UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

Posicionamento de Réplicas em Sistemas Distribuídos

por

ANDRÉ ZAMPIERI

Dissertação submetida à avaliação,
como requisito parcial para a obtenção do grau de
Mestre em Ciência da Computação

Profa. Dra. Taisy Silva Weber
Orientadora

Porto Alegre, julho de 2001.

CIP – Catalogação na Publicação

Zampieri, André
Posicionamento de Réplicas em Sistemas Distribuídos/André Zampieri. – Porto Alegre: PPGC da UFRGS, 2001.
82 f. :il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR – RS, 2001. Orientador: Weber, Taisy Silva.

1.Posicionamento de Réplicas. 2.Sistemas Distribuídos 3.Comunicação de Grupos 4.Tolerância a Falhas I. Silva, Taisy. II. Título.

Agradecimentos

Agradeço aos meus pais, por toda a dedicação e amor demonstrados em todos estes anos de estudo, tendo sempre uma palavra de conforto nos momentos mais difíceis.

À minha namorada Ana Cláudia, o meu agradecimento muito especial, por ter me apoiado durante toda esta jornada, suportando meus desabafos, mau humor, sempre com suas palavras de compreensão e otimismo.

Aos meus amigos, colegas de trabalho e a todos que de certa forma me ajudaram e apoiaram neste desafio, mas que com certeza ocupam um espaço em meu coração. Pelas horas de conversa, desabafos, e também pela descontração prestada.

À Universidade de Caxias do Sul, pela ajuda prestada, fornecendo o suporte necessário para o desenvolvimento do trabalho.

À minha professora orientadora, Taisy Weber, pela orientação e dedicação durante este período.

Sumário

Lista de Abreviaturas

ATM	*Assynchronous Transfer Mode*
CPU	*Central Processing Unit*
E/S	Entrada/Saída
FDDI	*Fiber Distributed Data Interface*
FIFO	*First In First Out*
Fig.	Figura
GMS	*Group Membership Service*
IMP	Índice de Média de Processos
MTTF	*Medium Time to Failure*
MTTR	*Medium Time to Repair*
PPGC	Programa de Pós-Graduação em Computação
RAM	*Random Access Memory*
RMI	*Remote Method Invocation*
RMS	*Replica Management System*
RPC	*Remote Procedure Call*
RPM	*Replica Placement Manager*
UFRGS	Universidade Federal do Rio Grande do Sul

Lista de Figuras

Lista de Tabelas

Resumo

Replicação de objetos é usada para garantir uma maior disponibilidade de recursos em um sistema distribuído. Porém, com a replicação, surgem problemas como o controle da consistência das réplicas e onde estas réplicas devem estar posicionadas. A consistência é garantida por um protocolo de consistência de réplicas.

Para facilitar a implementação dos protocolos de controle de réplicas, pode-se utilizar mecanismos de comunicação de grupo como suporte para a replicação.

Outro problema importante que surge com a replicação é o posicionamento das réplicas. A carga de processamento em um sistema distribuído muda continuamente e num determinado instante pode ser necessário mudar a distribuição atual das réplicas pela adição de novas réplicas, remoção de réplicas desnecessárias ou pela mudança de posicionamento das réplicas. Um sistema de gerenciamento de réplicas pode realizar esta tarefa.

Este trabalho apresenta o sistema RPM – *Replica Placement Manager* – responsável por fornecer ao serviço de gerenciamento de réplicas uma lista ordenada de nodos potencialmente ideais, num determinado momento do processamento, para receber uma réplica de um objeto. Esta lista é criada pelo RPM, considerando um pequeno conjunto de variáveis estáticas e dinâmicas, facilmente obtidas nos nodos do sistema distribuído.

Palavras-Chave: Sistemas Distribuídos, Tolerância a Falhas, Comunicação de grupos, Replicação, Posicionamento de Réplicas.

TITLE: “REPLICAS PLACEMENT IN DISTRIBUTED SYSTEMS”

Abstract

The object replication is used to improve resources availability in a distributed system. However, with replication, problems like the replicas’s consistency control and where these replicas must be placed can appear. The replicas consistency is garanteed by a replicas consistency protocol.

To make the implementation of the replicas´s control protocol easier, a group communication tool as support for the replication can be used.

Another problem that arises with the replication is replicas placement. The processing load in a distributed system changes continually and sometimes can be necessary to change the current replicas’s distribution by adding new replicas, removing old ones or changing the replicas placement. A replica management service can do this job.

This work introduces the RPM system – Replica Placement Manager – responsible to provide to the replicas’s management service a sorted list of nodes, that are considered ideals in a determined moment of the processing to receive an object’s replica. This list is created by the RPM considering a small set of static and dynamic variables easily obtained in the nodes of the distributed system.

Keywords: Distributed Systems, Fault Tolerance, Group Communication, Object’s Replication, Replica Placement.

1 Introdução

No panorama de descentralização de grandes bases de dados e uso em larga escala de aplicações na Internet surge a necessidade de serviços com alta disponibilidade, confiabilidade e desempenho. Estes objetivos podem ser atingidos com a construção de sistemas distribuídos com características de tolerância a falhas.

Sistemas distribuídos tolerantes a falhas utilizam a replicação de dados e serviços para manter a disponibilidade e, sob certas condições, o desempenho. Quando um nodo que possui um objeto crítico ao sistema falha, uma réplica desse objeto, localizada em outro nodo do sistema, pode assumir de forma transparente ao usuário, a execução das requisições direcionadas ao objeto no nodo falho. A melhora no desempenho é alcançada quando diferentes clientes trabalham com réplicas de um mesmo objeto, usando acesso concorrente. Porém, em sistemas com predominância de requisições de escrita de dados, a replicação pode não melhorar o desempenho, devido à necessidade de atualização de diversas cópias de um objeto, podendo gerar um grande *overhead* no sistema.

Com a implantação de réplicas, surge o problema da manutenção da consistência dos objetos replicados. Isso leva à necessidade da utilização de métodos de controle que garantam a integridade desses objetos. As duas classe fundamentais de protocolos para implementar o controle de consistência das réplicas são o ***protocolo primário-backup*** e o ***protocolo de réplicas ativas*** [SCH93] [BUD93] [JAL94] [GUE97].

Para facilitar a implementação dos protocolos de controle de réplicas, pode-se utilizar mecanismos de comunicação de grupo como suporte para a replicação. O propósito do conceito de grupo é permitir que processos tratem conjuntos de objetos como uma simples abstração. Assim sendo, um processo pode enviar uma mensagem para um grupo de forma transparente, sem ter de saber quantos são estes objetos ou onde seus servidores estão localizados.

Outro problema importante que surge com a replicação é o posicionamento das réplicas. A carga de processamento em um sistema distribuído muda continuamente e algumas vezes o posicionamento de uma réplica em um determinado nodo não é mais desejável. Pode ser necessário mudar a distribuição atual das réplicas pela adição de novas réplicas, remoção de réplicas desnecessárias ou pela mudança do posicionamento das réplicas. Um sistema de gerenciamento de réplicas realiza estas tarefas.

O ganho na disponibilidade, confiabilidade e desempenho, que são atingidos pela replicação de objetos é uma função complexa, dependente de vários fatores, tais como [LIT94]:

- número e posicionamento de réplicas;
- natureza das transações realizadas sobre os objetos;
- a carga média dos nodos onde as réplicas estão posicionadas;
- os protocolos de consistência e recuperação para manter as réplicas;
- as interdependências dos objetos;
- as características de disponibilidade e desempenho das máquinas e da rede que compõem o sistema.

Esses fatores mostram que a escolha de um nodo para receber uma réplica deve ser baseada em informações precisas obtidas dinamicamente do sistema e não determinada de forma empírica pelo administrador do sistema.

1.1 Trabalhos Sobre Replicação Desenvolvidos no PPGC

Vários trabalhos sobre replicação já foram desenvolvidos no PPGC da UFRGS. Dentre estes pode-se citar o de João Carlos Ferreira Filho [FJC2000], orientado pela professora Maria Lúcia Blank Lisbôa e co-orientado pela professora Ingrid Jansch-Pôrto. Este implementou a replicação de objetos através da utilização da linguagem Java e do seu sistema de invocação remota de métodos (*Remote Method Invocation* – RMI). Usando RMI, a implementação de grupos de objetos pode ser estruturada de acordo com a arquitetura cliente/servidor, sendo o cliente o representante (a interface) de um grupo de objetos e os servidores representam os demais componentes do grupo. A linguagem Java foi adotada por oferecer uma grande portabilidade, eficiência e facilidades para a criação e utilização de bibliotecas de componentes.

Outro trabalho, desenvolvido por Débora Nice Ferrari [FDN2001], orientada pelo professor Cláudio Geyer, propõe um modelo de replicação em ambientes que suportam mobilidade. Este modelo propõe uma camada de abstração, permitindo que a implementação da replicação seja facilitada. Deve-se ressaltar que, além de propor uma camada de abstração, outro objetivo deste trabalho é utilizar a replicação para proporcionar um desempenho satisfatório para as aplicações.

Sob a orientação da professora Taisy Weber, vários trabalhos que utilizam um sistema de comunicação de grupos como suporte à replicação foram desenvolvidos. A aluna de doutorado Marcia Pasin [PAS98] implementou um sistema de arquivos replicado e distribuído em um ambiente de comunicação de grupos e atualmente está projetando um sistema de gerenciamento de réplicas, o sistema *Ape*, apresentado no capítulo quatro. Outro trabalho, também orientado pela professora Taisy Weber é o de Lenise Geiss [GEI2000], que implementou os protocolos de consistência de réplicas – primário-backup e réplicas ativas – sobre um sistema de comunicação de grupos.

Os trabalhos já desenvolvidos apresentam um enfoque no uso da replicação de objetos para garantir tolerância a falhas e melhorar o desempenho das aplicações. Esta dissertação diferencia-se das citadas por enfatizar o problema do posicionamento das réplicas, tema não tratado nos trabalhos anteriores.

1.2 Objetivos do Trabalho

A quantidade de variáveis envolvidas no posicionamento de uma réplica em um nodo justifica a necessidade de um mecanismo que, através da análise de informações obtidas do sistema, consiga determinar qual é o melhor nodo para receber uma réplica, tirando essa responsabilidade do projetista do sistema. Assim, o principal objetivo deste trabalho é definir um conjunto pequeno e consistente de variáveis estáticas e dinâmicas que representem o estado dos nodos em um determinado momento do processamento do sistema e que possam ser usadas, de forma eficiente, por um serviço de gerenciamento de réplicas, responsável por determinar o posicionamento de réplicas de objetos em sistemas distribuídos.

Para validar este conjunto de variáveis foi desenvolvido um software, batizado de RPM – *Replica Placement Manager* – que consegue fornecer informações dos nodos do sistema distribuído para o serviço de gerenciamento de réplicas.

Quando o serviço de gerenciamento de réplicas decide criar uma nova réplica ou migrar uma existente, ele faz uma solicitação ao RPM. Este inquire os nodos componentes do sistema em relação a carga de trabalho, processa a escolha dos nodos

que atendem a solicitação e devolve o resultado, em forma de uma lista ordenada, ao serviço de gerenciamento de réplicas.

Os testes realizados mostraram que o RPM consegue indicar com exatidão quais os nodos mais leves do sistema distribuído, fornecendo corretamente a lista de resultados.

1.3 Organização do Trabalho

Este trabalho está dividido como segue:

No capítulo dois são apresentados os fatores que devem ser levados em consideração para a tomada de decisão sobre o posicionamento de réplicas. Também nesta seção são apresentados os principais protocolos de consistência de réplicas e o seu comportamento na presença de falhas de nodos.

O capítulo três introduz os principais conceitos relacionados a sistemas de comunicação de grupos. São apresentados alguns trabalhos de replicação utilizando comunicação de grupos e é descrito o sistema de comunicação de grupos Ensemble.

No capítulo quatro é descrito o sistema *Ape*, que é um serviço de gerenciamento de réplicas, em desenvolvimento na UFRGS.

O sistema RPM é apresentado no capítulo cinco. Nesta seção, o leitor tem uma visão de como ele deve se comportar e quais os serviços que ele oferece ao sistema de gerenciamento de réplicas.

A implementação do RPM é discutida em detalhes no capítulo seis. Neste capítulo é apresentado o algoritmo do programa e cada função componente do programa é explicada detalhadamente.

No capítulo sete é apresentado o ambiente onde os testes foram realizados e os resultados dos mesmos.

Por fim, as considerações finais deste trabalho e trabalhos futuros são descritos no capítulo oito.

2 Conceitos Básicos de Posicionamento e Replicação

Este capítulo trata do problema do posicionamento de réplicas e dos protocolos para manter a consistência das mesmas. A seção 2.1 apresenta alguns aspectos que devem ser levados em consideração por uma estratégia de posicionamento de réplicas. Na seção 2.2 é descrito o funcionamento dos principais protocolos de replicação e seu comportamento na presença de falhas. Para o leitor familiarizado com estes protocolos não é acrescentada nenhuma nova contribuição.

2.1 Posicionamento de Réplicas de Objetos

O problema da determinação de onde criar (ou para onde migrar) uma réplica num sistema distribuído é mais complexo do que parece à primeira vista. O posicionamento de objetos em um sistema distribuído tem um efeito crítico na confiabilidade e desempenho das aplicações que usam estes objetos.

Na inicialização do sistema, o administrador poderia determinar o número de réplicas e o posicionamento destas. Porém, o administrador pode não ter a base técnica necessária para determinar o nível de replicação, uma vez que a disponibilidade não varia proporcionalmente ao número de réplicas (aumentar o número de réplicas não garante sempre uma maior disponibilidade). Para determinar o posicionamento deve-se conhecer as características de confiabilidade e desempenho dos nodos nos quais as réplicas serão posicionadas. A medida que o sistema progride ocorrem mudanças na topologia da rede ou na carga do sistema, dessa forma, o posicionamento de um objeto que estava correto quando a aplicação foi iniciada pode se tornar incorreto após transcorrido um tempo de execução. Isto exige um monitoramento constante pelo administrador, a fim de que, quando estas situações surgirem, este tome providências para garantir o progresso do sistema.

2.1.1 Trabalhos Relacionados

Pouca pesquisa tem sido realizada sobre o problema do posicionamento de réplicas. Alguns sistemas, como o Hector [RUS98] [RUS99], se preocupam com a migração de tarefas entre os nodos, mas não lidam com replicação.

Christian [CHR92] descreve um serviço chamado *Availability Manager* o qual tenta manter um nível de disponibilidade detectando a falha e ajustando o grau de replicação de acordo com isto. Entretanto, o *Availability Manager* se preocupa em manter um nível de replicação e não um nível de disponibilidade. Manter um nível constante de replicação não assegura um nível constante de disponibilidade.

O sistema MARS [KOP90] é um dos mais avançados em decisões de posicionamento, tentando posicionar objetos em nodos que possuem características de confiabilidade consistentes com os objetivos da aplicação. Entretanto, sua análise e posicionamento é somente executada em tempo de compilação e é fixa enquanto dura a execução da aplicação.

O RMS [LIT94] – *Replica Management System* - trata detalhadamente o problema de posicionamento, permitindo ao programador especificar a qualidade do serviço requerida em termos de disponibilidade e desempenho. A partir da especificação da qualidade do serviço, de informações sobre o protocolo de consistência de réplicas

utilizado e dados sobre as características do sistema distribuído, o RMS determina um posicionamento inicial e um nível de replicação para um objeto replicado. Quando é detectada uma falha ou recuperação, nos nodos ou no sistema de comunicação, o RMS pode ser re-invocado para computar uma atualização no mapeamento nas réplicas, preservando a qualidade de serviço desejada.

2.1.2 Aspectos Relevantes Para o Posicionamento de Réplicas

Os aspectos descritos a seguir são considerados relevantes para o posicionamento de réplicas pelo sistema RMS [LIT94]. Os três primeiros fazem referência a indicação de confiabilidade somente em componentes individuais, enquanto que os próximos correspondem a uma visão mais geral da rede. Estes aspectos podem servir de base à construção de outras estratégias de posicionamento.

Confiabilidade dos Componentes

O termo componente refere-se aos componentes de hardware e software do sistema distribuído, como por exemplo nodos e *links* de comunicação. É necessário considerar as interações entre os componentes do sistema distribuído para determinar o grau de confiabilidade de um componente específico.

Por exemplo, um nodo que raramente falha pode estar conectado a uma rede que perde mensagens freqüentemente, passando a visão de não confiável ao usuário.

Confiabilidade e Disponibilidade do Nodo

As duas medidas mais comuns são: MTTF (tempo médio até defeito) e MTTR (tempo médio para reparo). O ideal é que os nodos possuam um alto MTTF - falhem raramente - e um baixo MTTR - recuperem-se rápido. Para se obter estas medidas é necessário monitorar continuamente os nodos do sistema por um longo período de tempo.

Confiabilidade dos Links de Comunicação

Para monitorar a confiabilidade dos *links* deve ser determinada a probabilidade de perda de mensagens e de particionamento da rede. A perda de mensagens ocorre por problemas no meio de comunicação e, mais freqüentemente, pelo *overflow* do *buffer* do nodo receptor.

Dependência de Falhas

O objetivo dessa indicação é avaliar como uma falha pode levar a outras, para isso deve-se levar em consideração as dependências entre os componentes. Por exemplo: fontes de energia compartilhadas, sub-rede comum, dependência do mesmo disco da rede. As interdependências são propriedades estáticas da topologia da rede ou de nodos individuais, mas fatores dinâmicos podem também afetar a confiabilidade.

Interdependência dos Objetos

Sempre que um objeto invoca um método de outro objeto ele é dependente daquele objeto. O grau de dependência do objeto inclui a freqüência de invocação dos métodos e as propriedades dos objetos sendo invocados. As dependências entre objetos mudam mais rapidamente que as interdependências entre os nodos e podem variar de uma execução da aplicação para outra. Esta qualidade dinâmica implica que esse fator deve ser continuamente monitorado. O desempenho e a disponibilidade serão aumentados se posicionarmos juntos os objetos que são dependentes. Se os objetos podem migrar de um nodo para outro, então os dados associados pela dependência podem indicar outros objetos que também devem ser migrados. Esse fator pode indicar também que a migração não é possível (o objeto a ser migrado pode ser criticamente dependente de outro objeto que deve ser fixo).

Desempenho dos Nodos

O desempenho de um nodo é uma função complexa da carga do nodo, isto é, o número de processos ativos sendo executados nele e da sua configuração, como por exemplo, a velocidade do processador e da memória disponível. Deve-se considerar o sistema distribuído como um todo, ao invés de um único nodo, para poder proporcionar um melhor desempenho e disponibilidade. Por exemplo, posicionar um objeto em um nodo levemente carregado, porém conectado com o resto da rede por um *link* de comunicação lento, pode, se o objeto necessitar de muitas interações remotas, reduzir o desempenho do objeto .

Neste contexto as dependências entre os objetos também devem ser consideradas. Por exemplo, se um objeto faz uso excessivo dos recursos de outros objetos que devem residir em um nodo carregado, pode fazer mais sentido posicionar o objeto naquele nodo, desde que o *overhead* gerado com esta realocação seja inferior ao *overhead* proporcionado pela comunicação remota.

Os padrões de utilização de recursos dos objetos a serem posicionados e os recursos providos pelos nodos nos quais eles podem ser posicionados necessitam ser levados em conta. É necessário um mapeamento dinâmico da utilização dos recursos pelos objetos e a correspondente disponibilidade de recursos nos nodos que compõem o sistema distribuído.

Desempenho do Link de Comunicação

O desempenho do *link* é expresso em termos de quão rápido ele entrega uma mensagem e a sua largura de banda. O meio de comunicação deve ser levado em consideração de acordo com o tipo de aplicação que será executada. Por exemplo, para aplicações multimídia ou de tempo real, as tecnologias de FDDI e ATM oferecem vantagens significativas.

O RMS toma suas decisões de posicionamento baseado nas medidas de confiabilidade dos nodos – MTTF e MTTR. Estas medidas são obtidas após um longo período de observação e são divulgadas para todos os nodos do sistema em grandes intervalos de tempo – geralmente a cada 24 hs [LIT94]. Se a freqüência com que novos nodos são adicionados ao sistema for considerável este período de divulgação pode ser diminuído. Outro fator a ser ressaltado é que o RMS não considera a carga dos nodos para tomar decisões sobre o posicionamento. Dessa forma, um nodo confiável e

sobrecarregado pode ser considerado para receber uma réplica de um objeto, situação esta que pode degradar o desempenho do sistema como um todo.

2.2 Protocolos de Consistência de Réplicas

Em um sistema distribuído, quando um nodo da rede falha, os objetos que se encontram neste nodo se tornam indisponíveis, fazendo com que as operações de leitura e escrita sobre esses objetos não se completem com sucesso. Para prover tolerância a falhas, deve-se utilizar uma política de replicação dos objetos em vários nodos, de forma que se algum dos nodos falhar, essa falha não torne o item de dados indisponível para as operações dos usuários.

A replicação dos dados, apesar de prover resiliência[1] contra falhas, introduz o problema de gerenciar e manter a consistência entre as réplicas. Como o objetivo da replicação é prover tolerância contra falhas, a replicação não deve ser visível ao usuário. A estes deve ser mantida uma visão de cópia única. Uma ação irá executar operações nos itens de dados e o sistema irá mapear estas operações em múltiplas cópias dos itens de dados, de forma transparente ao usuário.

A correção de uma base de dados replicados está baseada na propriedade de linearizabilidade, também chamada de *equivalência de cópia única* [GUE97]. A linearizabilidade garante que os usuários terão a visão de que existe apenas uma cópia única do dado e a visão é equivalente para todos os usuários.

Para assegurar a linearizabilidade, as invocações dos clientes devem satisfazer as seguintes propriedades:

- *Atomicidade*: Uma ação será efetuada sobre todas as réplicas ou não será efetuada.
- *Ordem*: As ações sobre as réplicas serão efetuadas na mesma ordem.

Os métodos para gerenciar dados replicados são chamados de algoritmos de controle de réplicas, nos quais o critério de linearizabilidade é satisfeito. Existem dois tipos de falhas que devem ser manipuladas pelos algoritmos de controle de réplicas: falhas de nodo e falhas de comunicação.

Falhas de nodo fazem com que cópias dos dados naquele nodo se tornem inacessíveis. O resto da rede permanece conectado e mantém as cópias dos objetos de dados disponíveis. Os algoritmos de controle de réplica devem assegurar que mesmo se alguns dos nodos falham, tornando algumas cópias dos itens de dados indisponíveis, as operações devem ser realizadas satisfazendo o critério de linearizabilidade.

A segunda falha é a falha de comunicação levando ao particionamento da rede. Nesta, nodos e *links* falham, de modo que os nodos remanescentes são particionados em grupos. Nodos em uma partição ou grupo podem se comunicar uns com os outros, mas não podem se comunicar com os nodos nos outros grupos. O protocolo de controle de réplicas deve impor restrições no processamento nas diferentes partições de forma que a consistência mútua não seja violada quando estas partições voltarem a se comunicar.

Os protocolos de controle de réplica podem ser otimistas ou pessimistas [DGS85]. Na estratégia otimista, se um particionamento da rede ocorre, nenhuma restrição é feita no processamento pois assume-se que as operações executadas nas diferentes partições não irão conflitar. Por outro lado, nas estratégias pessimistas, o protocolo de replicação limita o acesso aos dados, para prevenir inconsistências.

---

[1] Um objeto é dito ser resiliente se as operações sobre esse objeto podem ser completadas, mesmo na ocorrência de falhas [JAL94].

2.2.1 Métodos Otimistas

Se um particionamento da rede ocorrer, estas estratégias não fazem nenhuma restrição ao processamento, na esperança de que as operações sendo executadas nas diferentes partições não irão conflitar [JAL94]. Sob este aspecto, a linearizabilidade em cada grupo será preservada, mas o estado global pode ser inconsistente por não satisfazer o critério de equivalência de cópia única. Dessa forma, se inconsistências globais surgirem, as estratégias otimistas tentarão resolvê-las após as diferentes partições se juntarem e estarem capazes de se comunicar novamente.

As estratégias otimistas diferem-se entre si na maneira como elas resolvem as inconsistências. Durante o particionamento da rede, se as operações são executadas independentemente em cada partição, as cópias de dados em cada partição podem ser inconsistentes. Quando a partição une-se novamente, tais inconsistências devem ser detectadas e resolvidas, se possível.

2.2.2 Métodos Pessimistas

Primário-Backup

Esta técnica usa uma réplica, a primária, que possui um papel especial: ela recebe as invocações dos processos clientes e retorna respostas [GUE97]. As outras réplicas são os *backups*. Os *backups* interagem diretamente somente com a réplica primária e não com os processos clientes. Se uma requisição é enviada a um *backup*, esta é repassada ao primário.

Se a operação solicitada é de leitura, então o primário executa a operação e retorna o resultado ao processo que a requisitou.

Se a operação é de escrita, antes de executá-la, o primário envia aos *backup*s uma requisição de atualização. Quando estes recebem a mensagem, atualizam o seu estado e retornam uma mensagem de *ack* ao primário. Quando a réplica primária receber os *acks* de todos os *backups* corretos, ela envia a resposta ao solicitante da operação.

Desde que todas as requisições chegam primeiro ao site primário, o qual envia as requisições de atualização para os *backups*, todos os *backups* obtém as requisições na mesma ordem que o primário, assegurando assim a propriedade de ordenação. Dessa forma, na execução dessas requisições, os dados nos *backups* estarão no mesmo estado que os dados no primário. A recepção pelo primário, do estado dos *backups* assegura a propriedade de atomicidade.

Durante todo esse processo podem ocorrer falhas nos nodos ou nos *links* de comunicação. No caso de ocorrência de falhas em nodos, se todos os nodos contendo as réplicas dos objetos e também do primário falharem, então o objeto é perdido e nada pode ser feito. Enquanto um deles permanecer em funcionamento (primário ou um dos *backups*), a disponibilidade pode ser garantida.

Supondo que o nodo falho é um nodo de *backup*, então o serviço não é descontinuado, pois o processo cliente recebe as respostas do primário.

Se o primário falha então um novo primário deve ser eleito. Há várias maneiras de eleger um novo primário [GAR82]. Uma maneira simples é, tendo uma lista ordenada com todos os nodos do sistema, escolher o próximo nó não falho como o novo primário. Se o primário grava cada operação nos *backups*, então o novo primário pode iniciar executando as requisições dos usuários após ele ter processado todas as operações repassadas a ele pelo antigo primário.

Assegurar a linearizabilidade em alguns casos de falhas do primário é mais difícil. Podemos distinguir três casos, nos quais a réplica primária falha [GUE97]:

1. Antes de enviar a mensagem de atualização para os *backups* (ponto A, na figura 2.1);
2. Após (ou enquanto) envia a mensagem de atualização mas antes do cliente receber a resposta (ponto B);
3. Após o cliente receber a resposta (ponto C).

Em todos os três casos uma nova réplica primária deve ser escolhida.

No terceiro caso, a falha é transparente para o cliente. No primeiro e segundo casos, o cliente não irá receber a resposta a suas invocações e irá suspeitar de uma falha. Após ter aprendido a identidade da nova réplica primária, o cliente irá refazer sua solicitação. No primeiro caso, a nova réplica primária considera a invocação como nova.

O segundo caso é mais difícil de manipular. A solução deve assegurar a atomicidade: ou todos ou nenhum dos *backups* deve receber a mensagem de atualização. Se nenhum dos *backups* recebe a mensagem, o segundo caso se torna similar ao primeiro caso. Se todos recebem a mensagem de atualização, a operação do processo cliente atualiza o estado dos *backups*, mas o cliente não recebe a resposta e irá reenviar sua solicitação. Neste caso, informações armazenadas nos nodos irão garantir que uma mesma requisição não será executada novamente, o novo primário ao receber a segunda requisição irá apenas retornar a resposta ao processo cliente.

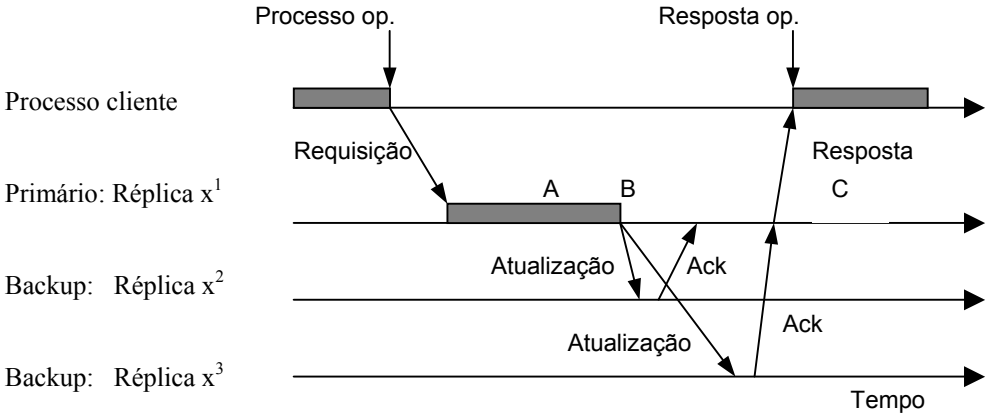

FIGURA 2.1 – Protocolo Primário-Backup

Assumindo um mecanismo perfeito de detecção de falhas, a técnica de replicação primário-backup é relativamente fácil de implementar.

Réplicas Ativas

Também chamada de **Máquina de Estados** [SCH93], esta técnica dá a todas as réplicas o mesmo papel sem o controle centralizado da técnica de primário-*backup*. Neste método, a invocação de um cliente – requisição - vai para todas as réplicas. Cada uma delas processa a invocação, atualiza o seu estado e retorna a resposta ao cliente. O cliente aguarda até receber a primeira ou a maioria das respostas idênticas [JAL94].

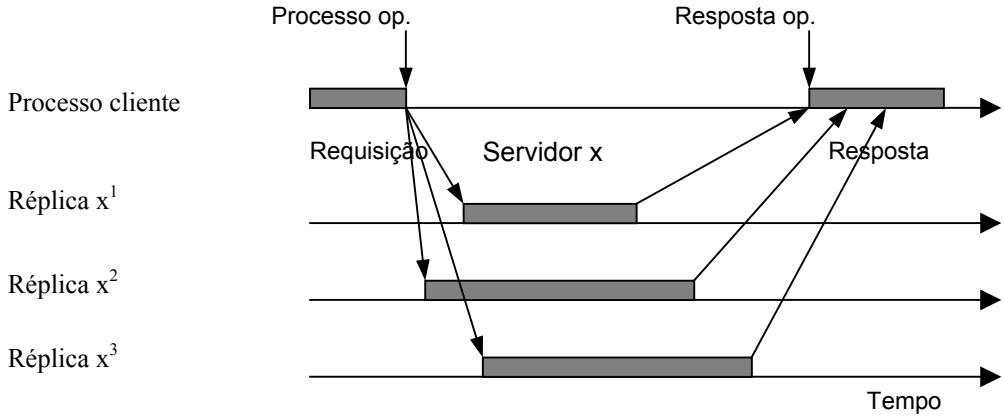

FIGURA 2.2 - Técnica de Replicação Ativa

Esta abordagem vê o sistema como consistindo de servidores e clientes. No caso de replicação de dados, os nodos que possuem as cópias dos dados serão os servidores, enquanto que os nodos que requisitam as operações nos dados serão os clientes. Os clientes e os servidores são nodos distintos. Esta abordagem pode suportar falhas Bizantinas bem como falhas *Fail-Stop* [SCH93]:

Falhas Bizantinas: o componente exibe comportamento arbitrário e malicioso;

Falhas Fail-Stop: em resposta a uma falha, o componente muda para um estado que permita aos outros componentes detectarem que uma falha ocorreu e então interrompe o seu funcionamento.

Quando o sistema apresenta falhas bizantinas, o cliente deve esperar pela maioria das respostas idênticas, para garantir que o resultado está correto. Por esse motivo, são necessárias $2k+1$ réplicas para suportar k falhas. Isto é devido ao fato que com $2k+1$ réplicas a maioria das saídas permanece correta mesmo após k falhas. Se o sistema apresenta somente falhas *fail-stop*, então os dados são replicados em apenas $k+1$ nodos, pois admite-se que as saídas sempre estarão corretas e o sistema necessita receber apenas uma resposta para a sua requisição.

A implementação de uma máquina de estados tolerante a falhas deve assegurar a coordenação das réplicas, que garante que todas as réplicas receberão e processarão a mesma seqüência de requisições. Se cada réplica está no mesmo estado inicial e processa o mesmo conjunto de requisitos, na mesma ordem, então cada uma irá produzir a mesma saída para uma operação. Deve-se assegurar que todas as réplicas possuam a mesma seqüência de requisições. Isto necessita que as propriedades de Acordo e Ordem sejam satisfeitas [SCH93]:

Acordo define que todas as réplicas que não estão em falha receberão todas as requisições;

Ordem define que todas as réplicas que não estão em falha processarão as requisições na mesma ordem.

Para falhas *fail-stop* a necessidade de acordo pode ser relaxada [SCH93]. Se uma requisição *r* (leitura) é tal que seu processamento não modifica o estado da máquina de estado, então a requisição pode ser enviada para somente uma réplica. Isto ocorre porque a resposta de uma máquina de estados não falha é garantida estar correta e é a mesma de outras máquinas de estado não falhas.

O requisito de ordem pode ser relaxado para requisições comutativas. Duas requisições *r* e *r'* são comutativas para uma máquina de estado se a saída produzida pela máquina de estados e o estado final da máquina de estados pelo processamento de *r*

antes de *r'* é o mesmo se tivesse sido processado *r*'antes de *r*. Se o resultado de executar duas requisições é o mesmo, apesar da ordem nos quais elas foram executadas, então a ordem na qual elas foram recebidas pelas diferentes réplicas não é importante.

A propriedade de *Acordo* pode ser satisfeita usando-se qualquer protocolo que permite a um processador, chamado transmissor, disseminar um valor para outros processadores de forma que:

- Todos os processadores não falhos concordam com o mesmo valor;
- Se o transmissor é não falho, então todos os processadores não falhos usam o seu valor como aquele com o qual eles concordam.

Os protocolos que seguem as regras acima são chamados de **protocolos de concordância bizantina** ou simplesmente de **protocolos de acordo**. Se as requisições são entregues a todas as réplicas da máquina de estados usando um protocolo que satisfaça as condições acima, então a propriedade de acordo é satisfeita.

A propriedade de *Ordem* pode ser satisfeita associando-se um identificador único às requisições e fazendo com que as réplicas da máquina de estados processem as requisições de acordo com a relação de ordem total entre os identificadores. Uma requisição à uma réplica da máquina de estado é considerada *estável* se nenhuma requisição chegar na máquina de estados com um identificador menor. Se cada réplica seguir esta regra para atender uma solicitação, então todas as réplicas irão atender as requisições na mesma ordem. Dessa forma, a *estabilidade* garante a propriedade de *Ordem*.

3 Comunicação de Grupos

A comunicação entre os processos é um assunto importante na construção de sistemas distribuídos. Os componentes devem se comunicar através da troca de mensagens. Uma RPC – chamada remota a procedimento - faz a comunicação entre um processo cliente e um processo servidor, mas muitas vezes um processo deve se comunicar com diversos processos e não com somente um, situação esta que a RPC não atende a contento. Por exemplo, um processo cliente fazendo uma solicitação para um grupo de servidores que provê um serviço de arquivos único, utilizando o protocolo de replicação ativa para prover um serviço tolerante a falhas, necessita de um meio de se comunicar com todas as réplicas e não com cada uma delas individualmente.

A abstração de grupos é um mecanismo alternativo de comunicação, no qual uma mensagem pode ser enviada a mais de um receptor em uma única operação. Ela provê as primitivas de comunicação necessárias para facilitar a implementação dos protocolos de primário-backup e de replicação ativa.

Para implementar o RPM foi utilizado o *Maestro Group Communications Tools* [VAY98]. Este provê um conjunto de ferramentas e interfaces que permite uma aplicação – o RPM – ser desenvolvida utilizando os recursos do sistema de comunicação de grupos Ensemble.

Este capítulo apresenta, na seção 3.1, os principais conceitos relacionados a comunicação de grupos. Na seção 3.2 são descritos trabalhos que utilizam a comunicação de grupos como suporte para a replicação e na seção 3.3 é descrito o sistema de comunicação de grupos Ensemble.

Para o leitor familiarizado com os conceitos de comunicação de grupos nenhuma nova contribuição é aqui acrescentada.

3.1 Conceitos de Comunicação de Grupos

Um grupo é um conjunto de processos que agem juntos, de maneira especificada pelo sistema ou por um usuário. A propriedade fundamental dos grupos é que quando uma mensagem é enviada para o grupo, todos os membros deste grupo devem recebê-la [TAN92].

Há fundamentalmente dois tipos de grupos, estáticos e dinâmicos [GUE97]. Os membros de um grupo estático não mudam durante a vida do sistema. Apesar de membros de grupos estáticos falharem, o grupo estático não muda o seu *membership* (conjunto de membros do grupo) para refletir a falha de um membro.

Em grupos dinâmicos, há mudanças em seus membros durante a vida do sistema. Quando um dos membros falha, o sistema remove o membro que falhou do grupo. Após voltar ao seu estado normal, o membro junta-se novamente com o grupo. É usada a noção de **visão** para modelar as mudanças dos membros dos grupos.

A forma de implementar a comunicação de grupos depende essencialmente do hardware. Em algumas redes é possível usar a técnica de ***multicast***. Nesta técnica é possível criar endereços especiais, que são vistos por várias máquinas. Quando uma mensagem é enviada para um destes endereços, todas as máquinas que tem acesso a este endereço recebem a mensagem. Para implementar a comunicação de grupos com esta técnica simplesmente se atribui a cada grupo um destes endereços especiais.

Nas redes que não possuem este esquema de endereçamento especial, a mensagem pode ser enviada por ***broadcast***. Nesta técnica, a mensagem é enviada para todas as

máquinas. Quando uma determinada máquina recebe uma mensagem, ela verifica se esta é destinada a ela, se não for, a mensagem é descartada. Esta técnica de *broadcast* também pode ser usada para comunicação de grupos, mas é menos eficiente que a *multicast* devido ao tempo necessário para processar este tipo de interrupção.

Se a rede não possuir *broadcast* nem *multicast* ela pode utilizar a técnica de ***unicast***. Nesta técnica é enviada uma mensagem para cada um dos membros do grupo, individualmente. Se tiver de ser enviada uma mensagem para os *n* membros de um grupo ela deverá ser enviada *n* vezes. Se o grupo for pequeno pode-se utilizar este esquema, apesar de ser menos eficiente.

3.1.1 Características dos Grupos

Grupos Fechados e Grupos Abertos

Alguns sistemas suportam o conceito de grupos fechados, onde um processo para enviar uma mensagem para o grupo deve ser membro do grupo. Processos que não pertençam ao grupo não podem enviar uma mensagem ao grupo como um todo, podendo, porém, enviar uma mensagem a um membro individual do grupo.

Por outro lado, alguns sistemas suportam o conceito de grupos abertos, onde qualquer processo do sistema pode enviar uma mensagem para qualquer grupo.

A escolha do tipo de grupo deve ser baseada na aplicação que será executada. Por exemplo, quando é necessário implementar servidores replicados é importante que os processos não membros do grupo (clientes) possam enviar suas requisições para o grupo, sendo assim necessário utilizar-se de grupos abertos.

Grupos Igualitários e Grupos Hierárquicos

Em alguns grupos todos os processos são iguais, nenhum deles é superior e todas as decisões são tomadas coletivamente. Em alguns outros existe algum tipo de hierarquia. Por exemplo, existe um processo coordenador, cuja missão é dirigir o trabalho dos demais processos [TAN92].

No protocolo de réplicas ativas, o conjunto de réplicas pode ser visto como um grupo igualitário, enquanto que o protocolo de primário-backup, pode ser visto como um grupo hierárquico, onde a cópia primária assume o papel de coordenadora do grupo.

A vantagem de um grupo igualitário é que não existe um ponto único de falhas. Se um dos membros do grupo falha, o grupo fica menor mas o processamento não é interrompido. Os outros membros continuam com o seu processamento normal. No caso de um grupo hierárquico, a falha do coordenador leva a falha inteira do grupo, gerando um atraso até que um novo coordenador seja eleito.

Atomicidade

As mensagens enviadas a um grupo devem chegar a todos os membros do grupo ou a nenhum deles. Isso é chamado de atomicidade ou ***broadcast atômico***. Quando um processo envia uma mensagem ao grupo ele não deve se preocupar com a indesejável situação de apenas parte do grupo receber a mensagem.

A implementação do *broadcast atômico* não é tão simples quanto possa parecer [TAN92]. A única maneira de estar absolutamente seguro de que todos os destinos

receberam a mensagem é solicitando que eles enviem de volta uma confirmação. Se considerarmos que as máquinas nunca vão falhar, este método vai funcionar.

No entanto, para garantir tolerância a falhas em sistemas distribuídos, é essencial que a atomicidade seja mantida, mesmo na presença de nodos ou *links* falhos.

Ordenação de Mensagens

A comunicação de grupos deve atender a duas propriedades: atomicidade e ordenação de mensagens. Para garantir a consistência do processamento, as requisições devem ser atendidas na ordem em que elas foram enviadas, ou seja, os receptores irão receber as mensagens na ordem em que elas foram expedidas.

Se um processo envia a mensagem A e um outro processo envia a mensagem B, então, em primeiro lugar deve ser entregue a mensagem A, e depois deve ser entregue a mensagem B, ambas para todos os membros do grupo. Chama-se a isto de **ordenação total**.

Grupos Sobrepostos

Um processo pode ser membro de diversos grupos ao mesmo tempo. Este fato pode levar a situações de inconsistência. Por exemplo, a situação mostrada na figura 3.1. Temos quatro processos e quatro mensagens. A ordem de recebimento das mensagens é dada pelos números. Os processos B e C recebem as mensagens de A e D em ordem diferente. O processo B primeiro recebe uma mensagem de A e depois uma de D. O processo C as recebe na ordem oposta.

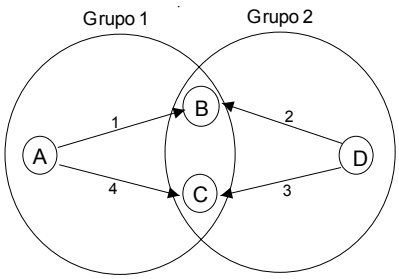

FIGURA 3.1 – Grupos Sobrepostos

O problema é que existe uma ordenação total dentro do grupo, mas não há uma coordenação entre os grupos. Esta situação pode levar a inconsistências indesejáveis. Alguns sistemas suportam uma ordenação muito bem definida entre grupos sobrepostos, mas outros sistemas não. A implementação de ordenação total entre grupos diferentes, apesar de importante, não é fácil de ser construída.

3.1.2 Controle dos Membros de um Grupo

Em grupos dinâmicos, o conjunto de processos que constituem o sistema em um dado momento pode variar. Novos processos são iniciados e se juntam ao sistema (*join*) enquanto que processos ativos deixam o sistema (*leave*) quando eles terminam, falham ou simplesmente escolhem se desconectar. Existem alguns aspectos importantes associados aos grupos que devem ser tratados quando de sua implementação [TAN92]:

- Identificação pelos membros do grupo quando um de seus componentes entra involuntariamente em estado de falha;

- Sincronismo das mensagens que estão sendo enviadas no instante de um membro ser inserido ou excluído do grupo.;
- Execução de um protocolo de reconstrução do grupo quando vários membros saem do ar, impedindo o grupo de funcionar.

O gerenciamento dos membros do grupo é uma das principais tarefas de um sistema de comunicação de grupos. Isso pode ser implementado através de um **Serviço de Membership de Grupo (GMS)** cujo papel é manter o *membership* de um sistema distribuído em benefício dos processos que o compõem. De forma simplificada, o GMS provê uma visão do grupo, composta dos membros alcançáveis do grupo num dado instante. Os processos que desejam se juntar ao grupo, primeiro contactam o GMS, que atualiza a lista de membros do sistema e então permite a requisição. Uma vez admitido no sistema, o processo pode interagir com os outros membros do sistema. Quando um processo termina, o serviço do GMS irá novamente atualizar a lista de membros do sistema. Naturalmente, as visões dos diversos membros do grupo devem ser atualizadas de uma maneira coerente, ou seja, os membros que as instalem (que desejam permanecer ou juntar-se ao grupo) devem concordar (*agree*) com sua composição. O GMS deve também lidar com falhas de partição e com a reunificação.

De forma geral, um GMS deve fornecer as seguintes propriedades [BIR96a]:

- Partindo de uma visão inicial, a cada adição (*join*) ou exclusão (voluntária ou por defeito ou suspeita de defeito), o GMS relata a nova visão ao grupo. Dependendo do protocolo adotado, o relato das mudanças ao processo da aplicação pode ocorrer imediatamente ou após a estabilização das mensagens difundidas na visão que se encerra;
- A visão do grupo não é alterada involuntariamente. Um processo é adicionado ao grupo somente se ele está ativo e tentando se juntar ao grupo (*join*), e é excluído somente devido a sua própria vontade, ou por estar falho, ou sob suspeita de falha por outro processo;
- Todos os membros do grupo observam uma seqüência contínua da mesma subseqüência de visões do grupo, iniciando com a visão em que o membro foi adicionado ao grupo e terminando com aquela em que ele foi excluído por saída voluntária, defeito ou suspeita de defeito;
- GMS não atrasa indefinidamente uma mudança de visão associada a um evento, isto é, se o serviço de GMS está ativo, um evento de *join*, saída voluntária, defeito ou suspeita, causa, num tempo finito, uma mudança de visão que corresponde ao evento;
- Ou o GMS permite o progresso somente num componente primário de uma rede particionada ou, se ele permite o progresso em componentes não primários, todas as visões do grupo são entregues com uma sinalização indicando sua associação, ou não, à partição primária da rede.

Estas propriedades não são suficientes para o funcionamento de um grupo dinâmico, pois elas não são suficientes para o suporte a aplicações distribuídas, devido a não referência a ordem das mensagens em relação às mudanças de visão. Isso permite que uma dada mensagem destinada a vários membros seja entregue à aplicação, por membros diferentes, em visões diferentes, causando inconsistências na aplicação. A percepção deste problema, bem como de problemas de reunificação de partições e desempenho, dentre outros, gerou a proposição do Sincronismo Virtual e do Sincronismo Virtual Estendido.

Sincronismo Virtual

Nos sistemas distribuídos que utilizam o modelo de comunicação de grupos para a troca de mensagens, a adesão de novos processos ao grupo (*join*) ou a exclusão de membros gera, dinamicamente, novas visões do grupo. Nessa situação, cada componente do grupo pode ter uma visão diferente da composição do grupo. Por esse motivo, o sincronismo entre os membros do grupo e a ordenação na entrega de mensagens por cada membro é importante para se garantir um estado consistente.

A sincronia virtual assegura que todos os processos no grupo recebam informações consistentes sobre os componentes do grupo, na forma de visões. Os componentes do grupo podem mudar ao longo do tempo pela união de novos processos ao grupo ou pela falha ou abandono voluntário de processos antigos do grupo. A sincronia virtual também ordena mensagens com mudanças de visão e garante que todos os processos que instalem duas visões consecutivas entregam o mesmo conjunto de mensagens entre estas visões. Isto é, se após uma falha, uma mensagem m_j é entregue aos seus destinatários, então uma mensagem m_i enviada antes da falha deve ser entregue pelo sistema, aos seus destinatários, antes dele entregar m_j.

O sincronismo virtual assegura que todos os processos pertencentes a um grupo de processos percebam as mudanças de configuração que ocorrem em um mesmo tempo lógico. Todos os processos que pertencem a uma configuração entregam o mesmo conjunto de mensagens para aquela configuração. Uma mensagem é garantida ser entregue, na mesma configuração em que ela foi difundida, a todos os processos [AMI95].

Ele pode ser definido formalmente da seguinte forma [ROD95]:

Considerando-se um conjunto de processos (grupo) g, uma visão $V^i(g)$ e uma mensagem m, difundida para os membros do grupo $V^i(g)$. Se existe um processo p que pertença a $V^i(g)$, o qual entregou m na visão $V^i(g)$ e já tenha instalado $V^{i+1}(g)$, então cada processo q que pertença a $V^i(g)$, o qual já instalou $V^{i+1}(g)$ deve entregar m antes de instalar $V^{i+1}(g)$. O sistema é virtualmente síncrono se e somente se cada *multicast* é um *multicast* virtualmente síncrono.

A implementação da sincronia virtual requer o uso de um detector de falhas e de um protocolo de esvaziamento (*flush*) para assegurar que todas as mensagens entregues a alguns membros em uma dada visão são entregues a todos os membros corretos naquela visão antes de uma nova visão ser instalada. Para garantir o término do protocolo de *flush*, o tráfego pode ser temporariamente suspenso durante a execução do protocolo. Existem protocolos que permitem a continuação do fluxo de mensagens durante as mudanças de visão.

O sincronismo virtual considera falhas de omissão de mensagens e falhas *fail-stop*, permitindo recuperar processos falhos, ou simplesmente excluídos por suspeita, como novos processos. Quando o particionamento da rede ocorre, ele assegura que os processos em pelo menos uma partição conectada da rede, a partição primária, são capazes de continuar operando, os processos nas outras partições são bloqueados.

Incapaz de lidar com partições de rede e reunificação, e com recuperação de processos, a sincronia virtual tem um valor prático limitado. Para superar estas limitações foi criada a sincronia virtual estendida.

Sincronismo Virtual Estendido

Este modelo, implementado pela primeira vez no sistema Totem [MOS96] é uma extensão do modelo de sincronismo virtual do ISIS [BIR93]. O sincronismo virtual suporta somente falhas *fail-stop* e de omissão de mensagens, enquanto que o sincronismo virtual estendido suporta falhas de *crash* e *recovery* e particionamentos da rede e reunificações.

O significado do sincronismo virtual estendido é que, durante o particionamento e reunificação e durante o processo de *crash* e *recovery*, ele mantém um relacionamento consistente entre a entrega de mensagens e a entrega de mensagens de mudança de configuração entre todos os processos do sistema [MOS94].

No sincronismo virtual estendido, uma configuração é composta pelo *membership* atual e um identificador único. O algoritmo de *membership* assegura que todos os processos em uma configuração concordam no *membership* daquela configuração. A aplicação é informada sobre mudanças na configuração pela entrega de mensagens de mudança de configuração.

É feita uma distinção entre a **recepção** de uma mensagem, que pode estar fora de ordem e a **entrega** da mensagem para a aplicação, que pode ser atrasada até que mensagens prioritárias sejam entregues. Mensagens podem ser entregues em ordem de acordo *(agreed order)* ou em ordem segura *(safe order)*. A entrega em ordem de acordo garante a entrega totalmente ordenada de mensagens dentro de cada partição e permite a uma mensagem ser entregue logo após que as as mensagens predecessoras tenham sido entregues. Uma entrega segura *(safe delivery)* requer, em adição, que se uma mensagem é entregue para qualquer processo dentro de uma configuração, essa mensagem será então recebida e será entregue para cada um dos processos na configuração a menos que o processo falhe.

Para obter entrega segura na presença de particionamentos da rede e reunificação, e *crash* de processos e *recovery*, o sincronismo virtual estendido apresenta dois tipos de configuração: **regular** e de **transição**. Em uma configuração regular novas mensagens são enviadas e entregues. Em uma configuração de transição, nenhuma mensagem nova é enviada, mas as mensagens remanescentes da configuração anterior são entregues. Essas mensagens não satisfazem os requisitos de entrega segura ou causal na configuração regular e, consequentemente não podem ser entregues naquela configuração.

Uma configuração regular pode ser seguida por diversas configurações de transição (uma para cada componente da rede particionada) e pode ser precedida por diversas configurações de transição (quando as partições se reúnem). Uma configuração de transição é precedida e seguida por uma única configuração regular [MOS94].

Cada processo em uma configuração regular ou de transição entrega uma **mensagem de mudança de configuração** para a aplicação terminar a configuração anterior e iniciar a nova configuração. A entrega de uma mensagem de configuração que inicia uma nova configuração segue a entrega de cada mensagem na configuração que ela termina e precede a entrega de cada mensagem na configuração que inicia. A mensagem de mudança de configuração, que inicia uma configuração de transição, define o *membership* dentro do qual é possível garantir a entrega segura das mensagens restantes da configuração regular anterior.

Os sistemas de comunicação de grupo Totem [MOS96], Transis [DOL96] e Horus [BIR96b], dentre outros, utilizam o sincronismo virtual estendido.

3.2 Replicação Através de Comunicação de Grupos

A comunicação de grupos permite construir sistemas distribuídos com as seguintes características [COL94]:

- Tolerância a falhas baseada em serviços replicados: As requisições dos clientes são enviadas (*multicast*) para todos os membros do grupo, cada um dos quais executa uma operação idêntica. Mesmo que alguns dos membros falhem, os clientes podem continuar a serem servidos.

- Localização de objetos em serviços distribuídos: Mensagens de *multicast* podem ser usadas para localizar objetos dentro de um serviço distribuído, como os arquivos dentro de um serviço de arquivos distribuído.

- Melhor desempenho através dos dados replicados: dados são replicados para aumentar o desempenho de um serviço. Cada vez que os dados mudam, o novo valor é difundido por *multicast* para os processos que gerenciam as réplicas.

- Múltiplas atualizações: O *multicast* para um grupo pode ser usado para notificar processos quando algum evento acontece, por exemplo, um sistema de notícias pode notificar usuários interessados quando uma nova mensagem foi postada em um *news group* em particular.

O Isis [BIR93] foi o primeiro sistema a introduzir primitivas de comunicação de grupo para suportar aplicações distribuídas confiáveis [GUE97]. Inicialmente desenvolvido na Universidade de Cornell como um projeto acadêmico, o Isis tornou-se, mais tarde, um produto comercial.

Pesquisadores desenvolveram diversos outros sistemas. Entre eles o Horus (Cornell University) [BIR96b], o Transis (Hebrew University, Jerusalém) [DOL96], o Totem (University of California, Santa Bárbara) [MOS96], o Amoeba (Free University, Amsterdam) [TAN92] e outros. Estes sistemas diferem nas primitivas de *multicast* que eles provêem.

A abstração de grupos, como mostrado na figura 3.2, é um modelo adequado para prover primitivas de *multicast* necessárias para implementar as técnicas de replicação ativa e de primário-backup [GUE97]. Com a abstração de grupos não é necessário enviar uma mensagem a cada uma das réplicas individualmente. Os servidores replicados formam grupos e quando é solicitada uma operação, a mensagem com a requisição é enviada ao grupo, que a processa, atualiza seu estado e envia a resposta ao processo cliente.

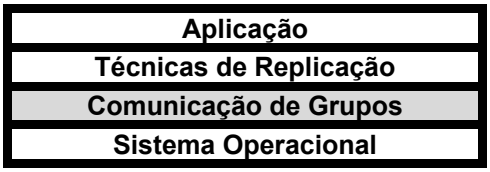

FIGURA 3.2 – Modelo Para Implementação de Replicação com Comunicação de Grupo

Para a implementação da técnica de réplicas ativas, é exigida uma primitiva de *multicast* que garanta as propriedades de ordem e atomicidade. A técnica de réplicas ativas não necessita de nenhuma ação específica quando uma réplica falha, pois a falha de uma réplica é transparente ao processo cliente. Dessa forma, esta técnica pode utilizar grupos estáticos, porém, ela é mais comumente implementada utilizando-se grupos dinâmicos [GUE97].

Para a implementação da técnica primário-backup, não é necessária uma primitiva de ordem, já que esta é naturalmente definida pela réplica primária. Esta técnica exige

para a sua implementação que o *membership* do grupo possa mudar. Se uma réplica primária falha, o grupo deve eleger uma nova primária, sendo assim, um grupo dinâmico é exigido.

Guerraoui [GUE96][GUE97] define duas primitivas que garantem as necessárias propriedades: **TOCAST** para a implementação de um *multicast* com ordenação total, necessária para a implementação da técnica de réplicas ativas e **VSCAST** para *multicast* de visão síncrona, necessária para a implementação da técnica de primário-backup.

Um outro trabalho que trata de replicação de bases de dados sobre um sistema de comunicação de grupos é apresentado por Amir, na Hebrew University of Jerusalem [AMI95]. A proposta é construir uma arquitetura e algoritmos para implementar a replicação ativa sobre uma rede particionável.

O sistema apresentado está sujeito a falhas de omissão de mensagens, falhas e posterior reunião de servidores e particionamento e reunião de partições. Como camada de comunicação de grupos foi utilizado o sistema Transis [DOL96] que implementa o sincronismo virtual estendido, necessário para o serviço.

A arquitetura do sistema é apresentada na figura 3.3. Ela é estruturada em duas camadas: um servidor de replicação e uma camada de comunicação de grupos. Cada um dos servidores de replicação mantém uma cópia da base de dados. Ações (consultas e atualizações) solicitadas pela aplicação são ordenadas globalmente, de maneira simétrica, pelos servidores de replicação. As ações ordenadas são aplicadas na base de dados e resulta em uma mudança de estado e uma resposta à aplicação.

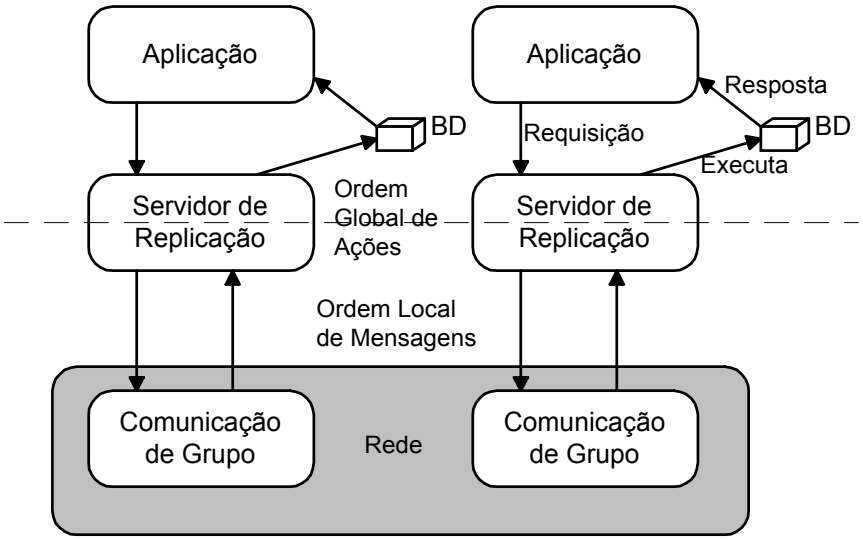

FIGURA 3.3 – Exemplo de Implementação Usando Comunicação de Grupo

Os servidores de replicação usam a camada de comunicação de grupos para disseminar as ações de maneira eficiente e tratar as mudanças de *membership*. A camada de comunicação de grupos ordena localmente as mensagens disseminadas dentro do grupo atual.

O autor chama de componentes as diversas partições da rede (quando elas ocorrem). Quando um novo componente é formado pela reunião de dois ou mais componentes, os servidores trocam informações sobre as ações e sua ordem no sistema. Ações que faltam em alguns servidores são difundidas por *multicast* para que todos os servidores atinjam um estado comum. Desta maneira, as ações são propagadas tão logo seja possível. Este método é chamado de **propagação por caminho eventual** [AMI95].

Como o sistema pode particionar, deve ser assegurado que dois componentes não tomem decisões contraditórias com relação à ordem global das ações. Um componente deve ser identificado como componente primário, que deve continuar ordenando as ações. Um método de votação dinâmica linear é utilizado para a escolha do componente primário.

A arquitetura é não bloqueante, isto é, as ações podem ser geradas pela aplicação a qualquer tempo. No componente primário, as requisições são atendidas imediatamente, de maneira consistente. No componente não-primário, o usuário pode escolher esperar por uma resposta consistente (que chegará tão logo a rede tenha sido reparada) ou obter uma resposta imediata, que poderá não estar consistente.

O uso de comunicação de grupo para a implementação de replicação no caso descrito melhorou o desempenho. Foi relatado que quando o *membership* dos servidores é estável, o *throughput* e a latência de ações é determinada pela performance do serviço de comunicação de grupo e não mais por outros fatores como número de réplicas e desempenho de escritas em disco [AMI95].

3.3 Sistema Ensemble

O Grupo de Tolerância a Falhas do PPGC da UFRGS vem desenvolvendo trabalhos de pesquisa utilizando o sistema de comunicação de grupos Ensemble. Este sistema é uma nova versão do sistema Horus, desenvolvido na Universidade de Cornell. Ele foi projetado para garantir confiabilidade, alta disponibilidade, tolerância a falhas, consistência e segurança em aplicações distribuídas que utilizam comunicação de grupos.

O Ensemble apresenta uma arquitetura modular onde conjuntos de micro-protocolos se comunicam e compõem um protocolo de mais alto nível. Isso permite as aplicações, em tempo de execução, selecionarem o conjunto de protocolos que necessitam.

Uma de suas vantagens é a sua portabilidade e o grande número de interfaces de programação disponíveis. O sistema Ensemble foi projetado para ser executado em plataformas UNIX, Windows 95 e Windows NT e suporta interfaces para aplicações implementadas em C, C++, Java, Ada, Tcl/Tk, Smalltalk, CORBA e ML. Outra vantagem é a distribuição e o suporte gratuitos.

3.3.1 Arquitetura do Ensemble

Uma descrição detalhada do sistema Ensemble pode ser encontrada na dissertação de doutorado de Mark Hayden [HAY98]. O sistema é composto pelos seguintes componentes [HAY98]:

Rede

Serve como meio para transmitir mensagens entre os processos. As redes não provêem temporizações ou garantias de confiabilidade, como detecção de falhas dos *links* da rede. Uma das características do Ensemble é prover confiabilidade sobre redes não confiáveis.

Processos

Um processo é uma unidade de estado e computação provida pelo sistema operacional para a execução de programas [HAY98]. Os processos contém um espaço de endereçamento e uma ou mais *threads* de controle. Na descrição do Ensemble, um processo contém ainda um ou mais endereços de rede através dos quais ele troca informações com os outros processos.

Endpoints

Os *endpoints* são abstrações usadas para estruturar a comunicação num processo. Ele modela a entidade de comunicação. Dependendo da aplicação, ele pode corresponder a uma máquina, processo, *thread*, *socket*, etc. Os identificadores de *endpoint* não contém informações de endereçamento, razão pela qual os *endpoints* não estão restritos a um processo em particular e podem migrar para outros processos. No Ensemble, após um *endpoint* migrar, deve ser usado um endereço diferente para enviar mensagens a ele. Se a aplicação permitir migração de *endpoints*, os protocolos necessários a isso devem ser alocados na pilha de protocolos.

Grupos

Os grupos correspondem a alguns recursos computacionais que são distribuídos através de alguns *endpoint*s, cada um dos quais coordenado com os outros, para prover um serviço. O serviço pode ser fornecido por vários grupos, cada um dos quais assume um subconjunto do serviço como um todo.

No Ensemble, não existe uma estrutura de dados chamada “grupo”. Os grupos são representados por identificadores, que servem como um mecanismo de nomeação a ser usado pelos *endpoint*s quando eles se comunicam. As mensagens são divulgadas ao grupo pelo envio de uma mensagem ao endereço do processo que contém *endpoint*s em um grupo.

Pensando em um grupo como uma coleção de *endpoints*, existem várias operações nos *endpoints* que podem afetar o grupo. Podem ocorrer a divisão ou a união de *endpoints* em partições, envio de mensagens ponto a ponto entre *endpoints* da mesma partição ou difusão para todos os *endpoints* da partição.

Mensagens

Mensagens são as estruturas de dados mais importantes do Ensemble. A arquitetura e o desempenho do sistema são fortemente influenciados pela implementação das mensagens.

No Ensemble, as mensagens são divididas em duas partes: conteúdo e cabeçalho. A medida que a mensagem é passada para baixo, na pilha de protocolos do emissor, cada camada acrescenta um cabeçalho a mensagem. No destino, a medida em que a mensagem é passada para cima na pilha de protocolos, cada camada remove o seu cabeçalho correspondente (fig. 3.4). Como todas as pilhas numa partição usam a mesma ordem de camadas, cada camada acessa o cabeçalho gerado pelo mesmo protocolo. Uma camada não pode acessar os cabeçalhos de outras camadas. Isso significa que não há interdependências entre diferentes protocolos na estrutura de seus cabeçalhos, de forma que o cabeçalho de um protocolo pode ser alterado sem afetar outros protocolos.

No Ensemble, o formato dos cabeçalhos não é fixo, ou seja, não constitui um formato padrão, dessa forma cada protocolo pode representar o seu cabeçalho com

formato diferente dos outros. Isto dá ao Ensemble uma grande flexibilidade e a capacidade de mudar os protocolos da pilha em tempo de execução.

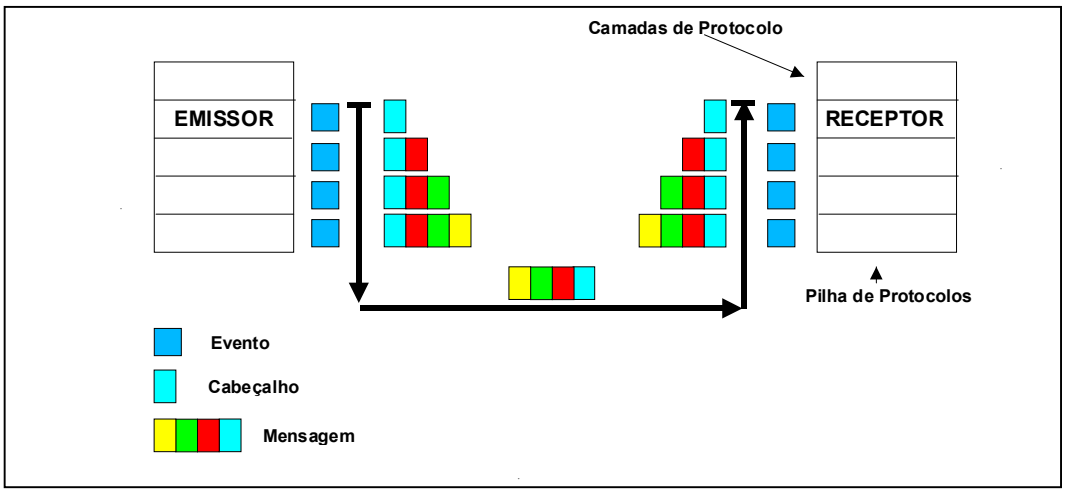

FIGURA 3.4 – Diagrama do Uso de Cabeçalhos e Eventos

Eventos

Eventos são estruturas de dados para comunicação intra-processos, enquanto que mensagens são usadas para comunicação entre-processos, ou seja, mensagens são usadas para a comunicação entre *endpoint*s e os eventos são usados para a comunicação dentro da pilha de procolos de um *endpoint*.

Um evento é um registro com um número de campos opcionais. O único campo que todos os eventos devem conter é o campo "*event type*". A tabela 3.1 lista alguns tipos de eventos.

TABELA 3.1 – Tipos de Eventos do Ensemble

Tipo de Evento	Descrição
Account	Informações de contabilização
Ack	Mensagem de recebimento
Block	Bloqueio do grupo
BlockOk	Recebimento de bloqueio do grupo
Cast	Mensagem de *broadcast*
Dump	Esvazia seu estado (para depuração)
Elect	Informa o novo coordenador
Exit	Desabilita uma pilha
Fail	Alguns *endpoints* falharam
GossipExt	Mensagem *gossip*
Init	Primeiro evento entregue
Invalid	Evento errôneo
Leave	Um *endpoint* quer sair
LostMessage	Uma mensagem foi perdida
MergeDenied	Requisição de união negada
MergeFailed	Falha de uma requisição de união
MergeGranted	Permite uma requisição de união
MergeRequest	Requisição de união

Migrate	Mudança de localização
Orphan	Mensagem órfã
Present	Descreve os *endpoints* presentes nesta visão
Prompt	*Prompt* para uma nova visão
Protocol	Requisita um novo protocolo
Rekey	Requisita um re-chaveamento (*rekeying*)
Send	Mensagem ponto a ponto
Suspect	Suspeita da falha em um *endpoint*
Timer	Requisita uma temporização
View	Notifica que uma nova visão está pronta
XferDone	Notifica que uma transferência de estado está completa

Viewstate

Os registros *Viewstate* são usados para configurar a pilha de protocolos. Eles contém a informação necessária para inicializar *endpoint*s membros de uma partição, incluindo informações como o nome do grupo, a lista de *membership*, endereços dos processos para cada *endpoint*, a descrição dos protocolos a usar, parâmetros opcionais, etc. Alguns destes campos são listados na tabela 3.2.

Os registros *Viewstate* são replicados para que todos os *endpoint*s numa partição usem registros equivalentes. Os protocolos estabelecem novas configurações do sistema pela disseminação de novos registros de *Viewstate*. Uma vez criado, um registro de *Viewstate* não é mais alterado e novas configurações são obtidas pela criação de cópias modificadas dos registros anteriores.

TABELA 3.2 – Campos de um Registro *Viewstate*

Nome do Campo	Descrição
Version	Versão do Ensemble
Group	Nome do grupo
Protocol	Pilha de protocolos em uso
Params	Parâmetros do protocolo
Coordinator	Coordenador inicial
Logical-time	Tempo lógico desta visão
View	Membros na visão
Address	Endereços dos membros
Out_of_date	Quem está desatualizado
Clients	Quem são os clientes do grupo?
Primary	É esta a partição primária?
Xfer_view	Esta é uma visão de transferência de estado?
Key	Chaves de segurança em uso
Prev_ids	Identificadores das visões prévias
Uptime	Tempo que este grupo foi inicializado

Camadas

No Ensemble os protocolos de alto nível são compostos de camadas de micro-protocolos (geralmente mais do que 15 camadas). As pilhas (e as camadas nelas) são geradas para cada grupo que um *endpoint* passa a pertencer e novas pilhas são criadas cada vez que o grupo se reconfigura. As camadas podem ser combinadas de várias formas, provendo um grande número de propriedades que as aplicações podem selecionar. Algumas das propriedades que o Ensemble pode selecionar aparecem na tabela 3.3. As propriedades *default* do Ensemble são {Gmp, Sync, Heal, Migrate, Switch, Frag, Suspect, Flow}. O uso destas propriedades gera a pilha de protocolos mostrada na tabela 3.4.

TABELA 3.3 – Propriedades Suportadas Pelo Ensemble

Propriedade	Descrição
Agree	Entrega em acordo (segura)
Auth	Autenticação
Causal	*Broadcast* ordenado casualmente
Cltsvr	Gerenciamento de cliente-servidor
Debug	Adiciona camadas para depuração
Evs	Sincronismo virtual estendido
Flow	Controle de fluxo
Frag	Fragmentação-Reunião
Gmp	Propriedades do *membership* do grupo
Heal	Reunificação de partições
Migrate	Migração de processos
Privacy	Criptografia dos dados da aplicação
Rekey	Suporte para re-chaveamento do grupo
Scale	Escalabilidade
Suspect	Detecção de falhas
Switch	Mudança de protocolo
Sync	Sincronização das visões
Total	*Broadcast* totalmente ordenado
Xfer	Transferência de estados

Cada camada Ensemble possui três partes:

- Um tipo de dados para seu estado local e uma função para gerar um estado inicial baseado num *Viewstate*;
- Um tipo de dado para armazenar os cabeçalhos colocados nas mensagens;
- Gerenciadores para a comunicação entre as camadas acima e abaixo na pilha.

Cada instância de uma camada mantém algum estado local. Diferentes protocolos usam diferentes tipos para seus registros de estado. As camadas são independentes umas das outras, de forma que nenhuma parte do sistema pode acessá-la ou modificá-la.

As camadas interagem somente com as camadas acima e abaixo delas, através de eventos de comunicação. As restrições nas interações entre as camadas são importantes para o projeto, implementação, otimização e verificação. A garantia de que uma camada não possa acessar outros estados de camadas ou cabeçalhos significa que não há

dependências entre camadas destes tipos de dados, de forma que o estado de uma camada ou cabeçalho pode ser alterado sem afetar as outras camadas. Fazendo-se as camadas comunicarem-se com o seu ambiente somente através de eventos de comunicação garante-se que o comportamento de uma pilha de camadas é completamente descrito através deste evento de comunicação e das atualizações dos estados das camadas individuais.

TABELA 3.4 – Pilha de Protocolos Criada Usando-se as Propriedades *Default* do Ensemble

Protocolo	Descrição
Top	Camada de protocolo mais superior
Heal	Reunião de partições
Switch	Troca e arbitragem de protocolos
Migrate	Migração de processos
Leave	Saída confável
Inter	Mudanças de visão em múltiplas partições
Intra	Mudanças de visão em uma única partição
Elect	Eleição do líder
Merge	Protocolo de junção confiável
Slander	Compartilhamento de suspeita de falhas
Sync	Sincronização de mudança de visão
Suspect	Detecção de falhas
Stable	Detecção de broadcast estável
Appl	Aplicação
Frag	Fragmentação-Reunião
Pt2ptw	Controle de fluxo ponto a ponto
Mflow	Controle de fluxo *multicast*
Pt2pt	Ponto a ponto confiável (fifo)
Mnak	Protocolo de nak *multicast*
Bottom	Camada de protocolo mais inferior

Pilhas

No Ensemble, pilhas de protocolos são composições lineares de camadas que trabalham juntas para implementar protocolos de alto nível. Como todas as camadas de protocolos implementam a mesma interface, todas as combinações de camadas são sintaticamente corretas, mas nem todas as combinações de camadas formam pilhas de protocolos úteis. O Ensemble provê um mecanismo para selecionar pilhas de protocolos que implementam um conjunto específico de propriedades.

O modelo de camadas apresenta duas características. A primeira é que eventos são passados entre as camadas numa ordem FIFO. A segunda é que cada camada executa um único evento a cada instante de tempo (execução serializada).

Eventos saindo da camada mais baixa da pilha causam uma mensagem ser transmitida pela rede. Os únicos eventos que surgem no topo da pilha são:

O evento *Newview* que é gerado quando a pilha de protocolos determina que está pronta para iniciar uma nova visão do grupo.

O evento *Exit*, que é gerado quando a pilha não tem mais nada para fazer.

Uma nova pilha de protocolos é gerada sempre que a configuração do grupo muda. Por exemplo, quando um *endpoint* falha ou quando duas partições se juntam, uma nova pilha é criada para aquela nova configuração em cada *endpoint*.

Aplicação

A maioria das arquiteturas de camadas colocam a aplicação no topo da pilha, porém no Ensemble a aplicação é considerada como parte de uma das camadas. Uma vantagem desta abordagem é que a aplicação não precisa estar no topo da pilha de protocolos, dessa forma eliminando o *overhead* das camadas acima dela quando ela envia mensagens. As camadas do protocolo de *membership* são colocadas acima da camada *Appl*. A camada *Appl* serve como um servidor para a aplicação na pilha, provendo interfaces para o envio e recebimento de mensagens.

3.3.2 Interação entre Componentes

Comunicação com a Rede

Ocorre quando uma mensagem surge na camada mais baixa da pilha de protocolos. A mensagem é constituída pela concatenação de um identificador de conexão, dos cabeçalhos e do conteúdo da mensagem. Um identificador de conexão especifica um destino particular (uma ou mais pilhas de protocolos dentro de um processo) de uma mensagem. No destino, eles são separados e o identificador de conexão é usado para identificar a pilha de protocolos para o qual devem ser entregues os cabeçalhos e o conteúdo.

Timeouts

As camadas requisitam os *timeouts* através de eventos de comunicação. Quando uma camada necessita que um *timeout* ocorra no futuro, ela gera um evento **Timer** com um campo **Alarm**, que especifica o tempo depois do qual ele deve ser disparado. Isso pode ser usado, por exemplo, para disparar a retransmissão de uma mensagem. Este evento é passado para baixo na pilha até atingir a camada mais baixa. Neste ponto, o valor do *timeout* é inserido numa fila de prioridades com outros *timeouts*. Quando o *timeout* expirou, um evento **Timer** é gerado com o tempo corrente no campo **Time**. Este evento é passado para cima na pilha de protocolos. Cada camada que está esperando por um *timeout* verifica o tempo e caso seu *timeout* tenha expirado, ele realiza a ação desejada e o evento é passado para cima na pilha.

Envio e Recebimento de Mensagens

Quando uma mensagem deve ser enviada, as seguintes ações ocorrem. A aplicação aloca um *buffer* e coloca a informação da mensagem no *buffer*. Ela então gera uma ação **Send** com o conteúdo da mensagem e o destino. Isto é passado para a camada *Appl* da pilha atual. A camada *Appl* gera um evento **Send** (diferente da ação Send) com o campo **Peer** setado para o destino. Esta camada (*Appl*) gera um cabeçalho **Send**. O evento, o conteúdo e o cabeçalho são passados para baixo na pilha.

Na seqüência, cada camada adiciona o seu cabeçalho correspondente e passa para baixo na pilha. Na camada *Pt2pt*, que implementa a ordenação das mensagens, mais trabalho é realizado. Na camada *Pt2pt*, o conteúdo e os cabeçalhos são colocados em um *buffer*, caso uma retransmissão seja necessária. Ela também coloca um cabeçalho

Data(seqno), onde **seqno** é o número de seqüência da mensagem atual. Cabeçalhos e eventos são passados para a camada abaixo. Na camada mais baixa eles são transformados em seqüências de bytes e transmitidos pela rede.

Quando a mensagem chega no destino, os cabeçalhos, conteúdo e eventos são separados e são entregues à camada mais baixa na pilha de protocolos do destino. Cada camada retira o seu cabeçalho e passa a informação restante para a camada acima. Na camada *Pt2pt*, o cabeçalho **Data(seqno)** é retirado e o número de seqüência é verificado. Se o número de seqüência é maior que o esperado, a informação é colocada em um *buffer* e uma mensagem com um cabeçalho **Nak(lo,hi)** é enviado de volta para a origem. Isto requisita a retransmissão das mensagens anteriores. Se o número de seqüência é o esperado a mensagem é passada para as camadas acima na pilha.

Para a camada *Appl* restarão o evento, o conteúdo e o cabeçalho Send. A camada *Appl* passa o conteúdo para o gerenciador **receive** da aplicação junto com a origem da mensagem.

Criação da Pilha

Uma nova pilha de protocolos é gerada sempre que a configuração da partição muda. Isto pode ser causado, entre outras causas, por uma falha de um processo, a junção de duas partições, a migração de um *endpoint* para outro processo, etc. Independentemente da causa, os *endpoint*s em uma partição executam um protocolo de reconfiguração.

Quando o protocolo de reconfiguração é completado, cada pilha na partição emite um evento **View** para o topo. Ele contém o registro de ***Viewstate*** para a nova partição.

Quando o evento **NewView** é gerado, o sistema usa o ***Viewstate*** para selecionar as camadas de protocolo apropriadas, cria um novo estado local para cada camada e as compõe. A pilha é então conectada a rede pela instalação de identificadores de conexão que a pilha recebe de uma tabela *hash* central. Quando a inicialização externa está completa, um evento **Init** é passado para a camada mais baixa da pilha para completar a inicialização. Este evento é então passado para cima na pilha e cada camada executa uma ação, como a requisição do primeiro *timeout*.

A pilha de protocolos antiga pode ficar ativa e operar em paralelo com a nova pilha de protocolos. Isso é necessário se algumas mensagens da pilha antiga necessitarem serem retransmitidas. No entanto, duas pilhas não podem se comunicar porque contém identificadores de conexão diferentes em suas mensagens.

Quando a pilha antiga não for mais necessária, um evento **Exit** é emitido no topo. Isto faz com que a conexão com a rede seja encerrada e finalizado o estado de todas as camadas.

3.3.3 Maestro Group Communication Tools

Maestro Group Communication Tools [VAY98] provê um conjunto de ferramentas e interfaces que possibilitam a um desenvolvedor de sistemas distribuídos programar diretamente com abstrações de comunicação de grupos, usando como suporte o sistema de comunicação Ensemble. O ponto central da ferramenta é uma hierarquia de classes que implementa tipos abstratos de dados fundamentais para grupos de objetos com protocolos de transferência de estado integrados e interfaces grupo-membro. A partir desta hierarquia de classes, sub-classes podem ser implementadas ou redefinidas com suas próprias políticas de protocolos, de acordo com as necessidades da aplicação.

Cada classe exporta uma coleção de *métodos downcall* públicos e *métodos callback* protegidos. Os *callbacks* não podem ser acessados diretamente, mas são automaticamente invocados quando eventos correspondentes ocorrem. Usando a implementação *default*, nenhuma ação é realizada quando um *callback* é invocado. Entretanto, o programador pode implementar uma ação específica definindo uma subclasse de uma classe grupo-objeto do Maestro e fazer a sobrecarga dos métodos *callbacks* apropriados.

A classe `Maestro_GroupMember` implementa um tipo abstrato de dados para grupos do Ensemble (fig. 3.5). Assim, uma aplicação define uma subclasse da classe `Maestro_GroupMember` que sobrecarrega (redefine) métodos da classe para implementar a aplicação desejada.

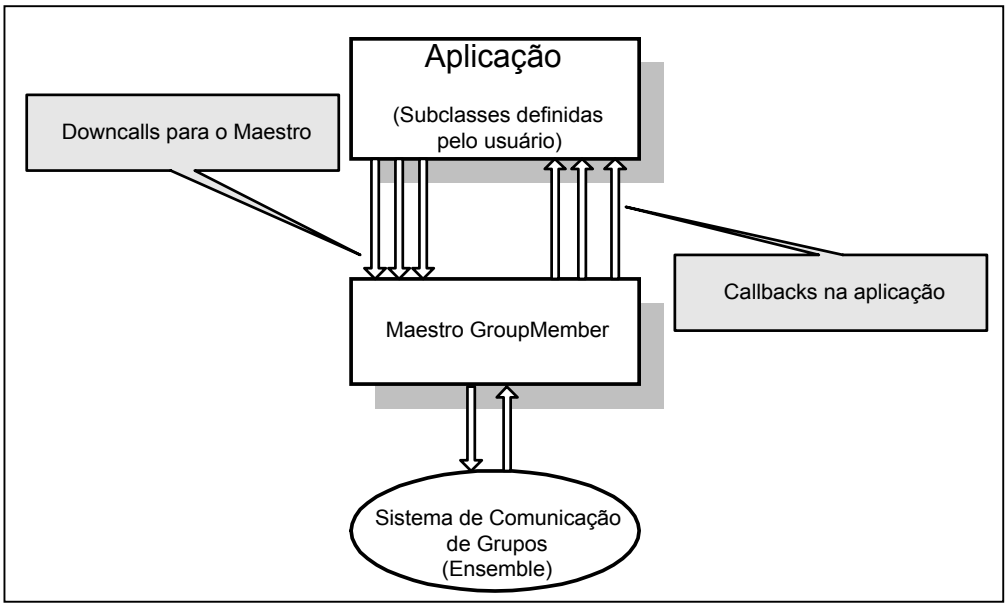

FIGURA 3.5 – Construção de Aplicações Pela Sobrecarga da Classe *Maestro_GroupMember* [VAY98].

No construtor da classe `Maestro_GroupMember` são colocadas as opções de inicialização usadas para especificar as propriedades dos objetos membros do grupo, tais como o nome do grupo de objetos a se juntar, protocolos de transporte suportados pela máquina onde o objeto está rodando, propriedades dos protocolos de grupo (ordenação, *membership*, etc), e outros. Estas opções de inicialização determinam a configuração para o Ensemble.

O Maestro provê dois métodos *downcall*, `join()` e `leave()` que são invocados por objetos para se juntar ou deixar um grupo. Um objeto `Maestro_GroupMember` pode pertencer a apenas um grupo, aquele cujo nome foi especificado na sua criação. Quando uma mudança no *membership* ocorre, o Ensemble informa o Maestro que repassa a informação para a aplicação através de um *callback* `AcceptedView()`. A aplicação pode sobrecarregar o método *callback* para fazer o que é necessário quando uma mudança no *membership* ocorre.

Quando uma nova lista de membros (visão) é instalada, um dos membros do grupo é eleito como coordenador. Como parte do protocolo de mudança de visão, o coordenador pode enviar por *multicast* uma mensagem de visão para todos os membros do grupo incluídos na nova visão.

Quando uma chamada para `leave()` ocorre, o grupo será reconfigurado para excluir o membro, o qual será notificado da terminação do protocolo pelo Maestro por um *callback* `Exit()`.

O Maestro suporta o modelo de grupos fechados, onde os objetos devem ser membros do grupo para enviar mensagens *multicast*. Clientes externos podem se conectar ao grupo através de canais de comunicação externos [VAY98].

Depois que um objeto `Maestro_GroupMember` se junta ao grupo, ele pode enviar mensagens ponto-a-ponto ou *multicast* para os outros membros do grupo. Para o envio de mensagens, o Maestro provê dois *downcalls*, `cast()` para mensagens *multicast* e `send()` para mensagens ponto-a-ponto. A invocação de qualquer um destes métodos por um objeto é seguida por uma chamada a um *callback* `ReceiveCast` (ou `ReceiveSend`) no destino da mensagem. Casos de falhas no envio de mensagens são tratados pelo Ensemble, de acordo com as opções de configuração escolhidas na inicialização.

O sistema de comunicação de grupos Ensemble implementa um mecanismo de detecção de falhas, usado para detectar a não disponibilidade de membros do grupo, usado pelo Maestro. Com o Maestro rodando sobre o Ensemble, o uso do mecanismo de detecção de falhas pode ser solicitado colocando-se a propriedade de grupo correspondente (chamada *Suspect*) nas opções de configuração. Suspeitas de falhas externas são relatadas para o Maestro com o *downcall* `suspect()`. Um parâmetro especifica a lista de objetos membros do grupo que devem ser excluídos do *membership*.

O Maestro também inclui uma implementação de um tipo abstrato de dado para grupos clientes e servidores. As interfaces para ambos são quase idênticas, sendo que clientes são os membros do grupo que não possuem o estado do grupo. Os clientes conhecem a semântica do grupo e são notificados de mudanças no *membership*. Servidores e clientes podem explicitamente enviar mensagens para outros membros do grupo e atendem o mesmo conjunto de propriedades (definidas pelo Ensemble). No Maestro, clientes podem ser transformados em servidores. Neste caso, um protocolo de transferência de estado é usado para atualizar o cliente com o estado corrente dos servidores membros do grupo. Também, servidores podem se tornar clientes em certas situações.

O Maestro implementa grupos clientes e servidores com a classe Maestro_ClSv, que é definida como uma subclasse de Maestro_GroupMember. A aplicação irá definir uma subclasse de Maestro_ClSv a qual irá sobrecarregar os callbacks protegidos exportados com uma implementação apropriada.

Outras classes que compõem o Maestro, suas características e operações podem ser encontradas na dissertação de doutorado de Alexey Vaysburd [VAY98] ou na Internet, no endereço http://www.cs.cornell.edu/Info/Projects/Ensemble/Maestro.

4 Serviço de Gerenciamento de Réplicas

A carga de processamento muda continuamente durante o progresso do sistema e, num dado momento, o posicionamento de uma réplica em um determinado nodo talvez não seja mais desejável. Um posicionamento estático de réplicas é uma solução inadequada, pois pode ser necessário mudar a distribuição das réplicas com a adição de novas réplicas, remoção de réplicas antigas ou modificação do posicionamento das réplicas atuais durante a execução do sistema.

O sistema *Ape*, em fase de projeto e desenvolvimento na UFRGS, é um **sistema de gerenciamento de réplicas** que se propõe a reconfigurar dinamicamente a distribuição de réplicas sobre uma rede, considerando as mudanças das variáveis do ambiente, tais como a densidade do fluxo de mensagens, a ocorrência de falhas e a carga do sistema distribuído, objetivando tolerância a falhas e bom desempenho.

Ape significa abelha em italiano. Este nome sugere um sistema distribuído "perfeito", tal como uma colméia de abelhas. Ele provê um conjunto de serviços – criação, eliminação, posicionamento e migração – para a reconfiguração das réplicas dos objetos, permitindo balanceamento de carga, detecção de falhas e reintegração automática de nodos.

O RPM foi construído para fornecer informações para o serviço de gerenciamento de réplicas poder realizar o seu trabalho, sendo esse o motivo por apresentar o *Ape* neste capítulo. O *Ape* está sendo desenvolvido sobre o sistema de comunicação de grupos Ensemble, com a utilização da sua interface Maestro.

4.1 Mapeamento de Objetos Para Nodos Físicos

Devido ao alto custo da replicação, nem todos os objetos do sistema necessitam ser replicados e nem todos os nodos possuem réplicas de todos os objetos. Somente objetos servidores, que possuem recursos críticos usados por outros objetos do sistema, devem ser replicados (replicação parcial).

O conjunto de réplicas de um dado objeto é chamado de **grupo de objetos replicados**. O objetivo de um serviço de gerenciamento de réplicas é o mapeamento de um grupo de objetos replicados a um **grupo de nodos servidores de objetos** ou simplesmente **grupo de nodos** (fig. 4.1). O sistema distribuído pode suportar diversos objetos em cada nodo. Cada objeto em um nodo pode ser membro de um grupo de objetos replicados ou não. Um bom mapeamento pode aumentar a disponibilidade do objeto e o desempenho do sistema.

O *Ape* é modelado usando conceitos e primitivas do sistema de comunicação de grupos Ensemble e sua interface de programação Maestro. Qualquer nodo na rede que deseja receber os serviços do *Ape* deve pertencer a um grupo de nodos e é chamado **membro do grupo de nodos**. Isto acontece porque a interface C++ do Ensemble (Maestro) usa o modelo de grupos fechados. O grupo de nodos é um grupo físico do Ensemble.

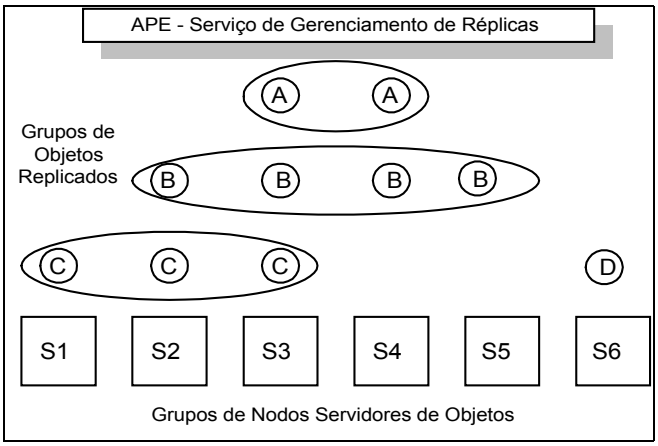

FIGURA 4.1 – Mapeamento de Objetos para Nodos Físicos

Se o cliente quer acessar um objeto replicado ele envia uma mensagem para o grupo de réplicas. O grupo é modelado como um único objeto e possui um único endereço lógico. O cliente não deve saber quantas réplicas compõem o grupo e nem em quais nodos elas estão posicionadas.

Cada réplica em um grupo recebe e executa todas as mensagens na mesma ordem, para refletir um estado global único. O estado global é suportado por um protocolo de consistência de réplicas e pelo sistema de comunicação de grupos - Ensemble.

4.2 Serviços do *Ape*

O *Ape* provê um conjunto de serviços para reconfigurar o mapeamento das réplicas de objetos visando tolerância a falhas e balanceamento de carga. Cada serviço do *Ape* é iniciado automaticamente quando alguns limites de confiabilidade ou desempenho do sistema são atingidos. Por exemplo, uma grande quantidade de mensagens trafegando na rede pode reduzir o desempenho da mesma. Dessa forma o número de mensagens que cruzam a rede em um determinado intervalo de tempo pode ser uma métrica para iniciar um serviço de reconfiguração.

O *Ape* usa métricas derivadas das variáveis do ambiente para iniciar os seus serviços. Diversas métricas podem ser utilizadas [HAC89]. Entretanto, por uma decisão de projeto, para não prejudicar o desempenho divulgando e processando várias métricas, está sendo definido um conjunto mínimo de variáveis do ambiente.

Os serviços fornecidos pelo *Ape* podem ser visualizados na figura 4.2 e são descritos a seguir.

FIGURA 4.2 – Serviços Executados Pelo *Ape*

Serviço de Replicação

O *Ape* sincroniza as réplicas usando o protocolo de réplicas ativas [SCH93]. Neste protocolo todas as réplicas recebem as requisições dos clientes, realizam o processamento e enviam a resposta ao cliente. O cliente aguarda até receber a primeira ou todas as respostas. Réplicas em estado de falha não realizam o processamento e, portanto, não enviam a resposta ao cliente.

Serviço de Migração de Réplicas

Para tornar a migração possível, um objeto espera até que a execução de todos os seus métodos sejam finalizados e então encapsula seu estado em uma mensagem. A mensagem será enviada e interpretada por outros nodos. Para evitar a indisponibilidade do objeto a imagem do objeto no nodo de origem somente é removida quando a nova versão do objeto é instalada no nodo de destino. Técnica similar é utilizada para criar uma nova réplica de um objeto, sendo que o objeto original não é removido.

Rubin [RUB96] explorando a migração automática de objetos sugere as seguintes condições para a decisão de migração de um objeto:

I - Forças gravitacionais: indicam que o objeto deve permanecer onde ele está. Isto inclui propriedades como o tamanho do objeto, a dependência dos métodos em utilizarem recursos locais e as dependências de comunicação dos objetos com outros objetos locais;

II - Forças atrativas: sugerem que um objeto deva migrar para um novo nodo. Inclui propriedades como dados, comunicação e dependência de recursos que os métodos necessitam para a sua execução;

III - Forças externas: desempenham um papel importante nas decisões de migração e estão relacionadas ao estado atual do sistema.

No projeto do *Ape*, está sendo investigado considerar forças similares para decidir quando um objeto deve ser migrado ou não.

Serviço de Remoção de Réplicas

Manter o estado global de todos os membros em um grupo de réplicas gera *overhead*, principalmente quando o sistema realiza diversas operações de escrita. Para evitar este *overhead* pode ser necessário remover as réplicas excedentes.

O serviço de remoção somente exclui uma réplica. Uma réplica pode ser removida quando o *Ape* determina que o desempenho do sistema está abaixo de um limite pré-determinado. A decisão de qual réplica será removida de um grupo é baseada em variáveis de ambiente tais como a carga de cada nodo em cada grupo de nodos e a relação entre leituras e atualizações nas requisições locais dos objetos.

A remoção não se aplica quando a réplica é o último objeto no grupo de replicação. Devem ser investigadas as mesmas forças consideradas no serviço de migração para a realização do serviço de remoção.

Serviço de Criação de Réplicas

Algumas vezes novas réplicas devem ser criadas para satisfazer um aumento nas requisições de serviço dos clientes. Réplicas que são posicionadas nos mesmos nodos

ou próximas dos seus clientes evitam longos tempos de transmissão e de resposta de mensagens.

Existem outras razões para uma réplica ser criada. Devido a falhas ou remoções pode existir um número muito pequeno ou somente um único objeto em um grupo de réplicas – o número de réplicas não é suficiente para garantir a necessária disponibilidade de serviço e desempenho. Nessa situação, uma nova réplica em um novo nodo deve ser adicionado ao sistema.

Entretanto, a criação de réplicas deve ser fortemente analisada antes de ser realizada [HAC89] devido ao fato de que cada operação de escrita solicitada pelo cliente a uma réplica deve ser difundida por *multicast* para todas as outras réplicas do grupo de replicação para manter o estado global consistente. Uma grande quantidade de membros em um grupo diminui o desempenho do sistema.

Serviço de Detecção de Falhas

As ferramentas de comunicação de grupos [HAY98] já possuem o serviço de detecção de falhas. O serviço de detecção de falhas mantém uma lista com os membros que estão com suspeita de falhas. Os membros suspeitos são removidos da nova visão.

Este serviço, que é provido pelo Ensemble, é utilizado pelo *Ape* para implementar o serviço confiável de detecção de falhas.

Serviço de Reintegração de Nodo

Para manter a tolerância a falhas o *Ape* fornece o serviço de reintegração de nodos. A reintegração inicia quando um nodo reinicia ou quando um nodo restaurado retorna à rede. O nodo divulga uma mensagem de reintegração. O *Ape* reconhece esta mensagem e o nodo se prepara para iniciar o serviço de reintegração.

Após reintegrado, o novo nodo está apto a receber réplicas por criação ou migração. Como ele entra no sistema sem carga, o novo nodo é um forte candidato para receber novas réplicas.

5 RPM – *Replica Placement Manager*

Para o *Ape* prover o seu serviço de reconfiguração dinâmica das réplicas de um sistema distribuído, ele necessita saber o estado dos nodos membros em relação à carga de trabalho. Novas réplicas não podem ser criadas ou movidas para nodos sobrecarregados porque isto não contribui para a melhoria do desempenho do sistema. Quando necessário criar ou mover uma réplica, o *Ape* utiliza o serviço do RPM – *Replica Placement Manager*(fig. 5.1-a).

O RPM pode ser considerado um sub-serviço do *Ape*. Ele, quando solicitado pelo *Ape*, obtém as cargas de trabalho dos nodos vivos (ativos) do sistema e cria uma lista ordenada, contendo as cargas e a identificação dos nodos, que é enviada ao serviço de gerenciamento, em resposta à solicitação (fig. 5.1-b). Com essa informação, o serviço de gerenciamento evita posicionar réplicas em nodos que não contribuam para que a aplicação possua um bom desempenho.

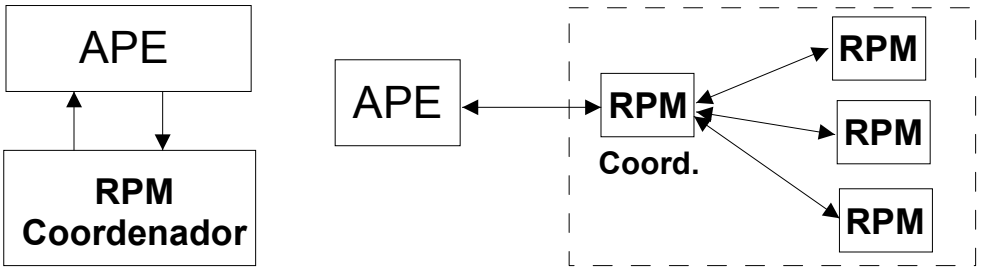

FIGURA 5.1 – Interação do Ape com o RPM

5.1 Descrição do Serviço de Posicionamento

O RPM está presente em cada um dos membros do grupo de nodos servidores de objetos. Estes nodos também formam um grupo gerenciado pelo Ensemble, que provê o suporte adequado para a troca confiável de mensagens entre os nodos, necessárias para o serviço (fig. 5.2).

Uma característica dos grupos do Ensemble é a presença de um nodo coordenador do grupo (fig. 5.2). Este é escolhido automaticamente pelo Ensemble e, na ocorrência de uma falha no nodo coordenador, um novo coordenador é eleito pelo *serviço de membership*. Os outros nodos são chamados de nodos membros do grupo.

O nodo coordenador desempenha um papel importante no RPM. Toda a comunicação do *Ape* com o RPM é feita através do nodo coordenador. Ele é o responsável pela solicitação das informações de carga dos nodos membros do grupo (inclusive a dele próprio, pois ele também é um candidato a receber uma réplica) e pelo processamento da escolha dos melhores nodos para receberem uma réplica. Esta escolha é baseada na carga de trabalho atual dos nodos. Outra função do coordenador é devolver a resposta ao solicitante do serviço, neste caso o sistema *Ape*.

O *Ape* envia mensagens ao RPM solicitando serviços. O RPM decodifica a mensagem, executa a solicitação nela contida e devolve a lista ordenada de nodos candidatos ao *Ape*.

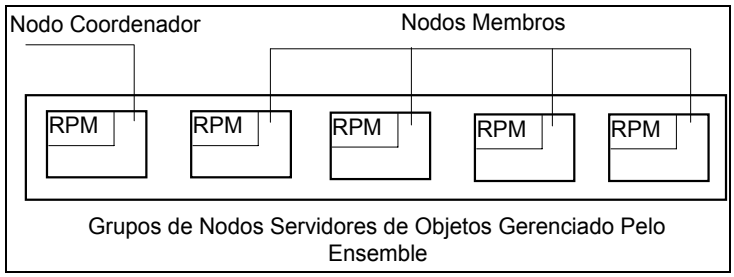

FIGURA 5.2 – Sistema RPM

5.2 Mensagens de Solicitação de Serviços

A mensagem de solicitação de serviços, interpretada pelo RPM deve ser composta dos seguintes campos (fig 5.3):

Mens_Tipo	Mens_Obj	Mens_N

FIGURA 5.3 – Mensagens de Solicitação de Serviços

- **Campo Mens_Tipo:** Este campo, de preenchimento obrigatório, determina o tipo da mensagem. Os tipos de mensagens podem ser um, dois ou três. As mensagens do tipo um são geradas quando o *Ape* deseja criar uma ou mais réplicas de um objeto. As mensagens do tipo dois são geradas quando o *Ape* quer verificar a carga dos nodos que possuem um determinado objeto para identificar se novas réplicas deste objeto devem ser criadas ou migradas. As mensagens do tipo três simplesmente solicitam que sejam retornadas os "N" nodos menos carregados do sistema.
- **Campo Mens_Obj:** Este campo é obrigatório para mensagens do tipo um e dois. Para mensagens do tipo três este campo é desconsiderado. Ele contém a indicação do nome do objeto que se quer criar uma réplica, para mensagens tipo um. Para mensagens tipo dois, indica que se deseja saber a carga dos nodos que possuem este objeto.
- **Campo Mens_N:** Este campo indica o número de réplicas que se quer criar, para mensagens do tipo um. Para mensagens do tipo dois este campo é desconsiderado e para mensagens do tipo três ele indica que se deseja saber quais são os "N" nodos menos carregados do sistema.

O RPM executa os seguintes serviços:

1. Determinação de quais nodos do sistema são melhores para criar uma ou mais de uma réplica de um objeto. Para o RPM executar esta tarefa, ele deve receber uma mensagem com o valor "1" no campo **Mens_Tipo** (fig. 5.4). A mensagem deve conter o nome do objeto que deve ser criado, no campo **Mens_Obj**, e o número de réplicas necessárias, no campo **Mens_N**. O RPM devolve ao *Ape* uma lista com os "N" melhores nodos (menos carregados), que não possuem o objeto, para a criação destas réplicas.
2. Determinação da localização e da carga dos nodos do sistema que possuem um determinado objeto. Quando o *Ape* necessitar o estado dos nodos (carga de trabalho) das réplicas de um determinado objeto ele envia uma mensagem com o

valor “2” no campo **Mens_Tipo** (fig. 5.4). Esta mensagem deve conter o nome do objeto e o parâmetro **N = 0**.

3. Determinação de carga. Quando o *Ape* necessitar saber quais nodos que estão menos carregados ele envia ao RPM uma mensagem com o valor “3” no campo **Mens_Tipo** (fig. 5.4). Esta mensagem solicita que os “N” nodos mais leves sejam retornados.

Mens_Tipo	Mens_Obj	Mens_N	Descrição
1	CLIENTES	2	Quais são os dois nodos mais leves do sistema que não possuem uma réplica do objeto CLIENTES?
2	CLIENTES	0	Qual a carga dos nodos que possuem o obj. CLIENTES?
3		5	Quais os cinco nodos mais leves do sistema?

FIGURA 5.4 – Exemplo de Mensagens Enviadas Pelo *Ape* ao RPM.

5.3 Determinação das Cargas dos Nodos

Para quantificar o conceito de carga, um índice de carga é utilizado. Esse índice, preferencialmente, é uma variável positiva que assume o valor zero se o recurso estiver ocioso, aumentando gradativamente à medida que a carga aumenta [FER87]. Um bom índice de carga deve apresentar as seguintes características [SHI95]:

- O índice de carga não deve levar em consideração apenas as necessidades de CPU de um processo, mas também os requisitos de operações de E/S e memória. Por exemplo, uma tarefa típica de E/S requer pouco tempo de CPU para computações. Se o índice de carga é baseado somente nas necessidades computacionais de um processo, então a carga do processador ao qual esta tarefa foi atribuída é subestimada;
- O índice de carga deve refletir quantitativamente as estimativas qualitativas da carga no *host*. Por exemplo, se o tamanho de uma fila é considerada como uma boa indicação da carga de um processador, então a definição do índice de carga deveria ser calculado em uma função do tamanho da fila, ou do somatório dos tempos de execução das tarefas presentes na fila;
- Visto que o tempo de resposta de uma tarefa é mais afetado pela carga futura de um processador do que pela carga atual, o índice de carga deveria ser utilizado para prever a carga em um futuro próximo. Por exemplo, se um processador tem sido altamente utilizado em um passado recente, ele também irá ser altamente utilizado no futuro próximo. Na verdade, baseado nesse argumento deve-se calcular o índice de carga em função da história de utilização do processador;
- O índice de carga deve ser relativamente estável, ou seja, flutuações de alta freqüência na carga devem ser desconsideradas ou ignoradas; e
- É necessário que exista uma relação direta entre o índice de carga e o desempenho do sistema. Esse relacionamento é importante sob duas perspectivas: 1) o índice de carga, como parte integrante do processo de balanceamento de carga, é usado para

melhorar o desempenho do sistema; e 2) o índice de desempenho pode ser utilizado para refinar o índice de carga.

Deve-se ressaltar que é difícil, talvez impossível, encontrar um índice de carga que satisfaça todos os requisitos acima. No entanto, os índices de carga podem ser julgados de acordo com o grau que satisfazem tais características.

Muitos índices para a medição de carga tem sido propostos, tais como: tamanho da fila de CPU (instantâneo), tamanho médio da fila da CPU (nos últimos *t* segundos), utilização da CPU (valores instantâneos ou médios), taxa de troca de contexto, taxa de chamadas do sistema e uso da memória [WOL97].

Ferrari e Zhou [FER87] apresentam um estudo comparativo dos diferentes índices de carga e relatam que os métodos de balanceamento de carga baseados na informação sobre o tamanho de fila, são melhores que os métodos baseados na utilização de CPU. Além disso, os índices que consideram o tamanho médio da fila são melhores que os índices instantâneos. Também concluíram que um intervalo de atualização relativamente curto é necessário para refletir as mudanças mais recentes na carga do sistema.

Wolffe afirma que o índice de carga mais comumente assumido é o tamanho da fila de processos na CPU [WOL97] e que métodos de balanceamento de carga baseados na informação do tamanho da fila de processos tem se mostrado serem tão efetivos quanto aqueles baseados em medidas mais sofisticadas, tais como funções agregadas (combinações de índices de carga lineares). Entretanto,apesar de um índice de carga simples ser desejado, está claro que o tamanho da fila de processos em execução na CPU representa um índice de carga de trabalho muito diferente para CPU's cujas capacidades e velocidades diferem significativamente.

Como o RPM é dinâmico, interessa a ele valores dinâmicos e simples, de fácil aquisição. Dessa forma, considerou-se, inicialmente, como índice de carga de trabalho apenas o número de processos em execução na CPU. Este índice, obtido de forma instantânea, revelou-se inconsistente com os objetivos do RPM, não conseguindo dar idéia do estado do nodo. Isso ocorre porque, mesmo em um sistema ocioso, há muitos processos rodando, incluindo *daemons* de correio eletrônico e *newsgroups*, gerenciadores de janelas e outros processos. Então, contar processos pode não esclarecer sobre a carga instantânea.

Foi escolhido então, como índice de carga para o RPM a média de processos em execução na CPU nos últimos 60 segundos, sendo que o número de processos na CPU é coletado a cada 5 segundos. Este é também um índice de carga simples e facilmente obtido que apresentou resultados consistentes com os objetivos do RPM.

Os resultados dos testes são apresentados com maiores detalhes no capítulo sete.

5.4 Funcionamento do RPM

O RPM é composto por dois módulos, um deles executado pelo coordenador do grupo e o outro executado pelos nodos membros do grupo.

Quando o *Ape* envia ao coordenador do grupo RPM uma solicitação de serviços, este envia esta mensagem por *multicast* a todos os membros do grupo e entra em um estado de espera pelo retorno de todas as mensagens.

Quando a mensagem chega nos nodos membros do grupo, estes procedem a sua análise. De acordo com o tipo da mensagem, os nodos verificam a presença ou não do objeto no nodo (isto é realizado somente nas mensagens tipo um e dois), obtém o índice

de carga do nodo e enviam uma mensagem de retorno para o coordenador. Esta mensagem de retorno é estruturada da seguinte maneira (fig. 5.5):

Id_Nodo	Carga_Nodo	Bogomips	Memória	Candidato

FIGURA 5.5 – Mensagem de Retorno dos Nodos

O campo **Id_Nodo** armazena a identificação (*endpoint*) do nodo.

O campo **Carga_Nodo** armazena o índice que determina a carga de trabalho do nodo.

O campo **Bogomips** armazena um índice que representa a velocidade da CPU do nodo. Essa informação é importante como critério de desempate. Se dois ou mais nodos apresentarem o mesmo índice de carga de trabalho, é dada preferência ao nodo que possui um processador mais veloz.

Bogomips é uma palavra inventada por Linus Torwalds, o criador do sistema operacional Linux. O kernel do Linux necessita de um loop de temporização **(tempo de *loop*)**, o qual deve ser calibrado para a velocidade do processador da máquina. Assim, o kernel mede, na inicialização, quão rápido um certo tipo de loop é executado, determinando esse índice e armazenando-o no arquivo **cpuinfo**, do sistema de arquivos **/proc**. "Bogo" provém de *bogus*, uma palavra em inglês que significa que algo é falso. MIPS significa "Milhões de Instruções Por Segundo" e serve para medir a velocidade de computação de um programa. Dessa maneira, o valor do Bogomips é uma indicação não-científica da velocidade do processador, mas que serve aos propósitos do RPM.

O campo **Memória** armazena a quantidade de memória do nodo. Essa informação também é utilizada como critério de desempate. Se dois ou mais nodos empatarem na média de processos e velocidade do processador, é dada prioridade ao nodo que possuir a maior memória.

O campo **Candidato** sinaliza, de acordo com o tipo da mensagem de solicitação de serviço (tipos um e dois), se o nodo deve ser considerado ou não na lista de resultados fornecida ao *Ape*. Esta sinalização é feita com o valor "1" se o nodo deve ser considerado e com o valor "0" se não deve ser considerado.

A medida que as mensagens chegam no nodo coordenador, este as vai colocando numa fila. Após chegarem as mensagens de todos os nodos o coordenador processa a escolha dos nodos que atendem a requisição do *Ape*. Após isso, o RPM envia uma mensagem ao *Ape* com o resultado da requisição.

A fig. 5.6 mostra o diagrama de estados do nodo coordenador.

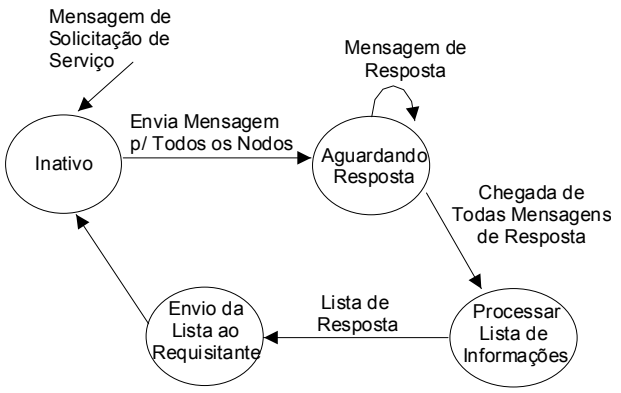

FIGURA 5.6 – Diagrama de Estados do Nodo Coordenador do RPM

5.5 Comportamento sob Falha

Enquanto o RPM está aguardando a chegada das mensagens de resposta, duas situações distintas de falha podem ocorrer, falha no nodo coordenador e falha nos nodos membros. Em caso de falha em um nodo membro, o serviço de *membership* exclui o nodo falho, tornando o grupo menor. Essa mudança é refletida no nodo coordenador, que aguarda um número menor de mensagens de retorno, não causando nenhum problema para o sistema de gerenciamento de réplicas. No caso de uma falha no nodo coordenador, o sistema de comunicação de grupos elege um novo coordenador, porém este não possui o estado do antigo. Assim, o *Ape* deve fazer uma nova requisição ao novo coordenador.

6 Implementação

Este capítulo apresenta, na seção 6.1, o ambiente de desenvolvimento do RPM. Na seção 6.2 é descrito o algoritmo do RPM e sua implementação e na seção 6.3 é descrito o programa NPROC.C, responsável por manter atualizado o arquivo que contém a média de processos em execução no nodo.

6.1 Ambiente de Desenvolvimento

O RPM foi desenvolvido em uma LAN Ethernet 10 Mbps, composta de quatro microcomputadores rodando o sistema operacional Linux, distribuição Red Hat, versão 6.0. Como camada intermediária entre o sistema operacional e a interface de programação foi instalado o sistema de comunicação de grupos Ensemble, versão 0.61.

O sistema Ensemble é um *middleware* que fornece suporte para a construção de aplicações de comunicação de grupos confiáveis (ver seção 3.3). Para o RPM interagir com o Ensemble foi utilizado o *Maestro Group Communication Tools*, que é uma interface em C++ para o Ensemble (ver seção 3.3.3). Com essa interface é possível utilizar as funções de gerenciamento e comunicação de grupos do Ensemble.

O *Maestro* provê um conjunto de ferramentas, interfaces e serviços que são executados no topo de sistema de comunicação distribuída do Ensemble. Os próprios projetistas definem o *Maestro* como um pacote aberto, que permite uma arquitetura flexível para adicionar novas interfaces, ferramentas, módulos e serviços apropriados para as necessidades específicas de aplicações de usuários.

A classe `GroupListener` implementa um tipo abstrato de dados para grupos do Ensemble. Há métodos para juntar elementos ao grupo (*join*), enviar mensagens e métodos que podem ser invocados quando uma mensagem chega, entre outros. Uma aplicação de usuário para o Maestro é a definição de uma sub-classe da classe `Maestro_GroupListener`. Nesta aplicação devem ser definidos os serviços, como ordenação total (Total), detecção de falha (Suspect), propriedades de grupo (Gmp), etc.

A classe `Maestro_GroupListener` possui os métodos para enviar mensagens ponto-a-ponto (*send*) e multicast (*cast*). Todas as mensagens são entregues na mesma visão em que são enviadas. Contudo, se um método *send* ou um *cast* são invocados durante uma mudança de visão, a mensagem será enviada e entregue na próxima visão.

Para a comunicação *multicast* deve ser utilizado o servidor *gossip* do Ensemble. O servidor *gossip* trabalha em conjunto com o transporte UDP para simular a difusão (*broadcasts*) e *multicasts* em sistemas que não possuem IP *multicast*.

O RPM coleta informações que estão disponíveis no nodo, as insere numa mensagem e utiliza o sistema de comunicação de grupos para enviar esta mensagem ao nodo coordenador. A figura 6.1 mostra os dados que o RPM recolhe. A carga do sistema é mantida num arquivo chamado MEDIAS.INF. Este é atualizado a cada cinco segundos por um processo em execução paralela (o processo NPROC). Maiores detalhes sobre a obtenção da média de processos são encontrados na seção 6.3. Os outros arquivos são gerados e lidos pelo RPM.

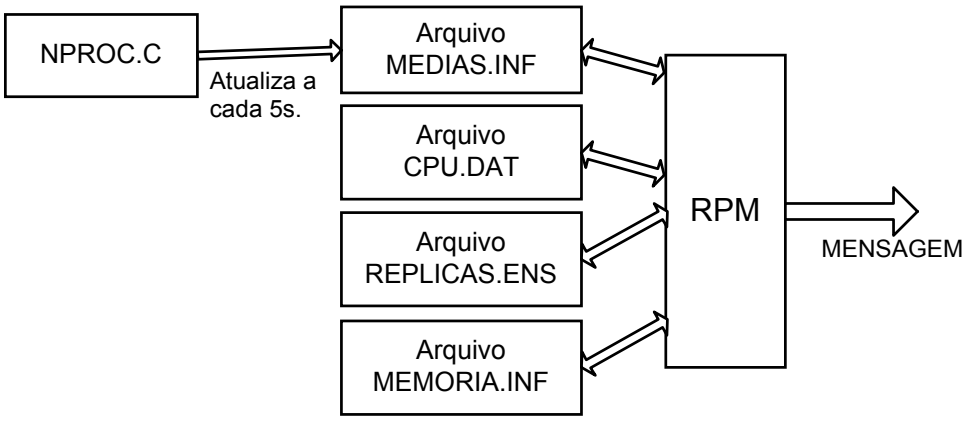

FIGURA 6.1 – Dados Coletados Pelo RPM

6.1.1 Algoritmo do RPM

Uma visão geral das funções que compõem o RPM é dada na figura 6.2. Nesta figura são brevemente descritas as tarefas que cada função realiza. O detalhamento da implementação é encontrado na seção 6.2.

Função CPU ( ){
Função para a obtenção da velocidade do processador (bogomips). Obtém esta informação consultando o arquivo **/proc/cpuinfo**.
Comando executado: **cat /proc/cpuinfo | grep bogomips > cpu.dat**
Com a execução do comando acima é gerado um arquivo texto, que é analisado pela função.
}

Função Localiza( ){
Função para verificar se um objeto está presente ou não no nodo. Para fins de implementação e testes, esta função consulta um arquivo, chamado **REPLICAS.ENS**, que contém a lista de objetos presentes no nodo.
}

Função Memória( ){
Esta função obtém a memória do nodo. Para obter esta informação, é utilizado o comando top.
Este cria um arquivo, chamado MEMORIA.INF, que é analisado pela função.
Comando executado: **top –i –n1 | grep Mem: > MEMORIA.INF**
}

Função Index( ){
Esta função obtém o número médio de processos em execução na CPU. Para obter este valor, o RPM consulta o arquivo MEDIAS.INF. Esta informação é periodicamente atualizada por um processo em execução paralela.
}

Classe MyGroupListener{
Classe para a recepção e o tratamento de mensagens. Os dois métodos utilizados pelo RPM são os métodos ReceivedSend – para recepção de mensagens ponto-a-ponto – e ReceivedCast – para a recepção de *multicasts*.

Método ReceivedSend( ){
Este método é executado no coordenador do grupo. Ele é responsável por ler o conteúdo das mensagens direcionadas ao coordenador e inserir este conteúdo em uma lista.
}

```
Método ReceivedCast( ){
        Este método é executado por todos os nodos do sistema quando uma mensagem de
        multicast é enviada. Ele lê o conteúdo da mensagem e realiza o processamento, de
        acordo com o tipo da mensagem (tipo um, dois ou três).

        Leitura e atribuição dos campos da mensagem a variáveis.
        Obtenção da média de processos
        Caso Mensagem_tipo
                Caso 1:
                        Verifica a existência do objeto no nodo
                        Se o objeto NÃO existir, a variável Considera recebe valor  1;
                Caso 2:
                        Verifica a existência do objeto no nodo
                        Se o objeto existir, a variável Considera recebe o valor 1;
                Caso 3:
                        Envia a carga para o coordenador
        CHEGOU=1 // Sinaliza o recebimento (chegada) de uma mensagem.
        Se é COORDENADOR, então
            insere na lista as informações do coordenador
  }
Função Ordena_Lista( ){
        Tem por função processar a lista de respostas, no coordenador. Esta função ordena a
        lista em relação ao número de processos, de forma crescente. Ordena também pela
        velocidade do processador (bogomips) e pela memória. Após ordenar a lista, ela cria a
        lista de resultados.
}
Função Main( ){
        Inicializa o grupo
        Obtém os Bogomips do nodo
        Obtém a memória do nodo
        Laço  // executado continuamente
          Se COORDENADOR então
                Envia mensagem de multicast para o grupo
                Aguarda a chegada de todas as mensagens
                Processa a escolha (procedimento Ordena_lista())
                Retorna a resposta ao solicitante
          Fim Se
          Se MEMBRO então
                Se chegou mensagem do coordenador (chegou=1)
                        Envia a mensagem para o coordenador
                Fim Se
          Fim Se
        Fim do Laço
}
```

FIGURA 6.2 – Algoritmo do RPM

6.2 Implementação do RPM

O RPM foi implementado na linguagem C, sobre o sistema operacional Linux, distribuição RedHat, versão 6.0. Para utilizar a interface Maestro, deve ser incluído o protótipo “Maestro_Group.h”.

6.2.1 Obtenção da Velocidade do Processador

O RPM necessita saber a velocidade do processador de cada nodo do sistema distribuído. Isso é necessário como um critério de desempate. Se ocorrer de dois ou mais nodos apresentarem como média de processos em execução na CPU o mesmo valor, será dada prioridade para receber uma réplica o nodo que possuir o processador mais veloz. O RPM obtém esta informação no arquivo **cpuinfo** do sistema de arquivos **/proc**. O RPM realiza uma chamada para o comando **cat** do Linux. Este comando apresenta um arquivo na saída padrão (normalmente o vídeo). Porém, este resultado é fornecido como entrada para o comando **grep**, que procura a linha que contém a palavra *bogomips*. O resultado da execução destes comandos é direcionado para um arquivo texto chamado **CPU.DAT** (fig. 6.3 - linha 9). Em seguida este arquivo é lido (fig. 6.3 – linha 15), a velocidade do processador é atribuída à **variável cpu** (fig. 6.3 – linha 17) e a função é finalizada.

```
// ***** Função para Obtenção dos Bogomips *****
1  float cpu(void){
2    struct leitura_cpu{
3      char str1[8];
4      char str2[1];
5      float lcpu;
6    }lt;
7    FILE *ptr;
8    float cpu;
9    system("cat /proc/cpuinfo |grep bogomips > CPU.DAT");
10   ptr = fopen("cpu.dat","r");
11   if (ptr==NULL){
12     printf(" Arquivo nao pode ser aberto \n");
13     exit(1);
14   }
15   fscanf(ptr,"%s%s%f",&lt.str1,&lt.str2,&lt.lcpu);
16   fclose(ptr);
17   cpu=lt.lcpu;
18   return cpu;
19 }
```

FIGURA 6.3 – Função CPU

6.2.2 Verificação da Existência de um Objeto no Nodo

O sistema de gerenciamento de réplicas não deve criar ou migrar uma réplica de um objeto para um nodo que já possua este objeto. Um servidor de nomes é necessário para o *Ape* fornecer o serviço de gerenciamento de réplicas. O RPM deve consultar o serviço de nomes para determinar a existência ou não do objeto no nodo.

Porém, o serviço de nomes ainda está em fase de projeto, sendo assim, para fins de teste e implementação, foi criado um arquivo chamado **REPLICAS.ENS**. Este arquivo simula o serviço de nomes. Ele armazena o nome dos objetos que estão presentes no nodo permitindo que o RPM o consulte.

A função recebe por passagem de parâmetro o nome do objeto (fig.6.4 – linha 1). Abre o arquivo **REPLICA.ENS** para leitura (fig. 6.4 – linha 8) e após verifica a presença ou não do objeto no nodo (fig. 6.4 – linhas 13-17). Se o objeto está presente no nodo, à **variável achou** é atribuído o valor “1”.

```
// ***** Verifica se um Objeto Esta Replicado no Nodo ****
1  int localiza(char *objeto2)
2  {
3    file *ptr2;
4    char linha[80],auxobj[80]={' '};
5    int teste,achou,xm;

     (...)

6    teste=0;
7    achou=0;
8    ptr2 = fopen("REPLICAS.ENS","r");
9    if (ptr2==NULL){
10     printf(" Arquivo nao pode ser aberto \n");
11     exit(1);
12   }
13   do{
14     fgets(linha,sizeof(linha)+1,ptr2);
15     teste=strcmp(linha,objeto2);
16     if(teste==0) achou=1;
17    }while(!feof(ptr2));
18   fclose(ptr2);
19   return achou;
20 }
```

FIGURA 6.4 – Função Localiza

6.2.3 Obtenção da Memória

Para obter o total de memória do nodo o RPM realiza uma chamada ao comando **top**. A saída deste comando é filtrada pelo comando **grep**, criando um arquivo chamado **MEMORIA.INF** (fig. 6.5 – linha 12). Após o arquivo **MEMORIA.INF** é lido e a quantidade de memória do nodo é atribuída à **variável mem** (fig. 6.5 – linha 20).

A memória também é utilizada como um critério de desempate. Se acontecer de dois ou mais nodos apresentarem a mesma média de processos e processadores com a mesma velocidade, o RPM elege o nodo que possui a maior quantidade de memória RAM.

```
// ***** Funcao para Obtencao da Memoria *****
1  long memoria(void){
2    struct le_mem{
3      char string1[4];   // String " Mem:"
4      char dado1[7];     // String que contem o total de memoria
5      char string2[3];   // String " av, "
6      char dado2[7];     // String que contem a memoria usada
7      char string3[5];   // String " used, "
8      char dado3[6];     // String que contem a memoria livre
9    }dados;
10   long mem=0;
11   FILE *ptr;
12   system("top -i -n1|grep Mem: > MEMORIA.INF");
13   ptr = fopen("MEMORIA.INF","r");
14   if (ptr==NULL){
15     printf(" Arquivo nao pode ser aberto \n");
16     exit(1);
```

```
17    }
18    fscanf(ptr,"%s%s%s%s%s%s%s",&dados.string1,&dados.dado1,
      &dados.string2,&dados.dado2,&dados.string3,&dados.dado3);
19    fclose(ptr);
20    mem=atol(dados.dado1);
21    return mem;
22  }
```

FIGURA 6.5 – Função Memória

6.2.4 Obtenção da Carga do Nodo

A carga de um nodo é determinada pela média de processos em execução na CPU nos últimos 60s. Esta média é gravada num arquivo chamado MEDIAS.INF, que é atualizado a cada 5s. Quando o RPM necessita da carga de um nodo ele simplesmente lê o conteúdo deste arquivo (fig. 6.6 – linha 10) e atribui o resultado da leitura para a **variável media**. Para ser transportada em uma mensagem do Ensemble, a variável media deve ter a sua parte inteira e fracionária separadas (o Ensemble só consegue transportar com perfeição valores inteiros). Isso é feito pelas linhas 12 a 14, da figura 6.6. No nodo destino, elas serão unidas novamente.

```
1  //  Obtencao do Indice da Carga de Trabalho do Nodo (Media) *****
2  void index(void)
3  {
4    FILE *ptr;
5    ptr = fopen("MEDIAS.INF","r");
6    if (ptr==NULL){
7      printf(" Arquivo nao pode ser aberto \n");
8      exit(1);
9    }
10   fscanf(ptr,"%f",&media);
11   fclose(ptr);
12   frac=modf(media,&inteiro);
13   mproci=inteiro;
14   mprocf=frac * 100;
15   return;
16 }
```

FIGURA 6.6 – Função Index

6.2.5 Recepção de Mensagens

Para a recepção das mensagens é utilizada a classe `MyGroupListener`. Esta classe é uma sub-classe da classe `Maestro_GroupListener`. Quando uma mensagem é difundida por *multicast*, todas os nodos que pertencem ao grupo recebem esta mensagem no método `receivedCast`. Quando uma mensagem é enviada de um nodo a outro (ponto-a-ponto), o nodo destinatário recebe esta mensagem no método `receivedSend`. Nos métodos é inserido o código necessário para fazer o tratamento das mensagens recebidas.

O nodo coordenador do grupo, quando recebe uma solicitação de serviços do *Ape*, divulga esta mensagem por *multicast* para todos os membros ativos do grupo. Esta mensagem é processada pelo método `receivedCast` em todos os nodos. Neste método, a mensagem é recebida e o seu conteúdo é lido (fig. 6.7 – linhas 31 - 34). Logo

em seguida, a mensagem é analisada e de acordo com o seu tipo (fig.6.7 – linhas 40 – 65) é verificado se o objeto existe ou não no nodo, através da função `localiza` (fig. 6.7 – linhas 42; 52). De acordo com o resultado retornado da `função` localiza, é determinado se o nodo deve ser considerado ou não na lista de resultados finais, atribuíndo-se o valor "0" (não considerar) ou o valor "1" (considerar) à variável considera.

Os nodos membros enviam a mensagem contendo o número médio de processos em execução na CPU, a velocidade do processador, o tamanho da memória, a sua identificação (*endpoint*) e se o nodo deve ser ou não considerado na lista de resultados (variável considera) para o nodo coordenador. Esta é uma mensagem ponto-a-ponto que é recebida e tratada pelo método `receivedSend`, executado somente no nodo coordenador. Após esta mensagem ser lida ela é colocada numa fila de mensagens recebidas (fig. 6.7 – linhas 10 - 26) e, quando todas as mensagens forem recebidas, esta fila é processada pela função `ordena_lista`.

```
// ***** Classe Para a Recepcao das Mensagens *****
1  class MyGroupListener: public Maestro_GroupListener {
2  public:
3    MyGroupListener() { myState = 0; }
4    int c,i;
5    Maestro_String s2;
6    int myState;
7    (...)
8    void receivedSend(Maestro_EndpID &sender, Maestro_Message &msg)
9    {
10     msg >> idnodo;
11     msg.read(mem_total2);
12     msg.read(aux_frac2);
13     msg.read(aux_int2);
14     msg.read(nprocf2);
15     msg.read(nproci2);
16     msg.read(considera2);
17     bogo2= (float) aux_frac2 / 100;
18     bogo2=bogo2 + aux_int2;
19     nproc2=(float) nprocf2/100;
20     nproc2=nproc2 + nproci2;
21     mensagens--;
22     lista[indl].ax_nproc=nproc2;
23     lista[indl].ax_bogomips=bogo2;
24     lista[indl].ax_mem=mem_total2;
25     lista[indl].ax_ident=idnodo;
26     lista[indl].ax_cons=considera2;
27     indl++;
28    }
29    void receivedCast(Maestro_EndpID &sender, Maestro_Message &msg)
30    {
31     msg.read(n2);
32     msg.read(tamanho);
33     msg.read(objeto2,tamanho);
34     msg.read(tipo2);
35     mensagens--;
36     // ***** Analise de Mensagens Recebidas *****
37     considera=0;
38     index();
39     busca=0;
40     switch(tipo2){
41       case 1:
42         busca=localiza(objeto2);
```

```
43          if(busca==0){ // Objeto nao existe no nodo
44          //  printf("Mens_tipo=1 objeto nao existe no nodo\n");
45            considera=1;
46          }
47          else{ // objeto existe - nao considerar o nodo;
48            considera=0;
49          }
50          break;
51        case 2:
52          busca=localiza(objeto2);
53          if(busca==1){
54          //  printf("Mens_tipo=2  objeto existe no nodo\n");
55            considera=1;
56          }
57          else{ // Objeto NAO existe - NAO considerar o nodo
58            considera=0;
59          }
60          break;
61        case 3:
62          //printf("Mens_tipo=3-Nao verifica o a pres. do objeto\n");
63          considera=1;
64          break;
65      } // fim switch
66      chegou=1;
67     // Insere as informacoes do COORDENADOR na lista
68      if(infgrupo.coordinator==infgrupo.myEndpID){
69        ax_media=(float) mprocf / 100;
70        ax_media=ax_media + mproci;
71        lista[0].ax_nproc= ax_media;
72        lista[0].ax_bogomips=bogomips;
73        lista[0].ax_mem=ax_mem;
74        lista[0].ax_ident=infgrupo.myEndpID;
75        lista[0].ax_cons=considera;
76      }
77    }
78   ( ... )
```

FIGURA 6.7 – Classe MyGroupListener

6.2.6 Ordenação da lista de resultados

A medida que as mensagens com a resposta – média de processos, velocidade do processador, quantidade de memória, identificação do nodo e se o nodo deve ser considerado ou não – chegam, elas são inseridas em uma lista. Após a chegada de todas as mensagens o RPM deve ordenar essa lista para apresentar o resultado que atenda à solicitação. A função `ordena_lista` realiza esta tarefa, seguindo os seguintes passos:

1. Ordena a lista em relação ao número de processos (média), de forma crescente (fig. 6.8 – linhas 14 - 33). O algoritmo de ordenação utilizado é o de ordenação por inserção. Inicialmente, é ordenado os dois primeiros membros da lista. Em seguida, é inserido o terceiro membro na sua posição ordenada em relação aos dois primeiros membros. Então insere o quarto elemento na lista dos três elementos. O processo continua até que todos os elementos tenham sido ordenados.
2. Uma lista auxiliar é criada para auxiliar no processo de ordenação secundário. Esta lista armazena a posição inicial e final das médias de processos repetidas na lista de resultados (fig. 6.8 – linhas 34 – 65).

3. Na eventualidade de dois ou mais nodos apresentarem a mesma média de processos, uma nova ordenação se faz necessária como critério de desempate. O RPM deve escolher o nodo que possuir o melhor processador. Esta é uma ordenação decrescente que utiliza como chave a velocidade do processador (bogomips) (fig. 6.8 – linhas 66 – 89). Por motivos de desempenho, esta ordenação não é realizada para toda a lista e sim somente para os membros da lista que apresentarem a mesma média de processos.
4. Se ocorrer um novo empate, ou seja dois ou mais nodos apresentarem a mesma média de processos e o mesmo processador, é gerada mais uma lista auxiliar (fig. 6.8 – linhas 90 – 121) e uma nova ordenação deverá ser feita, agora considerando-se como critério de desempate a quantidade de memória.
5. Se necessário, uma nova ordenação, agora somente dos nodos que possuirem a mesma média de processos e velocidade do processador é realizada. Esta é uma ordenação crescente que utiliza como chave a quantidade de memória dos nodos (fig. 6.8 – linhas 122 – 145).
6. Apresenta os resultados do processamento da lista, considerando o tipo da mensagem e o número de réplicas solicitadas. No estágio atual, estes resultados estão sendo impressos na tela, porém deverão ser repassados ao *Ape*, quando o mesmo estiver concluído (fig. 6.8 – linhas 150 - 193).

```
// ***** Ordenacao das Cargas *****
1 void ordena_lista(){
2  Maestro_EndpID ident2;
3  register int ind1,ind2;
4  float mproc,bogo;
5  int cons,mem,nitens;
6  struct auxiliar{  // lista auxiliar para ajudar na ordenacao
7    int inicio;
8    int fim;
9  }lista_aux[50];
10  int aux_in,aux_fim,ix,ctrl;
11  int l_in,l_fim,disp,controle;
12  cons=0;
13  nitens=indl;
14 // Primeira Parte - Ordena pela media de processos
15  for(ind1=1;ind1<nitens;++ind1){
16    mproc=lista[ind1].ax_nproc;
17    bogo=lista[ind1].ax_bogomips;
18    mem=lista[ind1].ax_mem;
19    cons=lista[ind1].ax_cons;
20    ident2=lista[ind1].ax_ident;
21    for(ind2=ind1-1;ind2>=0 && mproc<lista[ind2].ax_nproc;ind2--){
22      lista[ind2+1].ax_nproc = lista[ind2].ax_nproc;
23      lista[ind2+1].ax_bogomips = lista[ind2].ax_bogomips;
24      lista[ind2+1].ax_mem=lista[ind2].ax_mem;
25      lista[ind2+1].ax_cons = lista[ind2].ax_cons;
26      lista[ind2+1].ax_ident = lista[ind2].ax_ident;
27    }
28    lista[ind2+1].ax_nproc=mproc;
29    lista[ind2+1].ax_bogomips=bogo;
30    lista[ind2+1].ax_mem=mem;
31    lista[ind2+1].ax_cons=cons;
32    lista[ind2+1].ax_ident=ident2;
33  }
34  // Segunda Parte - Gera lista auxiliar para ordenacao
```

```
35  aux_in=0;
36  aux_fim=0;
37  for(ix=0;ix<50;ix++) lista_aux[ix].inicio=lista_aux[ix].fim=0;
38  mproc=0;
39  ctrl=0;
40  ind2=0;
41  for(ind1=0;ind1<nitens;ind1++){
42    if(mproc==lista[ind1].ax_nproc){
43      if(ctrl==0){
44        aux_in=ind1-1;
45        ctrl=1;
46      }
47    }
48    else{
49      if(ctrl==1){
50        aux_fim=ind1-1;
51        lista_aux[ind2].inicio=aux_in;
52        lista_aux[ind2].fim=aux_fim;
53        aux_in=aux_fim=0;
54        ind2++;
55        ctrl=0;
56      }
57    }
58    mproc=lista[ind1].ax_nproc;
59  }
60  if(ctrl==1){
61    aux_fim=ind1-1;
62    lista_aux[ind2].inicio=aux_in;
63    lista_aux[ind2].fim=aux_fim;
64    ind2++;
65  }
66 // Terceira Parte - Ordena pelos bogomips
67  for(ix=0;ix<3;ix++){
68    l_in=lista_aux[ix].inicio;
69    l_fim=lista_aux[ix].fim+1;
70    for(ind1=l_in+1;ind1<l_fim;++ind1){
71      mproc=lista[ind1].ax_nproc;
72      bogo=lista[ind1].ax_bogomips;
73      mem=lista[ind1].ax_mem;
74      cons=lista[ind1].ax_cons;
75      ident2=lista[ind1].ax_ident;
76      for(ind2=ind1-1;ind2>=l_in &&
            bogo>lista[ind2].ax_bogomips;ind2--){
77        lista[ind2+1].ax_nproc = lista[ind2].ax_nproc;
78        lista[ind2+1].ax_bogomips = lista[ind2].ax_bogomips;
79        lista[ind2+1].ax_mem=lista[ind2].ax_mem;
80        lista[ind2+1].ax_cons = lista[ind2].ax_cons;
81        lista[ind2+1].ax_ident=lista[ind2].ax_ident;
82      }
83      lista[ind2+1].ax_nproc=mproc;
84      lista[ind2+1].ax_bogomips=bogo;
85      lista[ind2+1].ax_mem=mem;
86      lista[ind2+1].ax_cons=cons;
87      lista[ind2+1].ax_ident=ident2;
88    }
89  }
90  // Quarta Parte - Gera lista auxiliar para ordenacao II
91  aux_in=0;
92  aux_fim=0;
93  for(ix=0;ix<50;ix++) lista_aux[ix].inicio=lista_aux[ix].fim=0;
94  bogo=0;
```

```
95  ctrl=0;
96  ind2=0;
97  for(ind1=0;ind1<nitens;ind1++){
98    if(bogo==lista[ind1].ax_bogomips){
99      if(ctrl==0){
100        aux_in=ind1-1;
101        ctrl=1;
102       }
103    }
104    else{
105      if(ctrl==1){
106        aux_fim=ind1-1;
107        lista_aux[ind2].inicio=aux_in;
108        lista_aux[ind2].fim=aux_fim;
109        aux_in=aux_fim=0;
110        ind2++;
111        ctrl=0;
112       }
113    }
114    bogo=lista[ind1].ax_bogomips;
115  }
116  if(ctrl==1){
117    aux_fim=ind1-1;
118    lista_aux[ind2].inicio=aux_in;
119    lista_aux[ind2].fim=aux_fim;
120    ind2++;
121  }
122 // Quinta Parte - Ordena pela memoria
123  for(ix=0;ix<3;ix++){
124    l_in=lista_aux[ix].inicio;
125    l_fim=lista_aux[ix].fim+1;
126    for(ind1=l_in+1;ind1<l_fim;++ind1){
127      mproc=lista[ind1].ax_nproc;
128      bogo=lista[ind1].ax_bogomips;
129      mem=lista[ind1].ax_mem;
130      cons=lista[ind1].ax_cons;
131      ident2=lista[ind1].ax_ident;
132      for(ind2=ind1-1;ind2>=l_in && mem>lista[ind2].ax_mem;ind2--){
133        lista[ind2+1].ax_nproc = lista[ind2].ax_nproc;
134        lista[ind2+1].ax_bogomips = lista[ind2].ax_bogomips;
135        lista[ind2+1].ax_mem=lista[ind2].ax_mem;
136        lista[ind2+1].ax_cons = lista[ind2].ax_cons;
137        lista[ind2+1].ax_ident=lista[ind2].ax_ident;
138      }
139      lista[ind2+1].ax_nproc=mproc;
140      lista[ind2+1].ax_bogomips=bogo;
141      lista[ind2+1].ax_mem=mem;
142      lista[ind2+1].ax_cons=cons;
143      lista[ind2+1].ax_ident=ident2;
144    }
145  }
146  for(ix=0;ix<nitens;ix++){
147    if(lista[ix].ax_cons==1)
148       disp++;
149  }
150  cout << " LISTA CONSIDERADA - TIPO  " << tipo << "\n";
151  switch(tipo){
152    case 1:
153        if(n>=disp)
154          controle=disp;
155        else
```

```
156           controle=n;
157         for(ix=0;ix<nitens;ix++){
158         if(lista[ix].ax_cons && controle>0){
159            cout << "  Proces.:  " << lista[ix].ax_nproc;
160            cout << "  Bogomips  " << lista[ix].ax_bogomips;
161           cout << "  Memoria   " << lista[ix].ax_mem;
162            cout << "  Consid..: " << lista[ix].ax_cons;
163            cout << "  Identif.: " << lista[ix].ax_ident;
164           controle--;
165         }
166       }
167       break;
168     case 2:
169       for(ix=0;ix<nitens;ix++){
170         if(lista[ix].ax_cons){
171            cout << "  Proces.:  " << lista[ix].ax_nproc;
172            cout << "  Bogomips  " << lista[ix].ax_bogomips;
173            cout << "  Memoria   " << lista[ix].ax_mem;
174            cout << "  Consid..: " << lista[ix].ax_cons;
175            cout << "  Identif.: " << lista[ix].ax_ident;
176         }
177       }
178       break;
179     case 3:
180       if(n<=nitens) controle=n;
181       for(ix=0;ix<nitens;ix++){
182         if(lista[ix].ax_cons && controle>0){
183            cout << "  Proces.:  " << lista[ix].ax_nproc;
184            cout << "  Bogomips  " << lista[ix].ax_bogomips;
185            cout << "  Memoria   " << lista[ix].ax_mem;
186            cout << "  Consid..: " << lista[ix].ax_cons;
187            cout << "  Identif.: " << lista[ix].ax_ident;
188           controle--;
189         }
190       }
191       break;
192   }
193   return;
```

FIGURA 6.8 – Função Ordena_Lista

6.2.7 Função Principal do RPM

Um nodo, para poder enviar e receber mensagens utilizando um sistema de comunicação de grupos, deve ser integrado a um grupo. Para pertencer a um grupo, alguns parâmetros de inicialização devem ser especificados. A estrutura `Maestro_GroupOptions` contém os campos que devem ser inicializados para permitir que um nodo se integre a um determinado grupo, tais como o nome do grupo (fig. 6.9 – linha 3), se o nodo possui o *status* de servidor ou não (fig. 6.9 – linha 4) e as propriedades do grupo (fig. 6.9 – linha 5), entre outras. As propriedades do Ensemble escolhidas para compor a pilha de protocolos são "*Total:Gmp:Sync:Heal:Switch:Frag:Suspect:Flow*". Estas propriedades garantem, principalmente, a ordenação total de mensagens, gerenciamento de *membership*, sincronismo virtual e detecção de falhas. Abaixo a função de cada uma destas propriedades:

- Total – Mensagens totalmente ordenadas
- Gmp – Propriedades do *membership* do grupo

- Sync – Sincronização das visões
- Heal – Reunificação de partições
- Switch – Mudança de protocolo
- Frag – Fragmentação – reunificação
- Suspect – Detecção de falhas
- Flow – Controle de Fluxo

Como mencionado anteriormente, o grupo possui um nodo com o *status* de coordenador, enquanto que os outros membros do grupo são simplesmente membros do grupo. No RPM, estes nodos executam tarefas distintas. O nodo coordenador recebe uma solicitação e cria uma mensagem com as informações da solicitação. Logo em seguida, ele a envia por *multicast* para todos os membros do grupo (fig. 6.9 – linhas 19 - 25).

Após enviar o multicast o coordenador entra em um estado de espera pelo retorno das mensagens dos nodos membros (fig. 6.9 – linha 26). Quando todas as mensagens retornarem, a função `ordena_lista` é invocada e o resultado é apresentado na tela.

Os nodos membros do grupo aguardam a chegada da mensagem de solicitação de serviços provinda do coordenador. Quando esta mensagem chega, o método `receivedCast` processa a mensagem e seta o valor da variável **chegou** em "1". Isto indica que uma mensagem foi recebida, processada e a resposta deve ser enviada ao coordenador (fig. 6.9 – linha 33).

O nodo membro cria uma mensagem de resposta, contendo as suas informações – média de processos, velocidade do processador, quantidade de memória e se o nodo deve ser considerado ou não na lista de resultados – e envia esta mensagem ao coordenador do grupo (fig. 6.9 – linhas 34 – 44).

```
// ***** Funcao Principal do Programa *****
1 int main(int argc, char **argv) {
2   Maestro_GroupOptions ops;
3   ops.groupName = "rpm";
4   ops.serverFlag = ((argc > 1) && (argv[1][0] = 's'));
5   ops.properties = "Total:Gmp:Sync:Heal:Switch:Frag:Suspect:Flow";
6   MyGroupListener listener;
7   Maestro_Group *group = new Maestro_Group(listener, ops);
8   bogomips=cpu();
9   frac=modf(bogomips,&inteiro);
10  aux_int=inteiro;
11  aux_frac=frac * 100;
12  mem_total=memoria();
13  aux_mem = mem_total / 1000;
14  while (1){
15    Maestro_Message msg;
16    if (infgrupo.coordinator==infgrupo.myEndpID){ //COORDENADOR
17        printf("\n COORDENADOR \n");
18        mensagens=membros;
19        mutex.lock();
20        msg.write(tipo);
21        msg.write(objeto,strlen(objeto));
22        msg.write(strlen(objeto));
23        msg.write(n);
24        group->cast(msg);
25        mutex.unlock();
26        while(mensagens!=0 || mensagens==membros);
27        ordena_lista();
```

```
28          sleep(5);
29        }
30      if (infgrupo.coordinator!=infgrupo.myEndpID){ //MEMBRO DO GRUPO
31        printf(" MEMBRO \n");
32        Maestro_Message msg2;
33        if (chegou==1){
34          mutex.lock();
35          msg2.write(considera);
36          msg2.write(mproci);
37          msg2.write(mprocf);
38          msg2.write(aux_int);
39          msg2.write(aux_frac);
40          msg2.write(ax_mem);
41          msg2 << infgrupo.myEndpID;
42          group->send(infgrupo.coordinator,msg2);
43          msg2.reset();
44          mutex.unlock();
45          chegou=0;
46       }
47       sleep(5);

          ( ... )

48        } // fim if MEMBRO DO GRUPO
49      } // FIM DO WHILE
50  }
```

FIGURA 6.9 – Função Principal do RPM

6.3 Cálculo da Média de Processos

Para obter o índice que determina a carga – tamanho médio da fila de processos em execução no último minuto - o RPM lê o arquivo **MEDIAS.INF**. Este arquivo é atualizado a cada 5 segundos por um programa, batizado de **NPROC,** que é executado concomitantemente ao RPM em todos os nodos do sistema distribuído.

O **NPROC** utiliza um algoritmo de janela deslizante. A figura 6.10 mostra uma representação gráfica do funcionamento deste algoritmo. Nesta figura a barra cinza representa a janela, que corresponde a quais processos do vetor serão considerados para o cálculo da média.

O número de processos em execução é coletado a cada 5 segundos e é atribuído a uma posição de um vetor (vetor de processos). Após ser atribuído o número de processos para a posição 11 do vetor, a média destes processos é calculada e gravada no arquivo **MEDIAS.INF**. Isso pode ser visualizado no ponto A, da figura 6.10.

A cada nova atribuição de valores ao vetor de processos (a cada 5 segundos), a janela desliza para a direita, para que sejam considerados os 12 processos mais atuais e uma nova média é calculada e gravada no arquivo **MEDIAS.INF**. Isso se repete até que seja atribuído o número de processos à posição 23 do vetor (ponto B, da figura 6.10).

Após a janela alcançar o limite superior do vetor (posição 23), ela se desloca para a esquerda. Os novos valores são atribuídos nas posições do vetor, de forma decrescente, a partir da posição 11, até atingir a posição 0 (ponto C, da figura 6.10). Após alcançar o limite inferior do vetor de processos, a janela desliza novamente para a direita, reiniciando o processo.

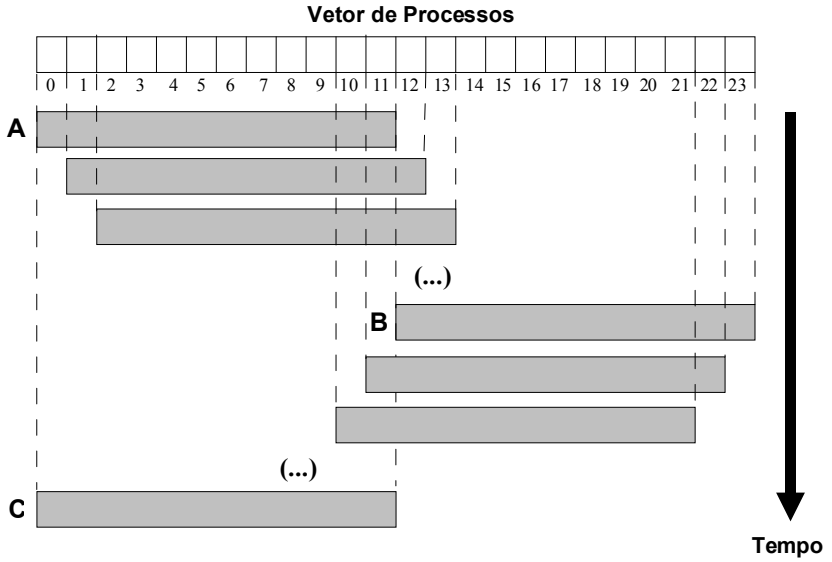

FIGURA 6.10 – Atualização da Média de Processos – Algoritmo de Janela Deslizante

Para se obter o número de processos em execução em um determinado instante, é realizada uma chamada para o comando **top** do Linux. O comando top é um monitor do sistema que mostra a atividade do processador em tempo real. Ele exibe na tela uma lista, atualizada regularmente (por padrão, a cada 5 s.), das tarefas no sistema que usam com mais intensidade a CPU e fornece uma interface interativa para manipulação dos processos.

A figura 6.11 mostra um exemplo de uma tela de saída do comando top. No topo da tela (primeira linha) são dadas algumas informações sobre o estado atual do sistema. A segunda linha mostra o número de processos, diferenciando-os entre seus estados individuais. Os processos em Linux podem estar nos seguintes estados:

- *Running*: O processo está rodando (é o processo corrente no sistema) ou está pronto para rodar (está esperando na fila do escalonador por uma CPU do sistema);
- *Waiting – Interruptable*: O processo está esperando por um recurso ou um evento, mas os sinais não estão bloqueados e ele pode ser interrompido;
- *Waiting – Uninterruptable*: O processo está esperando por um recurso ou evento mas tem os sinais desabilitados que não permitem que ele seja interrompido;
- *Stopped*: O processo está parado;
- *Zombie*: O processo está terminado e está pronto para morrer, mas o escalonador ainda não detectou isto.

```
 10:44am up    1:01,  5 users,  load average:   0.02,  0.06, 0.09
55 processes:  52 sleeping, 1 running,  0 zombie,  2 stopped
CPU states:   4.1% user,  3.6% system,   7.7% nice, 92.3% idle
Mem:  30956K  av, 30424K used,   532K free,  24636K shrd, 4164K buff
Swap: 34236K  av,     0K used, 34236K free               10840K cach

PID  USER    PRI   NI   SIZE   RSS   STAT    %CPU   %MEM TIME  COMMAND
352  magnus   18    0    688    688   R        5.5%   2.2 0:01  top
107  root      5    0   5952   5952   S        1.8%  19.2 1:39  X
111  root      2    0   1592   1592   S        0.4%   5.1 0:04  xterm
```

FIGURA 6.11 – Exemplo de Saída do Comando TOP

No programa NPROC (fig. 6.12), a cada 5 segundos é invocada a função `nprocess` (fig. 6.12 – linhas 56; 66; 80). Nesta função, é realizada uma chamada ao comando **top** do sistema operacional e a saída deste comando é direcionada para o comando **grep**, que vai procurar a linha que contém a palavra *running*. O resultado da execução destes comandos é gravado no arquivo **SYSTEM.DAT** (fig. 6.12 – linha 20). Após ser criado, este arquivo é lido e o número de processos que estão em running é atribuído à variável **nproc** (fig. 6.12 – linha 29).

O resultado da função `nprocess` é armazenado numa posição de um vetor (fig. 6.12 – linhas 57; 67; 81). Após 60 segundos decorridos do início da sua execução, o número médio de processos é calculado e gravado no arquivo **MEDIAS.INF** (fig. 6.12 – linhas 61-62). Em seguida à primeira gravação (após decorridos os 60 segundos iniciais), o programa utiliza um algoritmo de janela deslizante, dessa forma atualizando o arquivo **MEDIAS.INF** a cada 5 segundos (fig. 6.12 – linhas 64 – 94).

```
// *  Obtencao da Media de Processos em Execucao  *
1  #include <stdio.h>
2  #include <stdlib.h>
3  int nproc,inicio,fim,ctrl;
4  int processos[24],indp=0;
5  float media=0;
6  struct leitura{
7    int dado1;          // numero total de processos
8    char string1[10];   // string "processes:"
9    int dado2;          // processos inativos
10   char string2[9];    // string "sleeping,"
11   int dado3;          // processos rodadando
12   char string3[8];    // string "running,"
13 }dados;
// ***** Calculo do Indice da Carga de Trabalho do Nodo *****
14 void nprocess(void)
15 {
16   FILE *ptr;
17   char str1[15];
18   char aux1[6],aux2[6];
19   char linha1[80],linha2[80];
20   int i,k,ind,pos,tam,difer;
21   system("top -i -n1|grep running > system.dat");
22   ptr = fopen("system.dat","r");
23   if (ptr==NULL){
24     printf(" Arquivo nao pode ser aberto \n");
25     exit(1);
26   }
27   fscanf(ptr,"%d%s%d%s%d%s",&dados.dado1,&dados.string1,
            &dados.dado2,&dados.string2,&dados.dado3,&dados.string3);
28   fclose(ptr);
29   nproc = dados.dado3;
30   return;
31 }
// ***** Funcao para Calculo do Numero de Processos *****
32 void calc_processos(int inicio,int fim){
33   int xx,mediaac;
34   mediaac=0;
35   for(xx=inicio;xx<=fim;xx++){
36     mediaac=mediaac + processos[xx];
37   }
38   media=(float) mediaac / 12;
39  return;
40 }
```

```
// ***** Escreve a Media de Processos em um arquivo *****
41 void esc_saida(){
42   FILE *ptr;
43   ptr=fopen("MEDIAS.INF","w");
44   if(ptr==NULL){
45     printf(" Erro na Criacao do arquivo MEDIAS.ENS \n");
46     exit(1);
47   }
48   fprintf(ptr,"%f",media);
49   fclose(ptr);
50 }
// ***** Funcao Principal do Programa *****
51 void main(void){
52   indp=0;
53   inicio=0;
54   fim=inicio+11;
55   do{
56     nprocess();
57     processos[indp]=nproc;
58     indp++;
59     system("sleep 5");
60   }while(indp<12);
61   calc_processos(0,11);
62   esc_saida();
63   ctrl=0;
64   while(1){
65     if(ctrl==0){
66       nprocess();
67       processos[indp]=nproc;
68       inicio=indp-11;
69       fim=indp;
70       calc_processos(inicio,fim);
71       esc_saida();
72       indp++;
73       if(indp==24){
74         ctrl=1;
75         indp=12;
76       }
77       system("sleep 5");
78     }
79     if(ctrl==1){
80       nprocess();
81       processos[indp]=nproc;
82       inicio=indp;
83       fim=inicio+11;
84       indp--;
85       calc_processos(inicio,fim);
86       esc_saida();
87       if(indp<0){
88         ctrl=0;
89         indp=12;
90         inicio=-1;
91       }
92       system("sleep 5");
93     }
94   }
95 }
```

FIGURA 6.12 – Cálculo da Média de Processos

7 Ambiente e Resultados dos Testes

O RPM foi testado no laboratório de informática do curso de Ciência da Computação da Universidade de Caxias do Sul. Foram utilizados quatro microcomputadores, interligados por uma rede Ethernet de 10 Mbps. Nesse laboratório os computadores são identificados por nomes de países e possuem uma configuração semelhante, conforme observado na figura 7.1.

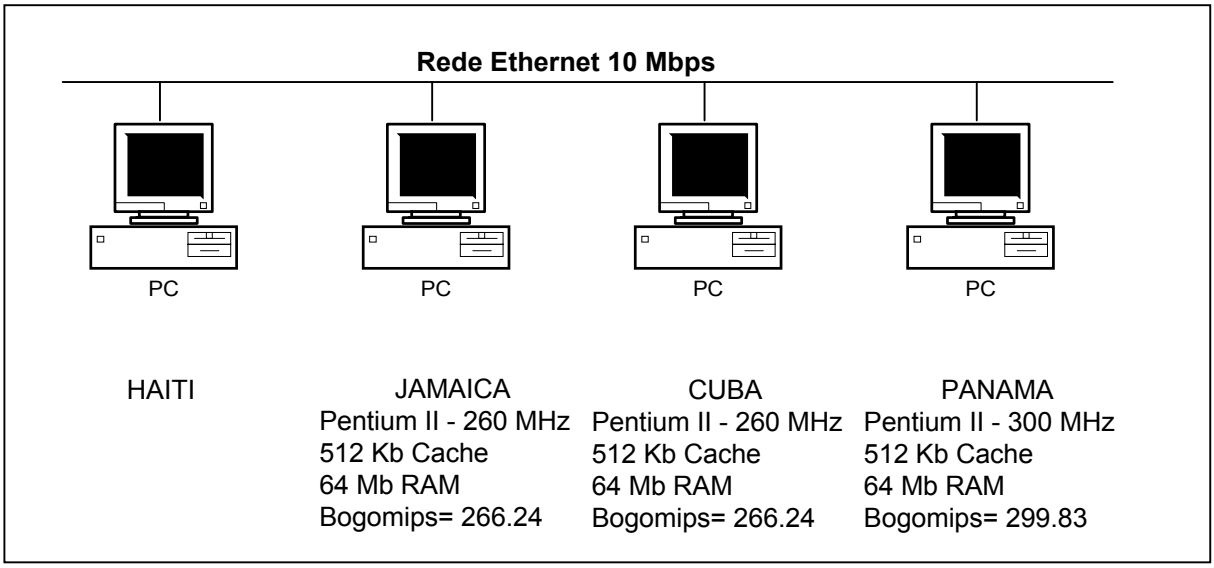

FIGURA 7.1 – Rede Onde os Testes Foram Realizados.

O arquivo REPLICAS.ENS, presente em cada um dos microcomputadores, assume o papel de um servidor de nomes, já que este ainda não está implementado. Este arquivo serve para simular a presença de objetos nos nodos e ele está configurado de acordo com a figura 7.2.

HAITI
Computador
Java
Pedidos
Impressora
Produtos

CUBA
Computador
Java
Tabela
Impressora
Clientes
Funcionários

JAMAICA
Computador
Java
Tabela
Pedidos
Clientes
Notas
Contas

PANAMA
Computador
Tabela
Pedidos
Notas
Estoque

FIGURA 7.2 – Arquivo REPLICAS.ENS em Cada um dos Microcomputadores

A tabela 7.1 mostra a quantidade de réplicas de cada objeto, distribuídas na rede.

TABELA 7.1 – Disposição dos Objetos nos Nodos

Objeto replicado em todos os nodos	*Objetos replicados em três nodos*	*Objetos replicados em dois nodos*	*Objetos Não Replicados*
COMPUTADOR	*JAVA* *TABELA* *PEDIDOS*	*IMPRESSORA* *CLIENTES* *NOTAS*	*PRODUTOS* *FUNCIONÁRIOS* *CONTAS* *ESTOQUE*

Os testes foram divididos em dois grupos: testes comportamentais e testes de execução. O objetivo dos testes comportamentais era provar que o sistema se comporta como o esperado, recomendando, para cada tipo de mensagem, os nodos corretos para receberem uma réplica.

Os testes de execução tem por objetivo provar que o índice de carga escolhido pelo RPM para realizar a escolha dos nodos é consistente e prático. Este índice, batizado de IMP – Índice de Média de Processos – representa a média de processos em execução na CPU nos últimos 60 segundos.

7.1 Testes Comportamentais

Nestes testes quer-se mostrar que o RPM consegue recomendar as máquinas corretas para receberem uma réplica, ou seja, não recomenda uma máquina que já possua o objeto que se quer criar ou migrar e que, no caso de várias máquinas com a mesma média de processos, o critério de desempate – velocidade do processador e quantidade de memória – é utilizado corretamente.

7.1.1 Teste de Mensagens Tipo 1

Procurou-se testar o comportamento do RPM quando o objetivo é criar ou migrar uma nova réplica. Para realizar esta tarefa, o *Ape* solicita uma mensagem do tipo um e o nodo deve verificar se o objeto existe no nodo (consultando o arquivo REPLICAS.ENS). Se o objeto existir, o nodo não pode ser considerado como candidato a receber uma réplica. Se, por outro lado, o objeto não existir, o nodo é apto a receber uma réplica. Procurou-se testar também o número de réplicas solicitadas e o resultado fornecido como resposta.

No primeiro teste realizado (fig. 7.3), procurou-se ver a possibilidade do sistema criar uma nova réplica do objeto COMPUTADOR. Este objeto está presente em todos os nodos do sistema, dessa forma o RPM não pode indicar nenhum nodo e foi exatamente este o resultado deste teste.

1º Teste:

Mens_Tipo	Mens_Obj	Mens_N
1	COMPUTADOR	2

Resultado: NÃO RECOMENDOU NENHUM NODO

FIGURA 7.3 – Primeiro Teste de Mensagem Tipo Um

No segundo teste (fig. 7.4), quer-se criar duas réplicas do objeto NOTAS. As máquinas CUBA e HAITI são as únicas que não possuem este objeto NOTAS, e a prioridade para receber uma réplica é a da máquina CUBA, por possuir um processador mais veloz que a máquina HAITI.

2º Teste:

Mens_Tipo	Mens_Obj	Mens_N
1	NOTAS	2

Resultado:

Identificação	*Bogomips*	*Memória*
Cuba	266,23	63
Haiti	231,83	63

FIGURA 7.4 – Segundo Teste de Mensagem Tipo Um

No próximo teste (fig. 7.5) quer-se criar uma única réplica do objeto TABELA. Este só não está presente na máquina HAITI, dessa maneira, ela é a única possível para receber uma réplica deste objeto.

3º Teste:

Mens_Tipo	Mens_Obj	Mens_N
1	TABELA	1

Resultado:

Identificação	*Bogomips*	*Memória*
Haiti	231,83	63

FIGURA 7.5 – Terceiro Teste de Mensagem Tipo Um

No quarto teste (fig. 7.6), deseja-se criar duas cópias do objeto PEDIDOS, porém este já se encontra replicado em três nodos do sistema. Dessa maneira, a única máquina onde ele pode ser criado ou migrado é a máquina CUBA.

4º Teste:

Mens_Tipo	Mens_Obj	Mens_N
1	PEDIDOS	2

Resultado:

Identificação	*Bogomips*	*Memória*
Cuba	266,23	63

FIGURA 7.6 – Quarto Teste de Mensagem Tipo Um

No quinto teste (fig. 7.7) o RPM foi solicitado para fornecer informações sobre quatro nodos, porém estão disponíveis somente três nodos para receber uma réplica. O RPM escolheu as máquinas PANAMÁ, JAMAICA e HAITI, que são as únicas disponíveis, para criar uma réplica do objeto FUNCIONÁRIOS.

5º Teste:

Mens_Tipo	Mens_Obj	Mens_N
1	FUNCIONÁRIOS	4

Resultado:

Identificação	*Bogomips*	*Memória*
Panamá	299,83	63
Jamaica	266,24	63
Haiti	231,83	63

FIGURA 7.7 – Quinto Teste de Mensagem Tipo Um

No sexto teste (fig. 7.8), deseja-se criar duas réplicas do objeto CONTAS. São candidatas a receber esta réplica as máquinas HAITI, CUBA e PANAMÁ. O RPM indicou as máquinas PANAMÁ e CUBA, pois são as máquinas que possuem o melhor processador.

6º Teste:

Mens_Tipo	Mens_Obj	Mens_N
1	CONTAS	2

Resultado:

Identificação	*Bogomips*	*Memória*
Panamá	299,83	63
Cuba	266,23	63

FIGURA 7.8 – Sexto Teste de Mensagem Tipo Um

No sétimo teste (fig. 7.9), deseja-se criar uma réplica do objeto ESTOQUE. São candidatas para receber esta réplica as máquinas HAITI, CUBA e JAMAICA. Neste caso, tanto a máquina JAMAICA quanto a máquina CUBA possuem o processador com a mesma velocidade. O algoritmo não prevê este tipo de situação. O RPM escolheu a máquina JAMAICA porém, em outra execução, poderia ser escolhida a máquina CUBA.

7º Teste:

Mens_Tipo	Mens_Obj	Mens_N
1	ESTOQUE	1

Resultado:

Identificação	*Bogomips*	*Memória*
Jamaica	266,24	63

FIGURA 7.9 – Sétimo Teste de Mensagem Tipo Um

No oitavo teste (fig. 7.10), deseja-se criar duas réplicas do objeto PRODUTOS. São candidatas a receberem estas réplicas as máquinas CUBA, JAMAICA e PANAMÁ. Foram escolhidas as máquinas PANAMÁ e JAMAICA por possuirem os melhores processadores.

8º Teste:

Mens_Tipo	Mens_Obj	Mens_N
1	PRODUTOS	2

Resultado:

Identificação	*Bogomips*	*Memória*
Panamá	299,83	63
Jamaica	266,24	63

FIGURA 7.10 – Oitavo Teste de Mensagem Tipo Um

7.1.2 Teste de Mensagens Tipo 2

Para determinar a necessidade de migrar uma réplica, o *Ape* pode solicitar a carga de trabalho dos nodos que possuem um determinado objeto. Para isso ele envia ao RPM uma mensagem de tipo dois. Para esse tipo de mensagem o número de réplicas não é necessário.

No primeiro teste (fig. 7.11) executado, procurou-se saber a carga dos nodos que possuem o objeto COMPUTADOR. Este objeto está presente em todos os nodos do sistema, dessa forma, o RPM recebeu informações de todos os nodos do sistema.

1º Teste:

Mens_Tipo	Mens_Obj	Mens_N
2	COMPUTADOR	0

Resultado:

Identificação	*Bogomips*	*Memória*
Panamá	299,83	63
Jamaica	266,24	63
Cuba	266,23	63
Haiti	231,83	63

FIGURA 7.11 – Primeiro Teste de Mensagem Tipo Dois

No próximo teste realizado (fig. 7.12), procurou-se saber a carga dos nodos que possuem o objeto JAVA. Este objeto está presente nas máquinas JAMAICA, CUBA e HAITI. O RPM apresenta estes nodos, com a sua respectiva carga, ordenados pela velocidade do processador.

2º Teste:

Mens_Tipo	Mens_Obj	Mens_N
2	JAVA	0

Resultado:

Identificação	*Bogomips*	*Memória*
Jamaica	266,24	63
Cuba	266,23	63
Haiti	231,83	63

FIGURA 7.12 – Segundo Teste de Mensagem Tipo Dois

No terceiro teste (fig. 7.13), deseja-se saber a carga de trabalho dos nodos que possuem o objeto PEDIDOS. As máquinas que possuem este objeto são PANAMÁ, JAMAICA e HAITI .

3º Teste:

Mens_Tipo	Mens_Obj	Mens_N
2	PEDIDOS	0

Resultado:

Identificação	*Bogomips*	*Memória*
Panamá	299,83	63
Jamaica	266,24	63
Haiti	231,83	63

FIGURA 7.13 – Terceiro Teste de Mensagem Tipo Dois

No quarto teste (fig. 7.14), o RPM solicita que os nodos que possuem o objeto CLIENTES informem a sua carga. Este objeto está presente em dois nodos JAMAICA e CUBA.

4º Teste:

Mens_Tipo	Mens_Obj	Mens_N
2	CLIENTES	0

Resultado:

Identificação	*Bogomips*	*Memória*
Jamaica	266,24	63
Cuba	266,23	63

FIGURA 7.14 – Quarto Teste de Mensagem Tipo Dois

Neste teste (fig. 7.15), o RPM solicita que os nodos que possuem o objeto IMPRESSORA informem a sua carga. Este objeto está presente em dois nodos HAITI e CUBA. O RPM devolve a máquina CUBA encabeçando a lista de resultados por ser a máquina que possui o melhor processador das que possuem o objeto IMPRESSORA.

5º Teste:

Mens_Tipo	Mens_Obj	Mens_N
2	IMPRESSORA	0

Resultado:

Identificação	*Bogomips*	*Memória*
Cuba	266,23	63
Haiti	231,83	63

FIGURA 7.15 – Quinto Teste de Mensagem Tipo Dois

No sexto teste (fig. 7.16), deseja-se saber a carga dos nodos que possuem o objeto FUNCIONÁRIOS. Este nodo se encontra somente na máquina CUBA.

6º Teste:

Mens_Tipo	Mens_Obj	Mens_N
2	FUNCIONÁRIOS	0

Resultado:

Identificação	*Bogomips*	*Memória*
Cuba	266,23	63

FIGURA 7.16 – Sexto Teste de Mensagem Tipo Dois

No sétimo teste (fig. 7.17), é solicitado ao RPM que este obtenha a carga das máquinas que possuem o objeto CONTAS. Este objeto se encontra presente somente na máquina JAMAICA.

7º Teste:

Mens_Tipo	Mens_Obj	Mens_N
2	CONTAS	0

Resultado:

Identificação	*Bogomips*	*Memória*
Jamaica	266,24	63

FIGURA 7.17 – Sétimo Teste de Mensagem Tipo Dois

7.1.3 Teste de Mensagens Tipo 3

Este tipo de mensagem é usado para determinar a carga do sistema. Quando o *Ape* necessita saber quais nodos que estão menos carregados, ele envia ao RPM uma mensagem do tipo três. Esta mensagem não solicita a consulta da presença ou ausência de objetos nos nodos porém deve, obrigatóriamente, conter o parâmetro "N" preenchido com um valor diferente de zero. No primeiro teste de mensagens tipo três o RPM solicitou que todas as quatro máquinas do sistema enviassem a ele a suas cargas. O resultado deste teste pode ser visualizado na figura 7.18.

1º Teste:

Mens_Tipo	Mens_Obj	Mens_N
3		4

Resultado:

Identificação	*Bogomips*	*Memória*
Panamá	299,83	63
Jamaica	266,24	63
Cuba	266,23	63
Haiti	231,83	63

FIGURA 7.18 – Primeiro Teste de Mensagem Tipo Três

No segundo teste realizado o RPM solicitou a carga das duas máquinas mais leves do sistema. O resultado deste teste pode ser visualizado na figura 7.19.

2º Teste:

Mens_Tipo	Mens_Obj	Mens_N
3		2

Resultado:

Identificação	*Bogomips*	*Memória*
Panamá	299,83	63
Jamaica	266,24	63

FIGURA 7.19 – Segundo Teste de Mensagem Tipo Três

7.2 Testes de Execução

O único objetivo destes testes é demonstrar que o índice de carga escolhido serve como uma boa indicação das atividades das máquinas que formam o sistema distribuído e que ele pode ser usado numa estratégia de posicionamento. Este índice,

batizado de IMP – Índice de Média de Processos – representa a média de processos em execução na CPU nos últimos 60 segundos. Ele é obtido da seguinte maneira:

- A cada 5 segundos, o número de processos em execução – no estado *running* - na CPU é capturado e inserido em uma posição de um vetor;
- Após 60 segundos, é somada todas as posições do vetor e o resultado é dividido por 12. Este resultado é gravado num arquivo chamado MEDIAS.INF, que pode ser acessado por qualquer programa que necessite desta informação;
- Após a primeira gravação, a cada nova inserção do número de processos no vetor (a cada 5 segundos), a média é calculada novamente e o resultado é gravado no arquivo MEDIAS.INF.

Para maiores detalhes de como o IMP é obtido, ver seção 6.3.

Estes testes foram realizados alterando-se, manualmente, a carga de processamento dos computadores que formam a rede. A carga é gerada pela execução de diversas instâncias de um processo CPU intensivo. Esse processo é, simplesmente, um *loop* executado eternamente, que não acessa os recursos de entrada e saída (E/S) do computador em que está sendo executado. Dessa forma, procurou-se demonstrar que o IMP é consistente com a carga real dos nodos.

Deve-se ressaltar aqui, que mesmo em uma máquina ociosa, sempre estão em execução diversos processos do sistema operacional, incluindo *daemons* de correio eletrônico e gerenciadores de janelas, entre outros. Os processos colocados em execução por um usuário somam-se aos processos em execução do sistema operacional, assim sendo, o número de processos em execução, num determinado momento do processamento, pode ser maior do que o número de processos disparados pelo usuário. No restante deste texto, é explicado o comportamento do sistema quando é alterado o número de processos de usuário em execução em um nodo.

Para ter conhecimento da carga de trabalho dos nodos, o RPM envia uma mensagem do tipo três, com o campo “N” (nº de réplicas) com o valor “4”. Essa mensagem faz com que todos os nodos enviem a sua média de processos no último minuto para o nodo coordenador.

Estes testes são aqui apresentados na forma de histogramas, para facilitar a visualização dos resultados.

No primeiro teste, foram colocados em execução 10 processos de usuário – processos CPU intensivos - que junto com os processos do sistema operacional formam a carga da máquina. As médias calculadas ficaram com valores muito próximos entre si, como pode ser visto na figura 7.20. Nessa situação, a escolha do RPM foi a máquina PANAMÁ e em segundo lugar a máquina HAITI. Ambas apresentaram a mesma média de processos, mas a máquina PANAMÁ possui o melhor processador dentre as duas, sendo por isso a escolhida prioritariamente para receber uma réplica.

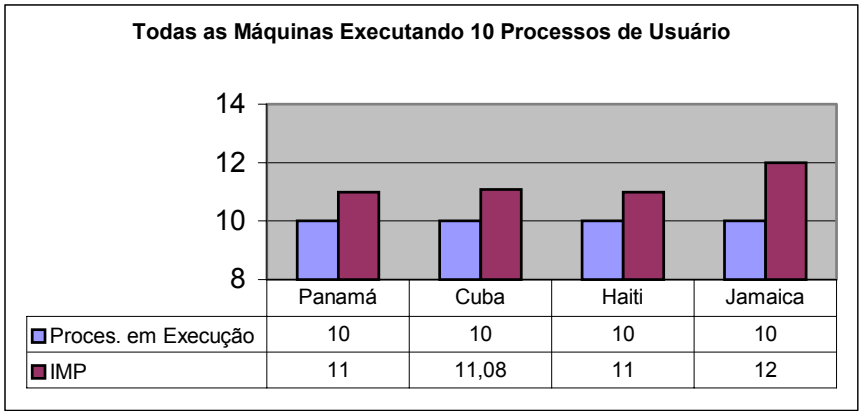

FIGURA 7.20 - Todas as Máquinas Executando 10 Processos de Usuário

No segundo teste (fig. 7.21), alterou-se a distribuição do número de processos de usuário em execução. Neste teste, deixou-se a máquina PANAMÁ executando cinco processos e todas as outras rodando 10 processos de usuário. A média para essa máquina foi bem inferior às demais e seguindo o critério de escolha do RPM, essa seria a máquina prioritária para receber uma réplica. A lista de retorno ficou assim: PANAMÁ em primeiro lugar, CUBA em segundo lugar, HAITI em terceiro lugar e JAMAICA em quarto lugar.

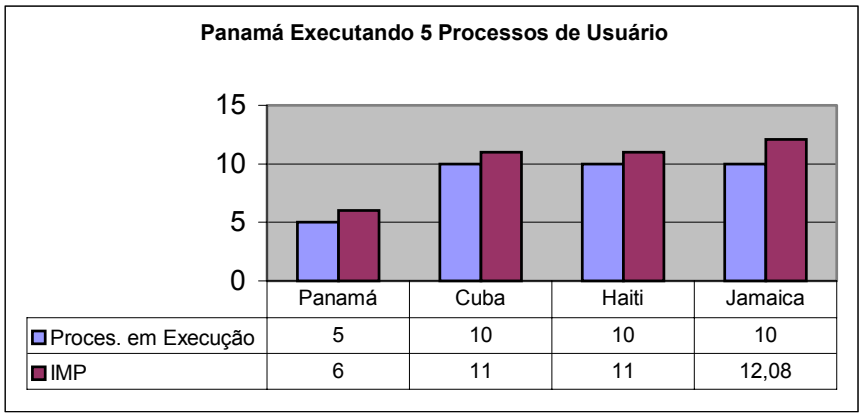

FIGURA 7.21 - Máquina Panamá Executando 5 Processos de Usuário

No terceiro teste (fig. 7.22), foram reduzidos para cinco os processos de usuário em execução na máquina JAMAICA. As médias das duas máquinas ficaram muito próximas, mas, nessa execução, a máquina PANAMÁ ainda é a recomendada por apresentar a menor média de processos em execução. A lista de retorno ficou assim: PANAMÁ em primeiro lugar, JAMAICA em segundo lugar, CUBA em terceiro lugar e HAITI em quarto lugar.

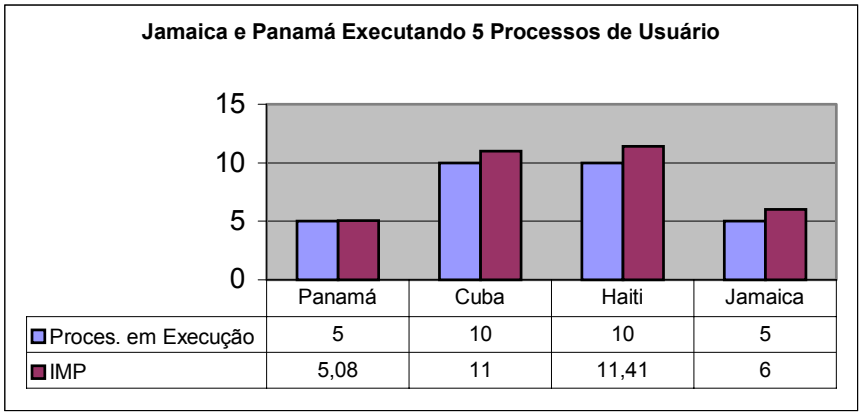

FIGURA 7.22 - Máquinas Jamaica e Panamá Executando 5 Processos de Usuário

O quarto teste é muito semelhante ao terceiro. Neste teste, deixou-se as máquinas CUBA e PANAMÁ rodando cinco processos de usuário. Porém, como pode ser observado na figura 7.23, o menor IMP obtido foi o da máquina CUBA. A lista de retorno ficou assim: CUBA em primeiro lugar, PANAMÁ em segundo lugar, HAITI em terceiro lugar e JAMAICA em quarto lugar.

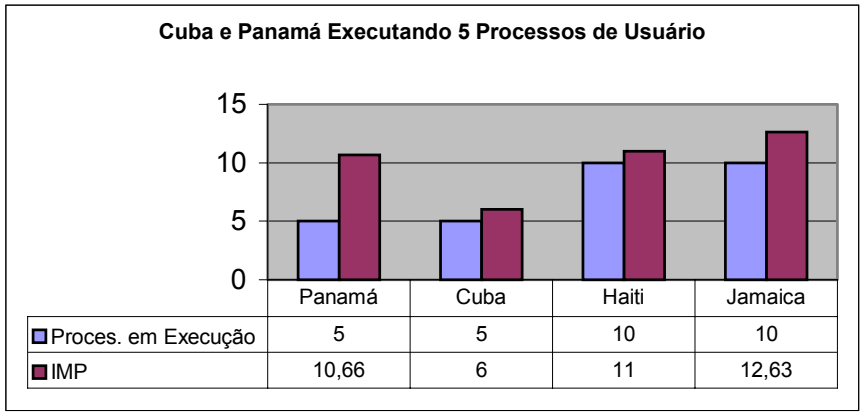

FIGURA 7.23 - Máquinas Cuba e Panamá Executando 5 Processos de Usuário

No quinto teste, deixou-se a máquina CUBA executando cinco processos de usuário enquanto que as restantes executavam dez processos de usuário. Novamente, o IMP – Índice da Média de Processos - dos nodos comportou-se como o esperado, ou seja, ele indicou que a máquina CUBA é a melhor para receber uma réplica porque está com a menor carga de trabalho (fig. 7.24). A lista de retorno ficou assim: CUBA em primeiro lugar, PANAMÁ em segundo lugar, JAMAICA em terceiro lugar e HAITI em quarto lugar.

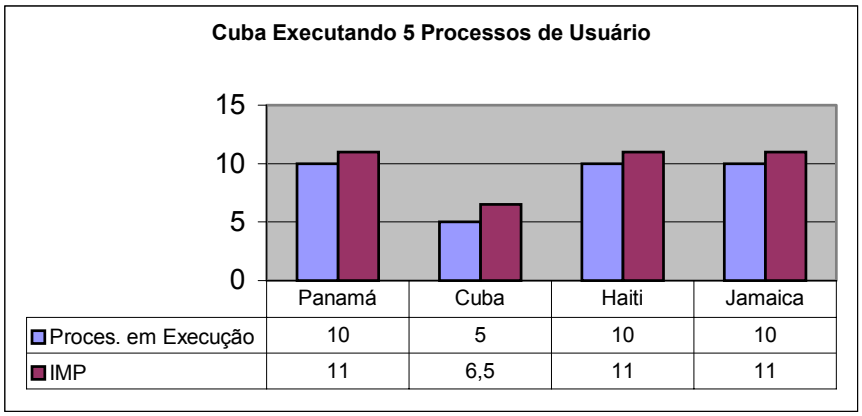

FIGURA 7.24 - Máquina Cuba Executando 5 Processos de Usuário

A máquina JAMAICA, no sexto teste (fig. 7.25), teve o seu número de processos de usuário reduzidos para cinco, juntamente com a máquina CUBA. O RPM fez a escolha certa, indicando, nesta ordem, a máquina JAMAICA, CUBA, PANAMÁ e HAITI.

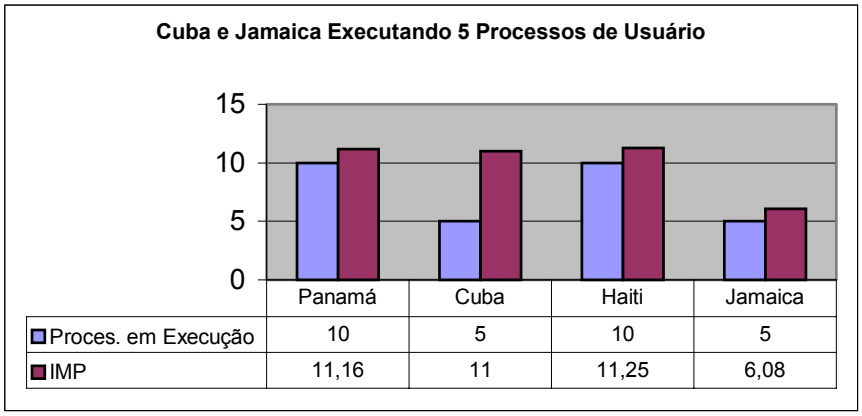

FIGURA 7.25 - Máquinas Cuba e Jamaica Executando 5 Processos de Usuário

No próximo teste (fig. 7.26), deixou-se a máquina HAITI rodando cinco processos e as demais rodando dez processos.A lista de resultados mostra, novamente, que o IMP está correto. Foram listadas, nesta ordem, as máquinas HAITI, JAMAICA, CUBA e PANAMÁ.

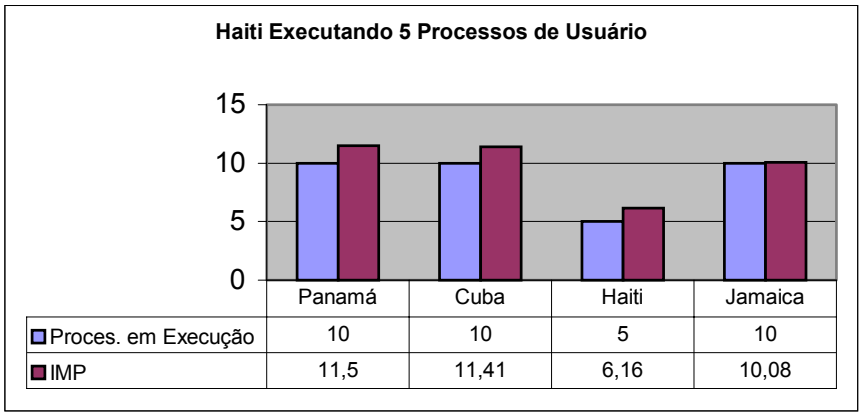

FIGURA 7.26 - Máquina Haiti Executando 5 Processos de Usuário

No oitavo teste, foram colocados cinco processos de usuário em execução nas máquinas HAITI e JAMAICA. Os resultados podem ser vistos na figura 7.27. A lista de resultados retornou a máquina JAMAICA em primeiro lugar, a máquina HAITI em segundo, a máquina CUBA em terceiro e a máquina PANAMÁ em quarto.

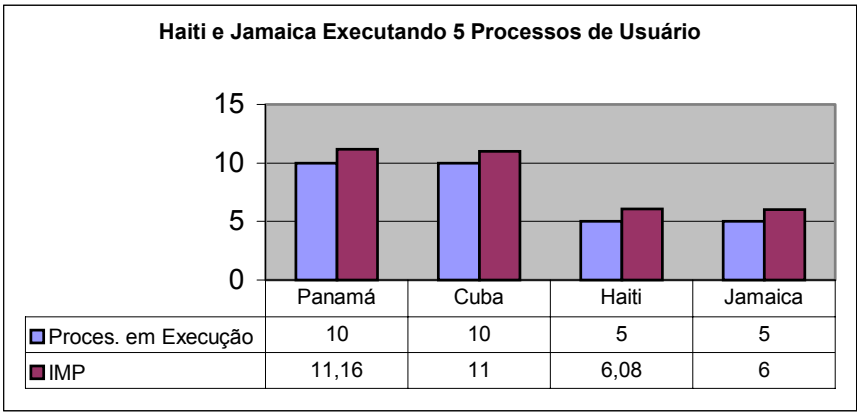

FIGURA 7.27 - Máquinas Haiti e Jamaica Executando 5 Processos de Usuário

Estes testes comprovam que o média do número de processos em execução no último minuto é um índice de carga consistente com os objetivos do RPM, ou seja, ele pode ser obtido de forma fácil e rápida e indica com segurança a atividade das máquinas que compõem o sistema.

8 Conclusões e Trabalhos Futuros

A principal contribuição deste trabalho é desenvolver uma estratégia para fornecer ao sistema de gerenciamento de réplicas uma lista de nodos potencialmente ideais para receber uma réplica de um objeto. Esta lista é criada pelo RPM tendo por base as informações estáticas (velocidade da CPU e memória) e dinâmicas (carga de trabalho) de cada um dos nodos. Com estas informações, o serviço de gerenciamento de réplicas pode criar ou migrar réplicas entre os nodos do sistema distribuído de uma forma mais precisa.

O posicionamento de uma réplica é baseado em informações obtidas dinamicamente nos nodos de um sistema distribuído. O índice que representa o estado de um nodo deve deve ser simples, de fácil obtenção e deve realmente representar o estado dos nodos. Estes objetivos foram alcançados pelo RPM. Os resultados dos testes mostram que, quando os nodos executam processos CPU intensivos, o índice de carga utilizado (média de processos em execução no último minuto) consegue indicar a atividade dos nodos.

Outra contribuição deste trabalho é o desenvolvimento do programa NPROC, responsável por coletar o número de processos em execução e calcular a média dos processos em execução nos últimos 60 segundos. Este programa grava esta informação no arquivo MEDIAS.INF, o qual pode ser acessado e lido por qualquer outro programa que necessite desta informação.

A implementação do RPM foi simplificada com o uso do sistema de comunicação de grupos Ensemble, responsável por disseminar, de forma confiável, as mensagens entre os nodos, necessárias para o fornecimento do serviço. Porém, um ponto negativo do Ensemble é a falta de documentação do sistema, que faz com que o programador inexperiente no uso da ferramenta enfrente grandes dificuldades iniciais.

A próxima etapa deste trabalho é integrá-lo com o sistema *Ape*, quando este estiver concluído. No estado atual do RPM, a ocorrência de falhas no nodo coordenador, enquanto este está aguardando as mensagens de resposta do nodo, faz com que o serviço seja interrompido. Quando integrado com o *Ape*, este deverá suspeitar da falha do coordenador, através de um mecanismo de *timeout* e, se necessário, submeter uma nova requisição para o novo coordenador.

Deve ser investigado se o mesmo índice se aplica a nodos executando processos E/S intensivos ou executando processos misturados – E/S intensivos e CPU intensivos. Deve também ser realizada uma investigação sobre as características do objeto que será posicionado em um nodo – se ele é E/S intensivo ou CPU intensivo – e quais as alterações que o seu posicionamento em um novo nodo ocasionará para o sistema distribuído.

Bibliografia

[AMI95] AMIR, Y**. Replication Using Group Communication Over a Partitioned Network.** Israel: Institute of Computer Science, The Hebrew University of Jerusalem, 1995. Doctor's Thesis.

[BIR93] BIRMAN, K. P. The Process Group Approach to Reliable Distributed Computing, **Communications of the ACM**, New York, v.36, n. 12, p. 37-53, Dec. 1993.

[BIR96a] BIRMAN, K. P. **Building Secure and Reliable Network Aplications.** Greenwich: Manning Publications Co., 1996. p. 249-298

[BIR96b] BIRMAN, K. P.; RENESSE, R. V.; MAFFEIS, S. HORUS: a Flexible Group Communication System. **Communications of the ACM**, New York, v. 39, n. 4, p. 76-83, Apr. 1996.

[BUD93] BUDHIRAJA, N. et al**.** The Primary-Backup Approach**.** In: MULLENDER**,** Sape (Ed.). **Distributed Systems.** 2nd ed. New York: ACM Press, 1993. p. 199-215

[CHR92] CHRISTIAN, F. Automatic Reconfiguration in the Presence of Failures In: IEEE INTERNATIONAL WORKSHOP ON CONFIGURABLE DISTRIBUTED SYSTEMS. **Proceedings**... [S.l.:s.n.], 1992. p. 4-17

[COL94] COLOURIS, G.; DOLLIMORE, J.; KINDBERG, T. **Distributed Systems – Concepts and Design**. 2nd ed. London: Addison-Wesley, 1994.

[DGS85] DAVIDSON, S. B.; GARCIA-MOLINA, H.; SKEEN, D. Consistency in Partitioned Networks**. Computing Surveys**, New York, v. 17, n. 3, p. 341-370, Sept. 1985.

[DOL96] DOLEV, D.; MALKI, D. The Transis Approach to High Availability Cluster Communication. **Communications of the ACM**, New York, v. 39, n. 4, p. 64-70, Apr. 1996.

[FDN2001] FERRARI, D. N. **Um Modelo de Replicação em Ambientes que Suportam Mobilidade.** Porto Alegre**:** PPGC da UFRGS, 2001**.** Dissertação de Mestrado.

[FER87] FERRARI, D.; ZHOU, S. **An Empirical Investigation of Load Indices for Load Balancing Applications.** In: INTERNATIONAL SYMPOSIUM ON COMPUTER PERFORMANCE MODELING, MEASUREMENT AND EVALUATION, 12., 1987. **Proceedings...** [Amsterdam]: North-Holland, [198-]. p. 515-528.

[FJC2000] FERREIRA, J. C. **Implementação de Objetos Replicados Usando Java.** Porto Alegre: PPGC da UFRGS, 2000. Dissertação de Mestrado.

[GAR82] GARCIA-MOLINA, H. Elections in a Distributed Computing System. **IEEE Transactions on Computers**, Los Alamitos, CA, v. C-31, n.1, p. 48-59, Jan. 1982.

[GEI2000] GEISS, L. C. **RAP:** um Experimento com Protocolos de Replicação Usando uma Ferramenta de Comunicação de Grupos. Porto Alegre: PPGC da UFRGS, 2000. Dissertação de Mestrado.

[GUE96] GUERRAOUI, R.; SCHIPER, A**.** Fault Tolerance by Replication in Distributed Systems. In: ADA-EUROPE INTERNATIONAL CONFERENCE ON RELIABLE SOFTWARE TECHNOLOGIES, 1996, Montreaux, Switzerland. **Proceedings...** Berlin: Springer-Verlag, 1996. p. 38-57. (Lecture Notes in Computer Science, v. 1088).

[GUE97] GUERRAOUI, R.; SCHIPER, A**.** Software-Based Replication for Fault Tolerance. **IEEE Computer**, Los Alamitos, CA, v. 30, n. 4, p. 68-74, Apr. 1997.

[HAC89] HAC, A. The Distributed Algorithm for Performance Improvement Through File Replication, File Migration and Process Migration. **IEEE Transactions on Software Engineering,** Los Alamitos, CA, v. 15, n.. 11, p. 1459-1470, Nov. 1989.

[HAY98] HAYDEN, M. G. **The Ensemble System.** The Dissertation Presented to the Faculty of the Graduate School of Cornell University, January 1998. Disponível em: http://www.cs.cornell.edu/Info/Projects/HORUS

[JAL94] JALOTE, P. **Fault Tolerance in Distributed Systems.** New Jersey: Prentice Hall, 1994.

[KOP90] KOPETZ, H. et al. Tolerant Transient Faults in MARS. In: INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, 20., 1990. **Digest of Papers.** Los Alamitos; IEEE, 1990. p. 466-473.

[LIT94] LITTLE, M.C.; McCUE, D.L**.** The Replica Management System: a Scheme for Flexible and Dynamic Replication. In: INTERNATIONAL WORKSHOP ON CONFIGURABLE DISTRIBUTED SYSTEMS, 2., 1994. **Proceedings...** Pittsburgh: Carnegie Mellon University, 1994.

[MOS94] MOSER, L. E.; AMIR, Y.; MELLIAR-SMITH, D. A. Extended Virtual Synchrony. In: IEEE INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SERVICES, 14., 1994. **Proceedings...** Los Alamitos, CA: IEEE Computer Society Press, 1994. p. 56-65.

[MOS96] MOSER, L. E. et al. Totem: a Fault-Tolerant Multicast Group Communication System. **Communications of the ACM**, New York, v.36, n. 12, p. 54-63, Dec. 1993.

[PAS98] PASIN, M. **Reintegração de Servidores em Sistemas Distribuídos.** Porto Alegre: CPGCC da UFRGS, 1998. Dissertação de Mestrado.

[ROD95] RODRIGUES, L. et al. A Transparent Light-Weight Group Service. In: IEEE SYMPOSIUM ON RELIABLE DISTRIBUTED SYSTEMS, 15., 1996. **Proceedings...** Los Alamitos, CA: IEEE Computer Society Press, 1996. p. 130-139.

[RUB96] RUBIN, R.; KISSIMOV, V. Object Migration in the Distributed Computing Environment. In: WORKSHOP ON MOBILITY AND REPLICATION, 2., 1996, Linz, Áustria. **Proceedings…**
Disponível em:
<http://www.diku.dk/distlab/workshops/wmr96/pgm.html>.
Acesso em: 16 nov. 2000.

[RUS98] RUSS, S. H. et al. The Hector Distributed Run-Time Environment. **IEEE Transactions on Parallel and Distributed Systems,** Los Alamitos, CA, v. 9, n. 11, Nov. 1998.

[RUS99] RUSS, S.H. et al. HECTOR: an Agent-Based Architecture for Dynamic Resource Management. **IEEE Concurrency**, Los Alamitos, CA, v. 7, n. 2, p. 47-55, Apr.-June 1999.

[SCH93] SCHNEIDER, F. B. Replication Management Using the State-Machine Approach. In: MULLENDER, Sape (Ed.). **Distributed Systems**. 2nd. ed. New York: ACM Press, 1993. p. 169-197.

[SHI95] SHIRAZI, B.A.; HURSON, A.R.; KAVI, K.M**. Scheduling and Load Balancing in Parallel and Distributed Systems.** Los Alamitos, CA IEEE Computer Society Press, 1995. 448 p.

[TAN92] TANENBAUM, A. **Distributed Operating Systems.** New Jersey: Prentice-Hall, 1992. p. 153-155.

[VAY98] VAYSBURD, A. **Building Reliable Interoperable Distributed Objects With The Maestro Tools.** The Dissertation Presented to the Faculty of the Graduate School of Cornell University, May 1998.
Disponível em: <http://www.cs.cornell.edu/Info/Projects/HORUS>
Acesso em: 10 dez. 1999.

[WOL97] WOLFFE, G.S.; HOSSEINI, S.H.; VAIRAVAN, K**.** An Experimental Study of Workload Indices for Non-dedicated, Heterogeneous Systems. In: INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED PROCESSING TECHNIQUES AND APPLICATIONS, 1997. **Proceedings...** [S.l: s.n.], 1997. p. 470-478
Disponível em <http://www.csis.gvsu.edu/~wolffe/publications.html>
Acesso em: 16 ago. 2000.