Serviços de Pertinência
para
Clusters de Alta Disponibilidade

Nélio Alves Pereira Filho

Dissertação apresentada ao
Instituto de Matemática e
Estatística da
Universidade de São Paulo
como parte dos requisitos
para obtenção do grau de
Mestre em Ciência da Computação

Área de Concentração: Ciência da Computação
Orientador: Prof. Dr. Arnaldo Mandel

— São Paulo, 20 de Agosto de 2004 —

Serviços de Pertinência

para

Clusters de Alta Disponibilidade

Este exemplar corresponde à redação final
da dissertação devidamente corrigida e
defendida por Nélio Alves Pereira Filho
e aprovada pela comissão julgadora.

São Paulo, agosto de 2004.

Banca examinadora:

- Prof. Dr. Arnaldo Mandel - IME-USP
- Profa. Dra. Taisy Weber - II-UFRGS
- Prof. Dr. Fábio Kon - IME-USP

À minha esposa Renata,
Aos meus pais Nélio e Maria Amália,
às minhas irmãs Maria Juliana, Mariana, Maria Luiza, Maria Clara.

Agradecimentos

Meus agradecimentos ao professor *Arnaldo Mandel*, pela paciência, interesse e dedicação em me orientar.

À minha esposa *Renata*, pela compreensão, motivação e atenção, que tantas vezes foram necessárias para a conclusão deste trabalho.

Aos meus pais, *Nélio* e *Maria Amália*, e às minhas irmãs, *Maria Juliana*, *Mariana*, *Maria Luiza* e *Maria Clara*, pela confiança e motivação.

E, claro, a *Deus*, que me concedeu força, disciplina e motivação para completar esta jornada, atingindo assim meu objetivo.

Resumo

Desde sua criação, o Linux trouxe muita atenção ao movimento open-source, e à concreta possibilidade de se usar soluções de baixo custo em missões críticas. Nos últimos anos, esta possibilidade tornou-se real com a criação de vários clusters de alta disponibilidade. Atualmente, existem pelo menos 10 soluções de clusters open-source e mais de 25 comerciais.

Cada um destes projetos teve uma abordagem diferente para o problema, embora todos tenham enfrentado dificuldades semelhantes. Se houvesse alguma padronização nesta área, esforços poderiam ter sido reaproveitados, e não duplicados. Neste contexto, o *Open Clustering Framework* (OCF) é um projeto open-source que visa definir um padrão para clusters em Linux.

Um dos serviços mais importantes em um cluster é o *serviço de pertinência*. Ele é responsável por criar e manter o grupo, sendo assim importante para inúmeras aplicações. Sistemas de alta disponibilidade baseiam-se no serviço de pertinência para garantir o funcionamento dos recursos oferecidos por um cluster.

Esta dissertação visa apresentar vários conceitos relativos a clusters, alta disponibilidade e serviços de pertinência. Com estes conceitos definidos, iremos implementar um serviço de pertinência, que será baseado no framework proposto pelo OCF, de maneira que esta implementação possa ser posteriormente incorporada a qualquer cluster que siga a especificação OCF.

Abstract

Since its creation, Linux has brought attention to the open-source movement, and to the concrete possibility of using low cost solutions in critical mission systems. In the last years, this possibility has become real due to the creation of several high availability clusters. Today there are at least 10 open-source solutions and more than 25 commercial ones.

Each one of these projects had a different approach to the problem, altought all of them had faced similar difficulties. If there was a standard in this area, efforts could be shared, and not duplicated. In this context, the *Open Clustering Framework* (OCF) is an open-source project that aims to define a cluster standard for Linux.

One of the more important services in a cluster is the *membership service*. It is responsible for creating and maintaing the group. For this reason, it is important for many applications: high availability systems rely on this service to garantee the execution of the resources provided by a cluster.

This thesis aims to present several concepts related to clusters, high availability and membership services. Having the concepts been defined, we will implement a membership service based on the OCF framework, in order to be able to incorporate this implementation in any OCF compliant cluster.

Índice

Lista de Figuras

Lista de Tabelas

Introdução

O sistema operacional Linux contribuiu muito para o movimento de software livre. Apesar da idéia de se fazer softwares livres ser antiga, ele conseguiu popularizar a tendência. Além de incentivar muitas pessoas a contribuírem com o movimento, ele ajudou a mudar a visão sobre sistemas de código aberto. A qualidade destes sistemas começou a ser valorizada, de maneira que logo eles passaram a ser usados em ambientes de produção, onde qualquer falha possivelmente acarreta em prejuízos financeiros. Entretanto, ainda faltava ao Linux uma característica fundamental para sistemas de produção: robustez. Para se tornar um sistema operacional robusto, o Linux precisava oferecer soluções de *alta disponibilidade*. Sistemas de Alta Disponibilidade provêem um aumento da disponibilidade de serviços através de técnicas computacionais, tanto através de recursos de software como de hardware.

Esta situação mudou quando se tornou mais evidente a percepção de que eram necessárias soluções de custo mais baixo dos que as existentes no mercado, e que o Linux poderia ser a solução para estes problemas. A um baixo custo era possível montar um *cluster* de Linux, ou seja, um grupo de máquinas rodando Linux, trabalhando juntas para aumentar a disponibilidade de seus serviços. Grupos de desenvolvimento começaram a se formar, e a montar projetos para pesquisar alternativas para o problema. Muitos grupos visavam criar sistemas open-source, mas também existiam as iniciativas comerciais. Desta maneira, os frutos dos projetos foram surgindo, e deles saíram muitos sistemas com capacidades de solucionar problemas de disponibilidade.

Estes projetos foram criados independentes somente por razões históricas, não por diferenças políticas ou filosóficas. Por causa disto, eles originalmente não compartilhavam código algum. Desta maneira, todos certamente tiveram que passar pelos mesmos problemas e dificuldades técnicas para poderem atingir seus objetivos. Ainda, tiveram que lidar com questões fora de seus escopos, mas das quais dependiam para poder criar o sistema. Percebe-se claramente que todos ganhariam muito se houvesse uma padronização de soluções para cluster, onde os esforços fossem unidos, e não repetidos.

Baseado nestas motivações, surgiu um novo projeto open-source, o *Open Clustering Framework*, que visa a criação de tal padrão, de maneira que aplicações de alta disponibilidade possam ser desenvolvidas sobre esta arquitetura. Ainda, ela seria totalmente baseado em componentes, de maneira que os sistemas já existentes possam ser adaptados e reaproveitados.

Este projeto de mestrado visa estudar clusters de alta disponibilidade, enfocando a abordagem realizada pelo OCF. Para tanto, realizamos um estudo profundo nos temas de clusters, tolerância a falhas e sistemas distribuídos, a fim de compreendermos os principais detalhes de tal formação. Ainda, de forma a contribuir com o projeto OCF, o objetivo do projeto foi implementar um serviço baseado em sua especificação que é essencial para clusters em geral: o *serviço de pertinência*. Este serviço é um protocolo distribuído que é responsável por criar e manter o cluster. É ele quem possibilita que máquinas regulares se unam em uma formação maior, e que esta organização exista durante o tempo. Em especial, sem ele não é possível construirmos clusters de alta disponibilidade, pois ele fornece informações necessárias para se implementar tolerância a falhas. Desta maneira, uma implementação do serviço de pertinência que segue a especificação OCF será muito útil para todos aqueles que precisam montar um cluster em geral.

Organização da Dissertação

No capítulo 1 descreveremos os principais conceitos relacionados a clusters. Veremos quais são as abstrações existentes em um grupo de computadores, e como é uma arquitetura geral que amarra estes componentes.

No capítulo 2 estudaremos em detalhes o assunto de alta disponibilidade. Muitas conceituações serão feitas, e veremos que a área de tolerância a falhas possui muitas nuâncias. Ao final deste capítulo, o leitor deverá ter formado um vocabulário básico da área, de maneira a poder entender as diferenças que existem em ambientes de alta disponibilidade.

No capítulo 3 abordaremos como os clusters podem ser usados para aumentar a disponibilidade de seus recursos. Antes de apresentarmos soluções para este problema, estudaremos alguns conceitos básicos relacionados a sistemas distribuídos que são fundamentais no projeto de um cluster de alta disponibilidade.

No capítulo 4 descreveremos o serviço de pertinência, que é um serviço distribuído essencial para a existência de clusters. Classificaremos o problema, e explicaremos todas as variações que ele pode apresentar. Por fim, faremos um estudo de protocolos existentes, compreendendo suas limitações e principais diferenças.

O capítulo 5 apresenta o *Open Clustering Framework*, que é um projeto open-source que visa reunir esforços para criar uma especificação padrão para clusters em Linux. Veremos o histórico de sistemas de cluster para Linux, e entenderemos qual é a proposta deste projeto.

No capítulo 6 descreveremos a implementação de um serviço de pertinência que foi realizado em conjunto com este estudo. O objetivo foi codificar um serviço seguindo a especificação OCF. Desta maneira, a implementação será um componente que poderá ser usado em qualquer cluster que siga esta especificação. Ao final, discutiremos os resultados obtidos. Devido à grande quantidade de dados coletados, no texto apresentaremos somente uma amostra dos resultados. Os resultados completos podem ser consultados em [74].

Por fim, o capítulo 7 apresenta os resultados gerais e futuras direções deste trabalho.

A versão final da dissertação foi impressa em preto e branco. Cópias coloridas podem ser obtidas em [74] e [1].

Clusters

cluster, n.:

A close group of usually similar things, often surrounding something.

– Cambridge International Dictionary of English.

1.1 Introdução

Para compreendermos bem o tema de alta disponibilidade, é necessário primeiramente entendermos o que são e como funcionam os clusters. Apesar de existirem outras técnicas de tolerância a falhas que não dependem de clusters (por exemplo, técnicas de tolerância a falhas por software), recorremos freqüentemente a eles devido a sua natureza distribuída, redundante e homogênea.

Este capítulo irá explicar os conceitos relacionados a clusters, a fim de darmos um breve esclarecimento sobre o assunto. Ainda, explicaremos abstrações e arquiteturas usadas nesta área a fim de criarmos uma terminologia própria para o assunto.

1.2 Conceitos Básicos e Definições

Em linhas gerais, um *cluster* ou *aglomerado* é uma coleção de computadores que trabalham juntos para criar um sistema muito mais poderoso [92]. Em outras palavras, um cluster é um conjunto de máquinas independentes, chamadas *nós*, que cooperam umas com as outras para atingir um determinado objetivo comum. Por serem fracamente agrupadas[1], para atingir este objetivo elas devem comunicar-se umas com as outras a fim de coordenar e organizar todas as ações a serem executadas. Ainda, freqüentemente elas precisam compartilhar algum hardware,

[1] Por *fracamente agrupadas*, entende-se que elas não fazem parte de uma mesma arquitetura de hardware, ou seja, não compartilham o mesmo barramento como dois processadores em uma única máquina compartilham. São máquinas regulares que foram acopladas para formar o cluster.

a fim de poder tolerar certas situações de falha. Assim, para um usuário externo, o cluster é visto como sendo um único sistema lógico. A figura 1.1 exibe um cluster de 4 nós, onde são compartilhados dispositivos de armazenamento e dispositivos de comunicação em rede.

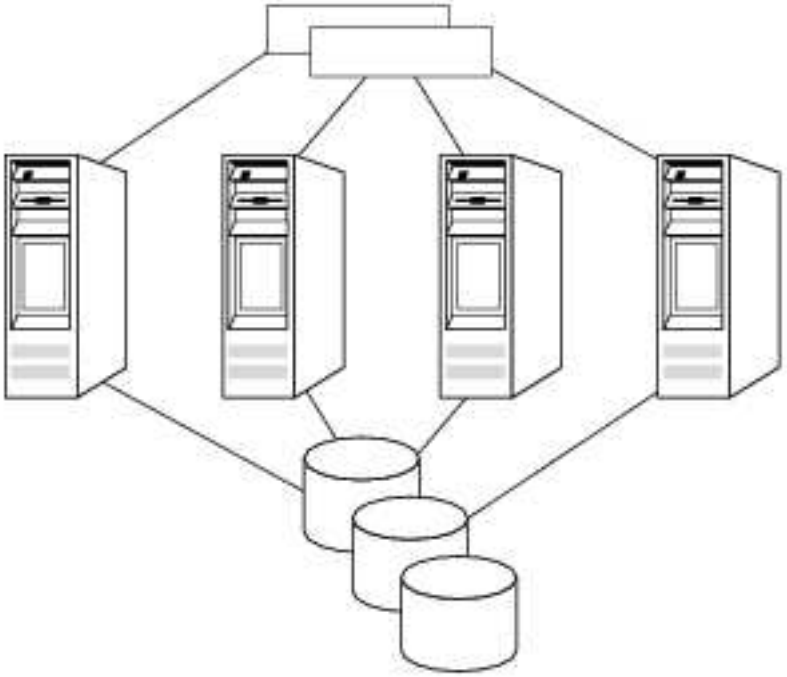

Figura 1.1: Cluster de 4 nós

Existem dois objetivos principais para a formação de clusters:

- *Alta Disponibilidade* (*High Availability* - HA), quando desejamos que o cluster forneça determinados serviços que devem estar sempre (ou quase sempre) disponíveis para receber solicitações. Este nível de disponibilidade do serviço é um fator dependente do cluster.
- *Alta Performance* (*High Performance Computing* - HPC), quando desejamos que o cluster execute determinadas tarefas, sendo que estas são divididas (na sua íntegra ou em frações de uma mesma tarefa) e processadas separadamente em vários nós, a fim de que a velocidade de processamento seja incrementada.[2]

É possível ainda termos uma situação onde o cluster deve atingir os dois objetivos juntos; às vezes, por razões de simplicidade, tal objetivo conjunto é atingido eliminando-se alguns rigores das definições acima.

[2]Este trabalho trata de clusters de alta disponibilidade, mas muitos dos conceitos explicados aqui se aplicam também a clusters de alta performance.

1.3 Princípios de um Cluster

Para ser útil, um cluster precisa seguir alguns princípios básicos [92]:

1. *Comodidade*: Os nós em um cluster devem ser máquinas normais interconectadas por uma rede genérica. O sistema operacional também deve ser padrão, sendo que o software de gerenciamento deve estar acima dele como uma aplicação qualquer.

2. *Escalabilidade*: Deve ser possível adicionar aplicações, nós, periféricos e interconexões de rede sem interromper a disponibilidade dos serviços do cluster.

3. *Transparência*: Apesar de ser constituído por um grupo de nós independentes fracamente agrupados, um cluster parece como um único sistema a clientes externos. Aplicações clientes interagem com o cluster como se ele fosse um único servidor com alta performance e/ou alta disponibilidade.

4. *Confiabilidade*: o cluster deve ter capacidade de detectar falhas internas ao grupo, assim como de tomar atitudes para que estas falhas não comprometam os serviços oferecidos.

5. *Gerenciamento e Manutenção*: Uma das principais dificuldades em se trabalhar com clusters é seu gerenciamento. A configuração e a manutenção de clusters são muitas vezes tarefas complexas e propensas a erros. Um fácil mecanismo de gerenciamento do ambiente deve existir a fim de que o cluster não seja um grande sistema complexo com um árduo trabalho de administração.

1.4 Abstrações em um Cluster

Um cluster possui vários elementos que, juntos com sua arquitetura, fornecem a funcionalidade desejada. Uma abstração destes elementos é necessária para podermos compreender qual o comportamento esperado de cada um deles [92].

1.4.1 Nó

Como visto anteriormente, o *nó* é a unidade básica do cluster; grupos de nós formam um cluster. Em um cluster, um nó comunica-se com os outros através de mensagens sobre conexões de rede, e falhas em nós podem ser detectadas através da ausência destas mensagens no canal de comunicação.

Um nó é um sistema computacional unicamente identificado, conectado a um ou mais computadores em uma rede. Assim, um nó tem quatro componentes principais:

- CPU: o componente de processamento principal de um computador, que lê e escreve na memória principal do computador.
- Memória: conhecido também como memória volátil. Entre outras coisas, este é o componente usado para executar programaticamente a *bufferização* de informações.
- Repositório de Armazenamento: um dispositivo que armazena informações. Geralmente um repositório persistente que deve ser acessado por transações de leitura/escrita para alterar seu conteúdo.
- Interconexão: este é o canal de comunicação entre os nós.

1.4.2 Recurso

Um *recurso* representa certa funcionalidade oferecida por um nó. Ele pode ser físico, como por exemplo uma impressora, ou lógico, como um endereço IP. Recursos são a unidade básica de gerenciamento de disponibilidade, e podem migrar de um nó a outro. Independentemente se for resultado de uma falha ou de intervenção humana, o processo de migração de um recurso de um nó para outro é chamado de *failover*.

Como um recurso representa uma funcionalidade, ele pode falhar. Por isso, monitores devem observar o estado dos recursos, a fim de tomar atitudes desejadas em caso de falha (por exemplo, inicializar o serviço em outro nó).

Um recurso tem associado a si um tipo, que descreve seus atributos genéricos e seu comportamento. Cada tipo de recurso deve ter associado a si mecanismos de inicialização/parada, a fim de que possam ser manipulados corretamente pelo gerenciador do cluster.

Dizemos que um nó p *hospeda* um recurso r, se, em um dado momento, o serviço associado a r está sendo fornecido a partir de p. Note que, para clientes externos ao cluster, isto deve ser transparente.

1.4.3 Dependências de Recursos

Freqüentemente, recursos dependem da disponibilidade de outros recursos. Por exemplo, um servidor HTTP geralmente depende da presença de uma interface de rede online e de um sistema de arquivos operacional para poder funcionar. Por outro lado, um sistema de arquivos depende de seu dispositivo de hardware. Estas dependências são descritas em um *grafo de dependências de recursos*. Este grafo de dependências descreve a seqüência na qual os recursos devem ser inicializados. Ainda, durante uma migração de recursos, ele descreve quais recursos devem ser migrados em conjunto. Uma representação gráfica do grafo de dependências do exemplo acima é apresentada na figura 1.2.

Figura 1.2: Dependência de recursos em um servidor HTTP

1.4.4 Grupos de Recursos

Um *grupo de recursos* é a unidade de migração [91]. Apesar de um grafo de dependências descrever os recursos que devem ser migrados juntos, podem haver considerações adicionais para agrupar recursos. O administrador de sistemas pode associar uma coleção de recursos independentes em um único grupo de recursos, a fim de garantir que todo o grupo seja migrado junto. Deste modo, as políticas de migração de recursos devem ser baseadas em grupos de recursos. Ainda, as dependências de recursos vistas acima não podem ultrapassar as barreiras de um grupo, ou seja, recursos de um grupo devem depender somente de recursos contidos em seu próprio grupo.

1.5 Arquitetura de um Cluster

Isoladamente, os elementos descritos acima não provêem a funcionalidade desejada do cluster. É necessário que haja uma arquitetura que una todos os elementos e forneça o funcionamento que cada um espera do outro, ou seja, que defina como estes componentes conversam entre si, além de definir uma série de serviços internos ao cluster necessários para sua operação.

Como cada cluster pode ter um objetivo diferente, sua arquitetura pode variar bastante. Entretanto, existem muitos componentes comuns, e que podem ser estudados para compreendermos uma arquitetura básica.

Atualmente não existe uma padronização de arquitetura para clusters[3], mas muitas propostas assemelham-se em vários sentidos. Segundo Stephen Tweedie [89], tal arquitetura seria formada pelas seguintes camadas:

- Camada de Comunicação (*Channel Layer*): trata das comunicações ponto-a-ponto entre os nós.
- Camada de Ligação (*Link Layer*): agrupa canais de comunicação em uma única ligação entre dois nós.
- Camada de Integração (*Integration Layer*): forma a cluster propriamente dito, ou seja, controla a entrada e saída de nós do grupo.
- Camada de Recuperação (*Recovery Layer*): executa a recuperação (*failover*) e a inicialização/parada controlada de serviços depois de uma alteração na formação do cluster.

Existem ainda quatro serviços chaves para um cluster:

- Serviço de Quórum (*Quorum Layer*): em caso de uma divisão do cluster, determina qual parte possui autorização para prosseguir com sua execução.
- Serviço de Informações (*JDB*): armazena persistentemente estados internos ao cluster. De modo geral, nada mais é do que um repositório de informações local a cada nó do cluster.
- Serviço de Barreiras (*Barrier Services*): provê um serviço de sincronização global ao cluster.
- Serviço de Nomes (*Namespace Service*): provê um serviço de nomes global ao cluster.

1.5.1 Camada de Comunicação

Esta é a camada de comunicação de baixo nível. Esta camada mantém múltiplas interfaces (que podem ser IP, serial ou SCSI, por exemplo). A descoberta de vizinhança é feita em cada interface, a qual é totalmente independente uma da outra. Após uma conexão ponto-a-ponto permanente ser estabelecida, vários tipos de mensagens são permitidos: entrega seqüencial de informações, verificação do estado do link (*heartbeat*), e reinicialização controlada. Ainda, cada canal pode ter uma métrica que determina sua qualidade para transportar o tráfico do cluster.

[3]Uma proposta de padronização é descrita no capítulo 5.

1.5.2 Camada de Ligação

Construída sobre a camada de comunicação, a camada de ligação estabelece um mecanismo de ligação de alto nível, o qual associa todos os canais para um dado nó em um único link. Este é um link "virtual", pois a informação ainda trafega pelos meios de comunicação estabelecidos na camada de comunicação. Dadas as métricas dos canais de comunicação, o link pode ter vários estados, os quais determinam sua possibilidade de operação.

A grande utilidade desta camada é abstrair a comunicação de dados e suas particularidades entre dois nós, a fim de tornar a tarefa de trocar mensagens mais simples para as camadas superiores.

1.5.3 Camada de Integração

Durante a operação de um cluster, este pode passar por várias transições em sua formação. Neste contexto, uma transição nada mais é do que a adição/remoção de um nó do grupo, ou seja, uma alteração na lista de membros do cluster. Sempre que esta lista é alterada, uma transição atômica e transacional ocorre a fim de que todos os nós membros tomem conhecimento desta modificação.

Deste modo, esta camada realiza transições na formação do cluster, aglomerando clusters vizinhos, despejando membros inativos ou com mau comportamento, e garantindo alterações transacionais na formação do grupo.

1.5.4 Camada de Recuperação

Esta é a camada que integra o núcleo do cluster com os serviços que ele fornece. A principal tarefa da camada de recuperação é "recuperar-se" de transições completadas pela camada de integração. Esta recuperação é constituída de uma série de operações:

Recuperação interna de todos os serviços permanentes registrados.

Em linhas gerais, envolve a reconstrução do estado global (isto é, das variáveis globais e de seus valores), baseado no conjunto de novos nós no cluster, e o estado deste serviço nos novos nós. Por exemplo, um gerenciador de lock distribuído poderia recalcular a função de hash usada para distribuir os diretórios de lock entre os nós disponíveis, e então reconstruir seu banco de dados baseado no conjunto de locks mantidos em cada nó.

Recuperação de serviços do usuário.

Estes serviços não estão necessariamente rodando todo o tempo, e podem não possuir estados globais (isto é, recursos/variáveis dos quais o gerenciador do cluster dependa).

Devemos possibilitar que serviços de usuários que desapareceram em uma transição sejam reinicializados em outro nó, sempre respeitando a ordem estabelecida em seu grafo de dependências. Este*failover* de serviços de um nó (que desapareceu do cluster) para outro nó é muito usado em clusters para garantir a disponibilidade de serviços.

Ainda, esta camada deve ser capaz de desabilitar acesso a certos serviços do cluster até que a recuperação esteja completa.

1.5.5 Serviço de Quórum

Em qualquer momento, um cluster pode ser particionado em um ou mais sub-clusters. Estes particionamentos podem ser gerados por vários motivos, como falhas em nós ou canais de comunicação. Ainda, cada nó possui um visão da lista de membros deste sub-cluster, acreditando que esta é a lista correta de membros do cluster. Em uma situação deste tipo, não desejamos que mais de uma partição acredite ser o cluster ativo e comece a fornecer concorrentemente os mesmos serviços (este problema é conhecido como *síndrome do cérebro dividido*, ou *split brain syndrome* [33]).

O Serviço de Quórum é responsável por determinar que somente uma destas partições seja considerada o novo cluster, isto é, que somente um sub-cluster "possua quórum". Desta forma, as partições que não possuírem quórum irão parar seu processamento assim que perceberem que a lista de membros não é válida. Freqüentemente, tal decisão é feita de duas maneiras:

- Votação: através de uma eleição entre os membros do sub-cluster podemos concluir se este grupo possui ou não a maioria dos membros originais do cluster.

- Recurso de Quórum: o grupo que possuir um certo recurso será considerado a partição ativa (isto é, partição com quórum), independente da quantidade de nós membros. Este método é útil em situações em que o necessário para poder prosseguir com o processamento é o acesso a uma rede ou a um repositório de dados externo, por exemplo.

1.5.6 Serviço de Informações

Um banco de dados transacional é necessário para armazenar estados locais. Em um cluster onde os nós podem votar para tomar certas decisões, todo nó com um voto válido (ou seja, um membro apto a votar) deve possuir uma área de armazenamento persistente, a fim de que possam ser executadas alterações confiáveis em suas informações de estado e configuração.

1.5.7 Serviço de Barreiras

O Serviço de Barreiras provê um mecanismo básico de sincronização para qualquer grupo de processos no cluster. Uma operação de barreira determina que todos os processos que cooperam entre si devem esperar pela mesma barreira. Somente quando todos estes processos atingirem esta barreira é que eles terão permissão para prosseguir sua execução. Desta maneira, este serviço é o núcleo para sincronizar transições no cluster.

1.5.8 Serviço de Nomes

O espaço de nomes do cluster é uma estrutura hierárquica não-persistente, na qual qualquer nó pode registrar nomes. Através dela, processos podem tanto registrar e consultar nomes, como definir dependências baseadas nestes nomes.

1.6 Clusters Hierárquicos

Quando lidamos com clusters muito grandes, certamente a freqüência de transições tende a aumentar. Digamos que estamos trabalhando com um cluster de 10000 nós. Nesta situação, não queremos que, devido à saída de um nó do grupo, todo o cluster passe por um processo de transição que possa parar todo seu processamento. Desejamos restringir esta transição a somente aqueles nós que se relacionavam diretamente com o nó que acabou de sair. Esta será a idéia por trás de um *cluster hierárquico* [89], que é um tipo de cluster mais complexo, que envolve os conceitos discutidos anteriormente.

Um cluster *plano* é uma entidade virtual nascida da colaboração entre nós em uma rede de comunicação. Podemos chamar este cluster de "cluster de *primeiro nível*"; estes nós são seus *membros*, e juntos formam sua *lista de membros*. Em cada momento, um cluster possui um único nó privilegiado chamado de *líder do cluster*, que possui um papel de coordenador em suas transições.

Quando desejamos combinar clusters em unidades maiores, é necessário tolerar a associação de clusters em um único "cluster *de alto nível*", ou *metacluster*. Ele é formado aglomerando-se os líderes dos *clusters membros* em um novo cluster; estes líderes formam a lista de membros do metacluster. Assim, há dois tipos de transições:

- Transições em um sub-cluster, que não alteram o líder do grupo. Este tipo de transição não afeta a lista de membros do metacluster, e, portanto, não causa mais nenhuma transição no cluster hierárquico.

- Transições em um sub-cluster, que alteram o líder do grupo. Este tipo de transição altera a lista de membros do metacluster, e, portanto, dispara uma transição no cluster hierárquico.

Assim, um cluster hierárquico nada mais é do que um "cluster de clusters". Esta associação entre clusters pode se estender por muitos outros níveis, criando clusters de segundo, terceiro, ou até mesmo décimo nível.

A grande vantagem de se criar clusters hierárquicos é o ganho expressivo de escalabilidade. Por exemplo, em uma grande rede de uma corporação administrada como um cluster, não desejamos que falhas em máquinas usadas para teste causem transições; queremos restringir as transições às áreas de um cluster onde elas façam sentido. Em um metacluster, transições dentro de um sub-cluster que não afetam seu líder não causam nenhuma modificação na formação do metacluster, implicando em uma melhor performance. O metacluster sofrerá uma transição somente se algum de seus membros (que são os líderes dos sub-clusters) for modificado.

Alta Disponibilidade

"Immortality is an adequate definition of high availability for me."

Gregory F. Pfister [75]

High Availability, n.:

Patching up complex systems with even more complexity.

– (anônimo)

2.1 Introdução

Com as redes e hardwares tornando-se cada vez mais rápidos a custos atrativos, dependemos cada vez mais de sistemas computacionais para realizar tarefas críticas, cujas falhas acarretariam em prejuízos materiais, financeiros, ou até mesmo em perda de vidas humanas. Como falhas são inevitáveis, utilizamos várias técnicas para garantir a disponibilidade destes serviços, mesmo em caso de erros. Estas técnicas podem ser tanto no nível do software como do hardware: através de hardwares redundantes tolerantes a falhas, a falha de um componente é compensada pela utilização de outro. Devido ao alto custo de tal solução, clusters são montados e configurados de modo a atingir um comportamento similar: a falha de um nó é compensada pela migração dos recursos comprometidos para outro nó operante.

Este capítulo visa explicar os detalhes de *alta disponibilidade*, tais como sua finalidade, variações e possibilidades. Compreender uma área tão complexa quanto tolerância a falhas em sistemas computacionais distribuídos requer identificar seus conceitos fundamentais e nomeá-los de maneira muito precisa. Entretanto, conforme Cristian [38] atesta, há uma falta de estruturação clara destes conceitos. É muito comum encontrarmos termos diferentes sendo usados para o mesmo conceito, ou um mesmo nome sendo usado para conceitos diferentes. Desta maneira, este capítulo visa também definir os principais conceitos básicos relacionados com a área, de maneira a não deixar ambigüidades.

2.2 Tolerância a Falhas e Alta Disponibilidade

"O termo *Tolerância a Falhas* foi cunhado por Avizienis em 1967 [20]. Desde então, tem sido usado pela comunidade acadêmica para designar toda a área de pesquisa ocupada com o comportamento de sistemas computacionais sujeitos a ocorrência de falhas" [94]. Entretanto, este termo nunca se tornou popular, e outros foram usados em seu lugar.

Sistemas Redundantes e *Alta Disponibilidade* são usados na indústria e por desenvolvedores para denotar sistemas que toleram a ocorrência de falhas. Aos poucos, o termo *dependabilidade*[1] (*dependability*) também vem sendo usado neste sentido, substituindo o termo *tolerância a falhas.*[2]

Neste texto, usaremos os termos *tolerância a falhas* e *alta disponibilidade* intercambiavelmente, pois são os mais consolidados e conhecidos, tanto na área acadêmica como na indústria.

2.3 Conceitos Básicos e Definições

Segundo Jalote [59], um sistema é *tolerante a falhas* se ele pode mascarar a presença de falhas no sistema. Ainda, conforme Gartner [55] demonstra, não é possível tolerar falhas sem redundâncias. Redundância em um sistema pode ser hardware, software ou tempo. *Redundância de hardware* compreende os componentes de hardware adicionados para tolerar falhas. *Redundância de software* inclui todos os programas e instruções que são usadas na tolerância a falhas. Uma técnica comum de tolerância a falhas é executar certa instrução várias vezes. Esta técnica necessita *redundância de tempo*, isto é, tempo extra para executar tarefas para tolerar falhas.

Devemos compreender que a redundância de recursos é um pré-requisito para se atingir um alto grau de disponibilidade, pois ela é necessária para se eliminar os *pontos únicos de falha* (*single point of failures*- SPOF) .

É importante lembrarmos que os SPOF não se restringem somente aos recursos do hardware das máquinas, existindo também nas:

- Redes de Computadores (switches, routers, hubs, cabeamento, etc) que disponibilizam os serviços ao mundo externo.
- Redes Elétricas que alimentam as máquinas. Ainda, nada adianta utilizar mais de uma rede elétrica para alimentar as máquinas se elas saem na rua em um mesmo poste, cuja destruição poderia comprometer todos os serviços.

[1] Apesar do termo "dependabilidade" não fazer parte da língua portuguesa, esta palavra será utilizada devido à falta de outra mais significativa neste contexto.

[2] Em 2000, o Fault Tolerant Computing Symposium, FTCS, foi renomeado para Dependable Systems and Networks.

- Localizações dos Recursos. Deve-se lembrar que desastres (terremotos, furacões, ataques terroristas, etc) podem inviabilizar toda a operação de uma corporação, se esta tiver suas informações e sistemas operando em somente um lugar físico.

Conforme Marcus e Stern [68] descrevem, muitos projetos de alta disponibilidade não são bem elaborados, deixando sem contingência os recursos quando ocorre alguma falha nos ítens acima. Deve estar claro que o nível de proteção contra falhas deve ser estudado, pois enquanto houverem SPOF como os acima enumerados, o sistema ainda estará sujeito a indisponibilidades.

Assim, fica claro que quando nos referirmos a um *sistema de alta disponibilidade*, este deve englobar muito mais do que um simples sistema de redundância de recursos. É claro que tudo isto depende do grau de disponibilidade que desejamos atingir; por exemplo, raramente planejamos um sistema visando uma disponibilidade mesmo no caso de ataques terroristas. Também, é fácil percebermos que quanto maior o grau de disponibilidade que desejamos garantir, maiores serão os gastos com esta solução. A figura 2.1 [68] exibe um gráfico que detalha o aumento de investimento necessário para se atingir certos níveis comuns de disponibilidade. Estes níveis são:

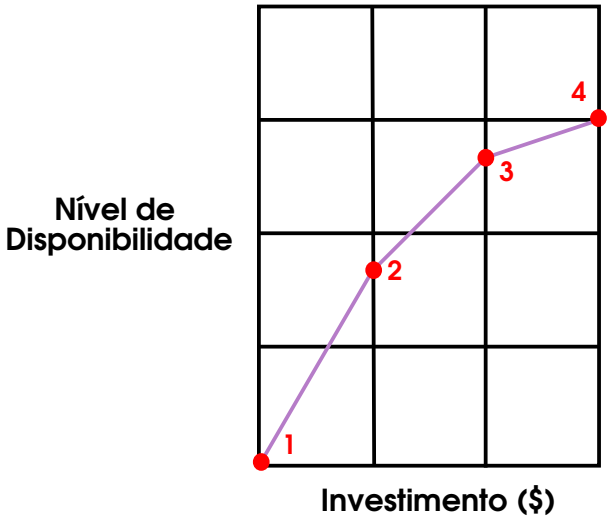

Figura 2.1: Custos da Disponibilidade

1. Sistemas Básicos: Tais sistemas não utilizam medidas especiais para se proteger de falhas. Cópias de segurança das informações existem, mas é necessária intervenção humana para corrigir o sistema.

2. Informação Redundante: algum nível de redundância da informação é utilizada, como redundância de discos.

3. Failover do Sistema: um modelo onde existe um sistema em stand-by, pronto para assumir caso o sistema principal falhe.

4. Recuperação de Desastres: além dos sistemas no local principal, existe um segundo conjunto de sistemas em um local secundário.

2.4 Medição de Disponibilidade

Outro detalhe a ser abordado sobre alta disponibilidade é como medi-la. Em termos técnicos, a disponibilidade de certo serviço é a probabilidade de encontrá-lo operando normalmente em determinado momento. Portanto, tal probabilidade leva em conta qual o provável *uptime* (tempo em que os serviços estarão funcionando) e o provável *downtime* (tempo em que os serviços ficarão fora do ar).

Uma outra notação muito usada atualmente é a que leva em conta o "número de noves de disponibilidade"[3]. Nesta notação, dizer que uma solução oferece "5 noves" de disponibilidade significa dizer que seu uptime é 99.999%. A tabela 2.1 exibe alguns uptimes, e mostra como muitas vezes não necessitamos de muita disponibilidade para certos serviços. É claro, também, que não desejamos que o downtime ocorra em momentos inoportunos, por menores que sejam.

Baseado nesta tabela, enumeramos na tabela 2.2 alguns exemplos de sistemas que podem necessitar dos níveis de disponibilidade listados na tabela 2.1.

Dadas estas definições, podemos calcular a disponibilidade através da equação 2.1 [68]

$$A = \frac{MTBF}{MTBF + MTTR} \tag{2.1}$$

onde A é o grau de disponibilidade expresso como uma porcentagem, $MTBF$ é o *tempo médio entre falhas* (*mean time between failures*), e $MTTR$ é o *tempo máximo para reparos* (*maximum time to repair*).

2.5 Dependabilidade, Confiabilidade, Disponibilidade e Segurança

Dependabilidade, Confiabilidade, Disponibilidade e Segurança são termos que muitas vezes possuem significados pouco claros e distintos. Por isso, uma cuidadosa definição deve ser feita para compreendermos o escopo de cada um.

[3]Tal notação é mais voltada a sistemas de alta disponibilidade de mercado, e não a estudos acadêmicos.

	UPTIME	DOWNTIME	DOWNTIME POR ANO	DOWNTIME POR SEMANA
1	90%	10%	36.5 dias	16 horas, 51 minutos
2	98%	2%	7.3 dias	3 horas, 22 minutos
3	99%	1%	3.65 dias	1 hora, 41 minutos
4	99.8%	0.2%	17 horas, 30 minutos	20 minutos, 10 segundos
5	99.9%	0.1%	8 horas, 45 minutos	10 minutos, 5 segundos
6	99.99%	0.01%	52.5 minutos	1 minuto
7	99.999%	0.001%	5.25 minutos	6 segundos
8	99.9999%	0.0001%	31.5 segundos	0.6 segundos

Tabela 2.1: Medindo a Disponibilidade

1	computadores pessoais, sistemas experimentais
3	sistemas de acesso
5	provedores de Internet
6	CPD, sistemas de negócios
7	sistemas de telefonia; sistemas de saúde; sistemas bancários
8	sistemas de defesa militar

Tabela 2.2: Exemplos de sistemas e suas disponibilidades necessárias

2.5.1 Dependabilidade

A *dependabilidade* (*dependability*) de um sistema computacional representa sua habilidade de entregar um serviço de maneira confiável [22]. O serviço entregue por um sistema é seu comportamento, conforme percebido pelo(s) seu(s) usuário(s); um usuário é um sistema físico ou um ser humano que interage com o sistema através de uma interface. A função de um sistema é o que ele é esperado realizar, conforme descrito em sua especificação.

Em outras palavras, o termo dependabilidade mede a qualidade do serviço fornecido por um dado sistema e a confiança depositada no serviço fornecido [94]. De acordo com a aplicação, diferentes atributos podem ser observados da dependabilidade. Apesar de diferentes, estes aspectos são complementares, e podem ser compreendidos como *pontos de vista* da dependabilidade [66]:

- a prontidão de uso leva à *disponibilidade*.
- a continuidade de serviço leva à *confiabilidade*.
- a não ocorrência de conseqüências catastróficas ao ambiente e ao usuário leva à *segurança*.

- a não ocorrência de revelações desautorizadas de informações leva à *confidencialidade*.
- a não ocorrência de alterações impróprias das informações leva à *integridade*.
- a possibilidade de submeter o sistema a reparos e evoluções leva à *sustentabilidade*.

Desta maneira, dependabilidade é um termo mais completo do que disponibilidade e confiabilidade[4]. Para esclarecermos os termos mais próximos deste texto (confiabilidade, disponibilidade e segurança), iremos melhor defini-los abaixo.

2.5.2 Confiabilidade

A *confiabilidade* (*reliability*) de um sistema é uma função $R(t)$, que representa a probabilidade do sistema operar conforme o especificado até o instante t [59]. Confiabilidade é a medida mais usada em sistemas críticos, ou seja nos seguintes tipos de sistemas:

- sistemas em que mesmo curtos períodos de operação incorreta são inaceitáveis.
- sistemas em que reparos são impossíveis.

Exemplos de sistemas *confiáveis* são sistemas de aviação e de exploração espacial.

2.5.3 Disponibilidade

A *disponibilidade* (*availability*) de um sistema é uma função $A(t)$, que representa a probabilidade do sistema operar corretamente no instante t [59]. Disponibilidade é o atributo mais usado e desejado em diversos tipos de sistemas, como sistemas de consulta a bases de dados on-line e servidores de rede.

Assim, percebemos que disponibilidade não pode ser confundida com confiabilidade: um sistema pode ser altamente disponível mesmo apresentando períodos de inoperabilidade, quando está sendo reparado, desde que esses períodos sejam curtos e não comprometam a qualidade do serviço. Disponibilidade está muito relacionada com o tempo de reparo do sistema, pois a diminuição do tempo de reparo resulta em um aumento de disponibilidade. A figura 2.2 [94] exibe o comportamento de um sistema durante o tempo. Através dela, podemos percebermos a relação da disponibilidade com o downtime do sistema, assim como a relação de confiabilidade com seu uptime.

[4]Mesmo sendo termos com significados distintos, é comum os encontrarmos usados de maneira equivalente.

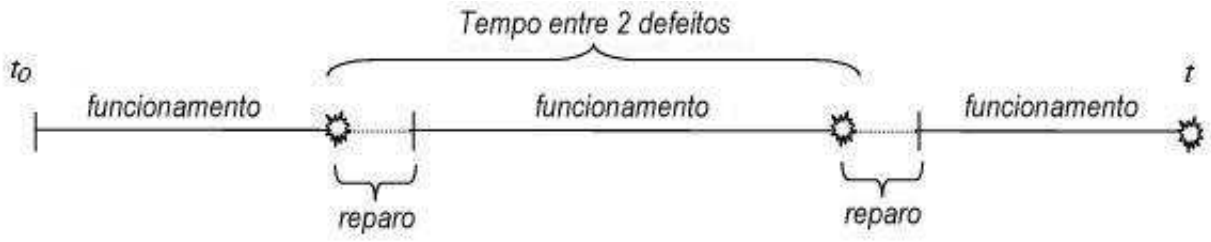

Figura 2.2: Alternância de períodos de funcionamento e de reparo

2.5.4 Segurança

Segurança (*safety*) é a probabilidade do sistema estar em uma das seguintes condições [94]:

- estar operacional e executar sua função corretamente;
- descontinuar suas funções de forma a não provocar danos a outros sistemas que dele dependam.

A segurança de um sistema representa sua capacidade de se comportar sem falhas (*failsafe*). Em um sistema failsafe, ou sua saída é correta ou o sistema é levado a um estado seguro. Um exemplo de um sistema failsafe é um sistema de transporte ferroviário, onde os controles de um trem providenciam sua desaceleração e parada automática quando não mais conseguirem garantir o seu funcionamento correto.

A tabela 2.3 ilustra os três termos vistos acima. Nela, para cada situação descrita, identificamos quais propriedades são asseguradas. Uma interrogação indica que, baseado na descrição da situação, nada pode ser afirmado.

2.6 Taxonomia de Alta Disponibilidade

Muitos termos distintos são encontrados na literatura para se referir ao conceito de alta disponibilidade. Apesar de diferentes, muitos possuem o mesmo significado, enquanto que outros são usados para denotar diferentes aspectos da disponibilidade. Nesta seção apresentaremos tais termos, a fim de formarmos um vocabulário padrão da área. Tais definições foram baseadas em [77], que também fornece uma lista completa de siglas e abreviações, juntamente com suas definições.

Situação	Confiabilidade	Disponibilidade	Segurança
Um sistema altamente confiável, onde a falha do banco de dados que ele depende faz com que todas suas operações sejam finalizadas com erro.	√	√	–
Um sistema com períodos de inoperabilidade, mas que está sempre disponível nos momentos críticos.	–	√	?
Um sistema que permanece operacional durante o dia todo, mas sofre uma parada à noite, justamente no momento de maior acesso pelos seus usuários.	√	–	?

Tabela 2.3: Exemplos de Confiabilidade, Disponibilidade e Segurança

2.6.1 Disponibilidade Básica (*Basic Availability* - BA)

Um sistema possui *Disponibilidade Básica* se foi desenhado, implementado e implantado com um número suficiente de componentes (hardware, software e procedimentos) para satisfazer sua funcionalidade, e nada além disso. Tal sistema fornecerá os serviços corretamente, desde que falhas e operações de manutenção não sejam executadas.

2.6.2 Alta Disponibilidade (*High Availability* - HA)

Um sistema possui *Alta Disponibilidade* se foi desenhado, implementado e implantado com um número suficiente de componentes (hardware, software e procedimentos) para satisfazer sua funcionalidade, além de possuir redundância suficiente nos componentes para mascarar algumas falhas definidas. Esta é uma definição ambígua, que permite classificarmos uma série de configurações como "Alta Disponibilidade". A ambigüidade está nos termos sublinhados ("suficiente", "mascarar" e "algumas"), que esclareceremos abaixo.

Mascarar uma falha significa impedir sua visualização por um observador externo. Esta abordagem é similar à máxima filosófica "se uma árvore cai e ninguém está perto para escutar, ela não produz nenhum som". A idéia não é impedir a ocorrência de falhas, e sim a sua observação.

Tal objetivo é alcançado através de redundância: quando um certo componente falha, deve existir outro igual para substituí-lo. O nível de transparência neste processo pode levar a

grandes variações de sistemas caracterizados como de "Alta Disponibilidade":

Mascaramento Manual (*Manual Masking* - MM)

Após uma falha, alguma intervenção manual é necessária para colocar o componente redundante em funcionamento. Enquanto isto não acontece, o sistema está indisponível.[5]

Cold Stand-By (CS)

Após uma falha, usuários do componente afetado são desconectados e perdem todo trabalho em progresso (isto é, ocorre um rollback para o último estado consistente de seu trabalho). Um failover automático do serviço ocorre. Porém, como o componente redundante estava desativado, este precisa ser inicializado para poder começar a operar. Um exemplo é um cluster de duas máquinas, onde a stand-by permanece desligada: quando ocorre a falha de algum recurso, ela é ligada e inicializada para poder oferecer os serviços.

Warm Stand-By (WS)

Assim como no Cold Stand-By, os usuários são desconectados e perdem o trabalho em andamento. Um failover automático ocorre, só que desta vez o componente redundante já está inicializado e sendo executado. Ainda, o componente em stand-by pode compartilhar algum estado com o componente que apresentou falha, possibilitando então a recuperação (parcial ou não) do trabalho que estava em progresso. O tempo de detecção da falha é o mesmo no caso anterior, mas o tempo de *takeover*[6] é drasticamente reduzido.

Hot Stand-By (HS) / Replicação Ativa (RA)

Os componentes redundantes e ativos estão fortemente agrupados e (logicamente) indistinguíveis aos usuários. O estado do processamento é ativa e completamente compartilhado entre os componentes do grupo. Após uma falha, os usuários do componente defeituoso não são desconectados e não observam erro algum, pois o trabalho em progresso continua sua execução com os componentes restantes. Assim, neste modelo a transparência é completa e a recuperação torna-se instantânea, já que os usuários não notam nenhuma parada no funcionamento do sistema.

[5] Usualmente, sistemas de alta disponibilidade implicam em uma recuperação automática dos serviços. Isto é obtido com uma monitoração dos recursos e um failover automático dos serviços afetados.

[6] *Takeover* é o processo que o componente redundante executa para assumir e responder pelos recursos que apresentaram falha.

Suficiente reflete a necessidade do sistema por alta disponibilidade. Por exemplo, em certo sistema poderia ser necessário mascarar somente falhas de hardware e não de software, e isto seria o *suficiente*. Ainda, "suficiente" também reflete quantas vezes será possível mascarar a falha. Por exemplo, em um sistema onde cada componente possui n réplicas, é possível mascarar n falhas simultâneas, isto é, falhas que ocorrem durante o período de recuperação de um mesmo componente.

Algumas, como usado acima, é um atestado de que não é possível mascarar todas as falhas. Assim, deve ser especificado o conjunto de falhas que serão compensadas por um sistema de alta disponibilidade.

2.6.3 Disponibilidade Contínua (*Continuous Availability* - CA)

Disponibilidade Contínua estende a definição apresentada de Alta Disponibilidade: como apresentado até então, sistemas de alta disponibilidade só mascaram falhas, isto é, paradas não planejadas dos serviços; através de CA deseja-se também mascarar paradas planejadas dos serviços. Ainda, este modelo de disponibilidade deve ser exclusivamente HS/RA, para qualquer tipo de parada dos serviços.

2.6.4 Domínios de Disponibilidade

Como visto anteriormente, um sistema de alta disponibilidade possui muitos componentes distintos (hardware, software, rede elétrica, procedimentos, etc). Devido a esta diversidade de componentes, freqüentemente não desejamos o mesmo nível de disponibilidade em cada um. Assim, necessitamos que alguns possuam um grau de disponibilidade maior do que outros, e por isso é razoável decompormos o sistema em *domínios de disponibilidade*, a fim de identificarmos qual a disponibilidade adequada a cada componente. Esta decisão de dividir o sistema em domínios de disponibilidade pode ser tomada por várias razões: custos, eficiência, falta de tecnologia existente, performance, etc.

Portanto, a tarefa de um projeto de alta disponibilidade é:

1. Identificar a tecnologia de middleware a ser utilizada para cada variante CA ou MM, CS, WS, HS/RA de HA[7].

2. Identificar, baseado no critério acima, quais componentes do sistema devem ser BA, MM, CS, WS, HS/RA ou CA.

[7]Assumimos que o suporte a BA não necessite de nenhuma tecnologia.

3. Organizar estes componentes em seus domínios, e prover o middleware de suporte a estes domínios.

A figura 2.3 ilustra os domínios de disponibilidade. Formas mais zelosas de disponibilidade são exibidas da esquerda para a direita.

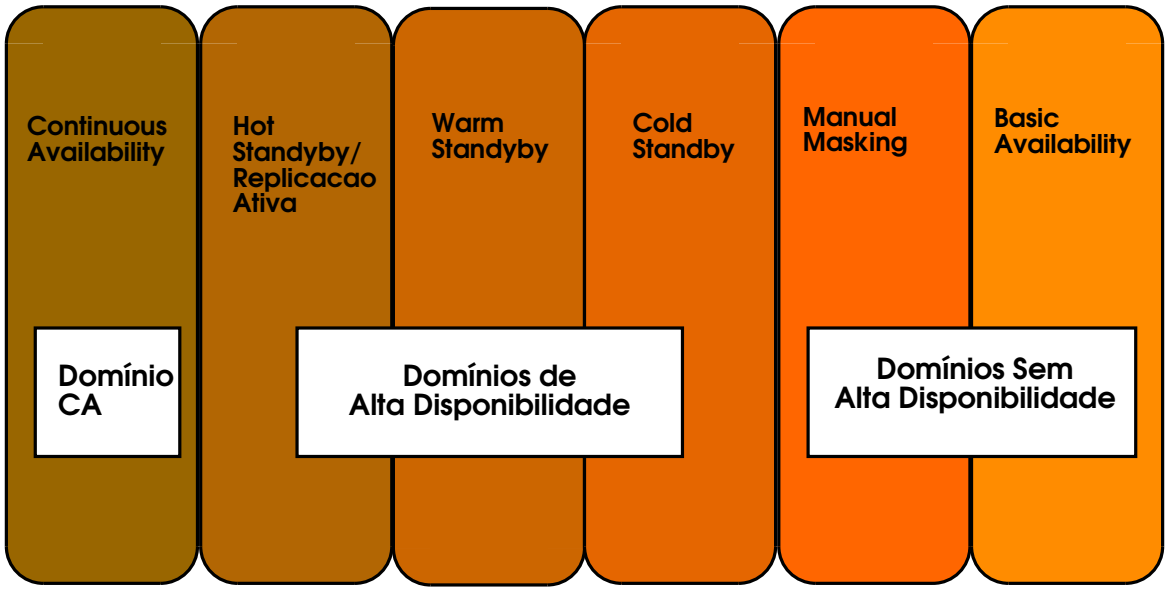

Figura 2.3: Domínios de Disponibilidade

2.6.5 Replicação Ativa e Passiva

O termo *Replicação Ativa* (RA) (*Active Replication - AR*, ou *Replicação Síncrona*) foi usado anteriormente como um sinônimo para *Hot Stand-By*. Este termo também é usado para descrever o conjunto de técnicas usadas para se alcançar Hot Stand-By, por compartilhar ativa e completamente o estado do processamento entre as réplicas. Como 'estado' é uma entidade dinâmica em sistemas distribuídos, compartilhar estado implica em não somente transferir informações entre as réplicas, mas também em coordenar e sincronizar as transações, a fim de que todas as réplicas possuam cópias consistentes das informações [77].

Técnicas de replicação ativa são usadas mais freqüentemente em clusters, onde as réplicas são os nós capazes de assumir os serviços defeituosos. Por isso, vários serviços básicos de cluster devem ser oferecidos para garantir tal replicação. Entre outros mecanismos, estes serviços compreendem a gerência do grupo de nós e da consistência do estado dos processos, e serão apresentados em detalhes na seção 3.5 e no capítulo 4.

Ao contrário de RA, o termo *Replicação Passiva* (*Passive* ou *Lazy Replication - LR*) descreve o conjunto de técnicas usadas para se atingir Warm Stand-By. Muitas vezes também descrita como Replicação Assíncrona, a Replicação Passiva é uma relaxação da RA. A idéia básica é que não queremos que todo o estado esteja compartilhado, mas somente parte dele [8]. O grande motivo de se optar por replicação passiva é o alto custo e complexidade de design que RA implica. Exemplos de abordagens Warm Stand-By incluem:

- Técnicas de replicação comuns em bancos de dados, onde um servidor principal envia ao stand-by um log de transações para ser aplicado. A diferença de sincronização entre as bases depende da periodicidade da replicação.

- Processos residentes em memória que são instanciados no processador ativo e stand-by. O processo primário atende a solicitações de todos os clientes, enquanto o secundário fica ocioso. No caso de falha, todos os clientes devem ser reconectados ao secundário, e devem realizar um rollback ao início de sua sessão de trabalho.

Replicação Passiva é freqüentemente associada com a noção de *Rollback & Recover (RR)*, já que em caso de falha, parte do trabalho é perdida e deve se retornar ao último estado consistente da informação.

2.6.6 Balanceamento de Carga

Balanceamento de Carga é um mecanismo usado para se aumentar a escalabilidade, dividindo a carga de processamento entre um conjunto de servidores, que é chamado de *fazenda de servidores* (*server farm*). Desta maneira, este conceito não possui nenhuma relação com Alta Disponibilidade. Entretanto, na prática os conceitos são freqüentemente ligados, pois o investimento feito para se adquirir sistemas redundantes para alta disponibilidade pode não ser justificável se o equipamento adicional está ocioso, ou se está meramente duplicando o trabalho executado nos servidores primários.

Freqüentemente, o requisito é possuir uma fazenda de servidores que estará preparada tanto para substituir servidores defeituosos em caso de falha, como também para dividir a carga de trabalho em uma situação de operação normal. Diferentemente de clusters de alta performance, esta divisão de carga é feita em um alto nível, ou seja, cada solicitação de um cliente é atendida completamente por um servidor; a idéia é dividir as solicitações, e não as sub-tarefas nelas envolvidas.

[8] Por também não ser específica, esta definição permite que várias configurações sejam definidas como Replicação Passiva.

O problema de tal abordagem é que complica-se o desenho de ambos aspectos (Alta Disponibilidade e Balanceamento de Carga), já que tal sistema não pode ser otimizado para as duas situações. Por exemplo, se um sistema é configurado para um determinada carga com n servidores, e sofre uma falha, a carga precisa ser agora dividida entre $n-1$ servidores. Isto claramente afeta a performance total, e, portanto, não pode mascarar devidamente a falha, pois a degradação da performance poderá ser observada por um cliente externo. Entretanto, de acordo com os requisitos, uma degradação ocasional do serviço pode ser aceitável. Mais uma vez, um projeto de alta disponibilidade deve tomar estas decisões, e concluir qual o nível de disponibilidade e transparência necessários.

2.7 Defeitos, Erros e Falhas

Até agora, vimos que um sistema de alta disponibilidade quer mascarar falhas para o usuário final. Entretanto, o termo "falha" foi usado muito vagamente, e nesta seção faremos seu devido esclarecimento.

Um *defeito* (*failure*) do sistema ocorre quando seu comportamento desvia do que designado por suas especificações [19]. Desta forma, um sistema está defeituoso quando ele não pode prover o serviço desejado. Um *erro* (*error*) é esta parte do estado do sistema que está suscetível a levar a defeitos subseqüentes [65]. Se há um erro no estado do sistema, então existe uma seqüência de ações que podem ser executadas e que levarão a defeitos no sistema, a não ser que medidas de correção sejam tomadas. A causa de um erro é uma *falha* (*fault*). Esta falha pode ser física ou lógica.

Como um erro é uma propriedade do estado do sistema, ele pode ser observado e avaliado. Já um defeito não é um propriedade do sistema, e não pode ser facilmente observado. Assim, a ocorrência de um defeito é deduzida detectando-se a ocorrência de algum erro no sistema. Como tipicamente não é viável avaliar todo o estado do sistema para determinar a ocorrência de erros, devemos escolher cuidadosamente qual parte do estado será avaliado, a fim de detectarmos a maioria dos defeitos. Esta escolha deve ser muito cuidadosa, e deve ser baseada no uso do sistema, a fim de podermos detectar os prováveis defeitos que porventura surjam.

Dizemos que um sistema não está se comportando corretamente devido à presença de uma falha. Falhas são associadas com a noção de defeitos: um sistema defeituoso é aquele que contém falhas. Definimos assim *falha* como sendo as deficiências no sistema que possuem o potencial de gerar erros.

Apesar de uma falha ter o potencial de gerar erros, ela pode não gerar erro algum durante o período de sua observação. Por exemplo, se uma célula de memória retorna sempre o valor 0 independentemente do que é armazenado nela, então ela contém uma falha. Entretanto, esta

falha pode não se manifestar até que a célula defeituosa seja usada e o valor 1 seja armazenado nela.

2.7.1 O Modelo de 3 Universos

Os conceitos de defeito, erro e falha podem ser melhor exibidos usando-se um modelo de 3 universos, o qual é uma adaptação do modelo de 4 universos, originalmente desenvolvido por Avizienis [21]. A figura 2.4 [94] descreve este modelo.

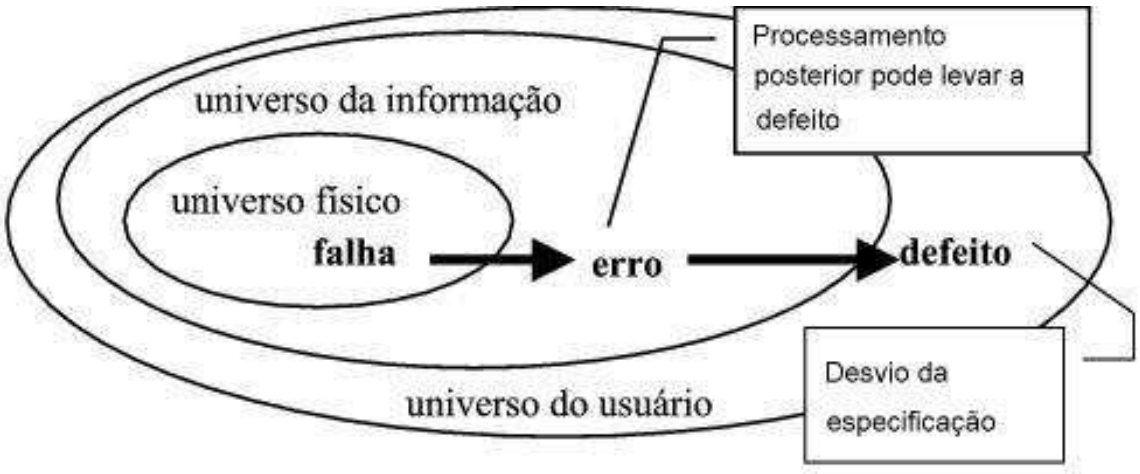

Figura 2.4: Modelo de 3 Universos

Neste modelo existem 3 universos distintos. O primeiro universo é o *universo físico* , no qual a falha acontece. Ele contém as entidades físicas que constituem o sistema. Uma falha é um defeito físico ou alterações de comportamento de componentes no universo físico. O segundo universo é o *universo da informação*, e é nele que os erros acontecem. Erros afetam unidades de informação dentro do computador. O terceiro universo é o universo exterior ou *universo do usuário* . É nele que o usuário de um sistema percebe a ocorrência de erros, através do aparecimento de defeitos.

2.7.2 Classificação de falhas

Podemos classificar as falhas de acordo com os critérios abaixo[9]:

[9]Nesta classificação estamos deixando de lado as falhas em sistemas distribuídos. Tais falhas serão tratadas na seção 3.3.

Natureza ou Causa

De acordo com sua natureza, as falhas podem ser classificadas em falhas físicas e falhas humanas [65]:

Falhas físicas são aquelas causadas por algum mau funcionamento de um componente físico. Este mau funcionamento pode ser originado por diversas razões, tais como curto-circuito, perturbações eletro-magnéticas, mudança de temperatura, etc.

Falhas humanas são imperfeições que podem ser:

- falhas de projeto, cometidas durante as fases de projeto e planejamento do sistema, ou durante a execução de procedimentos de operação ou manutenção.
- falhas de interação, cometidas por violar inadvertidamente ou deliberadamente procedimentos de operação ou manutenção.

Duração ou Persistência

De acordo com sua duração e momento de ocorrência, as falhas podem ser classificadas como transientes ou permanentes:

Falhas transientes são aquelas de duração limitada, causadas por mau funcionamento temporário ou por alguma interferência externa. Tais falhas podem ser também intermitentes, ocorrendo repetidamente por curtos intervalos de tempo.

Falhas permanentes são aquelas em que uma vez que o componente falha, ele nunca volta a funcionar corretamente. Muitas técnicas de tolerância a falhas assumem que os componentes falham permanentemente.

2.8 Fases na Tolerância a Falhas

Como visto anteriormente, um sistema tolerante a falhas tenta evitar a ocorrência de defeitos no sistema mesmo na ocorrência de falhas. Ainda, a implementação de um sistema tolerante a falhas será totalmente ligada à arquitetura e desenho dos recursos e serviços que serão disponibilizados. Apesar de não haver uma técnica geral sobre como adicionar alta disponibilidade a qualquer sistema, existem algumas atividades que todo sistema tolerante a falhas deve executar a fim de atingir este objetivo. Nesta seção, explicaremos tais atividades em detalhes, a fim de esclarecer todo o processo de alta disponibilidade. Grande parte desta seção está baseada em [59].

2.8.1 Detecção de Erros

A primeira fase na tolerância a falhas é a detecção de erros. Como as falhas e defeitos não podem ser detectadas diretamente, elas serão deduzidas a partir da detecção de erros no sistema. Assim, verificações deverão ser executadas para determinar suas ocorrências, a fim de dar-lhes o tratamento adequado. Deste modo, fica claro que a eficiência de um projeto de alta disponibilidade está baseada na sua eficiência em detectar erros. Idealmente, seria adequado que todos os erros fossem detectados. Entretanto, tal mecanismo exaustivo não seria viável na prática, fazendo com que seja necessário optarmos pelos erros a serem detectados. Portanto, devemos escolher as verificações que serão executadas a fim de conseguirmos observar a ocorrência destes erros desejados.

Devido à importância da detecção de erros, é necessário determinarmos como deve ser uma verificação ideal. Existem algumas propriedades importantes que devem ser satisfeitas [19]:

- Uma verificação ideal deve se basear somente na especificação do sistema, e não deve ser influenciada pelo seu desenho interno. O sistema deve ser considerado uma "caixa preta", de forma que sua implementação seja ignorada.
- Uma verificação ideal deve ser *completa e correta*, isto é, todos os possíveis erros projetados a serem verificados devem ser detectados, e nenhum erro deve ser declarado quando não existente.
- A verificação deve ser *independente* do sistema com respeito à suscetibilidade de falhas. Verificações também podem falhar, e queremos que suas falhas não sejam relacionadas com falhas no sistema que está sendo verificado.

Apesar das verificações de caixa preta serem as mais importantes, existem situações onde um conhecimento prévio da implementação pode ser bastante útil na escolha das verificações a serem executadas. Por exemplo, se todas as verificações escolhidas para se conferir um determinado componente passam pelo mesmo fluxo no sistema, então todas elas estão testando praticamente a mesma coisa. Por outro lado, se é conhecida a implementação, podemos forçar situações para acompanhar o resultado de linhas de execução distintas. Tal verificação é chamada de *Teste de Caixa Branca*, e idealmente também devem ser consideradas em projetos de alta disponibilidade[10].

As verificações de erro podem ocorrer de várias formas, dependendo do sistema e dos erros de interesse [19]:

[10]Na prática tal mecanismo não é muito usado devido à dificuldade em se verificar o código fonte dos sistemas onde está sendo aplicada a alta disponibilidade. Por exemplo, ao adicionarmos alta disponibilidade a um servidor HTTP de mercado, é muito comum realizarmos somente testes de caixa preta, que são basicamente simulações de requisições HTTP.

Verificações por Replicação

Verificações por Replicação são muito comuns e poderosas, podendo ser bem completas e implementadas sem o conhecimento do funcionamento interno do sistema. Tal verificação implica em replicar algum componente do sistema, e comparar resultados de diferentes componentes a fim de detectar erros. O tipo e a quantidade de réplicas dependem da aplicação. Tal forma de teste é usada freqüentemente em hardware.

Verificações Temporizadas

Se a especificação de um componente inclui restrições ao seu tempo de resposta, então *Verificações Temporizadas* podem ser aplicadas. Basicamente, tais verificações realizam uma solicitação a algum componente e verificam se o tempo de resposta excede ou não a restrição imposta em sua especificação. Verificações Temporizadas são usadas tanto em hardware como em software. Em sistemas distribuídos, ela possui um papel importante, pois a falha de um nó é determinada pelo atraso de sua resposta a determinada solicitação.

Verificações Estruturais

Em qualquer tipo de informação, dois tipos gerais de verificações são possíveis: verificações semânticas e estruturais. Verificações Semânticas tentam conferir se o valor é consistente com o resto do sistema. Verificações Estruturais só consideram a informação e garantem que internamente a estrutura dos dados é como deveria ser.

A forma mais comum de verificação estrutural é a codificação, que é usada intensamente em hardware. Nela, bits extras são adicionados aos dados, de forma que seja possível detectar se existe algum bit corrompido. Tal verificação também pode ser usada em software, sendo aplicada às estruturas de dados.

Verificações de Coerência

Verificações de Coerência determinam se o estado de algum objeto no sistema está "coerente". Um exemplo comum de tal mecanismo é verificar se determinado valor está dentro de um determinado intervalo. Outro exemplo é colocar instruções do tipo `assert` no meio do sistema, a fim de que incoerências sejam detectadas.

Verificações de Diagnósticos

Neste mecanismo, um sistema usa algumas verificações em seus componentes para determinar se ele está funcionando corretamente. A partir do conhecimento prévio de certos valores de entrada e de seus respectivos resultados de saída corretos, estes valores são aplicados ao componente e a saída é comparada com os resultados corretos.

2.8.2 Confinamento de Estragos e Avaliação

Se o sistema não for monitorado constantemente, haverá um intervalo entre a falha e a detecção do erro. Durante este intervalo, o erro pode se propagar para outras partes do sistema. Por isso, antes de executar medidas corretivas, é fundamental determinarmos exatamente os limites da corrupção, ou seja, o estado do sistema que está comprometido.

Erros se espalham em um sistema através da comunicação entre componentes. Por isso, para avaliar o estrago após a detecção de um erro, devemos examinar o fluxo da informação entre diferentes componentes. Ainda, devemos fazer algumas suposições sobre a origem e o momento de geração do erro, de forma a encontrarmos as barreiras no estado, além das quais nenhuma troca de informação tenha sido feita.

As barreiras podem ser encontradas *dinamicamente*, gravando-se e examinando-se o fluxo da informação. Por ser complexo, uma melhor maneira é projetar o sistema tal que "fire walls" são incorporados *estaticamente* ao sistema, de forma a garantir que nenhuma informação se espalhe para além deles.

2.8.3 Recuperação de Erros

Uma vez que o erro tenha sido detectado e sua extensão identificada, é o momento de removê-lo do sistema. Existem duas técnicas gerais para realizar a recuperação de erros:

Recuperação por Retorno (*Backward Recovery*)

Neste modelo, o estado do sistema é retornado a um estado anterior, na esperança de que este novo estado esteja livre de erros. Para isso, *pontos de controle* (*checkpoints*) periódicos são estabelecidos em um repositório estável. Quando algum erro é detectado, o sistema sofre um *rollback* para o último ponto de controle.

Este método é bem geral e um dos mais usados, além de não depender muito da natureza da falha. A principal desvantagem é o custo adicional necessário, pois além de ser necessário criarmos pontos de controle freqüentemente, o rollback envolve certo nível de computação pelo sistema.

Recuperação por Avanço (*Forward Recovery*)

Neste modelo, nenhum estado anterior está disponível. Ao contrário, é feita uma tentativa de se avançar, e tentar tornar o estado livre de erros aplicando-se medidas corretivas. Conceitualmente é interessante pois não há overhead. Entretanto, na prática torna-se difícil, pois depende de uma avaliação e suposições precisas sobre o estrago. Desta maneira, é um método muito dependente da aplicação, e por isso não é muito usado.

A tabela 2.4 resume tais técnicas de recuperação, e a figura 2.5 [94] exibe a idéia por trás destas técnicas. Deve ser observado que a recuperação é simples para um sistema com um único processo, mas complexa em um sistema distribuído [60], pois a recuperação nestes sistemas, usualmente de retorno, pode provocar um efeito dominó.

Técnica	**Definição**	**Características**	**Implementação**
recuperação por retorno	condução a estado anterior	alto custo, mas de aplicação genérica	pontos de controle
recuperação por avanço	condução a novo estado ainda não ocorrido	eficiente, mas danos devem ser precisamente previstos	específica a cada sistema

Tabela 2.4: Técnicas de Recuperação

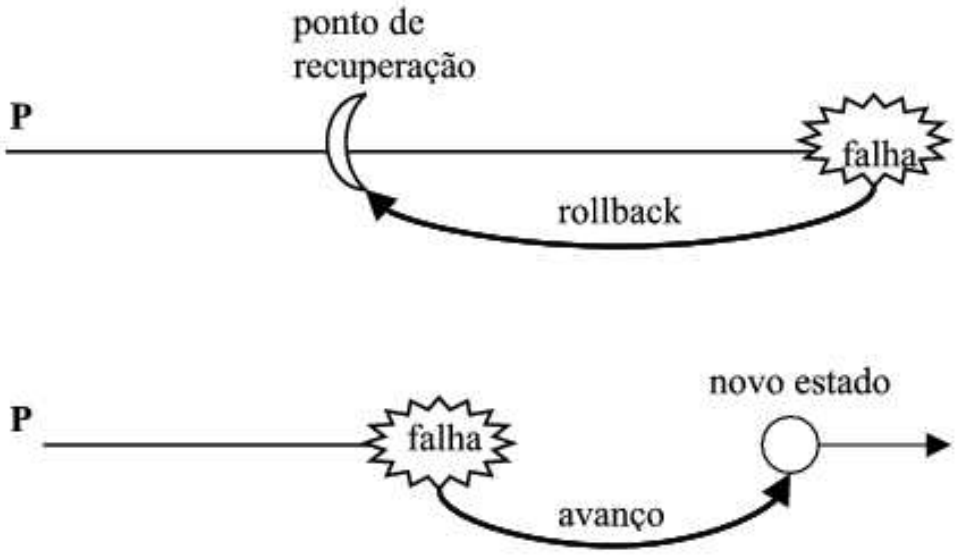

Figura 2.5: Recuperação por retorno e por avanço

2.8.4 Tratamento de Falhas e Serviços Continuados

Esta última etapa descreve o tratamento de falhas, que serve para impedir que futuros erros aconteçam. Nas três fases anteriores, o foco sempre foi o erro, e nunca a falha, que é justamente a origem do erro. Temos duas possíveis situações para a geração dos erros:

- Se o erro foi gerado por uma falha transiente, ela não precisa ser tratada, pois já desapareceu.
- Se o erro foi gerado por uma falha permanente, então ela ainda está presente após a recuperação. Desta maneira, se o sistema for reinicializado, o erro ocorrerá novamente. Para evitar este comportamento, o componente defeituoso deve ser identificado e não mais utilizado. Esta fase possui duas sub-fases:

  1. Localização da Falha.

     Nesta fase, o componente defeituoso precisa ser identificado. Se não for possível sua localização, não será possível reparar o sistema para que o erro não ocorra novamente. Tipicamente, após detectarmos o erro, o componente defeituoso é identificado como sendo aquele mais próximo da origem do erro.

  2. Reparo do Sistema.

     Nesta fase, o sistema é reparado para que o componente defeituoso não seja usado novamente. Deve ser notado que a manutenção é feita on-line, sem intervenção manual. O reparo é feito por um *sistema de reconfiguração dinâmica*. Em linhas gerais, tal sistema consiste em usar a redundância presente em sistemas distribuídos para substituir o componente defeituoso. Esta técnica será melhor detalhada na seção 3.6.

A tabela 2.5 [94] resume todas as etapas de tolerância a falhas vistas, através dos possíveis mecanismos para se executar cada uma das fases [19].

Fases	**Mecanismos**
detecção de erros	duplicação e comparação testes de limite de tempo cão de guarda (*watchdog timer*) testes reversos codificação: paridade, códigos de detecção de erros, Hamming teste de coerência, de limites e de compatibilidades testes estruturais e de consistência diagnóstico
confinamento e avaliação	ações atômicas operações primitivas auto encapsuladas isolamento de processos regras do tipo tudo que não é permitido é proibido hierarquia de processos controle de recursos
recuperação de erros	técnicas de recuperação por retorno (*backward error recovery*) técnicas de recuperação por avanço (*forward error recovery*)
tratamento da falha	diagnóstico reparo

Tabela 2.5: Fases na Tolerância a Falhas

Capítulo 3

Clusters de Alta Disponibilidade

> "Overall, the philosophy is to attack the availability problem from two complementary directions: to reduce the number of software errors through rigorous testing of running systems, and to reduce the effect of the remaining errors by providing for recovery from them. An interesting footnote to this design is that now a system failure can usually be considered to be the result of two program errors: the first, in the program that started the problem; the second, in the recovery routine that could not protect the system."
>
> A.L. Scherr [86]

3.1 Introdução

Para avançarmos no estudo de clusters de alta disponibilidade, é necessário compreendermos como eles são organizados e como operam para realizar as tarefas descritas nos capítulos 1 e 2.

Em linhas gerais, vimos que devemos mapear os sistemas e seus componentes em seus domínios de disponibilidade. Em seguida, devemos colocar redundância suficiente nestes componentes para que situações de falha possam ser contornadas. Este contorno ocorre através de um *failover* dos recursos afetados; ou seja, há uma *reconfiguração dinâmica* no cluster, a fim de que ele continue a prover seus serviços.

Este capítulo irá descrever todo este processo. Para tanto, conceituaremos as diferenças entre clusters e sistemas distribuídos, e abordaremos as falhas, modelos, serviços e blocos básicos existentes nestes sistemas.

3.2 Sistemas Distribuídos e Clusters

Até agora utilizamos o termo *sistemas distribuídos* no mesmo contexto de clusters. É necessário fazermos uma análise mais cuidadosa, a fim de compreendermos até que ponto estes termos

são equivalentes. A análise feita nesta seção foi baseada em [75].

3.2.1 Sistemas Distribuídos

Um sistema distribuído consiste em um conjunto de processos concorrentes que cooperam uns com os outros para executar determinada tarefa [59]. Ele é constituído de vários *nós*, que se utilizam da rede de comunicação para trocar mensagens uns com os outros. As máquinas são autônomas, formando um sistema fracamente agrupados, onde não existe memória compartilhada entre diferentes nós. Em nosso estudo, estamos mais interessados nos casos onde os processos citados acima executam em diferentes nós do sistema.

Exibimos na figura 3.1 o esquema de um sistema distribuído, de acordo com as características acima listadas. Entre as aplicações típicas de sistemas distribuídos "clássicos" estão serviços de arquivos de/para máquinas distribuídas e distribuição da carga de processamento entre várias máquinas.

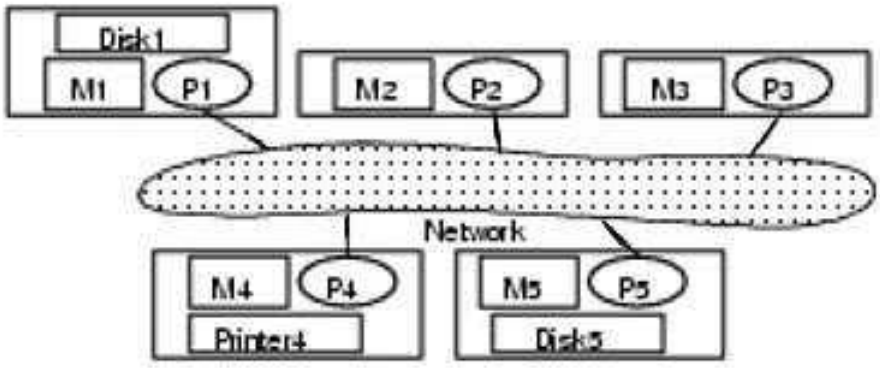

Figura 3.1: Exemplo de sistema distribuído

3.2.2 Clusters

Clusters são uma sub-classe de sistemas distribuídos. Geralmente eles são grupos de computadores homogêneos[1] em menor escala, dedicados a um número pequeno e bem definido de tarefas, nas quais o cluster atua com uma única máquina.[2] Entre os usos típicos de clusters estão

[1] Mesmo hardware e sistema operacional.

[2] O estudo de clusters nos mostrou que esta definição postulada em [75] nem sempre é verificada na prática. Muitas vezes construímos clusters de muitas máquinas para se atingir alta performance [24, 25]. Ainda, em algumas situações, nada nos impede de utilizar máquinas heterogêneas em um mesmo cluster.

adicionar alta disponibilidade a serviços e aumentar a capacidade de processamento do grupo (fazenda de servidores).

A figura 3.2 exibe o modelo mais simples de cluster de alta disponibilidade. Neste modelo, há um cluster de duas máquinas servindo arquivos para uma rede de computadores. Há uma máquina denominada *servidor primário*, e outra denominada *servidor de reserva*. Em todo momento existe somente um nó do cluster servindo arquivos. Em caso de defeito no servidor primário, o servidor de reserva assume o serviço, de maneira transparente para os usuários. Neste contexto, a transferência de controle entre os servidores pode ter duas denominações:

failover: apresentado na seção 1.4.2, ele ocorre quando o controle é transferido do servidor primário para o servidor de reserva (*backup*).

failback: quando o controle é transferido do servidor de reserva para o servidor primário.

Clusters com transferências automáticas de recursos (failover e failback) ajudam a evitar paradas planejadas e não planejadas do sistema.

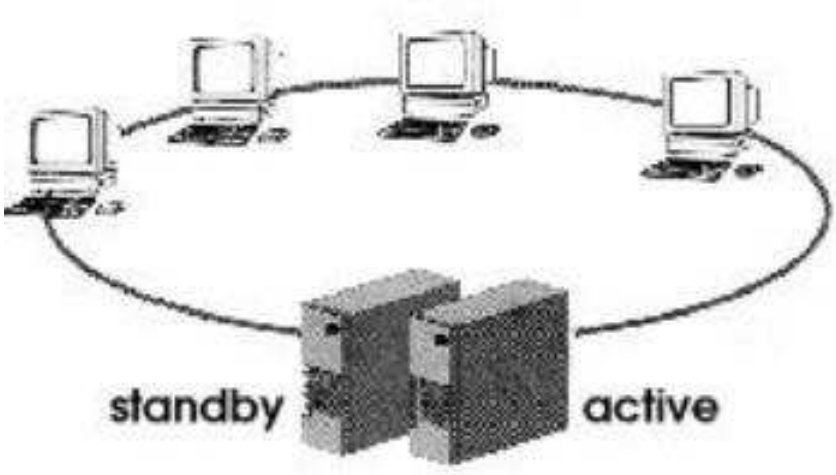

Figura 3.2: Exemplo de cluster

3.2.3 Clusters vs. Sistemas Distribuídos

Enumeramos algumas diferenças entre clusters e sistemas distribuídos na tabela 3.1. Analisando esta tabela, podemos concluir que um cluster nada mais é do que um sistema distribuído especializado para determinada função[3].

[3]Devemos nos lembrar que esta função resume-se em Alta Disponibilidade e/ou Alta Performance.

Características	Clusters	Sistemas Distribuídos
estrutura	homogênea - adquirido para realizar determinada tarefa.	heterogênea - montado a partir do hardware disponível.
escala	pequena escala - não é necessário garantir que a configuração seja escalável.	média/grande escala - devem ter o potencial de comportar um grande número de máquinas.
tarefa	especializada - feito para executar um conjunto pequeno de tarefas bem definidas.	generalizada - usualmente precisam ser ambientes de computação genéricos.
preço	relativamente baixo.	barato / caro.
confiabilidade	tão confiável quanto precisa ser.	pouco / muito confiável.
segurança	os nós confiam uns nos outros.	os nós não confiam uns nos outros.

Tabela 3.1: Clusters vs. Sistemas Distribuídos

Assim, qualquer técnica utilizada em sistemas distribuídos também se aplica a clusters. Estas técnicas pode ser tanto modelos de desenvolvimento de sistemas, como também métodos de comunicação e gerenciamento do grupo de computadores.

3.3 Falhas em Sistemas Distribuídos

Ao classificarmos as falhas no capítulo 2 não mencionamos sua ocorrência em sistemas distribuídos. Em um sistema distribuído, é possível classificarmos as falhas de acordo com o comportamento do componente defeituoso [40]:

Falhas de Colapso (*Crash Faults*).

São aquelas falhas que causam o colapso ou a perda do estado interno do componente defeituoso. Com este tipo de falha, um componente nunca se submete a alguma transição incorreta de estado quando falha.

Falhas de Omissão (*Omission Faults*).

São aquelas que fazem com que um componente não responda a algumas requisições.

Falhas de Temporização (*Timing Faults*).

São aquelas que fazem com que um componente responda ou muito cedo ou muito tarde. Também são chamadas de *falhas de performance*.

Falhas Bizantinas (*Byzantine Faults*).

São aquelas falhas que fazem com que o componente se comporte de uma maneira totalmente arbitrária.

A figura 3.3 ilustra esta classificação das falhas. Podemos perceber que falhas bizantinas são as mais indesejadas, devido à sua natureza arbitrária. Por outro lado, se todas as possíveis falhas em certo sistema se reduzissem a falhas de colapso, poderíamos elaborar rotinas de recuperação muito precisas, pois saberíamos que neste sistema um componente defeituoso simplesmente deixaria de funcionar, não corrompendo o estado do sistema. Este tipo de comportamento é muito desejado em sistemas distribuídos, e será melhor detalhado na seção 3.5.4.

Devido à dificuldade de se lidar com falhas em sistemas distribuídos, ao planejarmos um cluster de alta disponibilidade freqüentemente limitamos os modos de falha que este sistema pode apresentar[4]. Desta maneira, garantimos a disponibilidade do sistema somente no caso de falhas que sejam dos tipos previamente esperados.

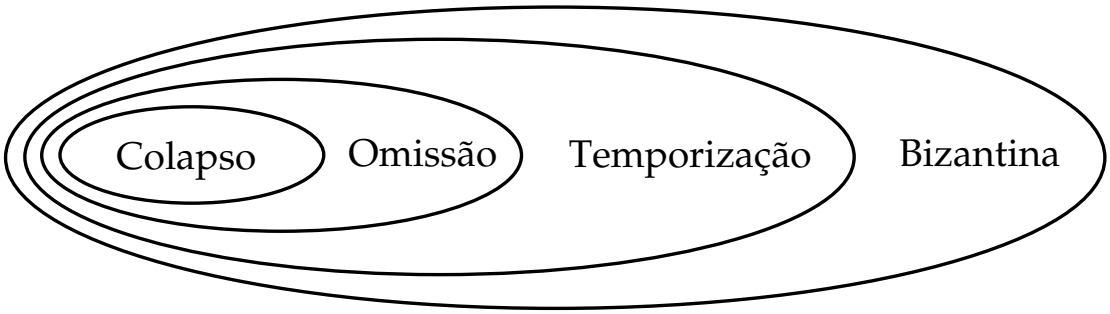

Figura 3.3: Classificação das Falhas em Sistemas Distribuídos

3.4 Modelos e Serviços Básicos de Sistemas Distribuídos

Modelamos um sistema distribuído a partir de dois atributos chaves: os atrasos de processamento e os atrasos de comunicação do sistema. Estes são fatores fundamentais para se definir o modelo de funcionamento de um sistema distribuído. Ainda, o conhecimento de serviços básicos para sistemas distribuídos é essencial para a construção de aplicações confiáveis. Esta seção visa esclarecer estes conceitos, e está baseada em [50].

[4] Geralmente não incluímos falhas bizantinas.

3.4.1 Modelos de Funcionamento

Consideramos a modelagem de um sistema distribuído a partir de nossa capacidade de determinar o pior caso da velocidade de execução de nós no sistema, e o pior caso do atraso de transmissão de uma mensagem entre qualquer par de nós do grupo. Se estes piores casos puderem ser determinados, então é possível estabelecer os limites superiores nos atrasos de processamento e comunicação para prover qualquer serviço. A confiança com a qual podemos especificar estes limites é crucial para determinar se é possível construir sistemas distribuídos, e, caso afirmativo, como devemos proceder.

Modelo Síncrono

O modelo síncrono permite a determinação de limites superiores com total certeza, e qualquer violação deste limite indica um defeito. Mais precisamente, em um sistema distribuído síncrono, assumimos como conhecidos os seguintes limites:

- Atrasos de Escalonamento: dentro de um nó correto, qualquer tarefa que for escalonada para ser executada no momento t irá iniciar sua execução antes de $t + \sigma$.
- Atrasos de Transmissão de Mensagens: se um processo p em um nó correto decide no momento t enviar uma mensagem m ao processo q em um outro nó correto, então m é recebida por q antes de $t + \delta$.
- Velocidade do Avanço de Relógios: todo processo em um nó correto tem acesso ao relógio de hardware, cuja velocidade de avanço a partir do tempo real é limitada por uma constante conhecida κ. O valor de κ é da ordem de 10^{-6} em relógios de cristal, o que significa que tais relógios podem diferir do tempo real no máximo 1 micro-segundo a cada 1 segundo. Veremos mais detalhes sobre sincronização de relógios em 3.5.2.

Modelo Assíncrono

No modelo assíncrono, os limites σ, δ e κ são finitos mas não podem ser determinados com exatidão. Comparado com o modelo síncrono, o modelo assíncrono pode representar mais precisamente uma classe maior de situações práticos:

- um nó que ocasionalmente fica sobrecarregado de processamento não será considerado defeituoso se o atraso de escalonamento exceder a cuidadosa estimativa limite.
- uma entrega de mensagem que demora mais do que o esperado devido a um congestionamento na rede não é um defeito.

Em outras palavras, o modelo assíncrono leva em conta *flutuações* na carga de processamento e da rede, e não considera elas como potenciais causas de defeitos. Entretanto, deve estar claro que mesmo não podendo determinar exatamente os limites de tempo no sistema, isto não significa que em modelos assíncronos não existe a noção de defeito. O modelo exige que os atrasos sejam finitos, apesar de indeterminados. Desta forma, os serviços do sistema devem ser fornecidos em algum ponto no tempo, mesmo que este ponto seja dificilmente previsível. Assim, o modelo admite todas os modelos de falha discutidos na seção 3.3, exceto falhas de temporização.

Apesar de flutuações na carga do processamento e da rede serem realistas, há conseqüências ao considerarmos normal a redução do fornecimento de serviços devido a estes motivos. Isto acarreta que o modelo assíncrono não permite que um componente travado seja deterministicamente distinguido de um componente que funciona a uma velocidade reduzida. Para ilustrar este aspecto indesejado do modelo, suponha que o processo p envie no momento local T uma requisição de um serviço para um processo remoto q, através de um canal confiável. Baseado nas interações passadas com q, p decide que q está travado se ele não receber sua resposta até $T+d$, para alguma constante finita d. Se q estiver funcionando lentamente, e enviar sua resposta depois de $T+d$, então a decisão de p havia tomado foi equivocada. Podemos afirmar que a decisão de p pode ser errada para qualquer valor finito de d que ele escolher. Assim, p precisa esperar infinitamente para tomar a decisão correta. Esta espera infinita é a causa básica para algumas impossibilidades de resultados que ocorrem com este modelo, como as que veremos na seção 4.6.3.

Devido a estas restrições do modelo assíncrono, geralmente é feita uma subdivisão deste modelo. Desta maneira, existem dois modelos assíncronos:

1. **Modelo Assíncrono Livre de Tempo (*Time-Free Asynchronous Model*)**

   O modelo time-free é o modelo assíncrono como descrito acima. Basicamente, não há limites superiores para os atrasos. Devido a esta característica, para termos certeza de certa decisão, devemos esperar possivelmente para sempre pela resposta. Como o próprio nome já diz, este modelo não possui a noção de *tempo*.

   A maior parte das pesquisas em sistemas assíncronos baseia-se no modelo time-free [54]. Este modelo é caracterizado pelas seguintes propriedades [41]:

   (a) A especificação dos serviços não coloca limites no tempo necessário para o sistema receber as requisições, processá-las e respondê-las.

   (b) A comunicação entre processos é confiável.

   (c) Os processos possuem semântica de falha[5] de colapso.

   (d) Os processos não possuem acesso a relógios locais.

[5] Trataremos melhor o assunto de semântica de falhas na seção 3.6.3.

2. **Modelo Assíncrono Temporizado (*Timed Asynchronous Model*)**

   O modelo assíncrono temporizado é uma variação do modelo assíncrono puro, onde é possível se determinar certos tipos de falha dentro de um intervalo de tempo finito.

   O modelo assíncrono temporizado (*timed asynchronous*) é um modelo assíncrono, pois não existem limites superiores para atrasos de escalonamento de processos e transmissão de mensagens. Entretanto, os processos possuem acesso a relógios locais, os quais permitem a definição de *timeouts*, a partir dos quais é possível definir falhas de performance [53]. Desta forma, este modelo é mais forte do que o modelo time-free.

   O modelo assíncrono temporizado assume que [41]:

   (a) Todos os serviços são temporizados: suas especificações incluem intervalos de tempo, dentro dos quais clientes podem esperar receber as respostas.

   (b) Os processos possuem semânticas de falha de colapso e de performance.

   (c) Os processos possuem acesso a relógios locais.

   (d) Não existem limites na freqüência de falhas de comunicação e de processos que podem ocorrer em um sistema.

   A utilidade de tal modelo baseia-se no fato que a maioria das aplicações tolerantes a falhas são *temporizadas* (*timed*), pois é exigido que elas respondam a requisições dentro de certo intervalo de tempo determinado.

3.4.2 Serviços Básicos

Para um grupo de computadores operar, existem certos serviços básicos que devem ser fornecidos internamente pelo cluster. Estes são serviços administrativos, isto é, serviços internos que não são disponibilizados para máquinas fora do grupo.

Considere um grupo G de processos distribuídos que cooperam para atingir determinado objetivo. Sejam p e q dois processos em G. Existem dois serviços que são fundamentais para a existência de um sistema distribuído:

Serviços de Difusão

Como o cluster é formado a partir de um conjunto de máquinas que se comunicam através de uma rede regular, a troca de mensagens é um pré-requisito para esta comunicação. Entre os vários mecanismos de trocas de mensagens, as difusões de mensagens são as mais usadas. Tais mecanismos permitem que um processo p envie uma mensagem m a alguns ou a todos processos de G, dependendo se a difusão for um *multicast* ou *broadcast*.

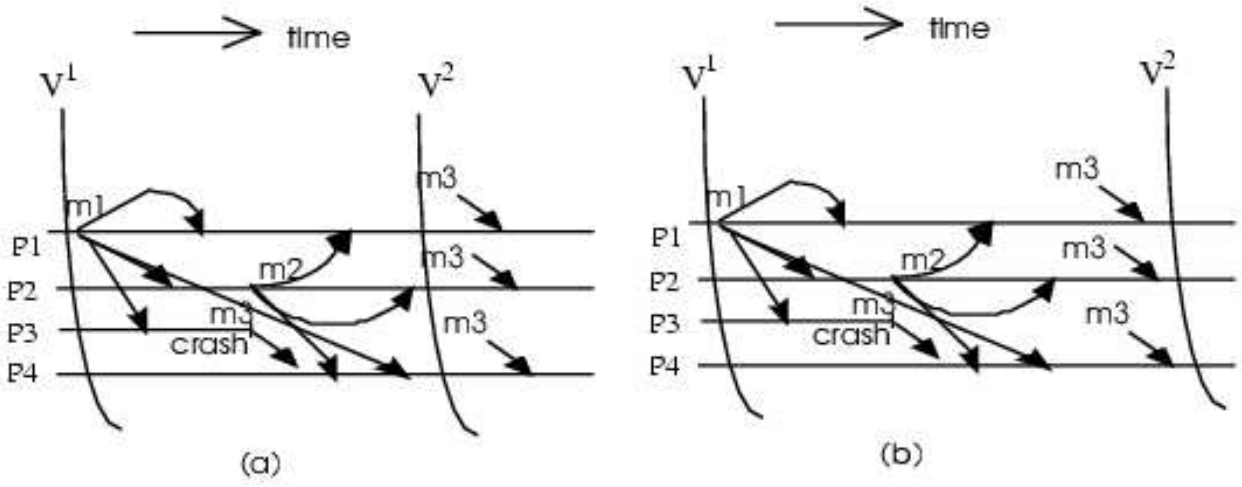

Figura 3.4: Entrega de Mensagens e Alterações da Visão do Grupo

O multicast geralmente aparece na forma de um *multicast confiável*, ou seja, é uma difusão de mensagem onde ou todos ou nenhum destinatário recebe e entrega a mensagem.

Em um cluster, geralmente o broadcast torna-se mais importante, pois ele é muitas vezes um pré-requisito para o *Serviço de Pertinência*. Sua forma mais utilizada é o *broadcast atômico*. De acordo com o modelo do sistema distribuído em questão, o broadcast atômico tem a seguinte definição:

Modelo Síncrono

Um broadcast atômico síncrono permite que o processo p envie uma mensagem m em qualquer momento (sincronizado) T ao grupo de membros ativos de G, tal que as seguintes propriedades são válidas [39]:

- Se p inicia o broadcast de m no momento T, então em $T + d$, ou m é entregue a todos os membros de G que estão ativos ou m não é entregue a nenhum membro ativo (*atomicidade*).
- Todas as mensagens são entregues na mesma ordem nos membros ativos de G (*ordenação*).
- Se o remetente p não falhar ao enviar m, então todos os membros ativos de G entregam m em $T + d$ (*término*).

Modelo Assíncrono

Um broadcast atômico assíncrono permite que o processo p envie uma mensagem m a todos

os membros ativos e corretos de G na mesma ordem [42]. Este serviço assegura a seguinte propriedade a quaisquer processos ativos p e q de G:

- Sejam $hist(p)$ e $hist(q)$ os históricos de mensagens entregues a p e q pelo serviço de broadcast atômico desde a criação do grupo. Então, ou $hist(p)$ é um prefixo de $hist(q)$, ou $hist(p) = hist(q)$, ou $hist(q)$ é um prefixo de $hist(p)$.

Desta maneira, independentemente do modelo assumido para o sistema distribuído, o broadcast atômico é um mecanismo de difusão de mensagens onde ou todos os integrantes do grupo recebem e entregam a mensagem na mesma ordem, ou nenhum o faz. Assim, ele nada mais é do que um multicast confiável, onde há uma garantia de que a mensagem será entregue em uma mesma ordem[6] a seus destinatários.

A figura 3.4(a) ilustra as garantias de um multicast confiável. P_1, P_2 e P_4 são processos corretos e P_3 trava antes de transmitir m_3 a P_1 e P_2. Um multicast confiável garante que as mensagens m_1 e m_2, enviadas respectivamente por P_1 e P_2, sejam entregues a todos os processos corretos, e m_3 seja entregue ou a todos ou a nenhum processo. A figura descreve o caso em que m_3 é entregue. Observe que para o esquema representar um broadcast atômico, os processos corretos deveriam entregar as mensagens na mesma ordem. No entanto, as mensagens são entregues em ordens diferentes: P_1 e P_2 entregam m_1, m_2, e m_3, enquanto P_4 entrega m_2, m_1, e m_3.

Serviço de Pertinência (*Group Membership Service*)

O serviço de pertinência (*membership service*) de um processo p (para o grupo G) provê a p a visão dos membros do grupo G que estão operando normalmente e estão conectados a p. Sempre que um novo processo se juntar a G, ou que um membro existente falhar ou deixar G, p deve obter uma nova visão que reflita as mudanças na lista de membros. A última visão que p receber é chamada de visão atual, ou simplesmente de *visão do grupo*. O serviço de pertinência fornece a lista de membros do grupo para todos os outros serviços internos ao cluster. Assim, este serviço é essencial para a operação de um sistema distribuído.

A figura 3.4 ilustra duas visões da lista de membros, V^1 e V^2, onde $V^1 = \{P_1, P_2, P_3, P_4\}$ e $V^2 = V^1 - \{P_3\}$. Repare que a visão V^2 ocorre em momentos diferentes nas figuras (a) e (b). Esta relação entre a visão do grupo e o momento de entrega de mensagens é muito importante, e será descrita quando estudarmos detalhadamente o Serviço de Pertinência no capítulo 4.

[6]Como não existe uma relação da ordem de entrega das mensagens com suas causalidades, freqüentemente encontramos textos que definem explicitamente esta ordem para o broadcast atômico. Por exemplo, encontramos textos que dizem que as mensagens devem ser entregues em uma ordem total a seus destinatários.

3.5 Blocos Básicos em Sistemas Distribuídos Tolerantes a Falhas

O objetivo de sistemas de alta disponibilidade é continuar a prover serviços mesmo na ocorrência de falhas em alguns de seus componentes. Existem diferentes métodos e esquemas para promover tal capacidade a sistemas distribuídos. Muitos deles fazem suposições implícitas ou explícitas sobre o comportamento do sistema e de seus componentes, e sobre seus modos de falha. Como estas suposições não são satisfeitas na prática, devem existir mecanismos para poder atingi-las, a fim de que estes esquemas possam funcionar. Desta maneira, estas suposições sobre o comportamento de sistemas distribuídos podem ser vistas como vários *blocos básicos* usados na construção de um sistema tolerante a falhas. Nesta seção, descreveremos cada um destes blocos básicos. Esta seção está baseada em [59], onde são encontradas descrições de possíveis métodos de se implementar estes blocos básicos.

3.5.1 Consenso Bizantino

Freqüentemente supomos que quando um componente falha, ele se comporta de uma maneira bem definida, apesar deste comportamento ser diferente do comportamento normal esperado. Entretanto, alguns componentes quando falham se comportam de maneira totalmente arbitrária. O problema de alcançar um consenso entre os componentes de um sistema distribuído, onde eles podem falhar de maneira arbitrária, é chamado de *Problema dos Generais Bizantinos* [63]. Os protocolos usados para se atingir este entendimento são chamados de *Protocolos de Consenso Bizantino*.

Definição do Problema

Considere um sistema distribuído com vários nós, onde estes nós trocam informações entre si através de mensagens. Os nós podem falhar, e um nó defeituoso pode mandar valores diferentes a nós distintos, relativos à mesma informação. O objetivo básico do *consenso bizantino* é atingir um consenso entre os nós não defeituosos sobre os valores corretos. Cada nó deve tomar uma decisão baseada nos valores recebidos dos outros nós, e todos os nós não defeituosos devem tomar a mesma decisão.

Para garantir que cada nó não defeituoso receba o mesmo conjunto de valores, podemos estabelecer o seguinte requisito: cada nó não defeituoso usa o mesmo valor de um nó i para tomar a decisão. Se esta propriedade for satisfeita por todos os nós, então o conjunto de valores para cada nó não defeituoso é o mesmo. Formalmente, este requisito é definido como [63]:

1. Todos os nós não defeituosos usam o mesmo valor $v(i)$ para um nó i.

2. Se o nó remetente i é não-defeituoso, então todo nó não-defeituoso usa o valor que i envia.

Este problema também é chamado de problema da *consistência interativa*, e a formulação acima permite qualquer tipo de comportamento durante a falha; o próprio nó remetente pode ser defeituoso. Dado um protocolo para resolver este problema, o protocolo pode ser usado por cada nó para enviar seu valor para outros nós. Assim, cada nó pode executar o mesmo procedimento para tomar a decisão baseada nos valores obtidos dos outros nós, alcançando assim um consenso.

Limitações de Solução

O Problema de Consenso Bizantino é difícil, porque a informação enviada por um nó pode não ser correta. Assim, para concordar com o valor enviado por um nó, além de obter o valor daquele nó, o valor que foi recebido pelos outros nós também é necessário para se verificar o valor original. O problema torna-se complexo, pois ao repassar um valor de um nó para outro, este nó pode estar defeituoso e passar uma informação incorreta. Assim, o problema pode ser resolvido somente se o número de nós defeituosos no sistema for limitado.

Se o sistema distribuído for assíncrono, podemos mostrar que não é possível atingir-se um consenso, mesmo se a falha for de colapso. Isto ocorre pois em tais sistemas não é possível distinguir uma falha do nó de uma lentidão do meio de comunicação.

Ainda, foi mostrado que com mensagens ordinárias é impossível resolver o problema, a não ser que pelo menos dois terços dos nós sejam não-defeituosos. Assim, para lidar com m nós defeituosos, pelo menos $3m + 1$ nós são necessários no cluster. O problema é simplificado se as mensagens são "assinadas", de maneira que nenhum nó pode alterar o conteúdo delas sem que isso não seja detectado por outro nó. Nesta situação, um número arbitrário de nós defeituosos pode ser tolerado.

Os algoritmos apresentados em [59] funcionam em rodadas, e são muito custosos no número de iterações e no número de mensagens necessárias para se atingir um consenso. Quando tratamos de algoritmos distribuídos, desejamos que um número relativamente pequeno de mensagens sejam trocadas, pois seu alto número pode causar lentidão excessiva na rede somente com informações de controle do cluster. Ainda, os protocolos apresentados necessitam de uma rede fortemente conectada, isto é, uma rede onde exista uma conexão ponto-a-ponto entre todos os nós do cluster. Apesar desta condição não ser sempre satisfeita, é possível adaptarmos os algoritmos para que não necessitem de tal requisito.

3.5.2 Relógios Sincronizados

Como definido anteriormente, um sistema distribuído consiste em um grupo de máquinas independentes fracamente agrupadas. Desta maneira, cada uma possui seu relógio interno. Como relógios são processos físicos, eles naturalmente diferem uns dos outros.

Através de relógios lógicos é possível definirmos uma ordem total e parcial de eventos, a qual não possui nenhuma relação com uma ordenação baseada em tempo real. Assim, manter os relógios sincronizados é um requisito para algumas aplicações distribuídas, pois a diferença de relógios em um sistema distribuído pode levar a sérios problemas de coordenação do cluster.

A sincronização pode ser *externa* ou *interna*:

- A sincronização *externa* de relógios implica em manter o relógio do processador com um desvio máximo de uma referência externa, que mantém o tempo real. É usada em sistemas de tempo real, onde o tempo real da ocorrência de certos eventos pode ser especificado.
- A sincronização *interna* de relógios implica em manter relógios de diferentes processadores com um desvio máximo uns dos outros. É usada em sistemas distribuídos para medir a duração das atividades.

Definição do Problema

Considere um cluster onde haja um relógio físico em cada nó, o qual é controlado por um processo neste nó. Como este processo que o controla é quem informa a hora marcada no relógio, uma falha no relógio pode ser encarada como uma falha neste processo. No que tange a sincronização de relógios, consideramos um modelo que pode perceber tanto falhas no processo como falhas no relógio. Os processos são numerados em $1, 2, ..., i, ...$, e estão conectados por uma rede de comunicação.

Seja $C_i(t)$ a leitura de um relógio C_i no momento físico t. Seja $c_i(T)$ o tempo real quando o i-ésimo relógio alcança um valor T. Em relógios sincronizados, desejamos que os valores de diferentes relógios sejam muito próximos uns aos outros, e próximos também ao tempo real.

Os relógios podem ser defeituosos ou não. Se ele não for defeituoso, supomos que o tempo exibido seja próximo ao tempo real, ou seja, ele corre a uma velocidade similar à do tempo físico, e sua variação é limitada por alguma constante, como especificado na equação 3.1. Em um relógio comum de cristal, ρ é da ordem de 10^{-6}.

$$|d(C_i)/dt - 1| < \rho \tag{3.1}$$

Os requisitos para uma sincronização de relógio são [62]:

1. Em qualquer momento, os valores de todos os relógios não defeituosos devem ser aproximadamente iguais. Isto é, para uma constante β,

$$|C_i(t) - C_j(t)| \leq \beta$$

2. Há um pequeno limite Σ na quantia pela qual um relógio não-defeituoso é alterado durante cada sincronização.

Ainda, assumimos que todos os relógios não-defeituosos correm aproximadamente à mesma velocidade que algum relógio que forneça o tempo real. O objetivo da sincronização de relógios é satisfazer as duas condições acima. A primeira condição garante que tempo fornecido pelos relógios estão próximos entre si. Como ela pode ser satisfeita trivialmente acertando cada relógio a algum valor qualquer (por exemplo, 0) em cada sincronização, existe a segunda condição que proíbe tais soluções triviais.

Limitações de Solução

Os protocolos que garantem os requisitos vistos acima podem ser classificados em determinísticos ou probabilísticos.

Sincronização de Relógios Determinística.

Em protocolos determinísticos, as condições e limites de sincronização são garantidos. Assim, a precisão solicitada do relógio é assegurada. Entretanto, freqüentemente eles necessitam de suposições sobre os atrasos nas mensagens, a fim de detectar suas perdas. Este problema pode ser modelado como um consenso bizantino, o que mostra uma forte semelhança com os problemas dos generais bizantinos.

Sincronização de Relógios Probabilística.

A grande vantagem de protocolos probabilísticos é que eles não necessitam de suposições sobre o atraso máximo nas mensagens. Neste modelo, os atrasos são aleatórios e ilimitados. Entretanto, uma sincronização de relógio probabilística garante a precisão do relógio baseada em uma alta probabilidade, e portanto não há uma garantia total como no modelo determinístico.

3.5.3 Repositório Estável

Técnicas de tolerância a falhas freqüentemente necessitam que algum estado do sistema esteja disponível após uma falha. Desta forma, supomos que o sistema possui algum *repositório estável*, cujo conteúdo tem sua integridade preservadada, mesmo em caso de falhas.

Definição do Problema

O problema resume-se em tornar estável para aplicações um sistema de armazenamento comum, com seus vários modos de falha. Assim como nenhum sistema pode ser tolerante a todos os possíveis tipos de falhas, um sistema estável de armazenamento aumenta sua confiabilidade tornando-se tolerante aos tipos de falhas mais prováveis e comuns.

Para entendermos o problema, é necessário primeiro estudarmos o sistema físico sobre o qual o repositório será construído, assim como seus modos de falha. Tipicamente, um repositório estável é construído a partir de um sistema de discos. Um sistema de armazenamento físico de discos é modelado como um conjunto de páginas (ou blocos ou setores) que possuem blocos de informação e um status associado a cada bloco, que pode ser *bom* ou *ruim*, dependendo se a informação está correta ou corrompida. Para interagir com o disco, processos podem usar duas operações [64]:

```
procedure read(adr) returns (status,data)

procedure write(addr, data)
```

A primeira operação lê um endereço do disco, retornando seu status e informação, e a segunda grava uma informação em uma determinada posição no disco.

Falhas em um Sistema de Armazenamento baseado em Discos

Vários tipos de erros podem surgir em sistemas de discos. Apesar de muitos deles poderem ser tratados com técnicas de codificação, outras falhas não podem, tais como:

1. *Falhas Transientes*: fazem com que o disco se comporte imprevisivelmente por um período de curta duração.
2. *Setor Ruim* (*Bad Sector*): uma página torna-se corrompida, e a informação nela armazenada não pode ser lida. Isto poderia ser causado por problemas físicas no hardware.
3. *Falha da Controladora*. A controladora do disco falha, tornando o conteúdo do disco inacessível, porém não corrompido.
4. *Falha do Disco*. O disco todo se torna ilegível. Este tipo de falha poderia ocorrer devido a uma falha de hardware, tornando o disco irrecuperável.

Em vez de enumerarmos todas as possíveis falhas físicas em um disco, convém estudarmos suas manifestações no comportamento do disco, e então provermos métodos de tolerar estes erros.

Os eventos que ocorrem em um sistema de discos compreendem os eventos gerados por operações de `read`/`write` e alguns eventos espontâneos. Alguns destes eventos são desejados (como um `read` retornando a informação correta), enquanto outros são indesejados, sendo estes causados por falhas físicas no disco. Assim, para construir um sistema de armazenamento estável devemos mascarar os eventos indesejados.

Os resultados indesejados para uma operação do tipo `read(a)` são [64]:

1. *Erro de Leitura Fraca* (*Soft Read Error*): a página `a` está *boa*, mas `read` retorna *ruim*. Esta situação pode não persistir por muito tempo, e é causada por falhas transientes.

2. *Erro de Leitura Persistente* (*Persistent Read Error*): a página `a` está *boa*, mas `read` retorna *ruim*, e `read`s sucessivos também retornam *ruim*. Pode ser causado por um setor ruim (ou uma falha do disco/controladora).

3. *Erro Indetectado* (*Undetected Error*): a página `a` está *ruim* mas `read` retorna *boa*, ou a página `a` está *boa*, mas `read` retorna informação diferente.

Se um `write` é bem sucedido, a página `a` é alterada para `d`. Entretanto, ele pode não ter sucesso, e um `write(a,d)` pode causar os seguintes eventos indesejados [64]:

1. *Escrita Nula* (*Null Write*): a página `a` é inalterada.

2. *Escrita Ruim* (*Bad Write*): página `a` torna-se `(ruim,d)`.

Além destes eventos que ocorrem com `read`s e `write`s, existem certos eventos espontâneos que causam falhas no disco. Freqüentemente supomos que tais eventos ocorrem raramente, e que aparecem nas seguintes formas [64]:

1. *Corrupção* (*Corruption*): uma página vai de `(boa,d)` para `(ruim,d)`.

2. *Restauração* (*Revival*): uma página vai de `(ruim,d)` para `(boa,d)`.

3. *Erro Indetectado* (*Undetected Error*): uma página vai de `(s,d)` para `(s,d')`, com `d` $\neq$ `d'`.

O objetivo de um repositório estável é tomar um ou mais sistemas de armazenamento de discos, os quais podem falhar nas maneiras acima descritas, e construir um sistema onde as operações abstratas do disco (isto é, `read` e `write`) continuem a funcionar mesmo em caso de falhas.

Limitações de Solução

Primeiramente, é necessário lembrarmos que não é possível tolerar todos os tipos de falha. Ainda, com uma quantidade finita de componentes, não é possível obtermos completa tolerância, pois todos os componentes podem falhar ao mesmo tempo.

Existem várias técnicas para se implementar repositórios estáveis descritas em [59]. Com elas, é possível tolerar todos os erros acima listados. Em um sistema com espelhamento de discos (*disk shadowing*) (figura 3.5), o $MTBF_{mirror}$ é dado pela fórmula 3.2, onde $MTBF$ e $MTTR$ se referem aos valores de um único disco do sistema [31].

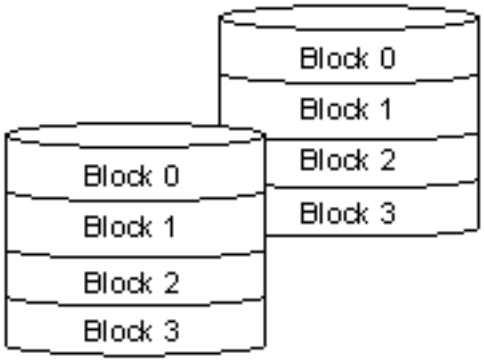

Figura 3.5: Espelhamento de Discos - RAID 1

$$MTBF_{mirror} = \frac{MTBF}{2} * \frac{MTBF}{MTTR} \tag{3.2}$$

Não daremos uma derivação formal da fórmula neste texto. Intuitivamente, ela significa que o tempo médio para falhar do espelhamento é o tempo médio até que ocorra a primeira falha ($MTBF$) dividido por 2, multiplicado pelo inverso da probabilidade de uma segunda falha durante o reparo do primeiro disco, que é igual a $MTTR/MTBF$. Com um $MTBF$ de 5 anos, e um $MTTR$ de 3 horas, o tempo médio entre falhas de um disco espelhado - $MTBF_{mirror}$ - será de mais de 30.000 anos! [31]

Já em um sistema com *RAID* (figura 3.6), o $MTBF_{RAID}$ é dado pela fórmula 3.3, onde G é o número de discos de informação, C o número de discos de verificação, e $MTBF$ e $MTTR$ se referem aos valores de um único disco do sistema [73]. Observe que um espelhamento de discos é o tipo mais simples e mais caro de *RAID*, pois é necessário um disco de backup (ou seja, de verificação) para cada disco original.

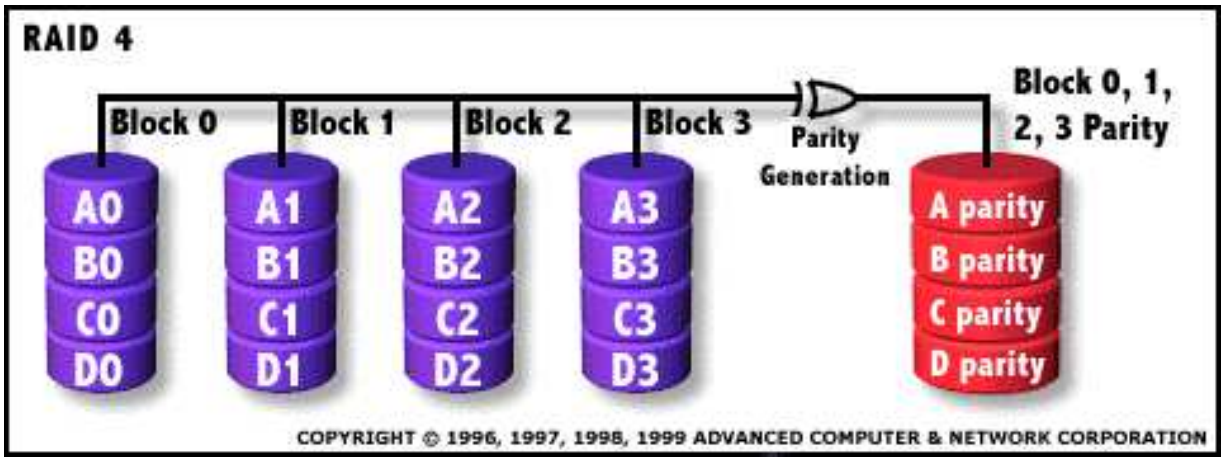

Figura 3.6: Exemplo de RAID: RAID nível 4

$$MTBF_{RAID} = \frac{MTBF}{G+C} * \frac{MTBF/(G+C-1)}{MTTR} \tag{3.3}$$

O primeiro termo é o tempo médio de falha do primeiro disco. O segundo termo relaciona a probabilidade de outro disco no grupo falhar antes que o primeiro disco seja reparado. Considerando-se um $MTBF$ de 5 anos e um $MTTR$ de 3 horas, e 4 discos de informação e 1 disco de verificação, o $MTBF_{RAID}$ será mais de 3.000 anos. Para os detalhes da demonstração desta equação, consulte [73].

3.5.4 Processadores Fail-Stop

Geralmente os esquemas de tolerância a falhas em clusters fazem algumas suposições sobre o comportamento de nós defeituosos. Supomos freqüentemente que, em caso de falha, o nó não irá executar operações inválidas e simplesmente deixará de funcionar. Nós que apresentam este tipo de comportamento são chamados de *processadores fail-stop*.

Definição do Problema

Um processador possui um conjunto de instruções, que são aquelas que ele pode executar. A execução de uma instrução pode alterar o estado interno do processador, e também o estado da memória que o processador acessa. Para um processador que funciona perfeitamente, os efeitos de se executar diferentes instruções podem ser precisamente definidos, enquanto que em um processador defeituoso, seu comportamento pode ser arbitrário. Geralmente, quando um processador falha, nada pode ser afirmado sobre seu comportamento, exceto que é diferente do

comportamento previsto. Tal modo de falha genérico torna a tarefa de se construir um sistema tolerante a falhas extremamente complexa.

Um *processador fail-stop* é caracterizado por seu modo de falha extremamente simples: durante uma falha, o processador simplesmente pára de funcionar. Ainda, o estado interno do processador e o conteúdo da memória volátil à qual ele está anexado são perdidos para sempre, e o conteúdo do repositório estável anexado ao processador não é afetado por tal falha. Além disso, geralmente supomos que a falha de um processador fail-stop pode ser detectada por outros processadores. Desta forma, os efeitos visuais de uma falha de um processador fail stop são [88]:

1. Ele pára de executar.
2. O estado interno e o conteúdo da memória volátil conectada ao processador são perdidos; o conteúdo do repositório estável é inalterado.
3. Qualquer outro processador pode detectar a falha de um processador fail stop.

Como processadores genéricos não possuem estas características, e elas são necessárias a um esquema tolerante a falhas, o objetivo é construir tal abstração a partir de um processador regular.

Possíveis Soluções

Como não é possível tolerar falhas simultâneas de muitos componentes, definimos um processador k-fail-stop como um sistema computacional que se comporta como um processador fail-stop a não ser que $k+1$ ou mais componentes falhem. Para se implementar tal abstração, é importante a interação do repositório estável com o processador fail-stop. Este irá ler e escrever no repositório, além de utilizá-lo para se comunicar com outros processadores. Claramente, o funcionamento de um processador fail-stop dependerá da confiabilidade do sistema de armazenamento de informações.

É possível imaginar nesta situação um processador que controla o meio de armazenamento, e este será chamado de *processador de armazenamento*. Ele é responsável pela interação dos processadores que desejam usar o repositório estável, além de gerenciar o meio de armazenamento. Assim, neste modelo existem processadores para rodar programas e processadores para prover o repositório. Desta maneira, serão usadas as palavras *processo* e *processador* com o mesmo significado, pois o processador estará executando uma tarefa (processo).

Ao definirmos um repositório estável, utilizamos propositalmente a suposição de que o processador de armazenamento funciona corretamente. Agora no estudo de processadores, esta suposição deve ser revista. Um repositório estável pode não funcionar corretamente se o meio

de armazenamento não funcionar corretamente, ou se o processador de armazenamento que controla este meio não funcionar. Estas duas situações podem ser facilmente modeladas como falhas de funcionamento do processador de armazenamento. Assim, um repositório estável que não funciona implica em um processador de armazenamento que não funciona corretamente.

Se o processador de armazenamento está funcionando corretamente, então a tarefa de se implementar um processador k-fail-stop é consideravelmente simplificada. Entretanto, se o processador de armazenamento puder falhar, então a implementação deve levar este fato em consideração. Desta maneira, existem duas maneiras de se implementar um processador k-fail-stop:

1. Implementação com um Repositório Estável Confiável

   Neste modelo, o processo de armazenamento (ou processo-s, de *storage process*) funciona corretamente e não falha. Um processador k-fail-stop é construído a partir de $k+1$ processadores ordinários, que podem falhar de maneira arbitrária. Isto permite a ocorrência de falhas *bizantinas* ou *maliciosas*, na qual um processador pode atuar de maneira a frustrar qualquer consenso no sistema distribuído.

   Um exemplo deste modelo seria a implementação de um processador 1-fail-stop. Pela definição, ele se comportará como um processador fail-stop a não ser que 2 processadores falhem. Caso somente um falhe, será detectado o erro e o processamento será interrompido. Caso contrário, seu comportamento será arbitrário.

2. Implementação com um Repositório Estável Não-Confiável

   Este modelo não supõe que o processo-s é estável. Entretanto, para implementarmos um processador k-fail-stop, é necessário implementarmos um repositório estável confiável também, e, portanto, são necessários múltiplos processadores-s. Igualmente aos processadores, presumimos que os processadores-s podem falhar de maneira arbitrária, inclusive maliciosas. Além dos $k+1$ processadores necessários para replicar a computação, são necessários $2k+1$ processos-s diferentes para implementar o repositório estável, cada um rodando em um processador diferente. Conforme descrito em [59], neste modelo deve ser usado um protocolo de consenso bizantino para implementar um processador k-fail-stop.

3.5.5 Detecção de Defeitos e Diagnóstico de Falhas

Uma vez que algum componente em um sistema distribuído falha, o objetivo de um sistema tolerante a falhas é mascarar esta falha para clientes externos. Isto significa que outros componentes devem executar atividades extras além das suas normais. Assim, devem existir outros componentes para detectar e diagnosticar a falha de um certo componente.

A maioria das aplicações e métodos tolerantes a falhas em sistemas distribuídos assume que

uma vez que um nó falha, outros nós perceberão esta falha dentro de um período de tempo finito, e então executarão as medidas de reparo necessárias. Este requisito é facilmente satisfeito por um método de força bruta, onde cada nó utiliza intervalos de tempo para detectar falhas. Entretanto, em um grande sistema, esta abordagem pode ter um custo alto demais. Nesta seção discutiremos o tema de diagnóstico de falhas em sistemas distribuídos. Para isso, estudaremos primeiramente sistemas centralizados de diagnósticos de falha, e então diagnósticos de falhas distribuídos.

Diagnósticos Centralizados de Falhas

O objetivo básico de diagnósticos de falhas é identificar as unidades defeituosas em um sistema. Claramente, nem sempre é possível atingir este objetivo. Por exemplo, quando todas as unidades em um sistema são defeituosas, nenhuma unidade pode ser usada para executar o diagnóstico. Assim, assumiremos um limite no número de unidades defeituosas.

Um modelo deve ser usado para o propósito de se determinar quão diagnosticável um sistema é, e para executar operações de diagnósticos. O primeiro modelo que foi apresentado é conhecido como modelo PMC.

- Modelo PMC

  Neste modelo, um sistema S é decomposto em n unidades, não necessariamente idênticas, denotadas pelo conjunto $U = \{u_1, u_2, ..., u_n\}$. Cada unidade é uma porção bem definida do sistema, que não pode ser mais decomposta para o propósito de diagnóstico. As unidades devem ser capazes de testar outras unidades no sistema, e determinar se elas estão ou não funcionando corretamente.

  No modelo PMC, cada unidade que pertence a U possui um conjunto de U para ser testado (nenhuma unidade testa a si mesma). O conjunto completo de testes é representado por um grafo G, onde cada vértice representa uma unidade, e uma aresta (u_i, u_j) existe em G se e somente se o nó u_i testa o nó u_j. Uma saída a_{ij} é associada a cada teste (u_i, u_j): se u_i é livre de falhas, então a_{ij} é 0 se u_j for não defeituoso, e 1 se for defeituoso; se u_i for defeituoso, o resultado não é confiável e pode ser qualquer valor. A figura 3.7 ilustra este modelo. Nela, a unidade 1 está defeituosa, e portanto os testes que ela realiza não são confiáveis.

  Um sistema S é dito t-falhas-diagnosticável se todas as unidades defeituosas em S podem ser identificadas, desde que o número de unidades defeituosas não exceda t. Em tal situação, o problema de se determinar t para um dado sistema, isto é, determinar o número máximo de unidades que podem ser defeituosas, é chamado de o *problema da diagnosticabilidade*.

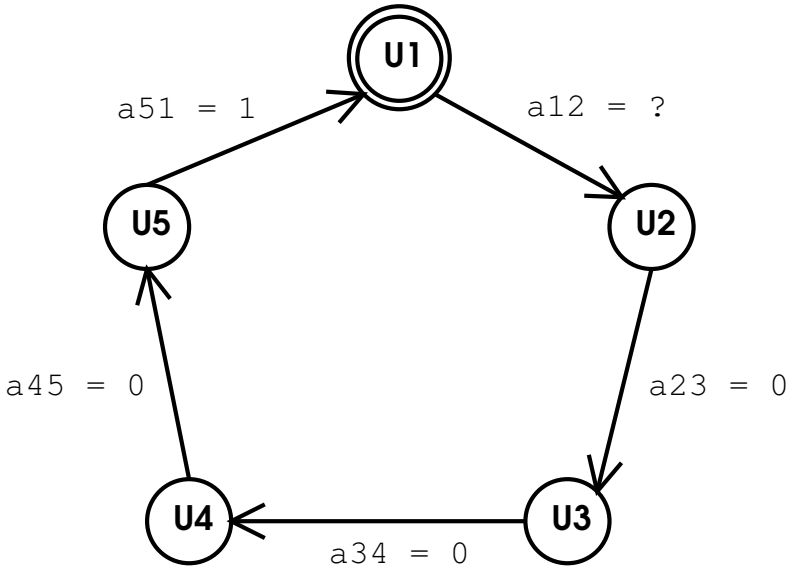

Figura 3.7: Grafo de testes do modelo de diagnósticos PMC

- Algoritmos de Diagnósticos

  Uma vez que o grafo de testes foi definido e os testes executados, a coleção de todos os resultados é chamada de *síndrome* do sistema. A síndrome do grafo de testes acima é um vetor de 5 bits, da forma

  $$(a_{12}, a_{23}, a_{34}, a_{45}, a_{51}).$$

  No caso de somente a unidade u_1 for defeituosa, a síndrome será:

  $$(x, 0, 0, 0, 1).$$

  O problema de diagnose é determinar a partir da síndrome o estado do sistema, isto é, quais unidades são defeituosas e quais estão funcionando corretamente. Isto é feito através algoritmos de diagnósticos. Como o modelo PMC utiliza um observador central, ele não é adequado para sistemas distribuídos.

Diagnósticos de Falhas em Sistemas Distribuídos

O objetivo do diagnóstico em sistemas distribuídos é garantir que se alguns nós falharem (ou se recuperarem), então outros nós no sistema notarão sua falha (recuperação) em um período de tempo finito.

O algoritmo NEW-SELF foi o primeiro proposto para auto-diagnósticos de sistemas distribuídos. Ele é distribuído por sua natureza, e todos os nós livres de falhas diagnosticam independentemente o estado do sistema. O algoritmo funciona corretamente, desde que um número de nós defeituosos no sistema não seja maior do que t. Há um grafo de testes fixo, e um nó é responsável por testar um subconjunto definido de seus vizinhos. Os nós livres de falhas passam seus resultados de diagnósticos para seus vizinhos. Cada nó determina independentemente o estado do sistema dependendo dos resultados de seus testes e dos resultados recebidos de seus vizinhos.

Mais recentemente, um algoritmo chamado *DSD adaptativo* foi proposto para diagnósticos em sistemas distribuídos. Ele foi projetado para grandes sistemas distribuídos, e considera somente falhas de nós, apesar delas precisarem ser limitadas. Os resultados dos testes executados por um nó livre de falhas é considerado correto, enquanto que nós defeituosos produzem resultados não confiáveis. A idéia básica desta abordagem é que um nó, além de seus próprios testes, também utiliza os resultados dos testes dos outros nós para determinar o estado do sistema. Durante a execução do algoritmo, um nó aceita as informações de teste de um nó somente se ele determinar que este nó transmissor é livre de falhas.

Outro aspecto importante em diagnósticos em sistemas distribuídos é o planejamento dos testes que serão executados. Testes complexos podem ser interessantes, mas devemos lembrar que um teste não deve executar operações muito pesadas, pois ele provavelmente será executado freqüentemente. Em ambientes distribuídos, nunca devemos esquecer de testar também os meios de comunicação, pois eles são componentes fundamentais para o funcionamento do cluster.

3.5.6 Entrega Confiável de Mensagens

Em sistemas distribuídos, assumimos freqüentemente que uma mensagem enviada por um nó chega incorrupta ao seu destino, e que a ordem das mensagens é preservada entre dois nós. Isto quer dizer que, se um nó i envia várias mensagens para outro nó j, então o nó j recebe as mensagens na mesma ordem em que foram enviadas. Entretanto, perdas de mensagens e erros de transmissão acontecem em meios de comunicação reais. Assim, são necessários mecanismos para satisfazer estes requisitos.

Definição do Problema

Considere uma rede de nós conectados por um meio de comunicação. Supomos que o grafo representando a rede é conexo, isto é, para todo par de nós i e j, existe um caminho entre eles no grafo. Para cada nó i, j do grupo, desejamos que as seguintes propriedades sejam válidas:

1. Uma mensagem enviada por i é recebida corretamente por j.

2. Mensagens enviadas por i são entregues a j na mesma ordem em que i as enviou.

Além disso, desejamos que estas duas propriedades sejam válidas mesmo em caso de falhas de nós ou de componentes da rede. Para satisfazer estas propriedades em um cluster particionado seria necessário atrasar a entrega de certas mensagens. Por isso, consideramos neste texto somente os casos onde não há particionamentos no cluster.

Possíveis Soluções

As duas propriedades acima são tipicamente partes da arquitetura de comunicação. Desta maneira, não forneceremos os detalhes destes protocolos ou suas implementações. Daremos somente uma breve descrição do funcionamento destes protocolos, e, então, uma explicação de como estas propriedades são mantidas mesmo em caso de falhas.

- Detecção de Erros

  Erros de transmissão são inevitáveis, e suas causas são as mais variadas: ruídos, atenuação de sinal, linha cruzada, etc. Estas causas são tipicamente transientes, isto é, ocorrem por períodos de curta duração.

  Erros de transmissão podem ser detectados pelo receptor. A maneira mais comum é usar algum tipo de codificação na informação, de maneira que qualquer alteração da mensagem possa ser verificada. Métodos como Bits de Paridade e CRC são muito usados.

  Se um receptor detecta que uma mensagem foi corrompida, geralmente ela é descartada, a não ser que esteja sendo usado algum mecanismo de correção de erros. Como a corrupção de mensagens geralmente causa a perda de pacotes, os protocolos de comunicação devem garantir que as mensagens sejam entregues. Assumimos geralmente que erros são detectados por mecanismos de detecção de erros, e falhas permanentes de um nó são detectadas por outros nós.

- Ordenação de Mensagens e Entrega Garantida

  Um protocolo de comunicação pode oferecer dois diferentes tipos de comunicação entre dois nós: orientado à conexão e não orientado à conexão. Um serviço orientado à conexão é modelado conforme o sistema telefônico: uma conexão primeiramente é estabelecida, e então a informação é transferida usando-se esta conexão. Em um serviço não orientado à conexão, cada mensagem carrega o endereço completo de seu destino, e é roteada independentemente das outras. Ainda, é possível que alguma mensagem se perca, ou chegue

fora de ordem no destino. Claramente, para fornecer as propriedades desejadas, um serviço orientado a conexão é necessário.

Um método popular para prover tal tipo de serviço é usar protocolos de *janelas deslizantes*. Protocolos de janelas deslizantes garantem as duas propriedades desejadas, mesmo em um meio de comunicação não confiável, além de executar controle de fluxo e outras funções interessantes.

- Falhas

  O protocolo de janelas deslizantes funciona desde que falhas em nós ou meios de comunicação não aconteçam. O ponto chave para tratar as falhas é o roteamento de mensagens. Na maioria das vezes, as mensagens precisarão de muitos pulos para atingir seu destino, isto é, atravessarão muitos nós para atingir seu destino. Um algoritmo de roteamento é usado então para se decidir qual é o caminho mais adequado que uma mensagem deve tomar.

  Suponha que um algoritmo de roteamento é usado numa rede para rotear mensagens do nó i para o nó j através do nó k. Se o nó k falhar (ou algum canal nesta rota falhar), precisamos de um algoritmo de roteamento robusto que comece então a rotear as mensagens através de outro caminho[7]. Se os protocolos de roteamento puderem garantir isso, então os protocolos de janela deslizante garantirão a entrega na ordem correta.

  Para tratar falhas é necessário um *algoritmo de roteamento adaptativo*, no qual as decisões de roteamento podem mudar de acordo com as mudanças na topologia da rede. Muitos algoritmos de roteamento adaptativos são conhecidos, cada um com diferentes pontos fortes e fracos.

3.5.7 Broadcast de Mensagens

Nas seções anteriores, discutimos a comunicação ponto-a-ponto confiável como um dos blocos básicos. Apesar da comunicação ponto-a-ponto ser suficiente para muitas aplicações, há muitas outras que necessitam que um nó envie uma mensagem para vários nós de uma única vez. Tal forma de comunicação foi apresentada na seção 3.4.2, onde foi colocado que o *broadcast* é a forma de comunicação onde um nó envia uma mensagem a todos os outros nós do grupo. Como a primitiva básica de comunicação fornecida por uma rede é ponto-a-ponto[8], esta primitiva deve ser usada para implementar broadcast. Desta forma, a implementação está suscetível a falhas de nós e de comunicação, e isto não é aceitável em sistemas tolerantes a falhas. Basicamente, há três propriedades que interessam quando mensagens são enviadas a diferentes nós:

- *Confiabilidade*: a mensagem deve ser recebida por todos os nós em operação.

[7] Como a rede é conexa e particionamentos não ocorrem, outra rota deve existir.
[8] Exceto em redes que suportam diretamente broadcasts.

- *Ordenação Consistente*: mensagens diferentes enviadas por nós diferentes devem ser entregues a todos os nós na mesma ordem.
- *Preservação da Causalidade*: a ordem na qual as mensagens são enviadas deve ser consistente com a causalidade entre os eventos de envio destas mensagens.

Descreveremos nesta seção cada um destes requisitos em broadcasts. Estaremos considerando somente falhas que não causem o particionamento do cluster.

Broadcast Confiável

Um broadcast confiável possui a seguinte propriedade básica: uma mensagem enviada deve ser recebida por todos os nós operacionais, mesmo na ocorrência de falhas. Esta propriedade deve ser válida mesmo se o nó transmissor falhar após ter enviado a mensagem para apenas alguns nós.

Duas possíveis soluções para este problema são:

- Usando Encaminhamento de Mensagens.

  Este protocolo modela a rede como uma árvore, que é usada como base para a disseminação das mensagens. A raiz da árvore é o nó transmissor. Se há uma aresta de um nó P para um nó Q na árvore, isto implica que durante o broadcast, o nó P irá encaminhar a mensagem ao nó Q.

  A árvore é somente uma estrutura lógica usada para organizar os nós, não possuindo nenhuma relação direta com a estrutura física da rede. O protocolo não se preocupa em como montar a árvore, apesar de que seria mais eficiente se vizinhos na rede fossem vizinhos na árvore. A árvore é estaticamente definida e conhecida por todos os nós.

  Este protocolo garante que se uma mensagem enviada pelo nó raiz S alcançou ao menos um nó que não falhou, então a mensagem alcança todos os outros nós que também não falharam.

- Usando "Empilhamento" de Confirmações.

  Este protocolo usa uma combinação de confirmações positivas e negativas para alcançar seu objetivo. Ele supõe que quando um nó realiza um broadcast, alguns nós recebem a mensagem e outros não.

  A idéia básica do protocolo é "empilhar" confirmações positivas e negativas numa mensagem de broadcast. Cada mensagem de broadcast carrega a identidade do transmissor, e um único número seqüencial. A partir das confirmações positivas e negativas, um nó receptor

sabe qual mensagem ele não precisa confirmar, ou quais mensagens ele perdeu e precisa solicitar sua retransmissão. Esta idéia é ilustrada através de uma seqüência de eventos em um sistema de três nós, P, Q, R:

1. O nó P faz um broadcast de m_1.
2. O nó Q recebe a mensagem, e empilha uma confirmação na próxima mensagem que ele enviar, por exemplo, m_2.
3. Ao receber m_2:
   - Se R recebeu a mensagem m_1, ele percebe que não precisa enviar uma confirmação para ela, pois Q já enviou.
   - Se R não recebeu m_1, ele percebe sua perda pela confirmação que consta m_2, e solicita uma retransmissão ao enviar uma confirmação negativa na próxima mensagem que ele enviar.

Broadcast Atômico

Conforme detalhado na seção 3.4.2, o paradigma de broadcast atômico é mais estrito que o broadcast confiável: ele não só implica que uma mensagem enviada por um nó é recebida por todos os nós operacionais, como também que mensagens enviadas por nós diferentes devem ser entregues nos nós receptores na mesma ordem.

Para compreendermos esta ordem de entrega, é necessário entendermos a diferença entre *receber* e *entregar* mensagens: *receber* uma mensagem significa que o nó a recebeu usando sua interface de rede; após receber a mensagem, o nó (ou o sistema operacional) precisa *entregar* a mensagem ao processo que irá consumi-la. Protocolos de broadcast atômico devem garantir que as mensagens sejam entregues na ordem correta, mesmo que não tenha sido esta a ordem de recebimento das mensagens.

Dois métodos conhecidos para se implementar este serviço são:

- Usando Relógios Sincronizados.

  O protocolo usa difusão de mensagens para alcançar seus objetivos: quando um nó inicia o broadcast de uma mensagem, ele marca na mensagem o horário de seu relógio e adiciona um identificador único do nó. Em seguida, ele envia a mensagem através de todos os links que possuir. Quando um nó intermediário recebe a mensagem, ele a encaminha por todos os canais que possuir, exceto pelo canal em que a mensagem chegou. Para satisfazer a propriedade de ordenação, os nós entregam as mensagens na ordem do timestamp das mensagens. Se os timestamps são iguais, o identificador do nó é usado para tomar a decisão. Atrasos de mensagens limitados e relógios sincronizados são usados por um nó

para determinar o quanto ele deve esperar antes de ter certeza de que mensagens com timestamps menores possam chegar.

- Usando "Empilhamento" de Confirmações.

  O protocolo acima garante somente que as mensagens são entregues na mesma ordem, mas não que elas serão recebidas por todos os nós operacionais. O protocolo aqui descrito é uma extensão do protocolo de "empilhamento" de confirmações para broadcasts confiáveis, de maneira que ele possa implementar uma primitiva de broadcast atômico.

  Neste protocolo, como confirmações positivas ou negativas são adicionadas às mensagens em broadcasts, um nó pode determinar se outro recebeu ou não uma mensagem. Para compreender como isto é possível, imagine a seguinte situação:

  1. É feito o broadcast da mensagem m_1;
  2. Posteriormente, o broadcast de m_2 é efetuado, contendo uma confirmação positiva de m_1;
  3. Se o broadcast de m_3 contiver uma confirmação positiva para m_2, então o nó que enviou m_3 *deve* ter recebido m_1, pois caso contrário teria enviado junto com m_3 uma confirmação negativa para m_1.

  Através de confirmações transitivas como exibidas acima, uma *ordem parcial* pode ser estabelecida de maneira que todas as mensagens sejam entregues nos nós em uma mesma ordem. Para maiores informações, consulte [70].

Broadcast Causal

Em broadcasts atômicos, a ordem na qual as mensagens eram entregues não era importante; desejávamos somente que qualquer que fosse esta ordem, ela deveria ser a mesma em todos os nós. Tal requisito pode ser suficiente para muitas aplicações, mas pode existir a necessidade de que as mensagens sejam entregues de acordo com a causalidade de seus eventos, isto é, se a mensagem m_1 foi enviada antes de m_2, então m_1 deve ser entregue antes de m_2 também.

Para isto, definimos a relação "aconteceu antes" (representada por $\rightarrow$) que define a causalidade de eventos. Assim, $e_1 \rightarrow e_2$ significa que o evento e_1 pode causalmente afetar o evento e_2. Ainda, esta relação cria uma ordem causal, onde e_1 acontece antes de e_2.

Para resolver este problema, duas abordagens podem ser usadas:

- Broadcast Causal sem Ordenação Total.

  Neste modelo, mensagens que possuam alguma relação de causalidade são entregues em todos os nós de maneira a respeitar esta ordem causal . Entretanto, mensagens que não

possuam nenhuma relação de causalidade podem ser entregues em qualquer ordem, possivelmente diferente.

- Broadcast Causal com Ordenação Total.

  Neste modelo, as mensagens são entregues na mesma ordem para todos os nós. Esta ordem deve respeitar a causalidade das mensagens.

3.6 Reconfiguração Dinâmica

Agora que já compreendemos como são os modelos de funcionamento dos sistemas distribuídos, seus serviços e blocos básicos, podemos entender como clusters podem atingir alta disponibilidade para seus recursos.

Já sabemos que eles devem mascarar a presença de falhas. Para isto ocorrer, os clusters devem ser capazes de se reorganizar e se reconfigurar dinamicamente, de maneira que os serviços afetados sejam restabelecidos nos nós restantes. Para esta reconfiguração dinâmica ser eficiente, o nó que passar a prover os serviços deve possuir seu estado interno atualizado com as informações necessárias; isto é, ele deve estar *sincronizado* com o nó que falhou, para prover o serviço com as mesmas informações utilizadas pelo servidor anterior.

Esta seção visa esclarecer como todas estas etapas ocorrem, e como funciona um Serviço de Gerenciamento de Disponibilidade. Ela está amplamente baseada em [39], [38] e [42].

3.6.1 Conceitos Básicos

Um serviço fornecido a um usuário é o comportamento do sistema como percebido por este usuário: uma seqüência de saídas disparadas por uma seqüência de invocações de operação do serviço [39]. Serviços com o mesmo conjunto de operações invocáveis e comportamentos potenciais possuem o mesmo *tipo de serviço*. As operações definidas para um serviço podem somente ser realizadas por uma *implementação do serviço*, que consiste em um ou mais servidores. Um servidor é a unidade de defeito e crescimento: em qualquer momento no tempo, uma implementação de um serviço possui um grupo de servidores capazes de prover o serviço. Assim, quando algum serviço deve permanecer disponível mesmo em caso de falha de alguns servidores que o implementam, torna-se imperativo distinguirmos os conceitos de serviços e servidores.

Se servidores redundantes são usados para implementar um serviço, o usuário não precisa saber que políticas de sincronização e replicação existem entre eles. A *política de sincronização* determina o quão distantes podem estar os estados locais dos servidores.[9]. Esta política de sincronização pode ser classificada em:

[9] A distância entre estados locais de dois servidores consiste na diferença do número de atualizações ao estado

- Sincronização Fraca (*Loose Synchronization*): um servidor primário mantém o estado atual do serviço, enquanto um ou mais servidores de reserva mantêm estados passados.

- Sincronização Próxima (*Close Synchronization*): os servidores atuam como "colegas", ao interpretar todas as solicitações do serviço em paralelo, e manter seus estados internos próximos uns dos outros.

A *política de replicação* para um serviço s especifica quantos servidores devem existir para s. As políticas de sincronização e replicação especificadas para um serviço formam a *política de disponibilidade* para este serviço. Assim, a política de disponibilidade de um serviço s é (PS, n), onde PS é uma política de sincronização, e n é a quantidade de servidores existentes para prover s (ou seja, a política de replicação).

O objetivo do *Serviço de Gerenciamento de Disponibilidade* (SGD) é aplicar automaticamente as políticas de disponibilidade especificadas para serviços críticos, os quais são oferecidos a seus clientes por um sistema distribuído. Por exemplo, se a política de disponibilidade de um serviço s é (sincronização fraca, 2), e o servidor primário de s falha, o Serviço de Gerenciamento de Disponibilidade deve reconfigurar dinamicamente o outro servidor para que este possa continuar fornecendo o serviço s.

3.6.2 Requisitos do Serviço de Gerenciamento de Disponibilidade

Para simplificar nossa apresentação, consideraremos que o SGD deve aplicar uma única política de disponibilidade para todo o conjunto S de serviços críticos, e que a única razão para um servidor de um serviço $s \in S$ travar é o colapso de seu nó. Consideraremos também que a política de disponibilidade especificada pelo administrador de sistemas é (sincronização fraca, 2).[10]

Seja P o o conjunto dos nós do sistema. Se denotarmos o conjunto de nós ativos do sistema por N ($N \subseteq P$), e o conjunto de nós que podem hospedar um serviço $s \in S$ por $H(s)$, é preciso que o SGD mantenha um servidor primário e de reserva para s em nós distintos enquanto existir pelo menos dois nós em $N \cap H(s)$, e um servidor primário para s enquanto que o número de nós em $N \cap H(s)$ for um.

O SGD oferece as seguintes operações a dois "usuários" concorrentes [39]:

- o administrador do sistema: as operações que o administrador pode invocar são:

inicial aplicadas a eles.

[10] Esta suposição serve simplesmente para facilitar nossa exposição. Sua adaptação para outros cenários não será aqui exibida, mas é perfeitamente possível.

`start-service(s)`: inicializa o serviço s.

`stop-service(s)`: pára o serviço s.

`add-hosts(s,h)`: indica que os nós h podem hospedar o serviço s.

`remove-hosts(s,h)`: indica que os nós h não podem mais hospedar o serviço s.

`start-node(n)`: inicializa o nó n, colocando-o na lista de nós ativos do sistema.

onde $s \in S$, $n \in N$, $h \subseteq P$.

- o ambiente adverso: o ambiente pode "invocar" a operação

  `crash-node(n)`: trava o nó n, tirando-o da lista de nós ativos do sistema.

  onde $n \in N$.

Estas operações alteram o estado do Serviço de Gerenciamento de Disponibilidade, que é composto pelas variáveis P, S e N, entre outras. Tais transições de estado devem ser executadas pelo SGD, enquanto que pelo menos um nó estiver rodando no sistema.

Mesmo que ocorram alterações na lista de membros do cluster, um SGD que satisfaz os requisitos acima provê *disponibilidade contínua* a clientes de qualquer serviço crítico s, enquanto existir pelo menos um nó ativo que possa hospedar o serviço s.

3.6.3 Semântica de Falhas do Servidor

Quando programamos ações de recuperação para uma falha de um servidor, é importante sabermos *qual* o comportamento de falha que ele provavelmente exibirá.

Como ações de recuperação disparadas por falhas em um servidor dependem de seus prováveis comportamentos de falha, em um sistema tolerante a falhas devemos *estender* a especificação padrão dos servidores. Além de sua semântica livre de falhas[11], devemos incluir seus prováveis comportamentos com falhas, ou seja, sua *semântica de falhas* [38]. Se a especificação de um servidor s descreve que seus usuários provavelmente observarão falhas da classe F, dizemos que "s possui semântica de falhas F". De maneira geral, quanto mais forte for a semântica de falhas, mais caro e complexo será para construirmos um servidor que a implementa.

Uma pergunta que surge neste contexto é: "quando é justificável assumirmos que os únicos comportamentos com falha de r estão na classe F?". Para contemplar esta resposta, a especificação de um servidor r deve consistir não somente de requisitos *funcionais* S_r e F_r que descrevem a semântica normal e com falhas do servidor, mas também uma especificação *estocástica*. Esta especificação deve determinar uma probabilidade mínima s_r para a qual o comportamento normal

[11] O conjunto de comportamentos livres de falhas.

S_r é observado em tempo de execução, assim como uma probabilidade máxima c_r para a qual uma (possivelmente catastrófica) falha diferente da especificada em seu comportamento com falhas F_r é observada. A partir das probabilidades s_r e c_r podemos verificar se F_r é uma semântica de falha apropriada, conforme descrito em [38].

3.6.4 Mascaramento de Falhas

Quando uma falha é verificada, ela deve ser mascarada, a fim de que o serviço continue sendo fornecido a seus usuários. Este mascaramento pode ser feito de duas maneiras [38]:

Mascaramento Hierárquico

Se um servidor depende de outros servidores em um nível inferior de abstração para prover seu serviço corretamente, então falhas de certo tipo em um nível inferior de abstração podem resultar em falhas de diferente tipo em um nível mais alto de abstração.

A tarefa de se verificar a exatidão de resultados fornecidos por servidores em níveis inferiores é muito incômoda. Por isso, projetistas de sistemas tolerantes a falhas preferem utilizar semânticas de falhas mais fortes, como colapso, omissão ou performance. Em sistemas hierárquicos baseados nestes servidores, o *tratamento de exceções* provê um método conveniente para se propagar a informação sobre a detecção de falhas através de níveis de abstração, e para se mascarar para níveis superiores falhas em níveis inferiores.

Se o servidor u em j depende do servidor r em i, mas pode prover seu serviço mesmo se r falhar, então dizemos que u *mascara* a falha de r. Se a tentativa de mascaramento de u não tiver sucesso, um estado consistente deve ser recuperado para u, antes que seja propagada a informação da falha de u para o próximo nível de abstração, onde outras tentativas de mascaramento ocorrerão.

Servidores que, para qualquer estado inicial e dados de entrada, ou fornecem seu serviço padrão ou sinalizam uma exceção sem alterar seu estado simplificam a programação tolerante a falhas, pois eles provêem a seus usuários uma semântica de falhas de omissão fácil de ser compreendida.

Para ilustrar este modelo de mascaramento, considere uma máquina multiprocessada. Quando uma tentativa de se executar uma instrução em um processador falha (ou seja, uma exceção de erro é detectada), há uma nova tentativa automática que é feita em um outro processador. Desta maneira, há um mascaramento automático, que tenta esconder a falha, e fornecer para o usuário o comportamento esperado. A mesma abordagem pode ser imaginada e adotada em sistemas distribuídos.

Mascaramento por Grupos

Para garantir que um serviço permaneça disponível mesmo em caso de falhas de servidores, podemos implementar este serviço por um *grupo* de servidores redundantes e fisicamente independentes, de maneira que se algum deles falhar, os sobreviventes continuem fornecendo o serviço. Dizemos que o grupo *mascara* a falha de um membro m quando o grupo (como um todo) responde como especificado para usuários, mesmo no caso de uma falha de m. Enquanto que em mascaramentos hierárquicos o usuário deve programaticamente mascarar as falhas, mascaramentos por grupos permitem que falhas dos membros do cluster sejam totalmente escondidas dos usuários pelos mecanismos de gerenciamento do grupo. Na prática, freqüentemente são adotadas abordagens que combinam elementos de mascaramento hierárquico e por grupos.

A *saída* de um grupo é uma função das saídas dos membros deste grupo. Por exemplo, a saída pode ser:

- a saída gerada pelo membro mais rápido do grupo;
- a saída gerada por um membro diferenciado do grupo;
- o resultado de uma votação por maioria baseada nos resultados dos membros do grupo.

Um grupo de servidores capaz de mascarar de seus clientes quaisquer k falhas simultâneas de seus membros é chamado de k-tolerante a falhas. Quando k for 1, chamamos o grupo de "tolerante a uma falha" (*single-fault tolerant*), e quando k for maior do que 1, o grupo será chamado de "tolerante a múltiplas falhas" (*multiple-fault tolerant*).

Quanto mais forte for a semântica de falhas dos membros do grupo e dos serviços de comunicação, mais simples e eficientes serão os mecanismos de gerenciamento do grupo. Como é mais caro construir servidores com uma semântica de falha mais forte, mas é mais barato lidar com o comportamento de falha destes servidores em níveis mais altos de abstração, um ponto chave no projeto de sistemas tolerantes a falhas multi-camadas é determinar como *balancear* as quantidades de detecção de falha, recuperação e redundância de mascaramento usadas nos vários níveis de abstração de um sistema, a fim de se obter o melhor resultado geral de custo/performance/dependabilidade.

3.6.5 Serviço de Gerenciamento de Disponibilidade

Como vimos nas seções anteriores, vários passos precisam ser seguidos para se implementar um Serviço de Gerenciamento de Disponibilidade (SGD). Precisamos escolher a política de disponibilidade do servidor, precisamos saber a semântica de falhas deste servidor, e precisamos optar

por um mecanismo de mascaramento de falhas[12]. Ainda, o SGD precisa satisfazer os requisitos descritos em 3.6.2.

Como este serviço também está sujeito a falhas, ele também deve estar replicado em todos os nós, a fim de que esteja disponível, enquanto houver um servidor ativo. Como estão replicados no grupo, estes servidores precisam atuar de maneira sincronizada para executar as operações de recuperação. Assim, já podemos visualizar que o SGD depende totalmente do Serviço de Pertinência do cluster, pois a informação da lista de membros é fundamental, tanto para realizar os mascaramentos das falhas como para replicar o serviço pelo cluster.

De acordo com o modelo utilizado no cluster, a implementação do SGD ser tanto síncrona como assíncrona.

Modelo Síncrono

A implementação apresentada em [39] depende diretamente dos serviços de pertinência e de broadcast atômico síncronos: qualquer atualização a uma variável de estado replicada é resultado ou da chegada de uma mensagem de broadcast atômico ou de uma notificação de alteração da lista de membros do cluster.

É necessário que quaisquer dois membros do grupo de gerentes de disponibilidade ativos possuam estados locais idênticos, a fim de que eles possam tomar sempre as mesmas decisões sobre o que deve ser feito. Assim, todas as atualizações à lista de membros e às variáveis de estado replicadas devem ser recebidas *na mesma ordem* por *todos* os gerentes de disponibilidades ativos.

Modelo Assíncrono

A implementação apresentada em [42] depende diretamente de um serviço de broadcast atômico assíncrono, o qual depende de um serviço de pertinência assíncrono, que por sua vez depende de um serviço de comunicação por datagramas. Tal dependência é exibida na figura 3.8.

Como estamos em um ambiente assíncrono, não é possível distinguirmos falhas de nós de atrasos ou sobrecargas no cluster. Portanto, o SGD precisa levar isto em conta, a fim de não levantar o mesmo serviço em dois nós concorrentemente. Existem algumas técnicas que são utilizadas para ter certeza de que o nó realmente está parado antes de se inicializar o serviço em outro nó. Geralmente, estas técnicas envolvem o uso de um dispositivo de hardware que desliga/reinicializa a máquina remotamente, garantindo assim que o servidor parou de fornecer

[12] Freqüentemente implementamos um mecanismo de mascaramento por grupos, pois este se torna mais transparente para os usuários do serviço.

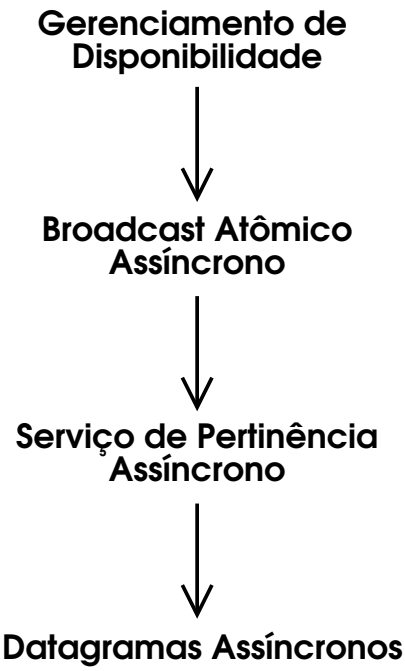

Figura 3.8: Dependências do Serviço de Gerenciamento de Disponibilidade Assíncrono

o serviço. Esta técnica é chamada muitas vezes de STONITH (Shoot The Other Node In The Head) [83].

Quando ocorrem particionamentos na rede, existem duas soluções para se manter a política de disponibilidade [42]:

1. Aplicada especialmente a serviços com estado persistente, a primeira solução assume que não é permitido que estados locais dos servidores divirjam devido a particionamentos na rede. Neste caso, servidores podem existir em somente uma partição (a partição ativa). Isto pode ser implementado obrigando-se os membros da partição minoritária a parar todos os serviços que são executados localmente.

2. A segunda solução é aplicada em casos onde não há mal em se ter servidores ativos em partições diferentes. Neste caso, a política de disponibilidade é garantida em cada partição isoladamente. Quando ocorre uma união destas partições, o número de servidores é reduzido, a fim de que a política de disponibilidade continue sendo mantida.

3.6.6 Replicação e Switch-Over

As atividades de gerenciamento de disponibilidade são diretamente relacionadas com a *sincronização* de informações. Isto ocorre pois como estamos trabalhando em um grupo de máquinas

independentes, sempre precisamos sincronizar alguma informação entre estas máquinas. Muitas vezes esta informação é algum estado interno dos mecanismos de gerenciamento do cluster. Entretanto, ela também podem ser a própria informação que é utilizada pelo serviço no qual desejamos gerenciar a disponibilidade. Um exemplo desta situação é a sincronização das informações armazenadas em um sistema de arquivos entre duas máquinas. O objetivo é aumentar a disponibilidade do servidor de arquivos, possibilitando que o servidor de reserva assuma em caso de falha do servidor principal. Para isso, o servidor de reserva deve possuir o mesmo estado do servidor primário; qualquer alteração nos arquivos do servidor primário deve ser replicada para o servidor de reserva.

Em caso de falha de um servidor, o serviço que ele provê deve ser recuperado em outro servidor. A fim de tornar possível o failover de serviços, existem dois mecanismos genéricos para sincronizar as informações em um cluster [75]:

- replicação (clusters sem compartilhamento - *shared-nothing clusters*): servidores de reserva devem manter sua própria cópia do estado.
- switch-over (clusters que compartilham informações - *shared-data cluster*): os servidores de reserva possuem acesso aos dispositivos de armazenamento utilizados pelo primário.

A tabela 3.2 apresenta as principais diferenças entre as técnicas de replicação e de switch-over.

Replicação	**Switch-Over**
mais fácil de ser adicionada a uma única máquina existente	mais difícil de ser adicionado, pois implica em modificar o cabeamento
mais fácil de se configurar	mais difícil de ser configurado
pode utilizar qualquer adaptadora e controladora de E/S antiga	necessita de dispositivos de E/S especializados
pode utilizar unidades de armazenamento simples	deve utilizar repositórios de armazenamento mais robustos, como RAID
cópia de dados de 1 para n máquinas é difícil	cópia de dados de 1 para n máquinas é possível desde que a interconexão permita
necessita de uma outra cópia da unidade de armazenamento	usa somente uma cópia da unidade de armazenamento
há um overhead na CPU em operação normal (toda alteração deve ser sincronizada com a outra unidade de armazenamento)	nenhum overhead em operação normal
para ocorrer failback é necessário uma cópia adicional das informações	nenhuma cópia necessária para failback

Tabela 3.2: Sincronização através de Replicação e Switch-Over

Capítulo 4

Serviços de Pertinência

"An algorithm must be seen to be believed."

D.E. Knuth

4.1 Introdução

Após descrevermos vários aspectos de clusters e tolerância a falhas, já possuímos a base e vocabulário suficientes para entrarmos no tema principal deste estudo.

Já vimos anteriormente que um cluster é formado por nós, e que o serviço de pertinência é responsável por fornecer a cada nó do cluster a visão do grupo a qual ele faz parte. Apesar desta definição já fornecer uma idéia do papel do serviço de pertinência, precisamos definir melhor os conceitos envolvidos e a importância do serviço para o grupo de computadores.

Veremos que o problema da pertinência tem uma definição básica muito imediata. Por isso, existem diversas implementações que variam nas propriedades que são garantidas dentro do cluster. Ainda, o modelo de sistema distribuído usado no cluster influencia totalmente o serviço de pertinência. Em algumas situações, é impossível resolver do problema de pertinência em grupos.

Por fim, listaremos alguns trabalhos relacionados a este estudo, descrevendo suas respectivas importâncias para a compreensão desta área.

4.2 Conceitos Básicos e Definições

Para a compreensão completa do tema de pertinência a grupos, vários conceitos precisam ser esclarecidos. A fim de podermos prosseguir no estudo deste assunto, esta seção irá reunir e explicar todas estas definições e idéias.

4.2.1 Grupos de Computadores

Como vimos no começo de nosso estudo, um sistema distribuído é constituído a partir de um grupo de computadores. Computadores podem se organizar em grupos para realizar várias tarefas, tais como aumentar a disponibilidade de seus serviços ou aumentar o poder de processamento. Na realidade, quanto mais servidores um grupo contiver, maior será sua disponibilidade e capacidade de servir paralelamente requisições de serviços. Entretanto, quanto mais membros um grupo possuir, maiores serão os custos de comunicação e sincronização necessários entre estes membros [38].

Existem duas abordagens diferentes para a formação um grupo de computadores. Estes modelos de formação de grupos são [27]:

1. **Modelo Estático**

   Esta abordagem é a mais usada, e se preocupa em formar grupos de processos a partir de um subconjunto de uma coleção maximal[1] estática de processos. Em tal modelo, o nome de um processo não é normalmente seu identificador (`pid`); este seria derivado a partir do nome do recurso que ele gerencia. Por exemplo, se existem três réplicas idênticas do banco de dados, poderíamos dizer que o processo que gerencia a primeira réplica é chamado de a, o segundo de b, e o terceiro de c, e reusar estes nomes mesmo se alguma réplica falhar e precisar ser reinicializada. Desta maneira, poderíamos modelar este sistema como um grupo fixo ($\{a, b, c\}$), mas que opera com uma variação dinâmica do conjunto de membros.

2. **Modelo Dinâmico**

   Em um modelo dinâmico de grupos, processos são criados, executados por um período de tempo, e então finalizados. Cada processo possui um nome único que jamais será reutilizado. Em tal modelo, o sistema é definido como o conjunto de processos que estão operacionais em certo momento. Um grupo de processos dinâmico começaria sua execução assim que o grupo fosse formado, e aí evoluiria conforme processos entrassem ou saíssem do grupo. O ponto importante que o difere do modelo estático é que os possíveis membros do grupo não fazem parte de um conjunto estático; em tese, qualquer processo pode vir a fazer parte deste conjunto.

[1] Suponha que S seja uma coleção de conjuntos. Um elemento X de S é maximal se não existe Y em S que seja um *super*conjunto próprio de X. Em outras palavras, X é maximal se X não é subconjunto próprio de algum outro elemento de S.

4.2.2 Serviço de Pertinência (*Membership Service*)

Já compreendemos o que é um grupo de computadores, e que seus possíveis membros podem tanto ser determinados estática como dinamicamente. Na seção 3.4.2 apresentamos o *Serviço de Pertinência*, que é o serviço responsável por fornecer esta identidade de grupo aos membros desta formação, fornecendo capacidades de formação e gerenciamento do grupo. Iremos agora detalhar este serviço, mostrando suas possíveis diferenças e particularidades.

De modo geral, definimos o Serviço de Pertinência da seguinte maneira: seja P um grupo de computadores, nos quais rodam servidores de pertinência[2]. O serviço fornecido por um servidor de pertinência $p \in P$ é o seguinte: quando um processo cliente local c declara interesse em conhecer a lista de membros, p fornece a c a lista de membros do grupo, isto é, a lista de computadores que pertencem ao grupo. Então, o servidor de pertinência notifica c de qualquer mudança subseqüente que vier a ocorrer no grupo [37]. Esta mudanças no grupo serão causadas por falhas de computadores, ou pela inclusão de novos membros no grupo. Estas notificações são enviadas até que c declare que não está mais interessado neste serviço, ou que a falha de c seja reportada a p pelo sistema operacional local.

A definição acima fornece uma boa idéia do que é um serviço de pertinência. Entretanto, a definição ainda é um tanto aberta, o que permite que vários comportamentos diferentes ocorram para um serviço de pertinência. Desta maneira, cada especificação de um protocolo de pertinência deve descrever seus requisitos, de forma a fechar totalmente o comportamento do serviço. A seção 4.4 exibe várias possibilidades para um serviço de pertinência, e é uma boa fonte de opções para especificações de serviços de pertinência. Ainda, apresentaremos em detalhes na seção 6.2.2 a especificação de um protocolo que será implementado neste projeto.

Tipos de Pertinência

Até o momento, consideramos que os grupos são formados exclusivamente por computadores, chamados de nós. Entretanto, a *pertinência* a um certo grupo pode ser observada por duas perspectivas diferentes. Podemos estabelecer um grupo a partir de um conjunto de nós que trabalham juntos, ou podemos formar um grupo de processos - que rodam em nós - para serem executados em cooperação. Desta maneira, existem as seguintes abordagens para o Serviço de Pertinência:

1. **Pertinência de Nós ou Processadores**

   Quando desejamos formar um grupo de máquinas para estabelecer um cluster, necessitamos de um mecanismo para organizá-las como um grupo. Neste conjunto, cada elemento será

[2] Estes servidores de pertinência são processos que fornecem o serviço de pertinência.

uma máquina, e ela fornecerá capacidades de processamento para o grupo. Assim, podemos compreender que estamos estabelecendo um grupo de nós (ou processadores), e estes nós possibilitarão a execução de processos no grupo. No contexto da alta disponibilidade, esta formação permite o aumento da disponibilidade destes processos, pois o grupo como um todo passa a gerenciar os recursos disponíveis.

Assim, o *Serviço de Pertinência de Nós ou Processadores* é um serviço de pertinência que provê a identidade de um grupo a conjuntos de *nós* (ou *processadores*). Um serviço de pertinência de nós ajuda a resolver dois outros problemas [37]:

- Disponibilidade de Serviços

  Este é o problema clássico da disponibilidade, e o definimos como se segue: escolha k processadores de um grupo de n processadores ($n \geq k$) para rodar um conjunto ordenado de k serviços $S = (s_1, s_2, ..., s_k)$. Estes serviços devem executar enquanto existir pelo menos k processadores funcionando corretamente no grupo.

  O Serviço de Pertinência de Processadores permite que o problema acima seja resolvido facilmente: se o processador p se une ao grupo de tamanho $j < k$, cujos antigos membros já forneciam o conjunto de serviços S', p inicia o serviço $\min(S - S')$; caso contrário, se $j \geq k$, p não faz nada. Se a falha do processador p é detectada quando um novo grupo g é criado, então, se p não estava rodando nenhum serviço previamente, nada deve ser feito; caso contrário, se p estava rodando o serviço s_i, e o tamanho j de g é maior do que k, o membro mais antigo de g que não estiver executando um serviço deve iniciar s_i.[3]

- Restrição da Comunicação aos Membros de um Grupo

  Considere um grupo g de processadores que devem garantir a disponibilidade de certo serviço s, tal que s deve ser fornecido corretamente somente enquanto existir somente um servidor para s. Suponha que, no momento T, um membro p de g deve inicializar s localmente. Se uma falha de performance impede p de inicializar s no momento, os membros sobreviventes de g formarão um novo grupo g' em $T + D$, excluindo p desta formação. Por questões de disponibilidade, s deve ser inicializado no novo grupo g' por outro processador q. Se p não transformar sua falha de performance em uma falha de colapso, p pode difundir uma mensagem depois de $T + D$ indicando que finalmente o serviço s foi inicializado. Os membros de g' devem ignorar esta mensagem para evitar confusão.

  Para resolver problemas como este, é necessário definir claramente o *escopo* da comunicação de grupo. Exceto pelas mensagens de união ao grupo, a comunicação deve ser

[3] Este exemplo ilustra brevemente como o Serviço de Pertinência de Processadores possibilita um incremento da disponibilidade de serviços. Entretanto, ele não é uma descrição formal de algoritmos ou protocolos de tolerância a falhas.

restrita somente a membros do grupo. Se todos os grupos que podem existir no tempo são unicamente definidos por identificadores de grupos distintos, estes identificadores podem ser usados para filtrar as mensagens.

2. **Pertinência de Servidores ou Processos**

   Um serviço de pertinência de *servidores* ou *processos* estabelece grupos, onde os elementos integrantes destes conjuntos são processos, os quais rodam sobre nós. A vantagem desta abstração de grupos de processos é que ela simplifica o desenvolvimento de aplicações distribuídas confiáveis.

   O objetivo de um serviço de pertinência de servidores é permitir que cada membro z de um time de servidores replicados Z conheça os membros deste grupo que operam corretamente [37]. O serviço de pertinência de processos provê a z a identidade de todos os outros membros corretos de Z, e notifica-o de subseqüentes alterações na lista de membros. Neste contexto, um servidor z de um time Z roda em um determinado processador p.

   Há vários casos em que um serviço de pertinência de processos é útil [30]. Várias aplicações usam inerentemente grupos de processos, como sistemas corporativos e de corretagem, onde a confiabilidade de se publicar e de receber mensagens utiliza um grupo de processos para prover a replicação necessária. Ainda, a construção de aplicações distribuídas fica muito simplificada quando usamos a abstração de grupos de processos, pois esta abstração já fornece todos os mecanismos para a manipulação do grupo.

   Devido à maior complexidade envolvida em grupos de processos, geralmente especificamos um serviço de pertinência para nós, e então derivamos a especificação correspondente para processos. Cristian descreve em [37] como um serviço de pertinência de processos pode ser construído a partir de um serviço de pertinência de processadores.

4.2.3 Visão

Tradicionalmente, o termo *visão* é usado para se referir à lista de membros corrente do grupo [67]. É muito atrativo definir uma visão simplesmente como o conjunto de processos, mas ao refletirmos um pouco mais, perceberemos que esta definição é inadequada.

Considere a seguinte situação: suponha que o conjunto dos membros de um grupo seja originalmente $\{p, q\}$. Então, um processo r é adicionado ao grupo, e os membros se tornam $\{p, q, r\}$. Em seguida, o processo r é removido, de maneira que o conjunto dos membros do grupo volta a ser $\{p, q\}$. É interessante distinguirmos a visão inicial desta visão final, apesar de seus conjuntos serem considerados idênticos.

Este exemplo nos leva a definir visão como identificadores em um conjunto. Assumimos ainda a existência de uma função que pode "decodificar" cada identificador v, ao mapeá-lo ao

conjunto de processos que constituem a lista de membros quando a visão é representada por v. Formalmente, uma *visão* é um identificador k, tal que $k \in \Psi$, onde $\Psi = \mathbb{Z}^+ \cup \{nil, fin\}$. A visão especial nil (visão nula) representa que nenhuma visão foi instalada e mapeia ao conjunto vazio. A visão especial fin (a visão final) representa o fim da participação de um processo no grupo, e também mapeia para o conjunto vazio. Dizemos que uma visão v *contém* o processo p quando a visão v mapeia para um conjunto de processos no qual p é um membro.

4.2.4 Particionamentos

Durante o decorrer de sua existência, os clusters podem ser particionados em subconjuntos disjuntos devido a várias razões, como:

- Falhas na rede que interliga os nós;
- Quantidade excessiva de mensagens;
- Falhas de performance de processos.

Cada um destes subconjuntos é referenciado informalmente como uma *partição* [35]. Em linhas gerais, uma partição é o conjunto maximal de processos que podem comunicar uns com os outros [67].

A figura 4.1 exibe um cluster no momento anterior a uma falha de comunicação, a qual acarreta em uma divisão deste cluster em dois subconjuntos disjuntos. O momento posterior ao particionamento é ilustrado na figura 4.2. Repare que após o particionamento, ocorre um failover das aplicações de maneira a fazer com que estas continuem acessando os repositórios de discos que acessavam antes da falha de comunicação.

Por ser uma complexidade a mais nos projetos de sistemas distribuídos, na prática um serviço de pertinência pode ser projetado para lidar ou não com particionamentos. Caso ele seja projetado para não lidar com este cenário, ele simplesmente ignorará a possibilidade de ocorrência de particionamentos; a formação física do cluster será a responsável por "impedir" ou "controlar" a ocorrência de particionamentos. Caso o serviço considere os particionamentos, duas possíveis abordagens para o cluster são possíveis [32]:

1. **Clusters Orientados a Quórum (*Quorate Clusters*)**

   Esta é a abordagem mais encontrada, e a mencionada até agora neste texto. Esta abordagem define que, quando um cluster é particionado em sub-clusters, somente um destes grupos prosseguirá com a execução. Este sub-cluster será chamado de *partição primária*, e

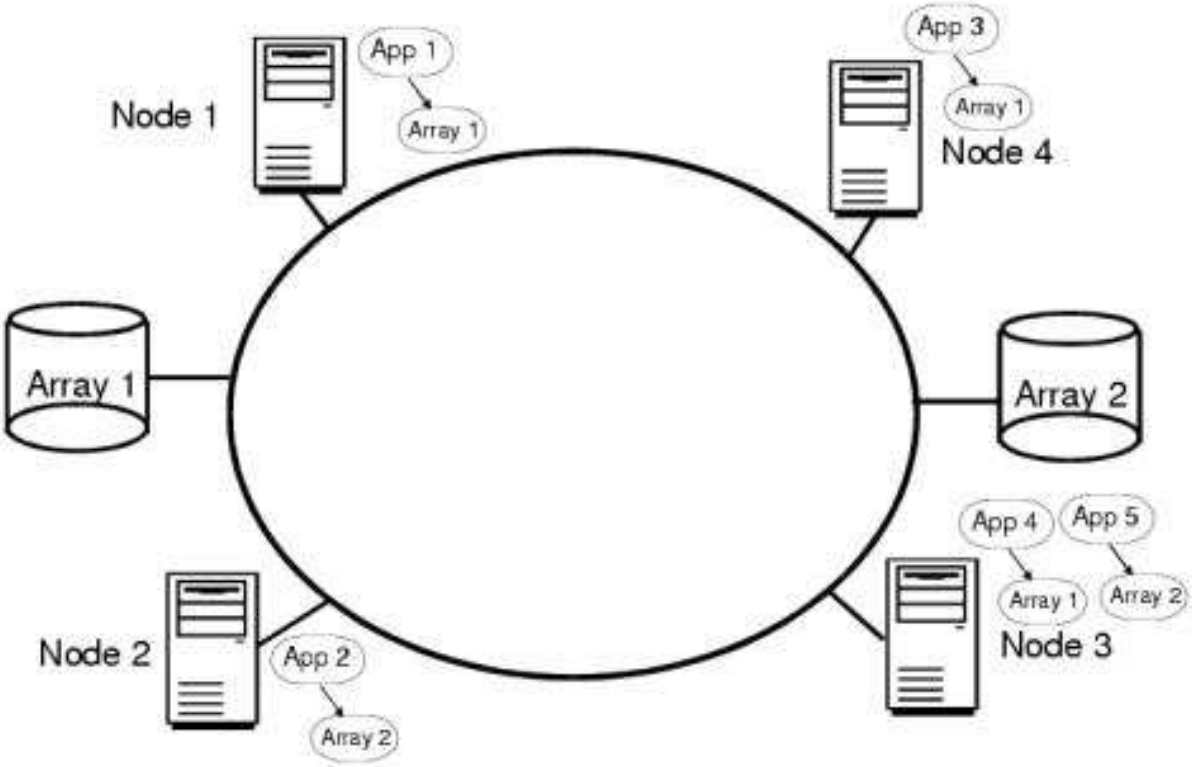

Figura 4.1: Momento anterior ao particionamento no cluster

pode ser determinado de várias formas[4].

Clusters Orientados a Quórum são centralmente controlados. Todas as ações direcionados pelo cluster são baseadas em sua visão da lista de membros. Geralmente, estas ações são acordadas entre todos os membros do grupo. Ainda, somente uma única entidade do cluster pode existir em cada momento.

Serviços de pertinência que lidam com particionamentos desta maneira são classificados como *serviços de pertinência de partição primária* (*primary partition membership services*) [67]. Estes serviços garantem que, em todo momento, há somente uma única visão do grupo, restringindo mudanças à lista de membros a somente esta partição.

2. **Clusters Orientados a Recursos (*Resource Driven Clusters*)**

   Clusters Orientados a Recursos são mais caóticos, pois não há um gerenciador central no grupo. As ações sobre um certo recurso são controladas pelo membro do cluster que "possua" (ou seja, hospede) o recurso. Desta maneira, cada nó não precisa do consentimento

[4]A forma mais comum de se determinar a partição primária diz que ela será o sub-cluster com a maioria dos nós restantes do grupo original.

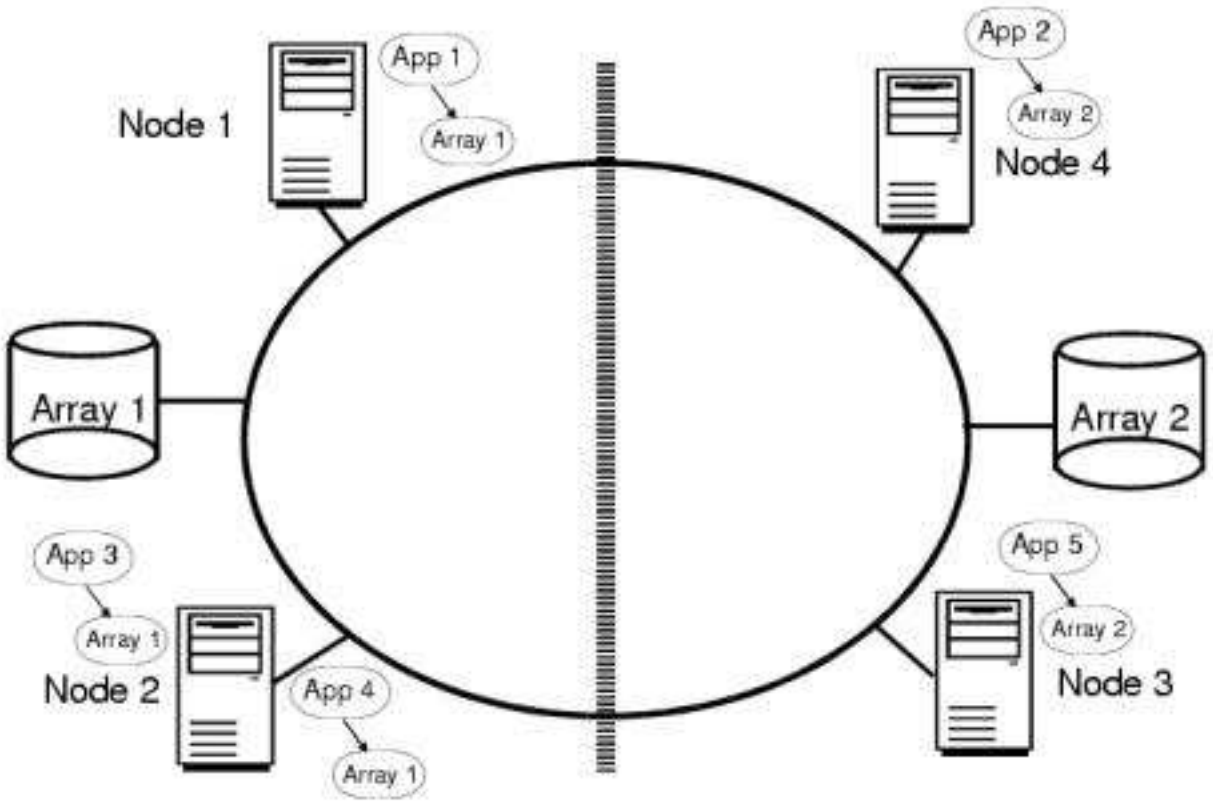

Figura 4.2: Momento posterior ao particionamento no cluster

de outros membros para executar ações sobre recursos que ele hospeda.

A inexistência de um gerenciador central significa que não existe a idéia de visões como números identificadores de instância. Um cluster pode ser formado a partir de comunicações parciais. Múltiplos recursos podem estar hospedados em múltiplos nós, que poderiam tomar múltiplas ações simultaneamente. Nesta formação, sub-clusters múltiplos e independentes podem ser formados.

Serviços de pertinência que permitem este tipo de formação são classificados como *serviços de pertinência particionáveis* (*partitionable membership services*) [67]. Estes serviços permitem a existência de visões múltiplas e independentes em diferentes partições, durante o mesmo momento lógico.

Apesar desta abordagem não ser muito utilizada, Dolev, Malki e Strong argumentam em [46] que o serviço de pertinência não deve impedir a execução de aplicações devido a particionamentos. Segundo os autores, o serviço faz parte de um sistema de comunicação multicast, e deve fornecer às aplicações informações sobre a situação do sistema. Eles

colocam várias vantagens no uso desta abordagem:

- Há várias maneiras que aplicações podem se beneficiar de operações particionadas. Por exemplo, um sistema que contém quatro máquinas A, B, C, D pode trocar entre as configurações $(\{A, B\}, \{C, D\})$ e $(\{A, C\}, \{B, D\})$. Neste caso, qualquer informação trocada entre A e B dentro da primeira configuração poderia atingir todo o sistema durante a segunda configuração. Operações particionadas poderiam difundir informações gradualmente, enquanto que impedir operações em ambas partições impediria qualquer progresso no sistema.
- Algumas aplicações podem ser executadas com garantias de consistências mais fracas, de maneira que as ações podem ser realizadas dentro da partição desconectada. Um bom exemplo é uma rede de caixas automáticos de um banco. As ações realizadas em uma máquina, como o saque de certa quantia de dinheiro, devem também ser realizadas nos computadores centrais do banco. Se a máquina estiver desconectada do resto do grupo, ela poderia continuar permitindo saques, limitados a certa quantia estática, independentemente do saldo da conta[5].
- Boas semânticas de particionamento evitam inconsistências que surgem quando não há definição do comportamento de execuções particionadas. Desta maneira, mesmo que somente uma partição possa continuar com sua execução, é importante definir o comportamento do sistema nesta situação.
- Há situações onde o sistema não consegue manter uma partição primária, como quando uma maioria das máquinas falha, ou quando o sistema se divide em múltiplas pequenas partições.
- No momento de sua inicialização, um sistema distribuído pode ter várias partições desconectadas. Cada uma destas partições contém um nó que está sendo inicializado. Posteriormente, o grupo de todos os nós será formado pela união dos grupos *individuais* de nós. Desta maneira, é natural entender que não há partição primária durante a união destes grupos.

Ao compararmos esta abordagem com clusters orientados a quórum, percebemos que ela possui vantagens e desvantagens [32]:

Vantagens:

- Mais fáceis de serem projetados e construídos, pois não há a necessidade de se construir uma camada de controle central[6].

[5] Segundo os autores, esta abordagem é utilizada na prática.

[6] Simplicidade é uma característica muito desejável em alta disponibilidade, pois reduz a possibilidade de ocorrência de erros.

- Melhor escalabilidade em grandes clusters com um grande número de recursos.
- Sobrevivem melhor a desastres, pois a formação de múltiplos sub-clusters geralmente provê melhores características de recuperação.

Desvantagens:

- Mais difícil de ser analisado, pois seu comportamento caótico torna difícil a prova de propriedades. Esta característica torna esta abordagem desinteressante para a comunidade acadêmica.
- Características multi-thread de failover podem causar problemas aos recursos do sistema operacional.
- Uma visão única do cluster é difícil de ser obtida, o que torna a administração do sistema muito complicada.

Cluster Orientados a Recursos permitem uma seqüência mais rica de cenários de recuperação. Um interessante esquema de recuperação de recursos após falhas é o uso de *funções de utilidade* (*utility functions*). Considere uma situação onde a recuperação de certo recurso pode ser realizada, mas a formação resultante do sub-cluster seria inútil. Esta situação poderia ser a recuperação de um servidor HTTP em um sub-cluster que não consegue se comunicar com o roteador: os serviços deste servidor não seriam visíveis para usuários externos. Em um particionamento, certos sub-clusters podem ser mais úteis para recuperar certos recursos. Desta maneira, é necessária a construção de uma medida de utilidade[7], que chamamos de *função de utilidade* da formação do sub-cluster. Mais informações sobre o conceito de funções de utilidade podem ser encontradas em [32].

4.3 Motivação

Como vimos acima, o problema da *pertinência em grupos* é fundamental para várias áreas. Reuniremos brevemente nesta seção os principais fatores que motivam seu estudo.

4.3.1 Sistemas Distribuídos

O problema da pertinência em grupos é um problema tão fundamental em computação distribuída quanto os problemas de roteamento, sincronização de relógios e broadcast atômico. Uma vez resolvido, ele permite soluções fáceis a outros problemas importantes encontrados no projeto de aplicações distribuídas tolerantes a falhas [37].

[7] Geralmente, *utilidade* é uma função definida pelo usuário ou aplicação.

A maioria dos algoritmos distribuídos publicados são baseados na existência de um grupo fixo de processos ativos que se comunicam através de mensagens para atingir um determinado objetivo. Entretanto, em sistemas reais, tais grupos são dinâmicos, diminuindo de tamanho com a ocorrência de falhas, e aumentando com a recuperação dos serviços de comunicação e processos [43].

4.3.2 Alta Disponibilidade

Ao projetarmos um serviço computacional que deve permanecer disponível mesmo na ocorrência de falhas, uma idéia chave é replicar o estado do serviço entre vários servidores rodando em processadores distintos [37]. O estado do sistema consiste tipicamente de:

- lista de membros do grupo de servidores, ou seja, o conjunto de todos os servidores que funcionam corretamente e cooperam para fornecer o serviço.
- informações de estado específicas do serviço, como a fila de requisições aceitas mas ainda não completadas, o estado de recursos físicos usados para prover o serviço, e a designação de trabalho atual dos vários servidores.

Desta maneira, a identidade de um grupo de servidores é fundamental para se atingir este objetivo.

4.3.3 Comunicação de Grupos (*Group Communication*)

Para implementar serviços distribuídos altamente disponíveis, os processadores são estruturados em times, que representam conjuntos de processadores capazes de implementar um certo serviço. Há duas abordagens para gerenciar a comunicação entre os membros de um time que implementam um serviço distribuído [43]:

Independente de Histórico (*History Independent*)

Na abordagem *independente de histórico*, quando um membro p de um time interpreta a requisição de um serviço que demanda que ele comunique com os outros membros, p tenta comunicar com estes membros independentemente do sucesso ou fracasso de comunicações recentes com estes membros. Um bom exemplo de tal abordagem é o serviço de votação distribuído.

Dependente de Histórico (*History Dependent*)

Na abordagem *dependente do histórico*, os membros do time adaptam-se às alterações das condições de comunicação, de maneira a tentarem se comunicar somente com os membros

com os quais tiveram sucesso em uma comunicação recente. A idéia é deixar os membros de um time formar grupos de trabalho e de comunicação dinâmicos. Assim, a maior complicação introduzida por este modelo é a necessidade de um *serviço de pertinência de grupos*.

Desta maneira, o serviço de pertinência a grupos é fundamental para sistemas de comunicação de grupo dependentes do histórico. Assim, podemos concluir que ele é um pré-requisito para a construção de outros sistemas distribuídos.

4.4 Propriedades de Serviços de Pertinência

Existem diferentes propriedades que um serviço de pertinência pode garantir. De acordo com o cluster em questão, e com o serviços que ele fornecerá aos seus usuários, certas propriedades podem não ser necessárias. Assim, a compreensão destas propriedades é fundamental para decidirmos qual o algoritmo necessário para certo cenário. Hiltunem e Schlichting apresentam uma interessante descrição destas propriedades em [57], onde são utilizados grafos de ordenação de mensagens para indicar o que a propriedade está realizando. Um resumo da notação utilizada é exibido na figura 4.3.

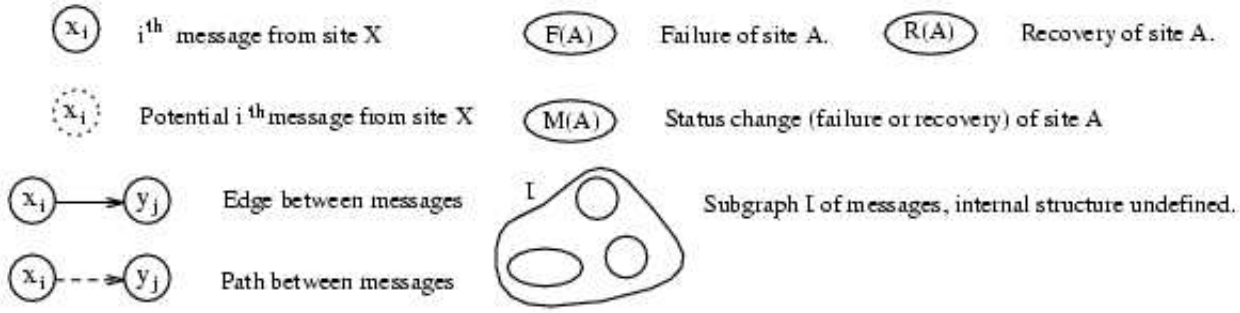

Figura 4.3: Notação de Grafos de Ordenação de Mensagens

Explicaremos estas propriedades nas seções abaixo.

4.4.1 Precisão e Vivacidade

Precisão (*Accuracy*)

Um serviço de pertinência é *preciso* se a falha/recuperação (ou inicialização) de um certo nó é informada à aplicação somente se ela realmente tenha ocorrido.

Vivacidade (Liveness)

Um serviço de pertinência é *vivo* se toda falha/recuperação dos nós é detectada e informada. Um tipo especial desta propriedade é *liveness delimitada*, onde toda falha/recuperação é detectada dentro de um intervalo de tempo definido.

4.4.2 Concordância (*Agreement*)

A propriedade de *concordância* demanda que toda mensagem de alteração da lista de membros entregue à aplicação em um certo nó também seja entregue nos outros nós. A figura 4.4 ilustra este conceito.

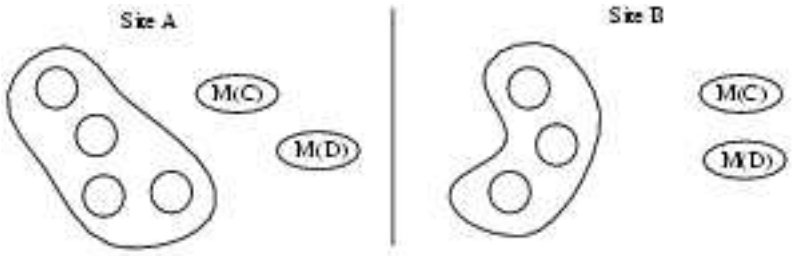

Figura 4.4: Propriedade de Concordância

4.4.3 Propriedades de Ordenação

As seguintes propriedades de ordenação podem ser satisfeitas por um serviço de pertinência:

FIFO

A ordenação FIFO demanda que mensagens de alteração do estado de um certo nó sejam entregues à aplicação em todos os nós na mesma ordem. A figura 4.5 ilustra esta ordem.

Ordenação Total

A ordenação total demanda que mensagens de alteração da lista de membros sejam entregues à aplicação em todos os nós na mesma ordem (figura 4.6). Para esta propriedade ser atingida, ela necessita que a entrega de mensagens seja ordenada e confiável.

Esta propriedade é útil em sistemas que classificam processos ou nós de acordo com a duração de suas pertinências ao grupo. De maneira que a classificação observada nos diversos nós seja a

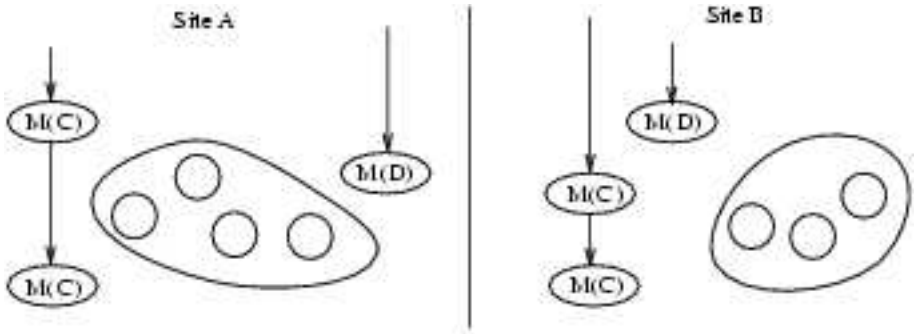

Figura 4.5: Propriedade de Ordenação FIFO

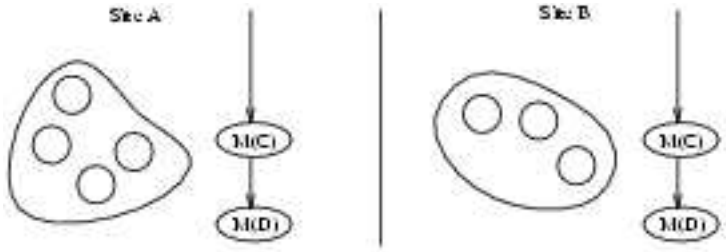

Figura 4.6: Propriedade de Ordenação Total

mesma, as mensagens de alteração na lista de membros devem ser processadas na mesma ordem total em todos os nós.

Concordância na Última Mensagem (*Agreement on Last Message*)

A propriedade de *concordância na última mensagem* demanda que uma mensagem "final", enviada por um nó que falhou, seja entregue à aplicação em todos os nós antes da própria mensagem de alteração dos membros, causada pela falha deste nó. Possíveis mensagens subseqüentes do nó defeituoso serão entregues após a alteração da lista de membros. Portanto, estas mensagens serão descartadas por não se originarem de um nó pertencente ao cluster. A figura 4.7 ilustra esta situação.

Esta propriedade é útil em aplicações onde um estado distribuído consistente deve ser mantido. Em tais situações, a mensagem final pode causar uma mudança de estado, de maneira a garantir que a alteração seja aplicada ou a todos ou a nenhum nó.

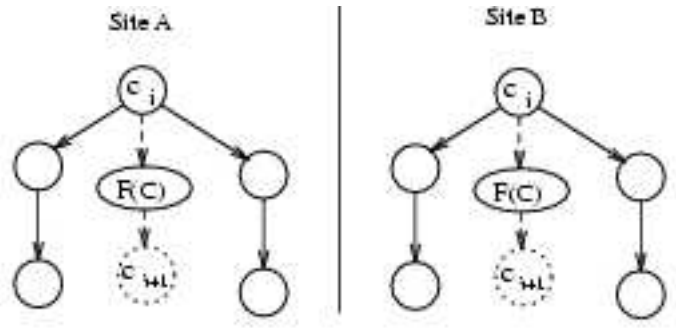

Figura 4.7: Propriedade de Concordância na Última Mensagem

Concordância nos Sucessores (*Agreement on Successors*)

A propriedade de *concordância nos sucessores* demanda que todos os nós entreguem uma mensagem de alteração dos membros antes que qualquer outra mensagem, em um conjunto sucessor definido (figura 4.8). Por exemplo, se o nó C está se recuperando, então o conjunto sucessor pode ser o corte no grafo de mensagens, após o qual todas as mensagens são consideradas terem chegado após a recuperação de C.

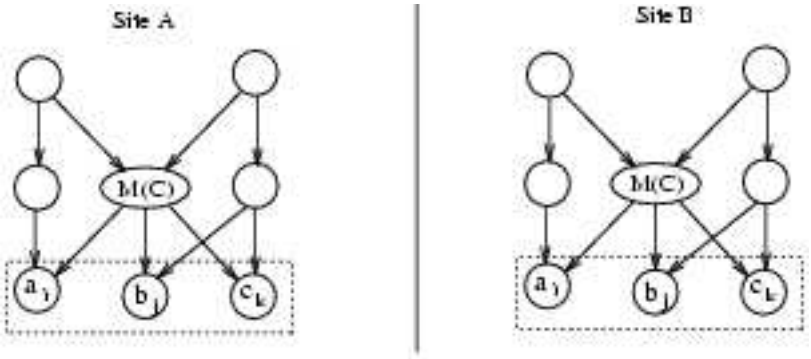

Figura 4.8: Propriedade de Concordância nos Sucessores

Entre outras coisas, esta propriedade é útil para se determinar a *estabilidade das mensagens*: uma mensagem é estável no nó remetente uma vez que ela foi reconhecida por todos os outros nós operacionais. Se m é um sucessor concordado de $R(C)$, então todo nó sabe que m deverá ser reconhecida pelo nó C para ser considerada estável.

Sincronia Virtual (*Virtual Synchrony*)

A propriedade de *Sincronia Virtual* restringe a ordem de entrega de mensagens de aplicação e de mensagens de mudança da lista de membros. Esta restrição se dá de tal modo, que, para a aplicação, os eventos ocorrem simultaneamente, mesmo que estejam na realidade ocorrendo em nós diferentes e em momentos distintos. A sincronia virtual é facilmente explicada no grafo de ordenação, como ilustrado na figura 4.9.

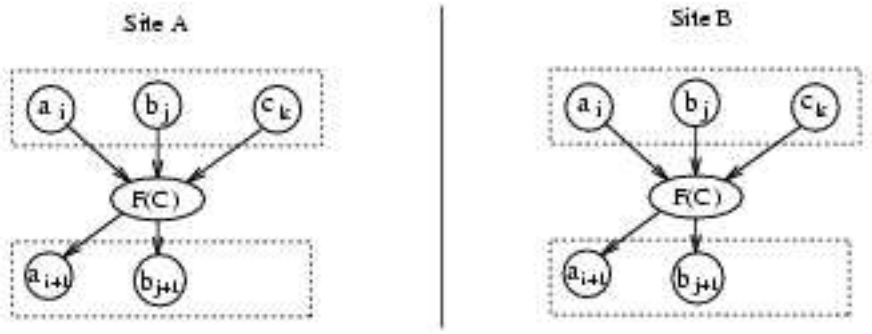

Figura 4.9: Propriedade de Sincronia Virtual

Relativamente ao serviço de pertinência, esta propriedade demanda um acordo entre todos os nós operacionais em uma divisão do fluxo de mensagens, tal que cada mensagem esteja ou em um conjunto de predecessores concordado[8], ou em um conjunto de sucessores concordado. Em outras palavras, sincronia virtual essencialmente cria um corte no fluxo de mensagens, tal que este corte seja comum e acordado entre todos os membros do grupo.

Como discutido amplamente na literatura sobre o Isis [26], a sincronia virtual torna mais fácil a construção de aplicações distribuídas. Mais especificamente, aplicações que podem ser vistas como máquinas de estado replicadas são muito beneficiadas com este modelo. Maiores detalhes sobre o modelo de sincronia virtual serão abordados na seção 4.5.

Sincronia Virtual Estendida (*Extended Virtual Synchrony*)

Uma desvantagem de sincronia virtual é que ela não relaciona a visão dos membros no momento em que uma mensagem é enviada para a coleção de nós que realmente recebem a mensagem. Por exemplo, o nó A poderia enviar a mensagem a_i em um momento quando os membros do grupo são $\{A, B, C\}$, mas, antes que ela seja recebida, o nó D se junta ao grupo. A *Sincronia Virtual Estendida* garante que todas as mensagens enviadas sob um consenso de membros serão entregues antes de qualquer mensagem de alteração dos membros, conforme ilustrado na figura 4.10.

[8]Conceito similar ao da propriedade anterior.

Os círculos sombreados representam mensagens que foram enviadas antes do recebimento da mensagem de alteração dos membros $M(C)$.

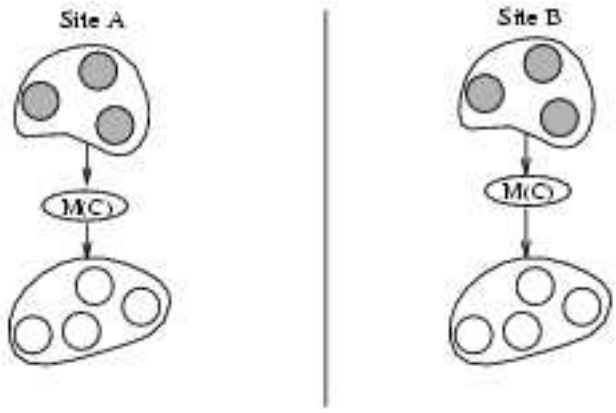

Figura 4.10: Propriedade de Sincronia Virtual Estendida

4.4.4 Propriedades de Alterações Delimitadas

Estas propriedades aplicam restrições no momento em que as alterações na lista de membros devem ser efetuadas.

Sincronia Externa (*External Synchrony*)

A propriedade de *Sincronia Externa* garante que se um nó entrega uma certa mensagem de alteração de membros, todos os outros já entregaram ou estão em um estado de transição no qual a entrega é iminente. Tal propriedade indica que os nós movem para um novo estado de membros de maneira coordenada. Esta idéia é relacionada ao conceito de barreiras de sincronização em programas paralelos, na qual a execução em todos os nós deve alcançar a barreira - isto é, alcançar o estado de transição - antes que qualquer nó possa prosseguir - isto é, realizar a alteração na lista de membros. A figura 4.11 ilustra esta característica. Baseado na definição, existe um tempo global t, tal que todos os nós entregam a mensagem de preparação antes de t, e a mensagem de alteração dos membros após t.

A *Sincronia Externa* é útil em várias situações, especialmente nos casos onde uma aplicação deve acessar um dispositivo externo ou enviar uma mensagem a um processo fora do grupo.

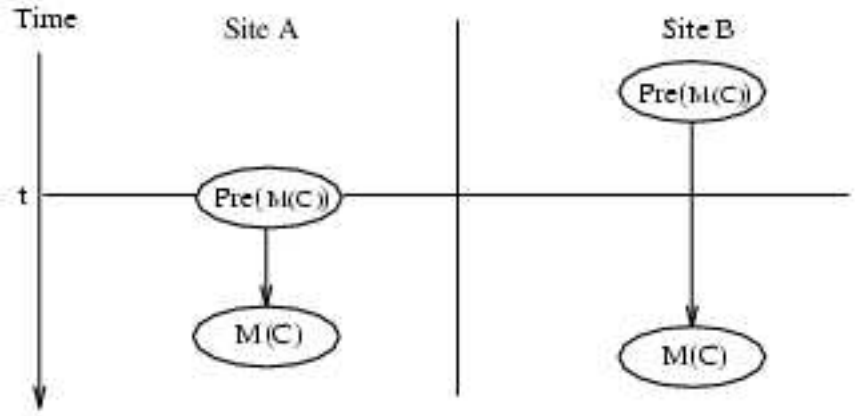

Figura 4.11: Propriedade de Sincronia Externa

Sincronia Delimitada por Tempo (*Timebound Synchrony*)

A *Sincronia Delimitada por Tempo* é uma propriedade de serviços de pertinência em sistemas síncronos, na qual todo nó entrega uma certa mensagem de alteração dos membros dentro de um certo intervalo de tempo real. A propriedade tem a mesma aplicação geral de sincronia externa, mas reduz o overhead de sincronização, diminuindo a janela na qual os membros não são idênticos em todos os nós. Ainda, tal propriedade é importante em sistemas de tempo real, de maneira que o sistema possa responder de uma maneira previsível e conveniente a eventos externos. A figura 4.12 ilustra esta propriedade. Na figura, linhas tracejadas indicam um momento no tempo real quando a mensagem de alteração dos membros é entregue à aplicação.

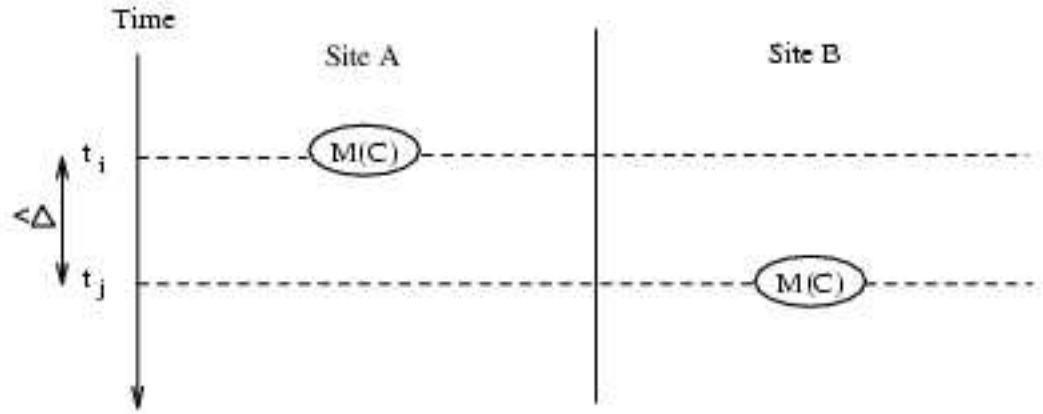

Figura 4.12: Propriedade de Sincronia Delimitada por Tempo

4.4.5 Propriedades de Inicialização e Recuperação

As seguintes propriedades de inicialização e recuperação podem ser oferecidas por um serviço de pertinência:

Inicialização

Duas abordagens genéricas foram desenvolvidas para coordenar a *inicialização* de um cluster:

- inicialização coletiva: a lista de membros inicial é conhecida previamente, com todos os nós inicializando aproximadamente ao mesmo tempo.
- inicialização individual: cada nó é inicializado com uma lista de membros consistindo somente de si mesmo. Conforme os nós tomam conhecimento uns dos outros, um grupo maior é formado pela junção das listas de membros individuais.

A inicialização está fortemente relacionada à maneira que particionamentos na rede são tratados. Portanto, mais informações sobre o processo de inicialização serão dadas na seção 4.4.6.

Recuperação

A *recuperação* envolve reinicializar um nó e reintegrá-lo de volta ao grupo. O requisito básico da recuperação de nós é que as informações sobre o grupo contidas no nó sendo recuperado seja reinicializada para um estado válido, que é definido de acordo com as propriedades que estão sendo garantidas. Por exemplo, se nenhuma propriedade de concordância ou ordenação é satisfeita, o nó recuperado pode usar sua última lista de membros que ele possuía antes de falhar, ou uma visão onde só exista ele.

Uma vez que este estado consistente tenha sido atingido, uma mensagem indicando a recuperação do nó é enviada e entregue aos outros nós. Quando esta mensagem de recuperação for entregue, o nó é considerado membro do grupo, e as garantias de ordenação e entrega de mensagens devem ser asseguradas. Por exemplo, se o serviço de pertinência oferece alguma ordenação a respeito das mensagens de aplicação, qualquer mensagem que chegou depois da mensagem de recuperação deve ser entregue no nó recuperado, enquanto que mensagens anteriores à mensagem de recuperação não devem ser entregues neste nó. A figura 4.13 ilustra estes grafos de ordenação. Note que mensagens sombreadas na figura não estão ordenadas em relação a $R(C)$, o que significa que elas podem ou são ser entregues à aplicação em C.

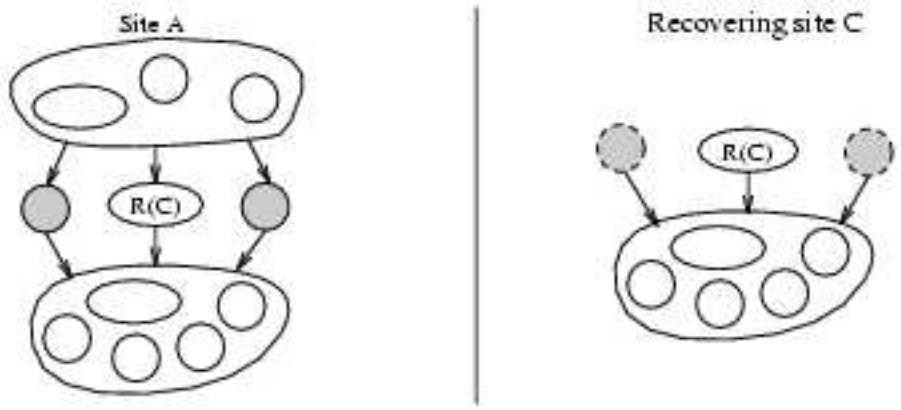

Figura 4.13: Garantias de Ordenação durante uma Recuperação

4.4.6 Tratamento de Particionamentos

Um particionamento na rede ocorre quando um subconjunto de nós em um grupo é incapaz de se comunicar com o restante dos nós. Em sistemas distribuídos, é impossível distinguir um particionamento de uma falha de nós. Portanto, em cada componente desconectado, o serviço de pertinência encaminhará à aplicação avisos de falhas dos outros nós. Quando a comunicação for posteriormente restabelecida, o serviço gerará uma mensagem de união similar à mensagem de recuperação de um nó.

Muitas abordagens existem para este problema, como supor que não existirão particionamentos, ou permitir que somente um sub-cluster prossiga com sua execução. Neste último caso, técnicas como partição primária, votação e tokens são usadas para garantir que somente uma partição possua direito de execução.

Notificação de Falha Aumentada (*Augmented Failure Notification*)

Nas situações em que somente um sub-cluster deve permanecer ativo, deve existir um predicado que é avaliado como verdadeiro em somente um destes grupos. Um exemplo deste predicado é verificar se o número de nós do sub-cluster é maior do que a metade do número total de nós. Para notificar a aplicação que um nó não faz mais parte do grupo majoritário, uma mensagem de notificação de falha é aumentada com um campo extra, indicando se a falha do nó causou uma diminuição do tamanho do grupo ao ponto de ser impossível verificar uma maioria de nós. Note que tal mensagem indica somente a possibilidade de um particionamento, não sua real existência. De maneira igual, uma mensagem de recuperação é aumentada para indicar se a recuperação do nó em questão aumentou o tamanho do grupo para a maioria inicial.

Notificação de Falha Coletiva (*Collective Failure Notification*)

A propriedade de *Notificação de Falha Coletiva* expande a noção de avisos de falhas de nós para um único aviso, no qual é indicada a falha de um grupo de nós. Tal aviso é implementado com uma mensagem única que contém os identificadores de todos os nós que falharam. Ao notificar falhas de maneira coletiva, este mecanismo favorece uma maior eficiência em transições de membros, pois evita um tempo maior de processamento de mensagens. A figura 4.14 ilustra esta propriedade. Nela, nós sombreados representam mensagens de alteração dos membros notificando mudanças que ocorreram anteriormente ao particionamento; a linha tracejada representa o momento em que o particionamento ocorreu. F_P e F_Q representam a notificação de falha coletiva entregue nos subconjuntos P e Q[9], respectivamente.

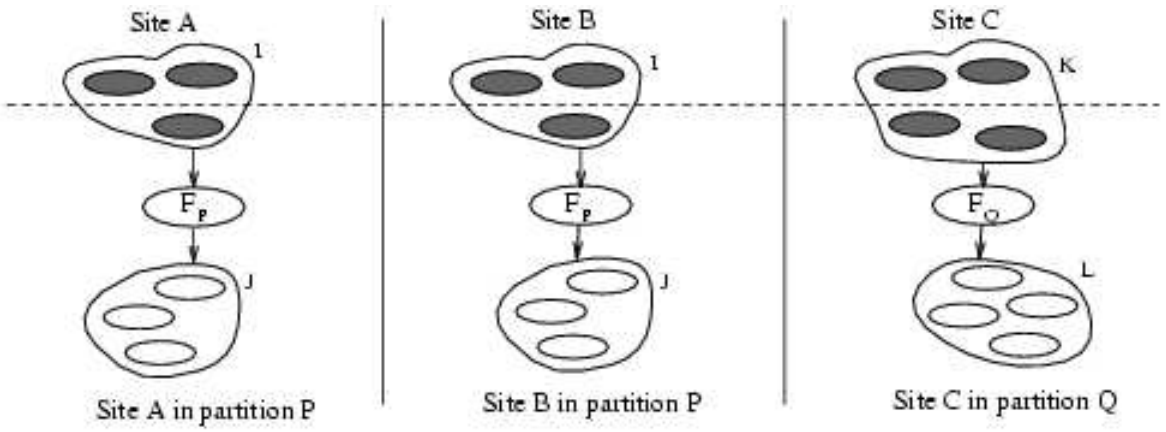

Figura 4.14: Propriedade de Notificação de Falhas Coletivas

União Coletiva Ordenada (*Ordered Collective Join*)

Quando um particionamento é reparado, um algoritmo de pertinência normal integraria cada nó das partições individualmente. A propriedade de *União Coletiva Ordenada* estende esta noção, de maneira a unir dois sub-clusters em um grupo maior coletivamente, e de um modo que seja ordenado consistentemente com as outras mensagens de alterações do grupo. Uma mensagem de união especial é usada para implementar esta ação. Esta mensagem inclui a lista dos membros do novo grupo, e possui uma ordenação total com as outras mensagens de alterações da lista de membros. A figura 4.15 ilustra esta propriedade, onde nós sombreados representam mensagens de alteração dos membros no novo grupo após a união, e o círculo maior no centro representa a mensagem de união. Assim, a grande vantagem desta propriedade é garantir que todos os nós

[9]Isto significa que se $S \in F_P$, então o nó S está incluído na notificação coletiva F_P.

irão observar a união das partições ao mesmo momento lógico, e de maneira coletiva no que se refere à recuperação dos nós.

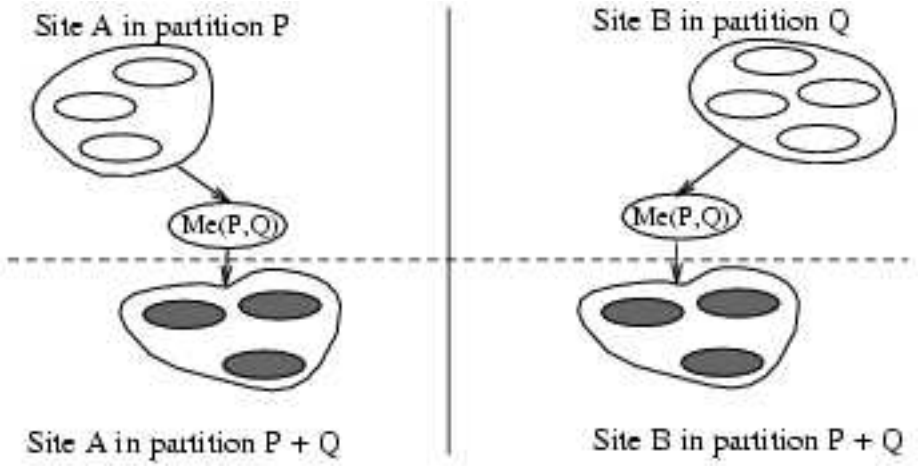

Figura 4.15: Propriedade de União Coletiva Ordenada

Ordenação de Mensagens de Tratamento de Particionamentos

Quando um particionamento ocorre, a evolução das mensagens e dos membros será diferente em cada grupo. Alterações nos membros de um sub-grupo não serão refletidas em outros grupos. Como resultado, quando a notificação coletiva é usada, uma mensagem de falha coletiva cria fluxos de mensagens diferentes em cada grupo, e uma mensagem de união coletiva indica a reunião destes fluxos diferentes em um único fluxo. Esta propriedade distingue *mensagens de tratamento de particionamentos* de mensagens normais de transições de membros.

Sincronia Virtual Estendida é um exemplo de uma propriedade mais forte que pode ser aumentada para incluir o gerenciamento de mensagens de particionamentos. A propriedade de *Sincronia Virtual Estendida com Particionamentos* demanda que mensagens enviadas antes do recebimento da mensagem de união sejam entregues antes desta mensagem, e, de maneira similar, que todas mensagens enviadas antes de receber a notificação de falha coletiva sejam entregues antes desta mensagem. Claramente, esta garantia aplica-se somente dentro de cada partição. A figura 4.16 ilustra esta propriedade para a união de partições.

Entre outras coisas, esta propriedade simplifica o problema de se fundir estados de uma aplicação após um particionamento.

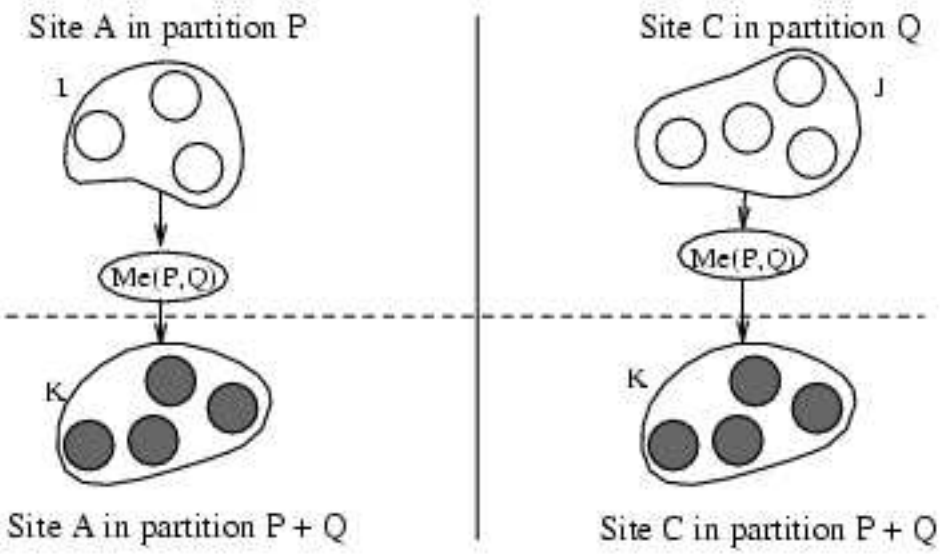

Figura 4.16: Sincronia Virtual Externa com Particionamentos

4.4.7 Relacionamento entre as Propriedades

Apesar de termos definido as propriedades de serviços de pertinência isoladamente, freqüentemente uma propriedade é construída a partir das funcionalidades de outra propriedade, ou é relacionada de alguma maneira a ela. A figura 4.17 fornece um grafo de dependências para as propriedades discutidas nas seções anteriores. Nesta figura, os nós formados com linhas tracejadas representam propriedades implementadas pelo componente de multicast confiável do sistema[10].

O grafo de dependências representa as relações entre as propriedades, e, portanto, todas as combinações possíveis e legítimas destas propriedades. O serviço de pertinência mais simples possível baseia a visão local dos membros somente nas propriedades de liveness e/ou precisão. Serviços mais avançados fornecem a concordância na lista de membros, acrescida de várias outras propriedades, como as de ordenação e tratamento de particionamentos [57].

4.5 O Modelo de Execução de Sincronia Virtual

Uma das propriedades acima mencionadas foi a *Sincronia Virtual*. Esta propriedade é muito importante em sistemas distribuídos, e por isso será melhor detalhada nesta seção.

O sistema Isis [27, 26] foi primeiro a identificar a importância de se sincronizar a entrega de mensagens com as alterações das visões em um modelo assíncrono.

[10]Apesar de não discutido, este componente é necessário para certas propriedades.

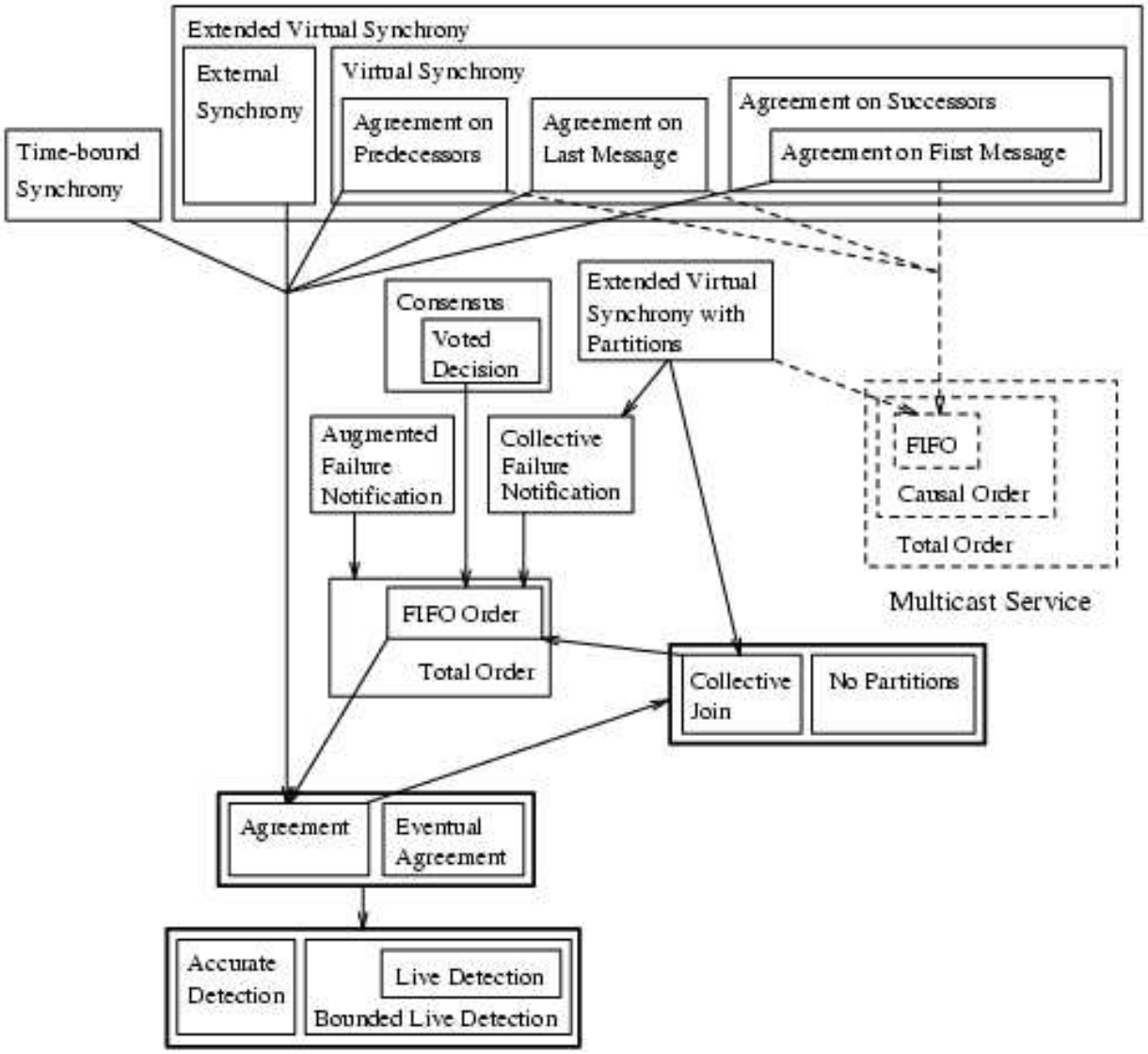

Figura 4.17: Dependência entre as Propriedades do Serviço de Pertinência

O modelo de execução de sincronia virtual não é baseado no modelo transacional, mas introduz uma abordagem similar para se programar com processos. Neste modelo, existe a abstração de um conjunto de processos, onde *todos observam os mesmos eventos na mesma ordem* [27]. Os eventos são as chegadas de mensagens ao grupo, e alterações na formação deste grupo. Uma observação interessante desta propriedade é que como todos os membros do grupo recebem os mesmos eventos, eles podem executar o mesmo algoritmo, e, desta maneira, permanecer em estados consistentes. Isto pode ser verificado na figura 4.18, onde é ilustrado um grupo de processos que recebe mensagens de vários processos não membros. Em seguida, ocorre a união do grupo com um novo membro, e o estado do grupo é então transferido para o novo membro. Em seguida, um dos membros antigos sofre uma falha.

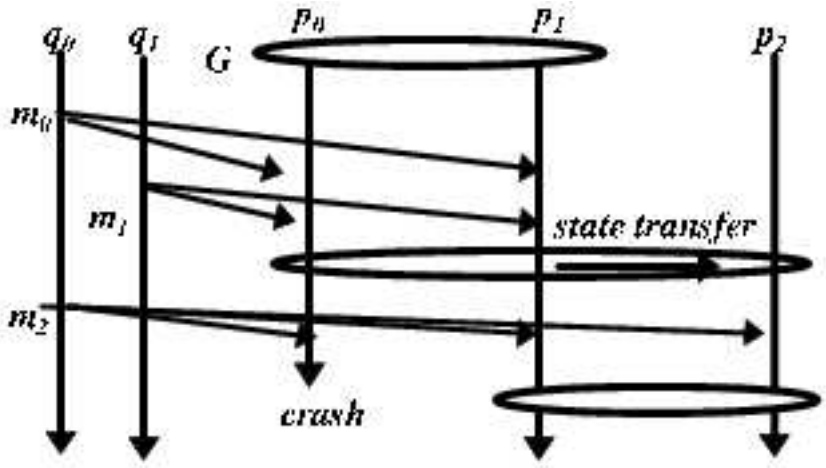

Figura 4.18: Execução com Sincronia Virtual

Este modelo se torna muito útil para a realização de replicações distribuídas, pois ele nos permite implementar *commit*s distribuídos. Desta maneira, o modelo de execução de sincronia virtual torna-se extremamente poderoso, e mais básico do que o modelo transacional, fornecendo uma visão mais ampla do problema de se construir aplicações tolerantes a falhas [87].

4.6 Protocolos de Pertinência em Grupos

Esta seção visa reunir informações sobre protocolos de pertinência em grupos. Para tanto, reunimos informações sobre especificações, algoritmos propostos para resolver o problema, além de uma discussão sobre a impossibilidade de se atingir este consenso em alguns sistemas distribuídos.

4.6.1 Abordagens na Construção de Protocolos

Assim como qualquer serviço distribuído, podemos projetar um protocolo usando-se alguma abordagem comum em algoritmos distribuídos. As abordagens mais comuns encontradas em algoritmos distribuídos são[11]:

Uso intenso de broadcasts

Uma estratégia muito comum em algoritmos distribuídos é o uso intenso de broadcasts. Esta abordagem indica que, quando necessário difundir uma informação pelo sistema, um broadcast é feito a todos os nós. Para implementar um serviço de pertinência, podemos imaginar grosseiramente que todos os nós poderiam ficar fazendo broadcast de sua visão do grupo até que um consenso fosse atingido. Se estamos trabalhando com n nós no grupo, e cada nó faz um broadcast, acabamos congestionando a rede com $\Theta(n^2)$ mensagens. Esta é a desvantagem desta abordagem: a rede pode ficar congestionada somente com informações de controle do cluster.

Uso de um "líder"

Para evitar que todos tenham um papel igual nos protocolos, e assim diminuir a quantidade de informações trafegadas pela rede, freqüentemente utilizamos a idéia de um "líder" dentro do grupo para coordenar as ações. A idéia é simples: escolhemos dinamicamente um nó para ser o líder, e a execução do protocolo será coordenada por ele. Assim, os outros nós precisariam somente trocar informações com este líder, pois ele posteriormente as transmitiria a todo o grupo. O problema desta abordagem é a existência de um ponto único de falha no processamento: se o líder falhar durante a execução do protocolo, uma rotina de recuperação deve ser executada para possibilitar uma nova eleição de líder. Desta maneira, uma complexidade maior é necessária para a monitoração constante deste líder, e de uma possível recuperação do estado do sistema após sua falha.

Uso de Anéis Lógicos

Anéis Lógicos são usados para se tentar evitar os problemas gerados pelas abordagens acima. Basicamente, o grupo de computadores é ordenado, e a partir desta ordenação é formado um anel lógico[12] de computadores. Por este anel, um token é circulado, e nele são difundidas as

[11] Esta seção não visa ser uma revisão de todas as técnicas utilizadas. Desejamos somente fornecer as estratégias mais recorrentes e comuns em sistemas distribuídos.

[12] Dizemos que o anel é lógico pois não há nenhuma exigência de que sua topologia siga esta ordenação. Somente a maneira de organizar logicamente o grupo é afetada.

informações pelo grupo. Ainda, este token é utilizado para detectar falhas e alterações no grupo. Uma desvantagem desta abordagem é o uso do token, que necessita de verificações constantes. Por exemplo, um nó pode falhar junto com o token, e por isso é necessário um mecanismo de detecção de perda e de geração de tokens.

4.6.2 Classificação dos Algoritmos

Os algoritmos que resolvem o problema da pertinência em grupos podem ser classificados de acordo com o modelo do sistema distribuído sobre o qual executam:

Algoritmos Síncronos

A propriedade característica de uma rede síncrona é que particionamentos que podem impedir que nós corretos realizem uma difusão de mensagens aos outros dentro de N unidades de tempo não acontecem. A única razão para dois nós não conseguirem se comunicar através do serviço de difusão da rede é a falha de pelo menos um deles [37].

Com um serviço de pertinência síncrono é possível atingir um consenso sobre os membros ativos de um grupo em um intervalo de tempo definido [39]. Para atingir este objetivo, um serviço de pertinência organiza os membros ativos do time em uma seqüência de grupos dinâmicos que existem no tempo, tal que as seguintes propriedades são válidas:

- estabilidade: novos grupos são criados somente em resposta a falhas ou recuperações de nós.
- concordância: todos os membros de um grupo concordam na lista de membros deste grupo.
- ordenação: depois de se unir a um grupo comum, dois membros ativos a e b se unirão à mesma seqüência de grupo, enquanto eles estiverem ativos. Desta maneira, eles observarão os eventos de falha e recuperação na mesma ordem.
- atrasos delimitados na detecção de falhas: qualquer falha de um membro ativo f do grupo A é detectada dentro de D unidades de tempo. Isto é, leva-se no máximo D unidades de tempo para a formação de um novo grupo que inclui todos os membros ativos do grupo A e exclui f.
- atrasos delimitados em uniões: qualquer recuperação de um membro j leva no máximo J unidades de tempo para a criação de um novo grupo que, além dos antigos membros, inclui também j.

Cristian descreve em [38] três protocolos para se implementar um serviço de pertinência síncrono. Estes protocolos dependem da existência de broadcast atômico síncrono e de sincronização de relógios entre os nós do sistema.

Algoritmos Assíncronos

Fornecer um serviço de pertinência para sistemas distribuídos assíncronos não é uma tarefa trivial. Para tanto, existem três modelos de sistemas distribuídos assíncronos que são usados:

1. **Livre de Tempo (*Time-Free*)**

   Segundo Cristian e Fetzer [35], o modelo assíncrono livre de tempo (*time-free*) não descreve precisamente sistemas distribuídos assíncronos construídos a partir de servidores e redes de comunicação locais. Uma razão para esta imprecisão é que este modelo tenta abstrair a noção de *tempo*. Segundo os autores, tempo é uma importância fundamental em sistemas distribuídos, pois a partir dele podemos resolver problemas que não são solúveis nos modelos time-free. Ainda, conforme veremos na seção 4.6.3, este modelo torna-se inviável para a implementação de serviços de pertinência, pois não podemos distinguir entre falhas de colapso e de performance.

2. **Temporizado (*Time-Constrained* ou *Timed*)**

   A comunicação baseada em "tempo" permite resolvermos problemas no modelo assíncrono temporizado (*timed asynchronous model*) que não são solúveis no modelo livre de tempo [36]. Como vimos na seção 3.4.1, neste modelo os nós possuem acesso a relógios locais, de maneira a determinar os tempos máximos que devem aguardar para receber as respostas.

3. **Detectores de Falhas**

   Uma possível variação para um sistema assíncrono é acrescentar um *detector de falhas* ao modelo. Um detector de falha é um mecanismo introduzido para fornecer aos processos informações sobre falhas de outros processos ou canais de comunicações [51]. Ele é um serviço distribuído implementado por um conjunto de *módulos de detecção de falhas* locais. Este modelo não define nenhum tipo de timeout, como para atrasos de mensagens, ou execução de processos. Ocorrem somente falhas de colapso nos processos e canais de comunicação.

Ao demonstrar que o modelo temporizado não é mais forte que um modelo time-free com detectores de falhas, Fetzer mostra em [51] que o modelo temporizado (*timed*) não é um modelo síncrono. Podemos observar o modelo temporizado como uma instância do modelo time-free, com um detector de falhas fixo (os relógios locais), e com propriedades de comunicação enfraquecidas.

Desta maneira, é possível introduzir *tempo* em sistemas time-free com o uso de detectores de falha.

4.6.3 Impossibilidade de Resultados

Apesar de ser mais real e fácil de ser implementado, o modelo assíncrono impõe sérias restrições ao serviço de pertinência de grupos. Basicamente, o modelo assíncrono time-free impede que o problema seja resolvido. Além disso, o estudo do problema de pertinência para grupos em sistemas assíncronos está longe de ser concluído [34]: não há consenso na definição do problema, e muitas pesquisas na área produziram resultados insatisfatórios.

Anceaume et al. descrevem em [18] que os artigos mais citados que tentam dar especificações formais para o serviço de pertinência assíncronas de partição primária e particionável ([78, 46]), bem como suas versões atualizadas ([80, 79, 48]) contêm falhas na especificação: [78, 80, 79] possuem falhas em seu formalismo lógico e permitem execuções indesejáveis, enquanto que as especificações contidas em [46, 48] podem ser satisfeitas por protocolos triviais e inúteis. A existência de tais falhas em artigos de pesquisa largamente citados deve ser observada como uma indicação da dificuldade de se especificar e implementar protocolos de pertinência assíncronos. Como apontado em [34], nenhum serviço de pertinência útil pode ser implementado no modelo assíncrono time-free, pois um sub-problema comum a tais serviços não pode ter sua resolução provada neste modelo de sistema[13]. Deve estar claro que a impossibilidade de resultados não se aplica ao modelo assíncrono temporizado, devido a duas razões:

1. Os relógios locais de sistemas temporizados permitem resolver problemas que não são solúveis no modelo time-free.

2. O problema de pertinência é especificado para sistemas temporizados de maneira diferente de sua especificação para sistemas time-free.

A concordância entre os membros do grupo requerida pelo serviço de pertinência deve ser cuidadosamente especificada, para que dois objetivos potencialmente opostos possam ser atingidos [34]:

1. Ela deve ser fraca o suficiente para ser possível resolvê-la.

2. Ela deve ser forte o suficiente para simplificar o projeto de aplicações distribuídas tolerantes a falhas. Contudo, ela não pode ser satisfeita por protocolos inúteis ou triviais.

[13] O sub-problema comum é o problema de *Consenso*. Fischer et al. mostram em [54] que este problema não pode ser resolvido em ambientes assíncronos, mesmo com somente um processo defeituoso.

Para serviços de pertinência de partição primária, estes objetivos são incompatíveis com sistemas assíncronos sujeitos a falhas, mesmo se a comunicação for confiável e os processos possuírem somente semântica de falhas de colapso. É importante ressaltar que que esta impossibilidade de resultados também se aplica a serviços de pertinência que podem remover ou matar um número arbitrário de processos não defeituosos que não desejavam serem removidos. Assim, ao contrário da visão geral [78, 16, 46, 49], permitir a retirada de processos suspeitos de terem travado *não* é suficiente para tornar o problema de pertinência em grupos de partição primária solúvel [34].

4.7 Trabalhos Relacionados

Conforme apontam [18, 34], o problema da pertinência em grupos foi definido para sistemas síncronos pela primeira vez por Cristian em [37]. Desde então, o problema tem sido alvo de intenso estudo em sistemas assíncronos (como [61, 71, 78]). Em particular, dois tipos de serviços surgiram: o serviço de pertinência de *partição primária* ([78, 61, 71, 72, 57]) e o serviço de pertinência *particionável* ([15, 16, 58, 76, 23, 49]). Alguns destes sistemas assumem um modelo de sistema time-free, como por exemplo [78, 61, 71, 57, 23], enquanto que outros não deixam claro se o modelo de sistema é time-free ou temporizado (*timed*).

Alguns autores propuseram especificações formais de serviços de pertinência, como os encontrados em [58] e em [52]. Neste último, Cristian e Fetzer derivam a especificação de um serviço de pertinência de nós particionável, que é então especializada para modelos síncronos e assíncronos.

Fetzer e Cristian propõem em [53] outra abordagem para o serviço de pertinência. Esta abordagem inclui a noção de *percepção a falhas* (*fail-awareness*). *Percepção a Falhas* é um conceito que permite o 'enfraquecimento" de uma especificação S de um sistema síncrono, criando uma nova especificação FS que seja implementável em sistemas assíncronos [53].[14]

[14] De modo geral, um servidor *sabe* quando ele não pode fornecer sua semântica padrão, e então sinaliza estes eventos aos clientes afetados através da indicação de uma exceção. Desta maneira, os clientes podem tomar ações de recuperação apropriadas em seu nível de abstração, de maneira similar à técnica de mascaramento hierárquico, discutido na seção 3.6.4.

Open Clustering Framework - OCF

bandwagon, n.:
An activity, group, movement, etc. that has become successful or fashionable and so attracts many new people.

a bandwagon effect:
The success of the product led many firms to try to jump/climb/get on the bandwagon (= copy it in order to have the same success).
– Cambridge International Dictionary of English.

5.1 Introdução

Após termos estudado todos os detalhes relacionados a clusters de alta disponibilidade e ao Serviço de Pertinência em grupos, apresentamos neste capítulo uma proposta de padronização arquitetural para clusters.

Devido ao crescente uso do Linux, vários grupos desenvolveram independentemente clusters de alta disponibilidade para este sistema operacional. Cada um destes sistemas teve uma abordagem diferente para o problema, mas todos certamente enfrentaram problemas semelhantes. Se houvesse alguma padronização nesta área, esforços poderiam ter sido reaproveitados, e não duplicados.

Neste contexto, o *Open Clustering Framework* [85] (OCF) é um projeto open-source que possui esforços dedicados a uma padronização de clusters para Linux[1]. Veremos que uma padronização é fundamental para a consolidação desta área, e que o tema já foi abordado por outros autores.

Este capítulo está bastante baseado em [69] e [85], além das informações discutidas em [7].

[1] Como veremos a seguir, este é somente o objetivo inicial; idealmente o padrão definido poderia ser implementado em qualquer plataforma.

5.2 Motivação

Apesar de poder parecer evidente a motivação para um projeto de padronização de clusters, é importante descrevermos o porquê da existência do OCF.

5.2.1 Panorama Atual de Clusters em Linux

Nos últimos anos, muitos projetos de clusters surgiram em Linux, tanto open-source quanto comerciais. Atualmente, existem pelo menos 10 soluções de clusters open-source, e mais de 25 comerciais [69]. No *Linux Cluster Information Center* [56], Greenseid reuniu muitas informações interessantes sobre clusters em Linux, desde projetos existentes até relações de documentos e livros da área.

5.2.2 O Problema

Por oferecer tantas opções, o cenário atual parece muito bom, mas esconde um problema real: estas soluções não se encaixam. Elas não compartilham uma API comum, nem conceitos comuns da área.

As razões para estes sistemas não serem padronizados são basicamente históricas, como projetos independentes, razões comerciais, projetos que visavam resolver somente partes do problema, etc.

Nas plataformas proprietárias existe o *gorila de 900 libras* (*900 hundred pound gorilla*) chamado "O Fornecedor". Ele dita as normas e rege os padrões. No Linux, não existe tal conceito, o que é comumente considerado uma boa coisa. Entretanto, sem um padrão único para ser adotado, todos saem perdendo [69]. Alguns exemplos dos que perdem com a falta de padronização:

- O usuário final. Sem padrões e interoperabilidade, o usuário final precisa adquirir muitos sistemas diferentes, que independentemente resolvem partes do problema.
- Fornecedores de software independentes. Eles querem vender seu software para um público grande, preferencialmente todos os usuários. Entretanto, sem uma padronização na área, eles são obrigados a adaptar seus sistemas para rodar nas diversas soluções, ou simplesmente abandonar parte do mercado.
- Projetos Open-Source. Tudo o que foi dito anteriormente também se aplica neste caso, com um agravante: os projetos open-source possuem poucos recursos. Para desenvolver sistemas para clusters é necessário mais recursos do que um programador regular possui em casa, como várias máquinas para montar o ambiente.

- Integradores de Sistemas. Para ser coerente, uma solução de integração deve incluir todos (ou grande parte) dos sistemas existentes. Sem padronizações, esta tarefa torna-se muito difícil.

Portanto, uma solução para este problema seria interessante para todos aqueles relacionados de alguma forma com grupos de computadores.

5.3 A Solução

Birman aponta em [27] que em 1995 já havia interesse considerável em se formar uma organização para desenvolver e prototipar potenciais padrões para uma API de clusters. Tal API forneceria ao desenvolvedor um conjunto de primitivas conhecidas, disponíveis em clusters de diversos fornecedores. Esta padronização possibilitaria a construção de aplicações para clusters altamente portáveis, libertando os desenvolvedores de plataformas e de soluções proprietárias. Para tanto, esta API deveria contemplar os seguintes itens:

1. Ela deveria possibilitar o software acessar a configuração do sistema e as interfaces de gerenciamento. Alguns aspectos destes requisitos poderiam ser atingidos por especificar o uso de SNMP[2] ou CMIP[3].

2. Ela deveria fornecer uma arquitetura simples para explorar as capacidades de hardware do cluster.

3. Ela deveria fornecer meios uniformes para se acessar as aplicações. Por exemplo, o cluster como um todo poderia ser acessado através de único IP, mas seus componentes teriam um IP secundário que possibilitaria um acesso direto a eles.

4. Ela deveria fornecer primitivas no nível do sistema operacional que, entre outras coisas, possibilitariam executar uma aplicação em um determinado nó, monitorar uma aplicação enquanto ela executa, migrar tarefas entre nós e acessar discos remotamente.

5. Ela deveria lidar facilmente com aplicações fora do clusters, e não deveria ser restrita somente a questões internas ao cluster.

Pfister também comenta sobre o problema de não se possuir uma padronização de clusters em [75].

[2]SNMP = Simple Network Management Protocol
[3]CMIP = Common Management Information Protocol

Uma possível solução para o problema seria a adoção de um padrão de fato do mercado em alguma outra plataforma. Entretanto, a busca por tal solução genérica para clusters não teve sucesso. Apesar de existirem padrões aceitos para partes do problema de clusters (como MPI), não existe uma arquitetura ou solução genérica que atacasse o problema como um todo, e que fosse reconhecida como um padrão.

5.3.1 Um Framework Padrão

Neste contexto é que se encaixa o *Open Clustering Framework*. Depois dos caminhos mais fáceis terem sido analisados e considerados incapazes de resolver o problema, foi verificado que quase todas as soluções existentes são conceitualmente similares. Assim, um modelo comum de APIs associadas poderia ser definido de maneira genérica a cobrir as funcionalidades necessárias.

Portanto, a solução para o problema seria a criação de um framework que padronizasse a comunicação entre os componentes de um cluster, e que contemplasse a resolução dos problemas mais comuns relacionados a clusters.

Desta maneira, o objetivo do projeto OCF é [85]:

- Definir um modelo comum para clusters em Linux.
- Definir e implementar um conjunto padrão de APIs para este modelo na plataforma Linux.
- Criar e dar suporte a projetos de desenvolvimento open-source que atuem como implementações de referência para estas APIs.

5.3.2 Escopo

Apesar do foco inicial ser no Linux, não há restrição de os padrões serem exclusivos ou dependentes do Linux. Como um framework padrão também não foi encontrado em outras plataformas, as definições realizadas pelo OCF podem ser úteis e interessantes para outros sistemas operacionais.

Apesar da inclinação ser mais forte em clusters de alta disponibilidade, esta padronização visa também atingir clusters de alta performance. Felizmente, a intersecção entre as duas áreas é grande o suficiente para acomodar os dois lados na padronização.

5.4 Requisitos

Vários requisitos foram definidos para o OCF [69]. Apresentaremos nesta seção os requisitos agrupados por categoria.

5.4.1 Projeto como um todo

O projeto como um todo possui os seguintes requisitos:

- *Bandwagon Effect*[4]: capturar atenção suficiente para que as definições sejam adotadas por uma parte substancial do mercado. Se não ocorrer desta maneira, o trabalho do OCF não terá sentido.
- O trabalho produzido não pode possuir restrições de uso. O OCF deseja tanto comportar implementações proprietárias como open-source. É aceitável que implementações individuais sejam patenteadas, mas as padronizações e APIs devem poder ser implementadas sem o custo de direitos autorais.
- Resultados Imediatos: foi observado que a rápida entrega de algo utilizável, mas com melhorias contínuas e iterativas, é importante para preservar o interesse pelo projeto.

5.4.2 APIs

Para que as APIs sejam amplamente aceitas, elas devem contemplar os seguintes requisitos:

- Sempre que possível, as APIs devem adotar as melhores práticas das implementações atuais, ou serem abstratas o suficiente para atender o maior número possível de implementações existentes.
- Apesar de focadas primeiramente no Linux, as APIs não devem ser aplicáveis somente ao Linux:
  - A implementação no Linux é o objetivo inicial, e a busca por compatibilidade com outros sistemas operacionais não deve afetar negativamente o desenvolvimento.
  - No entanto, o grupo deseja criar APIs que sejam facilmente implementadas em outros sistemas operacionais.
- As especificações das APIs devem objetivar uma conformidade com a POSIX. Elas deveriam estender especificações existentes, adicionando-se as funcionalidades de clusters.
- APIs não devem direcionar suas possíveis implementações. Por exemplo, elas não devem ditar se suas implementações executarão no espaço de usuário ou no espaço do sistema.
- Enquanto que todo projeto de clusters é livre para implementar estas APIs diretamente, também deve ser possível construir-se uma camada de compatibilidade entre o código legado e esta nova API.

[4] Veja a citação no começo do capítulo.

5.4.3 Implementação de Referência

Os seguintes requisitos para as implementações de referência foram colocados:

- Deve ser possível compilar e gerar a implementação em várias plataformas. Isto implica o uso de ferramentas como `autoconf`, que geram automaticamente as regras de compilação em um `Makefile`.
- Sempre que possível, os componentes devem ser portáveis.
- As interfaces dos componentes devem ser indiferentes ao sistema operacional, e ao fato da implementações estar dentro/fora do kernel.

5.5 Histórico

O projeto começou a surgir durante o *Ottawa Linux Symposium 2001*, quando várias pessoas se reuniram em uma reunião introdutória para discutir o problema. Na seqüência, ocorreu um workshop sobre clusters em Enschede, Holanda, durante o 8^{th} *International Linux Kongress* [93], de 26 a 28 de Novembro de 2001. Nesta ocasião, as linhas gerais do projeto foram definidas, e a primeira visão do modelo de componentes do projeto foi exposta [85]. Depois desta ocasião, o grupo se reencontrou nos dias 20 e 21 de Janeiro de 2003 em Nova Yorque. A proposta desta reunião seria discutir o andamento do projeto, e discutir como o trabalho poderia se aproveitar de outros projetos que possuem objetivos similares. Na seção 5.8 detalharemos quais são as afiliações com outros grupos que o OCF está realizando.

O grupo de trabalho consiste em pessoas que têm interesse em clusters, tanto desenvolvedores como consumidores da tecnologia. A discussão se dá através de uma lista de discussão, onde as idéias são expostas e discutidas [7].

5.6 O Modelo de Componentes Atual

A proposta atual dos componentes funcionais do framework OCF é exibida na figura 5.1 [69]. Eles representam uma estruturação comum dos componentes necessários em um cluster. Esta é a visão atual do grupo de trabalho, mas ainda está em discussão e sujeita a modificações.

Os componentes estão divididos segundo os *objetos* com os quais trabalham, como nós, grupos de processos, locks, etc. A API que está sendo preparada irá mapear as interfaces externas destes componentes.

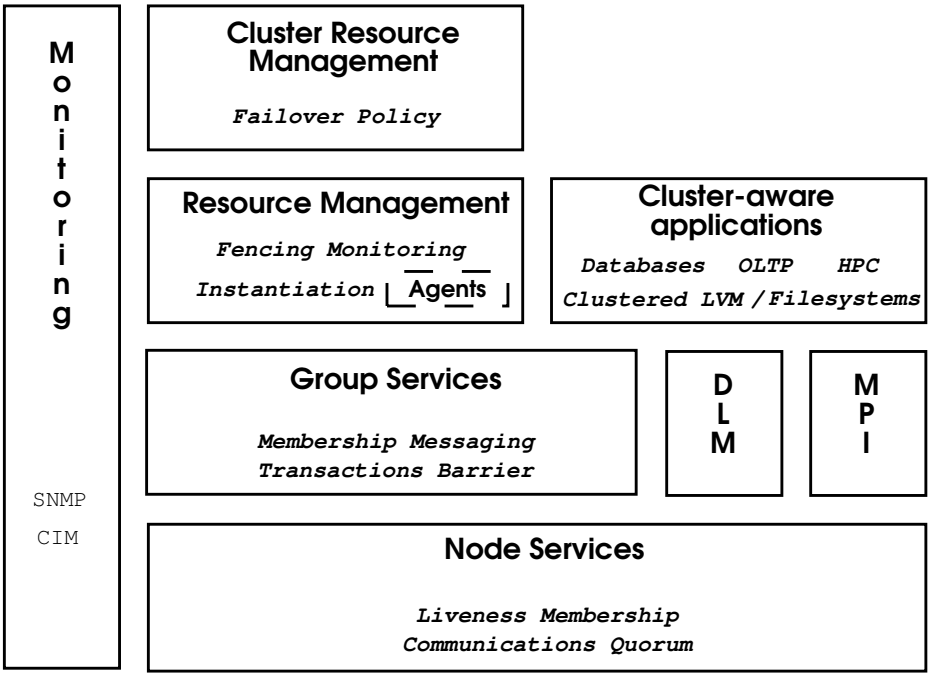

Figura 5.1: Modelo de Componentes do Framework OCF

Esta componentização do framework permite que arquitetos de sistemas possam construir clusters a partir das implementações dos componentes que mais lhes agradem. Esta capacidade é muito interessante, pois permite a fácil construção de soluções específicas para cada cenário. Por exemplo, não seria mais necessário construirmos uma solução que funcionasse para clusters de 2 a 100 nós; para cada necessidade, um sistema de clusters seria montado a partir de implementações existentes dos componentes.

5.6.1 Serviços de Nós (*Node Services*)

Serviços de Nós abrangem todos os serviços relacionados a nós. Uma das questões em aberto é como identificar um nó específico no cluster. Algumas possíveis abordagens incluem o uso do `hostname`, de um número seqüencial inteiro, ou mesmo do endereço `IP` primário do nó. Cada uma destas possibilidades possui vantagens e desvantagens em seu uso. Ainda, as camadas superiores de software ou aplicações poderiam ser otimizadas se conhecessem em detalhes como um nó é identificado.

Isto demonstra um dos pontos que o OCF está lidando: balancear o nível de abstração entre a API e o modelo, de maneira a possibilitar o melhor desenvolvimento dos componentes, mesmo que estes estejam definidos de uma maneira genérica e abstrata.

Saúde do Nó (*Node Liveness*)

Este é o componente responsável por monitorar a saúde do nó. Geralmente implementado como uma propriedade binária significando `vivo` ou `morto`, ele poderia também conter informações mais detalhadas sobre o estado dos nós.

Serviços de Comunicação para Nós (*Node Communication Services*)

Este seria o nível mais baixo de comunicação entre os nós. Ele forneceria a capacidade de comunicação entre os nós. Potencialmente, este serviço é usado não somente pelo Serviço de Pertinência, como também pelos Serviços de Comunicação de Grupo de mais alto nível.

O desafio será definir um componente que seja um "denominador comum", abstrato o suficiente para ser utilizado em diferentes modelos de autenticação, criptografia, mensagens e semânticas de endereçamento.

Serviços de Pertinência de Nós (*Node Membership Services*)

A lista de membros de um cluster - lista de nós ativos ou elegíveis do cluster - está em constante alteração. Entretanto, é vital que seus membros possuam uma visão coerente do cluster. Este componente é responsável por manter esta coerência entre as visões do cluster entre os nós, e utiliza o componente de *Saúde do Nó* para atingir um consenso na lista de membros.

Serviços de Quórum para Nós (*Node Quorum*)

Em um cluster de alta disponibilidade, uma das questões mais fundamentais é determinar que nós possuem autorização para modificar recursos e informações compartilhados. No caso de um particionamento no grupo, onde parte do cluster perde contato com o resto, o cluster deve garantir que no máximo uma partição continue a operar sobre a informação.

Um quórum é *o número de participantes necessário para que uma assembléia possa deliberar.* Em clusters, freqüentemente quórum é tratado da mesma maneira, indicando o número mínimo de membros de um grupo para que ele possa prosseguir com sua execução. Outras abordagens incluem o uso de algoritmos de decisão, que podem determinar que o acesso a um determinado dispositivo também é necessário para se prosseguir com a execução.

A maioria das implementações de clusters possui a noção de que somente uma partição deve possuir quórum. Entretanto, em grandes clusters ou clusters hierárquicos isto não vale, pois múltiplos níveis de quórum existem.

5.6.2 Serviços de Grupos (*Group Services*)

Uma aplicação desenvolvida para um cluster sabe que ela está sendo executada em um ambiente distribuído, possivelmente se estendendo por múltiplos nós. Um modelo comum para esta situação é a formação de um *grupo de processos* pelos nós do cluster. Este componente é responsável por lidar com as questões relacionadas a este tipo de formação.

Serviços de Pertinência de Grupos (*Group Membership Services*)

Este serviço de pertinência é muito similar ao Serviço de Pertinência de Nós. A diferença é que este serviço encontra-se em uma camada superior, e é responsável por gerenciar a lista de processos membros de um determinado grupo. Trata-se do serviço de pertinência de processos, como definido na seção 4.2.2.

Serviços de Quórum para Grupos (*Group Quorum Services*)

Há discussões no OCF para determinar se quórum é somente uma propriedade da camada de nós, ou se grupos de processos diferentes podem ter noções distintas do que quórum significa. Geralmente, os clusters atuais só fornecem quórum para nós. O framework OCF deve ser flexível o suficiente para ser estendido e permitir quórum para grupos de processos.

Serviços de Mensagens para Grupos (*Group Messaging Services*)

Este componente provê serviços de comunicação para grupos de processos. Deparando-se com os mesmos desafios encontrados em comunicação entre nós, o Serviço de Mensagens ainda deve lidar com outros problemas: mensagens para grupos geralmente possuem garantias adicionais a respeito da ordem das mensagens, como *sincronia virtual*[5].

Serviços de Votação para Grupos (*Group Voting Services*)

Freqüentemente, um grupo precisa votar para tomar uma decisão. Por exemplo, para escolher um coordenador do grupo para executar determinada tarefa, geralmente os clusters tomam esta decisão através do voto[6]. Este componente é responsável por gerenciar todo este mecanismo.

[5] Nem todos os serviços de mensagens implementam esta propriedade.

[6] Geralmente, o mecanismo de escolha é simplesmente conceder capacidades de líder ao nó que possuir o maior `id` no grupo.

Serviços de Barreiras para Grupos (*Group Barrier Services*)

Um grupo de processos distribuídos também precisa garantir que certas operações são iniciadas somente após todos os membros do grupo atingirem um determinado estado. Este estado é chamado de *barreira*. O Serviço de Barreiras fornece uma maneira fácil de se realizar esta sincronização.

Serviços de Transação em Grupos (*Group Transaction Services*)

Transações são uma abordagem comum para se conter falhas: ou uma transação é executada como uma transformação atômica e correta do estado do sistema, ou não é executada, e o sistema retorna ao seu estado anterior. A grande vantagem de transações é que o próximo estado após uma transação sempre será um estado válido. Em clusters, transações distribuídas com *two-phase commit* são especialmente úteis.

A vantagem de se construir aplicações sobre um framework transacional é que elas herdam as propriedades transacionais. Tarefas como gravação de logs, *journaling* e tarefas de recuperação são bastante simplificadas ao utilizar-se um framework transacional.

Este componente é responsável por fornecer semânticas transacionais ao framework. Isto implica em lidar com uma variedade de modos de falha, assim como garantir as propriedades ACID[7] para todos eles.

5.6.3 Gerenciamento de Recursos (*Resource Management*)

Clusters de alta disponibilidade, especialmente os chamados sistemas de failover e switch-over, geralmente são focados no gerenciamento de grupo de recursos. Um recurso é uma entidade única, física ou virtual, que provê um serviço a clientes ou a outros recursos. Um recurso está geralmente disponível para uso em dois ou mais nós do cluster, de maneira a estar sempre disponível, mesmo em caso de falha do nó que o hospeda.

Estes recursos básicos são combinados em grupos de recursos, os quais são tratados como unidades atômicas pelo sistema de failover. A tarefa do componente de Gerenciamento de Recursos é controlar estes grupos de recursos, assim como ser capaz de inicializar ou parar a execução de qualquer recurso isoladamente.

[7] Atomicidade, Consistência, Isolamento e Durabilidade.

Capacidade de Instanciação de Recursos (*Resource Instantiation Facility*)

Se o cluster decide inicializar ou parar certo recurso em um nó específico, um mecanismo genérico para executar esta tarefa deve existir. Portanto, o framework deve contemplar tal sistema genérico de inicialização de serviços.

Monitoração de Recursos (*Resource Monitoring*)

Para possuírem alta disponibilidade, os recursos também devem ter sua *saúde* monitorada. Soluções de failover que monitoram somente a saúde do nó em que certo recurso está hospedado não são muito úteis, visto que aproximadamente 80% das falhas são devidas a problemas de software [69].

Este componente deve fornecer uma interface genérica para se verificar a saúde de um recurso, assim como para notificar o cluster das alterações no estado do recurso. Desta maneira, os recursos podem ser tolerantes a falhas, sendo restabelecidos em outros nós em caso de falhas.

Serviços de Isolamento a Recursos (*Resource Fencing Services*)

A maioria dos recursos que não são desenvolvidos para um ambiente distribuído podem estar ativos somente em um nó em um dado momento. De outra maneira, há grandes chances de acontecerem estragos no cluster, como corrupção de informações compartilhadas.

O componente de isolamento (*fencing*) garante que existirá no cluster somente uma instância de cada recurso. Isto é realizado possibilitando ao cluster aplicar políticas de acesso a recursos compartilhados. Vários níveis de granularidade podem existir: alguns clusters podem isolar (*fence*) um nó defeituoso/particionado de todos seus recursos compartilhados desligando sua força[8]; outros podem controlar mais detalhadamente o recurso compartilhado, a fim de não permitirem mais acessos originados do nó indesejado.

Este componente também está relacionado ao quórum de nós ou grupos: alguns sistemas tratam quórum simplesmente como outro recurso, o qual pode estar presente ou não. Desta maneira, a ausência do recurso `quórum` implica em parar todo o processamento do cluster, e desalocar todos os recursos que estavam sendo utilizados.

Agentes de Recursos (*Resource Agents*)

Esta é a camada que interliga o software de switch-over e os próprios recursos sendo gerenciados. O objetivo de agentes de recursos é integrar o recurso com o software de switch-over,

[8]Esta é a idéia por trás de STONITH: *Shoot The Other Node In The Head.*

encapsulando-o cuidadosamente e assim tornando-o portável entre nós do cluster.

Todos os recursos possuem um conjunto comum de métodos que eles devem expor ao software de failover. Geralmente, estes métodos contêm as funções de iniciar, parar ou verificar o estado do recurso. Este requisito é diretamente identificado nos scripts de inicialização do *Linux Standard Base* [4] (LSB). A única exceção é que múltiplas instâncias do mesmo tipo podem estar ativas em um dado nó simultaneamente.

O conceito de *agentes de recursos* é comum entre soluções de switch-over. Entretanto, as interfaces variam nos detalhes, o que gera todos os problemas descritos na seção 5.2.2. Como o LSB já possui uma padronização no controle dos serviços oferecidos por uma máquina, esta padronização poderia ser estendida para contemplar os serviços oferecidos por um cluster.

Gerenciador de Recursos do Cluster (*Cluster Resource Manager*)

Se a lista de membros do cluster é alterada pela exclusão de certo nó, possivelmente algum grupo de recursos deixou de estar presente na nova configuração. Esta alteração deve ser percebida pelo *Gerenciador de Recursos do Cluster*, a fim de que ele coordene a recuperação do grupo de recursos. Ele também é responsável por garantir a exclusividade de acesso a recursos compartilhados, como descrito anteriormente.

5.6.4 Gerenciador de Lock Distribuído (*Distributed Lock Manager*)

Muitas aplicações desenvolvidas para clusters coordenam o acesso a recursos compartilhados travando o objeto em questão. Esta é uma abordagem comum para garantir coerência e sincronização. No caso de falhas, a recuperação destes *locks* é uma tarefa muito complexa.

Devido a fatores históricos, quase todo Gerenciador de Lock fornece uma interface muito similar ao do gerenciador de lock de clusters VAX, tanto sintática quanto semanticamente. Desta maneira, já existe uma API muito utilizada e "padronizada". Ela necessitará somente de pequenos ajustes para garantir a coerência da API.

5.6.5 Monitoração do Cluster (*Cluster Monitoring*)

Após determinarmos uma API coerente para um componente, também será possível determinarmos uma interface comum com *sistemas de gerenciamento de redes* para todos os clusters em Linux. Um caso seria a definição de uma MIB[9] de SNMP para monitorar o cluster.

[9]MIB = Management Information Base.

5.6.6 Configuração do Cluster (*Cluster Configuration*)

A configuração de um cluster é outro ponto complexo. A administração de um cluster deve ser automatizada e fácil de ser realizada. De outra maneira, o cluster seria uma tecnologia muito difícil de ser mantida e gerenciada.

5.6.7 Mecanismos de Comunicação entre os Componentes

Existem algumas possibilidades para interligar e comunicar com os componentes do framework.

Mecanismo de Carregamento de Plug-Ins

Muitos sistemas Linux modernos usam intensamente o carregamento dinâmico de módulos, mais conhecidos como *plug-ins*.

Entretanto, a maioria destes sistemas implementa seu próprio mecanismo e interface de gerenciamento, de forma a satisfazer sua necessidade básica e imediata. Desta maneira, um mecanismo mais genérico é necessário para garantir seu uso em diversas situações.

Robertson descreve em [82] um mecanismo de carregamento de plug-ins genérico que está sendo desenvolvido para contemplar este requisito do OCF.

Comunicação entre espaço do usuário e espaço do sistema

As implementações dos componentes podem residir tanto dentro do kernel, quanto no espaço do usuário. De qualquer maneira, os clientes destes componentes precisam ter acesso à funcionalidade exportada.

Para uma API comum existir, o framework deve possuir uma camada genérica capaz de traduzir as chamadas entre o espaço do usuário e do sistema. Desta maneira, a comunicação entre os componentes sempre será possível e transparente.

Mecanismo Genérico de Eventos

Quase todos os componentes possuem a necessidade de entregar e reagir ao disparo de eventos. A figura 5.2 [45] ilustra como um mecanismo de eventos interagiria com os outros componentes de um cluster.

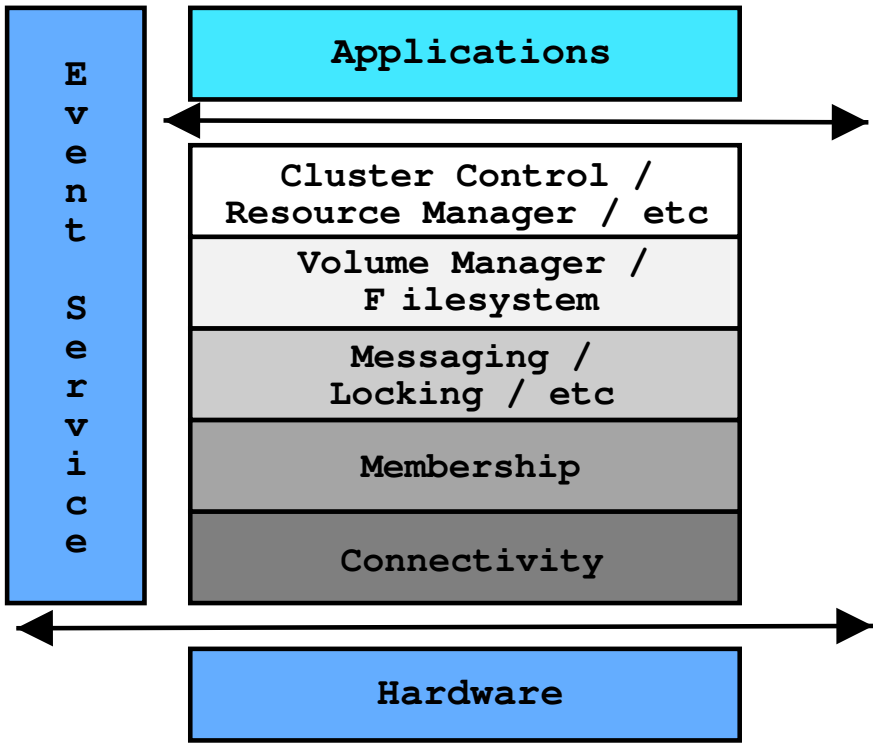

Figura 5.2: Pilha de Softwares de Clusters

5.7 Especificações

As especificações definidas pela OCF para contemplar o modelo acima apresentado estão reunidas no apêndice B. Ressaltamos que estas especificações aqui reunidas estão em suas últimas versões apresentadas, porém ainda não estão finalizadas.

5.8 Afiliações com Outros Grupos

A fim de ser mais eficaz na tarefa de padronizar a arquitetura de clusters, o OCF pretende trabalhar com outros grupos. Estas afiliações têm tanto a intenção de aproveitar as padronizações já existentes no mercado, como unir forças com grupos específicos que já possuam esforços similares.

5.8.1 Free Standards Group (FSG)

A *Free Standards Group* [2] (FSG) é uma organização independente sem fins lucrativos, dedicada a acelerar o uso e aceitação de tecnologias open-source, através do desenvolvimento,

aplicação e promoção de padrões.

O objetivo do OCF é tornar-se um grupo de trabalho da FSG. A expectativa é de se aproveitar a experiência organizacional e jurídica da FSG, sendo capaz desta maneira de se concentrar somente no trabalho técnico.

5.8.2 IEEE Task Force on Cluster Computing (TFCC)

A *IEEE Task Force on Cluster Computing* [3] (TFCC) é um fórum internacional de promoção à pesquisa e educação em clusters computacionais. Ela atua ajudando a definir e promover padrões técnicos nesta área. A TFCC está preocupada com questões relacionadas ao projeto, análise, desenvolvimento e implementação de sistemas baseados em clusters.

O foco da IEEE é basicamente dedicado a clusters de alta performance (HPC). O subgrupo trabalhando em alta disponibilidade (TFCC-HA) toma uma posição passiva e basicamente monitora o desenvolvimento. Este grupo mostrou interesse em trabalhar com a OCF.

5.8.3 MPI Forum

O *MPI Forum* [6] é um grupo aberto, com representantes de várias organizações que definem e mantêm o padrão *Message Passing Interface* (MPI). Este padrão é usado amplamente em HPC.

Por possuir objetivos diferentes, ele não é imediatamente adaptável às necessidades de alta disponibilidade. Entretanto, o OCF deve permitir uma camada MPI em seu topo, de maneira que aplicações de HA e HPC possam rodar no mesmo cluster.

5.8.4 Service Availability Forum (SAF)

O *Service Availability Forum* [8] é um grupo de empresas de comunicação e computação que trabalham para criar e promover padrões abertos de interfaces para serviços de comunicação. Em outras palavras, eles desejam fornecer basicamente os mesmos padrões do OCF, com um foco inicial na indústria de telecomunicações.

A principal diferença é que o SAF é mais fechado, com um alto custo para participar no projeto. Mesmo assim, as especificações deste grupo serão supostamente abertas e livres de direitos autorais, e devem possibilitar a existência de implementações open-source.

O futuro relacionamento entre o OCF e o SAF ainda é desconhecido. As possibilidades vão desde dois grupos totalmente disjuntos que não aproveitariam recursos, até uma participação conjunta na definição dos padrões.

5.9 Projetos de Clusters de Alta Disponibilidade

A fim de ilustrar a diversidade de projetos relacionados a clusters de alta disponibilidade, esta seção dará um breve esclarecimento sobre alguns sistemas distribuídos tolerantes a falhas existentes. Esta descrição dará uma abordagem especial aos aspectos dos Serviços de Pertinência utilizados, e será dividida entre protocolos propostos no meio acadêmico, e protocolos implementados e usados na prática por sistemas de código aberto.

5.9.1 Trabalhos Acadêmicos

Existem vários grupos de pesquisa que já trataram da questão de alta disponibilidade e de serviços de pertinência para clusters. Tal estudo foi feito geralmente em conjunto com o estudo de mecanismos de comunicação em clusters, por serem problemas bastante relacionados. Esta seção irá enumerar alguns destes grupos, mencionando suas principais características.

Isis, Horus e Ensemble

Isis [11, 26, 29] foi um sistema desenvolvido para o estudo de sistemas distribuídos tolerantes a falhas. O sistema implementa uma coleção de técnicas usadas para construir aplicações robustas para sistemas distribuídos, que toleram falhas de software ou de hardware. Sistemas distribuídos podem ser construídos a partir da interligação de sistemas convencionais, e, através das ferramentas fornecidas pelo Isis, é possível realizar o gerenciamento, sincronização, reconfiguração e replicação das informações. Isis já foi usado em várias situações diferentes, desde sistemas de negociação de ações a sistemas de chaveamento para telecomunicações.

O serviço de pertinência do Isis consiste em um componente de gerenciamento de visões de nós distribuído e uma primitiva de multicast ordenada (GBCAST), que é usada para garantir sincronia virtual [57]. Os nós trocam mensagens "hello" entre si, e o não recebimento de tal mensagem indica a falha de um nó.

Cada nó mantém uma visão do grupo, e o componente de gerenciamento garante que cada nó operacional passe pela mesma seqüência de visões. O nó mais antigo na visão é chamado de *gerente da visão*, e possui um papel de coordenador no protocolo. A mudança de visões segue um protocolo *two-phase commit*, e a primitiva GBCAST garante uma ordem total com respeito a todas as mensagens.

O projeto Horus [10, 28, 90] foi lançado inicialmente como um esforço para redesenhar o sistema de comunicação de grupo do Isis, mas evoluiu para uma arquitetura de comunicação com suporte avançado para o desenvolvimento de sistemas distribuídos robustos, em situações onde o Isis não era adequado. Ainda, o Horus é muito mais rápido e leve do que o sistema Isis.

Horus provê um framework para o desenvolvimento de aplicações distribuídas baseadas em comunicação de grupo. Para este framework, muitos sistemas e protocolos foram desenvolvidos para permitir que novos módulos pudessem ser incorporados com um custo mínimo. Além do uso prático do software, o projeto contribuiu para o estudo da teoria de *sincronia virtual*.

O Ensemble [9, 28] é a próxima geração do Horus. Ensemble é um toolkit altamente modular e reconfigurável para pesquisadores de sistemas distribuídos. Os protocolos de alto-nível providos às aplicações são construídos através de pilhas de camadas finas de protocolos. Várias características podem ser alteradas nestas camadas, a fim de experimentar novas propriedades ou mudar as configurações de performance do sistema. Tudo isso torna Ensemble uma plataforma muito flexível para realizar pesquisas.

Totem

O sistema Totem [12, 17] é um conjunto de protocolos de comunicação que facilitam a construção de aplicações distribuídas tolerantes a falhas. Suas características são:

- Multicasts ordenados e confiáveis a grupos de processos.
- Alta taxa de transmissão e baixa latência previsível.
- Código altamente portável por usar capacidades padrões de Unix.
- Rápida reconfiguração para remover processadores defeituosos, adicionar processadores novos e reparados, e reunir um sistema particionado.
- Operação continuada de todas as partes de um sistema particionado, até o ponto máximo que esteja consistente com corretude.

Para atingir todos estes objetivos, o sistema mantém um anel lógico por onde circula um token contendo informações de controle. A pertinência no Totem é baseada em reformar o grupo toda vez que a lista de membros se tornar *suspeita*, isto é, sempre que a falha de um nó, a perda de um token ou uma nova mensagem de fora do grupo for detectada [57]. O protocolo é vivo, garante ordenação FIFO das mensagens de pertinência, sincronia virtual estendida, e sincronia virtual estendida com particionamentos.

Transis

O sistema Transis [13, 47] é uma camada de comunicação multicast que facilita o desenvolvimento de aplicações distribuídas tolerantes a falhas. O sistema permite a comunicação entre

um grupo de processos, através de muitas formas de multicasts: ordenação FIFO, ordem causal, ordem total e entrega garantida.

A abordagem do Transis adquiriu grande reconhecimento da comunidade acadêmica, principalmente pelas seguintes propriedades:

- Ele usa um protocolo de multicast muito eficiente, baseado no protocolo Trans, que utiliza multicasts em hardware.
- Ele pode sustentar altas taxas de transferências de dados devido ao seu mecanismo de controle de fluxo e a seu simples modelo de grupos.
- Ele suporta operações particionadas, e provê os meios para reunir componentes em uma recuperação.

O serviço de pertinência do Transis está integrado ao seu serviço de comunicação, de maneira que as notificações de mudança do grupo são entregues no mesmo canal de mensagens normais [47]. O serviço tolera o particionamento do cluster em grupos menores. Entretanto, no caso de operações particionadas não serem desejadas, é possível adicionar uma camada fina acima do serviço de pertinência, de maneira que somente uma partição continue a operar.[10]

Quando o grupo se divide em um particionamento, cada sub-cluster continua a operar separadamente. Nesta situação, os nós em cada partição concordam sobre a pertinência da partição a qual pertencem, mas não com os outros sub-grupos formados. Desta maneira, o serviço de pertinência associa um contexto de pertinência a cada mensagem enviada. Após um particionamento, o serviço fornece mecanismos para juntar consistentemente as partições.

5.9.2 Implementações Open-Source

Esta seção irá descrever brevemente como alguns sistemas open-source conhecidos tratam a questão da alta disponibilidade e do serviço de pertinência.

Linux High-Availability (Heartbeat)

Provavelmente o sistema de clusters mais famoso, o *linux-ha* [84, 81] é muito simplista no que se refere ao serviço de pertinência e a clusters de alta disponibilidade. Este sistema comporta somente clusters de 2 nós, onde um nó envia continuamente ao outro mensagens indicando que ele está vivo (*heartbeat*). A falta do recebimento de uma destas mensagens indica que o outro nó está defeituoso, e que não deve mais pertencer ao cluster. Desta forma, o linux-ha trata a

[10] Mais informações sobre esta camada podem ser encontradas na seção 1.5.5.

inclusão de um nó ao grupo como sendo a junção deste nó ao meio de comunicação, e a saída de um nó do cluster como sendo sua saída do meio de comunicação.

Por possuir um serviço de pertinência muito limitado, é provável que aconteça uma situação de inconsistência (síndrome do cérebro dividido), na qual os dois nós não recebem os heartbeats dos outros e os dois passam a criar sub-clusters independentes que se consideram a remanescência do cluster original. Por não possuir um Serviço de Quórum para decidir qual sub-cluster deve prosseguir com sua execução, o linux-ha baseia-se em recursos de hardware para resolver o problema. Ele utiliza dispositivos que permitem que uma máquina desligue ou reinicialize remotamente a outra. Tais dispositivos são conhecidos pela sigla STONITH (Shoot The Other Node In The Head). Entretanto, o linux-ha não garante que uma situação onde os dois nós se desligam não acontecerá.

Mission Critical Linux - Kimberlite

Igualmente ao linux-ha, o Kimberlite [5] permite somente clusters de dois nós, onde ambos enviam continuamente mensagens de *keepalive* - heartbeats - ao seu parceiro, para indicar que está operando normalmente. Ainda, não existe um algoritmo distribuído para o serviço de pertinência. Um nó é excluído do cluster se ele parar de enviar seus heartbeats ao outro nó.

Para poder resolver o impasse de uma situação de cérebro dividido, em um cluster Kimberlite deve existir um repositório estável que persiste a falhas, o qual os dois nós podem acessar. Ele é usado como um recurso de quórum, pois a partir da acessibilidade momentânea dos nós ao repositório, e a partir das informações armazenadas nele, um nó pode saber se ele deve prosseguir ou não com sua execução. Desta maneira, dispositivos STONITH não são os únicos mecanismos de se evitar sub-clusters que funcionam em paralelo.

Linux FailSafe

O FailSafe [91] foi originalmente desenvolvido para IRIX, e portado posteriormente para o Linux. Ele é o mais avançado sistema de clusters de alta disponibilidade open-source. Ele permite clusters com até 16 nós, e grandes possibilidades para lidar com falhas nos serviços oferecidos. Diferentemente do heartbeat e do Kimberlite, o FailSafe é o único a monitorar a funcionalidade de serviços oferecidos pelos clusters, e não somente se certo nó continua enviando mensagens de keepalive.

Ele possui um Serviço de Pertinência distribuído, o qual elege um nó líder para gerenciar o estabelecimento de uma nova visão no grupo. Em casos de particionamentos no grupo, o FailSafe possui um Serviço de Quórum que decide se o sub-cluster possui autorização para prosseguir sua execução. Tal decisão é baseada simplesmente no número de nós desta nova lista, de maneira que

a partição que possuir a maioria dos nós do cluster original continuará a executar. Entretanto, ele não tolera a quebra do grupo em mais do que duas partes.

Capítulo 6

Implementação do Serviço de Pertinência

Finagle's First Law:

> If an experiment works, something has gone wrong.

6.1 Introdução

Para concluir este estudo, implementaremos um serviço de pertinência, que poderá então ser utilizado como base para um cluster de alta disponibilidade.

Iniciaremos o capítulo descrevendo o algoritmo a ser implementado, de maneira a resumir suas propriedades. Em seguida, discutiremos detalhes de sua especificação. Dividiremos o estudo do protocolo em duas partes: primeiramente compreenderemos como as falhas e recuperações são detectadas; em seguida, abordaremos a lógica necessária para que o grupo de nós atinja um consenso. Por fim, trataremos da implementação realizada.

Um dos objetivos determinados para este estudo foi a criação de um serviço de pertinência que seguisse a padronização definida pela OCF. Abordaremos também neste capítulo como esta conformidade com a padronização OCF foi tratada.

Finalmente, discutiremos os resultados obtidos da implementação. Os testes realizados com os protocolos serão apresentados, e futuras melhorias no protocolo e/ou implementação serão propostos.

No que se referir ao serviço de pertinência, utilizaremos intercambiavelmente neste capítulo os termos "nós" e "processadores" .

6.2 Algoritmo

O algoritmo a ser implementado será baseado no modelo assíncrono temporizado, e foi proposto por Cristian e Schmuck em [43]. Neste texto, foram descritas cinco especificações para um serviço de pertinência de nós em um modelo temporizado sujeito a particionamentos. Ainda, foram propostos cinco protocolos distribuídos que implementam estas especificações, mesmo com um número arbitrário de falhas de colapso/performance dos nós e de omissão/performance dos canais de comunicação. Por fim, o artigo inclui provas das corretudes dos respectivos protocolos.

Assim, as características do serviço de pertinência que implementaremos são:

- Serviço de Pertinência de Nós;
- Modelo Assíncrono Temporizado;
- Particionável;
- Semântica de Falhas:
  - Nós: colapso e/ou performance.
  - Comunicação: omissão e/ou performance.

6.2.1 Detecção de Alterações no Grupo

Um serviço de pertinência pode ser observado como constituído de dois componentes: um protocolo para detectar alterações no sistema (falhas e recuperações) e um protocolo para relatar estas informações consistentemente aos nós. Antes de discutirmos a especificação do protocolo que implementaremos, discutiremos brevemente alguns métodos de se detectar falhas e recuperações de nós [43], e como este componente será implementado no protocolo em questão.

Detecção de Falhas de Nós

O que torna protocolos tolerantes a falhas assíncronos difíceis de serem compreendidos e projetados é o fato que é impossível para um nó p distinguir entre:

- o colapso de outro nó q;
- uma falha de comunicação que desconecta p e q por um longo período de tempo;
- uma falha de comunicação transiente que causa o atraso ou perda de mensagens entre p e q;

Como desejamos mascarar a falha de nós, quando p não puder se comunicar com q devemos tomar uma decisão que permita mascarar falhas reais de q. Esta decisão significa proceder na execução do protocolo como se q estivesse realmente defeituoso. Assim, p decidirá que q falhou se p não receber uma mensagem de q dentro de um intervalo de tempo definido.

Consideremos um sistema com somente dois nós, p e q. Se desejamos garantir que os nós detectam as falhas uns dos outros dentro de um intervalo de tempo definido após uma falha, basta que os dois nós troquem periodicamente mensagens. Há duas maneiras de realizar esta tarefa:

1. O nó p poderia enviar uma mensagem "você-está-vivo?" para q dentro de π unidades de tempo após que ele tome conhecimento que q está vivo. Uma falha de q levaria a uma ausência de resposta de tal mensagem dentro de 2δ unidades de tempo, onde δ é o tempo necessário para uma mensagem sair de um nó e atingir outro.

2. O nó q poderia enviar mensagens "estou-vivo" a p a cada π unidades de tempo. Uma falha de q resultaria na ausência de uma nova mensagem "estou-vivo" de q a p, dentro de $\pi + \delta$ unidades de tempo após receber a última mensagem de q.

Como a segunda abordagem necessita de um número menor de mensagens para se detectar falhas, ela será utilizada no protocolo que implementaremos.

Agora, em um sistema com n nós, onde $n > 2$, precisamos de um método mais eficiente de se detectar falhas. Se cada nó enviar uma mensagem "estou-vivo" para todos os outros, teremos a rede congestionada com $O(n^2)$ mensagens para se detectar as falhas. A maneira mais econômica de se detectar falhas nestas condições é utilizar um protocolo de *lista de presença* ou de *inspeção da vizinhança*, ambos apresentados por Cristian em [37], que refletem as idéias discutidas na seção 3.5.5. Basicamente, a idéia é fazer com que cada nó seja testado por algum(s) outro(s) nó(s), de maneira a se reduzir o número de mensagens para $O(n)$. Cristian demonstra em [37] que este é o menor número possível de mensagens para se detectar falhas de nós em um grupo com n nós.

Detecção de Recuperação de Nós

Detectar-se recuperações de nós é mais simples do que se detectar falhas. Um nó que reinicializa após um colapso pode simplesmente enviar uma mensagem a todos os outros nós para lhes avisar do evento. Como algumas mensagens podem se perder, tais avisos devem ser reenviados periodicamente. Em linhas gerais, este será o protocolo de detecção de recuperações que implementaremos.

Um ponto importante a se destacar é que os mecanismos de detecção de falhas e recuperações não são totalmente independentes do resto do protocolo de pertinência, pois eles dependem de certa concordância entre os membros do grupo. Um exemplo desta dependência é o fato que o protocolo de lista de presença que implementaremos supõe que todos os nós conhecem seus respectivos vizinhos.

6.2.2 Descrição do Protocolo

O algoritmo que será implementado é o "Protocolo de Uma Rodada" (*The One Round Protocol*). Ele permite que um nó faça parte de somente um grupo em cada momento, e não coloca limites no tamanho[1] máximo do cluster. Em um sistema onde atrasos de mensagens são aleatórios, falhas de processadores e comunicação ocorrem em momentos indeterminados, e relógios podem divergir do tempo real de maneira imprevisível, não é realista exigirmos que processadores conectados concordem em uma lista de membros no mesmo momento real. Assim, desejamos garantir somente que processadores que se unem ao mesmo grupo concordem na identidade dos membros deste grupo, sem exigir que eles detectem em tempo real as modificações ocorridas no grupo.

Modelo do Sistema e Terminologia

Cada nó possui um identificador diferente $p \in P$, e cada grupo é identificado unicamente por um identificador $g \in G$, a fim de que diferentes grupos possam existir ao longo do tempo. Ainda, existe uma ordem total em G. Seja

$$joined_t : P \longrightarrow \{true, false\}$$

o predicado (total) que, para qualquer nó p, é verdadeiro quando p pertence a um grupo no momento t, e falso caso contrário. Seja

$$group_t : P \longrightarrow G$$

o mapeamento (parcial) que fornece o grupo ao qual o nó faz parte no momento t. Isto é, se $joined_t(p)$, então $group_t(p)$ representa o identificador do grupo ao qual p pertence. Ainda, seja

$$members_t : P \longrightarrow 2^P$$

o mapeamento (parcial) que fornece a visão de p da lista de membros do grupo no momento t. Isto é, se $joined_t(p)$, então $members_t(p)$ representa a visão de p dos membros do grupo $group_t(p)$. Veja que $members_t(p)$ também pode ser definido como

$$members_t(p) = group_t^{-1}(group_t(p))$$

[1] Quantidade de nós.

O sistema consiste em um conjunto P de processadores ligados por um canal de comunicação. Todo grupo possuirá um *líder*, que será o nó com o menor identificador. Na ausência de falhas, as mensagens trocadas entre dois nós são entregues dentro de δ unidades de tempo. Cada processador possui um relógio local e uma área de armazenamento local.

Em qualquer momento no tempo, a comunicação entre dois processadores p e q pode estar em um dos seguintes estados abstratos: *conectada*, *desconectada* ou *parcialmente conectada*. Suas definições são:

- Se p e q estão *conectados* durante um intervalo de tempo $I = [t_1, t_2]$, então p e q estão corretos durante I, e toda mensagem enviada por p ou q ao outro em $[t_1, t_2 - \delta]$ é entregue e processada em δ unidades de tempo.
- Se p e q estão *desconectados* durante um intervalo de tempo I, então uma das seguintes situações ocorre:
  - p permanece travado durante todo o intervalo I;
  - q permanece travado durante todo o intervalo I;
  - nenhuma mensagem trocada entre p e q é entregue em I.
- Se p e q estão *parcialmente conectados* durante I, então nenhuma conclusão pode ser feita sobre a corretude dos processadores ou sobre a entrega de mensagens em I.

Um sistema P é dito *estável* em um intervalo I se, durante I:

1. Nenhum processador falha ou se recupera;
2. Todas as mensagens entregues em I são entregues dentro de δ unidades de tempo após seu envio;
3. Todos os pares de processadores em P são conectados ou desconectados, e
4. A relação de conexão entre processadores é constante e transitiva.

Requisitos

Os requisitos colocados para este protocolo são:

1. *Concordância na lista de membros.* Se dois processadores p e q se unem ao mesmo grupo g (não necessariamente no mesmo instante), então os dois processadores possuem a mesma

visão dos membros deste grupo. Isto é, se em qualquer momento t $joined_t(p)$ for verdadeiro, $group_t(p) = g$ e $members_t(p) = m$, e no momento t' $joined_{t'}(q)$ for verdadeiro, $group_{t'}(q) = g'$ e $members_{t'}(q) = m'$, então se $g = g'$ então $m = m'$.[2]

2. *Reflexão.* Para todo processador p, se $joined_t(p)$ então $p \in members_t(p)$.

3. *Evolução dos Grupos.* Sejam $g = group_t(p)$ e $g' = group_{t'}(p)$. Se $t < t'$, então $g < g'$.

O primeiro requisito implica que um identificador de grupo determina unicamente a lista de membros de um grupo. Portanto, escreveremos freqüentemente somente $members(g)$, para algum $g \in G$, para denotar o conjunto dos nós membros do grupo g. O segundo requisito diz que um processador pode somente se unir a grupos dos quais ele seja membro[3]. O terceiro requisito demanda que a união a grupos seja feita em uma ordem temporal consistente com a ordem dos identificadores dos grupos. Como uma conseqüência, dois processadores que se unem aos mesmos grupos g e g' se unirão a eles na *mesma ordem.*

4. *Atrasos de União Delimitados.* Existe uma constante de tempo J tal que, se dois processadores p e q estão conectados durante $I = [t, t + J]$ e o sistema P está estável durante I, então deve haver um grupo comum g, tal que p se una a g em algum momento durante I e q se une a g em algum momento durante I (não necessariamente no mesmo instante que p).[4]

5. *Atrasos de Detecção de Partições Delimitados.* Existe uma constante de tempo D tal que, se dois processadores p e q estão desconectados em $I = [t, t + D]$, o sistema P está estável durante I, e p permanece correto durante I, então deve haver um grupo g com $q \notin members_t(g)$, tal que p se une a g em algum momento durante I.[5]

6. *Estabilidade de Visões Locais.* Se o sistema permanece estável durante $I = [t, t']$, então nenhum processador deixará seu grupo em $[t + \max(J, D), t']$.[6]

Estes três últimos requisitos são equivalentes ao seguinte requisito:

[2] Este requisito define que a união de um nó p a um grupo g no momento t implica que $joined_t(p)$ é verdadeiro, e que $group_t(p) = g$.

[3] Isto significa que, quando um nó p sugerir a outro nó q a formação de um grupo g, uma das condições para q aceitar esta formação é a existência de q neste conjunto proposto.

[4] Este requisito demanda que, em um sistema estável, todos os processadores conectados devem se unir em um um novo grupo em até J unidades de tempo.

[5] De maneira análoga ao requisito 4, este requisito demanda que, em um sistema estável, todos os nós devem formar um novo grupo excluindo os nós desconectados em até D unidades de tempo após esta desconexão.

[6] Como o sistema permanece estável durante I, este requisito demanda que o serviço não deve indicar a falha de nenhum processador do grupo, pois nenhuma falha de processadores ocorre em um sistema estável.

7. *Convergência de Visões.* Existe uma constante de tempo C tal que, se o sistema permanecer estável durante $I = [t, t']$, então todos os processadores corretos em cada partição física P_i estarão unidos ao mesmo grupo g_i durante o intervalo $[t + C, t']$, e a lista de membros de g_i, $members_t(g_i)$, será igual a P_i.

Finalmente, considere um grupo g tal que $p, q \in members_t(g)$, o qual é proposto em resposta à instabilidade detectada por um processador. Se o sistema não estiver estável, é possível que p se una ao grupo g, mas q nunca o faça[7]. Diremos neste caso que g é um grupo incompleto. Em geral, dizemos que um grupo g é *incompleto* se pelo menos um processador $q \in members_t(g)$ nunca se une a g. Grupos incompletos são indesejados, pois um cluster depende que todos seus membros possuam a mesma visão da lista de membros do grupo. Os requisitos 4 e 5 garantem que não haverá nenhum grupo incompleto se o sistema permanecer estável por pelo menos $\max(J, D)$ unidades de tempo. Ainda, demandamos também que haja um limite no momento em que um processador correto pode se unir a um grupo incompleto *mesmo quando o sistema estiver instável*:

8. *Atrasos de Detecção de Inconsistência Delimitados.* Existe uma constante de tempo D' tal que, se no momento t, p se une ao grupo g tal que $q \in members_t(g)$, p permanece correto durante $I = [t, t + D']$, e em nenhum momento durante I q se une a g, então p deixa g até $t + D'$.[8]

Este requisito também implica que, quando um processador q que se uniu a um grupo g deixa g (por exemplo, devido a uma detecção de instabilidade), todos os outros membros corretos de g deixarão o grupo não mais do que D' unidades de tempo depois. Assim, o tempo durante o qual um processador p acredita estar unido ao mesmo grupo g que um processador q, quando de fato q não está unido a g, é limitado por D'.

Especificação

Podemos agora descrever o "Protocolo de Uma Rodada" que foi proposto por Cristian para satisfazer os requisitos 1 a 8, apresentados na seção anterior.

Vamos primeiramente descrever a ação que um processador p unido ao grupo g tomará em resposta ao evento de uma única falha ou recuperação. Quando p detectar tal evento através de um timeout ou através da chegada de uma mensagem "sonda" (*probe*), em vez de atualizar somente sua visão de pertinência local, p irá:

[7] Isto pode ocorrer devido a perdas de mensagens, ou devido a falhas adicionais que causem a criação de um novo grupo g', o qual q se une em vez de g.

[8] Este requisito determina que inconsistências em um grupo sejam detectadas em até D' unidades de tempo. Dizemos que um grupo g está inconsistente no momento t se existe um nó $q \in members_t(g)$ tal que $group_t(q) \neq g$.

1. Propor a criação de um novo grupo g';
2. Se unir a g';
3. Difundir g' e seus membros em uma mensagem "novo-grupo" (*new-group*).

Esta mensagem será difundida para todos os processadores, de maneira que eles também possam se unir a g'. Assim, o requisito 1 será satisfeito, uma vez que o identificador de grupo g' escolhido por p seja diferente de todos os outros identificadores de grupos gerados por p ou outros processadores. De maneira a garantir unicidade global, os identificadores de grupos são compostos de um número seqüencial inteiro e de um identificador de processador. Se $g.n$ e $g.p$ denotam o número seqüencial e o identificador do processador, podemos definir a ordem total de G como se segue:

$$g < g' \iff (g.n < g'.n) \vee (g.n = g'.n \wedge g.p < g'.p)$$

O processador p constrói então o novo identificador do grupo g' a partir de g: $g' = (g.n+1, p)$. Como mais de um processador pode detectar uma falha ou uma recuperação, um processador q pode receber várias propostas de novos grupos ao mesmo tempo. Portanto, q irá se unir ao grupo g' se este é mais *recente* do que o identificador de seu grupo atual g'', isto é, $g'' < g'$. Tal comportamento irá satisfazer o requisito 3.

O comportamento do protocolo é de certa maneira mais complexo quando múltiplas falhas ou recuperações ocorrem. Por exemplo, um processador p unido a um grupo g pode receber uma proposta de novo grupo para g' de um nó q que p sabe que estava previamente incomunicável. Se $g' < g$, p não pode se unir ao novo grupo, mas seria um erro simplesmente ignorar a informação contida na mensagem "novo-grupo". Assim, p precisa de um jeito de extrair a "nova" informação contida na mensagem. Para indicar o quão recente é esta informação, adotamos a solução de demandar que remetentes de mensagens "novo-grupo" incluam nas mensagens um conjunto de timestamps, onde para cada processador $r \in P$ indique o quão recente é a informação sobre o status deste processador. Para tanto, cada nó mantém localmente um vetor de timestamps, que é atualizado a cada recepção de uma mensagem "estou-vivo" ou "novo-grupo".

Podemos agora descrever em um pouco mais de detalhes o funcionamento do protocolo. Quando um processador p é inicializado pela primeira vez, ele se une ao grupo $g = (0, p)$, cuja lista de membros contém somente p. Este identificador é armazenado em uma memória local não volátil. Esta memória é usada para que quando o nó for reinicializado após uma falha ele se una ao grupo $(g.n + 1, p)$.

Sempre que p estiver unido a um grupo com mais de um membro, se p for o *líder do grupo* ele deve enviar a cada π unidades de tempo uma mensagem de *lista de presença* para checar o

status do grupo, e aguardar seu retorno. Os outros processadores simplesmente monitoram a passagem desta mensagem.

Enquanto unido a um grupo com estado $(g, members_t(g), s)$, onde $s \in P \to \mathbb{N}$ armazena o último timestamp $s(r)$ enviado por r, um processador p pode detectar um dos seguintes eventos:

- provável falha de seu vizinho da esquerda: p cria um novo grupo $g' = (g.n + 1, p)$ e envia uma mensagem "novo-grupo" a todos os nós. Juntamente a esta mensagem é enviado $(g', members_t(g'), s')$, onde $members_t(g') = members_t(g) - \{q\}$, e $\forall r \in P : s'(r) = s(r)$.
- uma mensagem "novo-grupo" foi recebida: quando esta mensagem é recebida com argumentos $(g', members_t(g'), s')$, ele inicia um processo de fusão conforme descrito brevemente acima.
- uma mensagem "sonda" foi recebida: de maneira a reduzir o número de mensagens envolvidas para se processar mensagens de sonda, somente o líder do grupo responde a elas. O líder processa mensagens "sonda" da mesma maneira que mensagens "novo-grupo" são processadas; os outros membros do grupo simplesmente ignoram esta mensagem.

Em linhas gerais, este é o funcionamento do protocolo. Para maiores detalhes, consulte [43]. As figuras 6.1, 6.2, 6.3, 6.4, 6.5 e 6.6 foram extraídas do mesmo texto, e contêm o pseudo-código para o protocolo.

Cristian calculou as constantes definidas nos requisitos para seu protocolo [43]. Para o "Protocolo de Uma Rodada", as constantes valem:

$$J = \max(\pi + n\delta, \mu) + 5\delta \tag{6.1}$$

$$D = \max(\pi + n\delta, \mu) + 5\delta + (n-1)(\pi + (n+1)\delta) \tag{6.2}$$

$$D' = 2(\pi + n\delta) \tag{6.3}$$

onde:

- π é o intervalo de envio de mensagens "estou-vivo", trocadas entre os nós. Estas mensagens são enviadas a cada π unidades de tempo pelo nó líder do grupo a seu vizinho da direita, que então repassa a seu vizinho da direita, até atingir novamente o nó líder, formando assim um anel lógico.

```
const    P = ProcessorSet;
type     N = integer;
         T = TimeValues;
         G = record n: N; p: P end;
const    π = NeighborPeriod;
         μ = ProbePeriod;
         δ = TimeoutDelay;
persistent var
         CurrentGid: G initially (0, myid);
var      members: set of P;
         timestamps: map P→T;
         MissingNeighbor, SendIAmAlive, SendProbe: timer;
msg      ("new-group", G, set of P, map P→T);
         ("I-am-alive", G, T);
         ("probe", G, set of P, map P→T);
```

Figura 6.1: Definição de Variáveis e Tipos

- μ é o intervalo de envio de mensagens "sonda", enviadas pelo líder do grupo de maneira a detectar a recuperação de nós.

- δ é o atraso ocorrido desde o momento de envio de uma mensagem por um nó p até ela ser entregue e processada em outro nó q.

Um outro ponto a ser discutido sobre estes parâmetros é a relação existente entre π e δ. Devemos definir π e δ de maneira que $\pi \geq n\delta$. Se π for escolhido imediatamente menor que $n\delta$, quando um processador p receber a primeira mensagem "estou-vivo", ele não poderá inferir que esta primeira mensagem foi recebida por todos os outros membros. Isto ocorre porque o líder do grupo envia a segunda mensagem "estou-vivo" antes de receber a primeira mensagem através de seu vizinho da esquerda. Portanto, para tal escolha de π, um processador p precisaria esperar pelo menos três mensagens "estou-vivo" para ter certeza de que todos os membros do grupo se uniram a g, acarretando a um limite $D' = 3(\pi + n\delta)$.

```
task Main;
var     p: processid;
        gid': G; members': set of P; timestamps': map P→T; t': T;
begin
    timestamps(myid)← LocalTime();
    for all p ∈ P − {myid} do
        timestamps(p)← 0;
    end for
    CurrentGid← (CurrentGid.n+1, myid);
    members← {myid};
    SendProbe.set(0);
    loop
        select event
            when SendIAmAlive.timeout:
                % it is time to send "I am alive" message to right neighbor
                p← RightNeighbor(myid, members)
                send ("I-am-alive", CurrentGid, LocalTime()) to p;
                SendIAmAlive.set(π);
            when MissingNeighbor.timeout:
                % "I am alive" message from left neighbor is missing
                members← members − LeftNeighbor(myid, members);
                CreateNewGroup(CurrentGid);
            when SendProbe.timeout:
                % it is time to send a probe message
                timestamps(myid)← LocalTime();
                send ("probe", CurrentGid, members, timestamps) to P − members;
                % Note that probes are sent only by the group leaders of minority
                    groups because in procedure Join, the SendProbe timer is set only
                    if this processor is the leader of a minority group
                SendProbe.set(μ);
            when receive ("new-group", gid', members', timestamps') from p
                NewGroupInfo(gid', members', timestamps');
```

Figura 6.2: Laço principal do Protocolo - parte 1

```
            when receive ("I-am-alive", gid', t') from p:
                if gid' = CurrentGid then
                    timestamps(p)← t';
                    MissingNeighbor.set(π + Rank(myid, members)*δ);
                    if myid ≠ Leader(members) then
                        p← RightNeighbor(myid, members)
                        send ("I-am-alive", CurrentGid, LocalTime()) to p;
                    end if;
                end if
            when receive ("probe", gid', members', timestamps') from p:
                NewGroupInfo(gid', members', timestamps');
        end select
    end loop
end
```

Figura 6.3: Laço principal do Protocolo - parte 2

```
procedure NewGroupInfo(gid': G; members': set of P; timestamps': map P→T)
var     MyInfoUptodate: boolean initially true;
        OtherInfoUptodate: boolean initially true;
        p: processid;
begin
    % Merge my knowledge of the state of the system with the new information
    for all p ∈ P do
        if p ∉ members and p ∈ members' then
            if timestamps(p) < timestamps'(p) then
                members← members ∪ {p};
                MyInfoUptodate← false;
            else
                OtherInfoUptodate← false;
            end if
        else if p ∈ members and p ∉ members' then
            if timestamps(p) ≤ timestamps'(p) and p ≠ myid then
                members← members − {p}
                MyInfoUptodate← false;
```

Figura 6.4: Função `NewGroupInfo` - parte 1

```
                else
                    OtherInfoUptodate← false;
                end if
            end if
            timestamps(p)← max(timestamps(p), timestamps'(p));
        end for
        if CurrentGid < gid' then
            if OtherInfoUptodate then
                Join(gid');
            else
                CreateNewGroup(gid');
            end if
        else if CurrentGid > gid' then
            if MyInfoUptodate then
                % ignore
            else
                CreateNewGroup(CurrentGid);
            end if
        end if
    end
```

Figura 6.5: Função `NewGroupInfo` - parte 2

Vantagens e Desvantagens

O protocolo é muito eficiente para tratar a falha ou recuperação de um único nó: a falha ou recuperação de um único nó resulta na criação de um novo grupo com uma simples rodada de n mensagens em uma rede ponto-a-ponto. Ainda, o protocolo é útil para se implementar um serviço que é freqüentemente utilizado para evitar a "síndrome do cérebro dividido", o *Serviço de Líder Altamente Disponível* [9].

[9] Em linhas gerais, este serviço garante que:

- em qualquer momento há somente um líder, e
- existe uma constante E tal que, se todo sistema P está estável em $I = [t, t + E]$ e uma maioria dos processadores está conectada, então haverá um líder em $t + E$.

Ao impossibilitarmos a existência de dois líderes ao mesmo tempo, podemos evitar o comportamento de "cérebro dividido", apresentado na seção 1.5.5.

```
procedure CreateNewGroup(oldid: G)
var      newid: G
begin
    newid← (oldid.no + 1, myid);
    timestamps(myid)← LocalTime();
    send ("new-group", newid, members, timestamps) to P − {myid};
    Join(newid);
end

procedure Join(newid: G)
begin
    % update current group id
    CurrentGid← newid;
    % cancel all timers
    SendProbe.cancel();
    SendIAmAlive.cancel();
    MissingNeighbor.cancel();
    % set new timers
    if Leader(members) = myid and ¬ Majority(members) then
        SendProbe.set(μ);
    end if
    if |members| > 1 then
        if Leader(members) = myid then
            SendIAmAlive.set(π − δ);
        end if;
        MissingNeighbor.set(π + Rank(myid, members)*δ);
    end if
end
```

Figura 6.6: Funções relacionadas à criação de novos grupos

Apesar destes pontos positivos, o protocolo possui algumas desvantagens. Após múltiplas falhas, recuperações ou perdas de mensagens, o protocolo pode necessitar de tempo da ordem de n^2 para se estabilizar [43], onde n é o número de nós no grupo. É possível também que um certo nó não perceba *todas* as falhas e recuperações que ocorreram. Por exemplo, seja g um grupo com os nós participantes $\{p_1, p_2, p_3, p_4\}$. Se o processador p_4 falhar e se recuperar rapidamente, enquanto o processador p_5 (que havia falhado anteriormente) se recupera, é possível que o processador p_1 observe a seqüência de grupos g' e g'', tal que $members_{t'}(g') = \{p_1, p_2, p_3, p_4\}$ e $members_{t''}(g'') = \{p_1, p_2, p_3, p_4, p_5\}$. Assim, p_1 não perceberia que p_4 havia falhado e então se

recuperado. A tabela 6.1 resume as vantagens e desvantagens do protocolo.

Vantagens	**Desvantagens**
Muito eficiente para tratar falhas ou recuperações de um único nó.	Após múltiplas falhas, recuperações e perdas de mensagens, o protocolo pode levar um longo tempo para se estabilizar.
Útil para se evitar a *síndrome do cérebro dividido*	É possível que um nó correto não perceba *todas* as falhas e recuperações que ocorreram.

Tabela 6.1: Vantagens e Desvantagens do Protocolo

6.2.3 Implementação

O protocolo foi implementado em C, no sistema operacional Linux. Como não seria possível codificar somente o protocolo, foram criadas várias rotinas complementares, como uma camada de comunicação, definições dos tipos básicos em um cluster (como o nó e o grupo), mecanismos de tratamento de logs, etc. Para facilitar o desenvolvimento e diminuir os riscos do projeto, foram utilizadas várias bibliotecas auxiliares. Além de aumentar a portabilidade do código para outras plataformas, o uso destas bibliotecas permitiu concentrar os esforços de desenvolvimento no serviço de pertinência, otimizando assim o processo de desenvolvimento.

A camada de comunicação funciona com TCP ou UDP (inclusive multicast), de acordo com as opções definidas no momento de compilação. A configuração padrão e recomendada é realizar a comunicação ponto-a-ponto por UDP puro, e a difusão de mensagens a um grupo de nós por multicast. O formato escolhido para transferir dados entre os nós foi XML. Existem algumas justificativas para esta escolha:

1. Facilidade de se depurar mensagens escritas em XML;
2. Existência de bibliotecas para manipular XML;
3. Potencial facilidade de portabilidade e interoperabilidade do serviço.

A implementação seguiu diretamente o pseudo-código apresentado por Cristian e Schmuck, e, de certo modo, o protocolo está codificado em alto nível. Devido a uma boa separação de camadas, os módulos do programa são um tanto independentes uns dos outros, e suas funcionalidades são bem separadas e isoladas. Ainda, a documentação de todas as funções, tipos, macros e variáveis deixa o código mais claro e coeso. Devido a todos estes fatores, o código é bem direto e fácil de ser compreendido.

A tabela 6.2 resume alguns dados sobre a implementação.

Linguagem de Programação	C, compilado com GCC 3.2.2 20030222
Sistema Operacional	Linux 2.4.20-19.9 (RedHat 9.0)
Protocolo de Comunicação	TCP ou UDP (inclusive multicast)
Quantidade Total de Linhas	aprox. 6100
Tamanho do Executável	aprox. 61Kb (i386)
Bibliotecas usadas e suas licenças	glib 2.2.3 (LGPL) - dinâmica gnet 2.0.4 (LGPL) - dinâmica libconfuse 2.2 (LGPL) - dinâmica libxml 2.5.11 (MIT) - dinâmica
Outras ferramentas	doxygen 1.2.18: geração automática da documentação. ddd 3.3.5: depuração gráfica do programa.

Tabela 6.2: Detalhes da Implementação

6.2.4 Conformidade com a OCF

Um dos objetivos deste projeto era produzir um serviço de pertinência que seguisse a especificação da OCF. Desta maneira, a implementação seguiria um padrão, e poderia ser utilizada em diversos ambientes e situações. Ainda, a implementação se beneficiaria dos outros componentes de infra-estrutura do framework, eliminando esforços em outras áreas fora do escopo do Serviço de Pertinência (como no desenvolvimento de uma camada de comunicação, por exemplo).

Este objetivo do projeto dependia totalmente do desenvolvimento das especificações e do framework propostos pela OCF. Ocorre que, neste momento, a OCF não produziu nenhuma especificação fechada para o serviço de pertinência (uma proposta inicial é descrita na seção B.3.2), nem produziu nenhum framework básico para amarrar os componentes. Na realidade, mesmo as especificações mais discutidas pela OCF não se encontram totalmente fechadas. O movimento na lista de discussão da OCF é muito sazonal, e está muito baixo no momento. Todos os documentos que a OCF produziu se encontram no apêndice B, o que ilustra o fraco movimento no projeto.

Assim, a estratégia para prosseguir com este projeto de mestrado foi continuar com o desenvolvimento de um serviço de pertinência, mas sem seguir nenhuma especificação padrão, nem usar nenhum framework comum. Desta maneira, quando a OCF produzir o material necessário, este projeto poderá ser reescrito para seguir as especificações. Provavelmente não será necessário reescrever todo o código, mas somente a parte que trata do protocolo. Devido à boa separação de camadas mencionada acima, este retrabalho não deve ser grande, o que ainda torna esta implementação uma candidata para rodar em um framework padrão.

6.3 Resultados

Esta seção visa apresentar e discutir os resultados obtidos a partir da implementação do serviço de pertinência.

6.3.1 Tamanho das Mensagens

Uma das medições efetuadas foi calcular o tamanho das mensagens trafegadas pelo serviço de pertinência. Primeiramente, vamos definir as seguintes variáveis:

- n: quantidade de nós integrantes do grupo;
- p: quantidade máxima de participantes do grupo;
- h: tamanho do nome de um nó (*hostname*);
- s: tamanho do número seqüencial que compõe o gid;
- m_{id}: tamanho do número identificador de uma mensagem.

Seja $t : M \rightarrow \mathbb{N}$ a função que calcula o tamanho em bytes de uma mensagem $m \in M$. Com as definidas acima feitas, podemos calcular o tamanho das mensagens trocadas. Apesar do protocolo trocar três tipos de mensagens entre o grupo (`Probe`, `I-Am-Alive` e `New-Group`), temos somente duas variações de documentos XML trocados entre os nós:

1. Alive: representa a mensagem `I-Am-Alive`. O tamanho aproximado em bytes de uma mensagem m do tipo `alive` é representado pela equação 6.4.

$$t(m) = 326 + m_{id} + 2h + 3\lfloor \log_{10} p \rfloor + s \tag{6.4}$$

A figura 6.7 exibe um exemplo de documento XML `alive`.

```
<?xml version="1.0"?>
<message id="13">
  <header>
    <from>
      <node id="1">
        <hostname>nalvesp.ath.cx</hostname>
        <port>30001</port>
      </node>
    </from>
    <recipient>
      <type>1</type>
      <node id="2">
        <hostname>nalvesp.ath.cx</hostname>
        <port>30002</port>
      </node>
    </recipient>
    <type>2</type>
  </header>
  <data>
    <timestamp>1063074467</timestamp>
    <group-id sequence="14" node-id="1"/>
  </data>
</message>
```

Figura 6.7: Exemplo de mensagem `Alive`

2. Membership: representa as mensagens `Probe` e `New-Group`. O tamanho aproximado em bytes de uma mensagem m do tipo `membership` é representado pela equação 6.5.

$$t(m) = 342+m_{id}+s+\lfloor \log_{10} n * (p-1) \rfloor +(n+p)*(59+h+\lfloor \log_{10} p \rfloor)+(p+1)*(30+\lfloor \log_{10} p \rfloor) \quad (6.5)$$

As figuras 6.8 e 6.9 exibem um exemplo de documento XML `membership`.

```
<?xml version="1.0"?>
<message id="11">
  <header>
    <from>
      <node id="1">
        <hostname>nalvesp.ath.cx</hostname>
        <port>30001</port>
      </node>
    </from>
    <recipient>
      <type>2</type>
      <group>
        <size>3</size>
        <members>
          <node id="2">
            <hostname>nalvesp.ath.cx</hostname>
            <port>30002</port>
          </node>
          <node id="3">
            <hostname>nalvesp.ath.cx</hostname>
            <port>30003</port>
          </node>
          <node id="4">
            <hostname>nalvesp.ath.cx</hostname>
            <port>30004</port>
          </node>
        </members>
      </group>
    </recipient>
    <type>3</type>
  </header>
```

Figura 6.8: Exemplo de mensagem `New-Group` e `Probe` - parte 1

```
<data>
  <membership>
    <group-id sequence="6" node-id="1"/>
    <group>
      <size>1</size>
      <members>
        <node id="1">
          <hostname>nalvesp.ath.cx</hostname>
          <port>30001</port>
        </node>
      </members>
    </group>
    <timestamps>
      <node id="1">1063074153</node>
      <node id="2">0</node>
      <node id="3">0</node>
      <node id="4">0</node>
    </timestamps>
  </membership>
</data>
</message>
```

Figura 6.9: Exemplo de mensagem `New-Group` e `Probe` - parte 2

Para ter uma idéia de quanto estes valores representam, em uma situação onde $n = 50$, $p = 100$, $h = 32$, $s = 10$ e $m_{id} = 10$, uma mensagem `Alive` terá aproximadamente 420 bytes, e uma mensagem `Probe` ou `New-Group` 17500 bytes. Consideramos que estas mensagens são grandes, e isto se deve principalmente ao uso de XML. Na seção 6.4 discutiremos maneiras de se diminuir o tamanho destas mensagens.

6.3.2 Testes

O problema de se garantir a corretude da implementação de um serviço distribuído tolerante a falhas é muito difícil. Técnicas como verificação formal e modelagem analítica simplesmente não são adequadas para sistemas tolerantes a falhas complexos [14]. Além de observar a aplicação em cenários normais, as abordagens mais viáveis implicam em injetar falhas durante sua execução, a fim de se verificar o comportamento do serviço nos cenários de falha para os quais ele foi

projetado. Entretanto, existem algumas dificuldades consideráveis em se testar as propriedades dos protocolos em plataformas distribuídas [14]:

- inacessibilidade ao estado global do sistema;
- ocorrência de falhas indesejadas;
- falta de controle sobre a evolução dos experimentos de teste,

Como muitas propriedades de protocolos existentes são impossíveis de serem testadas em um sistema distribuído, a maneira mais adequada para realizar estes testes seria utilizar um simulador central, como o descrito em [14]. Todos os processos do sistema distribuído deveriam ser executados no mesmo espaço de endereçamento, de maneira transparente para cada processo. Este simulador seria mais poderoso do que um sistema distribuído real, pois ele permitiria um controle muito maior sobre o ambiente. Falhas poderiam ser injetadas e seus efeitos corretamente observados.

Após uma busca por tal simulador central, não encontramos nenhuma solução que pudesse ser acoplada ao nosso serviço de pertinência sem demandar novo desenvolvimento. Assim, esta técnica não poderia ser utilizada durante as medições. A solução mais adequada encontrada foi executar os processos isoladamente, e depois analisar os logs gerados pelo serviço, através de um programa desenvolvido para este fim.

Ambiente de Testes

Como plataforma de testes, algumas opções foram analisadas:

1. Rodar o serviço em um grupo de máquinas distintas, interligadas por uma rede local. Esta configuração representaria um cluster real, e poderia fornecer medidas reais da execução do protocolo.
2. Executar o serviço localmente em uma máquina, dentro de instâncias separadas que rodam *User Mode Linux* [44]. Este ambiente, apesar de local, poderia apresentar um comportamento similar ao de um cluster real, pois cada nó seria representado por uma instância UML.
3. Executar o serviço localmente em uma máquina, rodando vários processos. Cada processo escutaria em uma porta distinta, de maneira que cada um representasse um nó no cluster. Este ambiente proporcionaria um maior controle da execução do sistema, possibilitando uma análise mais apurada dos resultados.

A escolha da plataforma seguiu as opções acima. A plataforma 1 avaliada foi a rede LCPD[10] do IME[11]. Esta rede é composta de 7 máquinas linux, cada uma sendo um Pentium II 400Mhz, com 512Mb de memória RAM. Quando esta plataforma foi avaliada, alguns problemas surgiram:

- Falta de sincronização nos relógios: como as medições seriam feitas com posterior análise dos logs gerados, os relógios deveriam estar sincronizados para fornecer uma medição correta. Sem esta sincronização, várias situações de inconsistência ocorreram durante os estudos deste ambiente.

- Picos de carga comprometem os testes: por não controlarmos o ambiente de execução, picos momentâneos de carga na máquina e na rede local ocorrem freqüentemente. Estes picos desviam os valores dos testes, pois em uma situação real um cluster deve ser bem dimensionado, e preferencialmente deve possuir um canal de comunicação privado.

Devido a estes problemas, a plataforma 1 foi descartada, pois não seria possível obtermos medições qualitativas da implementação. Em seguida, analisamos a plataforma 2. Nesta plataforma existe um controle maior do ambiente, mas ainda não temos uma sincronização exata[12] dos relógios, apesar dos desvios verificados serem menores. Os picos de carga podem ser controlados, mas com o UML geramos uma carga muito grande na máquina: para rodar cada instância de um UML é necessária muita memória e processamento, o que acaba gerando um cluster de máquinas lentas. Mais uma vez, muitas inconsistências em testes com esta plataforma foram detectadas, inviabilizando também a solução. Finalmente, testamos a plataforma 3. As principais vantagens desta plataforma são a sincronização exata dos relógios, controle do ambiente e controle dos picos de carga. Entretanto, uma desvantagem seria que ela não representa fielmente um cluster, podendo oferecer resultados incompatíveis com um cenário real. Devido a todas estas características, escolhemos esta plataforma para testes. De maneira a garantir que os resultados dos experimentos refletissem características do protocolo sem a influência do ambiente, os experimentos foram executados em duas máquinas distintas, a `kofer` e a `nalvesp`. A tabela 6.3 descreve estas duas máquinas.

Experimentos

Nas seções abaixo descreveremos algumas medições e análises efetuadas, e discutiremos os resultados obtidos.

[10] Laboratório de Computação Paralela e Distribuída.

[11] Instituto de Matemática e Estatística, Universidade de São Paulo.

[12] Uma sincronização exata de relógios seria aquela onde os relógios estão sincronizados na unidade de milissegundos.

	kofer	**nalvesp**
processador	Athlon XP 2000	Athlon 1.3 Ghz
memória	512 Mb	512 Mb
sistema operacional	Linux (Debian)	Linux (Red Hat)
kernel	2.4.23-2	2.4.20-18.9

Tabela 6.3: Ambientes de Testes

1. **Alta Disponibilidade**

Vários testes iniciais visaram simplesmente exibir o comportamento da implementação. Estes testes demonstraram uma característica do protocolo não muito desejável para sistemas de alta disponibilidade. Para ilustrá-la, vamos demonstrar a execução do protocolo quando um novo nó 3 é iniciado, e começa então a execução para fazer parte do grupo. Esta situação é exibida na figura 6.10.

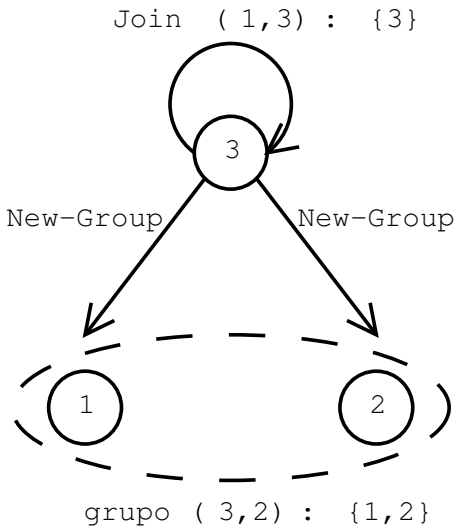

Figura 6.10: Teste da Implementação - inicialização de um novo nó

Após o recebimento da mensagem `New-Group` enviada pelo nó 3, os nós 1 e 2 criam um novo grupo que contém os nós $1, 2, 3$, e então difundem esta informação para que todo o grupo fique sabendo do novo grupo. Esta situação é ilustrada na figura 6.11. Repare que os nós 1 e 2 propõem novos grupos independentemente.

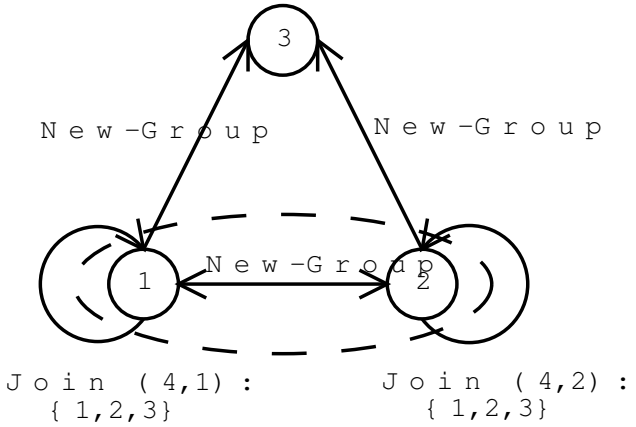

Figura 6.11: Teste da Implementação - reorganização do grupo

Ao receber as mensagens `New-Group` enviadas pelos nós 1 e 2, o nó 3 monta o grupo $(4, 2)$, conforme ilustrado na figura 6.12. Isto ocorreu pois os grupos se estabilizam com o maior identificador do grupo proposto.

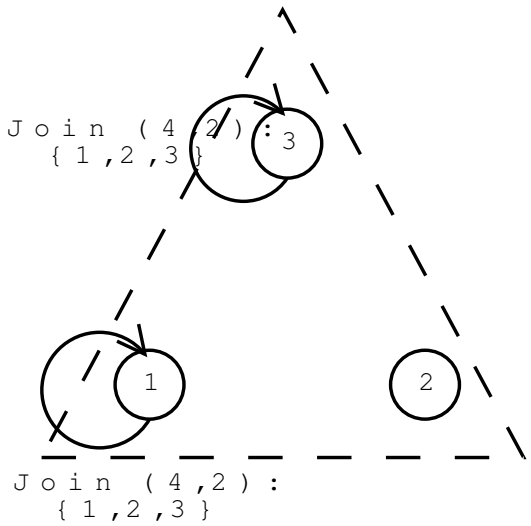

Figura 6.12: Teste da Implementação - formação do novo grupo

Note que apesar de já ter se unido ao grupo $(4, 1)$, o nó 1 passa a fazer parte agora do grupo $(4, 2)$. Isto ocorre pois seu identificador de grupo anterior era "menor" do que o novo proposto. Em um sistema de alta disponibilidade, alterações no grupo disparam outros tipos de eventos, como a reinicialização de um servidor. Se não tratarmos melhor os avisos, a união do nó 1 ao grupo $(4, 2)$ poderia disparar mais uma vez uma reorganização dos recursos. Na realidade, os membros do grupo não foram alterados: somente seu identificador

sofreu alteração. Assim, para integrar esta implementação do serviço de pertinência com um sistema de alta disponibilidade é necessário compreender se devemos avisar qualquer modificação no grupo, ou somente aquelas que alterem sua formação. Caso venhamos a decidir pela última opção, devemos criar um mecanismos para filtrar as mudanças de identificador, e avisar somente quando algum nó deixar ou entrar no grupo.

A tabela 6.4 exibe um rastreamento destes eventos ocorridos e das ações disparadas como resposta.

instante	visão 1	visão 2	visão 3	evento	ação
1	(3,2): 1,2	(3,2): 1,2	-	inicialização nó 3	nó 3: `Join(1,3)`
2	(3,2): 1,2	(3,2): 1,2	(1,3): 3	nós 1,2 recebem `New-Group` do nó 3	nós 1,2 propõem novos grupos
3	(4,1): 1,2,3	(4,2): 1,2,3	(1,3): 3	nós 1,3 recebem `New-Group` do nó 2	nós 1,3: `Join(4,2)`
4	(4,2): 1,2,3	(4,2): 1,2,3	(4,2): 1,2,3	-	-

Tabela 6.4: Rastreamento da Execução do Protocolo

2. **Tempo de Estabilização**

   Um importante teste realizado com a implementação foi verificar o tempo necessário para a estabilização dos grupos. A idéia deste teste é verificar como este tempo cresce com o aumento do grupo.

   Os experimentos visaram determinar o tempo de estabilização do grupo após ele sofrer alguma modificação. Esta modificação consistia tanto da inicialização ou finalização de algum nó. Desta forma, os experimentos foram divididos em *inicialização* e *finalização* dos nós. Em cada uma destas modalidades, aplicamos estas operações em duas formas: *do menor para o maior nó*, e *do maior para o menor nó*, de acordo com a ordem imposta pelo protocolo entre os nós. Ainda, quando possível, realizamos a operação de maneira simultânea, a fim de observar o comportamento em situações extremas.

   Desta forma, realizamos 5 experimentos no total:

   (a) Inicialização

      i. Menor para Maior

      ii. Maior para Menor

   (b) Finalização

i. Menor para Maior
ii. Maior para Menor
iii. Múltipla

Cada experimento foi executado 10 vezes em cada ambiente, onde em cada rodada foram montados grupos com 100 nós. Os resultados finais fornecem as médias e desvios padrão de cada medida. As medidas coletadas foram:

(a) O tempo necessário para estabilizar o conjunto de nós que pertencem ao grupo.
(b) O tempo necessário para estabilizar o identificador usado para representar este conjunto em questão (`gid`).
(c) A quantidade de alterações efetuadas no `gid` até a estabilização.

Os parâmetros do algoritmos apresentados na página 133 foram definidos da seguinte maneira durante os experimentos:

- π: 40 segundos
- μ: 5 minutos
- δ: 400 milissegundos

Coleta dos Resultados

Os experimentos foram todos executados localmente em uma máquina, de maneira que pudéssemos ter um relógio sincronizado para todos os nós. Cada nó foi representado por um processo rodando em uma determinada porta, gerando como saída um arquivo de log separado dos outros nós. De maneira a evitar interferências de outros processos no experimento, cada processo foi executado alta prioridade (definida em -15), e com a paginação desabilitada (implementado através da chamada `mlockall`).

Como dito anteriormente, executamos 10 vezes cada experimento. Ao final de cada uma destas rodadas, todos os logs foram unidos em um único arquivo, e tiveram seus registros ordenados de acordo com o momento de ocorrência do evento. Finalmente, scripts desenvolvidos em Perl foram executados sobre este "*logão*", a fim de se processar os dados e obter as medidas que estavam sendo buscadas. Após testes iniciais, percebemos que o desempenho da máquina era prejudicada pela execução simultânea dos 100 nós, causando efeitos colaterais nos resultados. Por exemplo, o tempo necessário para receber uma certa mensagem era afetado fortemente, o que causava um maior tempo de estabilização. Para corrigir estes problemas e observar valores que representassem um cluster real, algumas ações foram estudadas:

- Tentamos identificar alguma chamada de sistema que pudesse fornecer o tempo que o processo efetivamente ficou no processador, de maneira a usá-la para determinar o tempo de estabilização. Chamadas como `times` e `clock` foram estudadas, mas elas não forneciam o comportamento desejado, não sendo assim possível implementar este tipo de estratégia.
- Verificamos que os tempos de recebimento das mensagens eram os grandes fatores que contribuíam para um tempo de estabilização anormal. Assim, uma nova estratégia seria "normalizar" o tempo que as mensagens levavam para serem recebidas. Esta estratégia foi implementada nos scripts de coleta dos dados: durante a análise, alguns valores seriam alterados para contemplar esta modificação. Como conseqüência desta alteração, a análise dos logs passa a indicar que todos os nós recebem as mensagens simultaneamente, o que é um comportamento esperado se fizéssemos um experimento controlado em um cluster.

Nas seções abaixo descreveremos em detalhes os experimentos efetuados, fornecendo alguns[13] resultados obtidos. Apresentaremos uma tabela para cada teste, com um gráfico [14] correspondente. Neste gráfico, os grupos descritos são aqueles encontrados em suas respectivas tabelas, de acordo com o identificador utilizado. A unidade de tempo utilizada nas tabelas e nos gráficos é milissegundos, com uma precisão de três casas decimais[15]. Como um nó pode se unir a mais de um grupo (`gid`) até se estabilizar, exibimos os tempos necessários para a estabilização do conjunto de membros e para a estabilização do `gid` de cada nó, além da quantidade de alterações efetuadas em seu `gid` até a estabilização. A amostra de resultados exibida abaixo foi obtida no ambiente `nalvesp`.

(a) Tempo de Inicialização. Gostaríamos de compreender com este experimento qual é o tempo necessário para os nós se estabilizarem após a inicialização de um outro nó (ou seja, união de um nó ao grupo). Nos experimentos abaixo variamos a forma de criar os grupos, de maneira a verificar se o tempo é afetado por estas mudanças.

Para efeitos desta análise, consideramos que um nó foi inicializado após ele ter levantado todos os serviços de infra-estrutura[16], e estar apto a começar a executar o algoritmo.

i. Inicialização do menor para o maior nó. O algoritmo necessita de uma definição de precedência entre os nós. De acordo com a implementação desta definição de

[13] Por serem muito extensos, nem todos os resultados serão inseridos neste texto. Para obter todos os resultados completos dos experimentos executados, consulte [74].

[14] Devido à dissertação ter sido impressa em preto e branco, os gráficos encontram-se na versão final sem cores. Cópias coloridas podem ser obtidas em [74] e [1].

[15] Durante os experimentos geramos os tempos com precisão de microsegundos, o que possibilitou tal precisão.

[16] Camada de comunicação, mecanismos de log, leitura do arquivo de configuração, etc.

precedência, fizemos um primeiro teste onde iniciamos os nós do menor para o maior. Nos resultados abaixo, o grupo i representa o grupo formado pelos nós $1 \ldots i$.

A tabela 6.5 (página 155) contém uma amostra dos resultados do tempo necessário para estabilizar o *conjunto de membros* em cada nó. A figura 6.13 (página 156) ilustra graficamente alguns resultados deste experimento. Por esta figura, é fácil perceber que, para cada grupo, o último nó a se estabilizar é aquele recém inicializado.

A tabela 6.6 (página 157) contém uma amostra dos resultados do tempo necessário para estabilizar o *identificador do grupo* (`gid`) em cada nó. A figura 6.14 (página 158) ilustra graficamente alguns resultados deste experimento.

Após a geração destes dados, percebemos que o desvio padrão estava muito alto. Deste modo, a média e o desvio padrão não serviam para representar bem o comportamento da implementação. Assim, reformatamos estes resultados para utilizar a mediana e o valor máximo de cada experimento. A tabela 6.7 (página 159) contém uma amostra dos resultados (mediana e valores máximos) do tempo necessário para estabilizar o `gid` em cada nó. A figura 6.15 (página 160) ilustra graficamente alguns resultados deste experimento. Por esta figura, é fácil perceber que, para cada grupo i, o primeiro nó a se estabilizar é o nó $i - 1$.

A tabela 6.8 (página 161) contém uma amostra dos resultados da quantidade de alterações efetuadas no `gid` até atingir a estabilização. A figura 6.16 (página 162) ilustra graficamente alguns resultados deste experimento. De maneira similar à figura 6.15, podemos perceber por este gráfico que para cada grupo i, o primeiro nó a se estabilizar é o nó $i - 1$.

ii. Inicialização do maior para o menor nó. Em seguida, realizamos um experimento onde a inicialização dos nós foi feita começando-se pelo maior nó, e acabando com o menor nó. Neste modelo de experimento, o grupo i representa o grupo formado pelos nós $(100 - i + 1) \ldots 100$.

A tabela 6.9 (página 163) contém uma amostra dos resultados do tempo necessário para estabilizar o *conjunto de membros* em cada nó. A figura 6.17 (página 164) ilustra graficamente alguns resultados deste experimento.

A tabela 6.10 (página 165) contém uma amostra dos resultados do tempo necessário para estabilizar o *identificador do grupo* (`gid`) em cada nó. A figura 6.18 (página 166) ilustra graficamente alguns resultados deste experimento.

A tabela 6.11 (página 167) contém uma amostra dos resultados da quantidade de alterações efetuadas no `gid` até atingir a estabilização. A figura 6.19 (página 168) ilustra graficamente alguns resultados deste experimento.

(b) Tempo de Finalização. Gostaríamos de compreender com este experimento qual é o

tempo necessário para os nós se estabilizarem após a finalização (ou seja, desligamento) de um outro nó. Nos experimentos abaixo variamos a forma de criar os grupos, de maneira a verificar se o tempo é afetado por estas mudanças. Note que o último grupo deste teste deve possuir um elemento, de maneira que consigamos determinar o tempo necessário para a estabilização.

Para efeitos desta análise, consideramos que um nó foi finalizado após ele ter parado todos os serviços relacionados ao protocolo.

i. Finalização do menor para o maior nó. Seguindo a definição da precedência entre os nós, fizemos um primeiro teste onde, a partir de um grupo pré-estabelecido, finalizamos os nós do menor para o maior. Nos resultados abaixo, o grupo i representa o grupo formado pelos nós $i+1 \ldots 100$, ou seja, o grupo resultante da finalização do nó i.

   A tabela 6.12 (página 169) contém uma amostra dos resultados do tempo necessário para estabilizar o *conjunto de membros* em cada nó. A figura 6.20 (página 170) ilustra graficamente alguns resultados deste experimento.

   A tabela 6.13 (página 171) contém uma amostra dos resultados do tempo necessário para estabilizar o *identificador do grupo* (`gid`) em cada nó. A figura 6.21 (página 172) ilustra graficamente alguns resultados deste experimento. Repare que a curva do tempo de estabilização do `gid` segue o mesmo comportamento da curva do tempo de estabilização do conjunto de membros (figura 6.20).

   A quantidade de alterações que o `gid` sofreu se mostrou constante em todo este experimento: todos os nós precisaram trocar somente uma vez seu `gid` para se estabilizar. Por este motivo, omitiremos as tabelas e gráficos que correspondem a estes resultados.

ii. Finalização do maior para o menor nó. Em seguida, realizamos um teste onde, novamente a partir de um grupo pré-estabelecido, finalizamos os nós, começando pelo maior, e acabando com o menor nó. Nos resultados abaixo, o grupo i representa o grupo formado pelos nós $1 \ldots 100 - i$, ou seja, o grupo resultante da finalização do nó $100 - i + 1$.

   A tabela 6.14 (página 173) contém uma amostra dos resultados do tempo necessário para estabilizar o *conjunto de membros* em cada nó. A figura 6.22 (página 174) ilustra graficamente alguns resultados deste experimento. Por esta figura é fácil perceber que, conforme os maiores nós são finalizados, a estabilização torna-se cada vez mais rápida.

   A tabela 6.15 (página 175) contém uma amostra dos resultados do tempo necessário para estabilizar o *identificador do grupo* (`gid`) em cada nó. A figura 6.23 (página 176) ilustra graficamente alguns resultados deste experimento. Repare que a curva do tempo de estabilização do `gid` segue o mesmo comportamento da

curva do tempo de estabilização do conjunto de membros (figura 6.22).
A quantidade de alterações que o `gid` sofreu se mostrou constante em todo este experimento: quase todos os nós precisaram trocar somente uma vez seu `gid` para se estabilizar, enquanto pouquíssimos tiveram uma média de 1.1 troca até a estabilização. Por este motivo, omitiremos as tabelas e gráficos que correspondem a estes resultados.

iii. Finalização Múltipla. Testamos ainda uma situação onde todo o grupo é finalizado ao mesmo tempo.
A tabela 6.16 (página 176) contém os resultados do tempo necessário para atingir a estabilização em cada grupo. Os resultados apresentados representam tanto o tempo de estabilização do conjunto de membros, como o tempo de estabilização do `gid`.

Seguem abaixo as tabelas e gráficos dos resultados dos experimentos acima descritos. Estes dados serão expostos até a página 176.

Grupos	Nós						
	15	**30**	**45**	**60**	**75**	**90**	**100**
0	-	-	-	-	-	-	-
15	110.5 ± 0.1	-	-	-	-	-	-
20	66.2 ± 0.1	-	-	-	-	-	-
25	70.5 ± 12.6	-	-	-	-	-	-
30	66.9 ± 0.1	112.1 ± 0.2	-	-	-	-	-
35	67.4 ± 0.1	67.4 ± 0.1	-	-	-	-	-
40	67.9 ± 0.2	67.8 ± 0.1	-	-	-	-	-
45	68.3 ± 0.1	68.2 ± 0.1	113.6 ± 0.1	-	-	-	-
50	68.6 ± 0.1	68.7 ± 0.2	68.6 ± 0.2	-	-	-	-
55	69.1 ± 0.1	69.4 ± 0.5	69.1 ± 0.1	-	-	-	-
60	69.5 ± 0.2	69.5 ± 0.1	69.5 ± 0.1	115.0 ± 0.3	-	-	-
65	70.1 ± 0.4	70.2 ± 0.5	69.9 ± 0.3	69.9 ± 0.0	-	-	-
70	71.0 ± 0.4	71.0 ± 0.6	70.7 ± 0.5	70.6 ± 0.4	-	-	-
75	73.4 ± 6.2	73.2 ± 6.1	73.1 ± 5.8	73.1 ± 6.2	118.6 ± 6.0	-	-
80	72.1 ± 0.5	72.2 ± 0.4	72.1 ± 0.4	72.0 ± 0.5	71.9 ± 0.5	-	-
85	75.7 ± 10.4	75.7 ± 10.4	75.6 ± 10.3	75.4 ± 10.5	75.3 ± 10.4	-	-
90	81.9 ± 18.7	77.5 ± 14.2	77.5 ± 14.2	77.4 ± 14.2	77.4 ± 14.1	118.3 ± 0.2	-
95	90.7 ± 39.7	90.9 ± 39.8	90.8 ± 39.6	90.8 ± 39.7	90.4 ± 39.9	86.2 ± 38.8	-
100	104.4 ± 34.6	104.4 ± 34.5	104.4 ± 34.5	104.4 ± 34.6	104.2 ± 34.3	104.3 ± 34.6	140.9 ± 35.6

Tabela 6.5: Tempo de Estabilização - Membros - Inicialização dos nós do 'menor' para o 'maior'

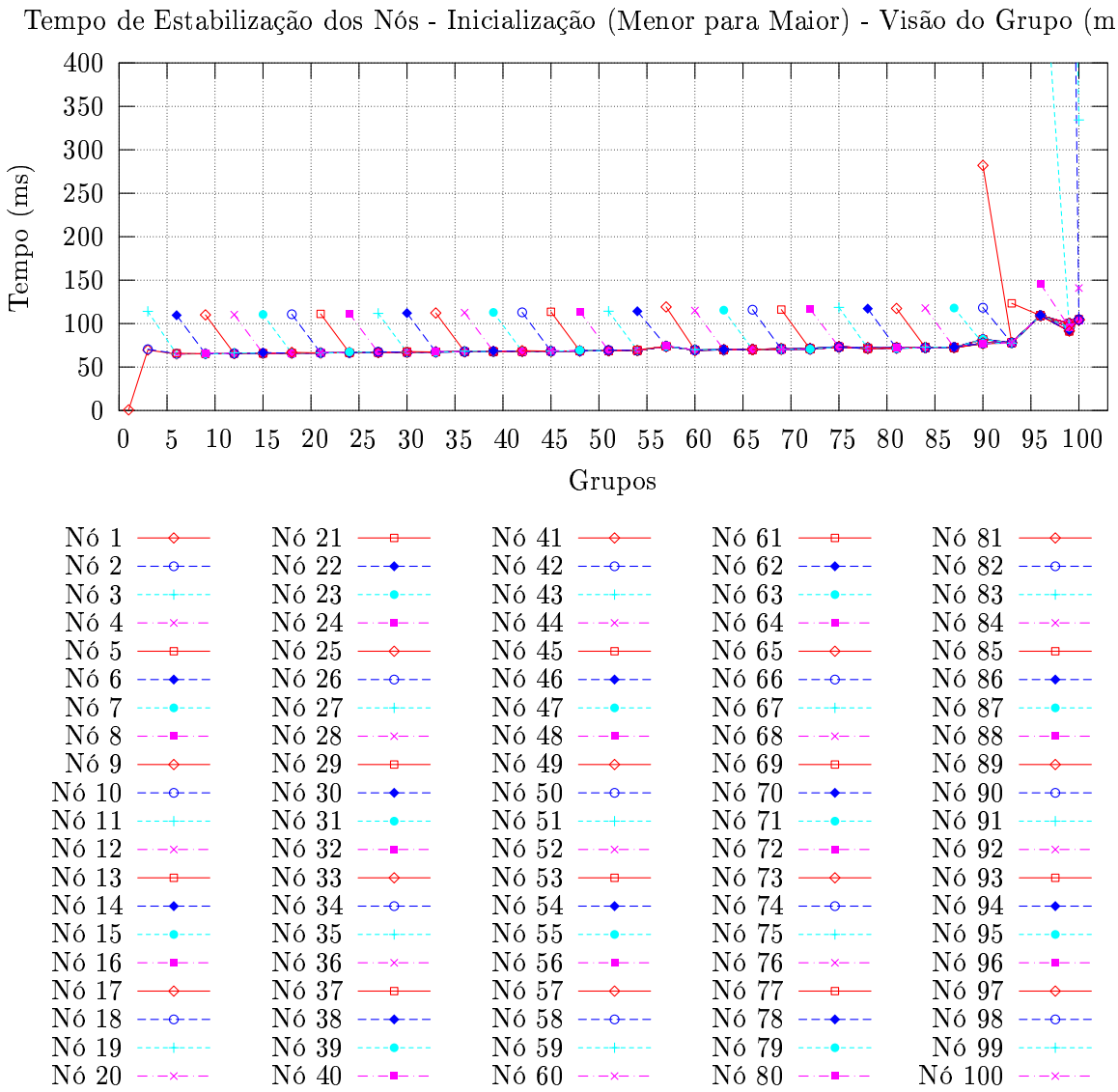

Figura 6.13: Tempo de Estabilização - Membros - Inicialização dos nós do 'menor' para o 'maior'

Grupos	Nós						
	15	**30**	**45**	**60**	**75**	**90**	**100**
0	-	-	-	-	-	-	-
15	8359.3 ± 17389.9	-	-	-	-	-	-
20	12558.3 ± 20042.3	-	-	-	-	-	-
25	12484.6 ± 19913.5	-	-	-	-	-	-
30	16432.8 ± 21071.9	16432.9 ± 21071.8	-	-	-	-	-
35	112.4 ± 0.1	112.4 ± 0.1	-	-	-	-	-
40	8265.1 ± 17186.5	8265.1 ± 17186.5	-	-	-	-	-
45	16221.7 ± 20797.1	16221.7 ± 20797.0	16221.8 ± 20796.8	-	-	-	-
50	20150.1 ± 21133.2	20150.1 ± 21133.2	20150.1 ± 21133.2	-	-	-	-
55	11970.7 ± 19091.3	11970.7 ± 19091.3	11970.7 ± 19091.3	-	-	-	-
60	16139.2 ± 20692.9	16139.2 ± 20693.0	16139.2 ± 20693.0	16139.5 ± 20692.8	-	-	-
65	23469.2 ± 20127.0	23469.2 ± 20127.0	23469.2 ± 20127.0	23469.2 ± 20127.0	-	-	-
70	27768.1 ± 19117.5	27768.1 ± 19117.5	27768.1 ± 19117.5	27768.1 ± 19117.5	-	-	-
75	15930.7 ± 20421.8	15935.1 ± 20427.5	15935.1 ± 20427.5	15935.1 ± 20427.5	15935.6 ± 20427.1	-	-
80	11923.3 ± 19033.6	11923.3 ± 19033.7	11923.3 ± 19033.6	11923.3 ± 19033.7	11923.4 ± 19033.6	-	-
85	15859.2 ± 20346.6	15859.2 ± 20346.6	15859.2 ± 20346.6	15859.2 ± 20346.6	15859.2 ± 20346.5	-	-
90	16365.7 ± 20139.0	16294.6 ± 20194.3	16266.5 ± 20214.2	16233.2 ± 20238.0	16186.5 ± 20272.4	15943.0 ± 20468.0	-
95	8716.0 ± 17123.7	8694.7 ± 17130.5	8665.9 ± 17139.8	8640.1 ± 17148.7	8567.3 ± 17175.8	8330.9 ± 17284.8	-
100	18971.7 ± 18024.4	18895.7 ± 18091.6	18813.5 ± 18165.5	18717.6 ± 18253.5	18640.4 ± 18325.6	18559.8 ± 18402.2	18043.0 ± 18921.9

Tabela 6.6: Tempo de Estabilização - GID - Inicialização dos nós do 'menor' para o 'maior'

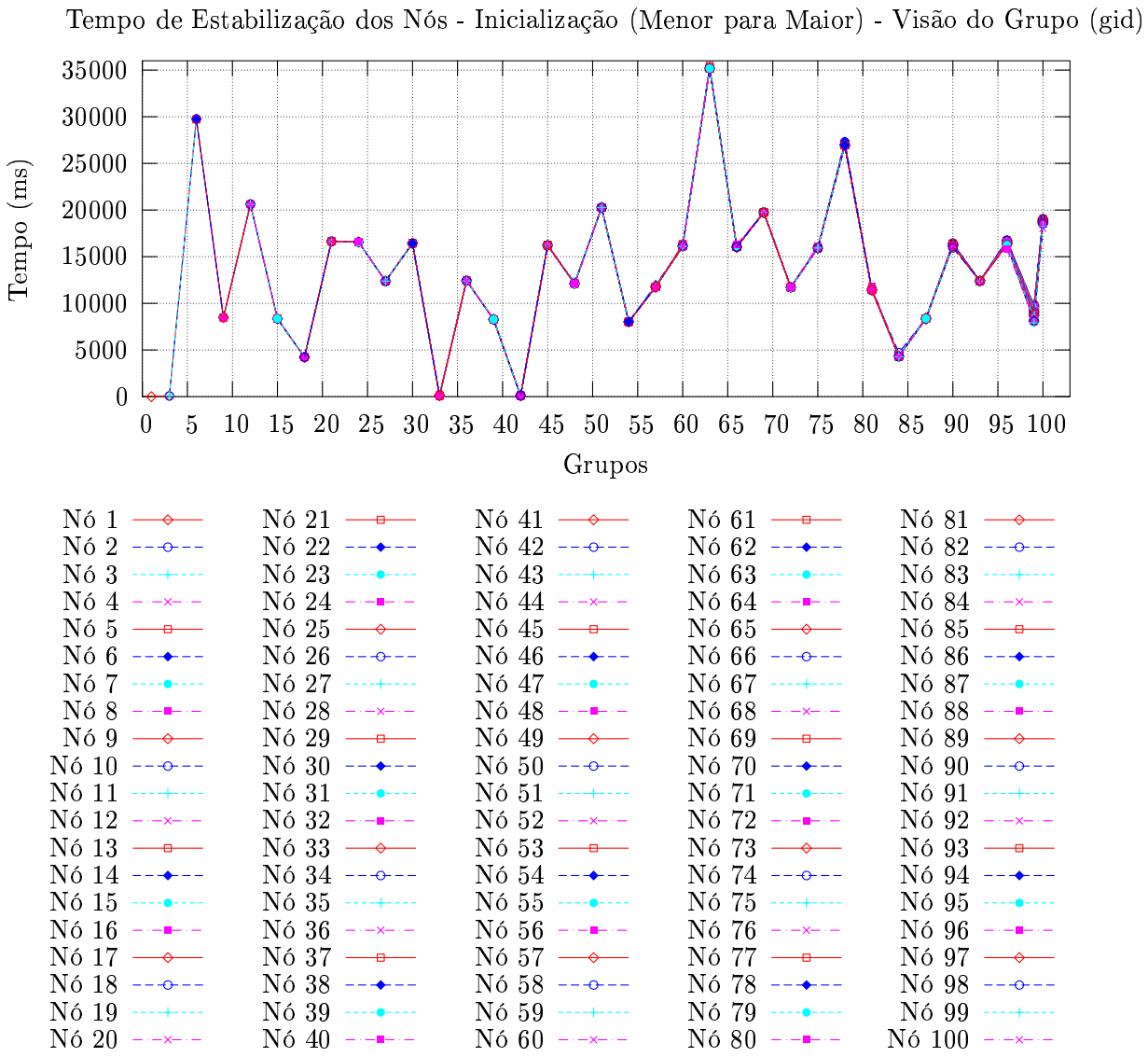

Figura 6.14: Tempo de Estabilização - GID - Inicialização dos nós do 'menor' para o 'maior'

Grupos	Nós						
	15	**30**	**45**	**60**	**75**	**90**	**100**
0	-	-	-	-	-	-	-
5	-	-	-	-	-	-	-
10	-	-	-	-	-	-	-
15	110.6 / 41372.0	-	-	-	-	-	-
20	111.0 / 41742.4	-	-	-	-	-	-
25	111.6 / 41407.4	-	-	-	-	-	-
30	112.1 / 41327.0	112.2 / 41326.9	-	-	-	-	-
35	112.4 / 112.5	112.4 / 112.6	-	-	-	-	-
40	112.9 / 40931.9	112.9 / 40931.9	-	-	-	-	-
45	113.4 / 40693.0	113.4 / 40692.7	113.7 / 40692.6	-	-	-	-
50	19516.6 / 41738.1	19516.6 / 41738.2	19516.7 / 41738.2	-	-	-	-
55	114.3 / 40034.7	114.4 / 40034.7	114.3 / 40034.8	-	-	-	-
60	114.7 / 40994.7	114.7 / 40994.7	114.7 / 40994.6	115.2 / 40994.7	-	-	-
65	37709.1 / 40614.5	37709.1 / 40614.6	37709.1 / 40614.5	37709.1 / 40614.5	-	-	-
70	38681.4 / 42141.2	38681.4 / 42141.2	38681.4 / 42141.2	38681.4 / 42141.2	-	-	-
75	125.6 / 41001.9	125.6 / 41001.8	125.6 / 41001.9	125.6 / 41001.8	126.4 / 41001.9	-	-
80	116.5 / 41753.3	116.5 / 41753.1	116.7 / 41753.3	116.5 / 41753.3	116.6 / 41753.3	-	-
85	116.9 / 41636.6	117.0 / 41636.5	116.9 / 41636.8	117.0 / 41636.6	117.0 / 41636.5	-	-
90	2234.2 / 41774.2	1879.0 / 41774.4	1738.1 / 41774.2	1571.5 / 41774.4	1337.8 / 41774.3	118.5 / 41774.4	-
95	117.9 / 41737.6	117.9 / 41738.1	117.9 / 41737.6	118.0 / 41737.8	117.9 / 41737.8	117.8 / 41737.8	-
100	19833.9 / 39406.4	19617.2 / 39406.4	19454.8 / 39406.4	19179.3 / 39406.4	19004.5 / 39406.5	18803.0 / 39406.5	17516.5 / 39406.6

Tabela 6.7: Tempo de Estabilização - GID - Inicialização - Mediana e Máximo

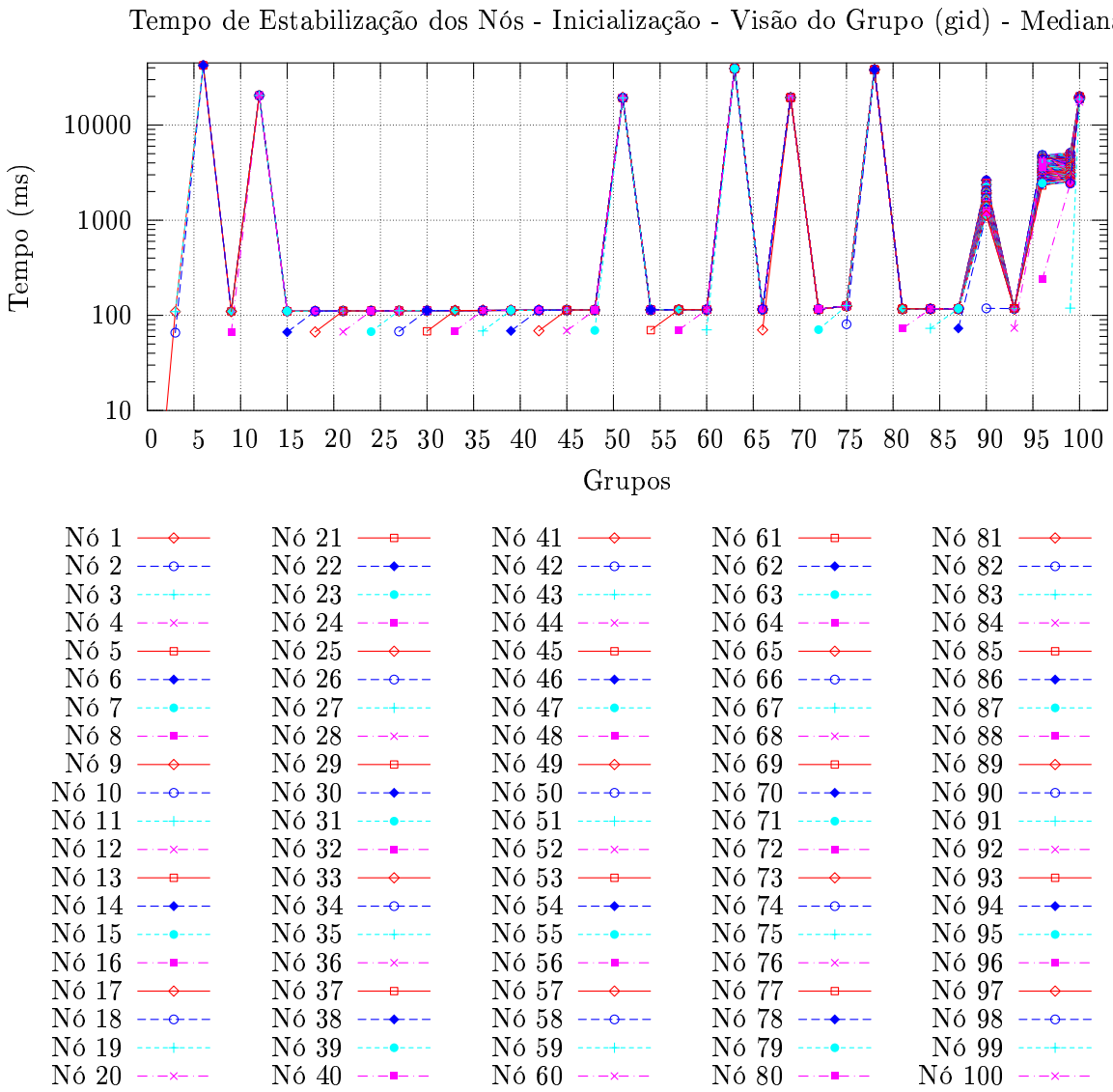

Figura 6.15: Tempo de Estabilização - GID - Inicialização do 'menor' para o 'maior' - Mediana

Grupos	Nós						
	15	**30**	**45**	**60**	**75**	**90**	**100**
0	-	-	-	-	-	-	-
15	1.5 ± 0.5	-	-	-	-	-	-
20	2.3 ± 0.5	-	-	-	-	-	-
25	2.3 ± 0.5	-	-	-	-	-	-
30	2.6 ± 0.5	1.6 ± 0.5	-	-	-	-	-
35	2.2 ± 0.4	2.1 ± 0.3	-	-	-	-	-
40	2.3 ± 0.5	2.3 ± 0.5	-	-	-	-	-
45	2.4 ± 0.5	2.4 ± 0.5	1.4 ± 0.5	-	-	-	-
50	2.6 ± 0.5	2.6 ± 0.5	2.6 ± 0.5	-	-	-	-
55	2.7 ± 0.5	2.7 ± 0.5	2.7 ± 0.5	-	-	-	-
60	2.5 ± 0.5	2.5 ± 0.5	2.5 ± 0.5	1.7 ± 0.7	-	-	-
65	3.0 ± 0.5	2.9 ± 0.3	2.9 ± 0.3	2.9 ± 0.3	-	-	-
70	2.8 ± 0.6	2.8 ± 0.6	2.8 ± 0.6	2.8 ± 0.6	-	-	-
75	2.6 ± 0.7	2.6 ± 0.7	2.6 ± 0.7	2.6 ± 0.7	1.7 ± 0.8	-	-
80	2.4 ± 0.7	2.3 ± 0.5	2.3 ± 0.5	2.3 ± 0.5	2.3 ± 0.5	-	-
85	2.8 ± 0.9	2.5 ± 0.5	2.5 ± 0.5	2.5 ± 0.5	2.5 ± 0.5	-	-
90	2.4 ± 0.7	2.4 ± 0.7	2.3 ± 0.5	2.3 ± 0.5	2.3 ± 0.5	1.4 ± 0.5	-
95	2.2 ± 0.4	2.2 ± 0.4	2.2 ± 0.4	2.2 ± 0.4	2.2 ± 0.4	2.2 ± 0.4	-
100	2.5 ± 0.5	2.5 ± 0.5	2.4 ± 0.5	2.4 ± 0.5	2.4 ± 0.5	2.4 ± 0.5	1.5 ± 0.5

Tabela 6.8: Tempo de Estabilização - GID - Quantidade de Alterações

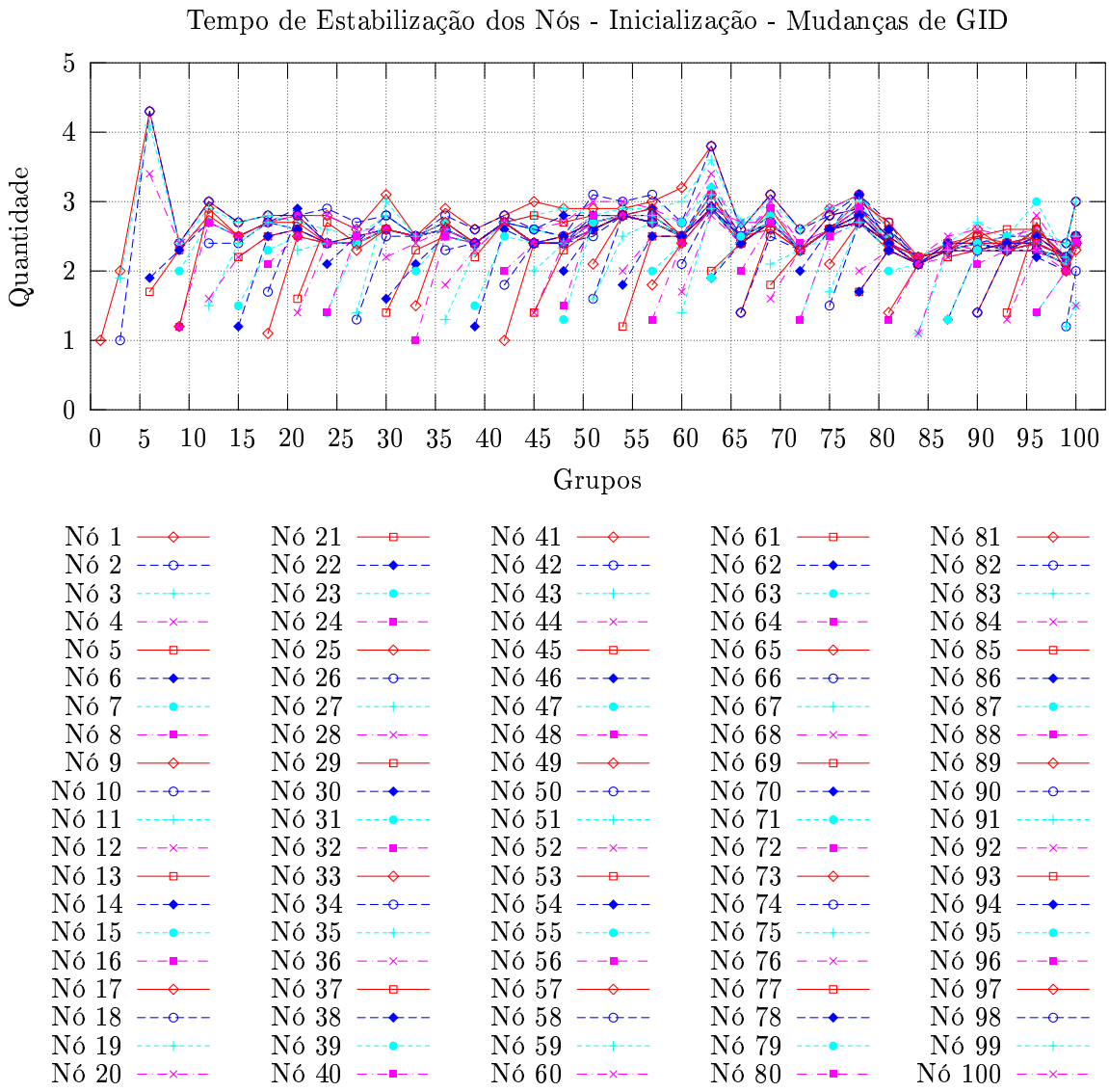

Figura 6.16: Tempo de Estabilização - GID - Quantidade de Alterações

Grupos	Nós						
	15	**30**	**45**	**60**	**75**	**90**	**100**
0	-	-	-	-	-	-	-
15	-	-	-	-	-	65.9 ± 0.2	65.8 ± 0.2
20	-	-	-	-	-	66.2 ± 0.1	66.2 ± 0.1
25	-	-	-	-	-	66.6 ± 0.1	66.6 ± 0.2
30	-	-	-	-	67.3 ± 0.8	67.4 ± 0.9	67.3 ± 0.9
35	-	-	-	-	67.4 ± 0.1	67.4 ± 0.1	67.5 ± 0.1
40	-	-	-	-	67.9 ± 0.2	68.1 ± 0.1	68.0 ± 0.3
45	-	-	-	68.4 ± 0.2	68.3 ± 0.1	68.3 ± 0.1	68.3 ± 0.2
50	-	-	-	68.7 ± 0.2	68.8 ± 0.3	68.7 ± 0.2	68.8 ± 0.2
55	-	-	-	69.1 ± 0.2	69.2 ± 0.2	69.2 ± 0.2	69.2 ± 0.2
60	-	-	69.6 ± 0.1	69.4 ± 0.1	69.5 ± 0.1	69.5 ± 0.1	69.5 ± 0.1
65	-	-	69.9 ± 0.1	69.9 ± 0.1	69.9 ± 0.1	69.9 ± 0.1	70.0 ± 0.2
70	-	-	92.8 ± 63.1	92.8 ± 63.1	92.9 ± 63.1	92.9 ± 63.1	92.8 ± 63.1
75	-	101.8 ± 40.9	486.3 ± 831.0	1034.1 ± 1240.7	1059.4 ± 1272.2	1032.8 ± 1248.1	1254.5 ± 1501.0
80	-	303.7 ± 655.4	508.3 ± 875.1	528.6 ± 924.6	924.9 ± 1344.1	1630.0 ± 1636.3	1855.6 ± 1571.6
85	-	347.6 ± 759.5	643.5 ± 1134.5	1455.8 ± 1237.5	1572.2 ± 1596.2	2189.8 ± 1804.3	2968.5 ± 1527.1
90	538.9 ± 913.5	587.5 ± 1014.8	643.5 ± 1133.2	704.7 ± 1262.0	1086.4 ± 1581.4	1527.6 ± 1839.9	3181.3 ± 1653.9
95	345.8 ± 738.5	113.0 ± 14.0	112.8 ± 14.4	112.8 ± 14.3	1212.8 ± 1774.7	2092.4 ± 2093.5	3396.6 ± 1741.6
100	888.1 ± 1249.6	712.2 ± 1263.6	1110.3 ± 1608.3	1582.5 ± 1902.9	2874.0 ± 1915.5	3550.9 ± 1827.8	4114.6 ± 1424.3

Tabela 6.9: Tempo de Estabilização - Membros - Inicialização dos nós do 'maior' para o 'menor'

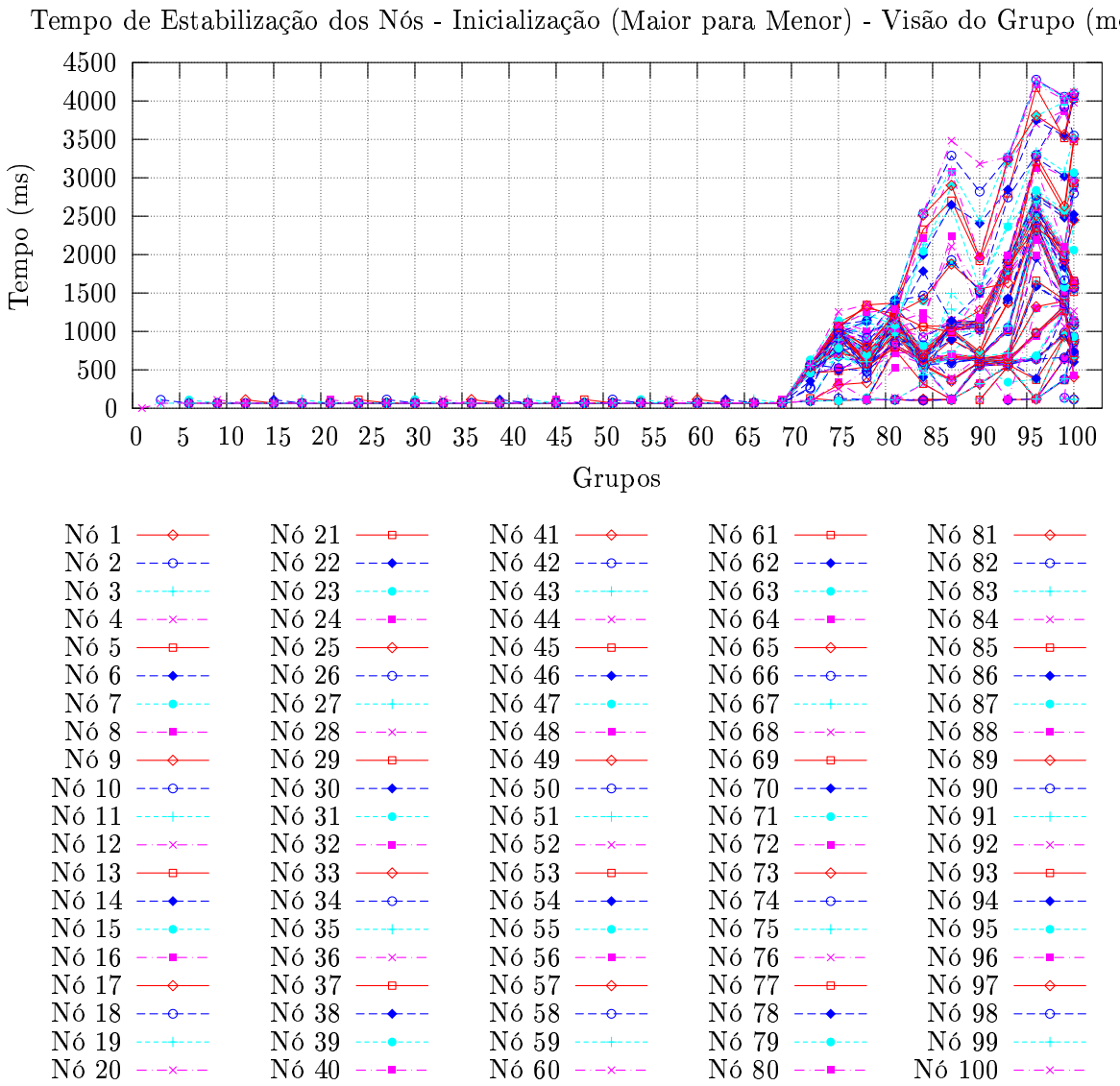

Figura 6.17: Tempo de Estabilização - Membros - Inicialização dos nós do 'maior' para o 'menor'

Grupos	Nós						
	15	**30**	**45**	**60**	**75**	**90**	**100**
0	-	-	-	-	-	-	-
15	-	-	-	-	-	40117.6 ± 89.2	40117.6 ± 89.2
20	-	-	-	-	-	39917.5 ± 5.0	39917.5 ± 5.0
25	-	-	-	-	-	40628.6 ± 4.0	40628.6 ± 4.0
30	-	-	-	-	40633.1 ± 4.1	40633.1 ± 4.1	40633.2 ± 4.1
35	-	-	-	-	40632.4 ± 4.2	40632.4 ± 4.2	40632.4 ± 4.2
40	-	-	-	-	40386.2 ± 645.7	40386.2 ± 645.7	40386.2 ± 645.7
45	-	-	-	40389.9 ± 767.9	40389.9 ± 767.9	40389.9 ± 767.9	40390.0 ± 767.9
50	-	-	-	60955.8 ± 22027.8	60955.9 ± 22027.8	60955.8 ± 22027.8	60955.9 ± 22027.8
55	-	-	-	77775.3 ± 13051.2	77775.3 ± 13051.2	77775.3 ± 13051.2	77775.4 ± 13051.2
60	-	-	69567.8 ± 20077.2	69567.8 ± 20077.2	69567.9 ± 20077.3	69567.8 ± 20077.3	69567.8 ± 20077.3
65	-	-	50370.0 ± 21927.2	50370.0 ± 21927.1	50369.9 ± 21927.1	50370.0 ± 21927.1	50370.0 ± 21927.2
70	-	-	46181.5 ± 19013.5	46181.5 ± 19013.5	46181.5 ± 19013.5	46181.6 ± 19013.5	46181.6 ± 19013.5
75	-	46676.6 ± 18996.4	46676.6 ± 18996.4	46676.6 ± 18996.4	46676.6 ± 18996.4	46676.7 ± 18996.4	46676.6 ± 18996.4
80	-	40094.1 ± 2762.9	40094.1 ± 2762.9	40094.1 ± 2762.9	40094.1 ± 2762.9	40094.0 ± 2762.9	40094.2 ± 2762.9
85	-	41590.9 ± 3973.7	41590.9 ± 3973.6	41590.8 ± 3973.6	41590.9 ± 3973.6	41590.9 ± 3973.6	41590.9 ± 3973.6
90	54369.9 ± 20059.0	54370.0 ± 20059.0	54369.9 ± 20058.9	54370.0 ± 20058.9	54370.0 ± 20059.0	54369.9 ± 20058.9	54370.0 ± 20058.9
95	42009.7 ± 1804.3	42009.7 ± 1804.3	42009.7 ± 1804.3	42009.7 ± 1804.3	42009.7 ± 1804.3	42009.6 ± 1804.3	42009.7 ± 1804.2
100	41647.7 ± 1405.1	41647.7 ± 1405.0	41647.7 ± 1405.0	41647.8 ± 1405.1	41647.8 ± 1405.1	41647.7 ± 1405.2	41647.7 ± 1405.2

Tabela 6.10: Tempo de Estabilização - GID - Inicialização dos nós do 'maior' para o 'menor'

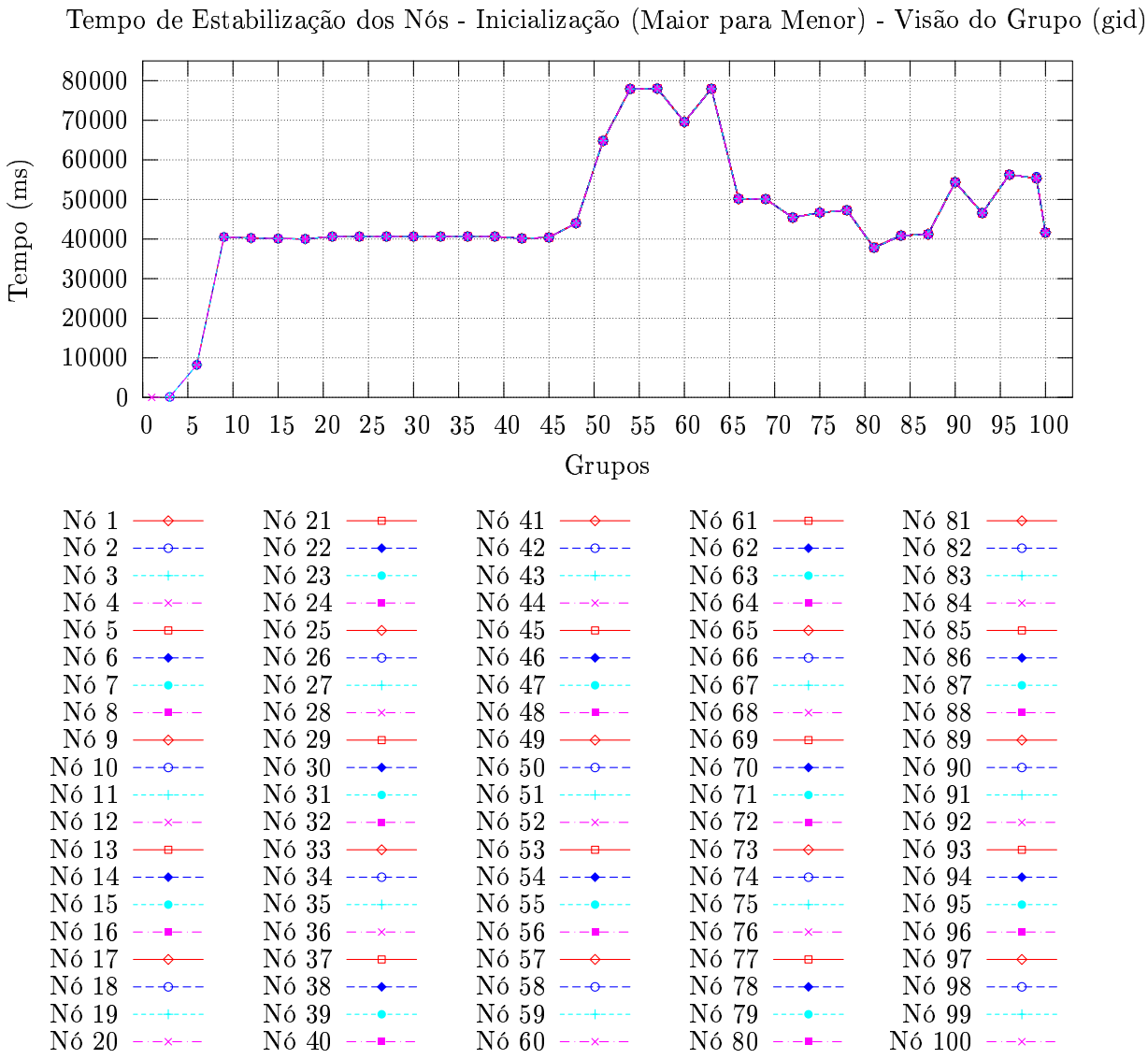

Figura 6.18: Tempo de Estabilização - GID - Inicialização dos nós do 'maior' para o 'menor'

Grupos	Nós						
	15	**30**	**45**	**60**	**75**	**90**	**100**
0	-	-	-	-	-	-	-
15	-	-	-	-	-	7.0 ± 0.0	2.0 ± 0.0
20	-	-	-	-	-	6.0 ± 0.0	2.0 ± 0.0
25	-	-	-	-	-	6.0 ± 0.0	2.0 ± 0.0
30	-	-	-	-	6.0 ± 0.0	6.0 ± 0.0	2.0 ± 0.0
35	-	-	-	-	6.0 ± 0.0	6.0 ± 0.0	2.0 ± 0.0
40	-	-	-	-	6.0 ± 0.0	6.0 ± 0.0	2.0 ± 0.0
45	-	-	-	6.0 ± 0.0	6.0 ± 0.0	6.0 ± 0.0	2.0 ± 0.0
50	-	-	-	8.5 ± 2.6	8.5 ± 2.6	8.5 ± 2.6	2.5 ± 0.5
55	-	-	-	8.6 ± 1.3	8.6 ± 1.3	8.6 ± 1.3	3.0 ± 0.5
60	-	-	7.8 ± 1.9	7.8 ± 1.9	7.8 ± 1.9	7.7 ± 1.9	2.9 ± 0.7
65	-	-	6.2 ± 1.9	6.2 ± 1.9	6.2 ± 1.9	6.2 ± 1.9	2.3 ± 0.5
70	-	-	5.8 ± 1.7	5.8 ± 1.7	5.8 ± 1.7	5.8 ± 1.7	2.4 ± 0.8
75	-	6.0 ± 1.9	5.2 ± 2.2	4.9 ± 2.5	4.8 ± 2.6	4.8 ± 2.6	2.4 ± 0.8
80	-	4.3 ± 0.9	4.2 ± 1.0	3.8 ± 1.1	3.6 ± 1.3	3.4 ± 1.4	2.0 ± 0.0
85	-	4.9 ± 0.9	4.3 ± 0.8	3.5 ± 1.2	3.1 ± 1.2	2.9 ± 1.3	2.0 ± 0.0
90	5.1 ± 1.5	4.7 ± 1.1	4.8 ± 1.0	4.7 ± 1.1	4.3 ± 1.2	3.8 ± 1.2	2.6 ± 1.0
95	4.5 ± 0.8	4.6 ± 1.0	4.3 ± 0.8	4.0 ± 0.9	3.5 ± 1.3	3.0 ± 1.2	2.0 ± 0.0
100	4.2 ± 1.0	3.9 ± 1.1	3.3 ± 0.9	3.0 ± 0.9	2.6 ± 1.0	2.4 ± 1.0	2.0 ± 0.0

Tabela 6.11: Tempo de Estabilização - GID - Quantidade de Alterações

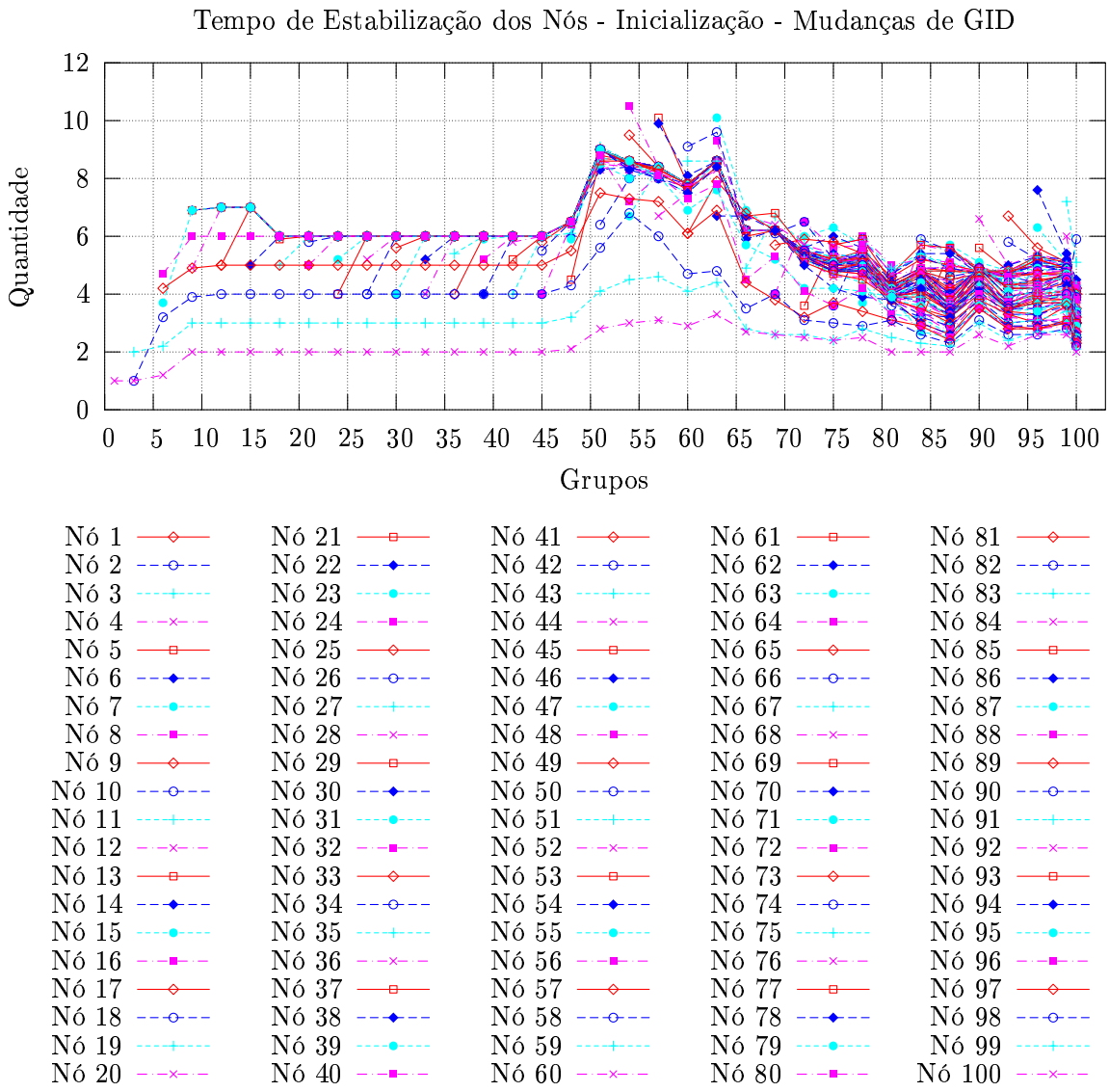

Figura 6.19: Tempo de Estabilização - GID - Quantidade de Alterações

Grupos	Nós						
	15	**30**	**45**	**60**	**75**	**90**	**100**
0	-	-	-	-	-	-	-
15	-	38596.6 ± 32.6	38596.7 ± 32.7	38596.7 ± 32.6	38596.7 ± 32.6	38596.8 ± 32.6	38596.8 ± 32.6
20	-	38756.3 ± 27.7	38756.2 ± 27.9	38756.2 ± 27.8	38756.3 ± 27.9	38756.4 ± 27.9	38756.4 ± 27.8
25	-	38910.4 ± 22.4	38910.4 ± 22.4	38910.4 ± 22.4	38910.5 ± 22.3	38910.5 ± 22.4	38910.6 ± 22.4
30	-	-	39073.5 ± 18.6	39073.5 ± 18.6	39073.6 ± 18.6	39073.7 ± 18.5	39073.6 ± 18.6
35	-	-	39211.4 ± 11.2	39211.4 ± 11.2	39211.4 ± 11.2	39211.5 ± 11.2	39211.5 ± 11.2
40	-	-	39336.8 ± 19.1	39336.8 ± 19.1	39336.8 ± 19.1	39337.0 ± 19.1	39336.9 ± 19.1
45	-	-	-	39467.0 ± 9.3	39467.0 ± 9.3	39467.1 ± 9.3	39467.5 ± 9.2
50	-	-	-	39572.9 ± 13.4	39572.9 ± 13.5	39573.2 ± 13.5	39573.1 ± 13.4
55	-	-	-	39671.9 ± 45.7	39672.0 ± 45.7	39672.1 ± 45.7	39672.1 ± 45.7
60	-	-	-	-	39790.7 ± 13.6	39790.8 ± 13.6	39790.8 ± 13.6
65	-	-	-	-	39893.8 ± 7.5	39893.9 ± 7.5	39893.9 ± 7.5
70	-	-	-	-	39985.0 ± 10.0	39985.1 ± 10.0	39985.1 ± 10.0
75	-	-	-	-	-	40096.2 ± 71.4	40096.3 ± 71.4
80	-	-	-	-	-	40126.2 ± 77.4	40126.1 ± 77.6
85	-	-	-	-	-	40176.4 ± 79.1	40176.5 ± 79.1
90	-	-	-	-	-	-	40133.7 ± 177.5
95	-	-	-	-	-	-	40115.9 ± 135.3
100	-	-	-	-	-	-	-

Tabela 6.12: Tempo de Estabilização - Membros - Finalização dos nós do 'menor' para o 'maior'

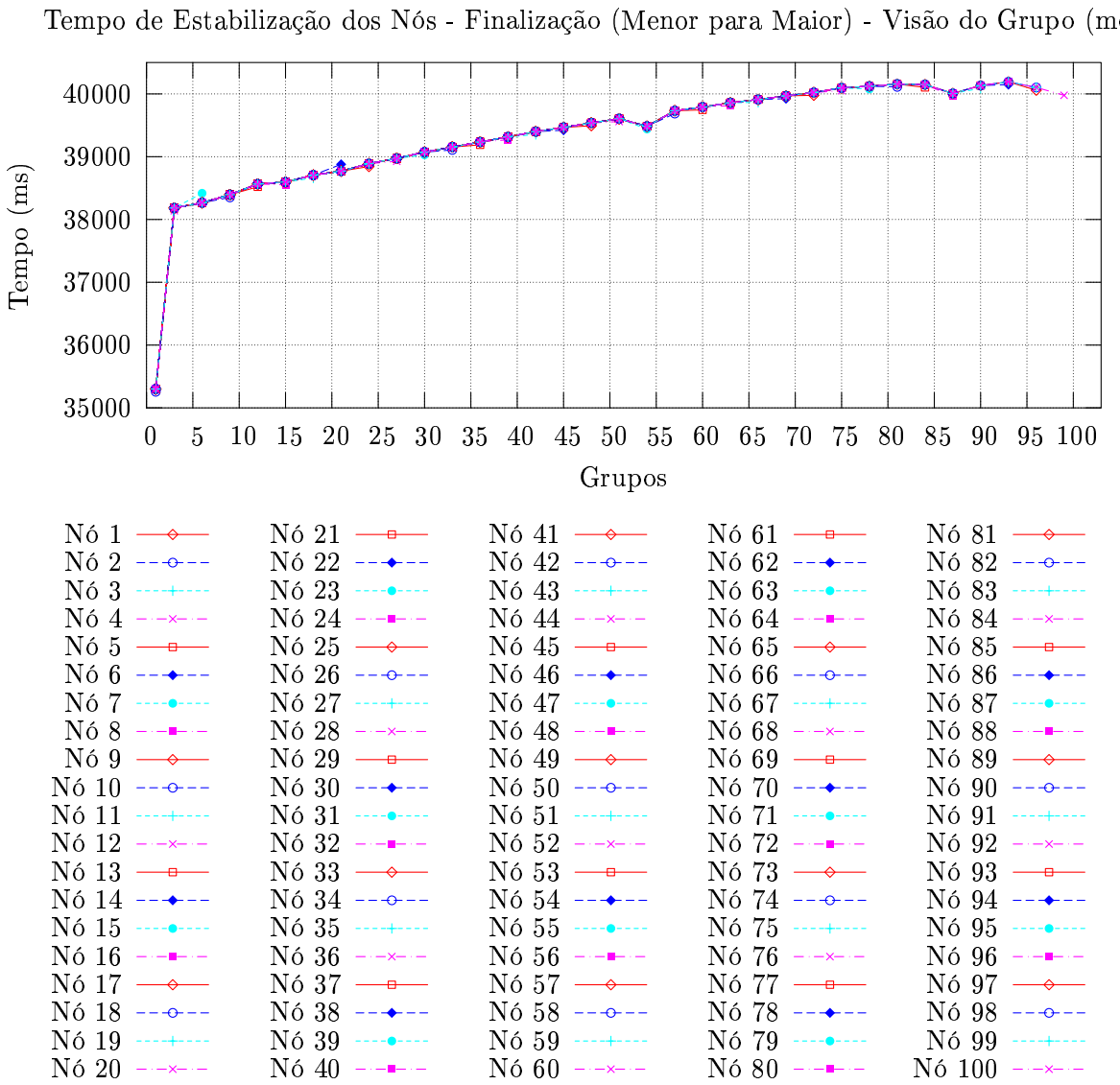

Figura 6.20: Tempo de Estabilização - Membros - Finalização dos nós do 'menor' para o 'maior'

Grupos	Nós						
	15	**30**	**45**	**60**	**75**	**90**	**100**
0	-	-	-	-	-	-	-
15	-	38596.6 ± 32.6	38596.7 ± 32.7	38596.7 ± 32.6	38596.7 ± 32.6	38596.8 ± 32.6	38596.8 ± 32.6
20	-	38756.3 ± 27.7	38756.2 ± 27.9	38756.2 ± 27.8	38756.3 ± 27.9	38756.4 ± 27.9	38756.4 ± 27.8
25	-	38910.4 ± 22.4	38910.4 ± 22.4	38910.4 ± 22.4	38910.5 ± 22.3	38910.5 ± 22.4	38910.6 ± 22.4
30	-	-	39073.5 ± 18.6	39073.5 ± 18.6	39073.6 ± 18.6	39073.7 ± 18.5	39073.6 ± 18.6
35	-	-	39211.4 ± 11.2	39211.4 ± 11.2	39211.4 ± 11.2	39211.5 ± 11.2	39211.5 ± 11.2
40	-	-	39336.8 ± 19.1	39336.8 ± 19.1	39336.8 ± 19.1	39337.0 ± 19.1	39336.9 ± 19.1
45	-	-	-	39467.0 ± 9.3	39467.0 ± 9.3	39467.1 ± 9.3	39467.5 ± 9.2
50	-	-	-	39572.9 ± 13.4	39572.9 ± 13.5	39573.2 ± 13.5	39573.1 ± 13.4
55	-	-	-	39671.9 ± 45.7	39672.0 ± 45.7	39672.1 ± 45.7	39672.1 ± 45.7
60	-	-	-	-	39790.7 ± 13.6	39790.8 ± 13.6	39790.8 ± 13.6
65	-	-	-	-	39893.8 ± 7.5	39893.9 ± 7.5	39893.9 ± 7.5
70	-	-	-	-	39985.0 ± 10.0	39985.1 ± 10.0	39985.1 ± 10.0
75	-	-	-	-	-	40096.2 ± 71.4	40096.3 ± 71.4
80	-	-	-	-	-	40126.2 ± 77.4	40126.1 ± 77.6
85	-	-	-	-	-	40176.4 ± 79.1	40176.5 ± 79.1
90	-	-	-	-	-	-	40133.7 ± 177.5
95	-	-	-	-	-	-	40115.9 ± 135.3
100	-	-	-	-	-	-	-

Tabela 6.13: Tempo de Estabilização - GID - Finalização dos nós do 'menor' para o 'maior'

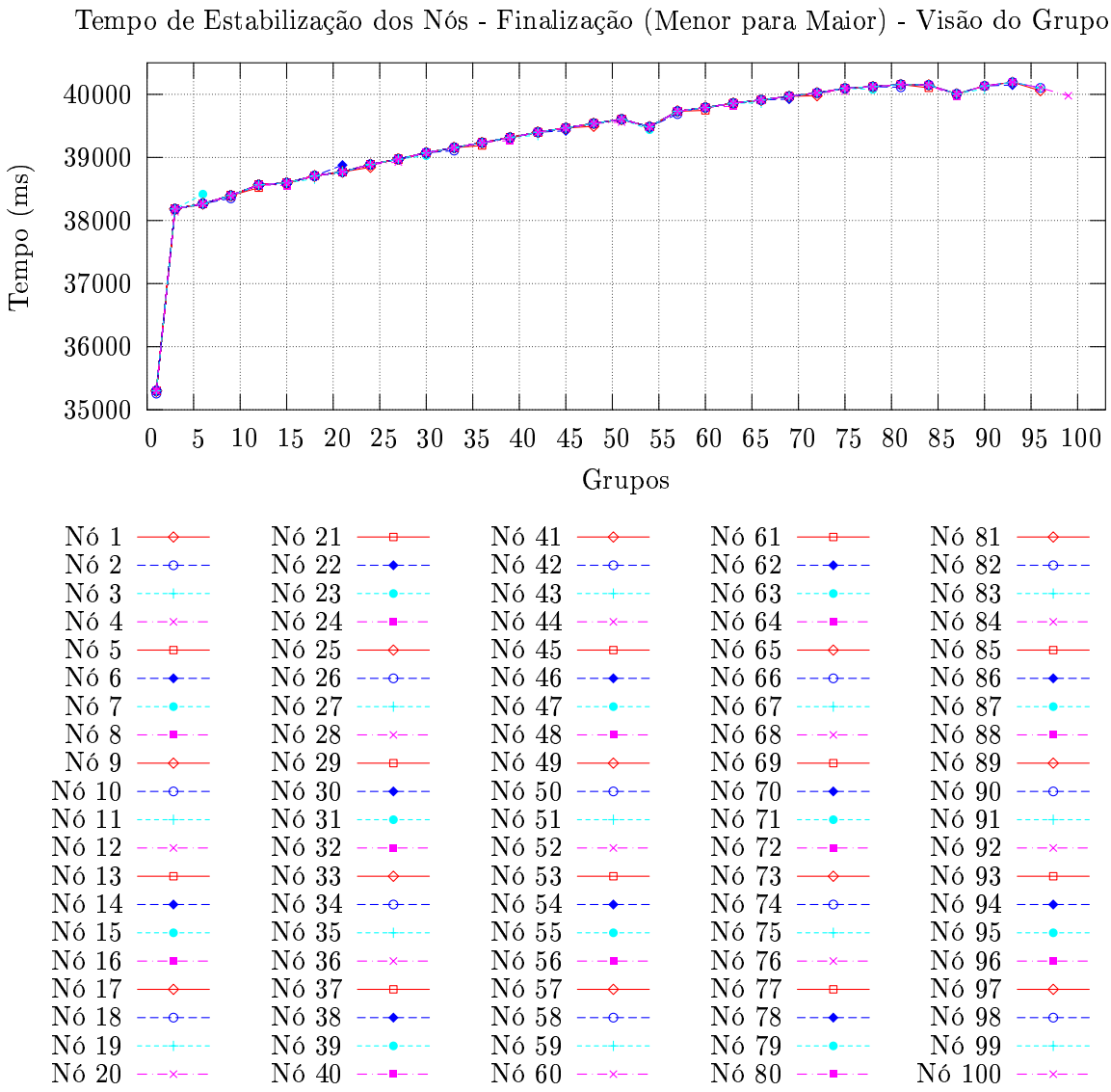

Figura 6.21: Tempo de Estabilização - GID - Finalização dos nós do 'menor' para o 'maior'

Grupos	Nós						
	15	**30**	**45**	**60**	**75**	**90**	**100**
0	-	-	-	-	-	-	-
15	72587.3 ± 10.8	72587.3 ± 10.7	72587.3 ± 10.8	72587.3 ± 10.8	72587.2 ± 10.8	-	-
20	70713.7 ± 70.9	70713.7 ± 70.9	70713.8 ± 70.9	70713.8 ± 70.9	70713.7 ± 70.9	-	-
25	68487.7 ± 191.8	68487.7 ± 191.8	68487.7 ± 191.8	68487.7 ± 191.8	68487.7 ± 191.8	-	-
30	66688.6 ± 110.4	66688.5 ± 110.4	66688.6 ± 110.4	66688.6 ± 110.6	-	-	-
35	64804.7 ± 198.9	64804.6 ± 198.8	64804.6 ± 198.8	64804.6 ± 198.8	-	-	-
40	62706.4 ± 578.8	62706.5 ± 578.8	62706.4 ± 578.8	62706.4 ± 578.8	-	-	-
45	60961.8 ± 212.7	60961.8 ± 212.7	60961.8 ± 212.6	-	-	-	-
50	59261.2 ± 60.0	59261.2 ± 60.0	59261.2 ± 60.0	-	-	-	-
55	57316.4 ± 150.4	57316.4 ± 150.5	57316.4 ± 150.5	-	-	-	-
60	55550.0 ± 135.6	55550.1 ± 135.5	-	-	-	-	-
65	52954.9 ± 483.7	52954.9 ± 483.7	-	-	-	-	-
70	50982.9 ± 465.1	50983.0 ± 465.1	-	-	-	-	-
75	49332.4 ± 529.3	-	-	-	-	-	-
80	47314.1 ± 576.9	-	-	-	-	-	-
85	45537.8 ± 561.9	-	-	-	-	-	-
90	-	-	-	-	-	-	-
95	-	-	-	-	-	-	-
100	-	-	-	-	-	-	-

Tabela 6.14: Tempo de Estabilização - Membros - Finalização dos nós do 'maior' para o 'menor'

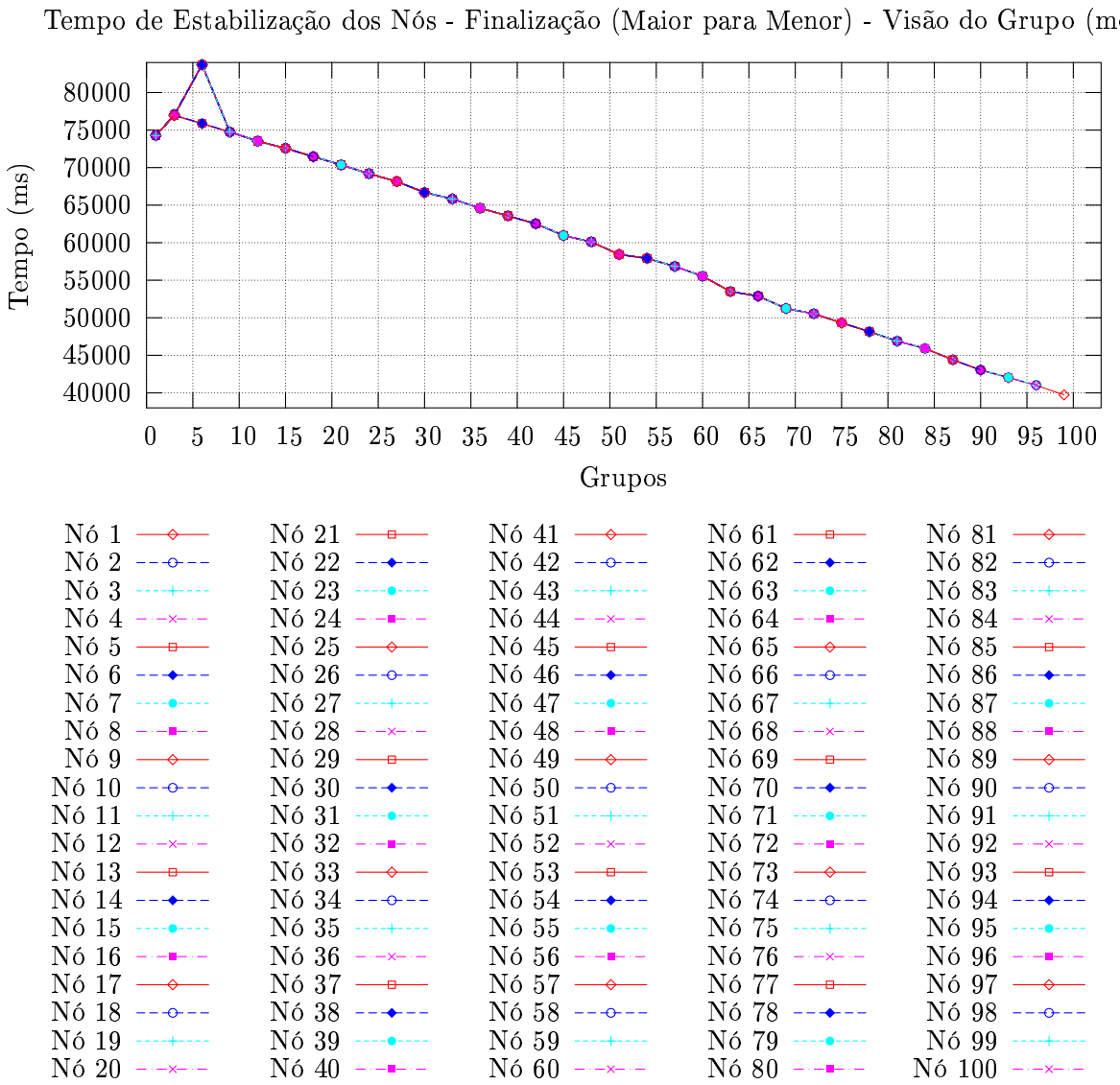

Figura 6.22: Tempo de Estabilização - Membros - Finalização dos nós do 'maior' para o 'menor'

Grupos	Nós						
	15	**30**	**45**	**60**	**75**	**90**	**100**
0	-	-	-	-	-	-	-
15	72587.3 ± 10.8	72587.3 ± 10.7	72587.3 ± 10.8	72587.3 ± 10.8	72587.2 ± 10.8	-	-
20	70713.7 ± 70.9	70713.7 ± 70.9	70713.8 ± 70.9	70713.8 ± 70.9	70713.7 ± 70.9	-	-
25	68487.7 ± 191.8	68487.7 ± 191.8	68487.7 ± 191.8	68487.7 ± 191.8	68487.7 ± 191.8	-	-
30	66688.6 ± 110.4	66688.5 ± 110.4	66688.6 ± 110.4	66688.6 ± 110.6	-	-	-
35	64804.7 ± 198.9	64804.6 ± 198.8	64804.6 ± 198.8	64804.6 ± 198.8	-	-	-
40	62706.4 ± 578.8	62706.5 ± 578.8	62706.4 ± 578.8	62706.4 ± 578.8	-	-	-
45	60961.8 ± 212.7	60961.8 ± 212.7	60961.8 ± 212.6	-	-	-	-
50	59261.2 ± 60.0	59261.2 ± 60.0	59261.2 ± 60.0	-	-	-	-
55	57316.4 ± 150.4	57316.4 ± 150.5	57316.4 ± 150.5	-	-	-	-
60	55550.0 ± 135.6	55550.1 ± 135.5	-	-	-	-	-
65	52954.9 ± 483.7	52954.9 ± 483.7	-	-	-	-	-
70	50982.9 ± 465.1	50983.0 ± 465.1	-	-	-	-	-
75	49332.4 ± 529.3	-	-	-	-	-	-
80	47314.1 ± 576.9	-	-	-	-	-	-
85	45537.8 ± 561.9	-	-	-	-	-	-
90	-	-	-	-	-	-	-
95	-	-	-	-	-	-	-
100	-	-	-	-	-	-	-

Tabela 6.15: Tempo de Estabilização - GID - Finalização dos nós do 'maior' para o 'menor'

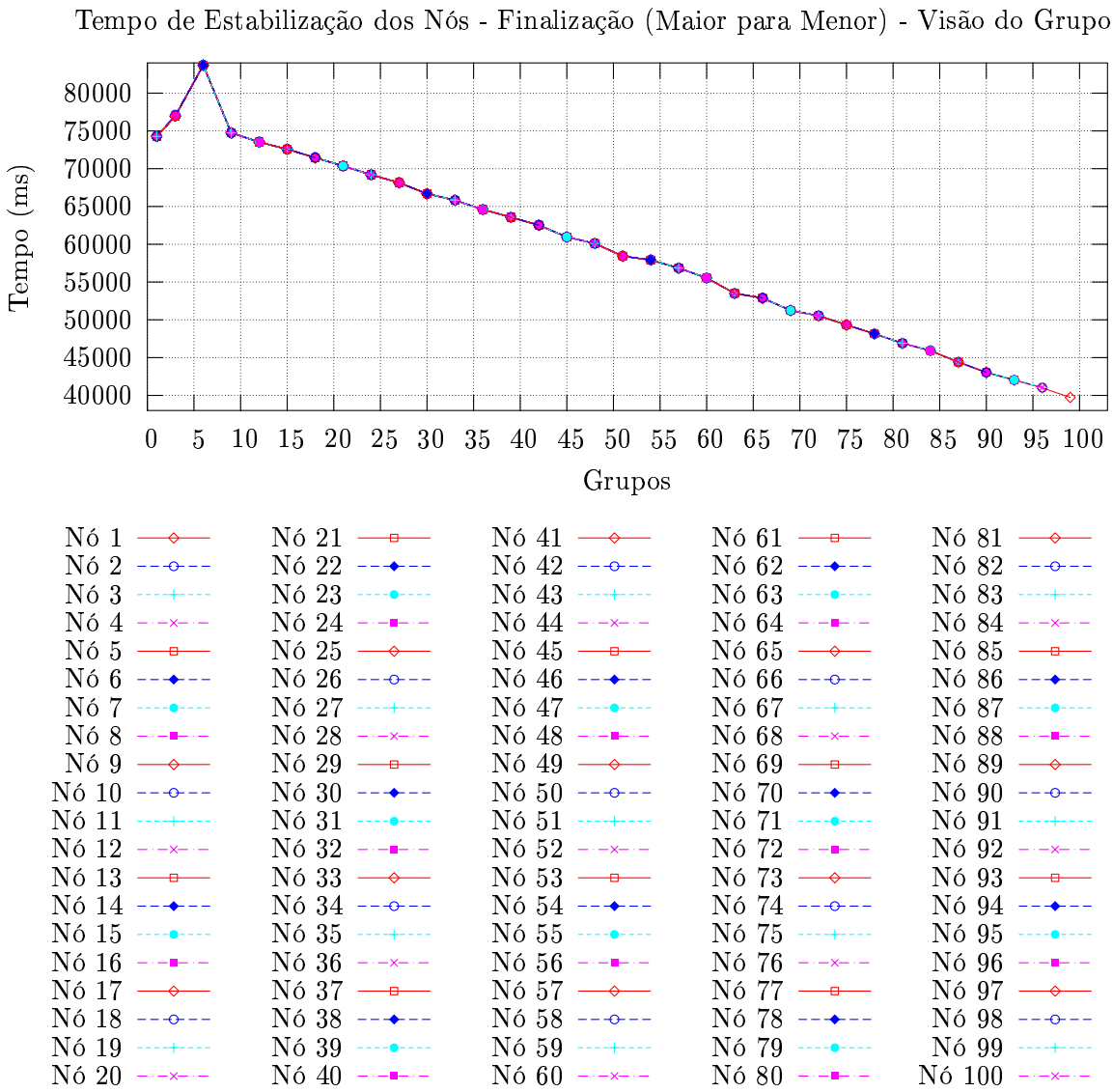

Figura 6.23: Tempo de Estabilização - GID - Finalização dos nós do 'maior' para o 'menor'

Grupos		**Nós**		
		1	...	**100**
0	1 ...100	-	-	-
1_a	1	5979466.011 ± 5911.049	-	-
1_b	100	-	-	5939524.242 ± 2107.436

Tabela 6.16: Tempo de Estabilização - Finalização Múltipla dos Nós

6.3.3 Análise dos Resultados

Após a execução dos experimentos, e posterior análise de seus resultados, algumas conclusões puderam ser tiradas:

1. De modo geral, a implementação exibiu o comportamento esperado. Nenhuma grande novidade foi apresentada, e o protocolo realmente mostrou as qualidades e defeitos antecipados pelos autores.

2. Obter uma medida precisa do tempo de estabilização em um cluster real é uma tarefa praticamente impossível. Sem a ajuda de um simulador central, diversos problemas surgem em um cenário real, como diferenças de relógios e falta de controle do ambiente.

3. Os valores obtidos são muito pequenos, e por isso facilmente influenciadas por outros fatores. Quando observamos o tempo de estabilização da inicialização dos nós, percebemos que normalmente este tempo fica na casa dos 50 a 100 milissegundos, ou seja, 0.05 a 0.10 segundos. Esta ordem de grandeza é tão pequena que é afetada por outros fatores relacionados ao sistema operacional e ao hardware, como acesso a disco e troca de contexto entre processos. Assim, quando tratamos de valores desta magnitude é difícil evitar que o tempo de estabilização não seja afetado por outros fatores.

4. A estabilização após uma finalização é praticamente simultânea em todos os nós. Ainda, o tempo de estabilização do `gid` é igual ao tempo de estabilização do conjunto de membros. Como conseqüência desta última característica, pudemos concluir e observar que o `gid` sofre somente uma alteração até sua estabilização.

   A estabilização é praticamente simultânea em todos os nós porque em um primeiro momento somente um nó percebe esta finalização, criando assim um novo grupo e comunicando esta alteração a todos os outros membros. Ao receber esta mensagem, estes membros simplesmente se unem ao grupo, não gerando mais nenhuma mensagem. Assim, a diferença do tempo de estabilização entre o nó que percebeu a modificação e os que receberam a notificação é o tempo necessário para o envio e processamento desta mensagem de notificação.

5. O tempo de estabilização do `gid` está diretamente relacionado à quantidade de trocas efetuadas neste `gid`. Este comportamento pode ser observado ao compararmos os gráficos 6.18 e 6.19. Assim, podemos esperar um maior tempo de estabilização do `gid` quando ocorrerem mais trocas de `gid`.

6. Mesmo com todo o controle efetuado, algumas medidas ainda foram influenciadas pelo ambiente. Isto é comprovado pelo alto desvio padrão encontrado em algumas medidas, como no tempo de estabilização do `gid` após uma inicialização.

Neste caso específico, analisamos mais a fundo os resultados para entender o que estava acontecendo. Por exemplo, se observarmos o tempo de estabilização do nó 15 no grupo 15 (tabela 6.6), encontraremos um tempo de 8359.3 ± 17389.9. Para este valor final, os seguintes valores parciais foram encontrados: 110.5, 41336.8, 110.4, 110.6, 110.5, 110.5, 41372.1, 110.6, 110.4, 110.4. Como o experimento em questão é a inicialização a partir do menor nó, a seqüência de eventos ocorrida a partir da inicialização do nó 15 foi:

(a) Inicialização do nó 15.

(b) Nó 15 propõe novo grupo aos demais processadores contendo somente este nó.

(c) Isoladamente dos outros, cada nó cria um novo grupo contendo os processadores atuais e o nó 15, e enviam esta informação para todos os nós. Note que são enviadas o equivalente a $14 * 15 = 210$ mensagens[17].

(d) O nó 15 recebe algumas destas mensagens e se une ao novo grupo. Note que neste momento o conjunto de membros já se estabilizou, mas há grandes possibilidades do `gid` ainda estar instável.

(e) Cada nó define seus timeouts, e aguardam o momento de varrer o grupo em busca de instabilidades.

(f) Após o tempo estipulado ter decorrido[18], começa a verificação pelo grupo. Neste momento é percebido que o nó 15 está unido ao grupo errado, necessitando atualizar seu `gid`.

(g) O nó 15 atualiza o `gid` e finalmente o cluster se estabiliza.

Portanto, com base nesta descrição e nos tempos observados, concluímos que os tempos de estabilização possuem uma distribuição discreta: seus possíveis valores seguem diretamente múltiplos de π. A grande maioria se estabilizou durante o momento 6d, mas em dois experimentos a estabilização se deu em somente em 6g. Isto só pode ter ocorrido devido a perdas de mensagens, causadas provavelmente por alguma carga no ambiente. Os resultados que apresentam a mediana (tabela 6.7 e figura 6.15) corroboram esta tese: o gráfico exibe que a mediana destas estabilizações quase sempre está perto de 100ms, o que significa que a estabilização ocorreu no momento 6d. Alguns picos também ocorreram, mas eles são decorrência de cargas na máquina e de osilações no ambiente.

Assim, este é um comportamento natural do protocolo, que não deve ocorrer freqüentemente em clusters reais corretamente dimensionados.

[17] Antes da inicialização do nó 15 o grupo consistia de 14 nós, que agora enviam esta mensagem para todos os 15 novos membros.

[18] Este tempo é a constante π, que foi definida em 40 segundos durante nossos experimentos.

7. A perda de mensagens influencia bastante o tempo de estabilização. Como o tempo de estabilização possui uma distribuição discreta, se em t_0 ocorrer a perda de uma mensagem, o grupo irá permanecer instável. Esta instabilidade poderá ser corrigida na próxima rodada, que ocorrerá em $t_0 + \Delta$. Desta maneira, a cada rodada que houver a perda de alguma mensagem necessária para a estabilização do grupo será necessário aguardar mais um intervalo Δ, o que acaba aumentando bastante o tempo de estabilização, pois esta constante tende a ser relativamente grande.

8. Falhas simultâneas tendem a deixar o estado do cluster errado por um longo tempo. Detectamos que a falha simultânea de 99 nós acarreta em um tempo de estabilização de aproximadamente 1 hora e 40 minutos. Isto ocorre porque em cada rodada de varredura do grupo em buscas de instabilidade é detectada somente a falha de um nó. Assim, são necessárias 99 rodadas, o que gera este alto tempo de estabilização.

9. De acordo com a ordem de precedência estabelecida entre os processadores, a finalização dos maiores nós gera um tempo de estabilização maior do que a finalização dos menores nós. Este comportamento é detectado ao compararmos os gráficos 6.20 e 6.22.

   Esta diferença de resultados deve-se à forma como são calculados os timeouts em cada nó. Após se unir a um grupo, cada nó calcula o tempo máximo que esperará por uma mensagem de seu vizinho da esquerda. Este tempo é composto do intervalo π e de por um peso baseado na ordem do processador dentro deste novo grupo. O primeiro nó tem o maior peso, seguido pelo último nó, descrescendo até o segundo nó, que é aquele com menor peso. Assim, o menor timeout sempre será o do segundo nó, e o maior do último nó. Desta forma, sempre esperaremos um tempo de no máximo $\pi + \delta$ após a finalização do menor nó, e um tempo entre $\pi + \delta$ e $\pi + n\delta$ após a finalização do maior nó, onde n é a quantidade de nós no grupo.

6.4 Futuras Melhorias

Após o término do desenvolvimento, algumas idéias surgiram para aprimorar a implementação do serviço de pertinência:

1. Alteração do formato das mensagens. O formato implementado para a troca de mensagens é XML. As desvantagens observadas deste formato são:

   (a) Cada mensagem fica muito maior do que realmente precisaria ser.

   (b) A avaliação de uma mensagem torna-se potencialmente mais custosa.

Para melhorar estes pontos, as mensagens poderiam ser binárias ou simplesmente em um formato texto mais enxuto. Uma solução imediata seria compactar as mensagens XML antes de enviá-las. A biblioteca libxml2 oferece esta capacidade, a qual foi utilizada durante os experimentos. Entretanto, qualquer solução de otimização no tamanho das mensagens deve ser analisada para comprovar que seu custo de processamento não impacta o serviço.

2. Criar um mecanismo de validação de mensagens. A implementação atual não realiza nenhuma verificação sobre as mensagens recebidas, a fim de determinar se elas estão bem formadas. Como os nós enviam sempre mensagens no formato definido, isto não se torna um problema. Entretanto, o recebimento de uma mensagem originada de um nó fora do grupo pode causar um erro de avaliação da mensagem, que pode derrubar o serviço. Utilizando-se ainda XML como estrutura, esta validação poderia ser feita a partir das definições das mensagens, que podem ser escritas em DTD ou *schemas*.

3. Alteração do mecanismo de detecção de falhas. O mecanismo de detecção de falhas proposto por Cristian usa um anel lógico, por onde uma mensagem é trafegada. Esta mensagem é originada pelo nó líder do grupo a cada π unidades de de tempo, e cada outro nó simplesmente a repassa para seu vizinho da direita. A ausência de tal mensagem indica a falha de um nó, que é detectada pelo seu vizinho da direita.

   Uma outra opção para o mecanismo de detecção de falhas seria fazer com que cada nó gerasse a cada π unidades de tempo esta mensagem de verificação, e a enviasse a seu vizinho da direita. Todo nó deveria esperar por esta mensagem, e o não recebimento dela implicaria na falha de seu vizinho da esquerda. Ao receber esta mensagem, um nó simplesmente teria a certeza de que seu vizinho da esquerda está funcionando corretamente, e não a repassaria adiante para formar um anel. Enquanto que no modelo de detecção de falhas implementado cada nó aguarda o recebimento de uma mensagem "estou-vivo" para enviar uma destas mensagens a seu vizinho (formando assim um anel lógico), neste modelo proposto cada nó envia independentemente dos outros estas mensagens. Este modelo é similar ao modelo PMC discutido na seção 3.5.5.

   Esta alteração permitiria uma detecção mais rápida de falhas, pois não seria necessário navegar uma mensagem por todo o grupo. Ainda, esta alteração poderia tirar a $O(n^2)$ do tempo máximo de estabilização do grupo. Entretanto, antes de ser aplicada, testes e estudos deveriam ser realizados a fim de se determinar se nenhum efeito colateral surge com esta alteração

4. Criação de um mecanismo de IPC. Para se construir um sistema de alta disponibilidade, é imprescindível que seja possível cadastrar processos clientes ao serviço de pertinência, a fim de que estes sejam notificados de alterações no grupo. A implementação atual contém um módulo responsável por esta notificação a clientes. Entretanto, este módulo praticamente

não está implementado. Assim, um mecanismo de cadastro de clientes e chamada de rotinas de *callback* deve ser implementado para tornar esta implementação utilizável por um sistema de alta disponibilidade. Idealmente, esta implementação deveria ser fornecida pelo framework OCF.

5. Rever o mecanismo de multicast. O mecanismo de multicast é muito eficiente para comunicações de grupo, mas possui inconvenientes. O tamanho das mensagens neste modelo é limitado, o que pode limitar a escalabilidade do serviço, pois quanto mais nós existirem no grupo, maiores serão as mensagens trocadas entre seus membros. Durante nossos experimentos com 100 máquinas nenhum problema foi encontrado. Entretanto, este mecanismo precisa ser revisto, a fim de não se tornar um impedimento ao crescimento do cluster.

Conclusão

> "If you have an apple and I have an apple and we exchange apples then you and I will still each have one apple. But if you have an idea and I have one idea and we exchange these ideas, then each of us will have two ideas".
>
> George Bernard Shaw

Esta dissertação cobriu uma vasta área. Partindo de clusters, estudamos seus principais conceitos e entendemos como sua arquitetura pode ser descrita. Entramos então no campo de tolerância a falhas, onde estudamos os principais termos e classificações usadas. Com este dois conceitos definidos, pudemos entender como clusters podem prover alta disponibilidade a seus recursos. Finalmente, estudamos um dos principais serviços que clusters devem oferecer: o serviço de pertinência. Percebemos que este serviço tem um objetivo relativamente simples, mas que pode ser atingido de vários modos diferentes. Abordamos na seqüência o projeto *Open Clustering Framework*, que visa suprir a necessidade da existência de um padrão para clusters. Desenvolvemos em seguida um serviço de pertinência, onde pudemos vivenciar as teorias estudadas, e perceber as dificuldades decorrentes deste tipo de atividade. Após termos concluído todos estes estudos, podemos fazer várias conclusões:

1. Clusters são muito poderosos. Por serem constituídos unicamente de máquinas regulares que em conjunto realizam tarefas muito importantes, os clusters tornam-se altamente escaláveis. Existem inúmeras possibilidades para seu uso, o que impulsiona o estudo e desenvolvimento desta área.

2. Clusters ainda são pouco explorados. Um ponto observado neste estudo de clusters é que, apesar de serem poderosos, eles ainda não são muito utilizados de maneira geral. Ainda não criamos a cultura de aproveitar máquinas mais antigas para formar clusters mais poderosos Como o custo de uma única máquina com grande capacidade é muito alto, deveríamos buscar mais soluções que impliquem na construção de clusters computacionais. Estas soluções

são consideravelmente mais baratas, e seu uso contribui para o desenvolvimento da área de sistemas distribuídos.

3. Termos relacionados a tolerância a falhas são muitas vezes ambíguos ou pouco claros. Concluímos com nosso estudo de alta disponibilidade que muitos termos são ambíguos, ou até são usados de maneiras diferentes. Por exemplo, sem uma definição anterior, qualquer leitor teria dificuldade para compreender a diferença entre *confiabilidade* e *dependabilidade*. Consideramos que nosso trabalho ajuda a minimizar este problema, pois ele apresenta todas as definições necessárias, sem depender de nenhum conhecimento anterior.

4. A área de alta disponibilidade possui muitas nuâncias. Quando falamos em *alta disponibilidade*, muitas coisas precisam ser definidas. A transparência das falhas e o tipo de recuperação de serviços são exemplos destas definições. Ainda, todo o processo de detecção, confinamento e recuperação de erros também precisa estar definido. Geralmente não percebemos os reais detalhes envolvidos, e simplesmente dizemos que um sistema possui *alta disponibilidade*. Isto causa muita confusão ao se analisar diferentes sistemas de alta disponibilidade, pois sem estes detalhes não é possível detectarmos suas reais características.

5. Um cluster de alta disponibilidade definitivamente não é um sistema simples. Para um sistema de alta disponibilidade ser realmente útil e completo, ele precisa oferecer mais do que garantias de disponibilidade de recursos. Assuntos como replicação de dados e sincronização de estado também precisam ser tratados. Cada um destes assuntos também irá possuir seus detalhes, aumentando assim a complexidade envolvida com clusters de alta disponibilidade. Desta maneira, um sistema de alta disponibilidade completo pode tornar-se muito complexo, sendo que muitas vezes nosso requisito não pede toda esta complexidade. Antes de optarmos por uma solução complexa de disponibilidade devemos analisar se realmente precisamos deste tipo de solução.

6. Pouca pesquisa específica para serviços de pertinência. Nosso estudo mostrou que freqüentemente o serviço de pertinência é estudado em conjunto com o tema de comunicação de grupos. Isto demonstra que, muitas vezes, para atingir propriedades como ordenação e sincronia virtual os serviços de comunicação e pertinência precisam estar mais amarrados. Para definir uma arquitetura padrão de clusters é necessário compreender a real dependência entre estes serviços, de maneira a isolá-los e poder então compreender as particularidades do serviço de pertinência.

7. Dificuldade de se criar serviços de pertinência. Apesar de aparentemente simples, serviços de pertinência podem esconder algumas dificuldades para quem os cria. Esta é a conclusão feita quando observamos que muitas propostas de serviços estavam erradas, isto é, elas não

garantiam as propriedades que diziam garantir. Isto prova que o tema não é simples, e que a especificação e prova formal de um protocolo distribuído não são fáceis.

8. Algumas propriedades de pertinência são muito difíceis de serem implementadas. O estudo teórico do serviço de pertinência nos apresenta uma série de diferentes propriedades que podem ser oferecidas. Entretanto, um olhar mais detalhado nestas propriedades nos revela que na prática elas podem ser difíceis de implementar. Muitas delas se relacionam diretamente com a camada de comunicação, o que torna difícil criar uma arquitetura onde estas camadas estejam realmente isoladas.[1]

9. A padronização de clusters em Linux é fundamental para o aprimoramento da área. O fato de não existir em Linux uma grande referência de clusters faz com que este tipo de solução seja restrita a poucas empresas. Um padrão contribuiria muito para que a área de clusters se torne mais estável e madura, de maneira que a indústria em geral não sinta dúvidas quanto à qualidade e capacidade deste tipo de solução.

10. O processo de definição de um padrão é mais difícil do que pode parecer. No momento, o projeto OCF não conseguiu progredir o quanto pretendia. Após uma profunda discussão em algumas áreas, e nenhuma em outras, pouquíssimas especificações foram propostas. Mesmo as que estão há mais tempo em discussão ainda não estão totalmente fechadas. Isto demonstra a dificuldade inerente ao processo de definição de um padrão, que é acentuada no caso do OCF por se tratar de uma iniciativa open-source. Sentimos que houve uma queda de interesse no projeto, e esta queda se deve também a esta dificuldade de se produzir rapidamente algum resultado.

11. Protocolos distribuídos podem tornar-se muito complicados para se implementar e depurar. A implementação de protocolos distribuídos possuem todos os problemas decorrentes do uso de threads, com o agravante de não existir um gerenciador central da execução. Assim, a implementação e depuração de tais programas podem ser muito complexas, necessitando de considerável experiência e conhecimento do desenvolvedor. Ferramentas auxiliares e bibliotecas já desenvolvidas ajudam a diminuir o risco do projeto e dificuldades durante o processo de desenvolvimento.

12. Dificuldade de se testar uma implementação de serviço de pertinência. A ausência de um controlador central para gerenciar execuções torna o processo de se testar um protocolo distribuído muito complexo. Estratégias como gravar a execução em um arquivo de log para depois ser processado podem ser usadas, mas mesmo assim o processo não é simples.

[1] De certa maneira, isto justifica o fato do serviço de pertinência ser muitas vezes estudado em conjunto com serviços de comunicação.

Futuras Direções

Algumas futuras direções podem ser traçadas para este trabalho. Um primeiro ponto é o fortalecimento do projeto OCF. No momento ele não está muito ativo, o que acaba desinteressando as pessoas que acompanham o projeto. Se um dos requisitos do projeto é capturar a atenção das pessoas, o grupo precisa estar mais ativo para isto ocorrer.

Para tornar o OCF mais atrativo, existe uma tendência no grupo de discutir menos especificações, e produzir rapidamente algum código que possa servir como base para as especificações. Certamente esta abordagem traria mais atenção, mas é importante que as discussões de padrões continuem ocorrendo. Sem elas, a definição do padrão pode ser guiada pela implementação, o que poderia não seguir as abstrações encontradas em clusters.

Após todo este processo de fortalecimento e definição de padrões do OCF, será então possível adaptar a implementação do serviço de pertinência para estar compatível com a especificação aprovada. Como o código está bem modularizado, teoricamente esta adaptação não será complicada. Como a partir deste momento a implementação estaria totalmente baseada no framework OCF, seria então necessário utilizar uma implementação de framework OCF para executar o serviço de pertinência.

Cremos que, quando estiver de acordo com um padrão, esta implementação poderá ser muito útil para qualquer usuário de clusters, pois ela é simples, rápida, e eficiente para atender a maior parte das necessidades.

Termos em Inglês

A.1 Introdução

Por todo este texto utilizamos muitos termos na língua inglesa. Isto ocorreu por algumas razões, como:

- falta de uma tradução que fornecesse o devido sentido da palavra;
- a tradução mais conhecida na língua portuguesa perderia parte do sentido do termo, por esta ser pouco utilizada;
- o termo é muito conhecido e praticamente utilizado somente na língua inglesa.

Este apêndice visa reunir estes termos, indicando-os juntamente com as páginas onde apareceram.

A.2 Índice de Termos em Inglês

Documentos OCF

B.1 Introdução

Este apêndice visa reunir as especificações propostas para o OCF, em suas últimas versões. Todas as informações aqui reunidas foram divulgadas ou em [7] ou no próprio site do projeto [85], e estão apresentadas da mesma maneira que foram publicadas.

B.2 Regimento OCF

```
The Open Cluster Framework (OCF) - A proposed charter

Alan Robertson

   IBM Linux Technology Center

   <alanr@{unix.sh|us.ibm.com}>

David Ham

   Delft University of Technology
   Environmental Fluid Mechanics Section

   <D.A.Ham@citg.tudelft.nl>
     ------------------------------------------------------

   Table of Contents
```

One of the most commonly identified features which is felt to
be necessary for Linux(TM) to be considered "enterprise-ready"
is High-Availability. High-Availability (HA) systems provide
increased service availability through clustering techniques.
Linux is also well-known for its Beowulf High Performance
clustering software - which provides the most cost-effective
supercomputing available for any platform.

As befits the importance of these capabilities, a number of open source clustering projects have been created. These projects were been created independently largely for complex historical reasons, not generally because of political, philosophical or licensing differences. Because of these kinds of historical reasons, they originally shared little or no code. This minimised the benefits of the open source model, which both encourages and benefits from the sharing of common components. However, many of them share the need for a component for resetting cluster members. A component was created for filling this need with the specific intent of being a common component across open HA systems. This was quite successful, and this component has become standard across most open source HA systems. All of the projects involved have benefited from this commonality.

In light of this success, the first author began to search for more ways to extend these benefits across a broader set of cluster infrastructure components. Towards this end, we have begun an effort to create a standard Open Clustering Framework specification, and have begun to implement a reference implementation.

---

1. The Open Cluster Framework

1.1. OCF APIs

The Open Cluster Framework will define a series of APIs - a set of external APIs and a set of internal APIs. It is intended that many clustering systems will eventually conform to the applicable external APIs. Defining these APIs is fundamentally a standards effort whose output includes these APIs. A second part of the OCF effort is to define a reference implementation of the standards. It is expected that the reference implementation will be developed in parallel with the standards themselves.

Clustering implementations which are based on the the OCF reference implementation will also conform to the internal (component) APIs as well. It is perfectly acceptable for a completely closed system not based in any way on the reference implementation to conform to the internal APIs, but it is not expected to be a common occurrence.

1.2. OCF Reference Implementation

The OCF reference implementation will be available for download - and will be licensed under an open source license - see the section on OCF IP policies for more details. This effort will be a full-fledged community development effort to develop a set of common clustering utilities which could be used in a demanding commercial setting. It is also the intent of the reference implementation that it could be used in conjunction with commercial components and implementations.

---

2. The OCF Process

The OCF working group consists of a steering committee and a number of technical teams. This structure is designed to be as open as possible while providing clear decision making processes which facilitate progress.

2.1. Steering Committee

There will be a steering committee to discuss issues pertaining to the OCF project as a whole and to make decisions on those issues.

2.1.1. Formal functions

Among other things, the steering committee:

1. coordinates the work of the technical teams to ensure consistency and interoperability of OCF component APIs; and
2. endorses proposals of technical teams for OCF standard APIs or returns them to the technical team for revision; and
3. appoints its own chair; and
4. appoints the leaders of the technical teams.

2.1.2. Membership

The full members of the committee are a chair and the leader of each technical team. One person may be both a chair and a team leader simultaneously. Participation in the discussions of the committee is open to all OCF participants.

Only Individual and Corporate members of OCF may be full members of the steering committee.

The first steering committee will be appointed by consensus by the participants in OCF.

2.1.3. Decision Making

The chair will attempt to establish consensus among the participants in the steering committee. In the event that the chair considers that consensus is unlikely to be achieved in reasonable time, the issue is to be decided by a vote of the full members. In any such vote the chair has both a substantive and a casting vote.

2.2. Technical Teams

There will be a number of technical teams to develop the APIs of particular OCF components.

2.2.1. Formal Functions

The function of a technical team is to develop a proposed API for its OCF component. This proposal is then to be forwarded to the steering committee for endorsement.

2.2.2. Membership

Each technical team will have a leader appointed by the steering committee. Participation in a technical team is open to all OCF participants.

2.2.3. Decision making

The team leader will attempt to establish consensus among the participants in the steering committee. In the event that the team leader considers that consensus is unlikely to be achieved in reasonable time, the team leader may make the decision.

---

3. OCF Intellectual Property Policies

The OCF IP policies are designed to ensure that the standard is and remains open and is implementable as Open Source Software. There are three IP policies relating to copyright in the standards, copyright in reference implementations and patents. In addition, provision is made for a certification mark to be used by certified OCF compliant applications.

These policies are believed to be consistent with Free Standards Group policy and practice.

3.1. Copyright in Standards Documents

All OCF documentation will be available free of charge under the GNU Free Documentation License. In addition, any code which accompanies the standards documents, for example header files and example routines, will be released under the GNU Lesser General Public License. In both cases, copyright will remain with the authors or their assignee. A template copyright notice is provided in Appendix A

3.2. Copyright in Reference Implementations

Any reference implementation must be released under a licence compatible with the GNU General Public License. Copyright in reference implementations will remain with the authors or their assignee.

3.3. Patents

OCF operates a royalty free (RF) patent policy. OCF will endeavour to ensure that the OCF APIs are implementable without the payment of royalties to patent holders. In particular, members of OCF will certify that no royalties are required to be payed to them in order to implement the standard. A mechanism for such declarations will be determined, possibly based on the patent policies of the W3C.

3.4. Trademarks and Certification

The OCF may register a trademark which will be licensed for use by software which is certified to conform to the OCF standards. It is likely that there would be a charge associated with the certification process.

---

4. Ways to Participate in the OCF Standards Process

For legal reasons, it is our expectation that there will be three classes of participation in the OCF standards working group.

Participants
: people who subscribe to the associated standards mailing lists. All of these people will be informed of the IP policies of the OCF process, and their continued participation constitutes agreement with them.

Individual members
: people who actively contribute language and suggestions to the standards process and have formally agreed to be bound by the OCF IP policy as individuals (not their employers).

Corporate members
: people who participate on behalf of their company, and who will be asked to conform to the IP policy on behalf of their company, and who's company has signed the appropriate agreements for them to do so.

---

5. Ways to Conform to the OCF Standards

There are several ways which one can write software which conforms to the OCF standards. A few examples are given below:

* Provide a product consisting strictly of (a subset of) software components taken from the reference implementation.
* Provide a product consisting of mainly components from the reference implementation, with a few unique components - which are known to meet the standard APIs.

* Provide a product consisting of a few components from the reference implementation, and a large body of software not from the reference implementation, but with a set of interfaces to the functions provided by this product which conform to the standardised APIs.

All of these are legitimate ways to conform to the standard. The first ones are lower-cost to write and maintain, and enjoy the well-known benefits of the open source development model. A company may start out with the in one model, and migrate to the "mostly open" approach over time.

This flexibility will allow a company to continue to meet the needs of their customers, and provide familiar interfaces, while migrating over time to have less and less unique software in their package. This lets them concentrate their development effort on the components which their customers have the most unique needs.

---

6. Amendments

This charter may be amended from time to time according to the following process:

1. A proposed amendment shall be considered by the steering committee in accordance with its normal processes. Two thirds of the votes cast for and against the motion are required for the amendment to pass.
2. The chair of the steering committee shall put the motion to a vote of active OCF participants. OCF participants who have participated for at least 3 months and who have participated (for example by posting to the mailing list) within the last 3 months shall be eligible to vote. Questions of eligibility to vote shall be determined by the chair of the steering committee. Two thirds of the votes cast for and against the motion are required for the amendment to pass.

```
   ----------------------------------------------------------

A. Template for Copyright Notices

   The copyright notice on an API definition or other standards
   document should have the following form:

     Permission is granted to copy, distribute and/or modify
     this document under the terms of the GNU Free Documentation
     License, Version 1.2 or any later version published by the
     Free Software Foundation; with no Invariant Sections, with
     no Front-Cover Texts, and with no Back-Cover Texts. A copy
     of the license is included in the section entitled "GNU
     Free Documentation License".

     In addition, any source code in this document is is free
     software; you can redistribute it and/or modify it under
     the terms of the GNU Lesser General Public License as
     published by the Free Software Foundation; either version
     2.1 of the License, or (at your option) any later version.

     This library is distributed in the hope that it will be
     useful, but WITHOUT ANY WARRANTY; without even the implied
     warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
     PURPOSE. See the GNU Lesser General Public License for more
     details.

     You should have received a copy of the GNU Lesser General
     Public License along with this library; if not, write to
     the Free Software Foundation, Inc., 59 Temple Place, Suite
     330, Boston, MA 02111-1307 USA
```

B.3 Especificações

As especificações propostas pelo grupo até o momento são:

B.3.1 Cluster Event Notification API

CLUSTER EVENT NOTIFICATION API.

Draft version 0.4

1. Introduction

The Cluster Event Notification API defines an asynchronous delivery framework for cluster events. This document describes the OCF Cluster Event Notification framework.

The event service uses a publish/subscribe model, where one or more publishers can post events that will be delivered to all subscribers interested in (subscribed to) events about certain topics.

In this model, event data that passes through the event service is opaque to the event service itself and only makes sense in the context of a particular event topic.

Event topics are administratively created. Some topics will be pre-created to align with common architectural components of a cluster.

2. Overview

In the picture below, the cluster software components on the left are responsible for generating events and subsequently publishing these events to the event service for further distribution to all subscribers. A subscriber can be either an individual application or a set of topic specific data filters.

Data passing through the event service is opaque to the event service.

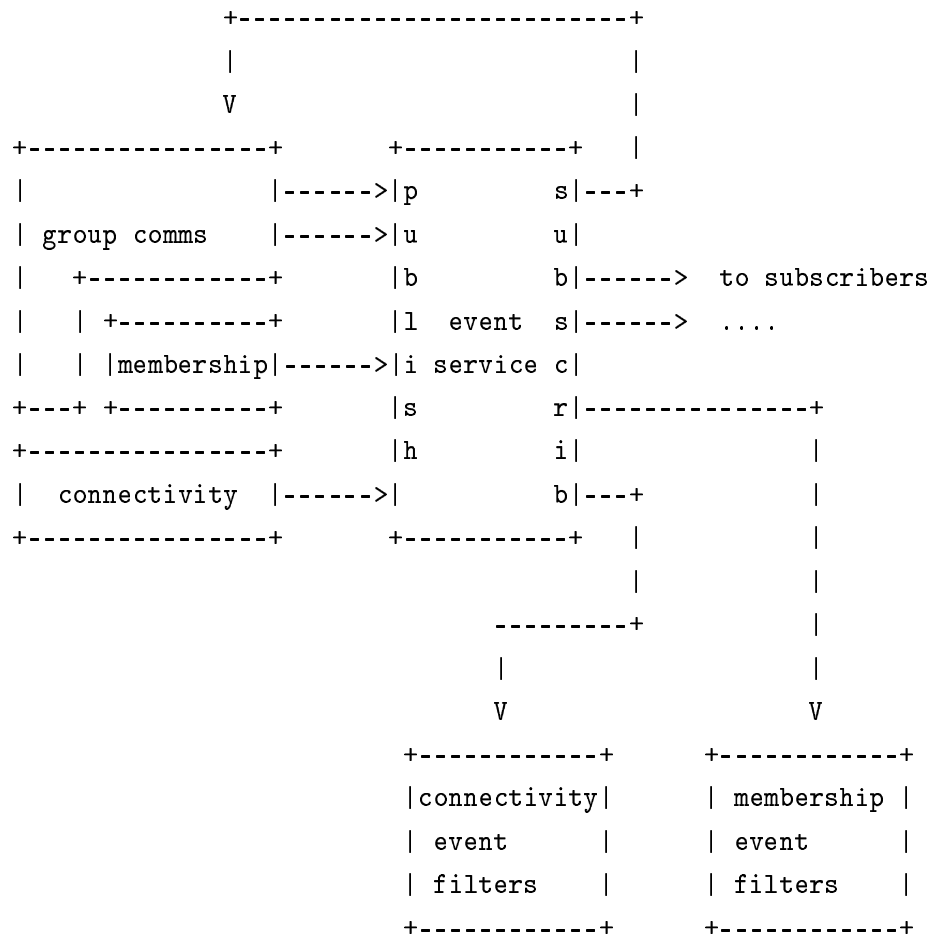

Figura B.1: Arquitetura Proposta para o Framework OCF

3. Event delivery API

This section describes the API for cluster event notification service. Unless otherwise specified, errors are indicated by non-zero return values. Errors with specific meanings are listed below. Common errors are listed here once for brevity.

EPERM need suitable privileges to subscribe

ENOMEM insufficient memory to complete request

3.0 Event Message Format

All events share a common message format that includes an event header, some event properties and the actual event data.

Although each event topic uses its own data format, each supply a common event header that describes the event. These are the components of an oc_ev_event_t.

An Event Message:

size	int32	size in bytes of this event message, including the event header, and all event data
offset	int32	the offset in bytes from the beginning of the event message to the event data
ev_id	int32	event id
re_id	int32	reference id that links this message to a particular event id
seq	int32	sequence number for this message in this topic
topic	string	the topic name for reference
<<<XXX add other header fields here>>>		
data	bytes	topic specific data, format specified externally

Each event topic has a corresponding data format. All events in a given topic supply data in the appropriate format.

3.1 Event service subscription

```
SYNOPSIS
        void *oc_ev_subscribe(char *topic, const oc_ev_on_event_t *fn);

        topic   the topic to associate with 'fn'

        fn      is the on_event function (see description below).
                If NULL, must use oc_ev_get_event().

DESCRIPTION
        This is the initial call to subscribe for notification of
        events in any published topic.  A separate subscription is
        required to enable notification for events in each topic of
        interest.  Multiple topics can be active at the same time.
        Any limit on concurrent open topics is imposed by an
        implementation, not by this specification.

        Events are delivered only for event topics with an active
        subscription.  A subscription can be termintated at any time
        [see oc_ev_unsubscribe()].

        A process can choose between two event delivery mechanisms.
        When an "on_event" function is supplied, an asynchronous
        callback method will be used.  When the "on_event" function
        is NULL, a subscriber is responsible for fetching events on
        its own [see oc_ev_get_event()].

RETURN VALUE
        Upon success, callers receive an event descriptor that must
        be passed into subsequent event service calls.  At user-level,
        the event descriptor can be converted into a file descriptor
        suitable for poll or select [see oc_ev_get_fd()].

        Failure returns NULL and sets errno.

ERRORS
        EINVAL  specified topic is invalid
```

3.1.1 on_event Callback Function

SYNOPSIS

```
typedef oc_ev_on_event_t void fn(const oc_ev_event_t *event);
```

event The event data varies based on the event topic. Events are allocated by the event service and are valid until oc_ev_event_done() is called. Delivery of subsequent events may be inhibited while another event is outstanding.

DESCRIPTION

An on_event function is optionally passed into oc_ev_subscribe() to indicate the use of an automatic callback.

3.2 Event service termination

SYNOPSIS

```
int oc_ev_unsubscribe(void *ed);
```

ed is the event service descriptor obtained from call to oc_ev_subscribe().

DESCRIPTION

Calling oc_ev_unsubscribe() will cancel the subscription for this event descriptor. This routine can be safely called from an on_event routine.

RETURN VALUE

oc_ev_unsubscribe() returns zero on success or -1 on error. Upon successful return, no further events will be delivered.

ERRORS

EINVAL ed not recognized by event service

```
3.3  Converting an event descriptor to a file descriptor

SYNOPSIS
        int oc_ev_get_fd(void *ed);

        ed      is the event service descriptor obtained
                from call to oc_ev_subscribe().

DESCRIPTION
        When a select/poll is required, an event descriptor can be
        converted into a file descriptor.

RETURN VALUE
        oc_ev_get_fd() will return a file descriptor or -1 if an error
        occurred, in which case errno is set appropriately.

ERRORS
        EINVAL  ed not recognized by event service


3.4  Automatic Event Delivery

SYNOPSIS
        void oc_ev_handle_events(void);

DESCRIPTION
        After subscribing with on_event functions to all desired topics,
        this function will pass control of the calling thread to the
        event service.  The event service will monitor the subscribed
        event topics and call the appropriate on_event function once
        for each event.

RETURN VALUE
        This function will return when there are no more subscribed
        topics.
```

NOTE: This function does nothing for kernel clients.

ERRORS

ENOENT on_event function not specified

3.5 Manual Event Extraction

SYNOPSIS

```
oc_ev_event_t *oc_ev_get_event(void *ed, int timeout);
```

ed is the event service descriptor obtained from call to oc_ev_subscribe().

timeout time to wait (in milliseconds) for a new event to arrive. If the timeout value is negative, oc_ev_get_event() will wait forever for a new event.

DESCRIPTION

As an alternative to using the on_event callback method of event delivery, a subscriber may extract events at their leisure, unless an on_event function was specified.

A user-level process determines that an event is pending using select/poll on the file descriptor acquired by oc_ev_get_fd().

If no event is pending, this call will block for 'timeout' milliseconds waiting for an event to arrive.

RETURN VALUE

On success, a pointer to an event structure is returned. The event pointer is valid until oc_ev_event_done() is called with this pointer.

If no event occurs in the specified 'timeout' window, a NULL will be returned and errno will be set to ETIME. If an error occurs, a value of NULL will be returned and errno will be

```
        set to something other than ETIME.

ERRORS
        ETIME   timeout expired and no event arrived

        EINVAL  ed not recognized by event service

        EEXIST  an on_event function was specified


3.6  Event Completion

SYNOPSIS
        int oc_ev_event_done(const oc_ev_event_t *event);

        event   the event message address passed to an on_event
                callback function or returned from oc_ev_get_event().

DESCRIPTION
        It is necessary to inform the notification service that event
        processing is complete for each event received.  This call is
        implicit with Automatic Event Delivery and will be performed
        after each on_event callback returns.

RETURN VALUE
        A return of zero indicates success and any data associated with
        this completed event is no longer valid.  A -1 will be returned
        if an error occurred.

ERRORS
        EINVAL  address not recognized by event service


3.7  Version number

SYNOPSIS
        int oc_ev_get_version(oc_ver_t *ver);
```

```
        ver     the version number of the service.

DESCRIPTION
        This is a synchronous call to return the event notification
        service version number.  It is safe to call anytime.
        (XXX what is an oc_ver_t - i.e., what does version data look like?)

ERRORS



4.  API Revision History

        * January 27, 2003 - Draft 0.4
                event descriptor changed to void * (was int),
                edited (improved?) function descriptions,
                reformatted to better resemble man pages,
                event_done() only needed with get_event()

        * December 19, 2002 - pre Draft 0.4
                removed all non-event subject matter (membership, etc),
                name changes for clarity:
                        replaced 'register' with 'subscribe',
                        replaced 'class' with 'topic',
                        replaced 'callback' with 'on_event'
                added get_event() interface for manual event processing,
                changed semantics of handle_event():
                        now more automatic event processing,
                        renamed to 'handle_events' [notice the plural],
                        different from get_event()

        * August 13, 2002 - Draft 0.3
                oc_ev_subscribe() now returns file descriptor,
                individual file descriptors per event class (or group),
                dropped oc_ev_set_callback(),
                dropped oc_ev_activate(),
                renamed oc_ev_callback_done() to oc_ev_event_done(),
                removed tokens - only fd is needed,
```

```
                removed cookies - only data address is needed,
                size and event descriptor now embedded in event data,
                renamed m_instance to m_sequence in membership data,
                updated examples to match new functions.

        * June 05, 2002 - Draft 0.2
                updated definitions,
                clarifications for event delivery,
                error handling improvements,
                removed priority bands,
                removed SIGNAL and NORETURN activation styles,
                added membership event semantics.

        * April 12, 2002 - Draft 0.1
                describes general event delivery.

        Individual contributors, ordered by last name:
                Joe DiMartino <joe@osdl.org>
                Mark Haverkamp <markh@osdl.org>
                Daniel McNeil <daniel@osdl.org>
                Ram Pai <linuxram@us.ibm.com>
```

B.3.2 Membership API

```
#ifndef OCF_OC_MEMBERSHIP_H
# define OCF_OC_MEMBERSHIP_H
/*
 * <ocf/oc_membership.h> - membership APIs (version 0.1)
 *
 * The structures and functions in this header file work closely with
 * the oc_event.h event infrastructure.  All (edata, esize) parameters
 * to functions in this header file refer to membership event bodies.
 * It is expected that all such are received by this mechanism.
 *
 *
 * There are a few things in this header file which don't really belong here
 * but are needed and they aren't in any other header file.
 *
```

```
 * These are:
 * definition of oc_node_id_t
 * oc_cluster_handle_t
 *
 * Maybe we ought to put common types into an <ocf/oc_types.h>
 *
 * The oc_cmp_node_id() and oc_localnodeid() functions also belong in
 * some more global header file.
 *
 * oc_member_eventttype_t and * oc_member_uniqueid_t are membership-unique
 * and don't belong in a set of ocf-common header files (IMHO)
 *
 * Copyright (C) 2002 Alan Robertson <alanr@unix.sh>
 *
 * This copyright will be assigned to the Free Standards Group
 * in the future.
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of version 2.1 of the GNU Lesser General Public
 * License as published by the Free Software Foundation.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 *
 */

#include <stddef.h>
#include <ocf/oc_event.h>

#ifdef HAVE_UUID_UUID_H
# include <uuid/uuid.h>
#else
```

```
typedef unsigned char uuid_t[16];

#endif
/* controversial? */
typedef void * oc_cluster_handle_t;
typedef uuid_t oc_node_id_t;
typedef enum oc_member_eventtype_e oc_member_eventtype_t;
typedef struct oc_member_uniqueid_s oc_member_uniqueid_t;

/*
 * A few words about the oc_node_id_t:
 *
 * An oc_node_id_t is assigned to a node no later than when it first
 * joins a cluster, and it will not change while that node is active
 * in some partition in the cluster.  It is normally expected to
 * be assigned to a node, and not changed afterwards except by
 * adminstrative intervention.
 *
 * The mechanism for assigning oc_node_id_t's to nodes is outside the
 * scope of this specification.  The only basic operation which
 * can be performed on these objects is comparison.
 *
 * See oc_cmp_node_id() for comparisons between them.
 */

/*
 * oc_member_uniqueid_t
 * The values of these fields are guaranteed to be the same across
 * all nodes within a given partition, and guaranteed to be different
 * between all active partitions in the cluster.
 *
 * In other words, if you exchange current oc_member_uniqueid_t objects
 * with another cluster node, you can tell with certainty, whether or not
 * you and the other node are currently members of the same partition.
 *
 * The m_instance field is guaranteed to be unique to a particular
 * membership instance while that node is active in the cluster.
```

```
 * If a node is shut down and restarts, then the m_instance might
 * repeat a value it had in the past.
 *
 * See oc_cmp_uniqueid() for comparing them.
 *
 * The meaning of the uniqueid field is not defined by this specification.
 * It may be the node_id of a node in the cluster or it may be a unique
 * checksum or it may be some other value.  All that is specified is that
 * it and the m_instance are unique when taken as a whole.
 */
typedef unsigned char oc_mbr_uuid[16];
struct oc_member_uniqueid_s {
unsigned m_instance;
oc_mbr_uniqueid uniqueid;
};
/*
 * This enumeration is used both to indicated the type of an event
 * received, and to request the types of events one wants delivered.
 * (see oc_member_request_events() and oc_member_etype() for more
 * details on how this is used).
 */

enum oc_member_eventtype_e {
OC_NOT_MEMBERSHIP,/* Not a (valid) membership event */
OC_FULL_MEMBERSHIP,/* full membership update */
OC_INCR_MEMBERSHIP,/* incremental membership update */
};


#ifdef __cplusplus
extern "C" {
#endif

/*
 * Returns 0 for equal node_ids,
 * negative for node id l less than node id r
 * positive for node id l greater than node id r
 *
```

```
 * No meaning may be ascribed to the fact that a particular
 * node id is greater or less than some other node id.
 * The comparison operator is provided primarily for
 * equality comparisons, and secondarily for use in
 * sorting them into a canonical order.
 */
int oc_cmp_node_id(oc_node_id_t l, oc_node_id_t r);


/* Return our local node id */
int oc_localnodeid(oc_node_id_t* us, oc_cluster_handle_t handle);
/*
 * On failure these functions return -1:
 * The following errno values are defined:
 * EINVAL invalid handle argument
 * EL2HLT cluster software not currently running
 */

/* What kind of event did we get? */
/* (see oc_member_request_events() for more details) */
oc_member_eventtype_t oc_member_etype(const void* edata, size_t esize);

/*
 * oc_member_uniqueid() returns the unique identifier associated
 * with this membership event.  See the description in the typedef
 * for more details.
 */
int oc_member_uniqueid(const void* edata, size_t esize,
oc_member_uniqueid_t* u);
/*
 * Failure of these functions return -1.
 * The following errno values are defined:
 * EL2HLT cluster software not currently running
 * EINVAL edata does not refer to a membership event
 */

/* How many nodes of each category do we have? */
int oc_member_n_nodesjoined(const void* edata, size_t esize);
```

```
int oc_member_n_nodesgone(void* edata, size_t esize);
int oc_member_n_nodesconst(void* edata, size_t esize);
/*
 * Failure of these functions return -1.
 * The following errno values are defined:
 * EL2HLT cluster software not currently running
 * EINVAL edata does not refer to a membership event
 * ENOSYS edata refers to an OC_INCR_MEMBERSHIP update, and
 * oc_member_n_nodesconst() was called.
 */

/* What nodes of each category do we have? */
oc_node_id_t* oc_member_nodesjoined(const void* edata, size_t esize);
oc_node_id_t* oc_member_nodesgone(void* edata, size_t esize);
oc_node_id_t* oc_member_nodesconst(void* edata, size_t esize);
/*
 * Failure of these functions return NULL.
 * The following errno values are defined:
 * EL2HLT cluster software not currently running
 * EINVAL edata does not refer to a membership event
 * ENOSYS edata refers to an OC_INCR_MEMBERSHIP update, and
 * oc_member_nodesconst() was called.
 */

/*
 *
 * OC_NO_MEMBERSHIP
 * No membership events will be delivered.  This is the default on opening
 * a membership event connection.
 *
 * OC_FULL_MEMBERSHIP
 * Deliver all membership information including information on
 * members that didn't change.  In this mode, the oc_member_nodesconst()
 * call is supported.
 *
 * OC_INCR_MEMBERSHIP
 * Deliver only changed membership events.  In this mode, calls to
 * oc_member_nodesconst(), et al. are not supported.
```

```
 *
 * Setting OC_FULL_MEMBERSHIP or OC_INCR_MEMBERSHIP will result in the
 * delivery of a single OC_FULL_MEMBERSHIP event soon after making
 * this call.  Subsequent events will be delivered as received in the
 * requested style (incremental or full).  Because events may already
 * be pending when this operation is issued, no guarantee can be made
 * regarding when this triggered event will be delivered.
 *
 */
int oc_member_request_events(oc_member_eventtype_t etype, oc_ev_t token);
/*
 * On failure this function returns -1:
 * The following errno values are defined:
 * EINVAL invalid etype or handle argument
 * EL2HLT cluster software not currently running
 * EBADF invalid oc_ev_t token parameter
 */

/*
 * if  l.m_instance < r.m_instance then return -1
 * if  r.m_instance > r.m_instance then return 1
 * if l.m_instance == r.m_instance and l.uniqueid == r.uniqueid
 * then return 0
 * otherwise return 2
 */
int oc_cmp_uniqueid(const oc_member_uniqueid_t l, const oc_member_uniqueid_t r);

#ifdef __cplusplus
}
#endif

#endif
```

B.3.3 Resource API

Além de conter a especificação da API de recursos, esta seção contém a definição da configuração de recursos, juntamente com um exemplo de tal configuração.

Resource Agent API

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

0. Header

Topic: Open Clustering Framework Resource Agent API
Editor: Lars Marowsky-Brée <lmb@suse.de>
Revision: $Id: resource-agent-api.txt,v 1.6 2003/07/30 20:13:29 lmb Exp $
URL: http://www.opencf.org/standards/resource-agent-api.txt

1. Abstract

Resource Agents (RA) are the middle layer between the Resource Manager
(RM) and the actual resources being managed. They aim to integrate the
resource type with the RM without any modifications to the actual
resource provider itself, by encapsulating it carefully and providing
generic methods (actions) to operate on them.

The RAs are obviously very specific to the resource type they operate
on, however there is no reason why they should be specific to a
particular RM.

The API described in this document should be general enough that a
compliant Resource Agent can be used by all existing resource managers /
switch-over systems who chose to implement this API either exclusively
or in addition to their existing one.

1.1. Scope

This document describes a common API for the RM to call the RAs so the pool of available RAs can be shared by the different clustering solutions.

It does NOT define any libraries or helper functions which RAs might share with regard to common functionality like external command execution, cluster logging et cetera, as these are NOT specific to RA and are defined in the respective standards.

1.2. API version described

This document currently describes version 1.0 of the API.

2. Terms used in this document

2.1. "Resource"

A single physical or logical entity that provides a service to clients or other resources. For example, a resource can be a single disk volume, a particular network address, or an application such as a web server. A resource is generally available for use over time on two or more nodes in a cluster, although it usually can be allocated to only one node at any given time.

Resources are identified by a name that must be unique to the particular resource type. This is any name chosen by the administrator to identify the resource instance and passed to the RA as a special environment variable.

A resource may also have instance parameters which provide additional information required for Resource Agent to control the resource.

2.2. "Resource types"

A resource type represents a set of resources which share a common set of instance parameters and a common set of actions which can be performed on resource of the given type.

The resource type name is chosen by the provider of the RA.

2.3. "Resource agent"

A RA provides the actions ("member functions") for a given type of resources; by providing the RA with the instance parameters, it is used to control a specific resource.

They are usually implemented as shell scripts, but the API described here does not require this.

Although this is somewhat similar to LSB init scripts, there are some differences explained below.

2.4. "Instance parameters"

Instance parameters are the attributes which describe a given resource instance. It is recommended that the implementor minimize the set of instance parameters.

The meta-data allows the RA to flag one or more instance parameters as 'unique'. This is a hint to the RM or higher level configuration tools that the combination of these parameters must be unique to the given resource type.

An instance parameter has a given name and value. They are both case sensitive and must satisfy the requirements of POSIX environment name/value combinations.

3. API

3.1. API Version Numbers

The version number is of the form "x.y", where x and y are positive
numbers greater or equal to zero. x is referred to as the "major"
number, and y as the "minor" number.

The major number must be increased if a _backwards incompatible_ change is
made to the API. A major number mismatch between the RA and the RM must be
reported as an error by both sides.

The minor number must be increased if _any_ change at all is made to the
API. If the major is increased, the minor number should be reset to
zero. The minor number can be used by both sides to see whether a
certain additional feature is supported by the other party.

3.2. Paths

The Resource Agents are located in subdirectories under
"/usr/ocf/resource.d".

The subdirectories allow the installation of multiple RAs for the same
type, but from different vendors or package versions.

TODO: As the resource type is also embedded in the metadata returned
from the RA, should this sentence go away and instead have the RM
discover the installed resource types?

The filename within the directories map to the resource type name
provided and may be a link to the real location.

Example directory structure:
FailSafe -> FailSafe-1.1.0/
FailSafe-1.0.4/
FailSafe-1.1.0/
heartbeat -> heartbeat-1.1.2/
heartbeat-1.1.2/

How the RM choses an agent for a specific resource type from the available set is implementation specific.

3.3. Execution syntax

After the RM has identified the executable to call, the RA will be called with the requested action as its sole argument.

To allow for further extensions, the RA shall ignore all other arguments.

3.4. Resource Agent actions

A RA must be able to perform the following actions on a given resource instance on request by the RM; additional actions may be supported by the script for example for LSB compliance.

The actions are all required to be idem-potent. Invoking any operation twice - in particular, the start and stop actions - shall succeed and leave the resource instance in the requested state.

In general, a RA should not assume it is the only RA of its type running at any given time because the RM might start several RA instances for multiple independent resource instances in parallel.

Mandatory actions must be supported; _optional_ operations must be advertised in the meta data if supported. If the RM tries to call a unsupported action the RA shall return an error as defined below.

3.4.1. start

Mandatory.

This brings the resource instance online and makes it available for use. It should NOT terminate before the resource instance has either

been fully started or an error has been encountered.

It may try to implement recovery actions for certain cases of startup failures.

"start" must succeed if the resource instance is already running.

"start" must return an error if the resource instance is not fully started.

3.4.2. stop

Mandatory.

This stops the resource instance. After the "stop" command has completed, no component of the resource shall remain active and it must be possible to start it on the same node or another node or an error must be returned.

The "stop" request by the RM includes the authorization to bring down the resource even by force as long data integrity is maintained; breaking currently active transactions should be avoided, but the request to offline the resource has higher priority than this. If this is not possible, the RA shall return an error to allow higher level recovery.

The "stop" action should also perform clean-ups of artifacts like leftover shared memory segments, semaphores, IPC message queues, lock files etc.

"stop" must succeed if the resource is already stopped.

"stop" must return an error if the resource is not fully stopped.

3.4.3. status

Mandatory.

Checks and returns the current status of the resource instance. The throughness of the check is further influenced by the weight of the

check, which is further explained in 3.5.3.

It is accepted practice to have additional instance parameters which are not strictly required to identify the resource instance but are needed to monitor it or customize how intrusive this check is allowed to be.

Note that "status" shall also return a well defined error code (see below) for stopped instances, ie before "start" has ever been invoked.

3.4.4. recover

Optional.

A special case of the "start" action, this should try to recover a resource locally.

It is recommended that this action is not advertised unless it is advantageous to use when compared to a stop/start operation.

If this is not supported, it may be mapped to a stop/start action by the RM.

An example includes "recovering" an IP address by moving it to another interface; this is much less costly than initiating a full resource group fail over to another node.

3.4.5. reload

Optional.

Notifies the resource instance of a configuration change external to the instance parameters; it should reload the configuration of the resource instance without disrupting the service.

It is recommended that this action is not advertised unless it is advantageous to use when compared to a stop/start operation.

3.4.6. meta-data

Mandatory.

Returns the resource agent meta data via stdout.

3.4.7. validate-all

Optional.

Validate the instance parameters provided.

Perform a syntax check and if possible, a semantic check on the
instance parameters.

3.5. Parameter passing

The instance parameters and some additional attributes are passed in via the environment; this has been chosen because it does not reveal the parameters to an unprivileged user on the same system and environment variables can be easily accessed by all programming languages and shell scripts.

3.5.1. Syntax for instance parameters

They are directly converted to environment variables; the name is prefixed with "OCF_RESKEY_".

The instance parameter "force" with the value "yes" thus becomes:
OCF_RESKEY_force=yes
in the environment.

See section 4. for a more formal explanation of instance parameters.

3.5.2. Global OCF attributes

The entire environment variable name space starting with OCF_ is considered to be reserved.

Currently, the following additional parameters are defined:

OCF_RA_VERSION_MAJOR
OCF_RA_VERSION_MINOR
Version number of the OCF Resource Agent API. If the script does not support this revision, it should report an error.

See 3.1. for an explanation of the versioning scheme used. The version number is split into two numbers for ease of use in shell scripts.

These two may be used by the RA to determine whether it is run under an OCF compliant RM.

Example: OCF_RA_VERSION_MAJOR=1
 OCF_RA_VERSION_MINOR=0

OCF_ROOT
Referring to the root of the OCF directory hierarchy.

Example: OCF_ROOT=/usr/ocf

OCF_RESOURCE_INSTANCE
The name of the resource instance.

OCF_RESOURCE_TYPE
The name of the resource type being operated on.

3.5.3. Action specific extensions

These environment variables are not required for all actions, but only supported by some.

3.5.3.1. Parameters specific to the 'status' action

```
OCF_CHECK_LEVEL
0 The most lightweight check possible, which should not
have an impact on the QoS.
Example: Check for the existence of the process.
10 A medium weight check, expected to be called multiple
times per minute, which should not have a noticeable
impact on the QoS.
Example: Send a request for a static page to a
webserver.
20 A heavy weight check, called infrequently, which may
impact system or service performance.
Example: An internal consistency check to verify service
integrity.

Service must remain available during all of these operation.
All other number are reserved.

It is recommended that if a requested level is not implemented,
the RA should perform the next lower level supported.


3.6. Exit status codes

These exit status codes are the ones documented in the LSB 1.1.0
specification, with additional explanations of how they shall be used by
RAs.

In general, all exit status codes except "0" shall indicate failure in
accordance to the best current practices.

3.6.1. All operations but "status"

0 No error, action succeeded completely
1  generic or unspecified error (current practice)
2  invalid or excess argument(s)
Likely error code for validate-all, if the instance parameters
do not validate. Any other action is free to also return this
exit status code for this case.
```

```
3  unimplemented feature (for example, "reload")
4  user had insufficient privilege
5  program is not installed
6  program is not configured
7  program is not running
Note: This is not the error code to be returned by a successful
"stop" operation. A successful "stop" operation shall return 0.
8-99  reserved for future LSB use
100-149 reserved for distribution use
150-199 reserved for application use
200-254 reserved
```

3.6.2. Exit status codes for "status"

That the LSB has chosen to deviate from the set of default exit status codes for the "status" operation is unfortunate, but this specification follows this path to stay close to the LSB specification.

However, a minor deviation from the values given in the LSB specification was made for the values "1" and "2". The existance of a pid file or lock file does not convey any meaning for a RA; these values have been redefined, but still imply errors.

```
0 program is running or service is OK
1 program is dead, hung or crashed. (Generic error)
2  invalid or excess argument(s)
3 program is not running
Note: This exit code shall be returned also for cleanly
inactive resource instances.
4 program or service status is unknown
5-99 reserved for future LSB use
100-149 reserved for distribution use
150-199 reserved for application use
200-254 reserved
```

4. Relation to the LSB

It is required that the current LSB spec is fully supported by the system.

The API tries to make it possible to have RA function both as a normal LSB
init script and a cluster-aware RA, but this is not required functionality.
The RAs could however use the helper functions defined for LSB init scripts.

5. RA meta data

5.1. Format

We have the following requirements which are not fulfilled by the LSB way of
embedding meta data into the beginning of the init scripts:

- Independent of the language the RA is actually written in,
- Extensible,
- Structured,
- Easy to parse from a variety of languages.

This is why we use simple XML to describe the RA meta data. The DTD for this
API can be found at http://www.opencf.org/standards/ra-api-1.dtd.

5.2. Semantics

An example of a vs fully supported by the system.

The API tries to make it possible to have RA function both as a normal LSB
init script and a cluster-aware RA, but this is not required functionality.
The RAs could however use the helper functions defined for LSB init scripts.

5. RA meta data

5.1. Format

We have the following requirements which are not fulfilled by the LSB way of
embedding meta data into the beginning of the init scripts:

```
- Independent of the language the RA is actually written in,
- Extensible,
- Structured,
- Easy to parse from a variety of languages.
This is why we use simple XML to describe the RA meta data.

C. To-do list

Move the terminology definitions out into a separate document common to all
OCF work.

An interface where the RA asynchronously informs the RM of failures is planned
but not defined yet.

D. Contributors

James Bottomley <James.Bottomley@steeleye.com>
Greg Freemyer <freemyer@NorcrossGroup.com>
Simon Horms <horms@verge.net.au>
Ragnar Kjørstad <linux-ha@ragnark.vestdata.no>
Lars Marowsky-Brée <lmb@suse.de>
Alan Robertson <alanr@unix.sh>
  Yixiong Zou <yixiong.zou@intel.com>

E. Change Log

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT
```

Resource Agent Metadata Definition (DTD)

```
<?xml version="1.0" ?>

<!ELEMENT resource-agent (version,parameters,actions,special) >
<!ATTLIST resource-agent
name CDATA #REQUIRED
version CDATA   #IMPLIED>

<!ELEMENT version (#PCDATA)>
```

```
<!ELEMENT parameters (parameter*)>

<!ELEMENT actions (action*)>

<!ELEMENT parameter (longdesc+,shortdesc+,content)>
<!ATTLIST parameter
name CDATA #REQUIRED
unique (1|0) "0">

<!ELEMENT longdesc ANY>
<!ATTLIST longdesc
lang NMTOKEN #IMPLIED>

<!ELEMENT shortdesc ANY>
<!ATTLIST shortdesc
lang NMTOKEN #IMPLIED>

<!ELEMENT content EMPTY>
<!ATTLIST content
type (string|integer|boolean) #REQUIRED
default CDATA #IMPLIED>

<!ELEMENT action EMPTY>
<!ATTLIST action
name (start|stop|recover|status|reload|meta-data|verify-all) #REQUIRED
timeout CDATA #REQUIRED
interval  CDATA #IMPLIED
start-delay  CDATA #IMPLIED
depth CDATA #IMPLIED>

<!ELEMENT special ANY>
<!ATTLIST special
tag CDATA #REQUIRED>
```

Resource Agent Metadata Example (XML)

```
<?xml version="1.0"?>
```

```
<!DOCTYPE resource-agent SYSTEM "ra-api-1.dtd">

<!-- Root element: give the name of the Resource agent -->
<resource-agent name="Filesystem" version="FailSafe 1.0.4">

<!-- Version number of the standard this complies with -->
<version>1.0</version>

<!-- List all the instance parameters the RA supports or requires. -->
<parameters>

<!-- Note that parameters flagged with 'unique' must be unique; ie no
     other resource instance of this resource type may have the same set
     of unique parameters.
 -->

<parameter name="Mountpoint" unique="1">
<!-- This is the long, helpful description of what the parameter is all
     about. A user interface might display it to the user if he asks for
     elaborate help with an option, and it would obviously also provide
     examples etc.

     You can have multiple ones with different "lang" attributes, but this is
     not required.
-->
<longdesc lang="en">
The resource name is the directory where the filesystem will be actually
mounted.  Please make sure it exists.
</longdesc>

<!-- The shortdesc may be displayed by the resource manager as a tooltip or
     equivalent -->
<shortdesc lang="en">Mountpoint</shortdesc>

<!-- Further definition of the content -->
<content type="string" default="/mnt" />

</parameter>
```

```
<parameter name="Device" unique="1">
<longdesc lang="en">
When mounting a filesystem on a specific mountpoint, you have to specify which
device should be mounted; this will usually be similiar to /dev/sda1 or
/dev/volumegroup/logicalvolume when using LVM.
</longdesc>
<shortdesc lang="en">Device to be mounted</shortdesc>
<content type="string" default="/dev/"/>
</parameter>

<parameter name="FSType">
<longdesc lang="en">
You should chose a journaled filesystem for the shared storage to ensure that
the filesystem remains consistent and that it can be mounted without an
expensive fsck run; recommendations include reiserfs, ext3 and XFS.
</longdesc>
<shortdesc lang="en">Type of the filesystem</shortdesc>
<content type="string" default="reiserfs"/>
</parameter>

<parameter name="mount_options">
<longdesc lang="en">
The mount options used for mounting a filesystem; normally this is set to
defaults, but you may want to modify this if you require a read-only
mount or something similar.
</longdesc>
<shortdesc lang="en">Mount options for this filesystem</shortdesc>
<content type="string" default="defaults" />
</parameter>

</parameters>

<!-- List the actions supported by the RA -->
<actions>
<!-- Valid actions: start, stop, recover, status, reload, verify-all

     The timeout is given in seconds (or s, m, h, d postfix and their
```

```
     usual meanings) and should be a reasonable _hint_ to the RM how
     long a certain action might take in the worst case.
  -->
<action name="start"   timeout="2m30s" />
<action name="stop"    timeout="100" />
<action name="recover" timeout="150" />
<!-- In what intervals the RM should poll the RA for status; and how early
     after the start of the RA is should start. Again, this are just
     _defaults_ and can be overridden by the RM. -->
<action name="status" depth="0"  timeout="20" interval="10" start-delay="1m" />
<action name="status" depth="10" timeout="60" interval="1h" start-delay="5m" />
<action name="status" depth="20" timeout="2m" interval="1d" start-delay="1d" />
<action name="reload"  timeout="60" />
<action name="meta-data"  timeout="5" />
<!-- As with all not mandatory actions, this one is only listed if supported
  -->
<action name="verify-all"  timeout="30" />

</actions>

<!-- Vendor specific attributes; as the content _inside_ the special tag is
     obviously not covered by the RA-API DTD, it will not validate, but oh
     well...
  -->
<special tag="FailSafe">
<Ordering>403</Ordering>
</special>

</resource-agent>
```

Glossário

Esta seção contém as siglas usadas neste texto, que geralmente aparecem na língua inglesa. O propósito é fornecer um guia de referência rápida para os termos relevantes desta área.

ACID Atomicity, Consistency, Isolation, Durability

API Application Program Interface

BA Basic Availability

CA Continuous Availability

CMIP Common Management Information Protocol

CS Cold Stand-By

DLM Distributed Lock Manager

FIFO First In First Out

FSG Free Standards Group

GMS Group Membership Service

HA High Availability

HA High Availability

HPC High Performance Computing

HS Hot Stand-By

IP Internet Protocol

LR Lazy Replication

MIB Management Information Base

MM Manual Masking

MPI Message Passing Interface

MTBF Mean Time Between Failures

MTTR Maximum Time To Repair

OCF Open Clustering Framework

POSIX Portable Operating System Interface based on uniX

RA Replicação Ativa

RR Rollback & Recover

SGD Serviço de Gerenciamento de Disponibilidade

SNMP Simple Network Management Protocol

SPOF Single Point of Failure

STONITH Shoot The Other Node In The Head

TFCC IEEE Task Force on Cluster Computing

WS Warm Stand-By

Referências Bibliográficas

[1] *Biblioteca digital de teses e dissertações*, `http://www.teses.usp.br`, Universidade de São Paulo. Citado na(s) página(s) 3, 151

[2] *Free Standards Group*,
`http://www.freestandards.org/`. Citado na(s) página(s) 118

[3] *IEEE Computer Society Task Force on Cluster Computing*,
`http://www.ieeetfcc.org/`. Citado na(s) página(s) 119

[4] *Linux Standard Base*,
`http://www.linuxbase.org/`. Citado na(s) página(s) 116

[5] *Mission Critical Linux - Kimberlite*,
`http://oss.missioncriticallinux.com/projects/kimberlite/`. Citado na(s) página(s) 123

[6] *MPI Forum*,
`http://www.mpi-forum.org/`. Citado na(s) página(s) 119

[7] *Open Clustering Framework Mailing List*,
`http://lists.community.tummy.com/mailman/listinfo/ocf/`. Citado na(s) página(s) 105, 110, 189

[8] *Service Availability Forum*,
`http://www.saforum.org/`. Citado na(s) página(s) 119

[9] *The Ensemble Distributed Communication System*,
`http://www.cs.cornell.edu/Info/Projects/Ensemble/`. Citado na(s) página(s) 121

[10] *The Horus Project*,
`http://www.cs.cornell.edu/Info/Projects/HORUS/`. Citado na(s) página(s) 120

[11] *The Isis Project*,
`http://www.cs.cornell.edu/Info/Projects/ISIS/`. Citado na(s) página(s) 120

[12] *The Totem System*,
`http://beta.ece.ucsb.edu/totem.html`. Citado na(s) página(s) 121

[13] *The Transis Group Communication System*,
`http://www.cs.huji.ac.il/labs/transis/`. Citado na(s) página(s) 121

[14] Guillermo A. Alvarez e Flaviu Cristian, Simulation-based testing of communication protocols for dependable embedded systems, *The Journal of Supercomputing* **16** (2000), no. 1–2, 93–116. Citado na(s) página(s) 144, 145

[15] Y. Amir, D. Dolev, S. Kramer e D. Malki, Transis: A communication subsystem for high availability, *FTCS-22: 22nd International Symposium on Fault Tolerant Computing* (Boston, Massachusetts), IEEE Computer Society Press, July 1992, pp. 76–84. Citado na(s) página(s) 104

[16] Yair Amir, Danny Dolev, Shlomo Kramer e Dalia Malki, Membership algorithms for multicast communication groups, *Distributed Algorithms, 6th International Workshop, WDAG '92* (Haifa, Israel) (Adrian Segall e Shmuel Zaks, eds.), Lecture Notes in Computer Science, vol. 647, Springer, November 1992, pp. 292–312. Citado na(s) página(s) 104

[17] Yair Amir, Louise E. Moser, P. M. Melliar-Smith, Deborah A. Agarwal e P. Ciarfella, Fast message ordering and membership using a logical token-passing ring, *International Conference on Distributed Computing Systems*, 1993, pp. 551–560. Citado na(s) página(s) 121

[18] Emmanuelle Anceaume, Bernadette Charron-Bost, Pascale Minet e Sam Toueg, *On the formal specification of group membership services*, Tech. Report TR95-1534, 25 1995. Citado na(s) página(s) 103, 104

[19] T. Anderson e P. A. Lee, *Fault tolerance principles and practice*, Prentice Hall, 1981. Citado na(s) página(s) 27, 30, 34

[20] A. Avizienis, Design of fault-tolerant computers, *Proceedings of Fall Joint Computer Conf.*, vol. 31, Thompson Books, 1967, pp. 733–743. Citado na(s) página(s) 16

[21] ———, The four-universe information system model for the study of fault tolerance, *In Proceedings of the Twelve Annual Symposium on Fault Tolerant Computing*, 1982, pp. 6–13. Citado na(s) página(s) 28

[22] A. Avizienis, J. Laprie e B. Randell, *Fundamental concepts of dependability*, Research Report N01145, LASS-CNRS, April 2001. Citado na(s) página(s) 19

[23] Özalp Babaoglu, Renzo Davoli, Luigi-Alberto Giachini e Mary Gray Baker, *RELACS: A communications infrastructure for constructing reliable applications in large-scale distributed systems*, Technical Report TR94-58, ESPRIT Basic Research Project BROADCAST, October 1994. Citado na(s) página(s) 104

[24] Amnon Barak, *The MOSIX Project*, `http://www.mosix.org/`. Citado na(s) página(s) 38

[25] Amnon Barak e Oren La'adan, The MOSIX multicomputer operating system for high performance cluster computing, *Future Generation Computer Systems* **13** (1998), no. 4–5, 361–372. Citado na(s) página(s) 38

[26] K. Birman, *ISIS: A system for fault-tolerance in distributed systems*, Tech. Report TR 86-744, Cornell University Computer Science Department, Ithaca, NY, April 1986. Citado na(s) página(s) 90, 97, 120

[27] ———, Building secure and reliable network applications, *WWCA*, 1997, pp. 15–28. Citado na(s) página(s) 76, 97, 99, 107

[28] K. Birman, R. Constable, M. Hayden, C. Kreitz, O. Rodeh, R. van Renesse e W. Vogels, The Horus and Ensemble projects: Accomplishments and limitations, *Proceedings of the DARPA Information Survivability Conference & Exposition (DISCEX '00)* (Hilton Head, South Carolina USA), January 2000 (English). Citado na(s) página(s) 120, 121

[29] K. Birman e R. van Renesse, *Reliable distributed computing with the ISIS toolkit*, IEEE Press, 1994 (English). Citado na(s) página(s) 120

[30] Kenneth P. Birman, The process group approach to reliable distributed computing, *Communications of the ACM (CACM)* **36** (1993), no. 12, 37–53,103, TR No. TR91-1216. Citado na(s) página(s) 79

[31] D. Bitton e J. Gray, Disk shadowing, *Proceedings of the 14th VLDB Conference* (Los Angeles, California), Semptember 1988, pp. 331–338. Citado na(s) página(s) 53

[32] James Bottomley, *Resource Driven Clusters*, OCF Workshop, January 2003. Citado na(s) página(s) 80, 83, 84

[33] R. Carr, The split brain syndrome, *Private Communication*, 1990. Citado na(s) página(s) 12

[34] Tushar Deepak Chandra, Vassos Hadzilacos, Sam Toueg e Bernadette Charron-Bost, On the Impossibility of Group Membership, *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC'96)* (New York, USA), ACM, 1996, pp. 322–330. Citado na(s) página(s) 103, 104

[35] F. Cristian e C. Fetzer, A Fail-Aware Membership Service, *Proceedings of The 16th Symposium on Reliable Distributed Systems (SRDS '97)* (Washington - Brussels - Tokyo), IEEE, October 1997, pp. 157–164. Citado na(s) página(s) 80, 102

[36] ———, The Timed Asynchronous Distributed System Model, *IEEE Transactions on Parallel and Distributed Systems* **10** (1999), no. 6, 642–?? Citado na(s) página(s) 102

[37] Flaviu Cristian, Reaching Agreement on Processor-Group Membership in Synchronous Distributed Systems, *Distributed Computing* **4** (1991), no. 4, 175–188. Citado na(s) página(s) 77, 78, 79, 84, 85, 101, 104, 127

[38] ———, Understanding fault-tolerant distributed systems, *Communications of the ACM* **34** (1991), no. 2, 56–78. Citado na(s) página(s) 15, 65, 67, 68, 76, 102

[39] ———, Automatic reconfiguration in the presence of failures, *Proceedings of the International Workshop on Configurable Distributed Systems* (London), IEE, March 1992, pp. 4–17. Citado na(s) página(s) 45, 65, 66, 70, 101

[40] Flaviu Cristian, Houtan Aghili e Ray Strong, Clock synchronization in the presence of omission and performance failures, and processor joins, *Global States and Time in Distributed Systems, IEEE Computer Society Press* (Zhonghua Yang e T. Anthony Marsland, eds.), 1994. Citado na(s) página(s) 40

[41] Flaviu Cristian e Cristof Fetzer, The timed asynchronous distributed system model, *Proceedings of the 28th IEEE Symposium on Fault Tolerant Computing Systems (FTCS-28)*, June 1998, pp. 140–149. Citado na(s) página(s) 43, 44

[42] Flaviu Cristian e S. Mishra, Automatic service availability management in asynchronous distributed systems, *In Proceedings of the Second International Workshop on Configurable Distributed Systems* (Pittsburgh, PA), March 1994. Citado na(s) página(s) 46, 65, 70, 71

[43] Flaviu Cristian e F. Schmuck, *Agreeing on processor group membership in timed asynchronous distributed systems*, Tech. Report CSE95-428, UCSD, 1995. Citado na(s) página(s) 85, 126, 133, 138

[44] Jeff Dike, User-mode linux, *Proceedings of the 8th International Linux Kongress* (Enschede, The Netherlands) (GUUE German UNIX User Group, ed.), vol. 1, November 2001. Citado na(s) página(s) 145

[45] Joe DiMartino, *Cluster Event Notification*, OCF Workshop, January 2003. Citado na(s) página(s) 117

[46] D. Dolev, D. Malki e R. Strong, *An asynchronous membership protocol that tolerates partitions*, Research report, The Hebrew University of Jerusalem, 1993. Citado na(s) página(s) 82, 103, 104

[47] Danny Dolev e Dalia Malki, The Transis approach to high availability cluster communication, *Communications of the ACM* **39** (1996), no. 4, 64–70. Citado na(s) página(s) 121, 122

[48] Danny Dolev, Dalia Malki e Ray Strong, A framework for partitionable membership service, *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, ACM Press, 1996, p. 343. Citado na(s) página(s) 103

[49] P. D. Ezhilchelvan, R. A. Macêdo e S. K. Shrivastava, Newtop: A fault-tolerant group communication protocol, *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS'95)* (Los Alamitos, CA, USA), IEEE Computer Society Press, May 30–June 2 1995, pp. 296–306. Citado na(s) página(s) 104

[50] Paul D. Ezhilchelvan, *Building responsive and reliable distributed services: Models and design options*. Citado na(s) página(s) 41

[51] C. Fetzer, A comparison of timed asynchronous systems and asynchronous systems with failure detectors, *Proceedings of the Third European Research Seminar on Advances in Distributed Systems (ERSADS'99)* (Madeira Island, Portugal), 1999, pp. 109–118 (English). Citado na(s) página(s) 102

[52] C. Fetzer e F. Cristian, Derivation of fail-aware membership service specifications, *Lecture Notes in Computer Science* **1388** (1998), 644–?? Citado na(s) página(s) 104

[53] Christof Fetzer e Flaviu Cristian, Fail-awareness in timed asynchronous systems, *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC'96)* (Philadelphia), 1996, pp. 314–321a. Citado na(s) página(s) 44, 104

[54] Michael J. Fischer, Nancy A. Lynch e Michael S. Paterson, Impossibility of distributed consensus with one faulty process, *Journal of the ACM (JACM)* **32** (1985), no. 2, 374–382. Citado na(s) página(s) 43, 103

[55] Felix C. Gartner, Fundamentals of fault-tolerant distributed computing in asynchronous environments, *ACM Computing Surveys* **31** (1999), no. 1, 1–26. Citado na(s) página(s) 16

[56] Joe Greenseid, *Linux Cluster Information Center*, `http://www.lcic.org/`. Citado na(s) página(s) 106

[57] M. Hiltunen e R. Schlichting, *Understanding membership*, Tech. report, Dept. of Computer Science, University of Arizona, Tucson, AZ, 1994, In preparation. Citado na(s) página(s) 86, 97, 104, 120, 121

[58] Farnam Jahanian, Sameh Fakhouri e Ragunathan Rajkumar, Processor Group Membership Protocols: Specification, Design and Implementation, *Proceedings of the 12th Symposium on Reliable Distributed Systems* (Princeton, New Jersey), IEEE, October 1993, pp. 2–11. Citado na(s) página(s) 104

[59] Pankaj Jalote, *Fault tolerance in distributed systems*, Prentice Hall, 1994. Citado na(s) página(s) 16, 20, 29, 38, 47, 48, 53, 56

[60] I. E. Jansch-Porto e T. S. Weber, Recuperação em sistemas distribuídos, *XVI Jornada de Atualização em Informática, XVII Congresso da SBC* (Brasília, BR), SBC, Agosto 1997, anais, pp. 263–310. Citado na(s) página(s) 33

[61] M. F. Kaashoek e A. S. Tanenbaum, Group communication in the Amoeba distributed operating system, *Proceedings of the* 11^{th} *International Conference on Distributed Computing Systems ICDCS* (Washington, D.C., USA), IEEE Computer Society Press, May 1991, pp. 222–230 (English). Citado na(s) página(s) 104

[62] Leslie Lamport e P. M. Melliar-Smith, Synchronizing clocks in the presence of faults, *Journal of the ACM (JACM)* **32** (1985), no. 1, 52–78. Citado na(s) página(s) 49

[63] Leslie Lamport, Robert Shostak e Marshall Pease, The byzantine generals problem, *ACM Transactions on Programming Languages and Systems (TOPLAS)* **4** (1982), no. 3, 382–401. Citado na(s) página(s) 47

[64] B. W. Lampson, Atomic transactions, *Lecture Notes in Computer Science* **105** (1981), 246–265. Citado na(s) página(s) 51, 52

[65] Jean-Claude Laprie, Dependable computing and fault tolerance: Concepts and terminology, *15th International Symposium on Fault Tolerant Computing Systems* (Ann Arbor, Michigan), June 1985, pp. 2–11. Citado na(s) página(s) 27, 29

[66] ———, Dependability of computer systems: from concepts to limits, *In Proc. of IFIP Intern. Workshop on Dependable Computing and Its Applications (DCIA98)* (Johannesburg, South Africa), January 1998. Citado na(s) página(s) 19

[67] Kal Lin, *Asynchronos membership and communication*, November 1999, Thesis Proposal. Citado na(s) página(s) 79, 80, 81, 82

[68] E. Marcus e H. Stern, *Blueprints for high availability - designing resilient distributed systems*, John Wiley & Sons, 2000. Citado na(s) página(s) 17, 18

[69] Lars Marowsky-Brée, *The Open Clustering Framework*. Citado na(s) página(s) 105, 106, 108, 110, 115

[70] P. M. Melliar-Smith, Louise E. Moser e Vivek Agrawala, Broadcast protocols for distributed systems, *IEEE Transactions on Parallel and Distributed Systems* **1** (1990), no. 1, 17–25. Citado na(s) página(s) 64

[71] S. Mishra, L. Peterson e R. Schlichting, A membership protocol based on partial order, *Dependable Computing for Critical Applications 2* (J. Meyer e R. Schlichting, eds.), Dependable Computing and Fault-Tolerant Systems, Springer-Verlag, Vienna, Austria, 1992, Proc. IFIP 10.4 Work. Conf. held in Tucson, AZ, USA, February 1991, pp. 309–331. Citado na(s) página(s) 104

[72] Louise E. Moser, P. M. Melliar-Smith e Vivek Agrawala, Processor membership in asynchronous distributed systems, *IEEE Transactions on Parallel and Distributed Systems* **5** (1994), no. 5, 459–473. Citado na(s) página(s) 104

[73] D. A. Patterson, G. Gibson e R. H. Katz, A case for redundant arrays of inexpensive disks (RAID), *Proceedings of ACM SIGMOD* (Chicago, IL), June 1988, pp. 109–116. Citado na(s) página(s) 53, 54

[74] Nelio Pereira, *Resultados dos experimentos do serviço de pertinência*, `http://www.ime.usp.br/dcc/posgrad/teses/nelio/`, August 2004. Citado na(s) página(s) 3, 151

[75] Gregory F. Pfister, *In search of clusters*, Prentice Hall PTR, 1998. Citado na(s) página(s) 15, 38, 72, 107

[76] Robbert Van Renesse, Kenneth Birman, Bradford B. Glade, Katie Guo, Mark Hayden, Takako Hickey, Dalia Malki, Alex Vaysburd e Werner Vogels, *Horus: A flexible group communications system*, Tech. Report TR95-1500, 23, 1995. Citado na(s) página(s) 104

[77] Ron I. Resnick, *A modern taxonomy of high availability*, 1996. Citado na(s) página(s) 21, 25

[78] Aleta M. Ricciardi e Kenneth P. Birman, Using process groups to implement failure detection in asynchronous environments, *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing* (Montréal, Québec, Canada) (Luigi Logrippo, ed.), ACM Press, August 1991, pp. 341–351. Citado na(s) página(s) 103, 104

[79] ———, *Process membership in asynchronous environments*, Technical Report TR93-1328, Cornell University, Computer Science Department, February 1993. Citado na(s) página(s) 103

[80] Aleta Marie Ricciardi, *The Group Membership Problem in Asynchronous Systems*, Ph.D. thesis, cornell, Ithaca, New York, November 1992, No. 92-1313. Citado na(s) página(s) 103

[81] A. L. Robertson, *Linux-HA Heartbeat System Design*. Citado na(s) página(s) 122

[82] _____, *PILS: A Generalized Plugin and Interface Loading System*. Citado na(s) página(s) 117

[83] _____, *Resource fencing using stonith*. Citado na(s) página(s) 71

[84] _____, *The High-Availability Linux Project*, 2001, `http://linux-ha.org`. Citado na(s) página(s) 122

[85] _____, *The Open Cluster Framework (OCF) - A Plan for World Domination in Clustering*, `http://www.opencf.org`, 2002. Citado na(s) página(s) 105, 108, 110, 189

[86] A.L. Scherr, Functional structure of ibm virtual storage operating systems, part ii: Os/vs-2 concepts and philosophies, *IBM Systems Journal* **12** (1973), no. 4, 382–400. Citado na(s) página(s) 37

[87] A. Schiper e A. Sandoz, Understanding the power of the virtually-synchronous model, *Proceedings of the 5th European Workshop on Dependable Computing*, 1993. Citado na(s) página(s) 99

[88] Richard D. Schlichting e Fred B. Schneider, Fail-stop processors: an approach to designing fault-tolerant computing systems, *ACM Transactions on Computer Systems (TOCS)* **1** (1983), no. 3, 222–238. Citado na(s) página(s) 55

[89] Stephen Tweedie, *Designing a linux cluster*, January 2000. Citado na(s) página(s) 10, 13

[90] Robbert van Renesse, Kenneth Birman e Silvano Maffeis, Horus: A flexible group communication system, *CACM* **39** (1996), no. 4, 76–83. Citado na(s) página(s) 120

[91] M. Vasa, *The Linux Failsafe Project*, 2000, `http://oss.sgi.com/projects/failsafe/`. Citado na(s) página(s) 9, 123

[92] W. Vogels, D. Dumitriu, K. Birman, R. Gamache, M. Massa, R. Short, J. Vert, J. Barrera e J. Gray, The design and architecture of the microsoft cluster service - a practical approach to high-availability and scalability, *Proceedings of FTCS*, IEEE, June 1998. Citado na(s) página(s) 5, 7

[93] Jos Vos, *International Linux Kongress*, `http://www.linux-kongress.org/`. Citado na(s) página(s) 110

[94] Taisy Silva Weber, *Um roteiro para exploração dos conceitos básicos de tolerância a falhas*, 2002. Citado na(s) página(s) 16, 19, 20, 21, 28, 33, 34

Índice Remissivo