos Campos, outubro de 1993.

[2] Chen, J. B., "Software Methods for System Address Tracing", Fourth Workshop on Workstation Operating Systems, Napa, CA/EUA, outubro de 1993.

[3] Cristian, F., "Synchronous Atomic Broadcast for Redundant Broadcast Channels", The Journal of Real-Time Systems, 2, pp. 195-212, 1990.

[4] Dean, R. W. e Armand, F., "Data Movement in Kernelized Systems. A Look at Chorus and Mach 3.0 Mapped Files", Carnegie Mellon University

[5] Doringer, W.A. et alli, "A Survey of Light-Weight Transport Protocols for High-Speed Networks", IEEE Transactions on Communications, 38(11), pp.2025-2039, novembro de 1990.

[6] Hutchinson, N.C. e Peterson, L.L, "The x-Kernel: An Architecture for Implementing Network Protocols", IEEE Transactions on Software Engineering, 17 (1), janeiro de 1991.

[7] Kaachoek, M.F. et alli, "An Efficient Reliable Broadcast Protocol", Operating Systems Review, 23(4), p. 5-19, outubro de 1989.

[8] Maeda, C., Bershad, B. N., "Networking Performance for Microkernels", Proceedings of the Third Workshop on Workstation Operating Systems (WWOS-3), abril de 1992.

[9] Sun Microsystems, "Programming Utilities and Libraries", Revision A, março de 1990.

[10] Powell, M.L. et alli, "SunOS 5.0 Multithread Architecture - A White Paper", Sun Microsystems, 1991

[11] Sztajnberg, A. e Loques, O. G., "O Sistema de Comunicação Multi-Protocolo do Ambiente RIO", V Simpósio de Computadores Tolerante a Falhas, São José dos Campos, outubro de 1993.

[12] Werner, J.A.V. e Loques, O. G., "Ambiente RIO: Metodologia e Suporte para Sistemas Configuráveis", XX SEMISH - Seminário Integrado de Software e Hardware, Florianopólis, outubro de 1993