Universidade Federal Fluminense
Instituto de Computação
Graduação em Ciência da Computação

Giulio Dariano Bottari

Gerenciamento de Energia em Aglomerados de Servidores

Niterói-RJ
2011

GIULIO DARIANO BOTTARI

GERENCIAMENTO DE ENERGIA EM AGLOMERADOS DE SERVIDORES

Monografia apresentada ao Curso de Graduação em Ciência da Computação da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Bacharel em Ciência da Computação.

Orientador: Prof. JULIUS C. B. LEITE, Ph.D

Niterói-RJ

2011

GIULIO DARIANO BOTTARI

GERENCIAMENTO DE ENERGIA EM AGLOMERADOS DE SERVIDORES

Monografia apresentada ao Curso de Graduação em Ciência da Computação da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Bacharel em Ciência da Computação.

Aprovada em Julho de 2011.

BANCA EXAMINADORA

Prof. Julius C. B. Leite, Ph.D. – Orientador
IC-UFF

Profa. Anna Dolejsi Santos, D.Sc.
IC-UFF

Prof. Orlando Loques, Ph.D.
IC-UFF

Niterói-RJ
2011

“*The biggest difference between time and space is that you can't reuse time.* ”

— Merrick Furst

“*If you don't know, learn. If you already know, apply. If you know and have applied, improve.* ”

— Anonymous

“*The best way to complain is to make things.* ”

— James Murphy

Agradecimentos

Há momentos chaves na vida de uma pessoa. O momento que culminou na conclusão deste trabalho, ocorreu em Dezembro de 2007. O Prof. Julius chamou a mim e outros dois colegas para conversar após a aula de Organização de Computadores I. Ele nos contou sobre o seu projeto, na área de economia de energia em servidores. Pela minha inexperiência de calouro, aquilo tudo soou vago na época, pois eu não sabia nada sobre o assunto. Por e-mail, pedi um artigo sobre o tema para tentar entender mais sobre o que se tratava (recebi este [9]). Abaixo, reproduzo um trecho do e-mail que enviei sobre o artigo, ao Prof. Julius.

> Eu li o artigo e gostei. Apesar de não entender nada da parte de matemática e sobre como funcionam servidores, eu vi que o trabalho é, como o Sr. mesmo já tinha explicado, sobre os métodos para economizar energia. Apesar de eu não ter a minima noção de como descobrir um jeito novo de medir um QoS (seja lá o que for isso) gostei da ideia de melhorar o gasto de energia, fazendo experiências, aprendendo, testando coisas novas e etc. Ainda mais nessa área.
>
> 07/12/07

É curioso ler essas palavras agora e observar o grande caminho que percorri, chegando até aqui. Mais importante, esse e-mail marca a minha escolha de iniciar um trabalho com o Prof. Julius que culminaria nesta monografia. Ele foi a pessoa que resolveu investir seu tempo em mim, mesmo numa época onde minha capacidade de contribuir com sua pesquisa era limitada. Através dele, aprendi muito além do que apenas conceitos como servidores, economia de energia e QoS. Aprendi a ser independente e buscar as minhas próprias soluções. Aprendi que não devo ser um *torcedor*, como ele diz, buscando ser neutro em relação as minhas próprias propostas. Aprendi a ser atento aos detalhes, mas que *o ótimo é inimigo do bom*. Aprendi que não posso *resolver todos os problemas do universo*, mas que posso começar com alguns. O Prof. Julius é o professor mais importante que

eu já tive. Eu nunca me esquecerei dos valores que aprendi com ele. Eu o agradeço, de verdade.

Eu gostaria de agradecer aos demais professores que encontrei ao longo do curso, que me inspiraram através do seu trabalho, despertando-me para áreas até então invisíveis para mim. Sinto que devo agradecer principalmente a Anselmo Montenegro, Carlos Martinhon, Leonardo Murta e Viviane da Silva, por isso.

Não importa para onde eu olhe no meu trabalho, haverá sempre alguma contribuição de Carlos Sant' Ana. Ao longo desses anos, Carlos se tornou muito mais do que um colega de trabalho. De uma forma sutil ele me adotou como se fosse seu orientando, ajudando-me a superar dificuldades e participando do meu desenvolvimento como pesquisador. Sempre que precisei, Carlos nunca esteve ocupado demais que não pudesse me atender. Eu quero agradece-lo por ser a pessoa que é: um amigo, professor, parceiro e o ser humano mais gentil que conheço.

Outra pessoa que contribui de maneira única para este trabalho e meu desenvolvimento como pesquisador foi Vinicius Petrucci. Ele tem um *approach* leve em relação a vida, com seu humor inteligente e companheirismo. Eu desejaria ter apenas metade da sua capacidade de abstração e inovação, que o tornam um grande pesquisador. Agradeço, profundamente.

Eu não poderia deixar de mencionar aqueles que me acompanharam desde o início do curso: Matheus Erthal e Rodrigo Ferreira. Ao longo desses quatro anos, compartilhamos angústias, risadas, ideias, valores, vitórias e derrotas. Matheus possui um caráter, inteligência e fibra que são únicos. Através dessa convivência, ele me convidou a ver o mundo sobre sua óptica racional e humanista. Rodrigo é uma pessoa carismática, prestativa e que valoriza, acima de tudo, seus amigos. Meu convívio com ele mostrou a importância de valorizar as pessoas e o trabalho em equipe. Obrigado!

Jim Rohn, escritor norte-americano, uma vez disse:

> "*You are the average of the five people you spend the most time with.*"

Se há alguma verdade nessa frase, eu seria a *média* entre estes *seis*: Carlos Sant' Ana, Douglas Mareli, Jefferson Mello, Matheus Erthal, Rodrigo Ferreira e Troy Kohwalter. A convivência com vocês tornou o caminho que me levou a conclusão deste curso em um ladrilho de boas lembranças. Por isso, agradeço.

Ao grande time do Laboratório Tempo, incluindo Alessandro Copetti, Carlos Oliveira, Prof. Orlando Loques e Sergio Carvalho, presto aqui a minha devida homenagem.

Seus trabalhos, coragem intelectual e garra me inspiraram a continuar avançando, frente a todos os obstáculos. Gostaria de agradecer também a Luciano Bertini, pelo legado que nos deixou durante a sua participação no Laboratório. Aos novos membros, Breno Carvalho, Daniel Heráclio e David Barreto, agradeço pelo convívio.

Gostaria de agradecer acima de tudo à minha mãe, Andréa. Ela que me apoiou desde o início, mesmo quando era apenas um garoto que fingia ser um cientista. Sua obsessão com minha educação e formação profissional tornaram este trabalho possível, mesmo em face das adversidades que passamos. Este trabalho eu dedico a você, da mesma forma que você dedicou seu carinho, amor e paciência à mim, durante toda a minha vida.

Por fim, gostaria de agradecer aos leitores desta monografia, pelo seu tempo e interesse no meu trabalho.

Lista de Figuras

Lista de Tabelas

Glossário

ACPI	:	Advanced Configuration and Power Interface
ADD	:	Alocação Dinâmica de Desempenho
CDS	:	Configuração Dinâmica de Servidores
CPD	:	Centro de Processamento de Dados
CPU	:	Central Processing Unit
DAQ	:	Data Acquisition System
DVFS	:	Dynamic Voltage and Frequency Scaling
ESSenCe	:	Energy-aware System for Server Clusters
FID	:	Frequency Identification
HTTP	:	Hyper-Text Transfer Protocol
HVAC	:	Heating, Ventilation & Air Conditioning
IP	:	Internet Protocol
LAMP	:	Linux, Apache, MySQL & Perl/Python/PHP
PHP	:	PHP: Hypertext Preprocessor
PID	:	Proportional Integral Derivative
QoS	:	Quality of Service
TCP	:	Trasnsmission Control Protocol
TI	:	Tecnologia da Informação
UPS	:	Uninterruptible Power Supply
VID	:	Voltage Identification
W3C	:	World Wide Web Consortium

Sumário

Resumo

A demanda energética em aglomerados de servidores *Web* vem se tornando um problema, tanto sobre uma perspectiva econômica quanto ambiental. Por esse motivo, mecanismos de controle de potência vêm sendo desenvolvidos para sistemas computacionais, como controle de frequências de processadores e estados de baixo consumo de potência.

A proposta deste trabalho é uma política de gerenciamento de energia para a camada de aplicação de um aglomerado *web* heterogêneo. A abordagem adotada trata de maneira independente o chaveamento de frequências e servidores. Para configuração de frequências foi desenvolvido um algoritmo que apresenta a configuração mais energeticamente eficiente para o problema. Já para a configuração de servidores, escolheu-se uma abordagem heurística baseada em uma análise do perfil de desempenho e potência das máquinas estudadas. A proposta demonstrou-se de implementação simples e escalável, permitindo a otimização em tempo real do consumo de energia. Além disso, a política desenvolvida é capaz de oferecer um balanceamento entre qualidade de serviço e economia de energia.

A implementação da proposta foi realizada sobre um sistema que possui uma infraestrutura reusável, possibilitando um estudo em alto-nível de políticas de economia de energia, ao apresentar uma abstração dos detalhes de implementação comuns das mesmas. O sistema conta também com um modo de simulação, que possibilita o estudo dessas políticas em ambientes diferenciados.

Palavras-chave: Economia de Energia, Computação Verde, Aglomerados de Servidores *Web*, Gerenciamento de Energia, Qualidade de Serviço, Configuração Dinâmica, DVFS.

Abstract

The energy consumption in Web server clusters is becoming an issue, both from an economical and an environmental perspective. For this reason, power control mechanisms have been developed for computer systems, such as processor frequency control and low power states.

This work's proposal consists of an energy management policy for the application layer of a heterogeneous web cluster. The chosen approach deals with frequency and server switching on an independent basis. For frequency switching it has been developed an algorithm that finds the most power efficient configuration. As for server switching, it was adopted an heuristic approach based on the analysis of each machine's power and performance profile. It has been shown that the proposal is easily implemented and scalable, allowing for energy consumption optimization in execution time. Moreover, the developed policy is able to offer a balance between quality of service and energy savings.

The implementation of the proposal was conducted on a system with reusable infrastructure, allowing for a high-level testing of energy saving policies, by presenting an abstraction from their common implementation details. The system also presents a simulation mode, which makes it possible to study these policies on different environments.

Keywords: Energy Saving, Green Computing, Web Server Clusters, Energy Management, Quality of Service, Dynamic Configuration, DVFS.

Capítulo 1

Introdução

O sucesso da Internet foi um evento marcante na história da humanidade. Essa rede de computadores, que se iniciou com a proposta de interligar algumas universidades norte-americanos, hoje é uma tecnologia quase onipresente na vida do homem moderno. Nos últimos anos, a Internet se tornou uma revolução dentro de uma revolução, ampliando a sua gama de aplicações cada vez mais. Hoje, é possível realizar virtualmente qualquer tarefa feita tradicionalmente num sistema operacional através de um navegador *Web*. Esse paradigma vem inspirando propostas ousadas, como um sistema operacional onde não há aplicações nativas, somente integração com serviços e aplicativos da *Web* [19]. A tendência é que a *Web* continue evoluindo, tornando-se cada vez mais presente na nossa experiência cotidiana.

Para sustentar o avanço da Internet é necessário que sua infraestrutura também cresça. Isso levou ao desenvolvimento de grandes Centros de Processamento de Dados (CPDs), ou *datacenters*. Esses locais abrigam sistemas computacionais para processamento e armazenagem de dados, bem como uma grande infraestrutura de suporte. O processamento de requisições é feito por aglomerados (ou *clusters*) de servidores interconectados, trabalhando cooperativamente. A infraestrutura de suporte inclui dispositivos de alimentação ininterrupta (UPS), aparatos de resfriamento e ventilação (HVAC) e sistemas de segurança, como anti-incêndio.

Com o crescimento dos *datacenters*, uma das principais preocupações recaiu sobre a sua demanda energética. Esforços vêm sendo feitos ao redor do mundo para tornar CPDs cada vez mais energeticamente eficientes. O *Open Compute Project* [17], liderado pelo Facebook, abriu a especificação de alguns de seus servidores e *datacenters*, especialmente criados com o objetivo de eficiência, para a comunidade científica mundial. *Designs* mais

eficientes em CPDs são um requisito de primeira ordem para viabilizar o crescimento desses centros.

Em [8] são apontados três motivadores principais para o estudo de economia de energia em *datacenters*: competitividade, escalabilidade e aquecimento global. A seguir, serão abordados cada um destes aspectos.

Competitividade. Evidentemente, o gasto em energia também contribui para o custo operacional de um *datacenter*. Sendo assim, a eficiência energética é, sobretudo, uma questão econômica. Empresas que adotam técnicas para explorar essa eficiência se sobressaem em relação aos seus concorrentes, uma vez que têm condições de oferecer um preço final mais atraente para seus clientes ou outras vantagens competitivas.

Escalabilidade e eficiência energética estão firmemente conectadas, uma vez que custo de manutenção dos equipamentos de TI pode tornar proibitivo a expansão de um *datacenter*. Algumas empresas ao redor do mundo simplesmente não possuem capacidade energética para suportar o crescimento requerido pela área de TI, mesmo quando esses avanços resultariam em dividendos para o seu negócio [12]. Devido a crescente demanda de processamento, a escalabilidade é um requisito fundamental na maioria dos CPDs.

Aquecimento Global. O consumo desenfreado de energia vem trazendo sérias consequências ao meio ambiente. Em vários países, grande parte da matriz energética é sustentada pela queima de combustíveis fósseis como petróleo, gás natural e carvão. Essa queima libera gases do efeito estufa, como gás carbônico e metano, que são responsáveis pelo aquecimento global.

A Figura 1.1[1] mostra previsões feitas para o aumento da temperatura no período de 2070 a 2100. Pode-se observar que vários países serão afetados severamente, incluindo a parte norte do Brasil. Uma consequência dessa mudança climática inclui a maior frequência de secas e enchentes que podem prejudicar a produção rural. Além disso, idosos, portadores de asma e problemas cardíacos são mais sensíveis a ondas de calor provocadas por esse fenômeno, diminuindo a qualidade de vida dessas pessoas [16].

A motivação para reduzir o consumo de energia em sistemas computacionais levou à criação da *Advanced Configuration and Power Interface* (ACPI) [1]. Trata-se de uma

[1]Imagem retirada de `http://en.wikipedia.org/wiki/Global_warming`.

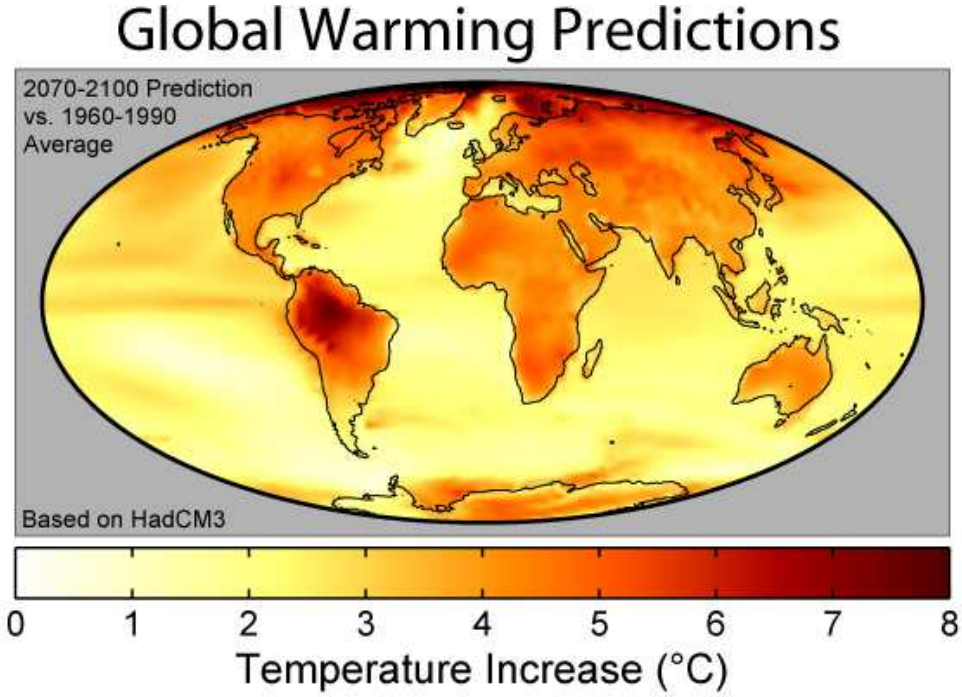


Figura 1.1: Previsões para o Aquecimento Global

especificação para o gerenciamento energético de computadores através do conceito de estados de energia. Um sistema computacional pode transitar para estados de baixo consumo através de diversos mecanismos. Pode-se variar sua frequência de operação, desligar dispositivos e partes do processador, como memória *cache*.

A Tabela 1.1 mostra uma listagem destes estados e sub-estados. Os estados `G0` e `G1` são chamados de estados globais. De maneira geral, no estado `G0` todos os componentes do sistema estão ligados, enquanto que no estado `G1` o sistema desliga algum componente (indicado por `S1`, `S2` e demais). O processador tem seus próprios estados de energia, indicados pela letra `C` (CPU). No estado normal de operação (`C0`), o processador pode variar frequências e voltagem para formar os chamados estados de performance, ou *P-States*. Os demais *C-States* indicam que algum componente interno da CPU foi desligado, como níveis de *cache* (L1, L2 e etc) e núcleos (*cores*). A medida que se avança nesses estados, a latência para retornar ao `C0` é maior, na ordem de microssegundos. Analogamente, transitar de algum sub-estado de `G1` para `G0` representa um atraso, mas na ordem de segundos.

A arquitetura de um aglomerado de servidores é frequentemente dividida em três camadas lógicas (ou *tiers*), como na Figura 1.2[2]. A primeira camada é responsável por distribuir carga entre os servidores de aplicação e encaminhar a resposta para os clientes,

[2]Imagem retirada de [34].

Tabela 1.1: Estados ACPI

G0	C0	P0	Estados de performance (processa dados).
		P1	
		⋮	
		Pn	
	C1		Estados de *sleep* da CPU (não processa dados).
	C2		
	⋮		
	Cn		
G1	S1		CPU para de executar instruções.
	S2		CPU é desligada.
	S3		*Standby*. Contexto é salvo em memória RAM.
	S4		Hibernação. Contexo é salvo em memória secundária.

através da Internet. A camada 2 possui a implementação da aplicação, consultando os servidores de dados na camada 3 quando necessário. Este trabalho se concentrará em prover uma solução de economia de energia para a camada 2. Por simplicidade, será desconsiderada a camada 3, para evitar problemas de disponibilidade e consistência do banco de dados.

Em resumo, a proposta deste trabalho é explorar estados de baixo consumo de potência para servidores da camada 2. Para tanto, foram desenvolvidas duas políticas: CDS: Configuração Dinâmica de Servidores e ADD: Alocação Dinâmica de Desempenho. O sistema utiliza heurísticas para minimizar o consumo energético dos servidores e manter conformidade à uma QoS (*Quality of Service*) previamente acordada. Para implementar essas políticas foi desenvolvido um sistema de nome ESSenCe: *Energy-aware System for*

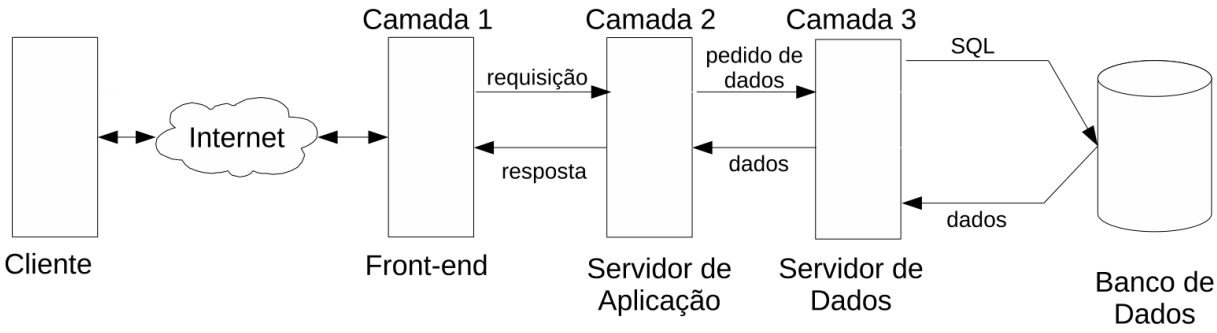


Figura 1.2: Arquitetura de três camadas

Server Clusters. Esse sistema consiste de um ambiente de alto-nível, extensível e de fácil uso para economia de energia em aglomerados de servidores.

Através dos experimentos conduzidos, obteve-se uma economia de cerca de 50% da energia consumida quando comparada a um modo que não emprega nenhuma política de gerenciamento de potência. Acompanhado desse ganho energético, o sistema proposto foi capaz de manter uma QoS acordada de 95%. Comparou-se também o trabalho desenvolvido com o método `ondemand`, um controlador de frequências largamente usado em sistemas Linux. Na maioria dos casos, a abordagem proposta de frequências dinâmicas se mostrou superior em relação ao `ondemand`. Além disso, essa política se demonstrou bastante escalável, sendo capaz de processar centenas de servidores em menos de um segundo.

O restante desta monografia foi separada em outros seis capítulos. No Capítulo 2 são apresentados trabalhos anteriores que viabilizaram o desenvolvimento da proposta. O Capítulo 3 discute o modelo que sustenta a proposta do trabalho, ilustrando em alto-nível como é realizado o controle energético do *cluster*. No Capítulo 4, é detalhada a topologia do ambiente de testes e ferramentas usadas para validar a proposta. Continuando, o Capítulo 5 apresenta a implementação e organização do sistema ESSenCe, bem como o funcionamento de outras ferramentas importantes. Já o Capítulo 6 discute a metodologia e os resultados dos experimentos realizados pelo sistema ESSenCe no ambiente de testes. Por fim, o Capítulo 7 concluirá este trabalho com uma avaliação do mesmo e uma apresentação de áreas que poderão ser exploradas em trabalhos futuros.

Capítulo 2

Trabalhos relacionados

Devida a relevância do tema de economia de energia em aglomerados de servidores, vários trabalhos desenvolveram propostas para o problema. Neste Capítulo será apresentada uma breve revisão da literatura sobre os trabalhos mais próximos da proposta desta monografia e a suas contribuições principais.

Uma gama de trabalhos [9, 15, 29, 33] observa que o problema da eficiência energética em CPDs pode ser simplificada com algumas premissas. A mais comum é que a carga em um CPD é baixa a maior parte do tempo, em relação à sua capacidade máxima. Assume-se também que páginas *Web* estão carregadas em memória principal, portanto, poucos acessos à memória secundária são necessários. Sendo assim, a maior parte do processamento de servidores *Web* é intensivo em CPU ou memória. Contudo, a maioria dos estudos apresentados neste Capítulo considera o processador como gargalo principal do sistema.

Enquanto que em servidores *Web* aplicações intensivas em CPU são mais preocupantes, não é possível estender essa perspectiva para outros ambientes, como sistemas embarcados. Em [38] há uma crítica a trabalhos que ignoram a energia gasta por outros componentes, como memória. O estudo mostra que há uma forte dependência entre o efeito de DVFS (*Dynamic Voltage and Frequency Scaling*) e o tipo de carga de trabalho sobre a qual o sistema é submetido. O trabalho demonstra que em cargas intensivas em memória, a frequência de operação que minimiza a energia gasta não é a mais baixa possível, como seria intuitivo pensar no caso CPU intensivo. Devido a essas características, uma política mais geral foi criada em [37], chamada de *generalized energy delay policy*. Tal política é ampla o suficiente para criar um compromisso entre performance e energia para diversos tipos de carga de trabalho. Simultaneamente, seu controle é simples, pois é confi-

gurável a partir de um único parâmetro. Entretanto, a manutenção desse sistema depende de uma série de *benchmarks* que consultam PMCs (*Performance Monitoring Counters*) do processador para estimar a degradação de performance do sistema. Essa tarefa é complexa e não escalável, pois requer um conhecimento de baixo-nível da arquitetura de cada processador analisado.

Em [15] foi feito um levantamento das técnicas mais comuns para eficiência energética em *clusters* de servidores, nomeadas VOVO (*Vary-On/Vary-Off*), IVS (*Independent Voltage Scaling*) e CVS (*Centralized Voltage Scaling*). A primeira consiste de desligar ou ligar servidores de acordo com a demanda. Essa técnica acumula a maior percentagem de economia de energia dentre as demais (de 22% a 42%), segundo o estudo. Em compensação, a latência de chaveamento de um servidor é considerável, e pode incorrer em saturação do aglomerado. Seguindo, são discutidos dois tipos de controle de frequência: IVS e CVS. O primeiro consiste de um controle independente, onde cada estação decide em qual frequência trabalhar. Um exemplo comum desse tipo de controle é o *governor* `ondemand` [27], presente na maioria das distribuições Linux. No método CVS o controle de frequências é coordenado por uma entidade centralizada, que determina a frequência de operação de cada servidor. O estudo, que consiste de simulações, concluiu que os melhores resultados são obtidos através da combinação entre VOVO e CVS.

Assim como no trabalho anteriormente citado, este trabalho também compara um método de controle independente de frequências (o `ondemand`) e um esquema centralizado proposto (ver Capítulo 6). Tal comparação é necessária para justificar o custo e *overhead* decorrentes de se utilizar uma política centralizada de controle de frequências.

Dois tipos de políticas são comuns na literatura, chamadas muitas vezes de *race-to-idle* e *just-in-time*. A primeira parte do princípio que a melhor estratégia é terminar o processamento o mais rápido possível para maximizar o tempo que o sistema permanece ocioso, em estados de baixo consumo. Projetos como [25] exploram essa ideia ao implementarem um dispositivo para transitar rapidamente entre o estado ocioso e ativo. Porém, essa estratégia não é frequentemente aplicada para perfis de carga onde há um fluxo ininterrupto de trabalho, como em servidores *Web* de tráfego intenso. Nesses casos, os intervalos ociosos são muito curtos, sendo difícil aproveitá-los em estados de energia reduzida.

A política *just-in-time* procura estender a finalização de tarefas o mais próximo possível do fim do seu prazo final (*deadline*). Para realizar esse controle, técnicas de variação de frequência são empregadas para obter um compromisso entre economia de

energia e cumprimento dos prazos estabelecidos. Um trabalho que abordou esse problema é [8]. Utilizando um controlador PID e o mecanismo de DVFS, obteve-se um controle fino do percentil de atendimento dos prazos. Para realizar esse controle utilizou-se uma variável chamada *tardiness*, que representa a fração entre o tempo de atendimento e o prazo de cada requisição. Através do controle dessa variável é possível convergir a QoS para um valor contratado. Em contrapartida, a proposta deste trabalho não realiza o controle direto do QoS. Ao invés disso, utiliza-se uma variável chamada saturação, descrita no Capítulo 3. Ao longo desse Capítulo será argumentado que através do controle de saturação é possível obter um nível adequado de QoS.

Um dos grandes problemas que se deve contornar para gerenciar políticas de chaveamento de servidores é a latência de transição. Uma das formas de se atacar esse problema é através da previsão de carga. Talvez um dos primeiros trabalhos que introduziram técnicas de previsão de carga para *clusters* de servidores foi [6]. Nesse trabalho discute-se um previsor linear simples, capaz de notificar com antecedência picos de carga (*hotspots*). Desde então, outros trabalhos como [35, 36] exploraram a previsão de carga para auxiliar o processo de configuração dinâmica de servidores. Através do Previsor Linear de Holt e uma variável de superestimação, esses trabalhos obtiveram melhores ganhos no atendimento de requisições comparados com políticas puramente reativas. Procurando explorar essa ideia, será proposto um mecanismo de antecipação de carga que será discutido em detalhes na Seção 3.1.3.

Trabalhos iniciais como [15] lidavam com a premissa de que os servidores possuem o mesmo *hardware*, ou seja, são homogêneos. Contudo, foi apontado em [5] que é comum haver máquinas de diversas gerações exercendo as mesmas funções em um *datacenter*. Esse problema foi atacado em [33], onde uma metodologia para abordar máquinas heterogêneas foi formulada. O trabalho utiliza um *front-end* que coordena a distribuição de carga, frequências de operação e quantidade de nós ativos no aglomerado. A técnica, que se baseia em heurísticas, foi capaz de reduzir o consumo energético com uma degradação de performance mínima. Além disso, um dos pontos fortes da proposta é a sua validação em um *cluster* físico, ao invés de simulações. Assim como o trabalho apresentado, a proposta da monografia foi validada num ambiente de testes real (ver Capítulo 4), que consiste de um aglomerado heterogêneo. Contudo, simulações realistas também são possíveis através do sistema proposto (ver Seção 5.4.5).

Percebendo a grande complexidade envolvida em políticas de gerenciamento energético em *clusters Web*, [28] desenvolveu um *framework* para facilitar a descrição e aplica-

ção dessas políticas. Tal *framework* foi construído de forma a aumentar o foco no *design* das políticas de energia, abstraindo detalhes de baixo nível. A proposta é baseada em uma arquitetura de componentes que podem ser reusados em outras políticas de configuração de servidores. Em conjunto com essa infraestrutura, foi elaborada uma linguagem para descrição de contratos que possibilita especificar adaptações customizadas em alto nível, diminuindo os custos de desenvolvimento dessas políticas.

O sistema desenvolvido para avaliar a proposta da monografia (ESSenCe) foi fortemente influenciado pelas ideias do *framework* apresentado. Assim como ele, o trabalho proposto permite que sejam desenvolvidas políticas independentes que compartilham uma mesma variável do sistema, (e.g. requisições por segundo). Na abordagem de [28] essas políticas são especificadas através de contratos. Apesar do trabalho proposto não ser orientado a contratos, foi criada uma infraestrutura que permite a implementação futura de outros requisitos não explorados por este trabalho, como controle de temperatura e virtualização.

2.1 Sumário

Neste Capítulo foram abordados os principais trabalhos nos quais a proposta se baseia. Foram vistas técnicas comuns usadas para o controle energético em *clusters* de servidores. Além disso, abordou-se a importância de usar um ambiente de testes que seja representativo de um *cluster* real, como um aglomerado *Web* heterogêneo.

Com base nesses trabalhos será apresentada a política proposta de economia de energia no próximo capítulo. Nele, será visto a base teórica do funcionamento da proposta, comparando-a com trabalhos desenvolvidos no passado.

Capítulo 3

Proposta

A proposta deste trabalho é apresentar um conjunto de políticas simples e escaláveis para economia de energia em *clusters* de servidores. A ideia básica desse sistema é viabilizar, de uma forma controlada, um compromisso entre performance e redução de energia do aglomerado. Para tanto, este sistema se baseia em dois mecanismos básicos para economia de energia: Configuração Dinâmica de Servidores e Alocação Dinâmica de Desempenho, que serão chamados também de CDS e ADD, daqui em diante. Esses mecanismos se baseiam em chaveamento de servidores e frequências, respectivamente. A sua atuação é feita de forma centralizada em um servidor, como o próprio *front-end*.

Em primeiro lugar, deve-se esclarecer o que se entende por *performance*. O desempenho de um servidor será definido como a sua **vazão máxima** de requisições: a quantidade máxima de pedidos que pode atender, por unidade de tempo. Apesar dessa grandeza variar com o tipo de requisição e a capacidade de processamento do nó, não se discutirão esses problemas até o Capítulo 4. O leitor deve atentar para o fato que a referida vazão máxima é uma medida de performance, enquanto que a vazão de um servidor num dado instante pode ser inferior ao valor máximo, já que a mesma depende da carga no instante em que se observa o sistema.

Considerando a vazão como sendo a métrica de desempenho desse sistema, é necessário especificar de que modo esse recurso impacta a qualidade de serviço oferecida aos usuários. Deseja-se controlar a degradação no serviço em função da vazão de atendimento que o usuário receberia se não houvesse nenhuma política de energia ativa. Para tanto, será definida como **saturação** a razão entre a carga do sistema e a vazão máxima de requisições da configuração, $carga \div vazão_{máx}$. Quando esta relação é menor do que 1, considera-se que uma qualidade de serviço adequada está sendo oferecida. Porém, quando

a saturação se aproxima do valor 1, não há vazão suficiente para atender a carga. Nessa situação, é considerado que a experiência do usuário está sendo degradada. [1]

Diferentemente de outros trabalhos como [30, 34], o sistema desenvolvido defende a atuação independente entre o chaveamento de frequências e servidores. Nesses trabalhos são utilizados modelos de programação inteira mista para buscar uma configuração que resulte na eficiência energética máxima para uma dada carga do sistema. Contudo, é possível subdividir esse problema em duas partes independentes. Primeiro, define-se o conjunto de servidores mais adequados para atender um nível de requisições. É observado o *overhead* de chaveamento desses servidores e trata-se a latência de adaptação do sistema, como descrito na Seção 3.1. Dado esse conjunto de servidores, procura-se um arranjo de frequências de operação que atenda a carga e minimize a potência consumida pelo sistema, como explicado na Seção 3.2.

Como será visto nas seções seguintes, CDS e ADD possuem tempo de atuação em ordens de grandeza distintas. Caso a atuação das duas políticas estivesse submetida ao mesmo período de tempo, a política ADD seria penalizada. A razão disso é que dentro de um intervalo de atuação da CDS é possível ajustar a frequência várias vezes para acompanhar a evolução da carga de trabalho. Acredita-se que é possível explorar o intervalo mais curto de configuração de frequências para economizar mais potência que métodos baseados em otimização.

Outra característica importante da proposta é a sua fácil manutenção a medida que mudanças ocorrem no aglomerado. Por exemplo, caso uma máquina apresente uma falha, basta remover o seu representante do *pool* de máquinas de entrada dos algoritmos CDS e ADD. Analogamente, a adição de um novo servidor é feita de maneira simplificada, dado que o perfil do novo servidor seja obtido através de uma metodologia como a descrita na Seção 4.5. Isso é possível porque os algoritmos não guardam nenhum estado interno e otimizam o *cluster* em tempo de execução.

Os algoritmos que serão apresentados exigem um esforço computacional muito baixo. Por conta disso, podem ser empregados para *clusters* de muitos servidores, como será visto nos experimentos da Seção 6.5. Implementações que dependem de *solvers* de programação inteira mista tem sua escalabilidade comprometida, dado o tempo necessário para se resolver o problema de maneira ótima em tempo de execução.

[1] Uma premissa que se assume é que o aglomerado é dimensionado de maneira ideal em relação à carga. Em outras palavras, sempre será possível atender requisições com saturação menor do que 1, já que as mesmas não excedem a capacidade máxima do *cluster*.

Por conta da independência entre CDS e ADD, pode-se aplicar cada método em conjuntos de servidores distintos. Por exemplo, todos os servidores podem estar sujeitos à política ADD, enquanto apenas um conjunto menor sofre a atuação de CDS. Isto pode ser útil para isolar máquinas que precisam estar sempre disponíveis. Essa possibilidade é interessante, mas foge ao escopo deste trabalho.

3.1 Configuração dinâmica de servidores

O objetivo da CDS é obter um conjunto de servidores que seja energeticamente eficiente para atender um determinado nível de carga. Porém, em um *cluster*, a quantidade de requisições varia em relação ao tempo. Sendo assim, é necessário configurar o aglomerado de maneira dinâmica para acomodar as oscilações de carga. Para tanto, os servidores transitam entre dois estados: *ativo* e *inativo*. Servidores *ativos* pertencem ao conjunto da configuração atual, enquanto que os *inativos* são colocados em um estado de baixo consumo energético e não são capazes de atender requisições. Para mais detalhes sobre esses estados, consulte a Seção 4.2.2.

Uma característica importante da CDS é que o chaveamento de servidores é um processo demorado, durando cerca de 6 segundos usando Suspend-to-RAM (detalhado na Seção 4.2.2). Esse atraso pode gerar uma saturação do aglomerado e uma subsequente perda da qualidade de serviço. Por conta disso, procura-se evitar ao máximo uma subestimação. Uma subestimação ocorre quando uma configuração escolhida pela política CDS não consegue prover a vazão requerida pela aplicação. Para contornar este problema, dois mecanismos foram implementados: reconfiguração conservativa e antecipação de carga. O primeiro determina que somente um servidor, no máximo, será ligado ou desligado por iteração do algoritmo CDS. Já o segundo mecanismo procura reconfigurar o aglomerado antes que a sua saturação ocorra. Esse último será estudado em detalhes na Seção 3.1.3.

3.1.1 Modelo

Para desenvolver a política CDS é necessário desenvolver um modelo de potência para os servidores. Um modelo suficiente para requisições intensivas em CPU considera a potência que um servidor gasta na forma descrita pela Equação 3.1. O termo $P_{estática}$ é constante e denota a potência mínima gasta pelo servidor quando nenhum trabalho é realizado. Já a parcela $P_{dinâmica}$ varia em função da vazão (V) que o servidor desempenha. A Seção 4.5

explica como são obtidos valores para essas variáveis. Um servidor pode possuir diversas frequências de operação implementadas no seu processador. Cada uma está associada a um nível de desempenho diferente. Porém, para CDS será considerada a vazão da frequência máxima.

$$P_{servidor}(V) = P_{estática} \ + \ P_{dinâmica}(V) \tag{3.1}$$

Uma característica importante desse modelo é que $P_{estática}$ representa uma fração maior da potência total de um servidor do que $P_{dinâmica}$. Para as máquinas do ambiente de testes, todas apresentam uma relação onde $P_{dinâmica} < P_{estática}$, exceto o servidor `ohm`. Uma vez que a potência estática contribui tanto para a potência total de um servidor, uma heurística simples é minimizar a quantidade de servidores ativos no aglomerado, como em [31].

3.1.2 Algoritmo

O algoritmo apresentado não é necessariamente uma solução ótima para o problema de chaveamento de servidores. Ao invés disso, seu foco é em escalabilidade e simplicidade, como foi argumentado anteriormente.

O Algoritmo 1 resume o funcionamento da política proposta. Esse algoritmo deve ser executado em *loop*, por um período que será detalhado mais adiante. A entrada do algoritmo consiste de um valor de carga ($load$) e uma lista de servidores representando a configuração atual ($currentConfig$). A variável $load$ deve ser um representante de uma janela de tempo de tamanho igual ao período do *loop*, como uma média dos valores. Neste trabalho foi escolhida uma média aritmética simples de amostras dessa janela, onde cada amostra é medida por segundo.

Na linha 1 determina-se a necessidade de ativar um servidor. Para tanto, a carga atual deve ser maior do que a vazão propiciada pela configuração atual. O parâmetro γ tem a função de antecipar a ativação de um servidor e será detalhado na Seção 3.1.3.

Para determinar qual máquina será ativada ou desativada, utiliza-se uma lista ordenada e fixa de servidores ($referenceList$). Elementos do início dessa lista são consultados primeiro para ativação, enquanto que elementos do fim tem prioridade de desligamento. É possível que um servidor do início da lista não seja capaz de atender completamente a demanda. Nesse caso, procura-se o servidor seguinte da lista e assim sucessivamente. Caso nenhum seja capaz de suprir a demanda, o servidor mais robusto (que pode oferecer

Algoritmo 1 Atuação da configuração dinâmica de servidores

```
1: if load > throughput(currentConfig) · γ then
2:    {Obtém o servidor mais eficiente que, uma vez ligado, será capaz de suprir a carga.}
3:    candidates ← (referenceList − currentConfig)
4:    for server in candidates do
5:       newConfig ← (currentConfig ∪ server)
6:       if throughput(newConfig) > load · γ then
7:          return newConfig
8:       end if
9:    end for
10:   {Nenhum servidor é capaz de suprir load, então o mais robusto é escolhido.}
11:   return currentConfig ∪ mostRobustServer(candidates)
12: else
13:   {Procura-se reduzir o número de máquinas que estão ligadas.}
14:   candidates ← reverse(referenceList) ∩ currentConfig
15:   for server in candidates do
16:      newConfig ← (currentConfig − server)
17:      if throughput(newConfig) · γ > load then
18:         return newConfig
19:      end if
20:   end for
21:   return currentConfig
22: end if
```

a maior vazão) disponível é escolhido.

O critério usado para escolha da lista visa minimizar o número de reconfigurações, ordenando-a pela vazão máxima de cada servidor. Ligando primeiro os servidores mais robustos evita-se que outros menos poderosos, que geralmente oferecem uma baixa taxa de performance por Watt, sejam escolhidos. Isso se justifica uma vez que a criação de novas arquiteturas de processadores vem trazendo uma maior eficiência por Watt, aliada a um aumento da velocidade de processamento.

Para avaliar a proposta apresentada não é necessário que a ordenação dos servidores em $referenceList$ resulte na configuração mais eficiente possível. Na realidade, a ordenação pode atender a outro critério de interesse. Por exemplo, a lista poderia ser

ordenada pelo tempo de chaveamento de cada servidor. O importante é que o algoritmo garante que os servidores serão ligados ou desligados de acordo com a demanda.

Se a condição da linha 1 for falsa, tenta-se reduzir o conjunto de servidores ativos. Percorre-se a lista de servidores na ordem reversa para desativá-los. Um servidor será desativado somente se a vazão da configuração resultante for maior que a carga corrente. Caso contrário, a configuração permanecerá a mesma.

No pior caso serão percorridos $N-1$ servidores na lista de candidatos, pois garante-se que sempre haverá pelo menos um servidor ligado quando a condição da linha 1 é verdadeira. Se a condição for falsa, N servidores candidatos serão analisados, na pior das hipóteses, quando todos os servidores encontrarem-se ativos. Assim, o Algoritmo 1 apresenta uma complexidade da ordem de $O(N)$, onde N é o número de servidores do aglomerado. O tempo de criar a lista ordenada de servidores não é contabilizado, pois $referenceList$ é dada como entrada do problema e permanece fixa durante o restante do experimento.

Uma vez definido o algoritmo CDS, é necessário especificar ainda o período de atuação da política. Caso o tempo de atuação seja muito curto, o algoritmo irá reconfigurar o *cluster* em presença de pequenas variações de carga. No pior caso, chaveamentos seguidos podem ocorrer, onde uma operação de ativação é seguida de uma desativação ou vice-versa. O período de atuação não deve ser muito longo, caso contrário a responsividade do sistema ficaria prejudicada em relação às oscilações da carga. Neste trabalho, o período do *loop* de execução foi escolhido como 20 segundos. Esse valor mostrou-se adequado em testes realizados em simulação para a determinação dessa variável.

3.1.3 Antecipação de carga

Devido às variações de tráfego em um *Web cluster*, uma das questões que devem ser pensadas quando se projeta uma política como a CDS é o erro de configuração. Esse erro pode ocorrer quando sobra desempenho no aglomerado. Nesse caso, o problema é que a política poderia ser mais energeticamente eficiente. Ainda, pode-se subestimar configurações de servidores. O problema então passa a ser a perda de qualidade de serviço, pois o aglomerado se encontra saturado de requisições. Apesar de ambos os problemas merecerem atenção, o segundo é mais grave, pois uma das premissas do sistema proposto é oferecer uma qualidade de serviço próxima daquela que seria obtida caso não houvesse nenhum gerenciamento energético ativo.

Em [6, 13, 34] foram propostos mecanismos de previsão de carga para *clusters* de servidores *Web*. A função principal de tal mecanismo é auxiliar o algoritmo CDS, de três maneiras. Em primeiro lugar, evitando erros de configuração de servidores através da análise da *tendência* da carga. Por exemplo, é possível que a carga encontre-se em um nível baixo de requisições, mas com uma tendência positiva. Essa informação extra oferecida pelo previsor pode levar a não desativar um servidor do *cluster*, já que o número de requisições tende a aumentar no próximo intervalo. Em segundo lugar, esse mecanismo muitas vezes antecipa elevações na carga, permitindo a ativação de servidores antes que o aglomerado se torne saturado. Esse último efeito é especialmente útil, tendo em vista que o tempo de ativação de um servidor é alto. Em terceiro lugar, o uso de previsores evita chaveamentos desnecessários de máquinas. Isso é possível pois o previsor é capaz de filtrar *picos* de carga que ocorrem no sistema. Esses picos consistem de grandes variações de carga que ocorrem em intervalos curtos de tempo. Em certas ocasiões, esses intervalos podem ser menores que o tempo necessário para aplicar uma configuração no aglomerado. Esses casos são considerados imprevisíveis e o melhor a se fazer é ignorar o chaveamento de servidores, evitando o desperdício de energia.

Em [11] comparou-se a utilização de duas políticas de antecipação de reconfiguração de servidores. A primeira delas é definida pelo previsor de nome Método Linear de Holt [24], explorada extensivamente em [34]. A segunda consiste de superestimar a carga por um fator. Desse modo, força-se que servidores sejam ativados antes de ocorrer a saturação do aglomerado e evita-se que servidores sejam desativados quando a saturação tende a aumentar no futuro próximo. Segundo as conclusões desse trabalho, o previsor Holt não introduziu uma melhoria em relação ao método citado anteriormente. Por isso, ele não será analisado por este trabalho. Além disso, a antecipação pelo fator mencionado revelou-se mais indicada por ser um método simples, intuitivo e flexível sobre o grau de superestimação da carga.

Assim como em [11], será utilizado um fator de subestimação chamado γ. O uso do fator γ pode ser visto no Algoritmo 1, nas linhas 1, 6 e 17. Quando o valor encontra-se no intervalo [0,1], a sua semântica é de antecipar reconfigurações que visam aumentar a capacidade de processamento (linhas 1 e 6) e retardar o desativamento de servidores devido à baixa carga (linha 17). Porém, para valores maiores que 1, o fator γ tem o efeito inverso.

Quando o valor de γ é próximo de zero, a condição da linha 1 sempre será verdadeira para uma carga não-nula. Consequentemente, todas as máquinas do *cluster* permanece-

rão ligadas e não haverá perda alguma de qualidade de serviço. Em compensação, a capacidade do aglomerado se encontrará frequentemente superestimada, visto que a capacidade máxima do mesmo é raramente usada (o perfil de carga poucas vezes chega ao máximo de requisições). É importante ter em mente que isso gera um compromisso entre superestimações e subestimações. Em experimentos reais esse compromisso será refletido em um aumento da qualidade de serviço em detrimento de um custo energético maior, e vice-versa. No Capítulo 6 é feita uma análise mais detalhada desses efeitos.

3.2 Alocação dinâmica de desempenho

A ADD é uma política de distribuição de desempenho entre as máquinas ativas no momento. Pode-se pensar na ADD como uma reserva de recursos, onde o recurso é o desempenho que os servidores devem oferecer para atender a demanda corrente. Para realizar essa alocação de recursos, é variada a frequência de operação dos servidores. O objetivo principal da ADD é minimizar o gasto energético de um grupo de servidores ativos, que são escolhidos pela política CDS.

Uma vez ajustada a vazão máxima de cada servidor, o balanceador de carga é informado da nova capacidade de cada servidor. De posse dessa informação, este se encarrega de distribuir requisições de forma proporcional a capacidade ajustada de cada máquina. Para mais detalhes à respeito do balanceador de carga, veja a Seção 4.4.

3.2.1 Modelo

Para determinar a maneira mais eficiente de realizar a alocação de performance, é necessário um modelo mais elaborado de potência dos servidores. Por se tratar de um controle de frequências, deve-se considerar a vazão e potência consumida em cada ponto de operação do processador. De maneira mais formal, pode-se estender a Equação 3.1 para representar diversas frequências (f), como na Equação 3.2.

$$P_{servidor}(V, f) = P_{estática}(f) + P_{dinâmica}(V, f) \tag{3.2}$$

A representação da Equação 3.2 pode ser vista na Figura 3.1, para a máquina `ampere`. Nela, pode ser vista a evolução da potência e da vazão máxima para cada valor de frequência discreta (P0, P1 e P2). Em [23], demonstra-se que uma aproximação válida é considerar que a potência dinâmica varia linearmente com a utilização da CPU. Por esse

motivo, os pontos do estado ocioso e ocupado de cada frequência foram interpolados com uma reta.

A variável V também é função de f, ou seja, $V(f)$. Certamente, quanto maior a frequência, maior será a vazão possível para o servidor[2], como pode ser visto na Figura 3.1. Porém, não é necessário utilizar a frequência máxima do processador quando o mesmo se encontra ocioso ($V(f) = 0$), por exemplo. Para uma vazão v qualquer, deve existir uma frequência $f \in [f_{min}, f_{max}]$, tal que $V(f) = v$, e a potência resultante é mínima. É seguro admitir que a relação entre P e f é sempre crescente, para um V fixo, como indica a Figura 3.1. Sendo assim, a variável f que se deseja obter é a frequência mínima que satisfaz $V(f) = v$. Com esta observação, passa-se a mapear apenas um valor de vazão por frequência, ao invés de várias combinações. Por isso, é possível simplificar a Equação 3.2 através da Equação 3.3. Como $P_{estática}$ é agora constante (e não pode ser alterada pela política ADD), basta analisar a função $P_{dinâmica}$ em cada máquina.

$$P_{servidor}(V(f)) = P_{estática} \ + \ P_{dinâmica}(V(f)) \tag{3.3}$$

Uma frequência f qualquer, como foi assumido até aqui, está no espaço contínuo de frequências. Porém o processador trabalha apenas com frequências discretas. É possível obter frequências contínuas através da combinação linear de duas frequências discretas disponíveis. Dessa forma, através de duas frequências discretas adjacentes f^+ e f^-, pode-se emular a frequência intermediária f, através de

$$f = \alpha \cdot f^+ + (1 - \alpha) \cdot f^-, \text{ onde } \alpha \in [0, 1].$$

Obtém-se a vazão V, por

$$V(f) = \alpha \cdot V(f^+) + (1 - \alpha) \cdot V(f^-), \text{ onde } \alpha \in [0, 1],$$

e a potência dinâmica por

$$P(V(f)) = \alpha \cdot P(V(f^+)) + (1 - \alpha) \cdot P(V(f^-)), \text{ onde } \alpha \in [0, 1].$$

Sabendo que $\alpha + (1 - \alpha) = 1$, $\alpha \geq 0$ e $(1 - \alpha) \geq 0$, a potência dinâmica e a vazão formam uma combinação convexa entre duas medidas adjacentes de vazão. Isso pode ser interpretado também como uma média ponderada entre as medidas, onde os pesos pertencem ao intervalo $[0, 1]$. Sendo assim, o valor de potência dinâmica no intervalo $[V(f^-), V(f^+)]$ está sobre uma reta que liga estes dois pontos.

[2]Essa observação vale para requisições intensivas em CPU apenas, como é o caso para este trabalho.

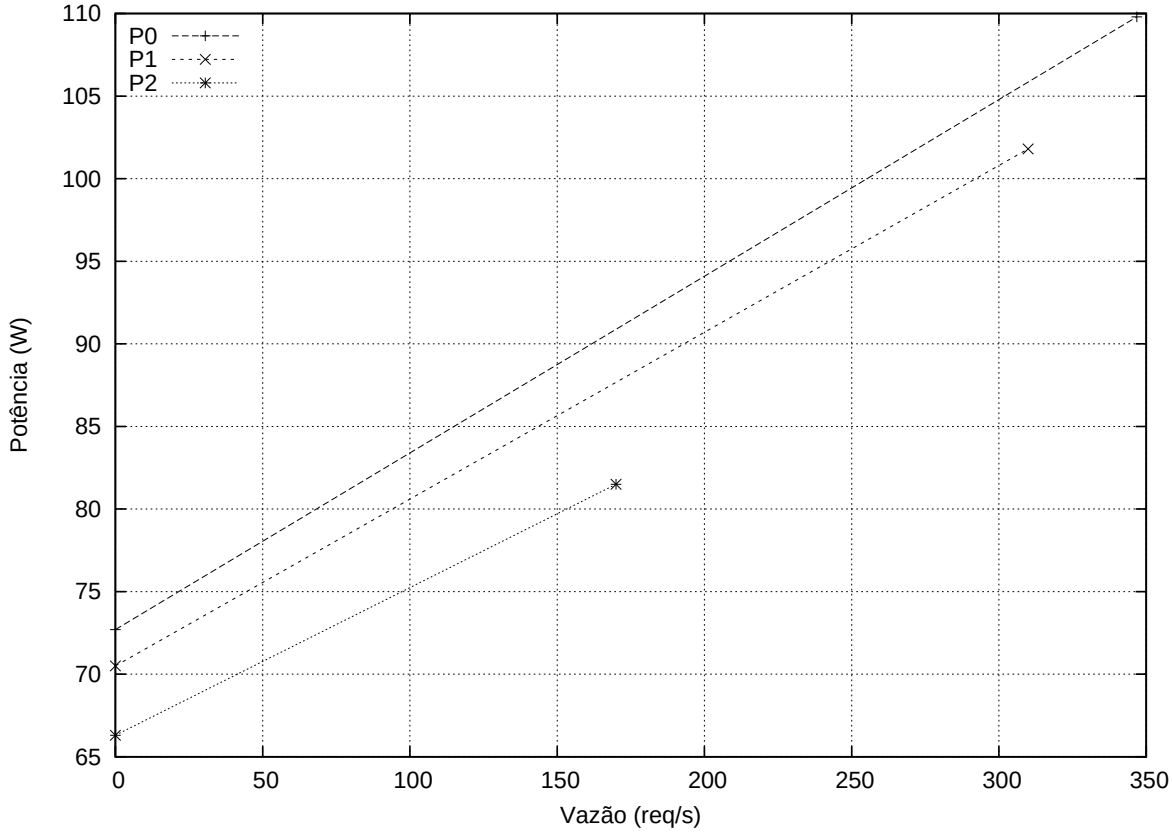


Figura 3.1: Frequências de operação da máquina ampere

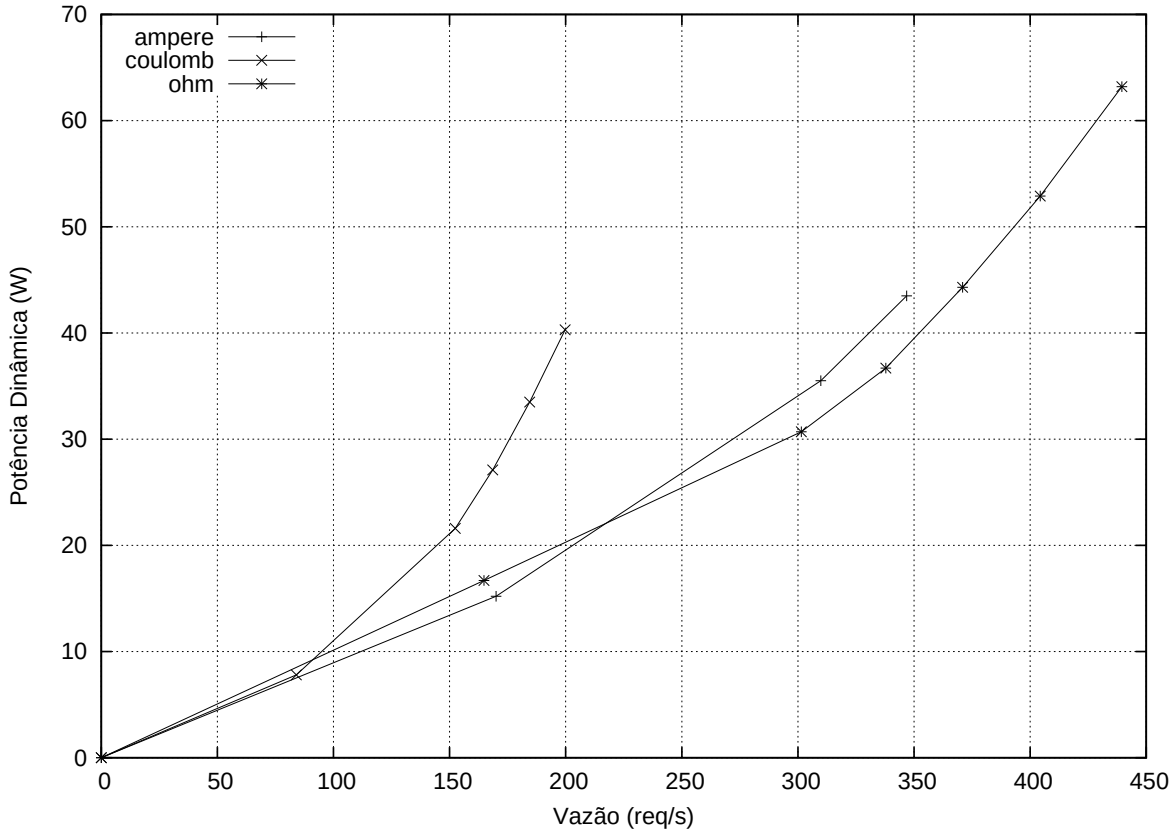


Figura 3.2: Relação entre potência dinâmica e vazão

A Figura 3.2 mostra o gráfico da potência dinâmica em função da vazão para três servidores. O primeiro ponto de cada função é $(P_{dinâmica}, V) = (0, 0)$, indicando o estado *ocioso* onde a máquina só gasta potência estática e não produz nenhum trabalho. Nesse estado, a frequência mínima está ativa. A partir desse ponto, os demais representam a vazão máxima para cada frequência de uma máquina. Com esse arranjo, obtém-se a potência mínima que cada servidor gasta para obter uma dada vazão.

A princípio, determinar a distribuição de carga entre um grupo de servidores é um problema difícil. Mesmo considerando apenas os pontos assinalados na Figura 3.2, para M máquinas, onde cada uma possui F pontos de operação, existirão F^M permutações possíveis. É claro que, num caso típico, deseja-se obter apenas configurações de frequências que podem atender a uma demanda determinada. Essa restrição limita o espaço de busca, mas dependendo da escala, a solução se torna muito custosa para que seja aplicada em tempo de execução. Visto esse problema, deve-se explorar mais a fundo maneiras de otimizar o processo de busca.

As funções representadas na Figura 3.2 possuem duas propriedades importantes: elas são monotonicamente crescentes e convexas [32]. Essa mesma observação também vale para as outras máquinas do *cluster*. Num intervalo I onde uma função $F(x)$ é monotonicamente crescente, a seguinte propriedade vale:

$$x_0 \leq x_1 \Rightarrow F(x_0) \leq F(x_1),\ \forall x_0, x_1 \in I.$$

Dize-se que uma função é convexa em um intervalo I, se sua derivada é monotonicamente crescente nesse intervalo. Aplicando a definição acima, obtém-se:

$$x_0 \leq x_1 \Rightarrow F'(x_0) \leq F'(x_1),\ \forall x_0, x_1 \in I.$$

A função analisada é na verdade uma função linear por partes (*piecewise linear function*) [42]. Como a derivada de uma reta é uma constante, é certo que a derivada da função entre dois pontos interpolados será constante. Em outras palavras, pode-se afirmar que se a taxa de variação entre potência e vazão muda de valor, então foi mudada frequência de operação do servidor.

Através dessas propriedades, pode-se otimizar a distribuição de desempenho. Em primeiro lugar, como a função derivada só varia em pontos de transição de frequência, só é necessário analisar a função nesses locais. Sabendo que a derivada da potência dinâmica é monotonicamente crescente, fica claro que o estado mais eficiente de um servidor é sempre o que possui a menor derivada. Como os valores estão ordenados, fica fácil comparar a

eficiência entre dois ou mais servidores. Ao invés de buscar todas as combinações entre todos os estados, basta buscar a relação de eficiência entre cada servidor. Essa propriedade faz com que o número de comparações possíveis caia de F^M para apenas $F \cdot M$.

Como foi visto, o algoritmo ADD se baseia em duas premissas: (i) a função de potência dinâmica é convexa e (ii) monotonicamente crescente. Caso a primeira premissa seja relaxada, e a função possua um intervalo concavo, por exemplo, é possível remediar a situação. Basta ignorar as frequências que apresentam essa característica e então interpolar os demais pontos. Alguns trabalhos, como [33], identificaram esse problema como frequências ineficientes e recomendam o procedimento sugerido. Em contrapartida, a segunda premissa é improvável. Caso seja possível existir um ponto de operação que gaste menos potência e possibilite uma vazão maior dos que os demais, então recomenda-se a mesma estratégia anterior, ignorando estados que apresentam anomalias.

3.2.2 Algoritmo

Através do modelo desenvolvido é possível projetar um algoritmo simples para a ADD. O objetivo de tal algoritmo é alocar capacidade de processamento (vazão máxima) entre as máquinas ativas do aglomerado. Apesar de haver diversas configurações possíveis onde os servidores ativos podem atender a carga, procura-se o conjunto mais energeticamente eficiente.

O Algoritmo 2 mostra o funcionamento da política ADD. A cada iteração do *loop* da linha 2 será associada uma fração de carga a um servidor. Mais tarde, essa fração de carga será mapeada em uma frequência de operação, de acordo com a Figura 3.2 (tal etapa não é mostrada no Algoritmo 2). A função da variável σ é similar à variável γ do Algoritmo 1 e seu uso será discutido na Seção seguinte.

Para iniciar o processo, é necessário criar uma lista de servidores que contém a eficiência de cada estado de operação. A eficiência é dada pela taxa em que o servidor consegue fornecer desempenho, em relação a potência gasta em cada estado de operação. Isso define a eficiência como o inverso da derivada da potência dinâmica, ou $\frac{dV}{dP}$. Por isso, a linha 3 é responsável por ordenar a lista de servidores de acordo com essa eficiência. Uma vez ordenada a lista de servidores, é possível isolar o servidor mais eficiente ($bestServer$), que obterá uma fração da vazão total de carga ($contribution$). Procura-se alocar uma fração igual ao desempenho máximo que o servidor pode oferecer ($\Delta throughput$) antes de avançar para o próximo estado, onde a eficiência será diferente. Ao final, calcula-se a

Algoritmo 2 Atuação da alocação dinâmica de desempenho

```
1: remainingLoad ← load / σ
2: while remainingLoad > 0 do
3:    sort_by_efficiency(servers) {Ordena a lista de servidores pelo critério de eficiência.}
4:    bestServer ← servers.last() {O servidor mais eficiente está no fim da lista.}
5:    contribution ← min(remainingLoad, bestServer.Δthroughput) {A contribuição do
      servidor não deve ser maior que a carga restante.}
6:    bestServer.requiredThroughput ← bestServer.requiredThroughput + contribution
7:    if bestServer.requiredThroughput = bestServer.maxThroughput then
8:       bestServer.efficiency ← -1 {O servidor se encontra na sua frequência máxima.
         Atribui-se um valor de eficiência negativo, para que este não seja mais escolhido.}
9:    else
10:      bestServer.efficiency ← bestServer.nextStateEfficiency()
11:   end if
12:   remainingLoad ← remainingLoad - contribution
13: end while
```

eficiência para o próximo estado (linha 10) e repete-se o processo até que toda a carga seja alocada.

A complexidade do algoritmo apresentado é dada em função de dois fatores principais. O primeiro é a taxa média em que se associa vazão aos servidores, em outras palavras, $load \div contribution$. Quanto maior o número de servidores (N), maior será também a carga total. Sendo assim, *load* é proporcional a N, enquanto que *contribution* é assumido constante. O outro fator a se considerar é a ordenação que ocorre na linha 3, pois cada iteração implica em uma ordenação da lista de servidores. Para uma ordenação do tipo *heapsort* ou *mergesort*, tem-se uma complexidade de $O(N \cdot log(N))$. Compondo esses dois fatores, define-se a complexidade do Algoritmo 2 como $O(N^2 \cdot log(N))$.

É possível refinar o algoritmo para diminuir sua complexidade assintótica. Ao invés de se ordenar a lista *servers* a cada iteração, é possível realizar uma inserção na lista do único elemento que foi modificado (*bestServer*). Para realizar essa inserção, poderia-se utilizar uma busca binária na lista, da ordem de $O(log(N))$. Assim, o algoritmo passaria a ter complexidade de apenas $O(N \cdot log(N))$.

O algoritmo projetado possui uma grande escalabilidade, permitindo que seja executado em um período mais curto do que a CDS. Por esse motivo, é possível acompanhar

melhor variações de carga para otimizar a energia gasta. Caso houvesse alguma dependência entre CDS e ADD, não seria possível obter essa vantagem.

Outra propriedade importante do algoritmo é que, qualquer que seja o valor de *remainingLoad*, no máximo um único servidor utilizará frequências contínuas. Pode-se verificar isso através da linha 5 do Algoritmo. Caso *remainingLoad* seja menor do que a variação de desempenho do servidor, esta só será obtida através de uma frequência contínua. Porém, quando isso ocorre, *remainingLoad* se torna zero (ver linha 12) e o laço terminará na próxima iteração. Em todos os outros casos, alguma frequência discreta será escolhida. Sendo assim, o algoritmo ADD minimiza o *overhead* causado pelas frequências contínuas (veja a Seção 4.2.1 para detalhes).

Toda atuação do algoritmo ADD é acompanhada de uma redistribuição das frequências em cada servidor. Além disso, o balanceador de carga deve redimensionar os pesos que atribui a cada servidor, de maneira proporcional à capacidade de cada máquina (processo não mostrado no Algoritmo). Devido a esse balanceamento, o efeito da distribuição de carga leva um certo intervalo de tempo para ser efetivo. Levando isso em conta, o período de 3 segundos foi considerado adequado para a atuação da ADD.

3.3 Controle de saturação

Uma vez definidos os esquemas CDS e ADD, resta saber de que maneira serão usados de modo a continuar oferecendo uma qualidade de serviço adequada. Como dito no início do Capítulo, a garantia dessa qualidade será dada indiretamente através de uma outra grandeza, nomeada saturação. Seguindo a premissa de que uma saturação menor do que 1 é suficiente para manter a qualidade de serviço, basta definir uma maneira de controlar a saturação do *cluster*. Tal controle utiliza um *valor alvo* (menor do que um) para a saturação. Ao manter o aglomerado próximo dessa saturação alvo, diminui-se a probabilidade de que a qualidade de serviço seja degradada quando a saturação se aproxima de 1, ao mesmo tempo que se mantém uma eficiência energética.

Na Seção 3.1 foi definido o parâmetro γ, capaz de modificar o comportamento do algoritmo CDS. A Figura 3.3 mostra o impacto do parâmetro γ em subestimações para diversos *traces* (ver Seção 6.1 para detalhes). Nessa simulação foi utilizado um intervalo de 20 segundos como período de atuação da política, e foi contado o número de vezes que a configuração escolhida subestimou a carga. É interessante observar que um valor de γ igual a zero não é necessário para que não haja subestimações. É possível obter zero

subestimações se for utilizado no máximo $\gamma = 0.57$[3], nos experimentos realizados. Isso significa que é possível chegar em um nível de qualidade máximo sem que seja necessário permanecer com todos os servidores ativos.

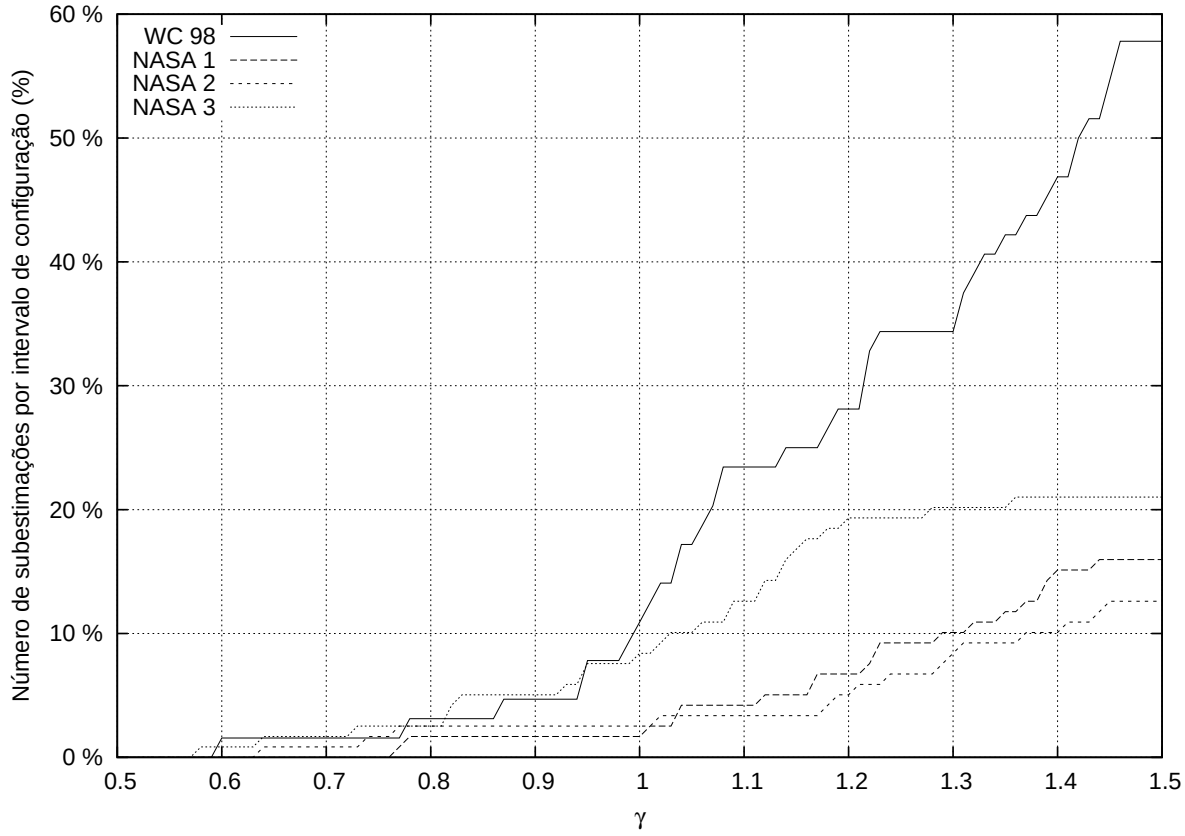


Figura 3.3: Percentagem de configurações subestimadas em função de γ

De acordo com a Figura 3.3, é possível identificar que encontram-se subestimações de até 10% das configurações quando $\gamma = 1$. Isto confirma a necessidade de haver um mecanismo para antecipar reconfigurações para que seja possível obter um número menor de subestimações.

No caso do Algoritmo 2, o valor σ da linha 1 é na verdade o nível de saturação acordado. Como a saturação é definida como $sat = \frac{load}{throughput_{max}}$, ao fixar um nível de saturação σ, ou seja, $sat = \sigma$, a vazão resultante será dada por $throughput_{max} = \frac{load}{\sigma}$. Dessa forma, o Algoritmo 2 irá alocar mais desempenho do que o necessário, acomodando oscilações de carga que podem saturar o *cluster*.

Uma abordagem empírica foi utilizada para o controle de saturação. A partir de um nível de sat de saturação, definiu-se que $\gamma = \sigma = sat$. Essa abordagem apresentou

[3]Em um experimento real, o valor de γ será diferente devido ao atraso entre a atuação e o efeito da ativação de servidores, não contabilizada por essa simulação.

uma capacidade de controlar a saturação com um erro de até 7,7%, no pior caso. Por esse motivo, ela foi usada nos experimentos do Capítulo 6. Pode existir uma combinação de γ e σ que seja capaz de se aproximar mais do alvo especificado. Porém, tal análise pode ser complexa, sendo deixada para investigações futuras.

3.4 Sumário

Neste Capítulo foi apresentada a proposta de gerenciamento de energia deste trabalho. Através das políticas CDS e ADD apresentadas, procurou-se desenvolver uma política abrangente, de atuação rápida e com *overhead* mínimo para economia de energia. Além disso, foi apresentado o esquema utilizado para prover um controle de saturação, que por sua vez está relacionado com o nível de QoS obtido.

O objetivo deste Capítulo foi introduzir os modelos matemáticos, bases teóricas e heurísticas da proposta. No entanto, é necessário entender de que maneira essa base abstrata apresentada se relaciona com o mundo real. Por isso, no próximo capítulo será apresentado o ambiente de testes, sua topologia e como os principais mecanismos de *hardware* utilizados funcionam.

Capítulo 4

Ambiente de testes

Neste Capítulo serão discutidos os principais componentes do ambiente de testes deste trabalho. Inicialmente será justificada a topologia escolhida, apresentando os servidores e a suas funções correspondentes. Em seguida, serão detalhados o funcionamento dos mecanismos de *hardware* utilizados para a economia e medição de energia. As seções que se seguem tratam da caracterização do tipo de requisição utilizada e do balanceador de carga. Por fim, serão apresentados os valores de performance e potência para os servidores do ambiente de testes.

4.1 Topologia

Em um artigo divulgado em 2003 [5], foi revelada a arquitetura de *clusters* usada pela Google no seu processamento de pesquisas na *Web*. O artigo explica que esse tipo de aplicação possui um alto grau de paralelismo, onde o processamento de cada requisição é realizada por várias máquinas antes de ser respondida ao usuário. Nesse cenário, onde o trabalho é dividido por vários servidores, o poder de processamento de um nó não precisa ser grande. De fato, segundo o artigo, cada *cluster* é composto de 15 mil máquinas *desktops*, que podem ser compradas pelo consumidor comum. A razão dessa escolha é o baixo custo por unidade e a alta relação entre desempenho e consumo de potência desses equipamentos em relação a servidores *high-end*.

Esses servidores ainda apresentam outra característica importante: não armazenam um estado privado da aplicação (são *stateless*). Do ponto de vista da tolerância a falhas, esse sistema possibilita que máquinas com problemas não prejudiquem o funcionamento correto da aplicação (*graceful degradation*). Mais ainda, servidores podem ser desativados propositalmente para economizar energia sem que haja uma perda de contexto

da aplicação.

Com a rápida evolução da arquitetura de computadores, em poucos anos o *hardware* dos servidores se torna obsoleto. Em contrapartida, o poder computacional de um *datacenter* deve continuar crescendo de maneira a acompanhar a demanda de seus serviços. Porém, é difícil manter um *datacenter* composto por servidores iguais (homogêneos). O custo de se trocar todos os servidores seria muito alto, e continuar a adquirir *hardware* ultrapassado não compensaria na relação de custo e manutenção por unidade. Por esses motivos, é comum que *clusters* possuam máquinas de diversas gerações convivendo lado a lado, como indicado em [5].

Aplicações que possuem requisitos semelhantes ao motor de busca da Google têm a oportunidade de usar uma arquitetura com servidores baratos e eficientes. Baseado nessa ideia, o ambiente de testes deste trabalho consiste de um *cluster* heterogêneo[1], composto de computadores do tipo *desktop*. Essas máquinas estão dispostas de modo a emular apenas uma camada de máquinas trabalhadoras (*workers*) em um aglomerado real que pode ser *multi-tier*. Há um *front-end* responsável por gerenciar o *cluster* e distribuir a carga entre os servidores, que por sua vez é gerada por uma outra máquina. A Figura 4.1 mostra o arranjo do sistema.

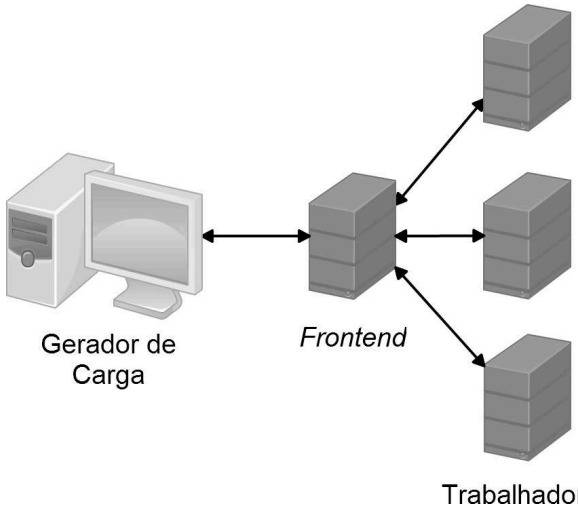


Figura 4.1: Topologia

Cada servidor roda o sistema operacional Gentoo Linux [18]. Essa distribuição Linux foi escolhida por ser leve e enxuta, evitando que os resultados dos experimentos

[1]O *cluster* estudado é heterogêneo por ser composto de máquinas com relação performance por Watt distintas, como pode ser visto na Seção 4.5. Apesar disso, todos os *workers* pertencem à mesma família, AMD Athlon 64.

Tabela 4.1: Propriedades dos computadores do ambiente de testes

Máquina	**Função**	**Frequências**	***Cores***	**Processador**
`watt`	*front-end*	n/a	n/a	AMD Athlon 64 3200+
`ampere`	*worker*	3	2	AMD Athlon 64 X2 3800+
`coulomb`	*worker*	5	1	AMD Athlon 64 3800+
`hertz`	*worker*	5	1	AMD Athlon 64 3800+
`joule`	*worker*	4	1	AMD Athlon 64 3500+
`ohm`	*worker*	6	2	AMD Athlon 64 X2 5000+
`camburi`	gerador de carga	n/a	n/a	Intel Pentium 4 2.80GHz
`putiri`	medidor de potência	n/a	n/a	Intel Pentium 4 2.80GHz

sejam afetados por interferência de outras tarefas concorrentes do sistema.

O servidor *Web* Apache [3] versão 2.2.10 foi instalado em cada trabalhador, juntamente com o interpretador PHP versão 5.2.4. Esse é um *setup* muito comum em servidores *Web*, composto de *softwares* livres e abertos. A variante mais usada dessa suíte de aplicativos recebe a sigla LAMP (Linux, Apache, MySQL e Perl/PHP/Python) [21]. Porém, esse trabalho não utilizará a camada de base de dados (MySQL).

As características dos computadores utilizados e suas funções correspondentes podem ser vistas na Tabela 4.1. Todos os servidores trabalhadores são capazes de DVFS (ver Seção 4.2.1), com um número de frequências intermediárias especificado na Tabela. Foram selecionadas máquinas menos potentes para o papel de *front-end* e gerador de carga, porém ambas apresentam desempenho mais do que o suficiente para cumprir suas funções. Em medições realizadas, o *front-end* e a máquina geradora de carga tem utilização média de 70%[2] e 54%, respectivamente.

4.2 Mecanismos de *hardware*

Nesta Seção serão detalhados os principais recursos de *hardware* utilizados para economia de energia: DVFS e a dupla Suspend-to-RAM/Wake-on-LAN. Em seguida, será mostrado como funciona o mecanismo de aquisição de potência, LabVIEW.

[2]O sistema ESSenCe ocupa apenas 14% de utilização, enquanto o Apache é responsável pelo restante.

4.2.1 *Dynamic Voltage and Frequency Scaling*

O mecanismo de DVFS está presente na maioria dos processadores modernos. Através dele, pode-se variar a frequência de operação (taxa de *clock*) de um processador para se adequar a diversos tipos de carga de trabalho (*workload*). Os níveis discretos de operação do DVFS são chamados de *P-States* (Estados de Performance), segundo o padrão ACPI. A implementação dos *P-States* é deixada a cargo de cada fabricante, e é divulgada com diferentes nomes como Intel (Enhanced) SpeedStep®, AMD PowerNow!®, VIA PowerSaver® e outros.

Pode-se pensar num *P-State* como uma tupla que contém dois campos chamados de VID e FID, que identificam a voltagem e frequência, respectivamente. Entretanto, é possível que mais de um *P-State* esteja associado à um mesmo VID. O FID está diretamente relacionado com a capacidade de trabalho que pode ser feita pela CPU. Já a voltagem associada (VID) serve para estabilizar o *clock* do processador e não aumenta a capacidade de processamento por si só. Em processadores multi-núcleo, como alguns do ambiente de testes, um *P-State* é compartilhado entre os *cores*. Porém, processadores mais modernos podem operar em estados independentes em cada núcleo.

A potência gasta por um processador é muitas vezes modelada como $P_{base} + C \cdot V^2 \cdot f$ [14]. Nessa expressão, P_{base} define uma potência gasta pelo processador independentemente da frequência ou voltagem de operação, e é causada por um fenômeno físico chamado *leakage current*. O termo C é tido como constante e representa a capacitância do circuito. Já os termos V e f representam voltagem e frequência, respectivamente, e variam de acordo com o *P-State* corrente.

No Linux existe um subsistema incluído no *kernel* chamado `CPUfreq`, responsável por gerenciar DVFS. Devido a grande variedade de arquiteturas, é necessário instalar um *driver* apropriado para que o `CPUfreq` possa atuar no sistema. Por exemplo, para a microarquitetura AMD K8 (Athlon™64, Sempron™, Opteron™, entre outros), utiliza-se o driver `powernow-k8`. Já processadores Intel mais atuais (Core™2, Core™i5, etc) utilizam o módulo `acpi-cpufreq`.

O `CPUfreq` acompanha um conjunto de controladores de DVFS chamados *governors*. Entre eles, é de interesse citar três: `performance`, `userspace` e `ondemand`. O primeiro deles não realiza transições de frequência, mantendo o processador no seu *P-State* máximo (`P0`). Já o *governor* `userspace` permite que uma frequência seja especificada através de um arquivo de configuração. Desse modo, controladores customizados po-

dem ser executados em nível de usuário (oposto ao nível de *kernel*, privilegiado). Esse *governor* foi usado para implementar o mecanismo de frequências contínuas, discutido na Seção 4.2.1. O `ondemand` implementa uma política de DVFS que procura manter a utilização da CPU em um determinado nível. Ele é o *governor* ativado por padrão em várias distribuições Linux. Por ser uma política de comparação do Capítulo 6, o seu funcionamento será discutido na Seção seguinte.

Ondemand

O controle do `ondemand` procura manter a utilização média de todos os núcleos do processador em um determinado nível. Para tanto, ele emprega duas faixas de controle, uma inferior e outra superior ao valor alvo. Se a utilização média u_{atual} em um intervalo de tempo for maior do que o limiar superior $u_{máx}$, a frequência máxima é ativada. Sendo assim, a frequência nova será ajustada de acordo com a Equação 4.1.

$$f_{nova} = f_{máx} \tag{4.1}$$

Analogamente, caso a utilização diminua abaixo do limiar inferior $u_{mín}$, o controlador `ondemand` abaixará a frequência para um certo nível. Esse valor é estimado como sendo o necessário para que a nova utilização seja igual ao limiar inferior. Para atingir esse objetivo, o *governor* assume que a utilização U do processador é inversamente proporcional à frequência de operação, gerando a Equação 4.2. Nela, C representa uma constante característica da CPU.

$$U(f) = \frac{C}{f} \tag{4.2}$$

Segundo a equação acima, basta que $U(f)$ e f sejam conhecidos para determinar C. O controlador conhece a frequência atual e amostra utilização obtendo u_{atual} e f_{atual}, e consequentemente $C = u_{atual} \cdot f_{atual}$. Para obter a frequência desejada, basta substituir o valor de C e $u_{mín}$ encontrados, na Equação 4.2. O resultado pode ser visto na Equação 4.3.

$$u_{mín} \cdot f_{nova} = u_{atual} \cdot f_{atual} \;\Rightarrow\; f_{nova} = f_{atual} \cdot \frac{u_{atual}}{u_{mín}} \tag{4.3}$$

O `ondemand` utiliza, por padrão, 70 e 80 porcento para o valor de $u_{mín}$ e $u_{máx}$, respectivamente. O período de atuação do algoritmo nas máquinas do ambientes de testes é, por *default*, 1,24 segundos.

Esse *governor* conta ainda com um parâmetro chamado *powersave_bias*, capaz de trocar performance por economia de potência. Essa variável funciona como um limitador para a frequência escolhida pelas Equações 4.1 e 4.3, multiplicando as mesmas por um fator $\beta \in [0, 1]$[3]. Assim, a equação completa desse controlador pode ser vista na Equação 4.4.

$$f_{nova} = \begin{cases} f_{máx} \cdot \beta & \text{se } u_{atual} > u_{máx} \\ f_{atual} \cdot \frac{u_{atual}}{u_{mín}} \cdot \beta & \text{se } u_{atual} < u_{mín} \end{cases} \tag{4.4}$$

Uma vez determinada f_{nova}, a mesma pode se encontrar em um valor intermediário entre a frequência de dois *P-States* implementados no processador. Caso $\beta = 1$, o valor encontrado será aproximado para a frequência discreta superior mais próxima. Porém, caso $\beta < 1$, o controlador utiliza um mecanismo especial para simular f_{nova}, chamado aqui de *frequências contínuas*. Esse mecanismo é usado tanto pelo `ondemand`, quanto pela proposta e será explicado em detalhes na próxima Seção.

Frequências Contínuas

Em [20] foi demonstrado que é possível simular frequências contínuas à partir das discretas existentes em um processador. O objetivo é obter pontos de operação com uma granularidade mais fina, otimizando o consumo energético.

O processo é simples e consiste em trabalhar nas duas frequências adjacentes àquela que se deseja simular, como na Figura 4.2. Para obter o efeito da frequência equivalente (f_{eq}), deve-se passar a unidades de tempo em f_1 e um intervalo $(b - a)$ operando na frequência f_2, como na equação:

$$b \cdot f_{eq} = a \cdot f_1 + (b - a) \cdot f_2.$$

A maioria dos termos são conhecidos e uma vez definido um período b arbitrário, o problema se reduz a descobrir o valor de a. Para tanto, basta rearrumar a equação acima da seguinte forma:

$$b \cdot f_{eq} = a \cdot f_1 + b \cdot f_2 - a \cdot f_2 \Rightarrow$$

$$a \cdot (f_1 - f_2) = b \cdot (f_{eq} - f_2) \Rightarrow$$

$$a = b \cdot \frac{f_{eq} - f_2}{f_1 - f_2} = b \cdot \frac{f_2 - f_{eq}}{f_2 - f_1}.$$

[3]Adotou-se a variável β no lugar do *powersave_bias*, por ser mais intuitiva. Pode-se converter entre as duas medidas através de $\beta = (1000 - powersave\_bias) \div 1000$.

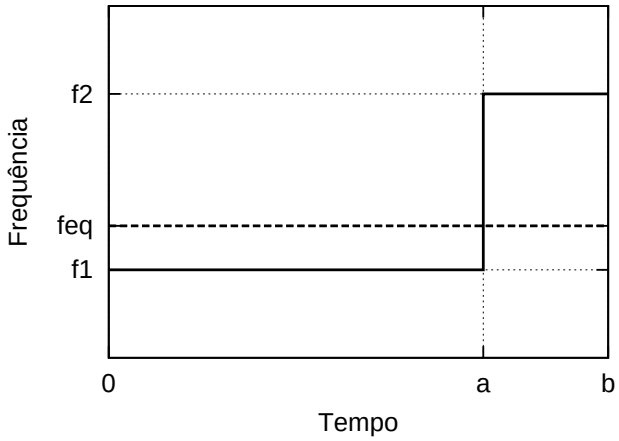


Figura 4.2: Ilustração de frequência contínua

O valor de b deve ser escolhido com atenção, pois durante as transições que ocorrem durante um período, o processamento para momentaneamente. Esse tempo é chamado *latência de transição* e seu valor depende da arquitetura do processador. Para as máquinas deste ambiente de testes, o maior tempo é da ordem de 100μs [2]. Para que o *overhead* de chaveamento não seja significativo, o valor de b deve ser muito maior que a latência das duas transições que ocorrem em cada período. Por esse motivo, o valor do período b na implementação é de 100ms.

Outro detalhe que deve-se atentar é que os valores de a e $(b-a)$ não podem ser arbitrariamente pequenos, caso contrário o tempo em que o processador passa em uma das frequências f_1 ou f_2 seria pequeno demais para surtir algum efeito. Considerando o tempo de latência $t_{latência}$, deve-se fazer com que a ou $(b-a)$ sejam maiores ou iguais a $t_{latência}$, ou seja $min(a, b-a) \geq t_{latência}$. Com essa restrição, pode-se representar frequências com uma granularidade de até $\frac{t_{latência}}{b} = \frac{100\mu s}{100ms} = 0,1\%$ de alguma frequência discreta do sistema. Em termos práticos, isso significa que não é possível representar variações menores que 3MHz, (e.g. 1001MHz, 2002MHz, etc).

4.2.2 *Suspend-to-RAM* e *Wake-on-LAN*

Um sistema computacional pode ser colocado em modos de baixo consumo de energia, agrupados no que se chama de estado `G1` (*Sleeping*) pelo padrão ACPI. Esse estado se opõem ao `G0`, onde o computador se encontra totalmente operacional. Em `G1`, o computador desliga algum dispositivo como CPU, disco ou memória RAM. Por esse motivo, `G1` é subdividido ainda em outros quatro estados, de nome `S1` até `S4`, de maneira progressiva a medida em que desliga componentes. A Tabela 1.1 da página 3 resume a organização

desses estados.

O estado `S3` é também conhecido como *Suspend-to-RAM*, ou modo *Standby*. Nesse modo de operação, a CPU salva seu contexto em memória RAM e em seguida é desligada. Segundo medidas feitas no ambiente de testes, um servidor gasta até 6W nesse estado, o que gera uma economia considerável de energia. A ativação desse estado é feita via *software*, através de um arquivo de configuração do `CPUfreq`.

Como a CPU é desligada pelo *Suspend-to-RAM*, não há como interagir mais com a interface ACPI do Linux para retornar ao estado `G0`. O único meio de despertar a CPU é através de uma interrupção especial de *hardware* de algum outro dispositivo que esteja ativo, como o botão de energia ou teclado.

O *Wake-on-LAN* [43] é um mecanismo presente em placas de rede que possibilita despertar a CPU de estados `G1`. O sinal de interrupção é gerado quando a placa de rede recebe um quadro especial, chamado *magic packet*[4]. Esse quadro pode ser enviado por qualquer nó que esteja na mesma sub-rede da estação que está em *standby*. Em Linux, um programa que é capaz de enviar esse quadro é o `ether-wake` [7].

É importante observar que, diferentemente do método de DVFS descrito na Seção 4.2.1, o tempo de chaveamento de um servidor entre os estados `G1` e `G0` não pode ser desprezado. Testes indicam um retardo de cerca de 2 segundos para entrar em *standby* [5], e de até 6 segundos para voltar ao estado de operação normal. Observe que em aplicações mais complexas, o retardo de chaveamento pode ser maior.

4.2.3 Aquisição de potência com o LabVIEW

O LabVIEW é um ambiente de programação gráfica desenvolvida pela National Instruments (NI) para medir, testar e controlar sistemas. Ele oferece suporte a diversos dispositivos de *hardware*, entre eles o sistema de aquisição de dados (DAQ) `NI USB 6009`. Esse dispositivo é responsável por amostrar valores de potência das máquinas do *cluster* e convertê-las para sinal digital.

A amostragem é feita pela potência combinada das máquinas de teste (*workers*). A Figura 4.3 mostra um diagrama do processo de aquisição de potência. As máquinas

[4]Apesar do nome conter a palavra *packet*, trata-se de um quadro da camada de enlace.

[5]Na prática esse intervalo é maior, pois deve-se contabilizar também o tempo que o servidor Apache leva para terminar de atender as requisições pendentes na sua fila de espera. Nos testes foram adicionados 3 segundos para garantir esse atendimento, totalizando 5 segundos para completar o *Suspend-to-RAM*.

estão ligadas em uma mesma régua de energia, que por sua vez é conectada à sensores de corrente e tensão. O DAQ amostra os sinais dos sensores para que possam ser processados pela máquina medidora de potência.

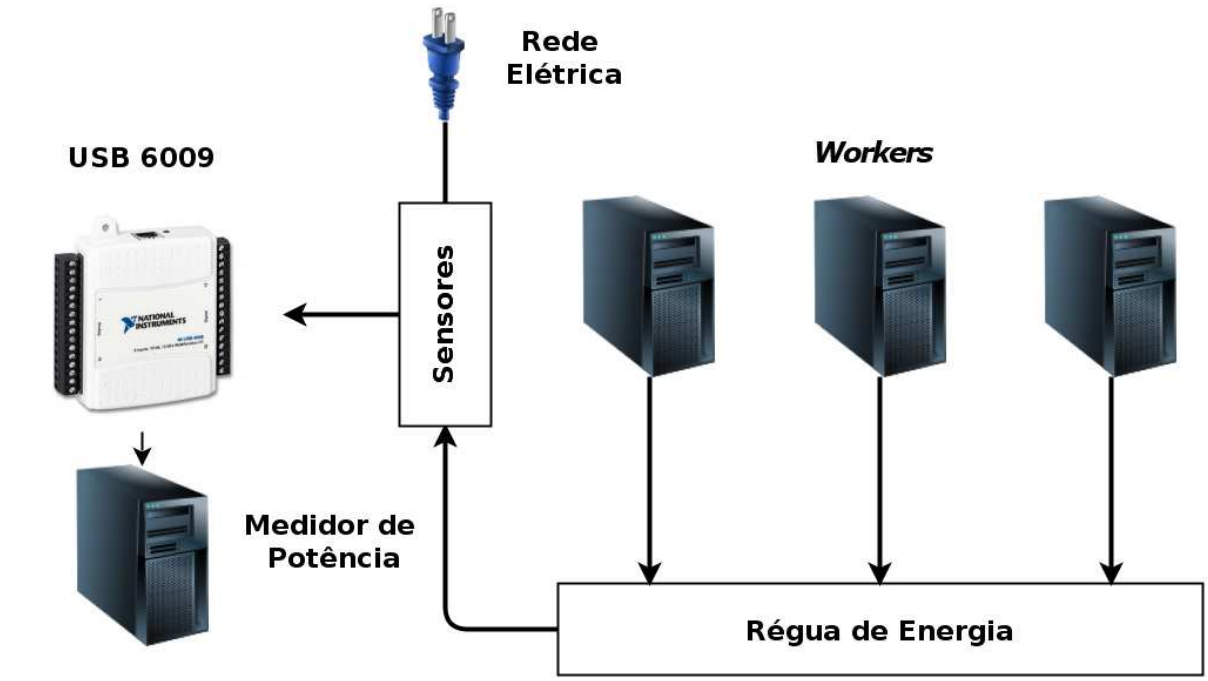


Figura 4.3: Aquisição de potência

Uma máquina especializada (`putiri`), rodando Windows XP, colhe através da interface LabVIEW as medidas de potência. Essa máquina é acessível através do *front-end* e é usada como um *medidor de potência*.

4.3 Carga de trabalho

Uma aplicação *Web* pode oferecer uma miríade de serviços para seus usuários, variando de notícias até *streaming* de vídeo. Seja como for, toda aplicação precisa atender a **requisições** de seus usuários.

As requisições, por sua vez, são tão diversas quanto os tipos de aplicações *Web* existentes. Pode-se classificá-las pelo uso de recursos do servidor, como intensivas em CPU, memória ou disco. É possível dividi-las pelo tipo de conteúdo gerado: estático ou dinâmico, e então definir tratamentos diferentes para os dois casos.

De fato, é impossível trabalhar com requisições representativas de todo tipo de aplicação existente na *Web*. Por isso, este trabalho limitou-se a estudar requisições intensivas em CPU do tipo estático. A requisição desenvolvida consiste de uma página PHP cuja função é calcular números primos de 2 até um valor determinado N (para efeito

dos experimentos do Capítulo 6, utilizou-se $N = 200$). A modelagem de apenas uma requisição no sistema generaliza o caso onde a variação entre o tempo de atendimento das requisições é baixa. A implementação consiste de cálculo aritmético puro, sem referências a conteúdo em memória ou disco.

O componente que mais gasta energia em um servidor é a CPU [10]. Dessa forma, é importante otimizar a utilização de energia de requisições intensivas nesse recurso. Além disso, trabalhar com requisições desse tipo tem outras vantagens importantes.

A primeira é a minimização da variância do tempo de atendimento das requisições. Podemos contar como contribuição para a variância das medidas o processamento aritmético do PHP, o chaveamento entre outras tarefas do sistema operacional e o tempo de rede. A montagem do ambiente de testes foi pensada de modo a minimizar essas interferências, utilizando requisições estáticas (N fixo), um sistema operacional enxuto, ferramentas que impõem *overhead* mínimo e uma rede *gigabit* dedicada.

A segunda é a existência de mecanismos implementados em *hardware*, manipuláveis com facilidade via *software*, que permitem trocar capacidade de processamento da CPU por um menor consumo de energia, como o DVFS (Seção 4.2.1).

4.3.1 Gerador de carga

Neste trabalho foi utilizada uma ferramenta para simular as requisições que são feitas pelos usuários em um *cluster* comercial. Essa ferramenta se chama `httperf` [26] e foi desenvolvida pela HP (Hewlett-Packard) com o propósito de realizar *benchmarks* em servidores *Web*.

O `httperf` é uma ferramenta de código livre poderosa, capaz de simular requisições elaboradas, permitindo que administradores possam obter uma medida confiável sobre a performance dos seus servidores. Apesar disso, foi necessário implementar uma funcionalidade extra nessa ferramenta[6]. É preciso variar a aceleração da carga, ou seja, a quantidade de requisições no tempo. Através dessa propriedade é possível enviar *traces* de carga que são mais próximos de um processamento realizado em um *datacenter* real. Esse aspecto será discutido em detalhes na Seção 6.1.

[6] A implementação dessa funcionalidade é de autoria de Carlos Sant' Ana [34].

4.4 Balanceamento de carga

Em um aglomerado homogêneo, onde cada máquina possui a mesma capacidade de processamento, o balanceamento de carga ideal pode ser feito de maneira simples. Através de um esquema *round-robin*, cada servidor trabalhador receberia aproximadamente a mesma quantidade de requisições. Dessa forma, garantiria-se também que a utilização de cada servidor seria a mesma. Porém, o ambiente de testes em questão consiste de um *cluster* heterogêneo. É necessário alterar a percentagem de requisições que cada máquina recebe, para que seja proporcional ao desempenho atual de cada uma. Por sua vez, esse valor de vazão varia entre máquinas e frequências distintas, como será visto na Seção 4.5.

No servidor Apache existe um módulo responsável pelo balanceamento de carga chamado `mod_proxy_balancer` [4]. Através desse módulo é possível especificar qual a percentagem de requisições a ser encaminhada para cada servidor. A interface com esse módulo será discutida em detalhes no Capítulo 5.

Para realizar o balanceamento por requisições, o `mod_proxy_balancer` implementa um escalonador *fair-share*. Esse escalonador funciona da seguinte maneira. Cada um dos n nós fica associado a um peso w_i (um inteiro positivo) e um contador que será chamado de t_i (iniciado com zero). A cada passo do algoritmo, escolhe-se o nó que possui o menor valor de t. Acrescenta-se à variável t do nó escolhido, o seu valor de w. Repetindo o processo várias vezes, a percentagem p_i de requisições que o nó i irá receber será dada pela razão: $p_i = \frac{1}{w_i} \cdot \sum_{\forall\, i \in n} w_i$. Um exemplo ilustrativo será dado à seguir.

Suponha três nós: A, B e C, recebendo um fluxo ininterrupto de requisições. Deseja-se que A, B e C recebam 50%, 40% e 10% das requisições, respectivamente. Por conveniência, o somatório da equação anterior será forçado para o valor 100. Isso implica que cada peso w_i será dado por: $\frac{100}{p_i}$. Logo, o peso para estes nós serão: $w_A = 2$, $w_B = 2,5$ e $w_C = 10$.

Nessas condições, a situação para as 10 primeiras requisições está ilustrada na Tabela 4.2. A cada requisição, incrementam-se a variável t do nó escolhido (indicado na linha "Destino"). Quando há empate, qualquer nó pode ser escolhido. Ao final do processo, foi calculado quantas requisições foram encaminhadas para cada nó. Como esperado, em média, cada nó recebeu a percentagem combinada de requisições.

Tabela 4.2: Ilustração do esquema *fair-share*

Tempo	1	2	3	4	5	6	7	8	9	10	**Total**
t_A	0^{+2}	2	2	2^{+2}	4	4^{+2}	6	6^{+2}	8	8^{+2}	5/10
t_B	0	$0^{+2,5}$	2,5	2,5	$2,5^{+2,5}$	5	$5^{+2,5}$	7,5	$7,5^{+2,5}$	10	4/10
t_C	0	0	0^{+10}	10	10	10	10	10	10	10	1/10
Destino	A	B	C	A	B	A	B	A	B	A	

4.5 Perfil dos servidores

O sistema proposto requer o conhecimento prévio do perfil entre performance e potência de cada servidor. Através desse perfil, será possível otimizar a energia gasta pelo aglomerado ao manter as máquinas mais eficientes ligadas e alocar o desempenho necessário de maneira a minimizar a potência gasta. Os mecanismos para atingir estes objetivos foram discutidos no Capítulo 3.

Para realizar o levantamento desse perfil, mediu-se a potência de cada máquina separadamente em dois estados: *ocioso* e *ocupado*. No estado *ocioso*, a máquina se encontra pronta para atender requisições, com o servidor Apache carregado em memória, mas não está recebendo nenhuma carga no momento. Já no estado *ocupado*, o servidor se encontra saturado: a utilização de CPU é de 100% e requisições se acumulam na fila de atendimento do Apache.

A potência foi medida nos dois estados através do sistema de aquisição de dados descrito na Seção 4.2.3. Para medir a vazão no estado *ocupado*, em cada frequência de cada servidor, repetiu-se o experimento a seguir. Primeiro, configurou-se o servidor *Web* da máquina *front-end* para direcionar tráfego para a máquina que se deseja testar. Em seguida, o gerador de carga foi configurado para criar uma carga alta o suficiente para saturar qualquer máquina do *cluster* (500 req/s). Esse valor permite manter o servidor sempre ocupado, obtendo-se efetivamente a sua vazão máxima. Essa carga é enviada durante um minuto e ao final do experimento obtém-se a quantidade de requisições processadas em média, por segundo, através de um relatório gerado pelo `httperf`. Esse processo foi repetido três vezes por frequência para garantir a significância da medida. A vazão resultante é a média entre os três experimentos.

A Tabela 4.3 lista os resultados do *profiling* dos servidores. Comparando os valores obtidos tanto de potência quanto de vazão, fica claro que trata-se de um *cluster* bastante heterogêneo. Uma tendência interessante é que os modelos mais modernos da linha AMD

Tabela 4.3: Perfil dos *workers*

Worker	**Frequência (MHz)**	**Potência ocioso(W)**	**Potência ocupado(W)**	**Vazão (req/s)**
`ampere`	1000	66,3	81,5	170,0
	1800	70,5	101,8	309,9
	2000	72,7	109,8	346,8
`coulomb`	1000	67,4	75,2	84,0
	1800	70,9	89,0	152,47
	2000	72,4	94,5	168,6
	2200	73,8	100,9	184,5
	2400	75,2	107,7	199,8
`hertz`	1000	63,9	71,6	82,83
	1800	67,2	85,5	151,3
	2000	68,7	90,7	167,6
	2200	69,9	96,5	183,8
	2400	71,6	103,2	198,5
`joule`	1000	66,6	74,7	82,0
	1800	73,8	95,7	148,3
	2000	76,9	103,1	163,3
	2200	80,0	110,6	179,4
`ohm`	1000	65,8	82,5	164,8
	1800	68,5	99,2	301,47
	2000	70,6	107,3	337,87
	2200	72,3	116,6	370,93
	2400	74,3	127,2	404,37
	2600	76,9	140,1	439,5

Athlon 64 possuem uma relação entre vazão e potência consumida bem superior aos modelos mais antigos (ver Tabela 4.1 para a lista de modelos). Por exemplo, a máquina `ohm` consegue atender 164,8 requisições por segundo, gastando 82,5W, operando a 1000MHz. Em contrapartida, um modelo mais antigo da série, como `joule`, gasta 103,1W para atender a um valor próximo de 163,3 requisições por segundo, na frequência de 2000MHz. Com quase 21% a menos de potência, `ohm` é capaz de oferecer uma performance equivalente à `joule`.

É interessante observar a relação entre o estado *ocioso* e *ocupado*, em termos de potência. Na máquina `coulomb`, por exemplo, gasta-se o mesmo valor de potência (75,2W) tanto atendendo a 84 requisições por segundo, ou no estado *ocioso* da frequência de 2400MHz. Isso confirma a intuição de que se manter uma máquina ociosa em uma frequência alta não é energeticamente eficiente.

Através do exposto na Seção 3.2, concluiu-se que a potência mínima da vazão máxima de cada servidor é dada através de uma função convexa. A Figura 4.4 mostra essa função para todas as máquinas trabalhadoras do *cluster*. Repare que as máquinas

menos potentes tem um crescimento mais rápido da função de potência. Isso coincide com a premissa que foi adotada na Seção 3.1, onde servidores menos poderosos tem uma relação performance por Watt baixa.

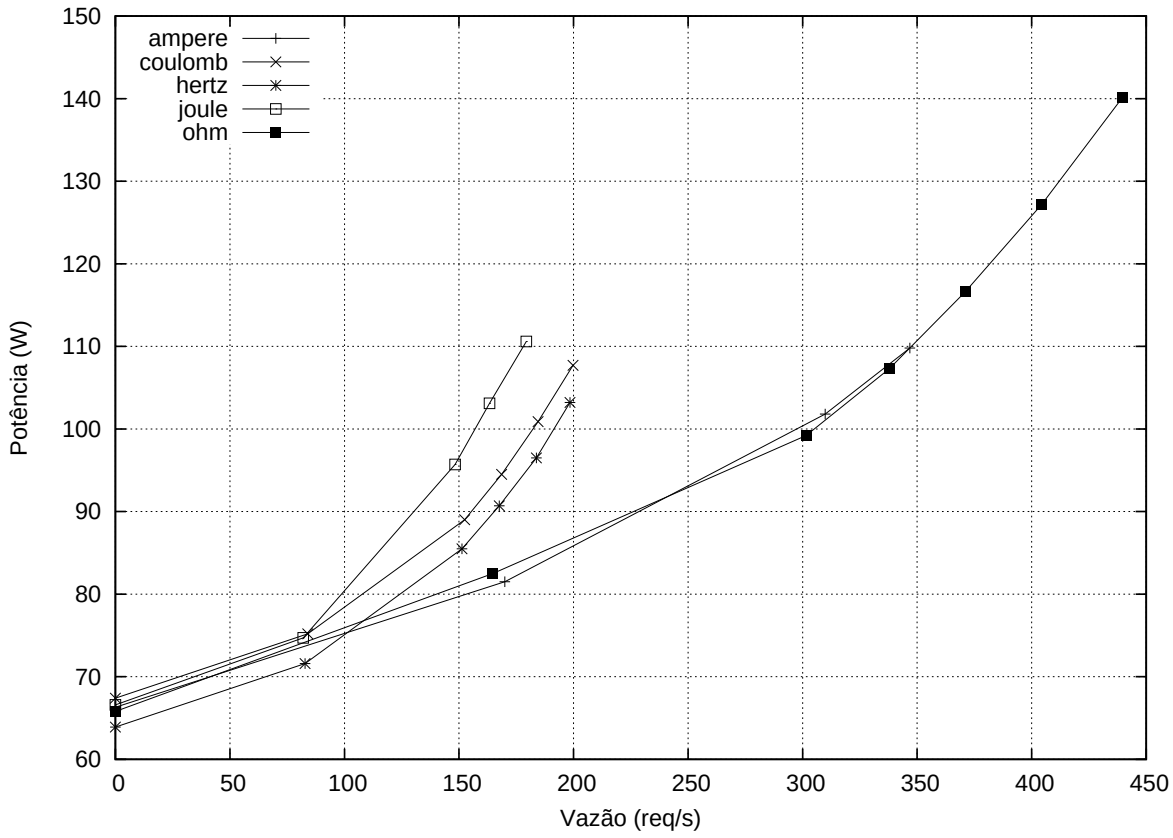


Figura 4.4: Relação entre potência e vazão

4.6 Sumário

Nesta Seção foram abordados os elementos que compõem o ambiente de testes escolhido para validar a proposta. Procurou-se criar esse ambiente de modo a emular, em pequena escala, um *cluster* de servidores *Web* comercial. Em seguida, foram explicitados os componentes de *hardware* explorados para economia de energia nos servidores. Por último obteve-se a relação entre potência e desempenho de cada servidor do aglomerado.

Agora que essa infraestrutura foi determinada, é necessário considerar a implementação em *software* dos componentes apresentados neste Capítulo. Além disso, deve-se apresentar uma implementação para as políticas propostas no Capítulo 3. Essas questões serão abordadas no capítulo seguinte.

Capítulo 5

Implementação

Neste Capítulo, serão detalhados os componentes de *software* do ambiente de testes. Serão apresentadas as ferramentas de terceiros que foram utilizadas para compor algumas funcionalidades como controle de frequências, módulo de comunicação com o Apache e aquisição de potência pelo LabVIEW. Em seguida, será apresentado o sistema ESSenCe que implementa a proposta de fato, integrando os demais subsistemas.

5.1 Controle local de frequências

Como foi visto na Seção 3.2, o *front-end* especifica as frequências contínuas que devem estar em efeito em cada *worker*. O chaveamento entre duas frequências adjacentes não convém de ser feito pelo próprio *front-end*, por causa do seu curto período (100ms), especificado na Seção 4.2.1. O *overhead* de mensagens na rede e processamento no *front-end* seria um gargalo para o sistema. Para efetuar a aplicação destas frequências contínuas, é necessário um controlador local em cada máquina.

Para realizar essa tarefa, foi empregada uma ferramenta chamada `DVS_daemon`[1]. Esse *software* foi projetado de forma a minimizar qualquer *overhead* causado pela transição de frequências. O processo do `DVS_daemon` é criado com alta prioridade, para garantir o funcionamento de troca de frequências mesmo se o servidor estiver saturado de requisições. Além disso, sua codificação é feita em C, por causa da sua conhecida capacidade de gerar código que minimiza a utilização de recursos da máquina.

Além de executar o controle de frequências, o `DVS_daemon` é responsável por moni-

[1]De autoria de Carlos Sant' Ana [34] e cedida cordialmente para o uso neste sistema. A sigla DVS foi mantida, no lugar de DVFS, por motivos históricos.

torar e enviar periodicamente informações da estação que ele está instalado até o *front-end*. Essas informações incluem frequência e utilização médias do processador.

5.2 Interface com servidor Apache

Para realizar o controle desejado no sistema é preciso obter em tempo real informações do servidor Apache, como carga atual, tempo de resposta e requisições respondidas. Além disso, é preciso modificar os pesos do balanceador de carga (`mod_proxy_balancer`) em tempo de execução, como previsto nas políticas ADD e CDS (ver Capítulo 3).

Para tanto, foi criado um módulo Apache de nome `frontend_module`[2]. Esse módulo é carregado na inicialização do Apache, na máquina que serve como *frontend*. A comunicação entre este e outros programas é realizada através de um servidor XML-RPC, embutido no módulo. Os serviços providos por este servidor são: manipulação do balanceador de carga e obtenção de medidas como tempo de resposta de cada requisição e carga do sistema.

5.3 Aquisição de potência

Na Seção 4.2.3 foi feita uma referência breve de como é organizada a aquisição de potência via DAQ e LabVIEW. Nesta Seção alguns detalhes de implementação serão discutidos.

A programação no LabVIEW usa o conceito de VIs (Virtual Instruments) e programação visual. A sua programação é feita conectando fios a blocos numa configuração que lembra um diagrama de fluxo. O VI construído consiste de um programa que mede a potência usando um *driver* chamado NI-DAQmx. A função do programa é esperar por conexões em um *socket* TCP. Quando uma conexão é feita, uma medida de potência é enviada (a última). A abordagem utilizada se baseia em trabalhos anteriores: [8, 28, 34].

5.4 ESSenCe

Nesta Seção será apresentado o sistema ESSenCe: *Energy-aware System for Server Clusters*. Esse sistema é responsável por executar os experimentos realizados no ambiente de testes.

[2]Esse *software* é de autoria original de Vinicius Petrucci [28], e pode ser obtido em `http://www.ic.uff.br/~vpetrucci/master.html`.

O código do ESSence é escrito em Python [39], uma linguagem dinâmica com suporte a orientação a objetos. O Python é uma linguagem poderosa, com um enorme acervo de bibliotecas disponíveis. O foco dessa linguagem está na simplicidade do seu código e minimalismo da sua sintaxe, permitindo que o desenvolvedor se preocupe apenas com o problema que tenta resolver.

Inspirado nessa filosofia, o ESSenCe procura prover uma camada de alto nível para a implementação de políticas de economia de energia em *clusters* de servidores. Dessa forma, o pesquisador pode se concentrar em testar e desenvolver suas ideias ao invés de se preocupar com detalhes de implementação.

No desenvolvimento desse sistema está sintetizada uma grande parcela de conhecimento prático sobre o gerenciamento de energia em *clusters* de servidores e uma implementação concisa. Através do sistema, garante-se que o conhecimento e *know-how* seja reusado por outros pesquisadores sobre o tema. Além disso, acredita-se que a ferramenta possa servir como material de estudo sobre o tema.

O ESSenCe conta também com um modo de simulação, permitindo que testes de várias horas sejam realizados em segundos. Pode-se calibrar alguns parâmetros do sistema e testar ideias novas, sem precisar gastar horas de experimentos em um ambiente real. Esse aspecto será aprofundado na Seção 5.4.5.

O sistema foi pensado desde o início para a extensão. Através do isolamento de responsabilidades obtido pelos componentes integrantes do sistema, é possível estender e adaptar as suas funcionalidades para outros fins não previstos inicialmente.

A Figura 5.1 ilustra os componentes principais do ESSenCe. Em seguida, serão detalhadas as funções de cada módulo.

5.4.1 *Cluster* e servidores

Talvez o ponto de partida mais intuitivo para modelar o sistema seja através dos servidores e do *cluster*, os objetos físicos do ambiente de testes. As ações que se deseja que o servidor seja capaz de realizar são: desligar ou ligar a máquina física que ele representa e executar a iniciação de outras ferramentas, como o Apache e `DVS_daemon`.

Cada objeto Worker possui um objeto Processor associado. Nesse último, são armazenadas informações de utilização e frequência da CPU. Há um método para detectar automaticamente quais frequências estão disponíveis em *hardware*. Esse objeto tem seus campos atualizados através do ResourceMonitor.

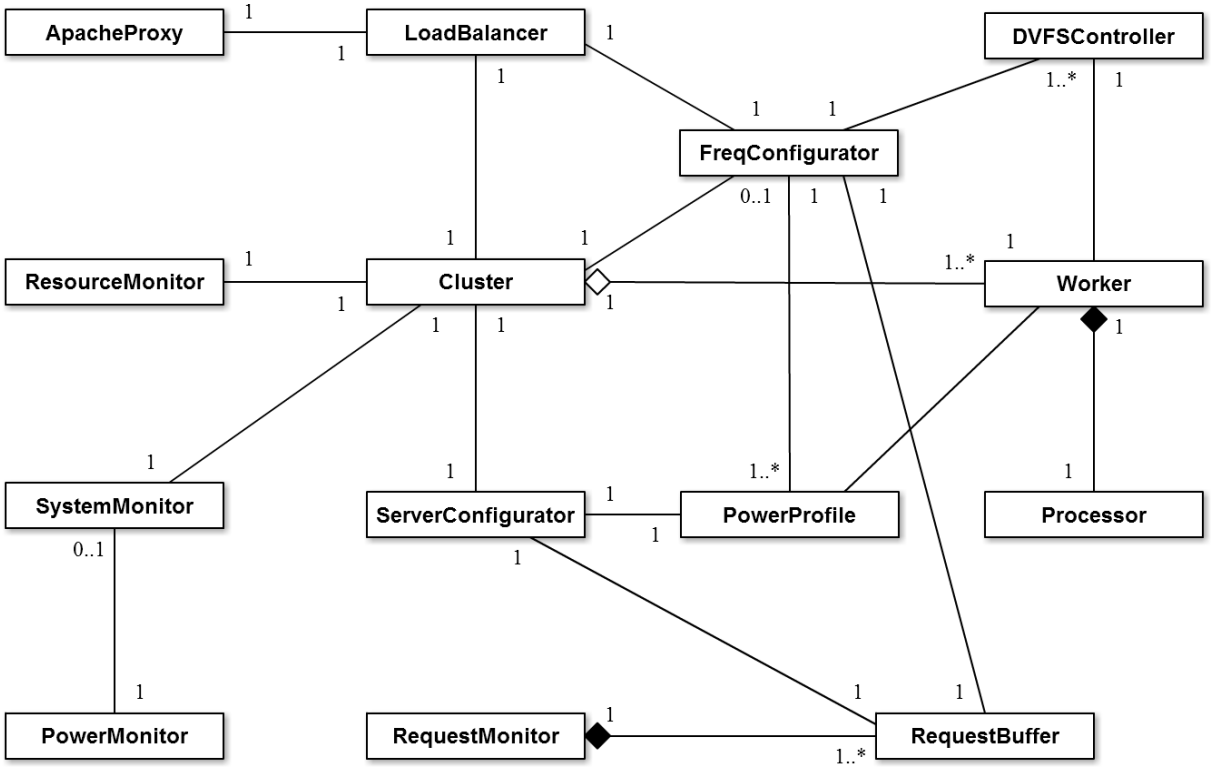


Figura 5.1: Diagrama UML dos principais componentes do ESSenCe

Uma outra característica desejada é a capacidade de associar um controlador local de DVFS ao Worker. A classe DVFSController especifica que tipo de controlador será usado. O módulo FreqConfigurator pode lidar com qualquer subclasse de DVFSController, desde que ela contenha um limite ajustável para a frequência máxima do Worker. Atualmente existem dois controladores de DVFS implementados (não mostrados na Figura 5.1), o controle de frequências contínuas, implementado através de trocas de mensagens com o `DVS_daemon`, e o controlador `ondemand`, que não foi explorado neste trabalho.

Um *cluster* é, por definição, composto de vários servidores (*workers*). O Cluster é responsável por gerenciar objetos Worker e seus estados (`ON`, `OFF`, `TURNING_ON` e `TURNING_OFF`), associando cada estado à uma lista para facilitar a consulta por outros módulos. A classe Cluster define métodos para desligar e ligar um Worker. A diferença principal entre esses métodos e os já presentes em Worker é que eles realizam chamadas para balancear a carga (através de LoadBalancer), quando apropriado. Por exemplo, antes de efetivamente dar a ordem de desligar um Worker, pede-se que LoadBalancer pare de direcionar carga para esse servidor.

5.4.2 Configuração de servidores e frequências

A implementação dos esquemas CDS e ADD foram mapeados em duas classes: ServerConfigurator e FreqConfigurator, respectivamente. Ambas as classes são construídas como *threads* para que os dois mecanismos possam agir de maneira independente.

Um módulo comum entre as duas classes é o PowerProfile, que está associado à cada Worker. Esse módulo guarda a relação entre potência e performance de cada frequência de operação do servidor, como as indicadas na Seção 4.5. Várias chamadas são feitas a ele para realizar conversões entre frequência, vazão máxima e potência.

Amostras de carga (e outras medidas) são armazenadas em RequestBuffer, uma fila circular. Outros módulos podem consultar um número n de amostras do *buffer*, começando da mais recente para a mais antiga. Tal comportamento é útil para se obter o valor médio de carga que serve de entrada para os algoritmos de ServerConfigurator e FreqConfigurator. Nos experimentos que foram feitos, só foi considerado um tipo de requisição, necessitando apenas de uma instância de RequestBuffer.

Um servidor está sempre presente em alguma lista de estados. Quando um servidor ligado é desativado, há uma transição de `ON` para `TURNING_OFF` e em seguida, quando o servidor estiver realmente desligado, de `TURNING_OFF` para `OFF`. Para evitar problemas de colisão, a classe Cluster é protegida com semáforos para prover exclusão mútua nas listas de estados. Para ilustrar uma colisão, considere o caso onde ServerConfigurator envia uma ordem ao Cluster para desligar um determinado Worker. Enquanto o processo de desativação está em andamento, FreqConfigurator atua para ajustar as frequências dos servidores. Pode haver casos onde FreqConfigurator considera o servidor que está para ser desligado, alocando desempenho para o mesmo. Isso é um erro, podendo levar à saturação do sistema. Os eventos de desligar um servidor e alterar sua frequência não podem ocorrer ao mesmo tempo. Para contornar esse problema, a transição e acesso a esses estados são feitos em áreas de exclusão mútua.

5.4.3 Monitoramento

Uma parte importante do sistema são os mecanismos de aquisição de medidas. No ESSenCe, esses módulos são identificados com o sufixo *monitor*. Cada monitor é implementado através de uma *thread*, e seu comportamento consiste em obter periodicamente uma amostra de alguma variável do ambiente de testes. Alguns módulos obtém as medidas e atualizam campos de objetos de outras classes, enquanto outros guardam o valor da

medida na sua própria instância.

RequestMonitor tem a função de amostrar o nível de requisições, tempo médio de atendimento e o número de requisições respondidas. Ele separa essas estatísticas de acordo com o tipo de requisição e transfere a amostra para o RequestBuffer correspondente. Novamente, deve-se lembrar que neste trabalho só será considerado um tipo de requisição. Ele desempenha essa função através de chamadas remotas ao `frontend_module`, que por sua vez tem um representante no sistema chamado ApacheProxy. Observe que a ligação entre ApacheProxy e RequestMonitor não está representada na Figura 5.1.

O módulo ResourceMonitor tem a tarefa de obter medidas sobre os recursos de cada *worker*, como utilização e frequência média. Na verdade, ele espera por mensagens do `DVS_daemon` instalado em cada Worker, e atualiza os respectivos campos dos mesmos com as amostras.

Para obter a potência dos servidores, utiliza-se o PowerMonitor. Atualmente, existem duas subclasses para ele: PowerEmulator e LabVIEW. O primeiro emula a potência através de medidas de utilização e frequência dos *workers*. Já o segundo, realiza uma interface com o LabVIEW para colher medidas de potência (ver Seção 4.2.3).

Por causa da grande quantidade de monitores é necessário sincronizar e agrupar as medidas para análise posterior, como geração de gráficos e estatísticas. Pensando nisso, foi criado o SystemMonitor, um monitor geral do *cluster*. Ele é responsável por medir todas as variáveis relevantes periodicamente e gerar um arquivo com o *trace* de todo o experimento.

5.4.4 Gerência dos experimentos

Para realizar experimentos no ambiente de testes usando o ESSenCe, é muito simples. Basta descrever os parâmetros do teste em um arquivo texto, no formato `INI` [41]. Esse arquivo de configuração é lido por um módulo especializado em criar experimentos. Ao final da leitura do arquivo, um objeto Experiment é criado. Veja a Figura 5.2, onde estão descrito três objetos do sistema.

A classe Experiment é responsável por iniciar e finalizar experimentos no ambiente de testes. Cada instância de Experiment contém todos os objetos da Figura 5.1 (e outros, não mostrados por simplicidade). Através do encapsulamento de todo o funcionamento do sistema nessa classe, vários experimentos podem ser criados e executados em sequência. Para tanto, basta que vários arquivos de configuração sejam criados.

```
[ServerConfigurator]
loopPeriod:  20
windowSize:  20
gamma:  0.9
```

```
[PowerMonitor]
type:  labView
port:  6971
ip:  192.168.10.3
```

```
[Ampere]
idlePower:  [66.3,70.5,72.7]
busyPower:  [81.5,101.8,109.8]
throughput:  [170.0,309.9,346.8]
```

Figura 5.2: Trechos do arquivo de configuração

Uma das muitas funções de Experiment, é gerar o relatório final sobre o experimento. Esse relatório indica se o experimento foi realizado com sucesso, foi abortado ou falhou. O usuário pode abortar o experimento através da combinação de teclas `CTRL+C`. Nesse caso, todos os experimentos que estiverem pendentes serão ignorados. Ainda, um experimento pode falhar, devido a um erro no programa. Graças ao encapsulamento de experimentos, caso um deles apresente erros, esse não afetará os demais que estão pendentes.

Caso o experimento tenha falhado, o usuário do sistema pode investigar a causa do problema no arquivo de *log*. Esse arquivo possui informações sobre cada ação importante do sistema, como troca de frequências e chaveamento de servidores. Caso ocorra uma exceção não tratada, o arquivo de *log* conterá uma cópia da pilha de execução (*stack trace*), bem como o nome do erro. A Figura 5.3 mostra o *log* da iniciação de um experimento. Observe que acompanhado da mensagem encontra-se o seu tipo e o nome da classe que originou a entrada no *log*.

```
2011-06-10 12:04:34,019: Experiment   : INFO : Clearing access_log...
2011-06-10 12:04:34,035: Experiment   : INFO : Starting Apache...
2011-06-10 12:04:34,582: Cluster      : INFO : Turning ON: ampere...
2011-06-10 12:04:34,587: Server       : INFO : ampere is now ready.
2011-06-10 12:04:34,590: LoadBalancer : INFO : Balancing servers...
2011-06-10 12:04:34,591: Cluster      : INFO : Turning ON: coulomb...
```

Figura 5.3: Trecho de um *log* da inicialização de um experimento

Além do *status*, informações estatísticas vão para o relatório final, como potência média, tempo de experimento e outras informações pertinentes. Esse arquivo, o *log* e o próprio arquivo de configuração são salvos em uma pasta com identificação única, para consulta posterior, ao final de cada experimento.

5.4.5 Simulação

Experimentos em ambientes reais são muito importantes para validar trabalhos como este. Porém, simulações podem ajudar em uma série de aspectos. Primeiramente, a análise de sensibilidade de algumas variáveis do sistema pode ser realizada através de simulações. Neste trabalho, os tempos de atuação dos configuradores foram determinados à partir de testes simulados, por exemplo. Além disso, é possível testar outros atributos importantes, como *overhead* de processamento das políticas de energia e tolerância a falhas de *software*. Outra vantagem é a possibilidade de se aplicar testes em diferentes cenários, apenas especificando diferentes arquivos de configuração.

O sistema ESSenCe foi codificado em Python, uma linguagem com implementações em várias plataformas. Por conta disso, o sistema é capaz de rodar em modo simulação em vários sistemas operacionais[3].

Foi possível aproveitar a maior parte do código do sistema na implementação do simulador[4], como pode ser visto na Figura 5.4. A solução adotada consiste de criar subclasses de Server, LoadBalancer e outras classes que se comunicam com o ambiente de testes real. Através dessas subclasses, os métodos que realizam de fato alguma ação no ambiente de testes, como enviar um quadro Wake-on-LAN (ver Seção 4.2.2), são sobrescritos com métodos que não fazem ação alguma. Dessa forma, o programa segue seu fluxo normal como se esses comandos tivessem ocorrido com sucesso em um ambiente real.

Todas as *threads* do sistema executam periodicamente. Seja para configurar servidores, medir potência ou outras tarefas. Para manter essa periodicidade, as *threads* realizam uma chamada à função $sleep(s)$ do Python, onde s é um intervalo de tempo em segundos. Caso essa função também fosse chamada no simulador, o experimento teria a mesma duração aproximada de um experimento realizado no ambiente de testes, derrotando a proposta de se poder realizar testes em poucos segundos. Para solucionar esse problema, foi desenvolvido um dispositivo para controlar o fluxo de execução das *threads* do sistema. Esse mecanismo substitui a função original *sleep*, por uma outra função chamada *sleep′*. Esta última, por sua vez, controla o fluxo de execução das *threads* periódicas do sistema. Quando uma *thread* faz uma chamada à *sleep′*, ela fica suspensa. A *thread* será liberada somente se s chamadas à $tick()$, uma outra função com o propósito

[3]Atualmente, o sistema foi testado apenas no Microsoft Windows 7 e Linux Ubuntu 10.10.

[4]A contagem de linhas de código foi feita utilizando a ferramenta `Cloc`, disponível em `http://cloc.sourceforge.net/`.

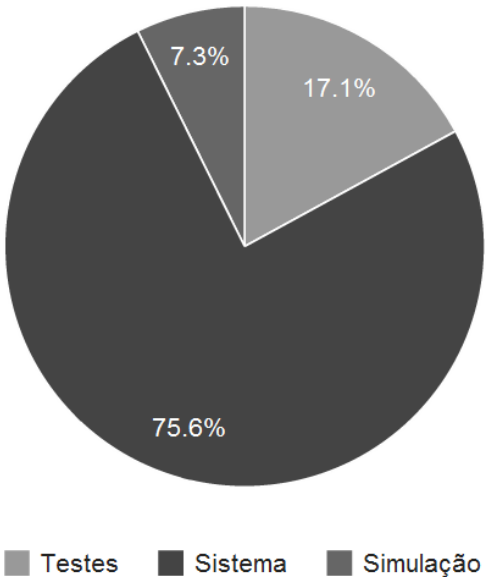


Figura 5.4: Linhas de código em cada área do sistema

de avançar uma unidade no tempo, forem realizadas. Agora, basta chamar *tick*() durante a execução do experimento, quando todas as *threads* estiverem sincronizadas e presas na função *sleep'*. Para mais detalhes sobre esse mecanismo, consulte o Apêndice A.

Algumas tarefas, como ativamento de servidores, demoram um tempo para serem realizadas. Através do mecanismo exposto anteriormente, é possível emular facilmente essas situações. Para tanto, basta inserir chamadas de *sleep'* com um tempo estimado de duração destas tarefas. Neste trabalho, estimou-se 6 e 5 segundos para o ativamento e desativamento de servidores, respectivamente. Outros atrasos presentes no ambiente real foram desconsiderados.

Um dos pontos chaves da simulação é estimar a potência gasta em diferentes casos. Dessa forma, pode-se investigar métodos mais eficientes que sejam posteriormente validados no ambiente de testes real. Para tanto, foi criada uma subclasse de PowerMonitor chamada PowerEmulator. Essa subclasse estima a potência gasta pelas máquinas do aglomerado através de uma interpolação linear entre os pontos de potência medidos, em função da utilização de cada máquina. Detalhes desse processo de interpolação seguem o raciocínio explicitado na Seção 3.2. Além disso, há uma potência gasta por máquinas submetidas ao Suspend-to-RAM, estimada em 6W.

A Figura 5.5 mostra uma comparação entre a potência emulada e a real para um mesmo experimento. Repare que, mesmo divergindo em alguns trechos, os dois experimentos seguem o mesmo perfil de potência. Isso confirma também que ações da CDS e ADD também seguem o comportamento esperado em um experimento real.

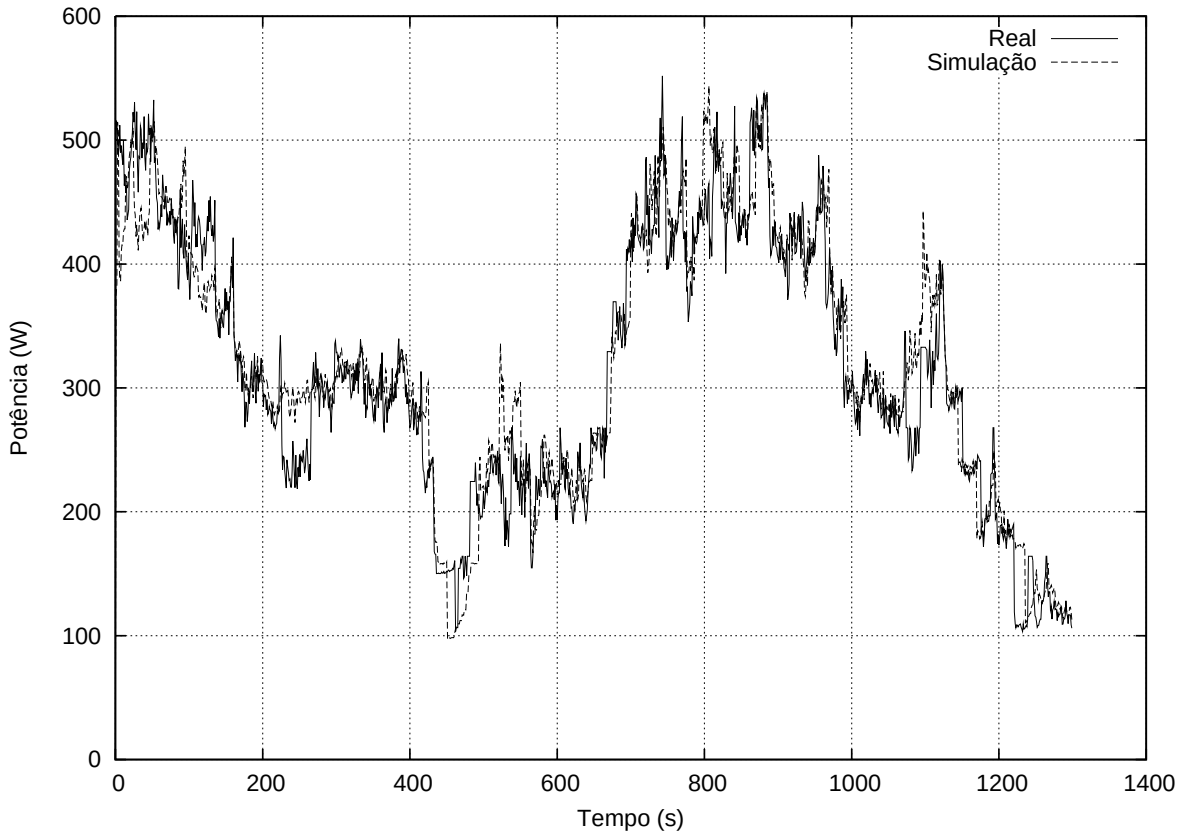


Figura 5.5: Simulação de potência no ESSenCe

5.5 Sumário

Neste Capítulo foi apresentado o ESSenCe, a infraestrutura de *software* usada para implementar a proposta deste trabalho. Foi visto que a arquitetura desse sistema visa o reuso e simplicidade. Uma característica importante é o modo de simulação, que consegue replicar experimentos que ocorrem no ambiente real, com fidelidade aceitável.

Uma vez que foi explicitada a implementação de todo o sistema, tem-se condições de avaliar a proposta através de experimentos. No capítulo que segue, serão abordados a metodologia e resultados dos experimentos realizados no ESSenCe.

Capítulo 6

Experimentos

Neste Capítulo será discutida a metodologia adotada para validar a proposta do trabalho e apresentados os resultados dos experimentos no ambiente de testes. Primeiramente, serão apresentados na Seção 6.1 os perfis de carga (*traces*) utilizados nos experimentos. Em seguida, na Seção 6.2, discute-se a medida de controle adotada, a saturação, e o seu impacto no atraso médio das requisições. Na Seção 6.3 foi medida a economia de energia obtida comparando-se o método proposto com uma situação onde nenhuma política de energia é usada no *cluster*. Por fim, foi realizado um teste comparativo para avaliar a economia obtida entre a política ADD e o `ondemand`.

6.1 *Traces*

Para avaliar a proposta de maneira adequada é necessário trabalhar com perfis de carga que sejam representativos de aplicações reais. Emular o comportamento de usuários ao utilizarem uma aplicação *Web* é uma tarefa que pode ser complexa. Ao invés disso, foi adotada uma estratégia diferente. Recriou-se o comportamento dos usuários a partir de *logs* de acesso aos servidores.

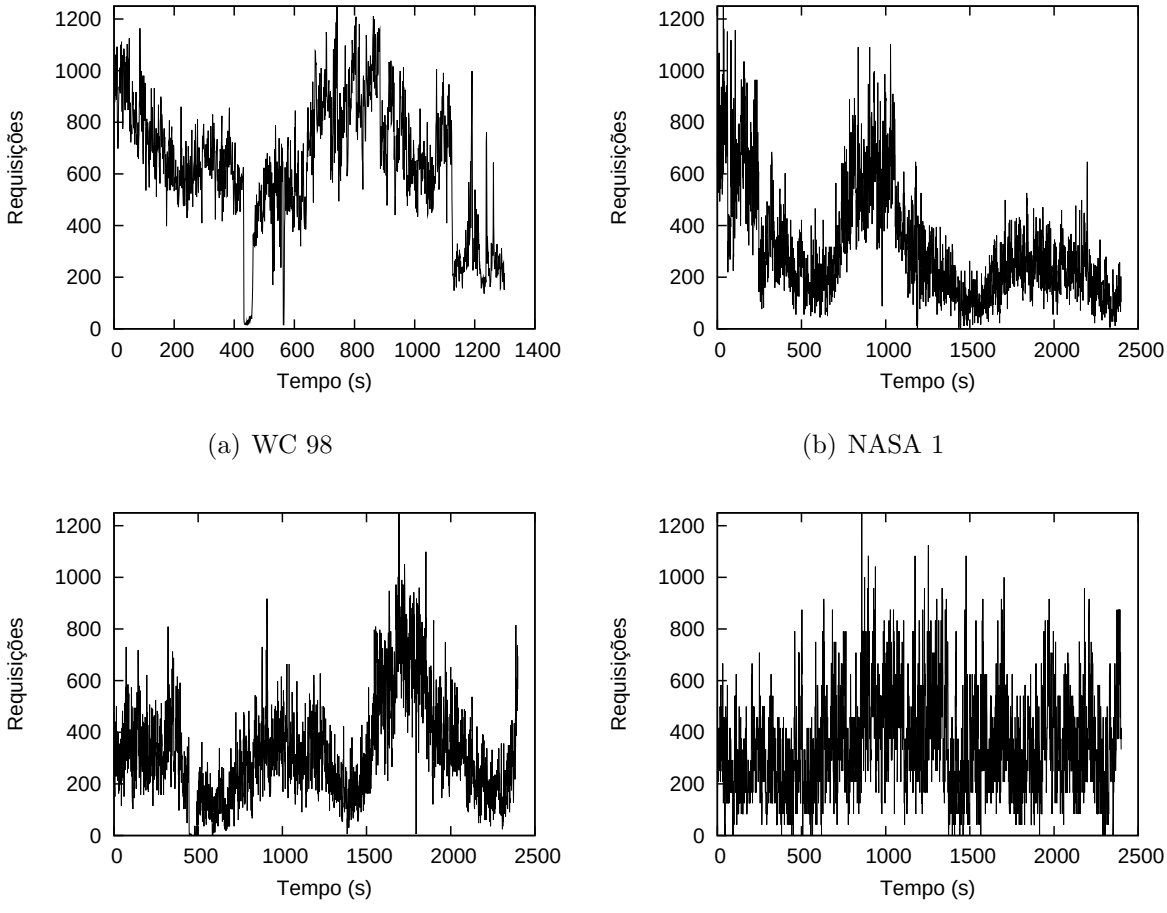
Esse *log* consiste de um arquivo texto onde são anotadas algumas informações sobre as requisições respondidas pelo servidor *Web*, como data e horário de recebimento e código de *status* HTTP [40]. O tráfego de *clusters* reais, como o servidor *Web* da NASA e a Copa do Mundo de Futebol de 1998 (WC 98) possuem *logs* publicados em [22].

De posse do arquivo de *log*, filtraram-se as linhas de modo que apenas requisições atendidas com código de *status* 200 (`OK`) restassem. Depois, contou-se o número de requisições que ocorriam por segundo, gerando o *trace* de acessos. Um *trace* como esse é usado para indicar ao gerador de carga (vide Seção 4.3.1) o número de requisições que devem

ser feitas à aplicação de testes (calculo de números primos).

Os *traces* de acesso gerados apresentam o nível de requisições para várias semanas. Realizar experimentos com o *trace* inteiro seria impraticável e desnecessário, já que um período de algumas dezenas de minutos é suficiente para avaliar a proposta. Por isso, foram selecionados apenas trechos dos *traces* originais. Outra observação importante é que o ambiente de testes deste trabalho não possui a mesma capacidade de processamento das máquinas onde as requisições foram processadas. Dessa forma, foi preciso dimensionar o número de requisições em cada *trace* para a capacidade de processamento disponível, utilizando uma normalização pela quantidade máxima de requisições.

A Figura 6.1 mostra os trechos dos *traces* selecionados. Foi escolhido um trecho da aplicação WC 98 e outros três do *site* da NASA. Cada *trace* foi escolhido por apresentar distribuição de cargas distintas, o que é oportuno para testar a proposta em diferentes cenários. Outros trabalhos como [31] usaram apenas um *trace* para validar suas propostas, colocando em disputa a proposição de que seus resultados podem ser obtidos em diferentes situações.



(a) WC 98

(b) NASA 1

(c) NASA 2

(d) NASA 3

Figura 6.1: *Traces* adotados nos experimentos

6.2 Controle de saturação

Na Seção 6.2 foi argumentado a favor de um controle de saturação do aglomerado como maneira de atingir um nível de qualidade de serviço (QoS) desejado. Nessa Seção, será demonstrado que não só o método proposto é capaz de controlar a saturação do aglomerado, como também esse controle determina o cumprimento de um certo nível de QoS.

Antes de continuar a análise, é necessário definir formalmente a medida de QoS estudada. A métrica adotada foi a porcentagem de vezes onde o tempo de atendimento de uma requisição é de até 100ms. Como o QoS é calculado a cada segundo, basta verificar o tempo de atendimento médio das requisições em cada instante. Quando o *cluster* se encontra com pouca saturação, o tempo médio de atendimento é de cerca de 5ms. O *deadline* escolhido é de 20 vezes esse valor, totalizando 100ms. Dessa forma, o QoS de um instante é um (1) se o tempo de atendimento for inferior a 100ms, e zero, caso contrário.

Na Seção 6.2 abordou-se o controle de saturação através dos parâmetros γ e σ. Combinações de valores diferentes dessas duas variáveis podem produzir a mesma saturação média resultante. Além disso, o tipo de carga também influencia na proximidade entre o valor obtido e esperado. Caso a política CDS subestime a carga várias vezes, a política ADD ficará comprometida. Independentemente do valor de σ, não é possível alcançar o nível de saturação desejado se a configuração de servidores em questão estiver saturada. O mesmo raciocínio se aplica para um caso onde a quantidade de servidores disponíveis é suficiente para atender a carga, mas o controle do ADD não provê um desempenho adequado.

Visando uma interação harmônica entre CDS e ADD, definiu-se a saturação alvo com o mesmo valor de γ e σ. Por exemplo, caso o valor alvo seja $0,7$, ajusta-se $\gamma = \sigma = 0,7$. Nos testes conduzidos obteve-se um controle adequado de saturação através desse método. A Figura 6.2 mostra um gráfico com a dispersão entre o valor obtido e o alvo. Nota-se que a dispersão varia tanto em função do *trace* quanto do valor alvo.

Uma vez definida a maneira de atingir a saturação alvo, pode-se buscar um mapeamento entre a mesma e a QoS resultante. A Figura 6.3 mostra essa relação. A princípio, não há um mapeamento claro entre as duas grandezas. Entretanto, ao examinar atentamente a saturação abaixo de 0,7, observa-se que são produzidos níveis de QoS similares para todos os *traces*, com exceção de NASA 3. Além disso, valores acima de 0,7 geram uma QoS menor do que 90%, e não são de interesse. Normalmente uma QoS contratada para um serviço *Web* é maior do que 90% [8, 33, 34].

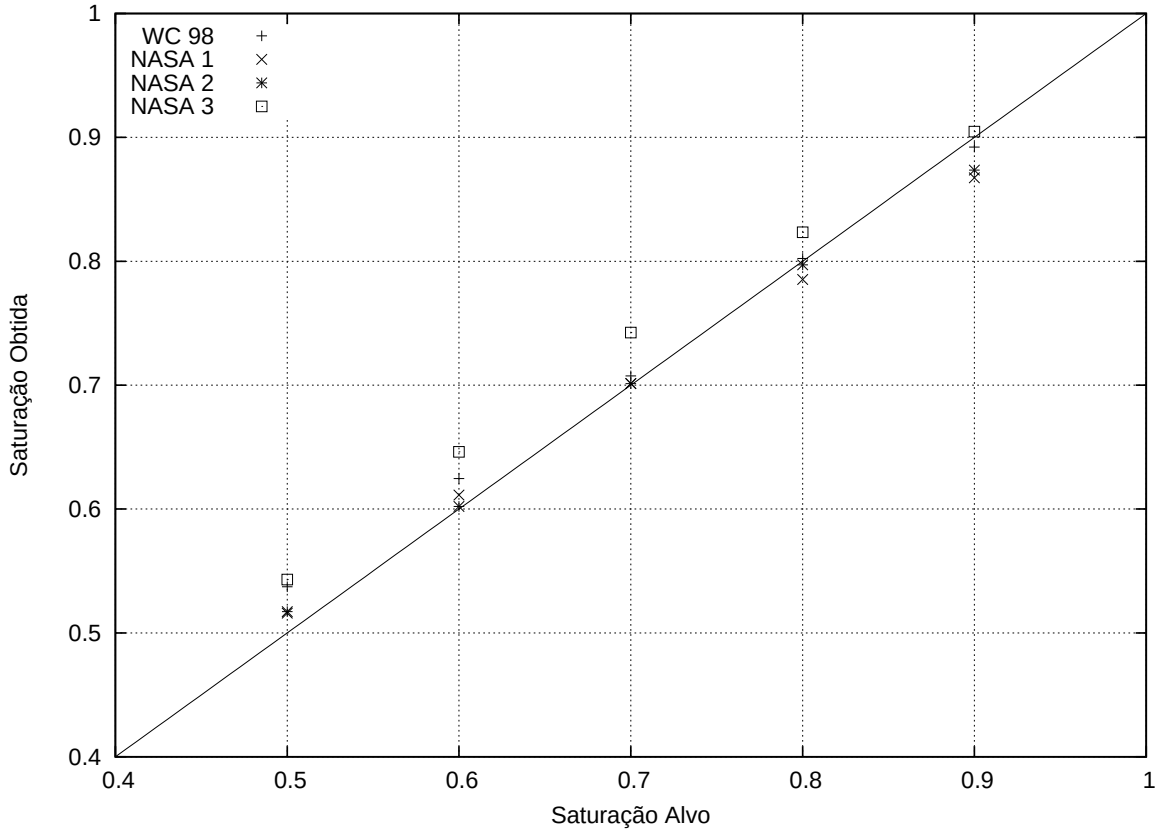


Figura 6.2: Saturação alvo e a obtida

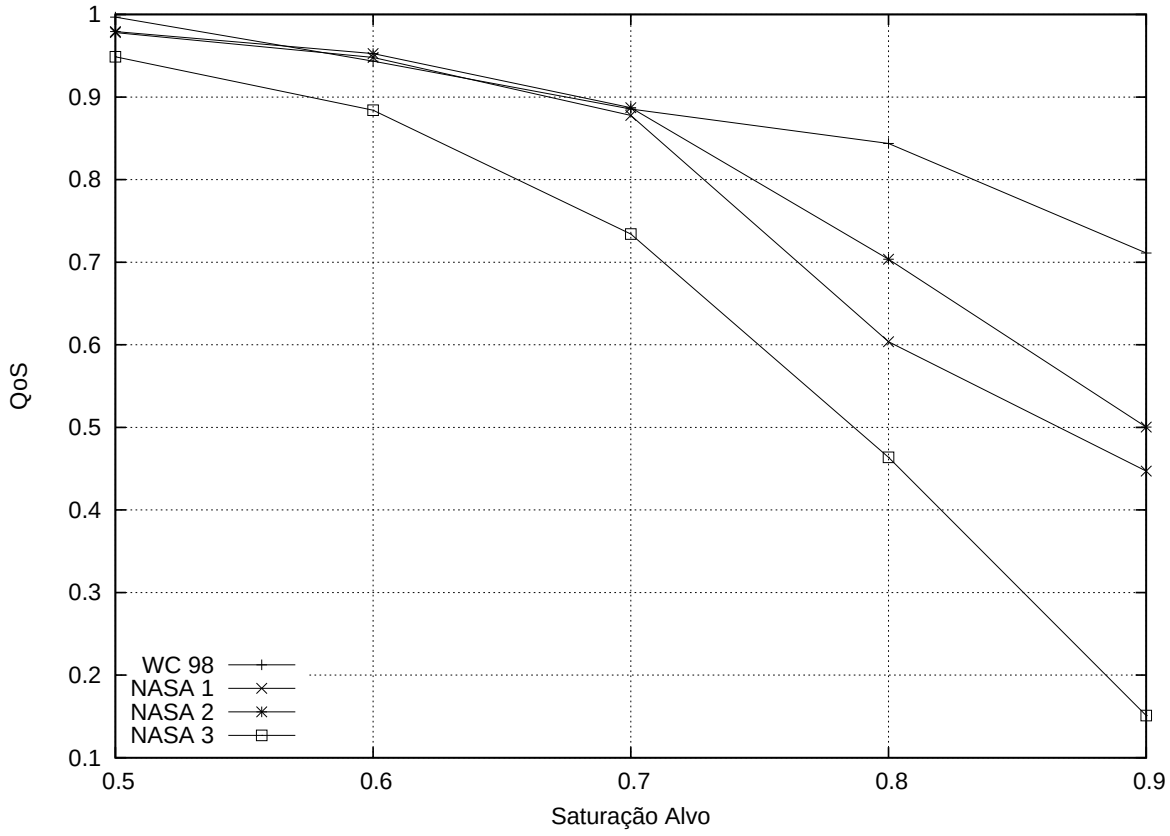


Figura 6.3: Relação entre saturação e QoS

Uma hipótese pela qual as curvas divergem a partir do ponto de saturação 0,7 é atribuída a CDS. A medida que se aumenta o valor de γ, diminui-se a capacidade de antecipar carga da CDS às custas de uma economia maior de energia. Com isso, subestimações começam a impactar a QoS de uma maneira muito severa, porém de um grau diferente para cada *trace*. Uma maneira de mitigar esse efeito é criar uma assimetria entre γ e σ, diminuindo o valor relativo de γ para um grau onde ocorram menos subestimações. Essa assimetria pediria uma análise mais complexa entre γ e σ, e será deixada a encargo de trabalhos futuros.

6.3 Comparação com o Modo Performance

Nesta Seção será feita uma comparação entre a política proposta (Proposta) e o Modo Performance (MP). Este último, é o modo no qual as máquinas se encontram com a sua frequência máxima e permanecem ligadas o tempo todo. Nenhum método de economia de energia é usado nesse modo. Essa comparação é válida quando se obedece um nível de qualidade de serviço acordado. Por esse motivo, será definido uma QoS alvo de 95% para os experimentos que se seguem.

As Figuras 6.4– 6.7 mostram uma melhoria significativa em relação ao Modo Performance. O perfil de potência da Proposta chega a relembrar o perfil de carga dos *traces* da Figura 6.1. Isso indica que o método foi capaz de dimensionar a potência e a performance do aglomerado para acompanhar a demanda de cada *trace*. A Tabela 6.1 mostra a economia gerada em cada experimento em relação ao MP. Pode-se perceber que a economia chega a valores de mais de 50% em alguns casos. Mesmo assim, a QoS obtida se mantém próxima do alvo desejado de 95%.

Tabela 6.1: Economia de energia em relação ao Modo Performance

Trace	**Sat. Alvo**	**Energia MP**	**Energia Proposta**	**Economia**	**QoS**
WC 98	0,6	160,61Wh	121,61Wh	24%	0,94349
NASA 1	0,6	277,01Wh	119,72Wh	57%	0,94787
NASA 2	0,6	279,51Wh	125,03Wh	55%	0,95269
NASA 3	0,5	281,88Wh	149,31Wh	47%	0,94877

No *trace* NASA 3 foi necessário utilizar um valor diferente de saturação alvo (0,5). Esse *trace* é importante para testar os limites da Proposta. Quando confrontada com um

trace de variações bruscas e frequentes como o NASA 3, o uso de $\gamma = \sigma = sat$ não permite que esse experimento obtenha uma relação de saturação e QoS parecida com os demais. Para cargas de trabalho com esse perfil, é necessário uma relação mais robusta entre γ, σ e sat. Esse tipo de análise será deixada para trabalhos futuros.

6.4 Comparação com o Modo Ondemand

Nesta Seção será comparado o esquema centralizado de troca de frequências desenvolvido (ADD), com a política local de DVFS presente nos sistemas Linux, o `ondemand`. Como foi explicado na Seção 4.2.1, o `ondemand` é um controle ativado por padrão na maior parte das distribuições Linux modernas, e tem como objetivo manter a CPU em um certo nível de utilização.

Outros trabalhos [28, 29, 34] não fizeram uso da variável β em suas avaliações com o *governor* `ondemand`. Isso pode causar comparações injustas em relação ao controlador e a política proposta, já que seu potencial não foi devidamente explorado. Essa comparação é importante para justificar o custo e esforço de manter uma política centralizada para gerenciamento de frequências, visto que o `ondemand` já está implementado e em uso na maioria dos ambientes Linux atuais.

Primeiramente foram feitos experimentos com o valor de $\beta = 1$, já que esse é o valor padrão do controlador. Todavia, esses testes resultaram em QoS = 1 em todos os *traces*. Esses resultados não serão mostrados aqui pois não são próprios para comparação com a política ADD. Quando o QoS é igual a 1, só é possível afirmar que **todas as requisições foram atendidas** no prazo. Porém, não se sabe **com que antecedência** tais requisições foram atendidas. De outra maneira, não se sabe qual é o nível de superestimação que o `ondemand` utilizou, e portanto não é possível construir um comparativo com o ADD. Sendo assim, é necessário ajustar o valor β para obter um QoS menor do que 1, no chamado Modo *Ondemand* (MO).

A comparação que se deseja fazer é apenas entre as políticas de variação de frequência. Não há variação do número de servidores durante o experimento. Portanto, a grandeza apropriada para medir a diferença entre as políticas é a potência dinâmica. Os testes foram realizados utilizando um valor arbitrário de β, que resulta num QoS < 1. A partir desse valor foi buscado um QoS equivalente na política ADD.

A Tabela 6.2 mostra os resultados desses experimentos. Repare que os valores de energia estão numa escala menor, comparados com a Tabela 6.1. Isso já era esperado,

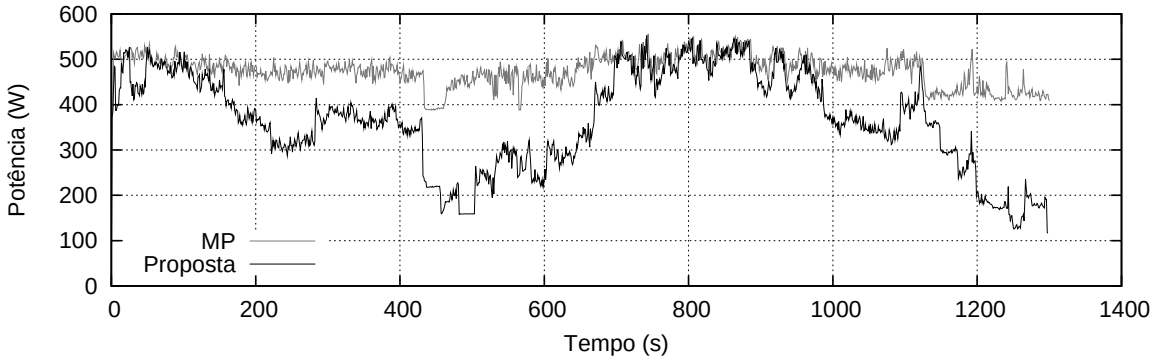


Figura 6.4: WC 98

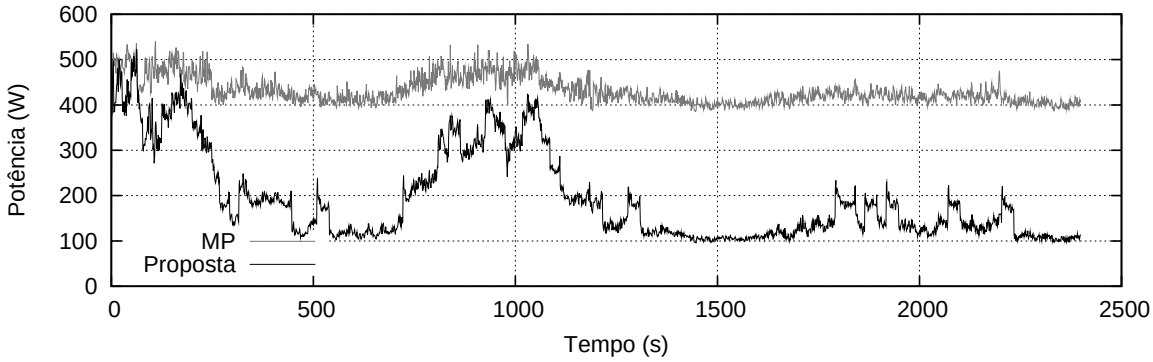


Figura 6.5: NASA 1

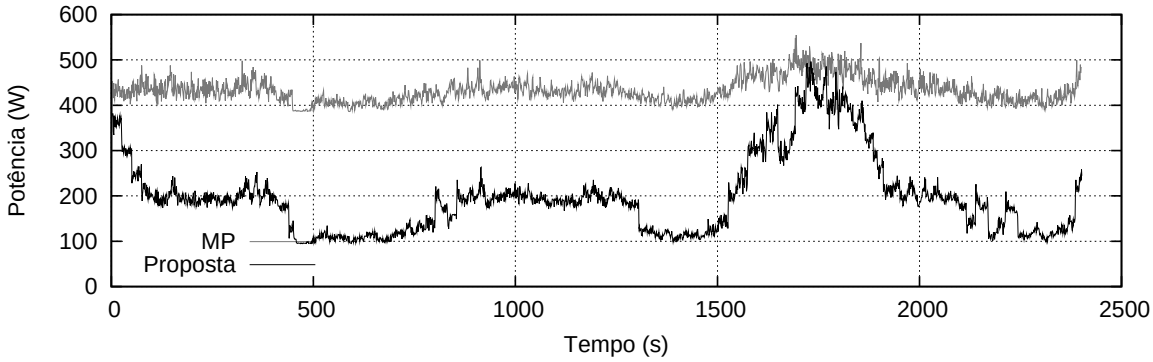


Figura 6.6: NASA 2

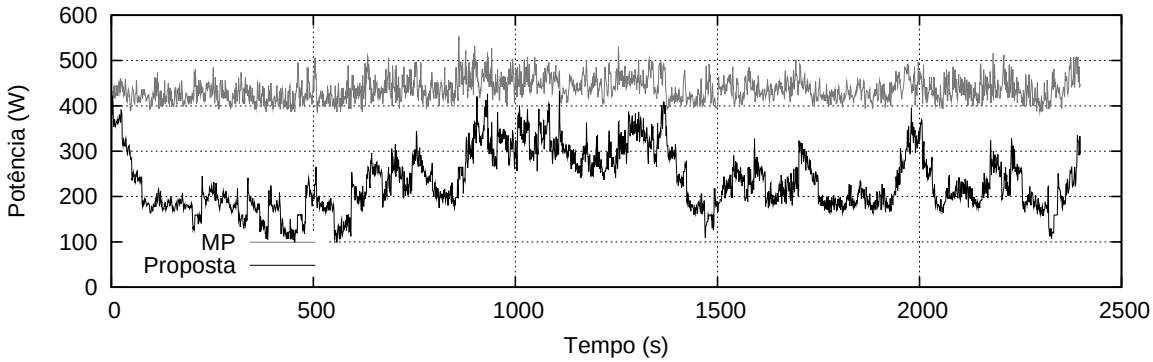


Figura 6.7: NASA 3

Tabela 6.2: Economia de energia em relação ao Modo *Ondemand*

Trace	**Energia ADD**	**Energia MO**	**Economia**	**QoS ADD**	**QoS MO**
WC 98	26,14Wh	27,14Wh	3,7%	0,94522	0,93282
NASA 1	25,06Wh	24,96Wh	-0,4%	0.96367	0.96075
NASA 2	24,37Wh	25,69Wh	5,1%	0,99916	0,99332
NASA 3	24,54Wh	25,30Wh	3,0%	0,95741	0,95492

dada a pequena contribuição da potência dinâmica em relação a potência total dos servidores descritos na Seção 4.5. Entretanto, a medida que se avança na área de eficiência energética em servidores, a potência estática se tornar cada vez menor, ao passo que a contribuição da potência dinâmica passa a ser mais significativa para o gasto energético total do sistema. Isto reafirma a pertinência de se estudar políticas de chaveamento de frequência em servidores, como o ADD.

De acordo com a Tabela, o método ADD se mostrou superior na maioria dos casos. Isso mostra que a distribuição de capacidade entre os servidores de acordo com a potência supera o período de atuação mais curto do MO[1]. É oportuno enfatizar que a economia do ADD sobre o MO é obtida sem nenhuma degradação de performance, como indicado pelas colunas de QoS da Tabela.

6.5 Análise de escalabilidade

Uma das principais premissas dos algoritmos apresentados no Capítulo 3 é a sua escalabilidade. Foi visto nesse capítulo que a complexidade assintótica de cada um é dada em função do número de servidores. Por esse motivo, nesta Seção será apresentada uma análise de escalabilidade da proposta.

O algoritmo CDS é muito rápido, pois corresponde a uma iteração sobre listas de N servidores. Sendo assim, sua complexidade é de apenas $O(N)$. Além disso, sua frequência de atuação é pequena, diminuindo ainda mais o impacto no sistema. Por esses motivos, o seu algoritmo não será analisado aqui.

Já a política ADD apresenta um *overhead* maior. Foi visto que sua complexidade, sem implementar nenhuma das melhorias apontadas, é de $O(N^2 \cdot log(N))$. Como agravante, seu período de atuação é curto: 3 segundos no caso deste trabalho. Por conta

[1]Os períodos são de 3 e 1,24 segundos para ADD e MO, respectivamente.

dessas características, o seu algoritmo se torna o gargalo da proposta.

O experimento de escalabilidade consiste de rodar o algoritmo ADD no seu pior tempo de execução. Isso ocorre quando a demanda é máxima para uma entrada de N servidores. O leitor pode verificar essa afirmação ao analisar o algoritmo da Seção 3.2, página 21. Para criar mais servidores além daqueles estudados no trabalho, replicou-se cada um deles em múltiplos de cinco, para atingir uma certa quantidade de N. O *benchmark* foi rodado na máquina `watt`[2], que serviu de *front-end* para outros experimentos. O algoritmo ADD foi executado na versão 2.7 da linguagem Python.

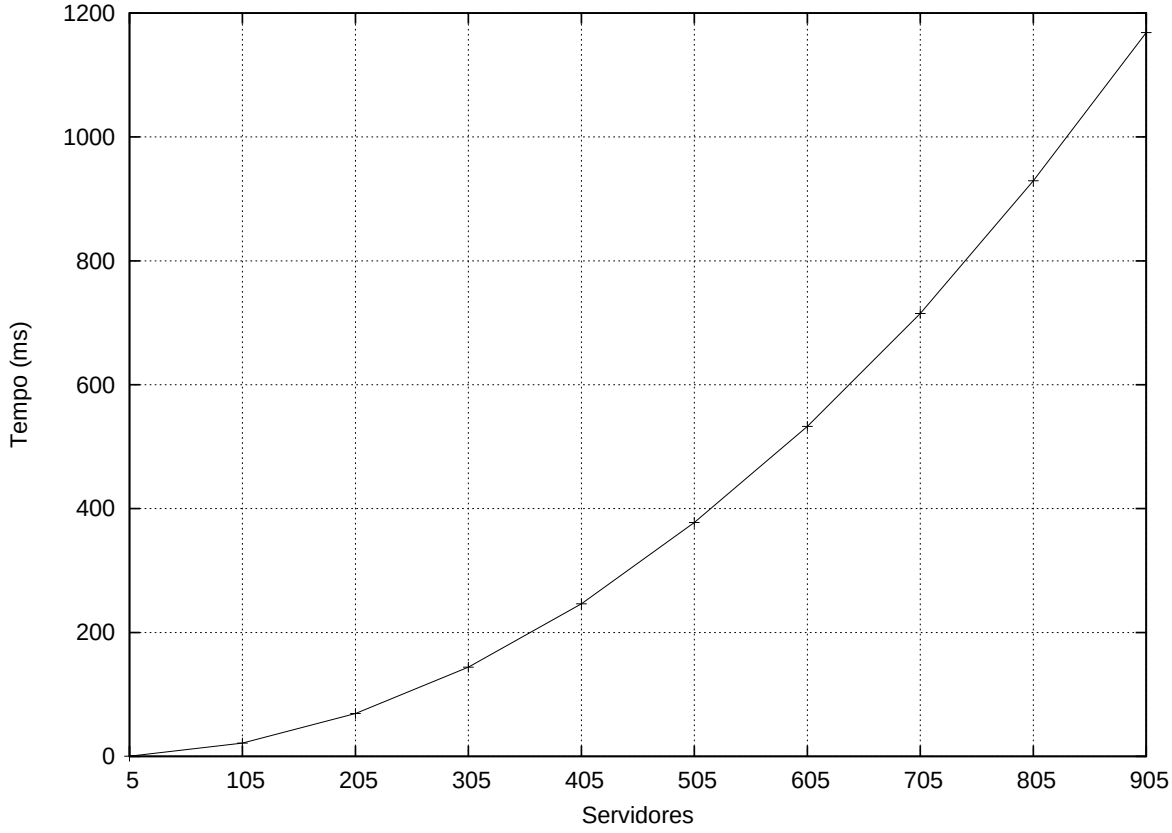


Figura 6.8: Tempo de execução em função do número de servidores

Os resultados podem ser vistos na Figura 6.8. Para cada teste com N servidores, foi executado 1000 vezes o algoritmo ADD e feita uma média entre os resultados[3]. Como é possível observar, o algoritmo proposto consegue processar centenas de servidores em instantes. Em um caso, a política chegou a processar 805 servidores em menos de um segundo. Isso indica uma vasta escalabilidade do método, podendo ser explorada em CPDs de muitos servidores.

[2]Trata-se de um Intel Pentium 4 de 2.8GHz.

[3]As barras de erro foram omitidas, por clareza.

6.6 Sumário

Este Capítulo apresentou os resultados dos experimentos realizados no ambiente de testes com *traces* de servidores reais. Através deles pode-se afirmar que a política proposta demonstrou ser capaz de oferecer um compromisso entre gasto de energia e qualidade de serviço, de acordo com as premissas do Capítulo 3. Além disso, trata-se de uma política leve, que consegue processar uma grande quantidade de servidores com *overhead* mínimo.

O Capítulo de experimentos foi a última peça desta monografia, restando agora sintetizar as suas principais contribuições e conclusões. Em seguida, serão traçados alguns rumos de pesquisa que poderão ser perseguidos a partir deste trabalho.

Capítulo 7

Considerações finais

7.1 Conclusão

Este trabalho desenvolveu uma política de economia de energia para a camada de aplicação de um aglomerado heterogêneo de servidores. A política desenvolvida apresenta duas técnicas de economia de energia: uma baseada em chaveamento de servidores e outra que utiliza troca de frequências. Ao longo deste trabalho, as duas políticas foram apresentadas como CDS e ADD, respectivamente.

Diferentemente de outros trabalhos da literatura, CDS e ADD atuam de maneira independente no aglomerado. Isso permite que o esquema de variação de frequências possa ser executado num período menor, aumentando os ganhos de potência. Além disso, foi desenvolvido uma abordagem para a configuração de frequências que tornou o problema extremamente rápido. Esse modelo se mostrou robusto, pois indicou-se que é possível contornar alguns casos onde a função de potência não apresenta as características assumidas.

Os algoritmos CDS e ADD apresentados possuem um baixo custo computacional, e portanto a otimização do aglomerado pode ser feita em tempo de execução. Além disso, foi mostrado que as técnicas desenvolvidas podem ser escaladas para uma quantidade grande de servidores. Tal característica é oportuna para ambientes de muitos servidores, como adotados em *datacenters* de empresas como o Google [5].

A proposta inclui um controle de saturação do *cluster*, que permitiu atingir níveis de QoS previsíveis na maioria dos casos analisados. Através desse sistema, foi possível obter uma economia de até 50% de energia quando comparado ao modo onde nenhuma política de energia está ativa. Foi comparado também o esquema ADD e o *governor* `ondemand`.

Nesse caso, obteve-se ganhos de energia em três dos quatro casos apresentados.

É importante enfatizar que a política ADD pode ser implementada de maneira complementar em vários trabalhos da literatura, como [29, 34]. Sendo assim, estes trabalhos passariam a usufruir de um controle de frequências mais eficiente sem que, para isso, precisem degradar a performance de suas políticas já consolidadas.

Além das políticas propostas, introduzimos o sistema ESSenCe. O objetivo desse sistema é criar uma camada de abstração para o estudo de técnicas de economia de energia em *clusters* de servidores. Dessa forma, pesquisadores tem a oportunidade de focar a sua atenção para a melhoria de técnicas de economia de energia, sem ter que se preocupar com detalhes de implementação. O ambiente desenvolvido possibilita a extensão e reuso, através do uso de boas práticas e modularização.

Um outro ponto forte do sistema ESSenCe é a sua capacidade de simulação. Através dela, é possível testar ideias rapidamente, sem precisar recorrer a horas de experimentos em um ambiente de testes físico. Graças a sua implementação em Python, a simulação pode ser feita em vários sistemas operacionais compatíveis com a linguagem e módulos utilizados.

7.2 Trabalhos futuros

Uma melhoria sugerida para o algoritmo CDS é o uso de heurísticas mais avançadas para o chaveamento de servidores. A premissa de que os servidores mais eficientes são os mais robustos pode perder oportunidades de economizar energia em alguns casos. Por isso, cabe uma análise mais profunda de quais servidores chavear, quando necessário. Outras heurísticas possíveis são: trocar servidores ativos por mais eficientes e consolidar carga de um ou mais servidores ativos em um servidor com mais capacidade de processamento.

Ao invés de usar heurísticas na política CDS, poderia ser utilizada uma abordagem de programação inteira mista. O problema da configuração dinâmica de servidores consiste em determinar o conjunto de máquinas mais eficientes, sem necessitar determinar quais frequências de operação serão utilizadas (esse é o papel da ADD). Sendo assim, resta saber se o problema de otimização resultante teria um menor custo computacional comparado com o problema mais geral que determina frequências e servidores simultaneamente.

A política ADD, por sua vez, talvez possa ser melhorada com a integração do `ondemand` no seu controle. O `ondemand` proveria um gerenciamento local de cada servidor, enquanto que o esquema ADD gerenciaria a frequência máxima de cada máquina. Através

do conhecimento de performance e potência de cada servidor, o algoritmo ADD continuaria a realizar seu trabalho de otimizar a alocação de desempenho do *cluster*. Porém, o `ondemand` teria o oportunidade de economizar energia em momentos onde a carga que o servidor recebe está abaixo do limite imposto, através do seu rápido período de atuação.

De uma forma geral, ambas as políticas propostas podem ser melhoradas através de um controle de saturação mais preciso. A política desenvolvida aqui mantém os parâmetros γ e σ fixos durante o experimento. Porém, é possível variá-los durante o experimento, através do *feedback* de QoS obtido. Esse tipo de controle requer uma análise que pode ser perseguida em investigações futuras.

A infraestrutura desenvolvida com o sistema ESSenCe pode ser expandida para trabalhos futuros. Uma possibilidade é a adição de mecanismos para virtualização de servidores, oferecendo outros recursos para economia de energia que podem ser integrados ao sistema. Outra possibilidade é o estudo do gerenciamento térmico dos servidores, de modo que se leve em conta a distribuição térmica das estações.

Apêndice A

Clock

Este Apêndice foi criado para esclarecer ao leitor interessado sobre o mecanismo de controle de execução de *threads*, criado para o módulo de simulação do sistema. A elaboração desse mecanismo se baseia em programação *multi-thread* e será assumido que o leitor já está familiarizado com os conceitos e primitivas mencionadas no decorrer do texto. De qualquer maneira, este Apêndice foi desenvolvido pois acredita-se que o mecanismo desenvolvido possa ser útil em outras aplicações de simulação, fora do escopo do ESSenCe.

O mecanismo desenvolvido foi nomeado Clock, como uma referência a sua capacidade de emular a passagem de tempo que ocorreria em um experimento real. Trata-se de uma classe que implementa dois métodos principais: *sleep()* e *tick()*, que já foram introduzidos na Seção 5.4.5. Para o mecanismo funcionar, basta que a função *sleep* convencional da *thread* seja substituída pelo método *sleep()* da classe Clock. Em uma outra linha de execução (e.g. *thread* do programa principal), chamadas a *tick()* são feitas quando se deseja avançar no tempo.

Este Apêndice será focado na implementação da classe Clock. Para tanto, serão utilizados trechos de código Python para que fiquem explícitos todos os artifícios empregados na implementação dessa classe. Ao mesmo tempo, não serão utilizados recursos particulares da linguagem, para que a legibilidade do código não seja prejudicada.

Clock assume que a *thread* que usa a sua função *sleep* tem uma certa estrutura. Um exemplo de *thread* compatível com o mecanismo pode ser vista na Figura A.1. De modo geral, a *thread* deve chamar periodicamente uma função indicada pela variável *sleep*. Quando executada em modo normal, a variável apontará para a função convencional de espera, enquanto que na simulação ela apontará para a função *sleep()* de Clock. A função *actuate()* representa alguma tarefa útil que a *thread* desempenha (e.g. obter amostras

```
1 class ClockThread(Thread):
2
3     def run(self):
4         self.stop = False
5         while not self.stop:
6             self.sleep(self.loopPeriod)
7             self.actuate()
```

Figura A.1: Estrutura de uma *thread* para funcionamento com o Clock

de potência). É assumido que *actuate()* não executará nenhum código que a bloqueie, como a disputa por um recurso que está de posse de outra *thread*, atualmente dormindo. Além disso, *actuate()* não deve alterar a variável *stop*, estando a mesma sobre controle do criador da *thread* (*parent*).

Clock mantém registros (*records*) das *threads* que chamam a função *sleep()*. Esses registros incluem um contador para o número de segundos restantes para acordar a *thread*, e um identificador da própria. Além disso, o registro guarda um semáforo que é utilizado para interromper a execução da *thread*, enquanto a função *tick()* não tiver sido chamada a quantidade de vezes necessária.

A.1 *Sleep*

O método *sleep* da classe Clock está representado na Figura A.2[1]. A variável *syncCondition* (linha 2) é uma variável de condição, usada pelo monitor[2] do sistema. Os métodos *acquire()* e *release()* de *syncCondition* são referentes a um *lock*, embutido na variável de condição. A ideia de *syncCondition* é sincronizar o fim da execução do código da *thread* (*actuate*) com a *thread* de controle. A utilidade desse artificio ficará mais clara quando for discutido o método *tick()*, na próxima Seção.

A linha 5 do código obtém a identificação da *thread* chamadora para possibilitar a busca do seu registro em seguida, na linha 9. Caso não seja encontrado, um registro novo será criado para identificá-la (linha 14). A linha 17 atribui o tempo especificado por

[1]As figuras deste capítulo não tem acentuação por que a ferramenta utilizada não permite o uso de caracteres deste tipo.

[2]O *monitor* a que se refere é a primitiva de sincronização.

```
1  def sleep(self, time):
2      self.syncCondition.acquire()
3
4      # obtem o ID da thread chamadora
5      thread = threading.currentThread()
6
7      # procura por um registro da thread
8      rec = None
9      for record in self.records:
10         if record.thread is thread:
11             rec = record
12
13     # cria um registro para a thread, caso necessario
14     if rec is None:
15         rec = self.Record(thread)
16         self.records.append(rec)
17     rec.counter = time
18
19     # notifica que uma thread esta presa
20     self.syncCondition.notifyAll()
21
22     self.syncCondition.release()
23
24     # bloqueia a thread
25     while rec.counter > 0:
26         rec.sem.acquire()
```

Figura A.2: Método *sleep()*

parâmetro para o contador de chamadas de *tick()*.

Em seguida, na linha 20, o método *notifyAll()* do monitor é chamado. Através desse método, identifica-se que uma *thread* acabou de chamar *sleep()*, e que em seguida ficará presa. Esse gatilho é importante, pois indica que o chamador da função *tick()* pode continuar sua execução saindo da função *wait()*. Em seguida, na linha número 26, a *thread* obtém repetidas vezes o seu semáforo, até que o contador interno do semáforo torne-se zero e a *thread* fique em espera.

A.2 *Tick*

O método *tick()* é chamado pela *thread* de controle, que geralmente é a linha de execução principal do programa. Através de chamadas a essa função indica-se para a classe Clock que se deseja avançar uma unidade no tempo (no ESSenCe, equivalente a um segundo).

O código desse método pode ser visto na Figura A.3. Como o tempo é absoluto para todas as *threads*, deve-se alterar todos os registros que Clock possui (linha 2). Ao fazer isso, *tick()* decrementa o seu contador (*rec.counter*) e libera seu semáforo (*rec.sem.release()*) através das linhas 5 e 6, respectivamente. Através dessas operações, é possível que a *thread* tenha se tornado livre e esteja rodando seu código interno (*actuate()*). Mesmo que a *thread* seja escalonada de forma a chamar a função *sleep()* novamente, não será possível entrar na região crítica dessa função, já que o semáforo de *syncCondition* se encontra fechado pela função *tick()*. Continuando, a função *tick()* verifica se a *thread* está livre pelo seu contador. Caso ele seja zero, significa que a *thread* em questão foi liberada, e então *tick()* espera até que a função *sleep()* seja chamada e ocorra uma notificação para que continue a processar os demais registros (linha 11).

Com esse arranjo, cada iteração de uma *ClockThread* é totalmente controlada pela função *tick()*. Observe também que é garantido que apenas uma *ClockThread* é executada por vez. Isso não é um requisito para que o mecanismo funcione. De fato, seria possível reimplementar *tick()* para que execute a função *wait()* ao final do *loop*, liberando outras *ClockThreads* para trabalhar em paralelo. No entanto, a implementação atual é útil para o sistema ESSenCe, onde deseja-se ter condições de replicar os experimentos. Caso as *threads* executassem em qualquer ordem, alterações nos resultados dos experimentos poderiam ocorrer.

```
1  def tick(self):
2      for rec in self.records:
3          self.syncCondition.acquire()
4          if rec.counter > 0:
5              rec.counter -= 1
6              rec.sem.release()
7
8              # caso a thread esteja livre:
9              if rec.counter == 0:
10                 # espere pelo notifyAll()
11                 self.syncCondition.wait()
12                 # obs: wait() chama release() automaticamente
13             else:
14                 self.syncCondition.release()
15         else:
16             self.syncCondition.release()
```

Figura A.3: Método *tick()*

Referências Bibliográficas

[1] ACPI. Advanced configuration and power interface. `http://www.acpi.info/`.

[2] AMD. *BIOS and Kernel Developer's Guide for AMD Athlon™64 and AMD Opteron™Processors*, Fevereiro 2006. `http://support.amd.com/us/Processor_TechDocs/26094.PDF`.

[3] Apache. The apache HTTP server project. `http://httpd.apache.org/ABOUT_APACHE.html`.

[4] Apache. Módulo mod_proxy_balancer, 2011. `http://httpd.apache.org/docs/2.2/mod/mod_proxy_balancer.html`.

[5] L.A. Barroso, J. Dean e U. Hölzle. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23:22–28, Março 2003.

[6] Y. Baryshnikov, E. Coffman, G. Pierre, D. Rubenstein, M. Squillante e T. Yimwadsana. Predictability of web-server traffic congestion. In *International Workshop on Web Content Caching and Distribution*, pp. 97–103, Washington, DC, EUA, Setembro 2005.

[7] D. Becker. `ether-wake`. `http://linux.die.net/man/8/ether-wake`. (Acessado em Maio de 2011).

[8] L. Bertini. *Energy Efficient Web Server Clusters with Stochastic QoS Control*. Tese de Doutorado, Universidade Federal Fluminense, Niterói, RJ, Brasil, 2009.

[9] L. Bertini, J.C.B. Leite e D. Mossé. Statistical QoS guarantee and energy-efficiency in web server clusters. In *19th Euromicro Conference on Real-Time Systems*, pp. 83–92, Pisa, Itália, Julho 2007.

[10] P. Bohrer, E.N. Elnozahy, T. Keller, M. Kistler, C. Lefurgy, C. McDowell e R. Rajamony. *Power aware computing*. Kluwer Academic Publishers, 2002.

[11] G.D. Bottari e J.C.B. Leite. Previsão de carga para configuração dinâmica de servidores web. *Revista Eletrônica de Iniciação Científica*, 2011.

[12] S. Brasen. Green computing: Using IT automation to achieve energy efficiency. `http://www.enterprisemanagement.com/research/`, Março 2008. White Paper.

[13] G. Dhiman, K. Mihic e T. Rosing. A system for online power prediction in virtualized environments using Gaussian mixture models. In *47th ACM Design Automation Conference*, pp. 807–812, 2010.

[14] DVS. Dynamic frequency scaling. `http://en.wikipedia.org/wiki/Dynamic_frequency_scaling`. (Acessado em Maio de 2011).

[15] E.N. Elnozahy, M. Kistler e R. Rajamony. Energy-efficient server clusters. In *Workshop on Power-Aware Computing Systems*, pp. 179–196, Cambridge, MA, EUA, Fevereiro 2002.

[16] Environmental Protection Agency EPA. Efeitos da mudança climática no meio ambiente. `http://www.epa.gov/climatechange/effects`. (Acessado em Junho de 2011).

[17] Facebook. Open Compute Project. `http://opencompute.org`. (Acessado em Junho de 2011).

[18] Gentoo. Gentoo Linux. `http://www.gentoo.org`. (Acessado em Maio de 2011).

[19] Google. Chromium OS. `http://www.chromium.org/chromium-os`. (Acessado em Junho de 2011).

[20] T. Ishihara e H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *International Symposium on Low Power Electronics and Design*, pp. 197–202, Monterey, CA, EUA, Agosto 1998.

[21] LAMP. Linux, Apache, MySQL e Perl/PHP/Python. `http://en.wikipedia.org/wiki/LAMP_(software_bundle)`. (Acessado em Maio de 2011).

[22] LBNL. The Internet traffic archive. `http://ita.ee.lbl.gov/index.html`. (Acessado em Abril de 2011).

[23] L. Li, T. RuiXiong, Y. Bo e G. ZhiGuo. A model of web server's performance-power relationship. In *International Conference on Communication Software and Networks*, pp. 260–264, Macau, China, Fevereiro 2009.

[24] S. Makridakis, S.C. Wheelwright e R.J. Hyndman. *Forecasting: Methods and Applications*. Kluwer Academic Publishers, 2002.

[25] D. Meisner, B.T. Gold e T.F. Wenisch. PowerNap: eliminating server idle power. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 205–216, Washington, EUA, Março 2009.

[26] D. Mosberger e T. Jin. httperf: A tool for measuring web server performance. In *Internet Server Performance Workshop*, pp. 59–67, Madison, WI, EUA, Junho 1998.

[27] V. Pallipadi e A. Starikovskiy. The ondemand governor: past, present and future. In *Linux Symposium*, pp. 223–238, Ottawa, Canadá, Julho 2006.

[28] V. Petrucci. A framework for supporting dynamic adaptation of power-aware web server clusters. Dissertação de Mestrado, Universidade Federal Fluminense, Niterói, RJ, Brasil, 2008.

[29] V. Petrucci, E.V. Carrera, O. Loques, J.C.B. Leite e D. Mossé. Optimized management of power and performance for virtualized heterogeneous server clusters. In *11th IEEE/ACM International Symposium on Cluster, Cloud and Grid*, Maio 2011.

[30] V. Petrucci, O. Loques e D. Mossé. Dynamic optimization of power and performance for virtualized server clusters. In *ACM Symposium on Applied Computing*, pp. 263–264, Nova Iorque, NY, EUA, Março 2010.

[31] E. Pinheiro, R. Bianchini, E.V. Carrera e T. Heath. Load balancing and unbalancing for power and performance in cluster-based systems. In *Workshop on Compilers and Operating Systems for Low Power*, volume 180, pp. 182–195, 2001.

[32] R.T. Rockafellar. *Convex analysis*. Princeton Mathematical Series. Princeton University Press, 1997.

[33] C. Rusu, A. Ferreira, C. Scordino e A. Watson. Energy-efficient real-time heterogeneous server clusters. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 418–428. San Jose, CA, EUA, Abril 2006.

[34] C. Santana. Previsão de carga em aglomerado de servidores web aplicada a economia de energia. Dissertação de Mestrado, Universidade Federal Fluminense, Niterói, RJ, Brasil, Março 2010.

[35] C. Santana, L. Bertini, J.C.B. Leite e D. Mossé. Applying forecasting to interval based DVS. In *Brazilian Workshop on Real-Time Systems*, pp. 13–20, Rio de Janeiro, RJ, Brasil, Maio 2008.

[36] C. Santana, J.C.B. Leite e D. Mossé. Load forecasting applied to soft real-time web clusters. In *ACM Symposium on Applied Computing*, pp. 346–350, Sierre, Suíça, Março 2010.

[37] D.C. Snowdon, E. Le Sueur, S.M. Petters e G. Heiser. Koala: A platform for OS-level power management. In *4th ACM European conference on Computer systems*, pp. 289–302, Nuremberg, Alemanha, Abril 2009.

[38] D.C. Snowdon, S. Ruocco e G. Heiser. Power management and dynamic voltage scaling: myths and facts. In *Workshop on Power Aware Real-time Computing*, volume 12, Jersey City, NJ, EUA, Setembro 2005.

[39] G. van Rossum. Python. `http://www.python.org`. (Acessado em Junho de 2011).

[40] World Wide Web Consortium (W3C). Códigos de status do HTTP. `http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html`. (Acessado em Junho de 2011).

[41] Wikipedia. Formato INI. `http://en.wikipedia.org/wiki/INI_file`. (Acessado em Junho de 2011).

[42] Wikipedia. Função linear por partes. `http://en.wikipedia.org/wiki/Piecewise_linear_function`. (Acessado em Junho de 2011).

[43] WoL. Wake-on-LAN. `http://en.wikipedia.org/wiki/Wake-on-LAN`. (Acessado em Maio de 2011).