

**UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE CIÊNCIAS DA COMPUTAÇÃO**

Diogo Miguel Martendal

**Replicação de Serviço em Ambiente de
Grade Computacional Autônoma**

Florianópolis–SC
Dezembro/2008

Diogo Miguel Martendal

Replicação de Serviço em Ambiente de Grade Computacional Autônoma

Dissertação apresentada junto ao Curso de
Bacharelado em Ciências da Computação da
Universidade Federal de Santa Catarina
como parte dos requisitos para a obtenção do
grau de Bacharel em Ciências da Computação

Orientador: Prof. Dr. Carlos Becker Westphall
Co-Orientador: Douglas de Oliveira Balen

UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE CIÊNCIAS DA COMPUTAÇÃO

Florianópolis–SC
Dezembro/2008

Diogo Miguel Martendal

Replicação de Serviço em Ambiente de Grade Computacional Autônoma

Esta Dissertação foi julgada adequada para a obtenção do título de Bacharel em Ciências da Computação Área de Concentração Sistemas de Computação e aprovada em sua forma final pelo Curso de Graduação em Ciências da Computação

Orientador: Prof. Dr. Carlos Becker Westphall
Co-Orientador: Douglas de Oliveira Balen

COMISSÃO EXAMINADORA

Prof. Dr. Carlos Becker Westphall
Universidade Federal de Santa Catarina

Douglas de Oliveira Balen
Universidade Federal de Santa Catarina

Prof.^a Dra. Carla Merkle Westphall
Universidade Federal de Santa Catarina

Florianópolis, 02 de dezembro de 2008

*Dedico este trabalho aos catarinenses que sofrem
com a tragédia deste final de ano.*

Aos que perderam tudo ou até mesmo suas vidas.

*Dedico não menos ao povo brasileiro, que se
prontificou imediatamente a ajudar de várias
maneiras os atingidos.*

*À minha família, namorada, amigos, professores,
colegas e todos que contribuíram para que eu
chegasse até aqui.*

O único lugar onde “sucesso” vem antes do “trabalho” é no dicionário.

Albert Einstein

Agradecimentos

Agradeço imensamente aos meus orientadores pela oportunidade neste projeto, por toda atenção dedicada, compreensão e paciência.

Agradeço a todos que contribuíram para este trabalho, aos que me deram ânimo e torceram por mim. Com certeza isso me ajudou muito.

Agradeço a Deus pela força que me deu para desenvolver este trabalho e por ter feito com que meus objetivos sempre tenham sido cumpridos.

Resumo

Este trabalho tem por objetivo definir e implementar um processo de replicação de serviços autônomo em um ambiente de grade computacional. O ambiente de grade computacional é largamente utilizado como uma forma barata de aumentar o poder computacional unindo-se equipamentos em uma rede e compartilhando o processamento e os recursos como se a rede fosse um único grande computador. Quando o objetivo é poder computacional e um ambiente confiável é importante que o mesmo consiga se gerenciar automaticamente visando um ambiente o mais otimizado possível. A replicação de serviços proporciona uma forma de transmissão de implementação de rotinas entre os dispositivos, permitindo uma melhor distribuição do processamento e um controle contra gargalos de processamento. Os objetivos foram alcançados, a melhoria foi constatada e os resultados estimulam ainda mais o estudo para evolução de um ambiente mais autônomo.

Palavras-chave: grade computacional, computação autônoma, replicação de serviço

Abstract

The goal of this dissertation is to define and implement an autonomous service replication process in grid environment. The grid computing environment is widely applied as a cheap mode to enhance computing power by grouping devices in a net and sharing processing and resources as if the net was a single great computer. When the purpose is computing power and a reliable environment it's important that it could self manage aiming at optimization. The service replication provides routine implementation transporting among devices, which allows better processing distribution and control against processing overhead. The targets have been reached, the improvement has been encountered and the results inspire even more studies on evolution at autonomous environment.

Keywords: grid computing, autonomous computing, service replication

Sumário

<i>Agradecimentos</i>	5
<i>Resumo</i>	6
<i>Abstract</i>	7
<i>Sumário</i>	8
<i>Lista de Figuras</i>	10
1 Introdução	11
1.1 Objetivo	11
1.2 Estrutura do Trabalho	12
2 Computação Autônoma	13
2.1 Auto-gerenciamento (Self-management)	14
2.1.1 Auto-configuração (self-configuration).....	14
2.1.2 Auto-regeneração (self-healing)	15
2.1.3 Auto-otimização (self-optimization)	15
2.1.4 Auto-proteção (self-protection).....	16
2.2 Níveis de Automação	16
2.3 Elemento Autônomo	16
3 Grades Computacionais	17
3.1 Funcionamento	19
3.2 Tipos de Grades.....	20
3.3 Grades Orientadas a Serviços.....	21
4 Grades Computacionais Autônomas.....	23
4.1 Gerente Autônomo	24
5 Replicação de Serviços	26

5.1 Roteamento de Serviços	28
5.1.1 Definições e Métricas.....	28
5.1.2 Algoritmo de Roteamento.....	30
5.1.3 Conexão entre nós	32
5.2 Ambiente de Testes.....	32
5.2.1 Middleware em Ambientes Distribuídos	32
5.2.2 Grid-M	33
6 Resultados	35
7 Conclusão	37
7.1 Trabalhos Futuros.....	37
<i>Referências Bibliográficas</i>	38
APÊNDICE A	42
APÊNDICE B	45
APÊNDICE C	47
APÊNDICE D	50
APÊNDICE E	52

Lista de Figuras

Fig. 3.1 União de domínios.....	18
Fig. 3.2 União de dispositivos e recursos heterogêneos.....	18
Fig. 3.3 <i>Grid middleware</i>	19
Fig. 3.4 Infra-estrutura orientada por serviços (ROURE; JENNIGS; SHADBOLT, 2003)	22
Fig. 4.1 Elemento autônomo.....	23
Fig. 4.2 Grade computacional de elementos autônomos.....	24
Fig. 4.3 Funcionalidades do gerente autônomo.....	25
Fig. 5.1 Diagrama de grade com replicação de serviço.....	27
Fig. 5.2 Fluxograma do roteamento de serviços (1).....	31
Fig. 5.3 Fluxograma do roteamento de serviços (2).....	31
Fig. 6.1 Utilização de recursos dos nós e replicação de serviço.....	36

1 Introdução

A computação distribuída surgiu da necessidade por integrar informações e aplicações em redes para que se tenha acesso a estas em qualquer ponto, assim como descrito em (WEISER, 1991) onde o autor cria o termo computação ublúqua que significa “em todo lugar”. Isto contribui com a crescente complexidade das redes e a necessidade de formas de gerência destes recursos.

O conceito de grade computacional (FOSTER, 2000), amplamente conhecido como *grid*, vem definir uma idéia para tornar homogênea uma rede com dispositivos, sistemas operacionais, informações e serviços heterogêneos, aonde todos os componentes da rede venham a ser igualmente tratados e onde tudo é um nó, virtualizando estes recursos e tornando o ambiente mais facilmente gerenciável e deixando transparente como e por quais recursos as tarefas são executadas na grade.

No passado a dependência da interação humana em gerência de redes de computadores era viável e suficiente. A complexidade alcançada por elas atualmente exige uma maior autonomia, diminuindo a interação humana em tarefas de baixo-nível e aproveitando a experiência destes profissionais em definições de gerência em alto nível, definindo políticas de rede que possam ser implementadas nos sistemas distribuídos e adotadas pela rede. Este contexto estimulou a pesquisa e aplicação de computação autônoma (HORN, 2001) nas redes de computadores, introduzindo propriedades de auto-gerenciamento (HINCHEY; STERRITT, 2006).

1.1 Objetivo

Este trabalho visa definir e construir uma proposta para a funcionalidade de replicação de serviços para um projeto de auto-gerenciamento em grades computacionais, uma arquitetura unindo os conceitos de grade e computação autônoma (HARIRI et al., 2006). São conceitos muito atuais envolvidos no ambiente de pesquisa definido o quê torna a proposta muito útil e interessante para o momento pelo qual passamos.

O objetivo é incrementar a autonomia de uma grade computacional impedindo que se criem gargalos de processamento em um recurso ou em um pequeno número de recursos, por

este(s) possuir(írem) serviços frequentemente utilizados pela grade, distribuindo este processamento e melhorando a utilização dos recursos.

A aplicação que servirá de base para a implementação da solução é o *middleware* Grid-M, uma biblioteca Java que fornece todos os recursos para a criação de um ambiente de grade, desenvolvido no “Laboratório de Redes e Gerência” (LRG) da Universidade Federal de Santa Catarina. Dentro do Grid-M ainda contará com outras funcionalidades desenvolvidas no projeto maior ao qual este faz parte, tais como o monitoramento de utilização do hardware e a gerência de serviços, necessárias para este trabalho, e este também agregará indiretamente com uma funcionalidade de redirecionamento de serviços.

1.2 Estrutura do Trabalho

Nos primeiros capítulos do trabalho são abordados os conceitos base do ambiente onde se propõe a funcionalidade: computação autônoma, grades computacionais e a união destes dois conceitos.

Em seguida, no capítulo 5, é desenvolvida a proposta da funcionalidade de replicação de serviços e definições relacionadas, e também é abordado o ambiente de implementação, o *middleware* Grid-M.

A seguir, no capítulo 6, são apresentados os resultados obtidos com os experimentos, e no capítulo 7 as conclusões e trabalhos futuros.

2 Computação Autônoma

A gerência de redes é um processo controlado manualmente que depende de um ou mais operadores fortemente integrados (IBM, 2003). O processo conta com o operador para, além de tarefas de alto-nível, agir em tarefas de baixo-nível sempre que ocorrem problemas iminentes, mas esta dependência é cada vez menos apropriada e gera muitas dificuldades para as organizações, visto a crescente escala destes ambientes e a necessidade de estarem sempre ativos e disponíveis.

Em (HORN, 2001), a pesquisa do Dr. Paul Horn, vice-presidente de pesquisa da IBM, constatou que a necessidade de mão de obra especializada para suportar os ambientes computacionais complexos fazia com que as organizações de TI chegassem a gastar com mão de obra até 70% dos seus custos totais e este percentual continuava subindo. No seu trabalho ele propôs o conceito de sistemas autônomos como uma forma de tratar o crescimento inevitável da complexidade de gerenciamento destes ambientes computacionais e evitar uma crise.

A origem do termo computação autônoma, criado por Paul Horn, vem de uma analogia à autonomia do sistema nervoso humano. O organismo humano percebe as mudanças que ocorrem e age nos sistemas e órgãos de forma que ele possa exercer as atividades, digamos, suportadas por ele ou continuar no seu estado basal onde mantém as funções vitais. Por exemplo, ao iniciar uma atividade física o corpo acelera o coração para bombear mais sangue e nutrir as células, a respiração acelera para oxigenação e a pele começa a suar mais para que o calor em excesso seja liberado. Assim como o sistema nervoso humano, um sistema autônomo deve ser capaz de alterar seus parâmetros de funcionamento para executar tarefas suportadas ou até mesmo para simplesmente se manter em funcionamento, devendo ser capaz de fazer isto mesmo em situações de estresse ou excesso de trabalho.

A capacidade de automação de um sistema é importante para a otimização da utilização dos recursos de uma rede, e isto é ainda mais evidente em um ambiente heterogêneo e dinâmico, assim como é um ambiente de grade computacional.

2.1 Auto-gerenciamento (*Self-management*)

O auto-gerenciamento (*self-management*) (HINCHEY; STERRITT, 2006) é a característica dos sistemas gerenciarem suas próprias operações sem a intervenção humana, sendo esperado que esta característica chegue à próxima geração de sistemas de gerenciamento de redes, já que a heterogeneidade das redes atuais incluindo redes fixas e móveis tornam difícil a gerência manual convencional, além de lenta e inclinada a erros.

Os sistemas que se auto-gerenciam monitoram continuamente sua própria estrutura e uso, e de acordo com mudanças ocorridas controlam suas operações, e, por estarem constantemente monitorando o ambiente, os usuários do mesmo têm ininterruptamente uma rede se gerenciando para evitar e corrigir falhas, e otimizar a sua utilização.

Atualmente a iniciativa industrial mais importante sobre auto-gerenciamento é a “*Autonomic Computing Initiative*” (ACI) iniciada pela IBM em 2001. A ACI define quatro características para um sistema auto-gerenciado, que são abordadas nos tópicos a seguir.

2.1.1 Auto-configuração (*self-configuration*)

O sistema autônomo deve configurar-se dinamicamente sem intervenção externa, de modo otimizado e garantindo seu perfeito funcionamento. Por ser complexo é um trabalho suscetível a erros quando efetuado manualmente, mas quando automatizado tem mais garantias e é mais rápido configurado sempre que necessário também proporcionando um ambiente melhor configurado o tempo todo. Toda configuração deve ser feita com base nas políticas de alto-nível do sistema.

Na utilização desta característica em grades computacionais podemos exemplificar com a auto-conexão de novos nós. Sempre que um novo nó cria uma rota com um nó já existente na grade, este nó existente pode prover uma lista de nós com os quais o novo nó deveria se conectar e este então criar todas as conexões automaticamente. A grade então se adapta a este novo dispositivo e passa a utilizar automaticamente recursos disponibilizados por ele.

2.1.2 Auto-regeneração (self-healing)

O sistema autônomo deve poder se auto-diagnosticar e conseguir se modificar para corrigir ou contornar alguma falha e continuar funcionando. O sistema conseguiria então encontrar problemas ou partes do sistema que não estejam funcionando corretamente e saber como agir de forma reativa a estes problemas previstos. Este controle é muito importante pois espera-se que sistemas possam continuar disponíveis mesmo quando ocorrem panes não generalizadas.

Em uma grade computacional pode-se apontar facilmente a utilidade deste controle visto que os serviços e informações disponibilizados pela rede são de grande importância e para tanto o sistema deveria saber como reagir em caso de indisponibilidade de seus dispositivos ou recursos.

2.1.3 Auto-otimização (self-optimization)

O sistema deve procurar utilizar os recursos disponíveis da melhor forma possível e mesmo em situações de carga de trabalho fora do normal estará apto a obter performance positiva de respostas. Isso é feito dinamicamente através do controle de parâmetros de configuração e políticas de alto-nível, e do balanceamento das operações. Os sistemas complexos existentes no mercado atualmente possuem diversos parâmetros de ajuste para se melhorar sua eficiência, mas poucas pessoas sabem configurá-los, portanto quando isto é feito automaticamente através de estatísticas e dados de utilização do sistema estes garantem muito mais sucesso.

Como funcionalidade em grades computacionais tem-se este próprio trabalho, o qual visa não sobrecarregar o processamento em pequenos grupos de nós devido à baixa disponibilidade de alguns serviços permitindo replicá-los e distribuir o processamento de suas requisições. Isto impede que se criem gargalos, que nós fiquem sobrecarregados e o tempo de resposta de requisições dos serviços aumente.

2.1.4 Auto-proteção (self-protection)

O sistema deve ter formas de garantir a segurança de informações e serviços importantes, proteger-se contra ataques maliciosos ou utilização indevida, não permitir acesso e alteração indevida de parâmetros do sistema, ajudando também a não permitir que ocorram falhas em cascata e o sistema como um todo seja prejudicado, e tornando-o mais seguro e confiável.

Em grades computacionais esta funcionalidade poderia atuar verificando as estatísticas de uso por um nó e definir se este nó pode estar tornando o ambiente crítico fazendo um mau uso dos recursos, mesmo que não propositalmente.

2.2 Níveis de Automação

Devido à complexidade de se implementar todas as características do auto-gerenciamento de uma só vez, a ACI definiu a evolução da automação em 5 níveis, de forma que no primeiro nível tem-se o estado atual do gerenciamento de redes, onde o operador gerencia os recursos manualmente, evoluindo do nível 2 ao 4, e no nível 5 tem-se o sistema totalmente autônomo.

2.3 Elemento Autônomo

Cada unidade básica de um sistema autônomo é um elemento autônomo de acordo com (HERRMANN et al., 2005). Um elemento autônomo é uma estrutura de controle fechado e possui dois componentes:

- Gerente autônomo: que é responsável pelo auto-gerenciamento do elemento. Uma estrutura de tomada de decisões através de estímulos internos e externos.
- Elementos gerenciados: que são os serviços e recursos, hardware ou software, de que este elemento dispõe e que são controlados pelo gerente autônomo.

No capítulo 4 o elemento autônomo e suas funcionalidades serão abordados novamente no contexto de grade computacional.

3 Grades Computacionais

Na história da computação apareceram problemas que exigiram mais poder computacional. Na década de 80 surgiu a idéia de multiprocessamento unindo-se diversos computadores em *clusters* a qual é bem apropriada, sendo utilizados até hoje, e muito mais acessível do que um supercomputador. No final da década de 90 (FOSTER, 2000) sugere outra técnica para multiprocessamento em um ambiente distribuído suportando aplicações em larga escala, a grade computacional. As grades computacionais conseguem unir uma grande quantidade de recursos com um custo muito menor que os *clusters*.

As primeiras iniciativas em grades visavam interligar supercomputadores e centros tecnológicos separados geograficamente (ROURE; JENNIGS; SHADBOLT, 2003), das quais foram precursores os projetos Fafner (FAFNER) e Globus (FOSTER; KESSELMAN, 1997), o segundo muito expressivo e de grande uso ainda hoje por americanos e europeus.

O nome grade computacional vem do termo em inglês *grid computing*, que tem sua origem em uma analogia com a *electrical power grid* (rede de energia elétrica norte-americana). O usuário final de energia elétrica não precisa conhecer como a rede é formada ou como funciona para poder utilizá-la, nem mesmo precisa conhecer quem está fornecendo, apenas se preocupa em conectar seu equipamento através de uma tomada e consumir a energia. Por trás da tomada existe toda estrutura que faz com que a energia seja transportada das usinas onde são geradas até seus clientes. Tal estrutura é gerenciada por órgãos responsáveis desde o controle da ativação das turbinas das usinas para geração de energia até sua transmissão e distribuição, fazendo com que a tensão na rede elétrica seja mantida entre uma determinada faixa de funcionamento ininterruptamente. Assim como na rede elétrica, a grade computacional visa tornar o que acontece na rede transparente ao usuário, assim como seu gerenciamento e sua estrutura.

A maior motivação para os pesquisadores de grades é prover ambientes com o máximo de poder computacional. Por exemplo, é uma forma de unir redes de diferentes domínios (figura 3.1) compartilhando dados, software e equipamentos, além de recursos de máquina e quaisquer outros recursos (figura 3.2), por isto pode-se dizer que a grade é um passo tecnológico seguinte em relação à *internet*, pois visa compartilhar todos estes recursos

além de compartilhar informações. Além disso, as implementações de grades tiram proveito das tecnologias *web*, pois geralmente utilizam seus protocolos e linguagens.



Fig. 3.1 União de domínios

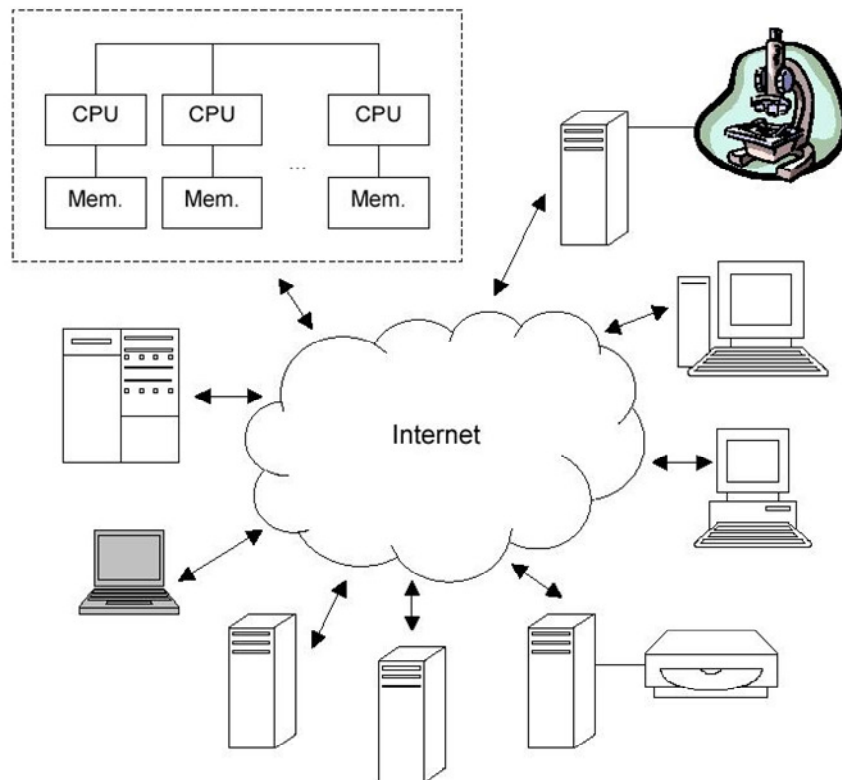


Fig. 3.2 União de dispositivos e recursos heterogêneos

A grade computacional tem o objetivo de suprir a necessidade por integrar o processamento através de uma rede com dispositivos, sistemas e recursos heterogêneos (figura 3.2), fazendo com que o sistema distribuído seja visto por aplicações de usuários como uma única entidade que agrega todos os recursos nele contidos, mesmo que de diferentes organizações (IBM, 2001). Internamente a interface entre os dispositivos faz com que qualquer um deles seja visto como um mesmo tipo de objeto tornando a rede homogênea, onde todos os componentes são igualmente tratados, e mais fácil de gerenciar. Uma grade pode ser vista então como um grafo onde cada componente é um nó e as arestas representam as conexões entre os nós, e por este motivo os dispositivos de uma grade são geralmente conhecidos como nós.

3.1 Funcionamento

As implementações de grades normalmente funcionam através da inclusão de uma nova camada de *software* entre as aplicações e a infra-estrutura física composta pelo computador, sistema operacional e redes. Assim os aplicativos de usuários interagem com os recursos computacionais através desta camada de *software* que esconde a complexidade e diversidade da infra-estrutura oferecendo uma interface uniforme para acesso aos recursos. Esta camada de *software* é chamada de *middleware* (figura 3.3).



Fig. 3.3 Grid middleware

Um *middleware* deve fornecer os métodos de comunicação necessários e se encarregar de localizar e buscar os recursos, assim não é necessário que o usuário conheça detalhes como a localização dos recursos, seu formato, forma de acesso ou protocolos utilizados. Para isto a

grade utiliza a virtualização de recursos, encapsulando seus recursos e funcionalidades em interfaces comuns a todas as implementações.

Três pontos fundamentais são descritos por (FOSTER, 2002) para uma grade computacional:

- Que os recursos de diferentes domínios coordenados pela grade não estejam sujeitos a um controle centralizado.
- Usar protocolos e interfaces com um padrão e propósito geral que atendem às necessidades fundamentais como utilização dos recursos, pesquisa e segurança.
- Garantir qualidade de serviço não trivial através da combinação de seus recursos.

3.2 Tipos de Grades

As grades podem ser úteis de várias formas e em diversas aplicações. Para (SKILLICORN, 2002) são quatro os tipos de grade de acordo com o seu objetivo, cada qual com suas vantagens conforme segue:

- Grades computacionais: visam melhorar o desempenho e poder computacional aos usuários, utilizando o sistema como um único computador executando as tarefas em paralelo.
- Grades de acesso: visam integrar ambientes, fazendo com que usuários de diferentes domínios usufruam dos mesmos recursos como se estivessem fazendo parte de um só ambiente.
- Grades de dados: têm como objetivo a disponibilidade de dados, permitindo o armazenamento de grandes quantidades de informações em repositórios de fácil manipulação.
- Grades datacêtricos: têm o objetivo de compartilhamento de dados também, mas com a diferença que os dados não são movidos do seu local original.

3.3 Grades Orientadas a Serviços

Neste trabalho são importantes as grades que utilizam um paradigma de orientação a serviços. Neste capítulo são abordadas as características deste paradigma.

Para (DANTAS, 2005) as grades são uma extensão do conceito e da tecnologia dos conhecidos *webservices* promovendo a comunicação entre os nós através de mensagens solicitando serviços e o compartilhamento de seus recursos.

A principal vantagem desta abordagem em grades é a capacidade de fornecer serviços com cunho comercial e científico aos usuários, não importando onde esteja sendo feito o acesso, criando-se assim uma organização virtual.

Na estruturação deste paradigma o serviço está associado sempre a um proprietário que, de acordo com sua política, pode ou não cobrar por seu consumo. Os consumidores são as entidades que solicitam a execução dos serviços e estes podem também oferecer este serviço a outros consumidores se for de seu interesse (ASSUNCAO, 2004). Na interface entre o proprietário e o consumidor existe uma entidade que coordena as solicitações de serviço e as encaminham para os componentes da grade aptos a executá-las. Na estrutura do proprietário deve existir também alguma forma de os serviços serem publicados e informados ao coordenador de solicitações.

Através da figura 3.4, de (ROURE; JENNIGS; SHADBOLT, 2003), são apresentados por uma visão comercial os principais componentes da arquitetura orientada a serviços. Na figura são representados os serviços (círculos vermelhos) envolvidos por seu proprietário (retângulo envolvendo os serviços), e suas relações de consumo, contratos (traços sólidos), com os seus consumidores (triângulos). Os contratos são estabelecidos em um ambiente que representa um mercado (formas ovais) onde as regras são definidas por seu dono (cruz). O dono do mercado pode ser o proprietário dos serviços, um consumidor que oferece os serviços a terceiros, ou ainda uma entidade neutra.

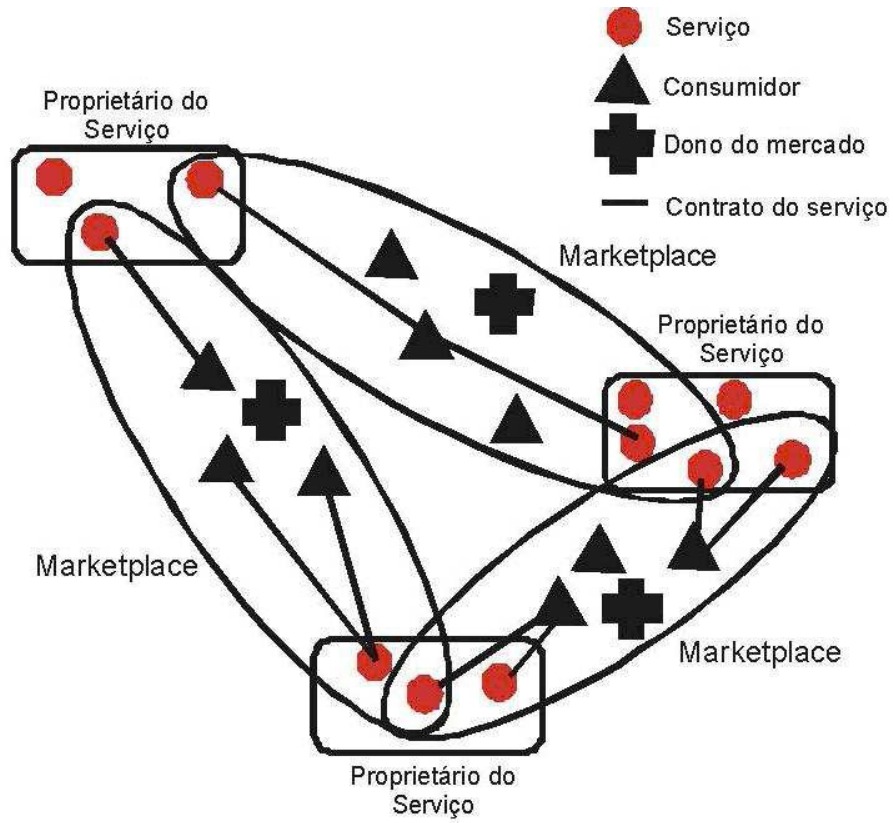


Fig. 3.4 Infra-estrutura orientada por serviços (ROURE; JENNIGS; SHADBOLT, 2003)

4 Grades Computacionais Autônomas

É possível unir os conceitos apresentados de grades computacionais e computação autônoma para se definir uma grade computacional autônoma, e neste capítulo serão abordados os conceitos e predicados que confirmam esta afirmação.

O diagrama da figura 4.1 representa a estrutura do elemento autônomo considerando um ambiente de grade, onde tem-se que os elementos gerenciados são os serviços e recursos disponíveis nos nós.

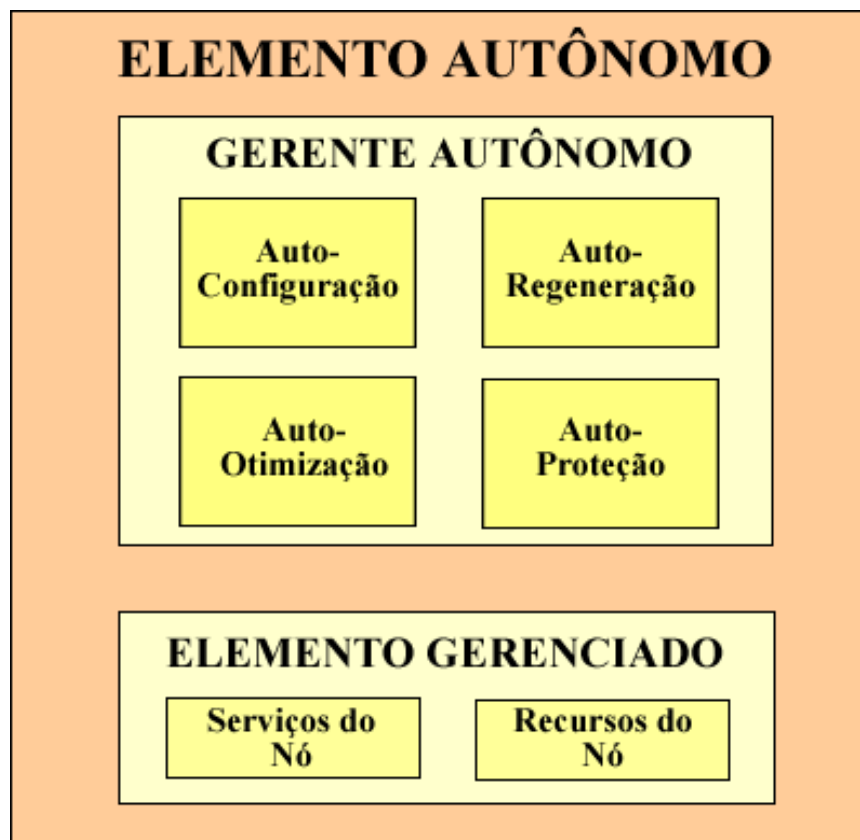


Fig. 4.1 Elemento autônomo.

Sabe-se que uma grade é um conjunto de unidades básicas chamadas de nós. Nos sistemas autônomos as unidades básicas são elementos autônomos, então se considera cada nó como sendo um elemento autônomo, assim como apresentado na figura 4.2. Os agentes representam gerentes autônomos.

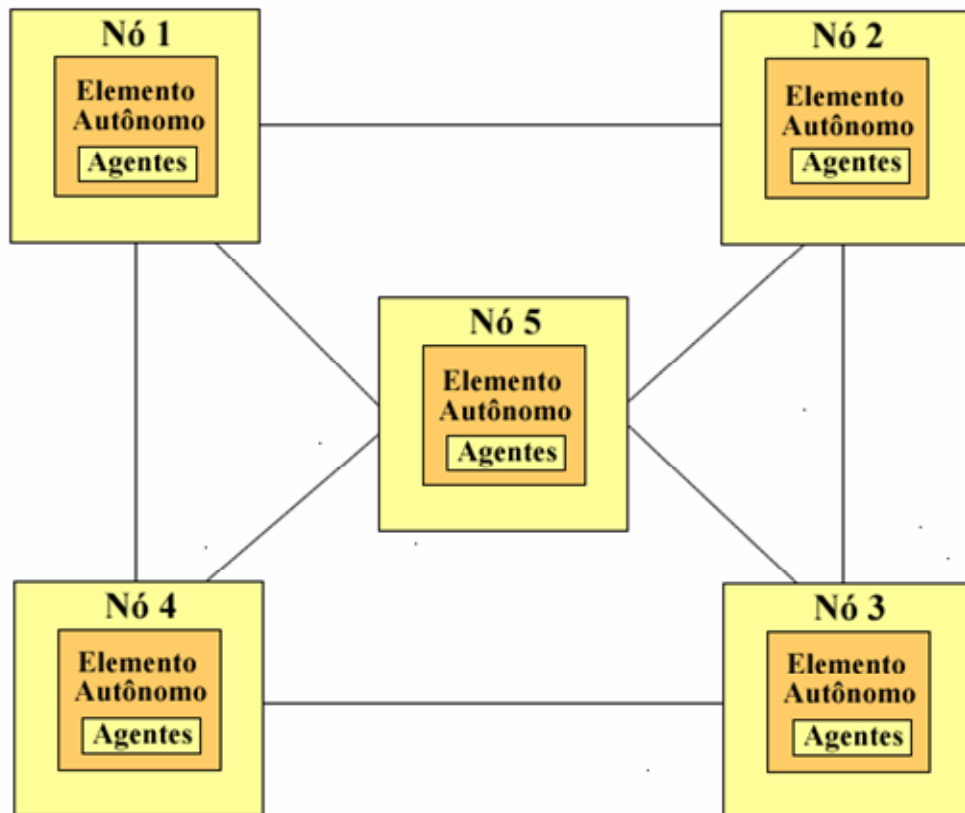


Fig. 4.2 Grade computacional de elementos autônomos.

Um sistema autônomo é definido pela presença de gerente autônomo em cada elemento autônomo. É o gerente autônomo que através do monitoramento de seus recursos e do seu ambiente externo é responsável por garantir um nível de auto-gerenciamento. Então se cada nó da grade é um elemento autônomo e cada elemento autônomo possui gerente autônomo, tem-se que a grade é um sistema autônomo.

4.1 Gerente Autônomo

Sobre o gerente autônomo, que é a peça mais importante para o escopo deste trabalho, pode-se representá-lo com suas funcionalidades voltadas para grade computacional conforme a figura 4.3.



Fig. 4.3 Funcionalidades do gerente autônomo.

Este trabalho envolve o desenvolvimento das funcionalidades de “Replicação de Serviços” e “Redirecionamento de Requisições de Serviços”, e utiliza a funcionalidade de “Monitoramento de Utilização do Hardware”.

5 Replicação de Serviços

A motivação para a implementação da funcionalidade de replicação de serviços é, conforme vimos, apoiar a gerência autônoma das redes. Esta funcionalidade tem por objetivo distribuir o processamento de determinado(s) serviço(s) que possa(m) estar sufocando o processamento em um nó ou um grupo de nós. É muito útil para grades onde a demanda por serviços fornecidos por poucos nós torna-se muito grande, pois se nenhuma ação fosse tomada nesta situação e a demanda continuasse alta ou crescente então esta grade certamente viria a passar por problemas de performance e seriam criados gargalos, por isso esta funcionalidade se encaixa no conceito de otimização no autogerenciamento.

A ação de replicar um serviço consiste basicamente em transferir, de um nó para outro, a implementação do código fonte do serviço na linguagem onde o ambiente de grade foi definido. Para exemplificar, dentre as formas de se fazer isto, podemos citar a transferência de arquivo compilado executável ou de código fonte para compilação no nó destino. Então, completando o processo, o nó recebendo a replicação inclui o fonte recebido com a implementação do serviço em sua biblioteca e inclui em sua lista de serviços disponibilizados o serviço replicado, disponibilizando-o à grade.

A figura 5.1 mostra a representação de dois momentos de uma grade para demonstrar o que é a replicação de um serviço. No início somente o nó 2 possui disponível determinado serviço, representado pela cor azul. Em determinado momento a demanda por solicitações de serviços no nó 2 faz com que diminua muito a capacidade de recursos nele e então o serviço é replicado para outro nó com maior disponibilidade naquele momento, nó 4, dividindo o processamento das demandas entre estes dois nós.

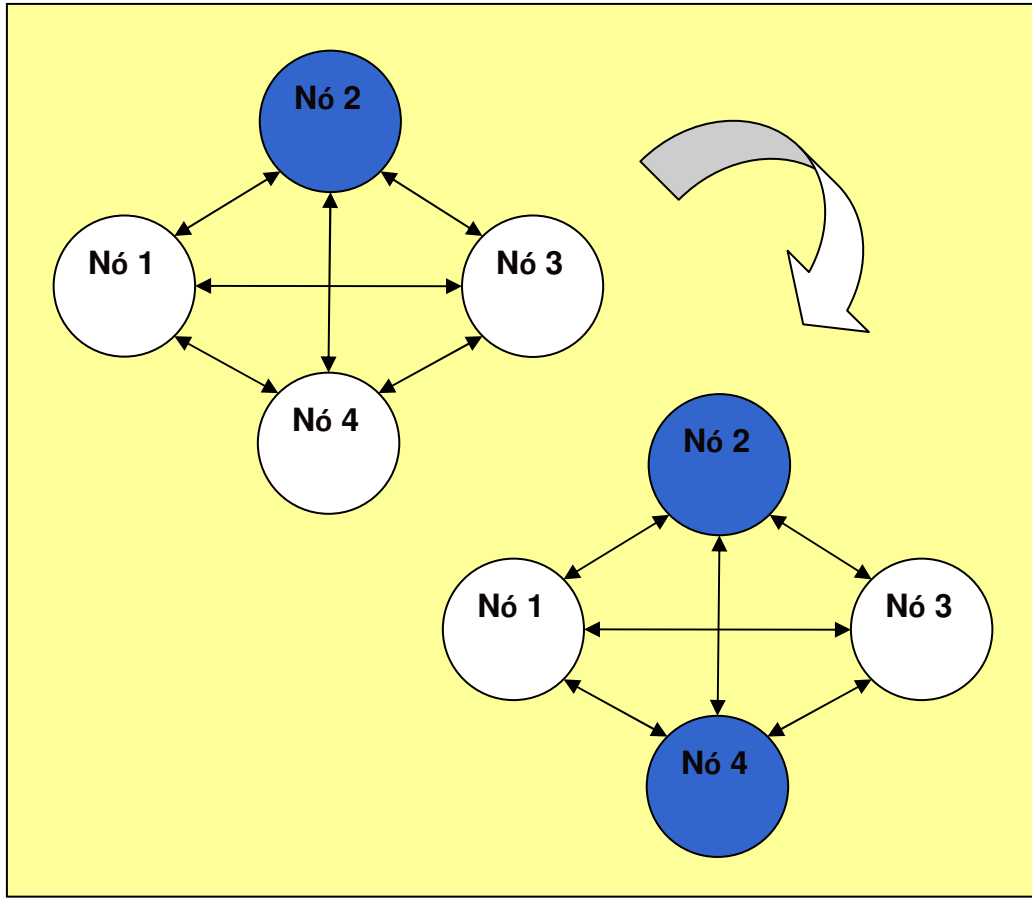


Fig. 5.1 Diagrama de grade com replicação de serviço.

Os nós de uma grade possuem uma lista de serviços básicos e conhecidos por todos que garantem o funcionamento do ambiente. Para realizar a replicação de serviços são necessários dois novos serviços básicos:

- **Replicar Serviço:** para que o nó possa replicar um serviço da grade quando solicitado. Este serviço tem como parâmetros de entrada o nome do serviço a ser replicado e o nó destinatário da replicação. Ao executar este serviço o executante envia uma tarefa com as informações necessárias para a implementação do serviço a ser replicado ao nó destinatário solicitando que este execute o serviço a seguir.
- **Receber Serviço:** para que o nó sem o serviço possa receber uma replicação. Tendo como parâmetros de entrada o nome e as informações da implementação do serviço replicado (e.g. código fonte, nome do objeto, localização do executável na rede, etc.). Este serviço deve adicionar a implementação em sua biblioteca e adicionar o serviço na lista de serviços disponibilizados pelo nó.

Sempre que houver uma replicação estes dois serviços são executados nesta ordem, “replicar serviço” e “receber serviço”, respectivamente pelo nó replicador e nó destino do serviço. Estas situações de replicação são abordadas no tópico a seguir que define uma política de roteamento de serviços.

5.1 Roteamento de Serviços

Tendo-se definido o que é a replicação e como ela acontece, deve-se determinar quando ela acontece. A intenção não é que serviços sejam replicados à vontade na grade e que todos os nós forneçam todos os serviços. Este trabalho tem uma proposta de roteamento a ser executado por qualquer nó, sempre que necessitar solicitar um serviço qualquer da grade, definindo as situações em que pode ocorrer replicação.

5.1.1 Definições e Métricas

Para a definição do roteamento proposto é necessário entender algumas definições e métricas dadas aos nós.

Percentual de recurso livre: é a relação entre recursos do sistema livre e recursos totais de um nó, ponderando 70% ao processador e 30% à memória. Fórmula:

$$\frac{(\text{Percentual_Processador_Livre} \times 0,7 + \text{Memória_Livre} \times 0,3) \times 100}{(\text{Percentual_Processador_Total} \times 0,7 + \text{Memória_Total} \times 0,3)}$$

Nível de recursos: os recursos de uma grade normalmente são heterogêneos, de diferentes tipos e com capacidade de recursos diferentes, portanto não se pode compará-los como se fossem todos iguais para fins de performance. Para resolver isto é possível criar uma escala de níveis com a granularidade que for mais conveniente. Nesta escala o nível zero é utilizado para os nós que nunca fornecerão serviços não-básicos à grade (e.g. sensores). O nível 1 é utilizado para os nós que fornecem serviços com menores capacidades de recursos e à medida que a capacidade de recursos melhora os nós avançam níveis na escala até o último nível que corresponde aos nós com melhores capacidades.

Recurso disponível: é o “nível de recursos” de um nó multiplicado pelo seu “percentual de recursos livre”. Este valor nivela os diferentes tipos de recursos da grade (nós) para um mesmo patamar de acordo com a escala de níveis de recursos criada. Fórmula:

$$\text{Nível_de_Recursos} \times \text{Percentual_Recursos_Livre}$$

Nó sobrecarregado: considera-se que um nó está sobrecarregado quando o seu “recurso disponível” no momento é menor do que N, sendo N um valor definido pela política da rede. N representa o limite de “percentual de recursos livre” onde um nó com “nível de recursos” igual a 1 ainda não é considerado sobrecarregado. Se, por exemplo, N for igual a 90, isto significa que um nó com “nível de recursos” igual a 1 passa a ser considerado sobrecarregado quando seu “percentual de recursos livre” for menor do que 90.

A tabela 5.1 mostra uma comparação com o percentual de recursos livre mínimo para que um nó não seja considerado sobrecarregado em cada nível de recursos, tendo uma granularidade de níveis de recursos que vai de zero a cinco e tomando-se N (limite de recursos livre do nível 1) igual a 90.

Nível de Recursos	Percentual de Recursos Livre Limite
0	*
1	90%
2	45%
3	30%
4	22,5%
5	18%

Tabela 5.1 Nível de recursos vs. Percentual de recursos livre limite

Utilizando a tabela apresentada, nós com níveis de 1 a 5 com os seus respectivos percentuais de recurso livre limite disponível estarão niveladas com o mesmo valor de recurso disponível (produto entre nível e percentual de recursos livres). Portanto, para um nó do nível 5 com 18% de recursos livre, um nó do nível 2 precisará ter 45% de recursos livre para

considerarmos que ambas tem o mesmo recurso disponível, mostrando que o nó do nível 5 é 2,5 vezes mais rica em recursos que o nó do nível 2.

5.1.2 Algoritmo de Roteamento

A aplicação de um nó, que utiliza o ambiente da grade, ao solicitar a execução de um serviço, não sabe quem executará a tarefa, portanto não define qual nó deve recebê-la. O algoritmo de roteamento de serviços no nó encarrega-se de pesquisar pelo nó mais adequado para executá-la no momento e, se for necessário para tanto, fará uma replicação de serviço. Este procedimento de roteamento ocorre na camada de controle interno do nó, onde é conhecida a sua tabela de rotas e como é feita a comunicação entre os nós, ficando transparente à aplicação que necessite fazer uma chamada de serviço que o nó executará a tarefa.

O algoritmo inicia seu processo fazendo uma varredura por todos os seus nós adjacentes, os quais o solicitante possui rota, e monta uma tabela com as informações retornadas. Para consultar os nós diretamente conectados em busca das informações necessárias inclui-se um novo serviço básico aos nós:

- **Pesquisar Serviço (Replicação):** quando solicitado por este serviço o nó responde se possui ou não o serviço, qual o seu recurso disponível atual e se aceita receber replicação de serviço (política do nó).

O objetivo é encontrar um nó, incluindo o próprio solicitante, que seja mais adequado a executar o serviço. Os critérios considerados para esta classificação, em ordem decrescente de importância, são: não estar sobrecarregado, possuir o serviço, ter mais recurso disponível e aceitar replicação, sendo este último obrigatório ser verdade caso chegue até este critério.

As figuras a seguir, 5.2 e 5.3, apresentam o fluxograma que utiliza estes critérios para definir qual nó deve executar o serviço e, se for o caso, entre quais nós deve ocorrer replicação. O fluxo do diagrama segue às condições das questões de cima para baixo, e quadros verdes significam respostas afirmativas e quadros vermelhos respostas negativas.

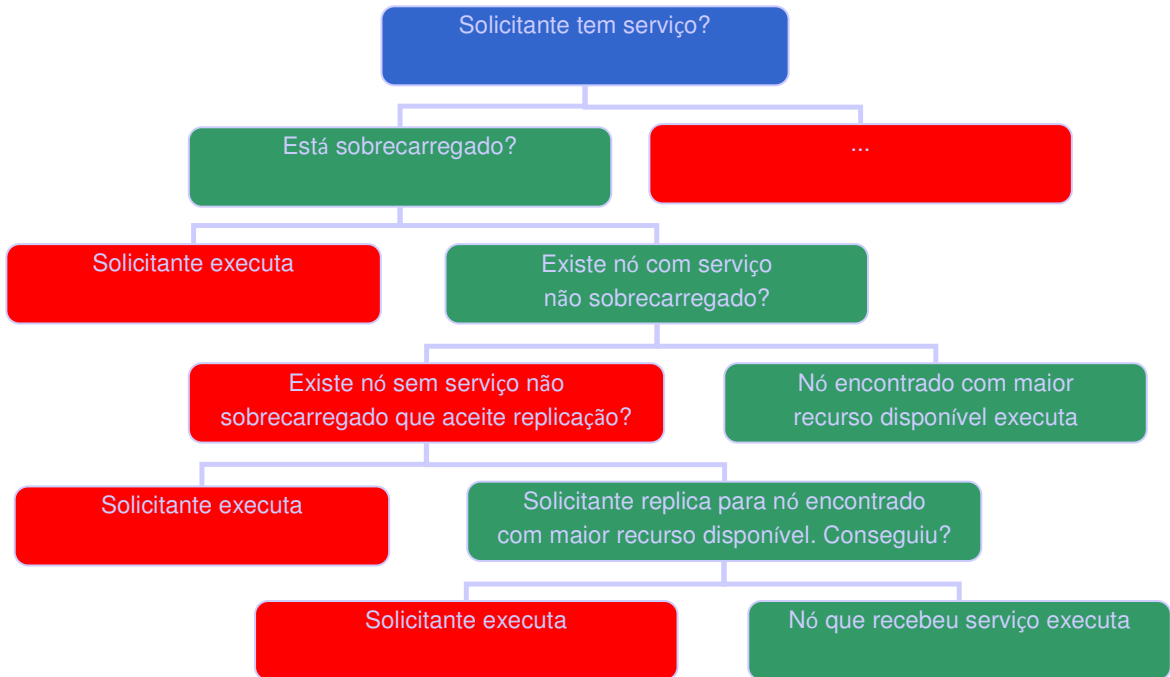


Fig. 5.2 Fluxograma do roteamento de serviços (1).

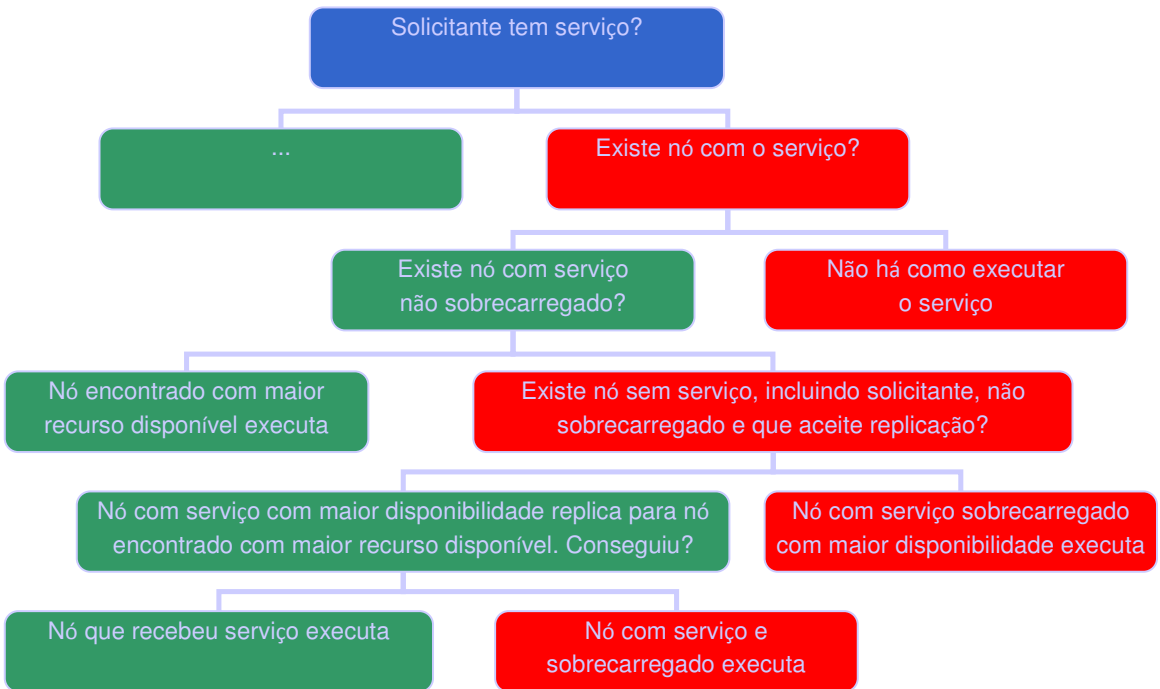


Fig. 5.3 Fluxograma do roteamento de serviços (2).

Como se pode perceber, a replicação só ocorre quando todos os nós que possuem o serviço estão sobrecarregados e existe algum nó sem serviço, não sobrecarregado e que aceite receber a replicação, e o nó que replica o serviço é o próprio solicitante caso ele possua o serviço ou então o nó com o serviço e com maior disponibilidade de recursos.

5.1.3 Conexão entre nós

Este trabalho propõe que o grafo que representa a grade seja completo, onde todos os nós possuem rotas para todos os outros. Não é mandatório construir a grade totalmente desta forma, mas é necessário que para cada nó solicitante de serviços da grade todos os nós conectados a ele estejam conectados entre si também. Isto porque em uma possível replicação de serviço os nós deverão ter comunicação entre si. Os resultados demonstram que o armazenamento destas informações na tabela de rotas e a execução do algoritmo nessas condições não oneram o desempenho das solicitações de serviço.

Para os melhores resultados desta proposta, é importante que os nós conectem-se ao menos aos principais nós provedores de serviços e com maior capacidade de recursos, permitindo assim a replicação entre estes nós.

5.2 Ambiente de Testes

O ambiente de testes onde a proposta apresentada neste trabalho foi implementada é o *middleware* Grid-M proposto em (Franke et al., 2007). Este aplicativo foi desenvolvido no “Laboratório de Redes e Gerência” (LRG) da Universidade Federal de Santa Catarina, e já foi publicado em diversos eventos, tendo sido também fonte de temas e servido como base para projetos desenvolvidos no mesmo laboratório.

5.2.1 Middleware em Ambientes Distribuídos

O *middleware* é o *software* integrado entre o sistema operacional e as aplicações que o utilizam tornando várias ações transparentes. Esta camada deve fornecer protocolos que permitam dispositivos de diversos tipos fazerem parte de um ambiente distribuído (Dantas,

2005), neste caso a grade. Este tipo de *software* é largamente utilizado na criação de ambientes distribuídos.

Pode-se citar como principais funções do *middleware*, de acordo com (FOSTER, 2000), deixar transparente o ambiente distribuído fazendo parecer um sistema único, ocultar a complexidade de diferentes dispositivos (sistema operacional, hardware, etc.) e fornecer interfaces padronizadas à aplicação para um fácil desenvolvimento, portabilidade e reusabilidade.

5.2.2 *Grid-M*

O Grid-M visa homogeneizar ambientes de redes com dispositivos heterogêneos, onde todos estes são vistos como nós de uma grade e utilizando virtualização de recursos (JOSEPH; ERNEST, 2004) abstraindo os vários tipos de recursos (serviços, processamento, recursos físicos, etc.) em uma interface comum, e este ambiente atualmente já possui certo nível de autonomia, unindo assim os conceitos abordados neste trabalho.

Este ambiente foi criado como uma biblioteca Java que integrada a uma aplicação provê uma *Application Programming Interface* (API) para a implementação de grades, oferecendo as funcionalidades básicas dos nós: comunicação através de tarefas, interfaces para implementação de serviços e sensores, segurança, *hook-up* para integrar lógica externa entre outras funcionalidades. Além destas funcionalidades o Grid-M oferece dinamismo à grade de forma que nós possam ser adicionados, removidos ou remanejados sempre que necessário.

A escolha do Grid-M para ambiente de implementação deste projeto se dá às características suportadas por ele e não suportadas na totalidade pelos *middlewares* convencionais, e necessárias para o projeto maior, em andamento, onde este trabalho se encaixa, são elas:

- Suporte a colaboração
- Suporte a sensibilidade ao contexto
- Suporte de alocação de recursos
- Suporte a ambiente dinâmico

- Suporte a execução de ambientes móveis
- Suporte a comportamento autônomo

6 Resultados

Através de testes em uma aplicação utilizando a biblioteca do Grid-M com esta proposta implementada foi constatado que o tempo de processamento do roteamento de serviços (i.e. pesquisa dos nós conectados mais processamento do algoritmo) é muito baixo ou insignificante, portanto a performance de uma solicitação de serviço depende apenas da capacidade de tráfego da rede para a troca de informações entre os nós, o que em redes fechadas não representa ônus devido à sua grande velocidade. Um teste com uma grade de 100 nós simulados em uma mesma máquina apresentou os seguintes resultados:

- Na média, pouco menos de um segundo para o processo de roteamento e execução de um serviço simples sem replicação do serviço.
- Na média, 1,6 segundo para o processo de roteamento e execução de um serviço simples com replicação do serviço, conhecido como o pior caso.

Vale lembrar que a replicação só é necessária uma vez e após isto o nó que recebeu o serviço passará a atender solicitações como conhecedor do serviço também.

O processo sem esta implementação vai variar de caso pra caso, pois não existe um padrão de processamento de tarefas e, portanto, não pode ser previsto com precisão.

A figura 6.1 mostra, através de dados retirados dos *logs* do Grid-M como ficou a utilização dos recursos de máquina ao longo do tempo em cada nó de uma grade com quatro nós de nível de recursos igual a 5 e N (limite de recursos livre do nível 1) igual a 90. Neste exemplo os nós 1 e 3 fazem requisições o tempo todo de um serviço que inicialmente somente o nó 2 possui. Em determinado momento este nó fica sobrecarregado (i.e. percentual de recursos livre se torna menor que 18%) e então ocorre uma replicação de serviço do nó 2, que possui o serviço, para o nó 4, que era o nó com mais recursos disponíveis naquele instante. Após isso as requisições pelo serviço passaram a ser respondidas pelo nó 4 também, distribuindo o processamento destas requisições. Como podemos perceber o algoritmo eliminou eminente saturação do nó 2 e criação de gargalo na rede.

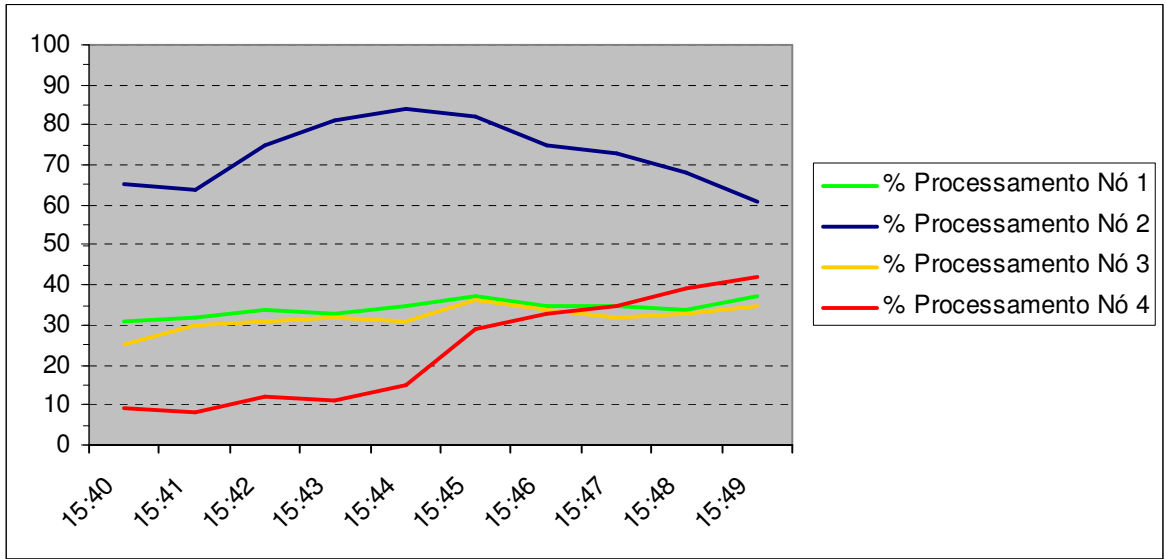


Fig. 6.1 Utilização de recursos dos nós e replicação de serviço

7 Conclusão

Conforme demonstrado nos testes, conclui-se que este controle além de não onerar o tempo de resposta dos serviços e de não aumentar significativamente o tráfego na rede devido aos seus pacotes pequenos, ainda cria uma proteção contra gargalos na rede e saturação de recursos em nós com serviços importantes para a grade. Os recursos necessários para a execução do controle também são insignificantes para o processamento no nó sendo que são muito simples.

Outro ponto é que em uma grade sem um bom roteamento de serviços o pior caso de execução de um serviço seria percorrer a grade inteira, talvez mais que uma vez, em busca do serviço. Já com o roteamento desta proposta conhecemos o pior caso que pode ocorrer que seria pesquisar informações de controle por todos os nós diretamente conectados ao solicitante, sendo que este não tem o serviço, todos que o tenham estão sobrecarregados e ocorre então uma replicação de um destes para outro nó sem o serviço e não sobrecarregado. Esta seria a situação onde se tem mais ações a serem tomadas e mesmo assim sabemos que o custo dela é muito baixo.

Portanto temos que qualquer tomada de recursos que esta proposta exija vale a pena tanto para a eliminação de gargalos e eventuais nós saturados na rede bem como para a o tempo de execução do serviço, pois conhecemos o pior caso de roteamento e o algoritmo sempre procura por nós mais ociosos para executar as tarefas balanceando as execuções de tarefas entre os nós na medida do possível, o que já é outro ponto otimizado em relação a grades com algoritmos de roteamento mal estruturados.

7.1 Trabalhos Futuros

É de grande valia para ambientes de grade computacional dar continuidade com os trabalhos de definição e implementação dos componentes do gerente autônomo para que se possam testar todos os benefícios que esta arquitetura tem possibilidades em proporcionar. Para tanto visa-se aumentar o nível de automação do *middleware* Grid-M em trabalhos futuros.

Referências Bibliográficas

- ALLEN, G. et al. The gridlab grid application toolkit. *High-Performance Distributed Computing, International Symposium on*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 411, 2002. ISSN 1082-8907.
- ASSUNCAO, M. D. de. *Implementação e Análise de uma Arquitetura de Grids de Agentes para a Gerência de Redes e Sistemas*. Dissertação (Mestrado) — Universidade Federal de Santa Catarina, 03 2004.
- BECKSTEIN, C. et al. Sogos - a distributed meta level architecture for the self-organizing grid of services. In: *MDM '06: Proceedings of the 7th International Conference on Mobile Data Management*. Washington, DC, USA: IEEE Computer Society, 2006. p. 82. ISBN 0-7695-2526-1.
- BRASILEIRO, F. et al. Bridging the high performance computing gap: the ourgrid experience. In: *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid/First Latin American Grid Workshop (LAGrid07)*. Rio de Janeiro, Brazil: [s.n.], 2007. p. 817–822.
- BRENNAND, C. et al. Automan: Gerência automática no ourgrid. In: *Anais do V Workshop de Grade Computacional e Aplicações*. Belém do Pará - BR: [s.n.], 2007.
- COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. *Distributed systems (4th ed.): concepts and design*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2005.
- DANTAS, M. *Computação distribuída de alto desempenho*. Rio de Janeiro, RJ, BR: Axcel Books do Brasil Editora, 2005.
- DASHOFY, E. M.; HOEK, A. van der. Towards architecture-based self-healing systems. In: *WOSS '02: Proceedings of the first workshop on Self-healing systems*. New York, NY, USA: ACM, 2002. p. 21–26. ISBN 1-58113-609-9.
- FAFNER. *Fafner-factoring via network-enabled recursion*. [S.l.]. Disponível em: <<http://www.lehigh.edu/~bad0/fafner.html>>.
- FOSTER, I. *Internet Computing and the Emerging Grid*. [S.l.]: Nature Web Matters. [S.l.], 2000. Disponível em: <<http://www.nature.com/nature/webmatters/grid/grid.html>>.
- FOSTER, I. What is the grid? a three point checklist. *GRIDToday*, 07 2002.
- FOSTER, I.; KESSELMAN, C. Globus: A metacomputing infrastructure toolkit. *Internacional Journal of Supercomputer Applications*, v. 11, n. 2, p. 115–128, 1997.
- FOSTER, I.; TUECKE, S. The anatomy of the grid: Enabling scalable virtual organizations. *IEEE*, 2001.

FRANKE, H. A. et al. Grid-m: Middleware to integrate mobile devices, sensors and grid computing. *The Third International Conference on Wireless and Mobile Communications - ICWMC*, 03 2007.

FRANKE, H. A. et al. Mobilidade em ambiente de grades computacionais. *XXIV Simpósio Brasileiro de Redes de Computadores. IV Workshop on Computational Grids and Applications - WCGA*, 06 2006.

GANEK, A. G.; CORBI, T. A. The dawning of the autonomic computing era. *IBM Syst. J.*, IBM Corp., Riverton, NJ, USA, v. 42, n. 1, p. 5–18, 2003. ISSN 0018-8670.

GARLAN, D.; SCHMERL, B. Exploiting architectural design knowledge to support self-repairing systems. In: *SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering*. New York, NY, USA: ACM, 2002. p. 241–248. ISBN 1-58113-556-4.

GEORGIADIS, I.; MAGEE, J. Self-organising software architectures for distributed systems. In: *WOSS '02: Proceedings of the first workshop on Self-healing systems*. New York, NY, USA: ACM, 2002. p. 33–38. ISBN 1-58113-609-9.

GONZÁLEZ-CASTA ñ. F. J. et al. Condor grid computing from mobile handheld devices. *SIGMOBILE Mob. Comput. Commun. Rev.*, ACM, New York, NY, USA, v. 6, n. 2, p. 18–27, 2002. ISSN 1559-1662.

GRIDBUS. *Grid Computing and Distributed Systems Laboratory*. [S.l.], 2005. Disponível em: <<http://www.gridbus.org/>>.

GRIDFORUM. *Understanding Grids*. [S.l.], 2006. Disponível em: <<http://www.gridforum.org/UnderstandingGrids%20-%20ggf%20grid%20understand.php>>.

HARIRI, S. *AUTONOMIA : An Autonomic Computing Environment*. [S.l.], 2007. Disponível em: <<http://www.ece.arizona.edu/~hpdc/projects/AUTONOMI>>.

HARIRI, S. et al. The autonomic computing paradigm. *Cluster Computing*, Kluwer Academic Publishers, Hingham, MA, USA, v. 9, n. 1, p. 5–17, 2006. ISSN 1386-7857.

HERRMANN, K.; MüHL, G.; GEIHS, K. Self-management: The solution to complexity or just another problem? *IEEE Distributed Systems Online (DSOnline)*, v. 6, n. 1, 2005. Disponível em: <<http://www.vs.uni-kassel.de/publications/2005/HMG05>>.

HINCHEY, M. G.; STERRITT, R. Self-managing software. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 39, n. 2, p. 107, 2006. ISSN 0018-9162.

HORN, P. *Autonomic Computing: IBM's Perspective on the State of Information Technology*. Armonk, NY, USA, 2001.

IBM. *IBM Grid computing*. [S.l.], 2001. Disponível em: <<http://www-1.ibm.com/grid/>>.

IBM. *An Architectural Blueprint for Autonomic Computing*. [S.l.], 2005. Disponível em: <<http://www.ibm.com/developerworks/autonomic/library/ac-summary/ac-blue.html>>.

JOSEPH, J.; ERNEST, M. Evolution of grid computing architecture and grid adoption models. *IBM Systems Journal*, v. 43, n. 4, 06 2004.

KEPHART, J. O.; CHESS, D. M. The vision of autonomic computing. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 36, n. 1, p. 41–50, 2003. ISSN 0018-9162.

LEGION. *World Wide Virtual Computer*. [S.l.], 2006. Disponível em: <<http://legion.virginia.edu/>>.

LEMOIS, R. de; FIADEIRO, J. L. An architectural support for self-adaptive software for treating faults. In: *WOSS '02: Proceedings of the first workshop on Self-healing systems*. New York, NY, USA: ACM, 2002. p. 39–42. ISBN 1-58113-609-9.

LIM, H. B. et al. Sensor grid: Integration of wireless sensor networks and the grid. In: *LCN '05: Proceedings of the The IEEE Conference on Local Computer Networks 30th Anniversary*. Washington, DC, USA: IEEE Computer Society, 2005. p. 91–99. ISBN 0-7695-2421-4.

LIU, H. et al. An autonomic service architecture for self-managing grid applications. In: *GRID '05: Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*. Washington, DC, USA: IEEE Computer Society, 2005. p. 132–139.

LOHMAN, G. M.; LIGHTSTONE, S. S. Smart: making db2 (more) autonomic. In: *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*. [S.l.]: VLDB Endowment, 2002. p. 877–879.

LRG. *Grid-m: Middleware for embedded and mobile grid computing*. [S.l.], 2006. Disponível em: <<http://grid.lrg.ufsc.br>>.

LUTHER, A. et al. *Alchemi: A.NET-based Grid Computing Framework and its Integration into Global Grids*. 2005.

MCCANN, J. A.; HUEBSCHER, M. C. Evaluation issues in autonomic computing. In: *Proceedings of Grid and Cooperative Computing Workshops (GCC)*, p. 597–608, 2004.

MICROSOFT. *Autoadmin*. [S.l.], 2005. Disponível em: <<http://research.microsoft.com/dmx-/autoadmin/>>.

MICROSYSTEMS, S. *Sun NI Service Provisioning System*. [S.l.], 2008. Disponível em: <http://www.sun.com/software/products/service_provisioning/index.xml>.

PATTERSON, D. et al. *Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies*. [S.l.], March 2002. Disponível em: <http://research.microsoft.com/~emrek/pubs/ROC_TR02-1175.pdf>.

ROLIM, C. O. *Uma arquitetura para submissão de aplicações de dispositivos móveis e embarcados para uma configuração de grade computacional*. Dissertação (Mestrado) — Universidade Federal de Santa Catarina, 04 2007.

ROURE, D. D.; JENNIGS, N. R.; SHADBOLT, N. The semantic grid: A future e-science infrastructure. In: . [S.l.]: John Wiley and Sons, 2003. cap. Grid Computing: Making The Global Infrastructure a Reality, p. 437–470.

SKILLICORN, D. B. *Motivating Computational Grids*. [S.l.]: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid, 2002. ISBN 0-7695-1582-7.

STERRITT, R. Towards autonomic computing: Effective event management. *27th Annual NASA Goddard Software Engineering Workshop (SEW-27'02)*, IEEE Computer Society, Los Alamitos, CA, USA, v. 00, p. 40, 2002.

UNICORE. *UNIform Interface to Computer Resources*. [S.l.], 2006. Disponível em: <<http://www.unicore.org/>>.

VALETTO, G.; KAISER, G. A case study in software adaptation. In: *WOSS '02: Proceedings of the first workshop on Self-healing systems*. New York, NY, USA: ACM, 2002. p. 73–78. ISBN 1-58113-609-9.

WEISER, M. The computer for the 21st century. *Scientific American*, February 1991. Disponível em: <<http://www.ubiq.com/hypertext/weiser/SciAmDraft3.html>>.

WOLF, T. D.; HOLVOET, T. Towards autonomic computing: agent-based modelling, dynamical systems analysis, and decentralised control. In: *First International Workshop on Autonomic Computing Principles and Architectures*. [S.l.: s.n.], 2003. p. 10.

APÊNDICE A

Trecho de código do algoritmo de roteamento definido. O nó deve utilizar para enviar tarefas e a aplicação utilizando esta interface não informa qual é o nó destino da tarefa por não saber qual nó a executará, podendo ser ele próprio.

Feito na linguagem Java utilizando a estrutura do Grid-M. As rotinas necessárias para esta proposta e utilizadas pelo algoritmo serão apresentadas nos próximos apêndices.

```

...
/** Lista de array[3] indexada pelos nomes dos nós:
    0 - tem serviço (0/1)
    1 - recurso disponível
    2 - aceita replicação (0/1) */
protected Hashtable nodeChoice;

...

int resource = this.getCurrentResource();
String executor, replicator = null;
int[] values = null;
int resourceBusy = 90; // política da rede - recurso livre limite
int resourceReplicator;

if (this.serviceTable.containsKey(task.getService())) {
// solicitante tem serviço

    if (resource < resourceBusy) {
// sobrecarregado

        populateNodeChoice(task.getService());
        executor = this.getServiceExecutor(resourceBusy-1, 1);

        if (!executor.equals("")) {
// há nó com serviço disponível

            task.setDestination(executor);

        } else {
// tenta replicar

            executor = this.getServiceExecutor(resourceBusy-1, 0);

            if (!executor.equals("")) {
// há nó sem serviço disponível - replica serviço

                xml = new XMLTree("parameters");
                xml.add("serviceName", task.getService());
                xml.add("destination", executor);
                Task replicateTask = new Task(this.name, "replicate-service", xml);
                replicateTask.setOriginator(this.name);
                TaskResult result = this.processServiceRequest(replicateTask);

```

```

        if (result.isOK()) {
            // replicou

            task.setDestination(executor);

        } else {
            // não replicou - solicitante executa

            task.setDestination(this.name);
            return this.processServiceRequest(task);
        }
    } else {
        // sem opção de replicar - deve executar

        task.setDestination(this.name);
    }
} else {
    // solicitante não sobrecarregado - pode executar

    task.setDestination(this.name);
    return this.processServiceRequest(task);
}

} else {
    // solicitante não tem serviço

    populateNodeChoice(task.getService());
    executor = this.getServiceExecutor(-1, 1);

    if (!executor.equals("")) {
        // há nó com serviço

        values = (int[]) this.nodeChoice.get(executor);
        resourceReplicator = values[1];

        if (resourceReplicator < resourceBusy) {
            // nó com serviço sobrecarregado

            replicator = executor;
            executor = this.getServiceExecutor((resource >=
resourceBusy)?resource:resourceBusy-1, 0);

            if (!executor.equals("") || resource >= resourceBusy) {
                // há nó sem serviço disponível - replica serviço

                executor = executor.equals("")?this.name:executor;

                xml = new XMLTree("parameters");
                xml.add("serviceName", task.getService());
                xml.add("destination", executor);
                Task replicateTask = new Task(replicator,"replicate-service",
xml);
                replicateTask.setOriginator(this.name);
                TaskResult result = this.sendServiceRequest(replicateTask);

                if (result.isOK()) {
                    // replicou

                    task.setDestination(executor);

```

```
        if (executor.equals(this.name)) {
            // replicou para o solicitante

            return this.processServiceRequest(task);
        }
        } else {
            // não replicou - nó com o serviço executa

            task.setDestination(replicator);
        }
        } else {
            // sem opção de replicar - nó com serviço executa

            task.setDestination(replicator);
        }
        } else {
            // nó com serviço não sobrecarregado - pode executar

            task.setDestination(executor);
        }
        } else {
            // nenhum nó conectado tem serviço

            return new TaskResult(task, TaskResult.RESULT_NOT_FOUND, new
XMLTree());
        }
    }

    // nó destino foi definido - enviar a tarefa
    return this.networker.sendTask(task);

    ...

```

APÊNDICE B

Rotinas da interface do nó para controle de recursos disponíveis.

```

...
/** Lista de array[3] indexada pelos nomes dos nós:
    0 - tem serviço (0/1)
    1 - recurso disponível
    2 - aceita replicação (0/1) */
protected Hashtable nodeChoice;

protected int resourceLevel = 5;
protected CurrentSystemStatus css;

...

/**
 * Get free CPU+MEM rate
 */
private double getFreeCpuMemRate() {

    int mem_livre = this.css.getMemoryStatus().getMemFree();
    int mem_total = this.css.getMemoryDesc().getMemTotal();
    int cpu_livre = this.css.getCPUStatus().getCPUFree();
    int cpu_total = this.css.getCPUDesc().getCPUTotal();

    double res = ((0.7 * cpu_livre + 0.3 * mem_livre) * 100 / (0.7 *
cpu_total + 0.3 * mem_total));

    return res;
}

/**
 * Get current available resource
 */
private int getCurrentResource() {

    return (int) (this.getFreeCpuMemRate() * this.resourceLevel);
}

/**
 * getServiceExecutor
 */
private String getServiceExecutor(int resource, int hasService) {

    String ret = "";
    Enumeration en;
    int[] values;
    int search = resource;

    if (!this.nodeChoice.isEmpty()) {

        en = this.nodeChoice.keys();
        while (en.hasMoreElements()) {

            String key = (String) en.nextElement();

```

```

        values = (int[]) this.nodeChoice.get(key);
        if (values[0] == hasService && values[1] > search) {

            search = values[1];
            ret = key;
        }
    }
}

return ret;
}

/**
 * Populate nodeChoice
 * @param service to search for
 */
private void populateNodeChoice(String service) {

    this.nodeChoice = new Hashtable();
    String destination = null;
    XMLTree xml = new XMLTree("discovery-parameters");
    xml.add("service", service);
    XMLTree routeTable = networker.getRouteTable(node.getName());
    Task discoveryTask;
    TaskResult result;
    XMLTree xmlresult;
    int hasService, resource, replication;

    Iterator it = routeTable.getSubTrees();

    while (it.hasNext()) {

        XMLTree next = (XMLTree) it.next();
        int metric = Integer.parseInt(next.getText("metric"));

        if (metric == 1) {

            destination = next.getTag();
            discoveryTask = new Task(destination, "discovery-rep", xml);
            discoveryTask.setOriginator(node.getName());
            result = this.networker.sendTask(discoveryTask);

            if (result.isOK()) {

                xmlresult = result.getParameters("result-discovery");
                hasService = new Integer(xmlresult.getText("hasService"));
                resource = new Integer(xmlresult.getText("resource"));
                replication = new Integer(xmlresult.getText("replication"));
                int values[] = {hasService, resource, replication};
                this.nodeChoice.put(destination, values);

            }
        }
    }
}

```

APÊNDICE C

Serviços presentes na interface do nó que permitem o roteamento e a replicação do serviço: “pesquisar serviço (replicação)”, “replicar serviço” e “receber serviço”.

```
...  
  
// DISCOVERY REPLICATION //  
else if (serviceName.equals("discovery-rep")) {  
  
    XMLTree xml = task.getParameters();  
  
    // Get discovery-parameters  
    XMLTree d = xml.get("discovery-parameters");  
  
    String service = d.getText("service");  
    String resource = "" + this.getCurrentResource();  
  
    if (this.serviceTable.containsKey(service)) {  
        // tem serviço  
        xml = new XMLTree("result-discovery");  
        xml.add("hasService", "1");  
        xml.add("resource", resource);  
        xml.add("replication", "0");  
  
    } else {  
        // não tem serviço  
        xml = new XMLTree("result-discovery");  
        xml.add("hasService", "0");  
        xml.add("resource", resource);  
        xml.add("replication", this.acceptReplication);  
  
    }  
  
    result = new TaskResult(task, TaskResult.RESULT_OK, xml);  
  
}
```



```

// REPLICATE SERVICE //
else if (serviceName.equals("replicate-service")) {

    // coleta parâmetros
    XMLTree taskParameters = task.getParameters().get("parameters");

    String repServiceName = taskParameters.getText("serviceName");
    String destination = taskParameters.getText("destination");

    Service service = hasService(repServiceName);
    if (service == null) {
        return new TaskResult(task, TaskResult.RESULT_FAILURE, new
XMLTree());
    }

    // carrega .JAVA
    String className = service.getClass().getSimpleName();
    String classString = JavaFile.loadJavaFromBuildDir("gridm/services/" +
this.name + "/" + className);

    // replica
    XMLTree parameters = new XMLTree("parameters");

    parameters.add("serviceName", repServiceName);
    parameters.add("className", className);
    parameters.add("classString", classString);

    Task replicateTask = new Task(destination, "receive-service",
parameters);
    TaskResult replicateResult = this.sendServiceRequest(replicateTask);
    System.out.println(replicateResult.toString());
    System.out.println(replicateResult.getResults());

    if (replicateResult.isOK()) {
        result = new TaskResult(task, TaskResult.RESULT_OK, new XMLTree());
    }
    else {
        result = new TaskResult(task, TaskResult.RESULT_FAILURE, new
XMLTree());
    }
}
}

```

```

// RECEIVE SERVICE //
else if (serviceName.equals("receive-service")) {

    // coleta parâmetros
    XMLTree taskParameters = task.getParameters().get("parameters");

    String repServiceName = taskParameters.getText("serviceName");
    String className = taskParameters.getText("className");
    String classString = taskParameters.getText("classString");

    String classDir = ClassLoader.getResource("").getPath();
    classDir = classDir + "gridm/services/" + task.getOriginator();

    // cria .JAVA
    try {
        JavaFile.writeFile(classDir, className + ".java", classString, true);
    }
    catch (IOException e) {
        System.out.println(e.getMessage());
        return new TaskResult(task, TaskResult.RESULT_FAILURE, new
XMLTree());
    }

    // compila
    try {
        JavaFile.compile(classDir + "/" + className + ".java");
    }
    catch (IOException e) {
        System.out.println(e.getMessage());
        return new TaskResult(task, TaskResult.RESULT_FAILURE, new
XMLTree());
    }

    // adiciona serviço
    Service service = hasService(serviceName);
    if (service != null) {
        this.serviceTable.remove(serviceName);
    }
    String classPackage = "gridm.services." + task.getOriginator() + "." +
className;
    Object[] constrParams = new Object[0];
    try {
        Object serviceObject =
Class.forName(classPackage).getConstructors()[0].newInstance(constrParams);
        this.addService(repServiceName, (Service) serviceObject);
    }
    catch (ClassNotFoundException e) {
        System.out.println("Classe do Serviço não encontrada!");
        return new TaskResult(task, TaskResult.RESULT_FAILURE, new
XMLTree());
    }

    result = new TaskResult(task, TaskResult.RESULT_OK, new XMLTree());
}

...

```

APÊNDICE D

Classe JavaFile utilizada para manipular o arquivo onde o serviço foi implementado. Esta classe considera que o arquivo fonte está em um diretório específico da biblioteca java e em um subdiretório com o nome do nó.

```
import java.io.*;
import java.util.Arrays;
import javax.tools.DiagnosticCollector;
import javax.tools.JavaCompiler;
import javax.tools.JavaFileObject;
import javax.tools.StandardJavaFileManager;
import javax.tools.ToolProvider;

public class JavaFile {

    //
    // COMPILE JAVA FILE
    //

    public static void compile(String javaFile) throws IOException {

        javaFile = javaFile.replaceAll("%20", " ");
        JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();

        DiagnosticCollector<JavaFileObject> diagnostics = new
DiagnosticCollector<JavaFileObject>();

        StandardJavaFileManager fileManager =
compiler.getStandardFileManager(diagnostics, null, null);

        Iterable<? extends JavaFileObject> compilationUnits =
fileManager.getJavaFileObjectsFromStrings(Arrays.asList(javaFile));

        JavaCompiler.CompilationTask task = compiler.getTask(null, fileManager,
diagnostics, null, null, compilationUnits);

        boolean success = task.call();
        fileManager.close();
        System.out.println("Success: " + success);

    }
}
```

```
//  
// LOAD JAVA FROM BUILT CLASSES DIRECTORY  
//  
public static String loadJavaFromBuildDir (String className) throws  
IOException {  
    InputStream classFile = ClassLoader.getResourceAsStream(className  
+ ".java");  
    byte[] byteFile = new byte[1000000]; //ateh 1MB  
    int tam = classFile.read(byteFile);  
    classFile.close();  
  
    if (tam == 0) {return "";}  
    byte[] classBytes = new byte[tam];  
    for (int i=0;i<tam;i++) {  
        classBytes[i] = byteFile[i];  
    }  
  
    return new String(classBytes);  
}  
  
//  
// WRITE FILE  
//  
public static void writeFile (String dir, String name, String text,  
boolean create) throws IOException {  
  
    dir = dir.replaceAll("%20", " ");  
    File file = new File(dir);  
    if (create) {file.mkdirs();}  
    file = new File(dir + "/" + name);  
    if (create) {file.createNewFile();}  
    FileWriter fw = new FileWriter(file);  
    fw.write(text);  
    fw.flush();  
    fw.close();  
}  
  
} //CLASS
```

APÊNDICE E

Replicação de Serviço em Ambiente de Grade Computacional Autônoma

Diogo Miguel Martendal

Curso de Ciências da Computação
Universidade Federal de Santa Catarina

dmm@inf.ufsc.br

***Abstract.** The goal of this dissertation is to define and implement an autonomous service replication process in grid environment. The grid computing environment is widely applied as a cheap mode to enhance computing power by grouping devices in a net and sharing processing and resources as if the net was a single great computer. When the purpose is computing power and a reliable environment it's important that it could self manage aiming at optimization. The service replication provides routine implementation transporting among devices, which allows better processing distribution and control against processing overhead.*

***Resumo.** O objetivo deste trabalho é definir e implementar um processo de replicação de serviços autônomo em um ambiente de grade computacional. O ambiente de grade é muito utilizado como uma forma barata de aumentar o poder computacional unindo-se dispositivos em uma rede e compartilhando processamento e recursos como se a rede fosse um único grande computador. Quando o objetivo é poder computacional e um ambiente confiável é importante que o mesmo consiga se gerenciar automaticamente visando a otimização. A replicação de serviço proporciona uma forma de transmissão de implementação de rotinas entre os dispositivos, permitindo uma melhor distribuição do processamento e um controle contra gargalos de processamento.*

Palavras-chave: grade computacional, computação autônoma, replicação de serviço

1 Introdução

A computação distribuída surgiu da necessidade por integrar informações e aplicações em redes para que se tenha acesso a estas em qualquer ponto [WEISER, 1991]. Isto contribui com a crescente complexidade das redes e a necessidade de formas de gerência destes recursos. O conceito de grade computacional [FOSTER, 2000], amplamente conhecido como grid, tem objetivo de criar poder computacional através da união de dispositivos comuns e vem definir uma idéia para tornar homogênea uma rede com dispositivos, sistemas

operacionais, informações e serviços heterogêneos, o que torna o ambiente mais fácil de gerenciar.

A dependência da interação humana em gerência de redes de computadores já foi viável e suficiente mas a complexidade alcançada por elas hoje exige uma maior autonomia, com a diminuição da interação humana em tarefas de baixo-nível e aproveitando a experiência destes profissionais para definições de gerência em alto nível, definindo políticas de rede. Este contexto estimulou a pesquisa e aplicação de computação autônoma [HORN, 2001] nas redes de computadores, introduzindo propriedades de auto-gerenciamento [HINCHEY; STERRITT, 2006].

A proposta é definir e construir uma funcionalidade de replicação de serviços para um projeto de auto-gerenciamento em grades computacionais. O objetivo é incrementar a autonomia de uma grade computacional impedindo que se criem gargalos de processamento em dispositivos que possuem serviços frequentemente utilizados pela grade, distribuindo este processamento e melhorando a utilização dos recursos.

A aplicação base para a implementação da solução é o middleware Grid-M, uma biblioteca Java que fornece todos os recursos para a criação de um ambiente de grade, desenvolvido no “Laboratório de Redes e Gerência” (LRG) da Universidade Federal de Santa Catarina. O Grid-M contribuirá com outras funcionalidades tais como o monitoramento de utilização do hardware e a gerência de serviços, necessárias para este trabalho, e este trabalho agregará indiretamente com uma funcionalidade de roteamento de requisições de serviços.

2 Computação Autônoma

A origem do termo computação autônoma vem de uma analogia com a autonomia do sistema nervoso humano. O organismo humano tem a capacidade de perceber mudanças internas e externas e agir sobre o corpo de forma que ele possa continuar as atividades que estamos aptos a fazer e manter as funções vitais. Assim como o sistema nervoso humano, um sistema autônomo deve ser capaz de se gerenciar para executar tarefas suportadas da melhor forma possível ou até mesmo para simplesmente se manter em funcionamento, devendo ser capaz de fazer isto mesmo em situações de estresse ou excesso de trabalho.

O processo de gerência de redes conta com operadores para, além de tarefas de alto-nível, agir em tarefas de baixo-nível sempre que ocorrem problemas iminentes [IBM, 2003], mas esta dependência é cada vez menos apropriada visto a crescente escala destes ambientes e a necessidade de estarem sempre disponíveis. A capacidade de automação de um sistema é importante para a otimização da utilização dos seus recursos, e isto é ainda mais evidente em um ambiente que normalmente visa poder computacional, assim como é um ambiente de grade computacional. Várias necessidades do sistema distribuído exigem um trabalho complexo e longo, e por isso são suscetíveis a erro quando efetuados manualmente, mas quando automatizado traz mais garantias de qualidade, de adesão às políticas e mais agilidades.

2.1 Auto-gerenciamento

Sistemas auto-gerenciados [HINCHEY; STERRITT, 2006] são aqueles que gerenciam suas próprias operações sem a intervenção humana. Atualmente a iniciativa industrial mais importante sobre auto-gerenciamento é a “Autonomic Computing Initiative” (ACI), iniciada pela IBM em 2001, que define quatro características principais para um sistema auto-gerenciado:

- **Auto-configuração:** capacidade de configurar-se dinamicamente sem intervenção externa, de modo otimizado e garantindo seu perfeito funcionamento. Em grades computacionais podemos exemplificar esta característica com a auto-conexão de novos nós, fazendo com que estes venham a fazer conexões vitais para a melhor utilização dos recursos disponíveis.
- **Auto-regeneração:** deve poder se auto-diagnosticar encontrando problemas e então conseguir agir de forma reativa e se modificar para corrigir ou contornar alguma falha e continuar funcionando. Espera-se que sistemas possam continuar disponíveis mesmo quando ocorrem panes não generalizadas. Em uma grade computacional a utilidade deste controle é clara visto que os recursos disponibilizados são de grande importância e para tanto o sistema deveria saber como reagir em caso de indisponibilidade de seus dispositivos.
- **Auto-otimização:** utilizar os recursos disponíveis da melhor forma possível e mesmo em situações de carga excessiva de trabalho estar apto a obter performance positiva de respostas. Como funcionalidade em grades computacionais tem-se este próprio trabalho, o qual visa evitar sobrecargas de processamento permitindo replicar serviços e distribuir o processamento de suas requisições. Isto impede que se criem gargalos, que nós fiquem sobrecarregados e o tempo de resposta de requisições dos serviços aumente.
- **Auto-proteção:** deve haver formas de garantir a segurança de informações e serviços importantes, proteger-se contra ataques maliciosos ou utilização indevida, não permitir acesso e alteração indevida de parâmetros do sistema, evitando que ocorram falhas em cascata e o sistema como um todo seja prejudicado, e tornando-o mais seguro e confiável. Nas grades computacionais esta funcionalidade poderia atuar verificando as estatísticas de uso dos nós e descobrir se algum nó está fazendo um mau uso dos recursos, mesmo que não intencionalmente.

Devido à complexidade de se implementar todas as características do auto-gerenciamento de uma só vez, a ACI definiu a evolução da automação em 5 níveis, de forma que no primeiro nível tem-se o estado atual do gerenciamento de redes, onde o operador gerencia os recursos manualmente, evoluindo do nível 2 ao 4, e no nível 5 tem-se o sistema totalmente autônomo.

3 Grades Computacionais

Na história da computação apareceram problemas que exigiram mais poder computacional. Supercomputadores têm um custo muito elevado e uma das primeiras idéias

que surgiram para contornar isto foram os clusters, utilizando multiprocessamento em diversos computadores e até hoje ainda muito adequados. Depois surgiu a idéia de multiprocessamento em ambientes distribuídos suportando grandes aplicações, as grades computacionais [FOSTER, 2000], conseguindo unir uma grande quantidade de recursos com um custo ainda menor que os clusters.

O nome grade computacional vem do termo em inglês *grid computing*, que tem sua origem em uma analogia com a *electrical power grid* (rede de energia elétrica norte-americana), onde o usuário final de energia elétrica não precisa conhecer como a rede é formada ou como funciona para poder utilizá-la, nem mesmo precisa conhecer quem está fornecendo, apenas se preocupa em conectar seu equipamento através de uma tomada e consumir a energia. Assim como na rede elétrica, a grade computacional visa tornar o que acontece na rede transparente ao usuário, assim como seu gerenciamento e sua estrutura.

A maior motivação para utilização de grades é prover ambientes com o máximo de poder computacional. Por exemplo, é uma forma de unir redes de diferentes domínios dispersos geograficamente compartilhando dados, software e equipamentos, além de recursos de máquina e quaisquer outros recursos, por isto pode-se dizer que a grade é um passo tecnológico seguinte em relação à internet, pois visa compartilhar todos estes recursos além de compartilhar informações.

As implementações de grades normalmente funcionam através da inclusão de uma nova camada de software entre as aplicações e a infra-estrutura física composta pelo computador, sistema operacional e redes, ou seja, um *middleware*, que esconde a complexidade e diversidade da infra-estrutura oferecendo uma interface uniforme para acesso aos recursos. As aplicações visualizam o sistema como se fosse um único computador e podem ignorar qualquer informação de localização preocupando-se apenas com a utilização dos recursos.

As grades podem ser úteis de várias formas e em diversas aplicações. De acordo com o seu objetivo podem priorizar uma característica ou outra, tais como, desempenho e poder computacional, uma forma de integrar ambientes e permitir acesso entre diferentes domínios, ou proporcionar disponibilidade de dados e informações [SKILLICORN, 2002].

3.1 Grades Orientadas a Serviços

Neste trabalho são importantes as grades que utilizam um paradigma de orientação a serviços. Neste paradigma, as grades são uma extensão do conceito e da tecnologia dos conhecidos *webservices* promovendo a comunicação entre os nós através de mensagens solicitando serviços e o compartilhamento de seus recursos [DANTAS, 2005].

Esta abordagem permite a criação de organizações virtuais, onde se cria uma espécie de mercado. Existem fornecedores de serviços que os disponibilizam para utilização podendo ou não obter lucro com isto. Os consumidores são as entidades que necessitam destes serviços e criam um vínculo de negócio para poderem utilizá-los.

4 Grades Computacionais Autônomas

É possível unir os conceitos apresentados de grades computacionais e computação autônoma para se definir uma grade computacional autônoma.

Cada unidade básica de um sistema autônomo é um elemento autônomo [HERRMANN et al., 2005]. Um elemento autônomo é uma estrutura de controle fechado e possui dois componentes:

- Gerente autônomo: que é responsável pelo auto-gerenciamento do elemento. Uma estrutura de tomada de decisões através de estímulos internos e externos.
- Elementos gerenciados: que são os serviços e recursos, hardware ou software, de que este elemento dispõe e que são controlados pelo gerente autônomo. No caso de uma grade computacional são os serviços e recursos do nó.

Sabe-se que uma grade é um conjunto de unidades básicas chamadas de nós. Nos sistemas autônomos as unidades básicas são elementos autônomos, então se considera cada nó como sendo um elemento autônomo. Um sistema autônomo é definido pela presença de gerente autônomo em cada elemento autônomo pois é o gerente autônomo que através do monitoramento de seus recursos e do seu ambiente externo é responsável por garantir um nível de auto-gerenciamento. Então se cada nó da grade é um elemento autônomo e cada elemento autônomo possui gerente autônomo, tem-se que a grade é um sistema autônomo.

O gerente autônomo, que é a peça mais importante para o escopo deste trabalho, possui várias funcionalidades quando voltado para o contexto de grade computacional, tais como: gerência de serviços e recursos, roteamento de requisições de serviços, replicação de serviços, monitoramento de utilização do hardware e integridade, tomada de decisões, etc. Este trabalho envolve o desenvolvimento das funcionalidades de replicação de serviços e roteamento de requisições de serviços, e utiliza a funcionalidade de monitoramento de utilização do hardware.

5 Replicação de Serviços

Esta funcionalidade do gerente autônomo visa evitar gargalos e distribuir as requisições de determinado serviço que possa estar sobrecarregando os recursos em um nó ou um grupo de nós. Se nenhuma ação fosse tomada nesta situação e a demanda continuasse alta ou crescente então esta grade certamente viria a passar por problemas de performance e seriam criados gargalos, por isso esta funcionalidade se encaixa no conceito de otimização no auto-gerenciamento.

A ação de replicar um serviço consiste basicamente em transferir, de um nó para outro, a implementação do código fonte do serviço. Dentre as formas de se fazer isto podemos citar a transferência de arquivo compilado executável ou de código fonte para compilação no nó destino. Então, completando o processo, o nó recebendo a replicação inclui o fonte recebido com a implementação do serviço em sua biblioteca e inclui em sua lista de serviços disponibilizados o serviço replicado, disponibilizando-o à grade.

Para realizar a replicação de serviços são necessários dois novos serviços básicos na interface do nó: um para que o nó replique um serviço seu para outro nó quando solicitado, e outro serviço para que o nó possa receber uma replicação agregando um novo serviço.

5.1 Roteamento de Serviços

Definido o que é a replicação e como ela acontece, deve-se definir quando ela acontece. A intenção não é que serviços sejam replicados à vontade na grade e que todos os nós forneçam todos os serviços. Este trabalho tem uma proposta de roteamento a ser executado por qualquer nó, sempre que necessitar solicitar um serviço qualquer da grade, definindo as situações em que possa ocorrer replicação.

5.1.1 Métricas

Para a definição do roteamento proposto são necessárias as seguintes definições de métricas:

- **Percentual de recurso livre:** é a taxa de recursos livres de um nó, ponderando 70% ao processador e 30% à memória.

$$\frac{(\text{Percentual_Processador_Livre} \times 0,7 + \text{Memória_Livre} \times 0,3) \times 100}{(\text{Percentual_Processador_Total} \times 0,7 + \text{Memória_Total} \times 0,3)}$$

- **Nível de recursos:** os recursos de uma grade normalmente são heterogêneos, de diferentes tipos e com capacidade de recursos diferentes, portanto não se pode compará-los como se fossem todos iguais para fins de performance. Para normalizá-los cria-se uma escala de níveis com a granularidade que for mais conveniente. Nesta escala o nível zero é utilizado para os nós que nunca fornecerão serviços próprios à grade (e.g. sensores). O nível 1 é utilizado para os nós que fornecem serviços com menores capacidades de recursos e à medida que a capacidade de recursos aumenta os nós avançam níveis na escala até o último nível que corresponde aos nós com melhores capacidades.
- **Recurso disponível:** é o “nível de recursos” de um nó multiplicado pelo seu “percentual de recursos livre”. Este valor nivela os diferentes tipos de recursos da grade para um mesmo patamar de acordo com a escala de níveis de recursos criada.

$$\text{Nível_de_Recursos} \times \text{Percentual_Recursos_Livre}$$

- **Nó sobrecarregado:** considera-se que um nó está sobrecarregado quando o seu “recurso disponível” no momento é menor do que N, sendo N um valor definido pela política da rede. N representa o limite de “percentual de recursos livre” onde um nó com “nível de recursos” igual a 1 ainda não é considerado sobrecarregado.

A tabela 1 mostra o percentual de recursos livre mínimo para que um nó não seja considerado sobrecarregado em cada nível de recursos, considerando que os níveis de recursos vão de zero a cinco e N igual a 90. Nesta tabela os percentuais de todos os níveis estão normalizados, portanto, para um nó do nível 5 com 18% de recursos livre, um nó do nível 2 precisará ter 45% de recursos livre para considerarmos que ambas tem o mesmo recurso disponível, mostrando que o nó do nível 5 tem 2,5 vezes mais recursos que o nó do nível 2.

Nível de Recursos	Percentual de Recursos Livre Limite	Nível de Recursos	Percentual de Recursos Livre Limite
0	*	3	30%
1	90%	4	22,5%
2	45%	5	18%

Tabela 1 – Nível de recursos vs. Percentual de recursos livre limite

5.1.2 Algoritmo de Roteamento

A aplicação de um nó, que utiliza o ambiente da grade, ao solicitar a execução de um serviço, não sabe quem executará a tarefa, portanto não define qual nó deve recebê-la. O algoritmo de roteamento de serviços na interface do nó encarrega-se de pesquisar pelo nó mais adequado para executá-la no momento e, se for necessário para tanto, fará uma replicação de serviço.

O algoritmo proposto aqui faz uma varredura por todos os seus nós adjacentes para a tomada de decisão. Para consultar os nós diretamente conectados é necessário um serviço na sua interface que retorne as seguintes informações: se possui ou não o serviço, qual o seu recurso disponível atual e se aceita receber replicação de serviço (política do nó).

O objetivo é encontrar um nó, incluindo o próprio solicitante, que seja mais adequado a executar o serviço. Os critérios considerados para esta classificação, em ordem decrescente de importância, são: não estar sobrecarregado, possuir o serviço, ter mais recurso disponível e aceitar replicação, sendo este último obrigatório ser verdade caso chegue até ele. As figuras 1 e 2 apresentam o fluxograma que utiliza estes critérios e os casos onde ocorrem replicação. O fluxo do diagrama segue às condições das questões de cima para baixo, e quadros verdes significam respostas afirmativas e quadros vermelhos respostas negativas. Como se pode perceber a replicação só acontece se todos os nós que possuem o serviço estão sobrecarregados, o mais apto a executar não possui o serviço, não está sobrecarregado e aceita replicação, sendo a última opção considerando as prioridades listadas.

A solicitação de replicação é emitida do nó que está executando o algoritmo, e ela pode ser determinada entre um segundo e um terceiro nós, então para que a replicação seja bem sucedida deve existir uma forma de conexão entre estes nós. A sugestão é que o grafo que representa a grade seja completo, onde todos os nós possuem rotas para todos os outros. Desta forma o nó solicitante consegue consultar diretamente todos os nós em busca das informações para o algoritmo, e é garantido que os nós envolvidos na replicação estarão diretamente conectados também. Não é mandatório construir a grade totalmente desta forma,

mas é necessário que para cada nó solicitante de serviços da grade todos os nós conectados a ele estejam conectados entre si também, assim também é possível prever qualquer execução do algoritmo. O armazenamento destas informações na tabela de rotas e a execução do algoritmo nessas condições não oneram o desempenho das solicitações de serviço.

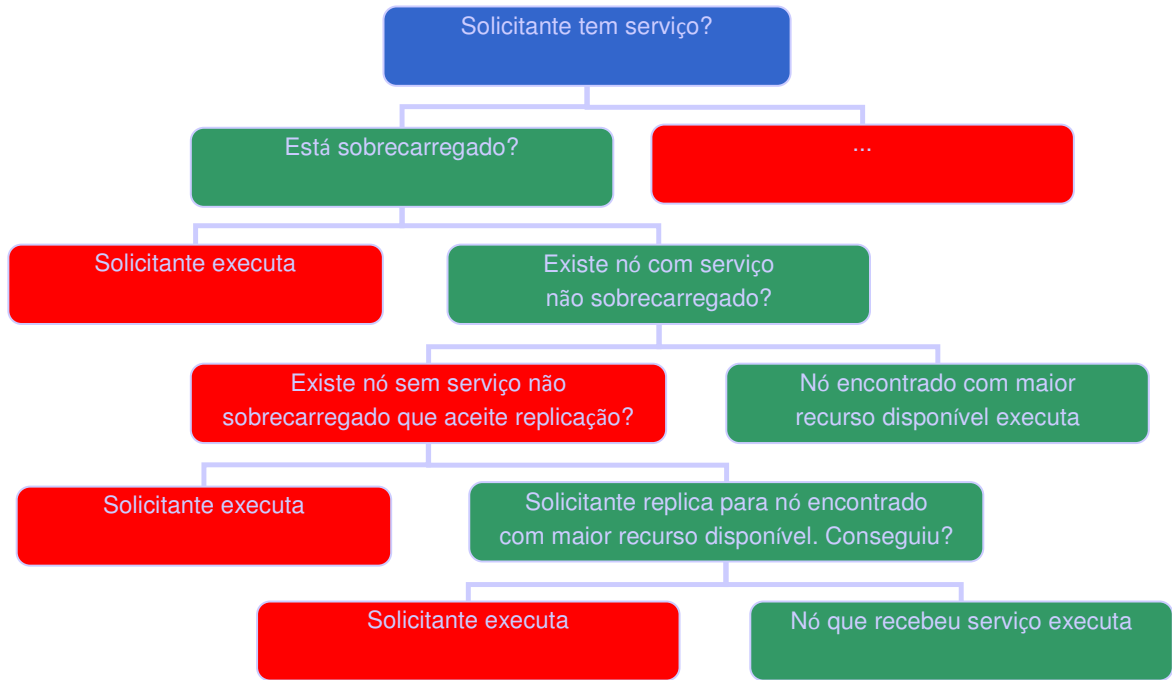


Fig. 1 – Fluxograma do roteamento de serviços (1)

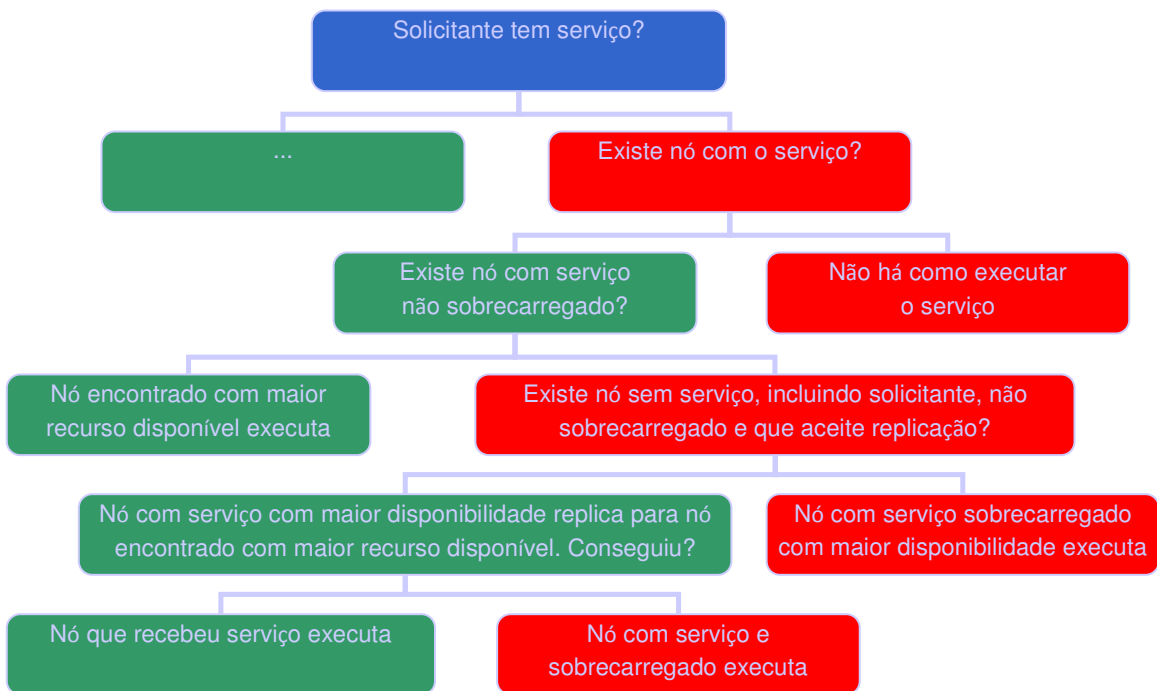


Fig. 2 – Fluxograma do roteamento de serviços (2)

6 Resultados

O ambiente de testes utiliza a biblioteca do Grid-M desenvolvida no “Laboratório de Redes e Gerência” (LRG) da Universidade Federal de Santa Catarina.

Um teste com uma grade de 100 nós simulados em uma mesma máquina apresentou os seguintes resultados:

- Na média, pouco menos de um segundo para o processo de roteamento e execução de um serviço simples sem replicação do serviço.
- Na média, 1,6 segundo para o processo de roteamento e execução de um serviço simples com replicação do serviço, conhecido como o pior caso.

Vale lembrar que a replicação só é necessária uma vez e após isto o nó que recebeu o serviço passará a atender solicitações como conhecedor do serviço também.

O processo sem esta implementação vai variar de caso pra caso, pois não existe um padrão de processamento de tarefas e, portanto, não pode ser previsto com precisão.

A figura 3 mostra, através de dados retirados dos logs do Grid-M como ficou a utilização dos recursos de máquina ao longo do tempo em cada nó de uma grade com quatro nós de nível de recursos igual a 5 e N (limite de recursos livre do nível 1) igual a 90. Neste exemplo os nós 1 e 3 fazem requisições o tempo todo de um serviço que inicialmente somente o nó 2 possui. Em determinado momento este nó fica sobrecarregado (i.e. percentual de recursos livre se torna menor que 18%) e então ocorre uma replicação de serviço do nó 2, que possui o serviço, para o nó 4, que era o nó com mais recursos disponíveis naquele instante e passou a responder requisições do serviço também. Como podemos perceber o algoritmo eliminou eminente saturação do nó 2 e criação de gargalo na rede.

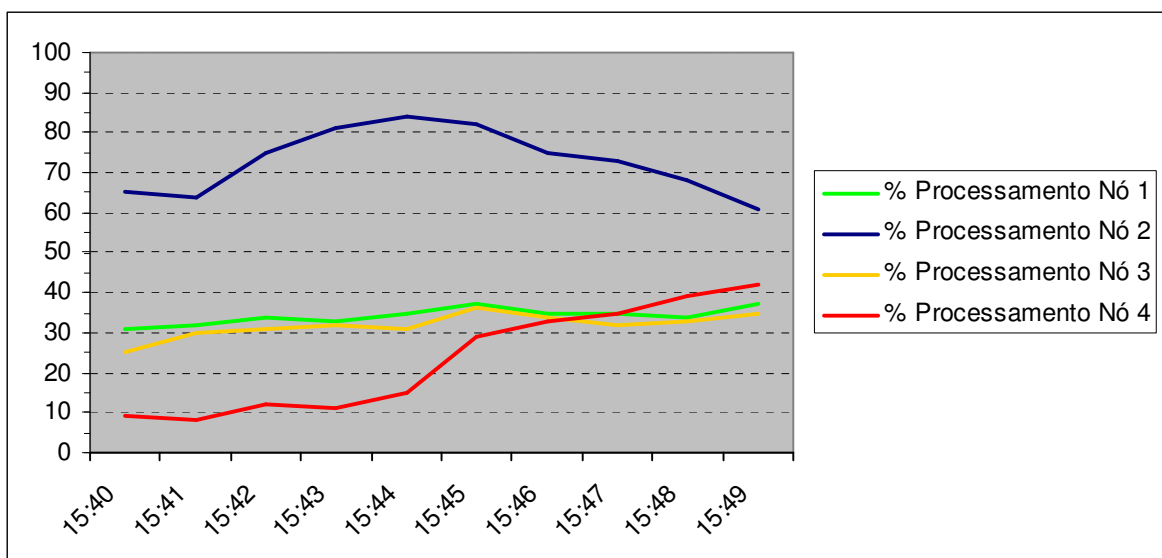


Fig. 3 – Utilização de recursos dos nós e replicação de serviço

7 Conclusão

Os testes constataram que o tempo de processamento do roteamento de serviços (i.e. pesquisa dos nós conectados mais processamento do algoritmo) é muito baixo ou insignificante se comparado aos benefícios oferecidos, estando a performance de uma solicitação de serviço ligada apenas à capacidade de tráfego da rede para a troca de informações entre os nós, o que em redes fechadas não representa ônus devido à sua grande velocidade.

Conclui-se que este controle além de não onerar o tempo de resposta dos serviços e de não aumentar significativamente o tráfego na rede devido aos seus pacotes pequenos, ainda cria uma proteção contra gargalos na rede e saturação de recursos em nós com serviços importantes para a grade. Outro ponto é que em uma grade sem um bom roteamento de serviços o pior caso de execução de um serviço seria percorrer a grade inteira, talvez mais do que uma vez, em busca do serviço. Com o roteamento desta proposta conhecemos o pior caso que pode ocorrer que seria pesquisar informações de controle por todos os nós diretamente conectados ao solicitante, onde este não tem o serviço, todos que o tenham estão sobrecarregados e ocorre então uma replicação de um destes nós sobrecarregados para outro nó sem o serviço e não sobrecarregado. Esta seria a situação mais complexa e mesmo assim sabemos que o custo dela é muito baixo, e ainda que só ocorre quando realmente necessária.

Portanto temos que qualquer custo que esta proposta exija vale a pena tanto para a eliminação de gargalos e eventuais nós saturados na rede bem como para a o tempo de execução do serviço, balanceando as execuções de tarefas entre os nós na medida do possível.

Referências

- [1] DANTAS, M. *Computação distribuída de alto desempenho*. Rio de Janeiro, RJ, BR: Axcel Books do Brasil Editora, 2005.
- [2] FOSTER, I. *Internet Computing and the Emerging Grid*. [S.l.]: Nature Web Matters. [S.l.], 2000. Disponível em: <<http://www.nature.com/nature/webmatters/grid/grid.html>>.
- [3] HERRMANN, K.; MÜHL, G.; GEIHS, K. Self-management: The solution to complexity or just another problem? *IEEE Distributed Systems Online (DSOnline)*, v. 6, n. 1, 2005. Disponível em: <<http://www.vs.uni-kassel.de/publications/2005/HMG05>>.
- [4] HINCHEY, M. G.; STERRITT, R. Self-managing software. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 39, n. 2, p. 107, 2006. ISSN 0018-9162.
- [5] HORN, P. *Autonomic Computing: IBM's Perspective on the State of Information Technology*. Armonk, NY, USA, 2001.
- [6] IBM. *An Architectural Blueprint for Autonomic Computing*. [S.l.], 2005. Disponível em: <<http://www.ibm.com/developerworks/autonomic/library/ac-summary/ac-blue.html>>.
- [7] SKILLICORN, D. B. *Motivating Computational Grids*. [S.l.]: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid, 2002. ISBN 0-7695-1582-7.
- [8] WEISER, M. The computer for the 21st century. *Scientific American*, February 1991. Disponível em: <<http://www.ubiq.com/hypertext/weiser/SciAmDraft3.html>>.