**UNIVERSIDADE FEDERAL DE SANTA MARIA**
**CENTRO DE TECNOLOGIA**
**CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO**

# PROPOSTA DE UM ESCALONADOR ADAPTATIVO USANDO A LINGUAGEM DE PROGRAMAÇÃO LUA

**TRABALHO DE GRADUAÇÃO**

**Felipe Santos da Silva**

**Santa Maria, RS, Brasil**

**2014**

# PROPOSTA DE UM ESCALONADOR ADAPTATIVO USANDO A LINGUAGEM DE PROGRAMAÇÃO LUA

**Felipe Santos da Silva**

Trabalho de Graduação apresentado ao Curso de Bacharelado em Ciência da Computação da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para a obtenção do grau de

**Bacharel em Ciência da Computação**

**Orientadora: Profª. Dr. Marcia Pasin**

**Trabalho de Graduação N.373**
**Santa Maria, RS, Brasil**

**2014**

Santos da Silva, Felipe

Proposta de um Escalonador Adaptativo Usando a Linguagem de Programação Lua / por Felipe Santos da Silva. – 2014.
48 f.: il.; 30 cm.

Orientadora: Marcia Pasin
Monografia (Graduação) - Universidade Federal de Santa Maria, Centro de Tecnologia, Curso de Bacharelado em Ciência da Computação, RS, 2014.

1. Escalonamento de processos. 2. Adaptação. I. Pasin, Marcia. II. Título.


E-mail: felipes@inf.ufsm.br

**Universidade Federal de Santa Maria**
**Centro de Tecnologia**
**Curso de Bacharelado em Ciência da Computação**

A Comissão Examinadora, abaixo assinada,
aprova o Trabalho de Graduação

**PROPOSTA DE UM ESCALONADOR ADAPTATIVO USANDO A LINGUAGEM DE PROGRAMAÇÃO LUA**

elaborado por
**Felipe Santos da Silva**

como requisito parcial para obtenção do grau de
**Bacharel em Ciência da Computação**

**COMISSÃO EXAMINADORA:**

**Marcia Pasin, Dr.**
(Presidente/Orientadora)

**Iara Agustin, Dr.** (UFSM)

**Ana Trindade Winck, Dr.** (UFSM)

Santa Maria, 8 de Julho de 2014.

# AGRADECIMENTOS

Muito obrigado!

*Para chegares a saborear tudo,*
*Não queiras ter gosto em coisa alguma.*
*Para chegares a possuir tudo,*
*Não queiras possuir coisa alguma.*
*Para chegares a ser tudo,*
*Não querias ser coisa alguma.*
*Para chegares a saber tudo,*
*Não queiras saber coisa alguma.*
*Para chegares ao que não gostas,*
*Hás de ir por onde não gostas.*
*Para chegares ao que não sabes,*
*Hás de ir para onde não sabes.*
*Para vires ao que não possuis,*
*Hás de ir para onde não possuis.*
*Para chegares ao que não és,*
*Hás de ir por onde não és.*

*Quando reparas em alguma coisa,*
*Deixas de arrojar-te ao tudo.*
*Porque para vir de todo ao tudo,*
*Hás de negar-te de todo em tudo.*
*E quando vieres a tudo ter,*
*Hás de tê-lo sem nada querer.*
*Porque se queres ter alguma coisa em tudo,*
*Não tens puramente em Deus teu tesouro.*
— São João da Cruz

# RESUMO

Trabalho de Graduação
Curso de Bacharelado em Ciência da Computação
Universidade Federal de Santa Maria

**PROPOSTA DE UM ESCALONADOR ADAPTATIVO USANDO A LINGUAGEM DE PROGRAMAÇÃO LUA**

AUTOR: FELIPE SANTOS DA SILVA
ORIENTADORA: MARCIA PASIN
Local da Defesa e Data: Santa Maria, 8 de Julho de 2014.

Escalonamento de processos é uma parte muito importante de um sistema computacional e define o desempenho do mesmo. Existem diversas possibilidades de implementações (por exemplo, FIFO, SJF, por tempo, etc.) e, devido à natureza heterogênea de muitas aplicações, a escolha de uma política de escalonamento nem sempre é uma tarefa trivial. De fato, a escolha da política mais adequada está relacionada com o cenário e demanda atual. Se o cenário e a demanda mudam, uma adaptação pode ser necessária. Neste sentido, este trabalho representa um passo a adiante na adaptação para o escalonamento de processos em sistemas computacionais.

Mais precisamente, neste trabalho é realizado o estudo sobre as políticas de escalonamento simples e tradicionais. Em seguida, as políticas tradicionais são combinadas em um método adaptativo, que modifica o seu comportamento, conforme a carga de trabalho dos processos já finalizados. A modificação do comportamento do escalonador ocorre através do uso de heurística. Troca-se ao acaso a política. Se a nova política apresenta melhores resultados, a execução segue com a nova política até que ocorra uma nova avaliação.

A avaliação das políticas foi realizada com o apoio de simulações, considerando diferentes cenários e demandas. Para estudo e avaliação das políticas, foi usado um ambiente estável, que considera a rede sem atrasos e falhas. Através destas estimativas, foram tomadas as decisões para o desenvolvimento do escalonador adaptativo. Neste estudo, foi considerado o uso de apenas um servidor. Posteriormente, poderá se seguir o trabalho com problemas envolvendo mais servidores e outras políticas mais robustas para adaptação.

Para conduzir os experimentos, foi usada a linguagem de programação Lua, devido ao suporte adequado para o desenvolvimento de sistemas distribuídos. Com a realização deste trabalho, destaca-se que a linguagem se mostra muito simples e poderosa para a prototipação de sistemas computacionais.

**Palavras-chave:** Escalonamento de processos. Adaptação.

# ABSTRACT

Undergraduate Final Work
Undergraduate Program in Computer Science
Federal University of Santa Maria

**TOWARDS AN ADAPTIVE SCHEDULER USING LUA PROGRAMMING LANGUAGE AS SUPPORT**

AUTHOR: FELIPE SANTOS DA SILVA
ADVISOR: MARCIA PASIN
Defense Place and Date: Santa Maria, July 8st, 2014.

Process scheduling is a very important part of a computer system and defines the performance of the same. There are several possibilities of polices to deal with it (e.g., FIFO, SJF, on time, etc..). And, in addition, due to the heterogeneous nature of many applications, the choice of a scheduling policy is not always a trivial task. Indeed, choosing the most appropriate policy is related to the scenario and current demand. If the scenario and demand change, an adaptation may be necessary. Thus, this work represents a step forward in adapting to the process scheduling in computer systems.

More precisely, this work is about the application of scheduling policies and adaptation in face of changes. Traditional policies may be combined into an adaptive method, which changes its behavior as the workload of the processes changes. The adaptation of the scheduler behavior occurs through the use of heuristics. It trades at a random policy. If the new policy shows better result than the former, execution continues with the new policy until a new evaluation occurs.

The policy evaluation was performed with the support of simulations, considering different scenarios and demands. To allow the study and the evaluate of the policies, we used a stable environment, which considers the network without delays and failures. Through our experiments, the decisions for the development of adaptive scheduler may be taken. This study considered the use of only one server. But the adaptation method can be applied to decentralized systems as well.

To conduct the experiments, we used the Lua programming language due to the suitable support in the development of prototype in distributed systems. With this work, we emphasize that the language proves to be very simple and powerful for prototyping in computing systems.

**Keywords:** Adaptive Control, Scheduling, System Process.

# LISTA DE FIGURAS

# LISTA DE APÊNDICES

# LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| CPU | *Central Processing Unit* |
| FCFS | *First Come First Serve* |
| FIFO | *First In First Out* |
| SJF | *Shortest Job First* |

# SUMÁRIO

# 1 INTRODUÇÃO

## 1.1 Qualificação e fundamentação teórica do problema

Aplicações distribuídas estão sujeitas à demanda heterogênea, com picos de carga e ociosidade. Este cenário conduz à necessidade de gerenciar essas ocorrências, que é uma atividade complexa. Soluções de infraestrutura computacional atuais não são projetadas para se adaptarem automaticamente a condições de dinamismo e complexidade do sistema. Tipicamente, soluções existentes são dependentes de operação manual, deixando o sistema sujeito à indisponibilidade e tempo inadequado de resposta. Há carência de infraestruturas computacionais auto-adaptativas, capazes de suportar mudanças de demanda e diferentes adversidades simultaneamente.

Mais especificamente, aplicações distribuídas atuais precisam de soluções mais adaptáveis e que permitam facilidades para a evolução de política de escalonamento. Soluções adaptativas para escalonamento com foco em adaptação são mais adequadas às demandas das aplicações distribuídas atuais e consideram requisitos como sobrecarga de servidores (ZHANG, 2010), informação contida na requisição de cliente (LI et al, 2006), como tempo de resposta, e ou então prioridade de processos clientes (EWING, 2009).

## 1.2 Objetivos e metodologia

Este trabalho propõe a construção de um escalonador de processos que implementa diferentes políticas de escalonamento, buscando melhorar o tempo de resposta nos clientes. As políticas de escalonamento em questão incluem FIFO, SJF e *sorteio*. Basicamente, a modificação do comportamento do escalonador ocorre através do uso de heurística. Troca-se ao acaso a política. Se a nova política apresenta melhores resultados, a execução segue com a nova política até que ocorra uma nova avaliação.

Para avaliar o escalonador, foi construído um protótipo deste sistema usando a linguagem de progamação Lua. Foi escolhida esta linguagem por suas características desejáveis à prototipação.

Para alcançar este objetivo, este trabalho seguiu a seguinte metodologia:

- Pesquisa bibliográfica.

- Levantamento de requisitos do sistema.
- Projeto do escalonador adaptativo.
- Implementação de um protótipo do escalonador, usando a linguagem de programação Lua.
- Construção de cenários de testes.
- Realização de uma bateria de testes usando os cenários previamente definidos.

## 1.3 Estrutura do texto

O capítulo 2 apresenta a revisão da literatura, com duas partes: conceituação e trabalhos relacionados. O capítulo 3 apresenta o projeto do escalonador adaptativo, descrevendo a arquitetura geral e a adaptação de políticas. O capítulo 4 apresenta a avaliação experimental do escalonador adaptativo e indica como a adapação pode melhorar o desempenho de um sistema computacional. Finalmente, o capítulo 5 apresenta as conclusões e considerações finais.

# 2 REVISÃO DA LITERATURA

## 2.1 Escalonamento de processos

O problema de escalonamento de processos em sistemas computacionais distribuídos não é novidade, e já foi investigado e discutido na literatura (TANENBAUM, 2007). Entretanto, ainda não existe um consenso sobre o uso dessas políticas. A escolha da política adequada depende de características da aplicação, da demanda e do sistema computacional.

Neste sentido, este trabalho investiga o uso de políticas de escalonamento de processos em cenários heterogêneos, buscando avaliá-las quanto à otimização do tempo de resposta para os clientes. Como é difícil afirmar que exista uma única política que ofereça resultados excelentes em todas as possíveis combinações de *hardware* e *software*, um mecanismo adaptativo é uma idea interessante, na medida que o processo de escalonamento se adapta conforme a necessidade do sistema computacional. Estes resultados podem colaborar na construção de um escalonador adaptativo.

Através do conhecimento das necessidades das aplicações a serem executadas, de suas cargas e do comportamento de diferentes políticas de escalonamento de acordo com dado cenário, pode ser realizada a escolha da melhor política de escalonamento naquele momento ou, pelo menos, pode ser usada solução que apresente com desempenho próximo à melhor solução.

Como restrição, neste trabalho será assumido que o sistema nunca falha e não tem atrasos de resposta, para fins de se estimar ao melhor política dado determinado cenário. É esperado que uma política com um desempenho bom na situação ideal também a seja na situação real, políticas que não se mostrem neste estado estarão fora das escolhas para o escalonador adaptativo.

No texto que segue, é definida a nomenclatura usada ao longo deste texto.

### 2.1.1 Conceitos

Um **processo** é uma abstração de um programa em execução, e deve ser executado por uma CPU. Na condição de multiprogramação, onde mais de um programa roda em uma única CPU, considera-se que cada processo roda em sua CPU virtual. Quando o suporte de programação paralela ou distribuída está disponível, realmente cada processo terá a sua própria CPU.

Uma **tarefa** é uma quantidade de processamento de dados ou transferência de dados por uma rede ou qualquer outra demanda que um processo pode pedir a outro processo.

Um processo, dependendo do comportamento que ele executa, pode ser classificado em cliente ou servidor. Um processo **cliente** possui uma ou mais tarefas, ou serviços, a serem executadas por um outro processo chamado **servidor**. Esta dinâmica recebe o nome de arquitetura cliente-servidor.

Quando processos clientes estão disputando os serviços de um processo servidor que não os pode atender simultaneamente, é necessário realizar uma escolha para o escalonamento das tarefas solicitadas pelos processos clientes. O processo servidor deve escolher de alguma maneira qual entre todas as tarefas recebidas será executada primeiro. Essa escolha é feita atravéz de **políticas**, que serão tratadas com mais detalhes nas sessões a seguir. Tipicamente essa escolha é realizada pelo módulo ou serviço de escalonamento de processos.

Por fim, clientes e servidores estabelecem comunicação através de troca de mensagens. Neste contexto, cada processo pode enviar mensagens para os demais em um processo de sincronização e colaboração mútua. Para receber estas mensagens, cada processo possui uma caixa de correio de onde ele pode ler as suas mensagens na mesma ordem em que chegaram. Esta caixa de correio funciona como um *buffer*. Quando um processo espera receber uma mensagem (solicitação de uma tarefa/serviço ou resposta a uma tarefa/serviço), porém a sua caixa de correio está vazia, o mesmo fica bloqueado, aguardando a chegada de uma mensagem. Deste modo, é possível ter um processo responsável pela leitura e gravação de dados em determinada área de memória ou arquivo, que se comunica com os demais através de troca de mensagem. Assim, é garantido que nunca haverão dois ou mais processos gravando ou lendo dados de um mesmo lugar simultaneamente.

### 2.1.2 Políticas de escalonamento de processos

As políticas de escalonamento de processos são responsáveis por definir o que deve ser feito para que ocorra o escalonamento. A literatura descreve diferentes políticas para escalonamento de processos. No escopo deste trabalho, as políticas estudadas incluem: SJF, FIFO e algoritmo da loteria (com agendamento por *sorteio*). Na sequencia, estas políticas serão brevemente descritas.

#### 2.1.2.1 A tarefa mais curta primeiro

Usando a política *a tarefa mais curto primeiro*, ou SJF (*Shortest Job First*), o escalonador executa processos de acordo com o tamanho de cada processo. O processo mais curto tem maior prioridade sobre os processos mais longos.

Esta política é especialmente apropriada para tarefas em lote, onde o tempo de execução é conhecido previamente. Segundo (TANENBAUM, 2007), supondo quatro tarefas, $p_1$, $p_2$, $p_3$ e $p_4$, com tempos de execução $t_1$, $t_2$, $t_3$, e $t_4$, respectivamente, tem-se que o tempo de retorno médio é ($4t_1$ + $3t_2$ + $2t_3$ + $t_4$)/4. Esta política é, portanto, uma interessante estratégia para sistemas computacionais com demanda de tempo de resposta eficiente.

Essa política tem a desvantagem de que a informação sobre o tempo de duração de cada tarefa precisa estar disponível previamente. Entretanto, em muitas aplicações, como a transferência de arquivos entre máquinas distintas, esses valores podem ser facilmente estimados.

#### 2.1.2.2 FIFO

A política FIFO, *First In First Out*, aplica a regra o primeiro que entra na fila é primeiro que sai. É um das mais simples políticas a serem implementadas, e não requer informação sobre o tempo de duração de cada tarefa disponível previamente.

Basicamente, o algoritmo coloca as tarefas a serem executadas em uma fila, obedecendo a ordem em que as requisições foram entregues no servidor. A primeira tarefa que entra na fila é primeira tarefa que será executada.

Este é um algoritmo que a cada execução a ordem da execução das tarefas pode mudar substancialmente, a medida a ordem de execução das tarefas é imposta pela ordem de entrega das mensagens de requisição dos clientes no servidor (isto é, não importa a ordem de envio). Esta ordem de execução impacta, também, no tempo de resposta para os clientes.

#### 2.1.2.3 Algoritmo da loteria (Agendamento por *sorteio*)

Este algoritmo escolhe de forma aleatória qual o próximo processo a ser executado distribuindo bilhetes de loteria aos processos. Quando o escalonador está disponível para uso, um novo bilhete é sorteado. Segundo (TANENBAUM, 2007), este algoritmo pode priorizar um processo $p_i$ alocando mais bilhetes para este processo.

### 2.1.3 Métricas para avaliação das políticas

Métricas são usadas para quantificar o quanto uma política é melhor que outra. Entre as diferentes métricas relatadas na literatura destaca-se (TANENBAUM, 2007):

1. Imparcialidade assegura que cada processo receba uma parte justa na alocação da CPU.
2. Eficiência mantém a CPU ocupada 100% do tempo.
3. Tempo de resposta procura minimiza o tempo de resposta para usuários interativos.
4. *Turnaround* minimiza o tempo que os usuários de lote devem esperar pela saída.
5. *Throughput* busca maximizar o número de tarefas processadas por hora.

A métrica usada neste trabalho é o tempo de resposta. No escopo deste trabalho, as políticas são avaliadas periodicamente, e o escalonador seleciona a política com o melhor tempo de resposta para dado cenário.

## 2.2 Trabalhos relacionados

A adaptação em sistemas computacionais não é assunto novo na literatura. Este assunto tem sido estudado no contexto de arquiteturas centralizadas e distribuídas, visando a implantação do conceito de sistemas autonômicos (KEPHART, 2003).

### 2.2.1 Adaptação de escalonamento em sistemas CORBA

Cervieri (CERVIER, 2002) propõe o uso de escalonamento adaptativo para aplicações de tempo real em CORBA. O período das tarefas é controlado, variando dentro de uma faixa pré-estabelecida. O principal objetivo é reduzir o atraso médio das tarefas em execução.

### 2.2.2 Adaptação por tempos de resposta em um *cluster*

Em Ewing & Menascé (EWING, 2009), o escalonamento é tratado a nível de balanceamento de carga para múltiplos servidores em um sistema de leilão usando páginas de Internet. O objetivo é priorizar os grupos de usuários que estão executando as maiores quantidades de lances. O escalonamento é realizado através de classificação de pedidos, alocando pedidos prioritários a um *cluster* com maior disponibilidade de processamento.

Como característica comum, este trabalho também utiliza o tempo de resposta nos pedidos dos usuários como métrica de adaptação no sistema computacional. Em contraste, o escalonador adaptativo aqui proposto atua sobre uma infraestrutura computacional de apenas um servidor e não foca na escolha de máquina apropriada, mas na escolha política apropriada.

### 2.2.3 Adaptação por tempo e abandonos de pedidos

Badonnel & Burgess (BADONELL, 2008) utilizam métricas de tempo e abandonos de pedidos para realizar calibrações em um sistema computacional distribuído. São realizadas medições de tempo de resposta dos pedidos dos clientes e geradas estatísticas com a quantidade de pedidos e o tempo médio em comparação com os métodos *pull-based* e *push-based*.

Em contraste com este escalonador adaptativo aqui proposto, no trabalho de (BADONELL, 2008), o enfoque é na comunicação entre clientes e o módulo de escalonamento. No escalonador, e em cada servidor existe, uma fila finita de pedidos que podem sofrer abandonos quando a fila estiver cheia. Uma estatística sobre o número de pedidos abandonados é usada como base para a decisão entre os dois métodos. É utilizada a política FCFS (First Come First Serve) ou FIFO para a distribuição das tarefas.

### 2.2.4 Adaptação através de computação autonômica e uso de CPU

Bouchenak *et al.* (BOUCHENAK, 2006) utiliza sensores para verificar a utilização de CPU com finalidade de ter um sistema formado por componentes para gerenciamento uniforme. Esse gerenciamento é capaz de implantar aplicativos distribuídos de obter uma reconfiguração de forma autônoma caso tenha necessidade.

A adaptação é processada por um módulo centralizado que coleta informações sobre CPU em componentes locais. Se a configuração não está adequada, o que é verificado através da definição de um *threshould*, o sistema dispara um mecanismo que adapta a configuração do sistema.

Em contraste, neste trabalho adaptação é aplicada ao escalonamento de processos e não no balanceamento de carga ou na configuração do sistema. O enfoque adotado aqui é nas políticas de escalonamento.

## 2.3 Programação concorrente em Lua

A linguagem Lua (http://lua.org ou http://www.lua.org/portugues.html) foi projetada e desenvolvida a partir de 1993 por Roberto Ierusalimschy, Luiz Henrique de Figueiredo e Waldemar Celes na PUC-Rio (Pontifícia Universidade Católica do Rio de Janeiro). Atualmente, Lua é desenvolvida em laboratório próprio, o LabLua.

A linguagem combina sintaxe simples para programação procedural com construções para descrição de dados baseadas em tabelas associativas e semântica extensível. Lua possui as seguintes características que merecem destaque:

- é uma linguagem tipada dinamicamente. As variáveis não possuem tipos, apenas os dados referenciados por elas possuem. São eles *nil, boolean, number, string, function, userdata, thread*, e *table*.
- é interpretada a partir de *bytecodes* para uma máquina virtual baseada em registradores, e
- tem gerenciamento automático de memória com coleta de lixo incremental.

Essas características fazem de Lua uma linguagem interessante para configuração, automação (*scripting*) e prototipagem rápida de programas. Essas características pesaram na escolha desta linguagem para a construção do protótipo do escalonador adaptativo. Outra característica importante é que a linguagem é distribuída sob uma licença muito liberal (a conhecida licença MIT) e é a implementação do ambiente de programação é *software* livre de código aberto.

Em seu manual de referência são explicadas as suas principais características. Primeiramente, apesar de ser uma linguagem de extensão que necessita um programa hospedeiro para executá-lo, a distribuição oferece o programa lua, um interpretador de linhas de comando completo.

Na sequencia deste texto é exposta a sintaxe básica da linguagem.

### 2.3.1 Atribuição de valores às variáveis

Para atribuir o valor numérico 3 a uma variável chamada $a$ e um valor lógico $true$ a uma variável chamada $b$, é usado o seguinte código:

```
a = 3
b = true
```

O seguinte fragmento de código troca os valores de $a$ e $b$, sem a necessidade de criar uma variável auxiliar.

```
a, b = b, a
```

### 2.3.2 Uso de tabelas

Para se construir uma tabela, um exemplo de comando simples é:

```
t = {5, 3, 3, 4, 1}
```

Onde é gerada uma tabela referenciada pela variável $t$, na qual os valores de $t[1]$ é 5, de $t[2]$ é 3 e assim por diante. Para alterar o valor de $t[2]$ para 7, deve ser realizada uma simples atribuição:

```
t[2] = 7
```

Também é possível atribuir novos valores a outras chaves de $t$. Não sendo necessário que o valor seja numérico, nem que todos os índices menores que o índice pretendido sejam previamente preenchidos.

```
t[8] = "palavra"
```

Do mesmo modo, os índices não precisam ser todos numéricos. É permitida a inserção em uma tabela $t$ usando uma palavra-chave "chave"como índice:

```
t["chave"] = false
```

A linguagem oferece uma otimização para facilitar a legibilidade. A mesma atribuição pode ser escrita da seguinte forma:

```
t.chave = false
```

Uma tabela $t2$, pode ser construída da seguinte forma:

```
t2 = {5, false, [10] = 12.5, "terceiro", id=12, true}
```

O que tem o mesmo resultado de:

```
t2 = {}
t2[1] = 5
t2[2] = false
t2[3] = "terceiro"
t2[4] = true
t2[10] = 12.5
t2.id = 12
```

### 2.3.3 Estruturas de controle

Existem quatro estruturas de controle em Lua, são elas: $if$, $while$, $repeat$ e $for$.

O comando $if$ possui a forma:

```
if exp then bloco {elseif exp then bloco} [else bloco] end
```

Sendo a parte com $elseif$ opcional. É permitido o uso de mais de um $elseif$ em sequência e apenas um $else$, que é opcional.

Os comandos $while$ e $repeat$ possuem as respectivas formas:

```
while exp do bloco end
repeat bloco until exp
```

O comando $for$ tem duas formas, a primeira:

```
for v = a, b, c do bloco end
```

Onde $a$ é o valor inicial de $v$, $b$ é o valor final e $c$ o incremento.

A segunda forma do comando for é a seguinte:

```
for var_1, ..., var_n in listaexps do bloco end
```

O exemplo a seguir mostra um comando $for$ que passará por todos os índices e valores da tabela $a$, com o uso da função $pairs$. Em cada passo do $for$, $i$ itera sobre os índices da tabela $a$ e $j$ itera sobre o valor correspondente de ao índice $i$ na tabela.

```
a = {5, 4, 3, id = 12}
for i, j in pairs(a) do
   print(i, j)
end
```

O próximo exemplo mostra o uso da função $ipairs()$, a qual faz o $for$ iterar apenas nos índices numéricos de $a$

```
a = {5, 4, 3, id = 12}
for i, j in ipairs(a) do
   print(i, j)
end
```

Por último, a declaração de funções possui a forma:

```
function funcao(arg)
   bloco
end
```

Funções são valores normais da linguagem. No caso acima, é criada a variável $funcao$ que contém esta função como valor. Esta variável pode ser usada como argumento para outras funções.

### 2.3.4 Programação cliente-servidor em linguagem Lua

Lua possui suporte para a programação concorrente usando clientes e servidores. Para fazer uso da programação paralela e distribuída em Lua, neste trabalho, foi usada a biblioteca $ConcurrentLua$. Esta biblioteca, cujo manual pode ser encontrado em `github.com/lefcha/concurrentlua/blob/master/doc/manual.html`, implementa programação concorrente e distribuída baseada no modelo de troca de mensagens. Cada processo tem uma caixa de correio correspondente, onde as mensagens podem ser lidas a qualquer momento e na ordem em que foram recebidas.

Um sub-grupo das funções disponíveis nesta biblioteca foram usadas neste trabalho. Segue a descrição destas funções. Para a criação de um novo processo, existe a função $spawn$, que recebe como argumento uma função que será executada no processo criado por essa função. Ela tem como retorno o número identificador deste processo, que é usado para comunicação com este processo. Também existe a função $self()$, que retorna para um processo o seu próprio número identificador.

As funções $send$ e $receive$ são usadas para a troca de mensagens. A função $send(d, m)$ serve para enviar a mensagem $m$ para o processo com o identificador $d$. Retorna o valor $true$ caso a operação tenha obtido sucesso.

Com a função $receive()$, o processo recupera a mensagem mais antiga de sua caixa de correio. Caso a caixa do correio esteja vazia, o processo permanece bloqueado até que chegue uma mensagem. Também pode ser chamada com um argumento numérico, o qual limita o tempo de espera para este valor em milissegundos. Com a função $loop$ são iniciados os processos.

Um pequeno exemplo de código é apresentado na sequencia. Neste exemplo, uma aplicação do tipo cliente-servidor, onde o cliente $f1$ envia mensagens $n$ para o $servidor$ imprimir na tela. O cliente possui o seguinte código:

```
require "concurrent"
function f1(n, id_servidor)
   for i = 1, n do
      concurrent.send(id_servidor, i)
   end
end
```

O servidor possui o seguinte código:

```
function servidor()
   while true do
      msg = concurrent.receive()
      print(msg)
   end
end
id_servidor = concurrent.spawn(servidor)
concurrent.spawn(f1, 20, id_servidor)
concurrent.loop()
```

Note que o cliente conhece o $id$ do servidor e que o servidor conhece o $id$ (f1) do cliente.

# 3 PROJETO DO ESCALONADOR ADAPTATIVO

Este capítulo descreve detalhes do projeto do escalonador adaptativo. Inicialmente é descrita a arquitetura do escalonador de processos e é realizado o detalhamento do algoritmos relacionados. Por fim, é descrito, em especial, o processo de adaptação baseado em heurística.

## 3.1 Visa geral da Arquitetura

A arquitetura do serviço de escalonamento segue um esquema centralizado. A arquitetura do serviço de escalonamento é composta por $n$ clientes e um servidor. Basicamente, os clientes solicitam a execução de tarefas ao servidor, através do envio de requisições. O servidor usa uma fila para gerenciar as requisições. Deste modo, o serviço do servidor foi fracionado em executor de tarefa propriamente dita e gerenciador de fila de requisições.

Mais precisamente, o esquema obedece os seguintes passos, de acordo com o esquema apresentado na Figura 3.1:

- Cada processo cliente executa em um *loop* onde tenta enviar as suas requisições de trabalho para o servidor (1).
- O servidor, através do gerenciador de fila, recebe essas requisições dos clientes e as coloca na fila (2).
- O executor de tarefas, quando está ocioso, fica constantemente pedindo uma nova tarefa para o gerenciador da fila (3).
- Quando a fila não está vazia, o gerenciador de fila escalona os processos dispostos na fila e escolhe qual deve ser enviado para o executor de tarefas, caso a fila esteja vazia, retorna uma mensagem avisando que não há tarefas na fila (4).
- O executor de tarefas, ao terminar a tarefa, retorna ao gerenciador e ao cliente que o trabalho foi concluído (5).

Para implementar a fila de tarefas no servidor, foi usado o recurso de caixa de mensagens oferecido por $Lua$.

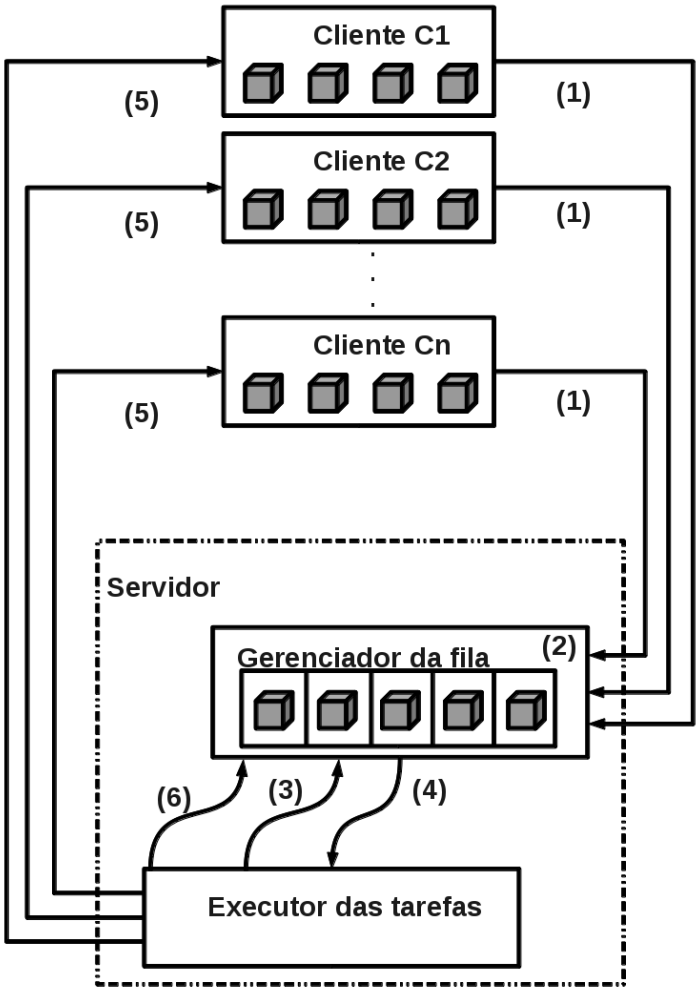


Figura 3.1: Arquitetura cliente-servidor para o escalonamento de tarefas.

## 3.2 Descrição dos algoritmos

Os clientes e o servidores executam os algoritmos descritos como segue. Os clientes solicitam tarefas ao servidor, executando o código esboçado pelo Algoritmo 1. O servidor, basicamente, retira as tarefas de uma fila e executa as tarefas. O servidor também armazena informação sobre o tempo de resposta coletado e enviado pelos clientes, qeu será usado para a adaptação. O código é esboçado pelo Algoritmo 2.

**Algoritmo 1:** Processamento executado pelos clientes

```
1:  function roda_cliente(lista_tarefas, servidor_id)
2:  for i ⇐ 1 until tamanho (lista as tarefas) do
3:      enviou ⇐ false
4:      while not enviou do
5:          enviou ⇐ send(servidor_id, {
6:          from ⇐ self(),
7:          tarefa ⇐ lista_tarefas[i]})
8:      end while
9:      msg ⇐ receive()
10: end for
```

O Algoritmo 1 apresenta o código executado pelos clientes. Na linha 1, ocorre a declaração da função roda_cliente, que deve ser usada na função *concurrent.spawn()* para iniciar a execução de um processo cliente. A função roda_cliente() recebe dois argumentos. O primeiro é uma lista simples de tarefas a serem enviadas ao servidor, o segundo é o identificador deste servidor para ocorrer a comunicação com o mesmo. Na linha 2, começa um laço de repetição onde a variável $i$ tomará valores de 1 até o tamanho da lista de tarefas recebida, que é um valor configurável. Na linha 3, é inicializada uma *flag* chamada $enviou$ com o valor $false$. Na linha 4, é iniciado um laço de repetição com a variável $enviou$ como condição de parada.

Nas linhas 5-7, ocorre a chamada da função concurrent.send(). A função tem como valor de retorno um *booleano*, confirmando o sucesso no envio da mensagem ou insucesso. No primeiro argumento existe um identificador do servidor, que é o destino da mensagem. O segundo argumento, entre chaves, possibilita a construção de uma tabela com o campo *from* preenchido pelo identificador do processo cliente, para o retorno da mensagem, a tarefa a ser executada pelo servidor no campo *tarefa*.

Na linha 9, a variável $msg$ recebe a confirmação da execução da tarefa pelo servidor. Enquanto o processo cliente não receber esta confirmação ele fica bloqueado aguardando.

O Algoritmo 2 apresenta o algoritmo executado pelo servidor. Inicialmente, a fila de

**Algoritmo 2:** Procedimento executado pelo servidor

```
 1: function servidor()
 2: fila ⇐ {}
 3: executor_id=spawn(executor, self())
 4: while true do
 5:    msg ⇐ receive()
 6:    if msg.from=executor then
 7:       if #fila>0 then
 8:          escolhido ⇐ escolhe(fila)
 9:          send(executor_id,escolhido)
10:       else
11:          send(executor_id, {cliente=nil, tarefa=0})
12:       end if
13:    else
14:       insert(fila,{cliente=msg.from, tarefa=msg.tarefa})
15:    end if
16: end while
```

processos está vazia (linha 2) e o processo encarregado de realizar as tarefas é iniciado (linha 3). Continuamente, o servidor recebe novas mensagens. Se a nova mensagem for originária do processo executor (linha 6) e a fila de processos não estiver vazia (linha 7) uma tarefa da fila será escolhida pelo escalonador (linha 8), de acordo com uma política específica, e enviada ao executor (linha 9). Se a fila de tarefas está vazia, é enviado ao executor uma mensagem indicando que não existe tarefa a ser executada no momento (linha 11). Caso a mensagem recebida não seja originária do executor, é de um cliente, assim a tarefa pedida por este é inserida na fila.

Quanto ao programa executor, ele apenas permanece constantemente solicitando tarefas ao servidor. Quando uma tarefa é recebida por ele, ela é realizada e a seguir uma mensagem de tarefa pronta é enviada ao respectivo cliente. O executor volta então a pedir constantemente tarefas para o servidor até receber uma nova tarefa.

Nesta arquitetura, o servidor implementa diferentes políticas de escalonamento (FIFO, SJF ou *sorteio*; veja a linha 8 no código do servidor (Algoritmo 2)), mas executa somente uma política por vez. A escolha da política é realizada com base nos valores de tempo recebidos de cada cliente.

A modificação do comportamento do escalonador ocorre através do uso de heurística. Troca-se ao acaso a política. Se a nova política apresenta melhores resultados, a execução segue com a nova política até que ocorra uma nova avaliação dentro do intervalo de tempo especificado.

## 3.3 Adaptação

Esta sessão detalha o funcionamento do escalonador adaptativo. Basicamente, o algoritmo de adaptação é dividido em duas partes, seguindo o que foi apresentado no item 3.2: procedimento do cliente, executado por todos os clientes, e procedimento do servidor, executado pelo servidor.

O **procedimento do cliente** é executado a cada intervalo de tempo $T_c$, e segue a seguinte ordem de tarefas:

1. Clientes informam ao servidor os seus valores de tempo de resposta

O **procedimento do servidor** é executado a cada intervalo de tempo $T_s$, e segue a seguinte ordem de tarefas, exibidas no fluxograma da Figura 3.2:

1. Recolher as informações de tempo de resposta dos clientes;
2. Sortear nova política;
3. Executar nova política;
4. Recolher as novas informações de tempo de resposta dos clientes;
5. Se os novos tempos de resposta são melhores que os anteriores, então seguir aplicando a nova política; caso contrário, seguir voltar ao passo 2.

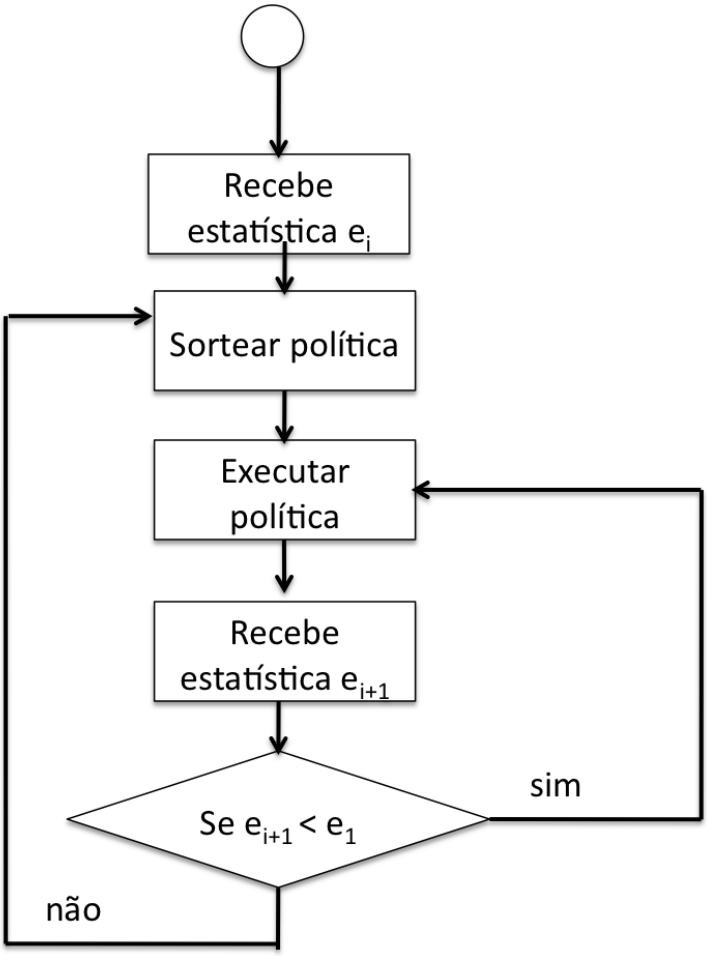


Figura 3.2: Fluxograma para o servidor para a escolha de políticas.

# 4 EXPERIMENTAÇÃO

## 4.1 Ambiente de implementação e experimentação

Os experimentos foram realizados em uma máquina usando o sistema operacional Debian 7.2 (wheezy) 32-bit, com o Kernel Linux 3.2.0-4-686-pae, no ambiente de trabalho GNOME 3.4.2. A máquina possui 3,7GiB de memória RAM e quatro processadores Intel®Core™i3 CPU 540 @ 3.07GHz. A versão da linguagem Lua usada foi a 5.1, com o sistema $LuaRocks$ para a instalação da biblioteca $concurrentLua$.

No total, foram escritas $420$ linhas de código, contando com a implementação do sistema, cenários de experimentação e programas auxiliares para avaliar os resultados dos $logs$ gerados pelos experimentos.

## 4.2 Experimentação e definição de cenários de testes para as políticas

Para a avaliação experimental, foram definidos quatro cenários de testes, com diferentes demandas. Os cenários e respectivos resultados são descritos a seguir.

Para medir os tempos de resposta do servidor para os clientes, e possibilitar avaliar métricas buscando adaptação, foi implementado um relógio que incrementa o seu tempo de acordo com a tarefa que está sendo executada pelo servidor. Foi usada a medida de unidades de medida de avaliação.

Para prover mais precisão nestas medições, foram realizadas três execuções de cada cenário e o valor apresentado é a média dos valores encontrados em cada execução.

### 4.2.1 Cenário 1: tarefas homogêneas e de curta duração

No Cenário 1, o servidor recebe requisições de tarefas de três clientes idênticos, cada um solicita ao servidor a execução de $100$ tarefas. As tarefas são homogêneas e de curta duração. Como as tarefas são de curta duração, e a execução de uma nova tarefa é realizada justamente após a finalização da tarefa anterior, a demanda é alta. Nos experimentos do Cenário 1, a duração de cada tarefa é de $10$ unidades de medida de avaliação.

Os resultados obtidos com a execução de experimentos usando este cenário, e as políticas *sorteio*, FIFO e SJF são apresentados na Figura 4.1.

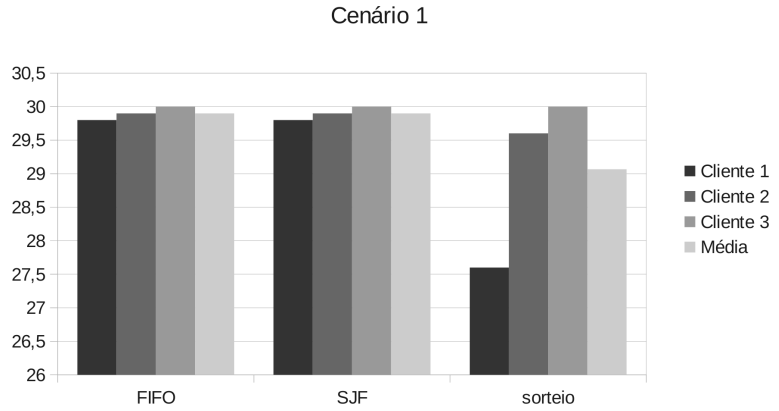


Figura 4.1: Comparação de desempenho das políticas usando tarefas homogêneas e de curta duração.

Neste cenário, tem-se que a política de *sorteio* obteve tempo de resposta melhor que as políticas FIFO e SJF, que por sua vez apresentaram comportamento semelhante.

### 4.2.2 Cenário 2: tarefas homogêneas e de longa duração

No Cenário 2, o servidor recebe requisições de tarefas de três clientes idênticos, cada um solicita ao servidor a execução de 100 tarefas. As tarefas são homogêneas e de longa duração, para contrastar com o Cenário 1. Como as tarefas são de longa duração, e a execução de uma nova tarefa é realizada justamente após a finalização da tarefa anterior, a demanda é mais baixa que a usada no Cenário 1. Nos experimentos do Cenário 2, a duração de cada tarefa é de 1.000 unidades de medida de avaliação.

Neste segundo cenário, obteve-se um resultado muito semelhante ao do Cenário 1. Mostrando que se os serviços pedidos pelos clientes forem homogêneos, temos um resultado melhor, na média do tempo de resposta, com a política de *sorteio*.

Os resultados obtidos com a execução de experimentos usando este cenário, e as políticas *sorteio*, FIFO e SJF são apresentados na Figura 4.2.

### 4.2.3 Cenário 3: tarefas heterogêneas

No Cenário 3, o servidor recebe requisições de três clientes distintos, um com 100 tarefas pequenas, outro com 100 tarefas médias e o terceiro com 100 tarefas grandes. Cada tarefa curta dura 10 unidades de medida de avaliação. Cada tarefa média dura 100 unidades de medida de avaliação. Finalmente, cada tarefa longa dura 1.000 unidades de medida de avaliação.

Neste terceiro cenário, foi observado um tempo de resposta muito menor no cliente

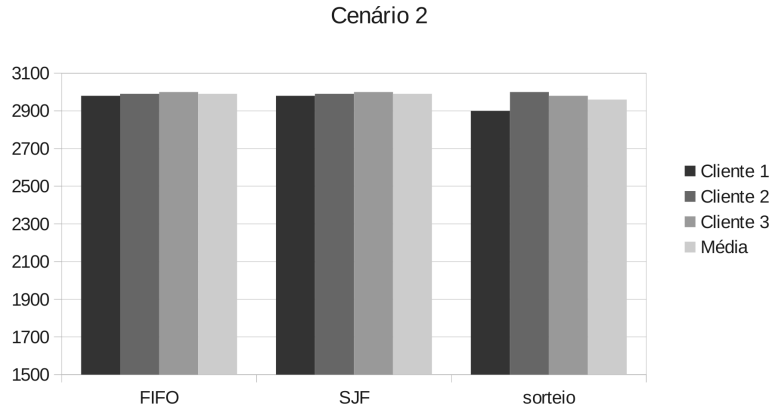


Figura 4.2: Comparação de desempenho das políticas no Cenário 2, usando tarefas homogêneas e de longa duração.

com tarefas curtas e no com tarefas médias com a política SJF. Esta política foi em média aproximadamente 64% melhor que as demais. Isso se deve ao fato de que o cliente que possuia apenas processos grandes passou por um período de $starvation$: ficou esperando na fila até que todas as tarefas dos dois primeiros clientes foram executadas. Nota-se também que o tempo de resposta para os clientes com tarefas pequenas e médias é o mesmo. Isso se deve ao fato de que ao ser finalizada uma tarefa do cliente que possui apenas tarefas pequenas, a fila de processos possuia apenas uma tarefa média e uma longa. Assim, a tarefa média era a menor no momento, sendo escolhida para ser a tarefa realizada antes de chegar a requisição da nova tarefa pequena. Foi observado, ainda, que as tarefas pequenas e média ficaram se intercalando durante o escalonamento.

Os resultados obtidos com a execução de experimentos usando este cenário, e as políticas *sorteio*, FIFO e SJF são apresentados na Figura 4.3.

Uma particularidade deste cenário, que pode ser observada na Figura 4.3, é que a política SJF foi a mais injusta entre as avaliadas.

### 4.2.4 Cenário 4: tarefas heterogêneas, com mudança de demanda

No Cenário 4, o servidor recebe requisições de três clientes, cada um deles possui $50$ tarefas grandes seguidas de $50$ tarefas pequenas. Isto é, a demanda muda no decorrer do período em questão.

Neste cenário, apesar das diferenças nos tempos, foi observado que ocorreu a mesma situação de $starvation$ do cenário anterior, não tão severa devido ao fato do cliente $3$ ter tido metade das suas requisições feitas na metade do experimento. As $50$ tarefas pequenas foram

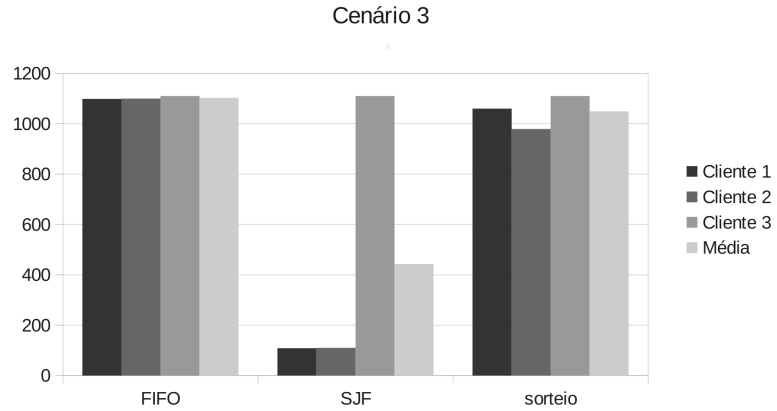


Figura 4.3: Comparação de desempenho das políticas no Cenário 3, usando tarefas heterogêneas.

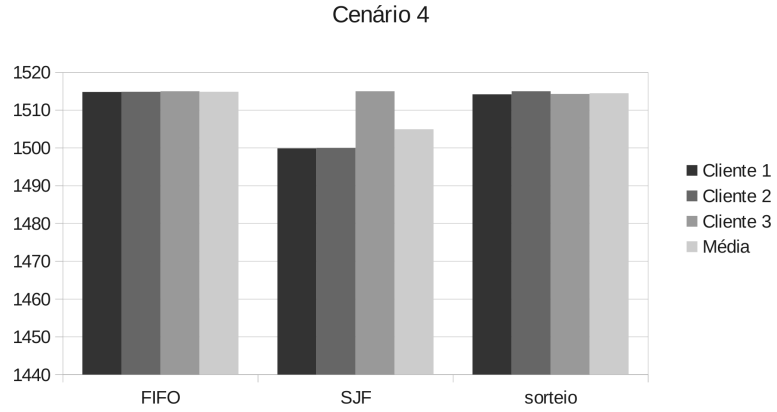


Figura 4.4: Comparação de desempenho das políticas no Cenário 4, com mudança de demanda.

executadas antes de se começar a execução das tarefas grandes dos clientes 1 e 2.

Para a solução da situação de $starvation$, neste caso, poderia ser implementada a mesma política, porém diminuindo o peso das requisições segundo a sua antiguidade.

Os resultados obtidos com a execução de experimentos usando este cenário, e as políticas *sorteio*, FIFO e SJF, são apresentados na Figura 4.4.

## 4.3 Experimentação e definição de cenários de testes para a adaptação

Depois de comparar as políticas e verificar qual política é mais adequada para cada situação, as políticas foram avaliadas novamente desta vez aos pares, visando adaptação no escalonamento. Ao invés de mudar o cenário e aguardar que o escalonador decida a política, como deveria ser o procedimento do escalonador adaptativo, nestes novos experimentos, a mudança da política é forçada justamente para verificar se o desempenho do serviço melhora. De forma geral, conforme o esperado, o serviço melhorou.

Para demonstrar situações de adaptação, foram realizados testes com os quatro cenários


Figura 4.5: Adaptação mudando da política SJF para *sorteio* no Cenário 1.

analizados. Porém, nesta nova bateria de experimentos, o escalonamento muda sua política no momento em que chega à metade da demanda dos clientes, buscando melhoria no serviço.

### 4.3.1 Adaptação no Cenário 1

O gráfico da Figura 4.5 apresenta a simulação da adaptação no Cenário 1, que é o cenário com tarefas pequenas e homegêneas. No início do teste, o escalonador usa a política SJF. Porém, ao executar a metade da demanda dos 3 clientes, ou seja 150 tarefas, o escalonador muda a sua política para a de *sorteio*. Nesta situação, o tempo médio de resposta diminuiu sutilmente de 29 para 28 unidades de medida de avaliação.

### 4.3.2 Adaptação no Cenário 2

O gráfico da Figura 4.6 ilustra a simulação da adaptação no Cenário 2, com tarefas homogêneas e de longa duração. Neste experimento, o escalonador usa, inicialmente, a política SJF. Porém, ao executar metade da demanda dos 3 clientes, 150 tarefas, o escalonador, novamente, muda para a política de *sorteio*. Nesta situação, o tempo médio de resposta diminuiu, novamente, sutilmente, de 2.979 para 2.869 unidades de medida de avaliação.

### 4.3.3 Adaptação no Cenário 3

O gráfico da Figura 4.7 ilustra a simulação da adaptação no Cenário 3, com tarefas heterogêneas. Neste experimento, o escalonador usa, inicialmente, a política FIFO. Porém, ao executar metade da demanda dos 3 clientes, 150 tarefas, o escalonador muda a sua política para a SJF. Nesta situação, o tempo médio de resposta diminuiu consideravelmente, mais que a metade,

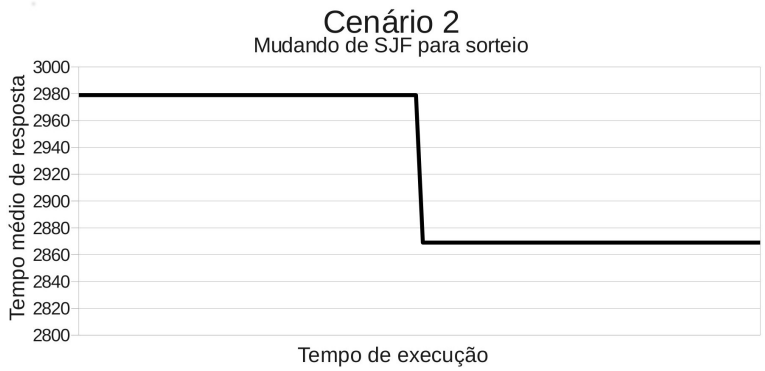


Figura 4.6: Adaptação mudando da política SJF para *sorteio* no Cenário 2.

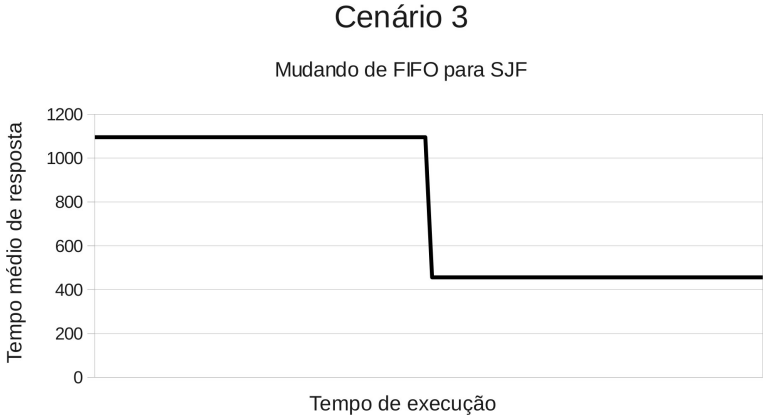


Figura 4.7: Adaptação mudando da política FIFO para SJF no Cenário 3.

de $1.095$ para $456$ unidades de medida de avaliação. No cenário com tarefas heterogêneas, fica mais fácil de se perceber o ganho com a troca de política. De fato, sabe-se que cenários com tarefas heterogêneas são frequentes em aplicações reais.

### 4.3.4 Adaptação no Cenário 4 (adaptado)

Finalmente, o gráfico da Figura 4.8 ilustra a simulação da adaptação no Cenário 4, tarefas heterogêneas, com mudança de demanda. Porém, este cenário foi ligeiramente adaptado para este novo experimento. Agora, os clientes solicitam $25$ tarefas grandes, seguidas por $25$ tarefas pequenas, seguidas por mais $25$ tarefas grandes e $25$ pequenas.

No início do experimento, o escalonador usa a política FIFO. Porém, ao executar metade da demanda dos $3$ clientes, $150$ tarefas, o escalonador muda a sua política para a SJF. Nesta situação, o tempo médio de resposta diminuiu, muito sutilmente, de $1.484$ para $1.469$ unidades de medida de avaliação.

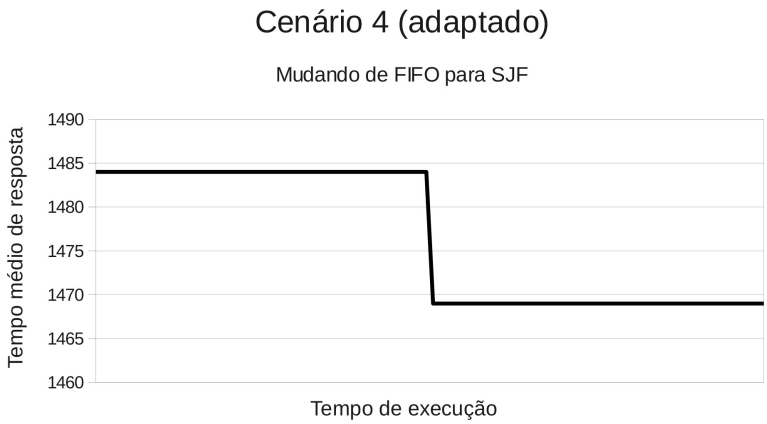


Figura 4.8: Adaptação mudando da política FIFO para SJF no Cenário 4 (adaptado).

## 4.4 Análise de resultados

Analisando-se os resultados obtidos através nos experimentos nos diferentes cenários, pode ser identificada uma sutil vantagem do escalonamento por *sorteio* em situações onde todos os clientes possuem tarefas de tamanho semelhante.

Entretanto, em um cenário com tarefas de tamanhos distintos, observa-se melhor desempenho da política SJF. Assim sendo, observando o perfil dos clientes, o escalonador pode adaptar a sua política para melhor atendê-los.

Por fim, quando a adaptação, a mudança da política FIFO para SJF apresentou os melhores resultados.

# 5 CONSIDERAÇÕES FINAIS

## 5.1 Conclusões

Este trabalho apresentou um escalonador de tarefas adaptativo, no contexto de sistemas computacionais. Foram implementadas e avaliadas diferentes políticas de escalonamento de processos (FIFO, SJF e *sorteio*) usando o suporte da linguagem de programação Lua. O escalonador adapta-se conforme o tempo de resposta fornecido pelos clientes.

Para possibilitar a avaliação, foram realizados experimentos contemplando quatro diferentes cenários de testes, com diferentes quantidades de tarefas e diferentes pesos para estas tarefas (tarefas pequenas, médias e grandes).

Analisando-se os resultados obtidos através nos experimentos nos quatro diferentes cenários, em um ambiente controlado e livre de falhas, pode ser identificada uma certa vantagem do escalonamento por *sorteio* em situações onde todos os clientes possuem tarefas de tamanho semelhante. Entretanto, em um cenário com tarefas de tamanhos distintos, observa-se melhor desempenho da política SJF. Quando a adaptação, os experimentos demostraram que a mudança da política FIFO para SJF apresentou os melhores resultados.

## 5.2 Trabalhos futuros

Como continuidade deste trabalho, outras políticas e métricas poderiam ser usadas para avaliar a adaptabilidade em escalonamento de processos. Assim, trabalhos futuros incluem os seguintes itens:

- Usar um esquema descentralizado e investigar distribuição de carga entre os servidores bem como balanceamento de carga;
- Investigar cenários onde a frequência do envio de tarefas entre os clientes é diferente. Por exemplo, um processo demanda muitas tarefas ao servidor enquanto outro processo solicita a execução de poucas tarefas.
- Avaliar outras políticas para escalonamento de tarefas. Políticas mais complexas poderão ser avaliadas, por exemplo, aquelas que implementam filas de requisições através de árvores (*heaps*).

- Utilizar mineração de dados para predizer o comportamento dos clientes e escolher a política apropriada.

# REFERÊNCIAS

BADONELL, R.; BURGESS, M. Service Load Balancing with Autonomic Servers: Reversing the Decision Making Process, In: *Proceedings of AIMS 2008. LNCS 5127.* pp. 92-104.

BOUCHENAK, S.; DE PALMA, N.; HAGIMONT, D.; TATON, C. Autonomic Management of Clustered Applications, In: *Cluster Computing, 2006 IEEE International Conference on*, vol., no., pp.1-11, 25-28 Sept. 2006.

CERVIER, A. Uma Abordagem de escalonamento adaptativo no ambiente Real-Time CORBA. SBRC'2002 - XX Simpósio Brasileiro de Redes de Computadores. Buzios - RJ, 2002.

EWING, J. M.; MENASCÉ, D. A. Business-Oriented Autonomic Load Balancing for Multitiered Web Sites In *Proceedings of MASCOTS 2009, London, UK, 2009*

KEPHART, J. O.; CHESS, D. M. The Vision of Autonomic Computing In *Cover Feature, IEEE Computer Society, January 2003.*

LI, W.-S.; ZILIO, D. C.; BATRA, V. S.; SUBRAMANIAN, M.; ZUZARTE, C.; NARANG, I. Load Balancing for Multi-tiered Database Systems through Autonomic Placement of Materialized Views, Data Engineering, In: International Conference on, página 102, 22nd International Conference on Data Engineering (ICDE'06), 2006.

ROSSI, F. D. Alocação Dinâmica de Recursos do XEN, In: *Dissertação de Mestrado. Pontifícia Universidade do Rio Grande do Sul.* 70p.

TANENBAUM, A. S. 2007. Modern Operating Systems (3rd ed.). Prentice Hall Press, Upper Saddle River, NJ, USA.

TANENBAUM, A. S. 2007. Sistemas Distribuídos (2a ed.). Prentice-Hall, Rio de Janeiro, RJ, Brasil.

ZHANG, H.; QIU, X.; MENG, L.; ZHANG, X. Design of Distributed and Autonomic Load Balancing for Self-Organization LTE, In: IEEE 72nd Vehicular Technology Conference Fall (VTC 2010-Fall), 2010 pp. 1-5.

# APÊNDICES

# APÊNDICE A – Detalhes da implementação

## A.1 Cliente

```
function roda_cliente(lista_tarefas)
   --[[
   função que recebe uma lista de tarefas a serem
   enviadas ao servidor.
   listas da forma:
   L={300, 50, 400, 5000}
   que comtenham o tamanho das tarefas a serem
   cumpridas na ordem em que elas aparecem na lista.

   chamar usando:
   concurrent.spawn(rodacliente, L)
   ]]

   for i = 1, #lista_tarefas do
      --[[
      Laço que itera sobre cada tarefa da lista.
      ]]

      --[[
      Envia mensagem para o relógio, solicitando
      o tempo atual e recebe esse valor na variável tempoi.
      ]]
      concurrent.send(concurrent.whereis("relogio"),
            {from=concurrent.self(), le=true})
      local tempoi=concurrent.receive()

      --[[
      Envia a mensagem com a tarefa para o servidor.
      ]]
      local enviou=false
      while not enviou do
         enviou=concurrent.send(concurrent.whereis("servidor"),
                  {
                  from = concurrent.self(),
                  tarefa=lista_tarefas[i]
                  }
      )
   end
      --[[
      Recebe a resposta do servidor de que a tarefa foi concluída.
      concurrent.receive() é bloqueante, o processo cliente fica
      inativo até que receba essa resposta.
      ]]
   local msg = concurrent.receive()
      --[[
      Pede mais uma vez o tempo atual ao relógio.
      Guarda este valor na variável tempof.
      ]]
   concurrent.send(concurrent.whereis("relogio"),
         {from=concurrent.self(),le=true})
   local tempof=concurrent.receive()
      --[[
      Nesta parte do código podemos usar os tempos
      recebidos do relógio para tratar destes dados.
      Pode-se gravar esses dados em um log.
```

```
56          Pode-se enviar estes dados a um outro processo
57          que armazene estas informações.
58          Este outro processo pode ser o processo que torna
59          o escalonamento adaptativo.
60          Estas informações estão disponíveis
61          nas variáveis:
62          tempoi - tempo do relógio no momento em que a mensagem
63          pedindo   a realização da tarefa ao servidor foi enviada.
64          tempof - tempo do relógio no momento em que a mensagem
65          avisando que a tarefa foi concluída.
66          tempof-tempoi - tempo de execução da tarefa.
67          concurrent.self() - identificador deste processo cliente.
68          lista_tarefas[i] - tarefa que foi enviada ao servidor
69          i - número da tarefa.
70          ]]
71      end
72  end
```

## A.2 Servidor

```
function fifo(f)
    --[[
    função que retira o primeiro elemento da fila.
    Como neste caso os novos elementos estão sendo inseridos
    no começo da fila, temos que o primeiro da fila é o que
    foi há mais tempo inserido, logo retirando esse elemento
    temos implementada uma fila FIFO.
    ]]
    return table.remove(f, 1)
end

function sjf(f)
    --[[
    função que retira um elemento da fila no modo
    trabalho mais curto primeiro.
    ]]
    local indice=1
        for i,j in ipairs(f) do
            if j.tarefa<f[indice].tarefa then
                indice=i
            end
        end
    return table.remove(f, indice)
end

--[[
seta um valor como semente para o gerador de números aleatórios de Lua.
]]
math.randomseed (100)

function sorteio(f)
    --[[
    função que retira um elemento da fila no modo
    sorteio, aleatório.
    ]]

    return table.remove(f, math.random(#f))
end

--[[
Guarda na variável escolhe a função que será usada
para escalonar os processos.
Esta variável pode receber qualquer uma das funções acima
(fifo, sjf, sorteio), ou receber qualquer outra função
implementada para isso.
Esse valor pode ser modificado em qualquer momento,
mudando assim a política de escalonamento.
]]
escolhe=fifo

function trabalho(t)
    --[[
    Faz o trabalho pedido pelo cliente.
    Neste caso apenas um sleep.
    ]]
```

```
56      concurrent.sleep(t)
57  end
58
59  function executor(server)
60      --[[
61      Função que implementa o comportamento do processo encarregado de
62      realizar as tarefas dos clientes.
63      Estas tarefas ele recebe do servidor que gerencia a fila de
64      tarefas.
65
66      chamar usando:
67      servidor_id=concurrent.spawn(servidor)
68
69      ]]
70      local cliente=nil
71      while true do
72          if cliente then
73          --[[
74          Envia a mensagem ao cliente informando que a
75          tarefa requisitada foi concluída.
76          Caso a variável cliente seja nula
77          (cliente==nil), indica que este processo não
78          recebeu tarefa alguma do servidor na última
79          iteração do laço.
80          ]]
81              local enviou=false
82              while not enviou do
83                  enviou=concurrent.send(cliente,"PRONTO")
84              end
85          end
86
87          --[[
88          Solicita nova tarefa ao servidor que gerencia a fila.
89          ]]
90          local enviou=false
91              while not enviou do
92                  enviou=concurrent.send(server,{from="executor"})
93              end
94
95          --[[
96          Recebe nova tarefa do servidor da fila.
97          Caso haja tarefas esperando na fila, o servidor
98          devolverá uma mensagem com o identificador do cliente e
99          com a tarefa a ser realizada.
100         Caso a fila esteja vazia, o servidor devolverá nil como
101         identificador do cliente.
102         ]]
103         local msg=concurrent.receive()
104
105         cliente=msg.cliente
106         --[[
107         A função trabalho(t) recebe o trabalho a ser computado
108         para o cliente.
109         ]]
110         trabalho(msg.trabalho)
111         --[[
112         Envia uma mensagem ao processo que cuida do "relógio"
113         para incrementar o tempo, segundo o tamanho da tarefa
```

```
114         do cliente.
115         concurrent.whereis("relogio"), retorna o identificador
116         do processo registrado com a string "relogio".
117         ]]
118         concurrent.send(concurrent.whereis("relogio"), {grava=true, tempo=msg
        .tarefa})
119     end
120 end
121
122
123 function servidor()
124     --[[
125     Processo que gerencia a fila, recebe as tarefas dos clientes e
126     as envia para serem executadas pelo processo executor.
127     ]]
128
129     --[[
130     Regitra este processo com a string "servidor", para que outros
131     processos possam entrar em contato com este.
132     concurrent.self() retorna o identificador do processo atual.
133     ]]
134     concurrent.register("servidor", concurrent.self())
135
136     --[[
137     Inicia uma fila vazia.
138     ]]
139     local fila={}
140
141     --[[
142     Inicia o processo executor.
143     ]]
144     executor_id=concurrent.spawn(executor, concurrent.self())
145
146     --[[
147     Inicia aqui o laço onde este processo se comunica com os demais.
148     ]]
149     while true do
150         --[[
151         Recebe uma mensagem de outro processo, guarda esta
152         mensagem na variável msg.
153         concurrent.receive() mantém este processo bloqueado até
154         que ele receba uma mensagem.
155         ]]
156         local msg = concurrent.receive()
157         if msg.from=="executor" then
158         --[[
159         Se a mensagem recebida for do processo executor,
160         ele está requisitando uma nova tarefa para ser
161         feita.
162         ]]
163             if #fila>0 then
164             --[[
165             Caso existam tarefas a serem feitas na
166             fila, envia uma mensagem ao executor
167             com a tarefa escolhida pelo escalonador.
168             escolhe(filha) é a função do escalonador.
169             ]]
170             local escolhido = escolhe(fila)
```

```
171             local enviou=false
172                 while not enviou do
173                     enviou=concurrent.send(executor_id,
174                         escolhido)
175                 end
176             else
177             --[[
178             Caso a fila esteja vazia, envia ao processo
179             executor uma mensagem informando que não
180             existem tarefas na fila por enquanto.
181             ]]
182             local enviou=false
183                 while not enviou do
184                     enviou=concurrent.send(executor_id,
185                         {cliente=nil, tarefa=0})
186                 end
187             end
188         else
189         --[[
190         Caso a mensagem não seja do executor,
191         ela é de um cliente.
192         A tarefa recebida pelo cliente é inserida na
193         fila de tarefas.
194         ]]
195             table.insert(fila,{cliente=msg.from, tarefa=msg.tarefa})
196         end
197     end
198 end
```

## A.3 Inicialização

```
function relogio()
   --[[
   Função que implementa o comportamento do processo relógio.
   ]]
   concurrent.register("relogio", concurrent.self())
   local tempo=0
   while true do
      local msg=concurrent.receive()
      if msg and msg.le==true then
         concurrent.send(msg.from, tempo)
      elseif msg and msg.grava==true then
         tempo=tempo+msg.tempo
      end
   end
end

--[[
Dispara o processo servidor
]]
concurrent.spawn(servidor)

--[[
Dispara o processo do relógio
]]
concurrent.spawn(relogio)

--[[
Cria as listas de processos para serem usadas pelos clientes.
Modificar estas listas modifica os diferentes cenários.
]]
lista1={}
lista2={}
lista3={}
--[[
Inicia as listas com 100 posições com o valor 1000.
]]
for i=1, 100 do
   lista1[i]=1000
   lista2[i]=1000
   lista3[i]=1000
end

--[[
Dispara os processos clientes.
]]
concurrent.spawn(roda_cliente,lista1)
concurrent.spawn(roda_cliente,lista2)
concurrent.spawn(roda_cliente,lista3)

--[[
Inicia o loop de execução em paralelo.
]]
concurrent.loop()
```