UNIVERSIDADE FUMEC
FACULDADE DE CIÊNCIAS EMPRESARIAIS - FACE

SÉRGIO HENRIQUE MIRANDA JÚNIOR

Arquitetura de Sistemas *Web* baseado em *evented i/o*

BELO HORIZONTE
2011

SÉRGIO HENRIQUE MIRANDA JÚNIOR

Arquitetura de Sistemas *Web* baseado em *evented i/o*

Monografia apresentada à Universidade FUMEC, no curso de Ciência da Computação, apresentado à disciplina Trabalho de Conclusão de Curso.

Orientador:
Professor Flávio Veloso Laper
Co-orientador:
Professor Ricardo Terra

BELO HORIZONTE
2011

Monografia **Arquitetura de Sistemas *Web* baseado em *evented i/o***, apresentada ao Curso de Ciência da Computação, da Universidade FUMEC-FACE, de autoria de Sérgio Henrique Miranda Junior, aprovada pela Banca Examinadora constituída pelos seguintes professores:

_____

Professor Flávio Veloso Laper

_____

Professor Osvaldo Manoel Corrêa

_____

Professor

Belo Horizonte, _____ de Dezembro de 2011.

AGRADECIMENTOS

Em especial ao Orientador, Professor Flávio Veloso Laper e Co-orientador, Professor Ricardo Terra,pelo ensinamento e dedicação dispensados no auxílio à concretização deste trabalho.

RESUMO

Com a evolução da tecnologia e dos sistemas *web*, os *sites* que antes apresentavam apenas conteúdo estático passaram a abranger uma grande variedade de categorias. Esses novos serviços oferecidos pela *web* apoiam o trabalho de equipes em escala geográfica mundial, sendo assim, o tempo de resposta para uma requisição deve ser o mais reduzido possível para manter a produtividade na utilização. Ainda, esses novos serviços realizam uma grande quantidade de operações de *input*/*output*. Essas operações, por demandarem um alto custo computacional, passam a aumentar o tempo de resposta desses sistemas. Nesses casos, a utilização da Arquitetura *Evented I*/*O* é adequada, pois realiza operações de *I*/*O* de forma não bloqueante. Em vistas disso, este estudo apresenta os conceitos de operações de *I*/*O* bem como a utilização da Arquitetura *Evented I*/*O* para minimizar o impacto dessas operações. Ainda, este estudo ilustra um estudo de caso comparando a Arquitetura *Thread Based* e a Arquitetura *Evented I*/*O* demonstrando através de uma análise de desempenho os reais ganhos da Arquitetura *Evented I*/*O*.

Palavras-chave: *Event*, Disponibilidade, *Evented I*/*O*, *Thread*, *Thread Based*,*I*/*O*

ABSTRACT

As technology and web-based system evolve, websites that used to have only static content started to have a large variety of services and categories. These new services available at the web support team work on a global level. This means the response time of a request must be the lowest possible so that productivity can be maintained upon utilization. Also, these new services make a lot of input/output operations. These operations, for demanding high computational costs, tend to raise response time on those systems. In that cases, the use of Evented I/O Architecture is suited, because it makes I/O operations in a non-blocking way. Therefore, this study presents the concepts of I/O operations as well as the utilization of the Evented I/O Architecture to minimize the impact of such operations. Further on, this work illustrates a case study which compares Thread Based Architecture and Evented I/O Architecture showing, through a performance analysis, the real benefits of Evented I/O Architecture.

Keywords: Event, Availability, Evented I/O, Thread, Thread Based, I/O

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE SIGLAS

CPU	*Central Process Unit*
DMA	*Direct Memory Access*
FSM	*Finite State Machine*
RPM	Requisições por minute
HTTP	*Hypertext Transfer Protocol*
LTS	*Long Term Support*
RAM	*Random Access Memory*
SGDB	Sistema Gerenciador de Banco de Dados
URI	*Uniform Resource Identifier*

SUMÁRIO

INTRODUÇÃO

A Internet nos seus primórdios oferecia *sites* meramente informativos e com conteúdo estático. O usuário não interagia de forma ativa com o sistema web, era apenas um consumidor de informações. No entanto, com a evolução da tecnologia, esse cenário se alterou. Os *sites* passaram a oferecer serviços que abrangem uma grande variedade de categorias: negociação de ações, transmissões ao vivo, *e-commerce*, serviço de mensagem instantânea, *download p2p* e hospedagem de aplicações o que acaba por demandar um número significativo de processamento e recursos de *input/output* para processar cada requisição (WELSH, 2002). Ainda, segundo Jamil e Bax (2001), esses sistemas apoiam o trabalho de equipes em escala geográfica mundial, sendo assim, o tempo de resposta para uma requisição deve ser o mais reduzido possível para manter a produtividade na utilização. Ainda, o sistema deve estar preparado para atender ao crescimento no número de requisições realizadas pelos usuários e ser capaz de responder a todas.

Sendo assim, a arquitetura utilizada para desenvolver um sistema deve ser escolhida de forma que atenda suas necessidades acima mencionadas. Dessa forma, duas arquiteturas se fazem presente: Arquitetura *Thread Based* e Arquitetura *Evented Based*. A Arquitetura *Thread Based* utiliza um *thread* para cada requisição aceita. Ao criar um novo *thread* é necessário alocar memória para sua pilha de execução, ainda, essa pilha precisa ser gerenciada na memória principal (TANENBAUM, 2003). Ao realizar a troca de contexto entre processos o estado de cada *thread* precisa ser salvo para que possa ser restaurado posteriormente quando o *thread* voltar a executar. Assim, essa operação de troca de contexto demanda um grande esforço computacional para ser realizada (LI *et al*, 2007). Dessa maneira, a Arquitetura *Thread Based* gera um *overhead* na troca de contexto, pois podem existir vários *threads* criados. Já a Arquitetura *Evented Based* tem como característica a execução de um *loop* eterno, com poucos *threads*, que fica processando eventos. Assim, esse tipo de arquitetura evita o *overhead* na troca de contexto bem como a alta utilização de memória.

Além disso, sistemas *web* atuais realizam uma grande quantidade de operações de *input/output*, isto é, transferem informações armazenadas para o mundo externo (EL-GHAZAWI, 2003). Elas também apresentam um custo considerável para o sistema computacional, aumentando o tempo de resposta de um sistema para o usuário final. Esse esforço computacional é necessário, pois é preciso que a CPU execute o código do driver do dispositivo e ainda continue escalonando os processos de forma eficiente quando bloqueiam e desbloqueiam. Segundo Silberschatz (2004), existem dois tipos de chamadas de operações de *I/O* possíveis: *I/O* não bloqueante (ou assíncrono) e *I/O* bloqueante, sendo esta última a mais

utilizada. Ao realizar uma chamada de *I/O* bloqueante a aplicação é movida da fila de execução do sistema operacional para uma fila de espera. Quando a operação de *I/O* for completada a aplicação volta a ficar elegível para execução. Já a chamada de *I/O* assíncrona retorna imediatamente fazendo com que a aplicação possa continuar seu código enquanto a operação não é finalizada. Uma função de *callback* é vinculada junto a essa operação de *I/O* para ser executada quando a mesma terminar.

Nesse contexto, os sistemas que utilizam a Arquitetura *Thread Based* fazem uso de operações de *input/output* bloqueantes aumentando o tempo de resposta para o usuário final. Porém, a Arquitetura *Evented I/O*, que tem como característica um *loop* principal que fica processando eventos, utiliza operações de *input/output* assíncronas. Desse modo, esse tipo de arquitetura consegue diminuir o tempo de resposta para o usuário final, uma vez que ao realizar operações de *input/output* de forma assíncrona, o sistema é capaz de continuar processando as próximas requisições até que a operação de *input/output* seja completada.

Esta monografia está organizada conforme a seguir: O Capítulo 1 apresenta a Arquitetura *Thread Based* e a Arquitetura *Evented Based,* conceitualizando e comparando tais arquiteturas. Já o Capítulo 2 apresenta o conceito e o desempenho das operações de *I/O*, demostrando o custo computacional demandado por esse tipo de operação. O Capítulo 3 aborda a arquitetura *Evented I/O* demostrando seu contexto de utilização, diretrizes para o desenvolvimento, o padrão *Reactor* e um sistema de exemplo. Já no Capítulo 4, é desenvolvido um estudo de caso para demonstrar os reais ganhos da Arquitetura *Evented I/O*. Esse estudo de caso apresenta duas versões do sistema MyIO: uma utilizando a Arquitetura *Thread Based* e outra utilizando a Arquitetura *Evented I/O*. Ademais, é realizado uma análise de desempenho entre ambas as arquiteturas demonstrando o ganho obtido através da Arquitetura *Evented I/O*. Por fim, na Conclusão são apresentadas as considerações finais deste estudo.

CAPÍTULO 1 – ARQUITETURA *Thread Based* x *Evented Based*

De acordo com Terra (2009) "Arquitetura de software é geralmente definida como o conjunto de decisões de projeto que tem impacto em cada aspecto da construção e evolução de sistemas de software". Sabendo que os serviços atualmente disponibilizados na Internet são altamente dinâmicos e proporcionam grande interatividade com o usuário, precisa-se pensar em uma arquitetura que permita o crescimento da capacidade de atender a novas requisições[1] e que ainda mantenha o desempenho.

Nesse contexto, duas arquiteturas se fazem presente: *thread based* e *evented based.* A Arquitetura *Thread Based* basicamente, segundo Welsh (2002), cria um *thread* para cada requisição aceita. Já a Arquitetura *Evented Based*, segundo Welsh (2002), tem como característica a criação de poucos *threads* que ficam em *loop* contínuo processando eventos.

Sendo assim, este capítulo abordará de maneira detalhada esses dois tipos de arquiteturas além de demonstrar um comparativo entre as mesmas. O restante do capítulo está organizado da seguinte maneira: na Seção 1.1 é apresentada a Arquitetura *Thread Based* e, em seguida, na Seção 1.2, é apresentada a Arquitetura *Evented Based*; já a Seção 1.3 demonstrará um comparativo entre as arquiteturas apresentadas anteriormente; por fim, a Seção 1.4 traz as considerações finais.

1.1 Arquitetura *Thread Based*

Para conceituar *thread* deve-se definir um processo. Segundo Tanenbaum (2003), um processo é a abstração de um programa em execução que possui um espaço de endereçamento, ou seja, uma lista de locais da memória principal a partir de um mínimo até um máximo onde o processo pode ler e gravar. O espaço de endereçamento contém o programa executável, os dados do programa e sua pilha de execução. Ainda, segundo Tanenbaum (2003), associado a cada processo está um conjunto de registradores, incluindo o contador de programa, o ponteiro da pilha e outros registradores de hardware e as demais informações necessárias para executar o programa. Esse registro de estado, segundo Tanenbaum (2003), é necessário, pois a CPU é compartilhada entre os vários processos que podem estar executando e cada processo pode estar em três estados possíveis: em execução, pronto e bloqueado. Desta forma, ao voltar a executar o processo começa do ponto onde foi

[1]Requisição: Nessa monografia requisição é referenciada como uma requisição *Hypertext Transfer Protocol* (HTTP) que é um protocolo de comunicação (na camada de aplicação segundo o modelo OSI) baseado em cliente e servidor. Disponível em: <http://pt.wikipedia.org/wiki/Hypertext_Transfer_Protocol>. (Acesso em: 12 out. 2011)

interrompido. A essa alternância de processo em execução dá-se o nome de troca de contexto. Basicamente, a troca de contexto executa algumas operações: salvar e carregar registradores e mapas de memória, atualizar várias tabelas e listas etc. Assim, em termos de ciclos de CPU, a troca de contexto apresenta um custo alto. Resumidamente, Bovet *et al* (2005) metaforicamente definem um processo como o ser-humano: eles são gerados, possuem uma vida significativa, podem gerar um ou mais processos filhos e eventualmente morrem.

Nesse contexto, segundo Tanenbaum (2005), em sistemas operacionais modernos é fornecido suporte para múltiplas linhas de controle dentro de um processo. Desta forma, *thread* pode ser definido como tarefas de um mesmo processo que executam concorrentemente, compartilhando recursos, ou ainda, processos leves. Os *threads*, além de poderem estar nos mesmos estados de um processo, possuem seu próprio contador de programa, registradores e pilha de execução. Assim, a troca de contexto entre *threads* é mais rápida do que entre processos, mas ainda demanda certo esforço da CPU.

De acordo com Welsh (2002), arquiteturas de sistemas *web thread based* possuem um processo que irá atender as requisições dos usuários e esse processo irá gerar um *thread* para cada requisição aceita. Assim, à medida que o número de *threads* aumenta, a performance do sistema cai, uma vez que, ao criar um *thread* o *overhead* na troca de contexto aumenta, assim como a utilização de memória devido a criação do espaço de endereçamento para cada *thread*. A FIG. 1 ilustra a criação de um *thread* para cada requisição aceita pelo sistema. Basicamente, uma requisição será atendida pelo *dispatcher* e encaminhada para um novo *thread* que irá processar a mesma.

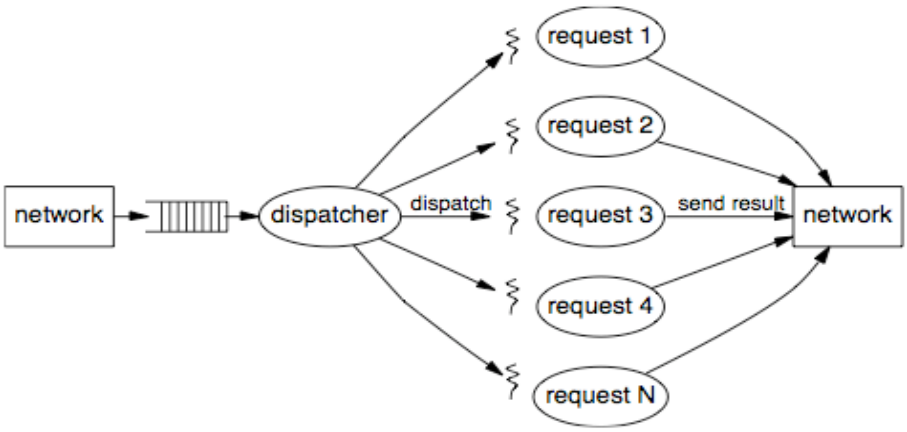

Figura 1 - Comportamento da Arquitetura *Thread Based*
Fonte: WELSH (2002, p. 16)

Uma forma para tentar minimizar esse problema é utilizar-se de poucos *threads* e, quando todos os *threads* estiverem ocupados – processando requisições – devem-se adicionar

as próximas requisições em uma fila de espera. Novos *threads* vão sendo criados para processar toda a fila existente até um número limite definido. Tal técnica é denominada *thread pooling* (WELSH, 2002). Além disso, é importante mencionar que pode acontecer contenção de recursos compartilhados entre os *threads* fazendo com que o tempo de resposta seja demorado.

Para exemplificar o uso dessa arquitetura um sistema real – `CupomNow` – foi escolhido. Este é um sistema web de compra coletiva com uma média de 3000 RPM (Requisições por minuto) diários. Por se comportar exatamente da maneira descrita anteriormente o sistema precisou utilizar *clusters* de servidores *web* para atender a todas requisições, pois aumentando do número de servidores, consequentemente, o número de *threads* que podem ser criadas também aumenta. Dessa forma, novas requisições que antes não eram atendidas passaram a ser processadas.

1.2. Arquitetura *Evented Based*

Segundo Núñez *et al* (2009), evento denota algo acontecendo em algum momento. Ainda, seguindo a definição de Núñez *et al* (2009), um evento tem uma fonte e um destino. Dessa maneira, arquiteturas *evented based* são organizadas processando eventos, ou seja, quando um programa ou processo não consegue completar alguma operação – como exemplo temos a espera por algum pacote solicitado via rede ou até mesmo o término de um operação de *I/O* – uma função de *callback*[2] é vinculada a essa operação (DABEK *et al*, 2002). Assim, arquiteturas *evented based* têm como característica um número limitado de *threads* que ficam em um *loop* eterno processando eventos de forma não bloqueante (WELSH, 2002).

Os eventos podem ser gerados pelo sistema operacional ou internamente pela própria aplicação, indicando que algum processamento deve ocorrer, por exemplo: uma notificação de término ou inicio para uma operação de *I/O*, uma operação de leitura utilizando a interface de rede ou outros eventos específicos da aplicação (WELSH, 2002). Esses eventos são processados pelas funções de *call-backs* vinculadas para serem executadas quando se fizer necessário.

Dessa maneira, cada requisição ao sistema é representada como uma máquina de estado finita (*Finite State Machine* - FSM), cada estado na FSM representa um conjunto de

[2]callback: É uma referência a um código executável, ou a uma parte de código executável, que é passado como argumento para outro código. Disponível em: <http://en.wikipedia.org/wiki/Callback_(computer_programming)>. Acesso em: 12 out. 2011.

passos que devem ser executados na requisição que está sendo processada (WELSH, 2002). Esse comportamento está ilustrado na FIG. 2.

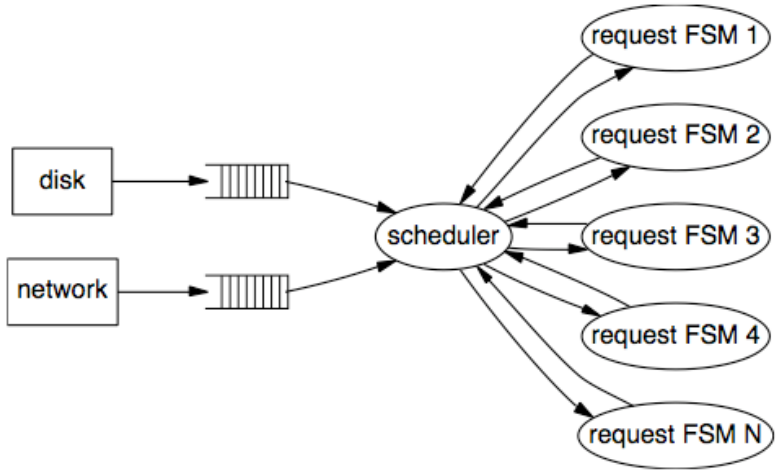

Figura 2 - Representação de uma requisição em uma Arquitetura *Evented Based*
Fonte: WELSH (2002, p. 23)

Exemplificando: a máquina de estado finito para a requisição de um arquivo estático poderia ser a seguinte:

1 – Receber e ler a requisição do usuário;

2 – Realizar o *parse* da requisição (processar os cabeçalhos HTTP);

3 – Procurar o arquivo em disco solicitado;

4 – Ler o arquivo;

5 – Construir o pacote que será devolvido aos usuários;

6 – Responder a requisição do usuário;

A FIG. 3 ilustra esse comportamento.

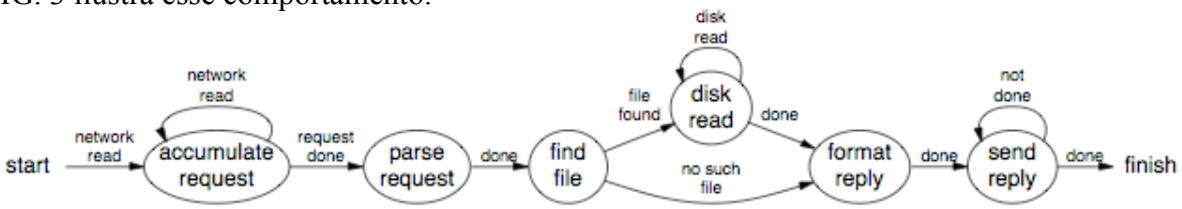

Figura 3 - Representação da maquina de estado finito para uma requisição de um arquivo estático
Fonte: WELSH (2002, p. 24)

Assim, quando uma requisição chega ao processo que irá executá-la, ela é encaminhada a um dos poucos *threads* existentes, processada de acordo com seu estado (WELSH, 2002) e, quando o processamento é finalizado, a FSM passa para um novo estado e o *thread* utilizado é encaminhado para o próximo estado da FSM (WELSH, 2002).

O processamento de várias requisições, utilizando um pequeno número de *threads*, é feito através da "troca de contexto" após cada interação do *loop* eterno em que o processo se encontra (WELSH, 2002). Essa "troca de contexto" realizada é diferente da que acontece originalmente através do sistema operacional, pois o sistema operacional não precisa realizar nenhuma operação, pois a aplicação implementa sua própria política de escalonamento (WELSH, 2002).

Para exemplificar o uso dessa arquitetura um sistema real – `PostRank` – foi escolhido. O sistema é um sistema *web* que monitora e coleta o engajamento social, relacionado com algum conteúdo que foi publicado, reunindo onde e quando o conteúdo publicado gerou comentários, *tweets* e outras formas de interação. Esse monitoramento é realizado em um formato *real-time*. O PostRank utiliza um servidor criado por eles, *Goliath*[3], que por sua vez utiliza a Arquitetura *Evented Based*. Tudo que acontece é realizado de forma assíncrona e processado por *callbacks* que estão vinculadas aos eventos que podem ocorrer. Com tal arquitetura, eles conseguem atender mais de 500 requisições por segundo.

1.3 Comparativo

O debate existente entre escolher a Arquitetura *Thread Based* ou a Arquitetura *Evented Based* para o sistema de software, não é recente (DABEK *et al,* 2002). Esse debate é centrado em questões de escalabilidade, propriedade fundamental que todo sistema *web* deve atender. Assim, quando o pensamento é criar um serviço que irá escalar para milhares de requisições, existem duas características que influenciam na escolha: computação e *I/O*. Computação é a parte da aplicação que utiliza a maior quantidade disponível de CPU, isto é, a parte de componentes funcionais que transformam dados de uma forma para outra, por exemplo: compressão de imagens, tradução de textos para outras línguas, ordenar dados de acordo com alguma regra, etc (SEEGER *et al*, 2010). Já *I/O* são operações de entrada ou saída de dados em dispositivos conectados, os quais podem ser placa de rede, disco rígido ou qualquer outro que não esteja diretamente conectada à memória RAM ou à CPU (SEEGER *et al*, 2010).

Assim, segundo Welsh (2002), o rápido crescimento da Internet favoreceu novos serviços que fazem grande utilização de processamento computacional e operações de *I/O*.

---

[3]*Goliath:* É um servidor de aplicação e um *framework* arquiteturado para atender os seguintes objetivos: processamento totalmente assíncrono, configuração simples e alto desempenho. Disponível em: <http://www.igvita.com/2011/03/08/goliath-non-blocking-ruby-19-web-server/>. (Acesso em: 12 out. 2011)

Esses serviços abrangem uma grande variedade de categorias, como: negociação de ações, transmissões ao vivo, *e-commerce*, serviço de mensagem instantânea, *download p2p* e hospedagem de aplicações. Segundo Jamil e Bax (2001), esses sistemas apoiam o trabalho de equipes em escala geográfica mundial, sendo assim, o tempo de resposta para uma requisição deve ser o mais reduzido possível para manter a produtividade na utilização dos serviços.

Entende-se que, nesse contexto, uma Arquitetura *Thread Based* a cada nova requisição aceita, cria-se um novo *thread*, uma vez que na Arquitetura *Evented Based* poucos *threads*, que ficam em um *loop* eterno respondendo a eventos, são criados.

	Arquitetura *Thread Based*	Arquitetura *Event Based*
Overhead na mudança de contexto	**X**	
Alto uso de memoria *RAM*	**X**	
Implementação de um *scheduler*		**X**
Realização*I/O* de forma bloqueante	**X**	
Programação de maneirasequencial	**X**	

Quadro1 - Comparativo entre Arquitetura *Thread Based* e *Evented Based*
Fonte: Do autor.

Analisando o Quadro 1, que realiza um comparativo entre as arquiteturas, o primeiro ponto apontado é o *overhead* na mudança de contexto. Sabemos que esse tipo de operação demanda um grande esforço computacional para ser realizada (LI *et al,* 2007). À medida que a quantidade de *threads* aumenta, a troca de contexto fica mais demorada, pois o estado de cada *thread* precisa ser salvo para que possa ser restaurado posteriormente, quando o *thread* voltar a executar (WELSH, 2002). Já na Arquitetura *Evented Based* esse tipo de *overhead* não acontece, pois não são criados vários *threads* para cada requisição aceita. Dessa maneira, conseguimos eliminar o alto custo imposto pela troca de contexto e minimizar o tempo de resposta à requisição do usuário final.

O segundo ponto apresentado é o alto uso da memoria RAM por parte da Arquitetura *Thread Based*. Cada *thread* possui seu próprio controle de execução junto ao processador e, também, sua própria pilha. A cada novo *thread* criado uma nova estrutura de controle, junto ao processador, é criada e uma nova pilha de execução precisa ser gerenciada na memória principal (TANENBAUM, 2003). Já na Arquitetura *Evented Based,* a aplicação possui um *loop* eterno – com poucos *threads* – que fica processando eventos, e esses eventos são processados pela função de *callback* vinculada. O alto uso de memoria RAM é evitado, pois não existe uma grande quantidade de *threads* para ser gerenciado (WELSH, 2002).

O terceiro ponto diz respeito à implementação de um *scheduler* necessário na Arquitetura *Evented Based*. O *scheduler* é o responsável em delegar os eventos que ocorrem na aplicação (WELSH, 2002). A Arquitetura *Thread Based* não precisa desse tipo de funcionalidade já que os *threads* são todos gerenciados pelo sistema operacional (TANENBAUM, 2003).

O quarto ponto mostra que operações de *I/O* são realizadas de forma bloqueante na Arquitetura *Thread Based.* Como exposto anteriormente, nesse capítulo, a utilização desse tipo de operação se faz bastante presente nos sistemas *web* atuais, além de seu custo ser determinante no tempo de resposta final. Sistema *web* que utilizam a Arquitetura *Evented Based* não sofrem desse problema, pois utilizam interfaces que não bloqueiam e realizam tarefas de forma assíncrona (WELSH, 2002).

O quinto ponto colocado é a maneira como a programação é realizada. Na Arquitetura *Thread Based* a linha de execução do código é realizada de maneira sequencial, facilitando a leitura e o entendimento do código escrito. Já na Arquitetura *Evented Based* toda a aplicação é baseada em funções de *callback* que estão vinculadas aos eventos que podem ocorrer. Assim, a leitura e o entendimento do código se tornam mais complexos, pois o código não mais é executado de forma sequencial (DABEK *et al*, 2002).

1.4 Considerações Finais

A Arquitetura *Thread Based* apresenta um grande *overhead*, tanto na utilização de memória quanto na troca de contexto, por criar um novo *thread* a cada requisição aceita. Dessa maneira é preciso gerenciar toda a pilha de execução e o espaço de endereçamento de cada *thread*. Operações que bloqueiam, qualquer tipo de operação de *I/O*, fazem com que o tempo de resposta aumente na Arquitetura *Thread Based*. Em contrapartida, a Arquitetura *Evented Based* tenta resolver esses problemas e diminuir o tempo de resposta que é necessário para completar cada requisição. Com a utilização de um *loop* eterno, com poucos *threads*, que fica processando eventos, a utilização de memória bem como o *overhead* na troca de contexto é menor, diminuindo assim o tempo gasto para processar a requisição do usuário.

Segundo Welsh (2002), os serviços oferecidos via *web* hoje demandam cada vez mais da utilização de *I/O*, uma operação impactante no tempo de resposta ao usuário final. Assim, o próximo capítulo aborda esse tipo operação.

CAPÍTULO 2 -*I/O*

Como citado no primeiro Capítulo desse estudo, operações de *I/O* apresentam um custo considerável para o sistema computacional, aumentando o tempo de resposta de um sistema para o usuário final. Nesse contexto, segundo Welsh (2002), os serviços oferecidos pela *web* atualmente demandam significativo processamento e recursos de *input/output* para processar cada requisição.

Diante disso, este capítulo apresenta um detalhamento das mais usuais operações de *input/output* organizado como a seguir: na Seção 2.1 conceitua-se o que é *I/O*; na Seção 2.2 apontam-se os custos e desempenho de tais operações; por fim na Seção 2.4 são apresentadas as considerações finais.

2.1 Conceitualização

Ao visitar uma página na *web* ou editar um arquivo, o interesse imediato é ler ou entrar com algumas informações, e não calcular uma resposta. Para a apresentação dessa página, ou arquivo, ao usuário, o computador precisa realizar alguma operação que viabilize tais ações; o mesmo acontece ao entrar com informações desejadas para serem salvas. A essa operação dá-se o nome de *input/output (*SILBERSCHATZ *et al,* 2004).

Assim, sistemas de *input/output* transferem informações entre a memória principal do computador e o mundo externo (EL-GHAZAWI, 2003). Um sistema de *input/output* é composto por dispositivos ou periféricos, *I/O control unit* e um software (*drive*), que será responsável por cuidar da transmissão de dados através de uma sequência de operações de *input/output.*

Os dispositivos que compõem um sistema de *input/output* podem ser de armazenamento (discos, fitas), de transmissão (placas de rede, modems) e de interface humana (monitor, teclado, mouse). Ainda, é importante mencionar que variam muito em funcionalidades e também em sua velocidade (SILBERSCHATZ *et al,* 2004*)*.

Em um sistema de *input/output* um ponto importante, e também a principal diferença entre eles, é o quanto a CPU precisa se envolver para a operação ser completada. Em um sistema, com o mínimo de *I/O control unit*, a CPU é que realiza todas as operações de *input/output* necessárias. Isso inclui inicializar a transação, verificar o *status* do dispositivo, transferir os dados entre o dispositivo e a memória principal e finalizar a operação de *input/output*. Já em sistemas que possuem um co-processador de *I/O*, a CPU precisa apenas

inicializar a transação, sendo que, o resto é realizado pelo co-processador de *I/O* (EL-GHAZAWI, 2003).

Os sistemas de software, ou drivers de dispositivos, apresentam uma interface uniforme de acesso ao dispositivo de *I/O* para que o subsistema de *input/output* possa controlar a transmissão de dados entre o dispositivo e a memória principal (SILBERSCHATZ *et al,* 2004*)*. A FIG. 4 ilustra uma arquitetura de sistema de *input/output* controlada exclusivamente pela CPU. Basicamente, a CPU comunica com a interface de *input/output* (*driver*) através do barramento, o controle do *driver* e a leitura ou escrita de dados é feita utilizando os registradores de controle e dados respectivamente. A comunicação com o dispositivo é feita pelo *driver*, tornando a interface de comunicação com a CPU independente.

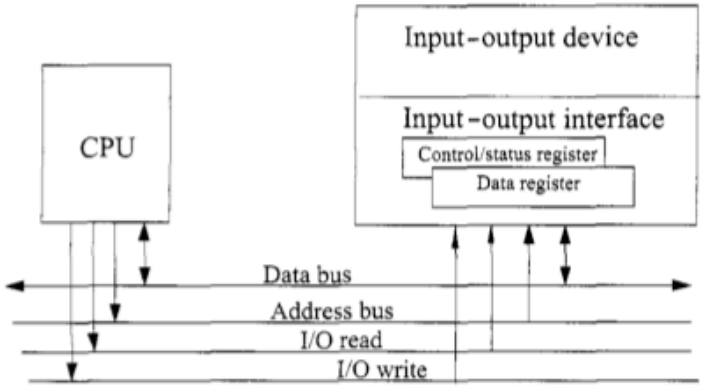

Figura 4 - Representação de um sistema de input/output controlado pela CPU
Fonte: EL-Ghazawi (2003)

Nesse contexto, para que a CPU possa se comunicar e diferenciar entre os diversos dispositivos de *I/O* no sistema, um esquema de endereçamento se faz necessário. É possível utilizar-se de dois métodos de endereçamento: *memory mapped I/O* e *I/O mapped I/O* (EL-GHAZAWI, 2003).Quando se faz o uso de *memory mapped I/O*, uma parte de endereços da memória principal é utilizada para representar as portas de *I/O*, a porta de dados é utilizada para realizar o *buffer* de dados a serem transmitidos, já a porta de controle de *status* permite a CPU não só controlar o dispositivo de *I/O* como também ler em qual *status* o mesmo se encontra. Já, quando se utiliza o sistema *I/O mapped I/O* de endereçamento, existem dois espaços de endereçamento distintos para a memória principal e as portas de *I/O*. Uma porta de *I/O* pode ter o mesmo endereço como um local de memória, entretanto, a CPU usa linhas de controle separadas para *I/O*; desta forma, somente dispositivos de *I/O* são permitidos em operações de *I/O* (EL-GHAZAWI, 2003).

Assim, não importa se a CPU possui ou não *memory mapped I/O* ou *I/O mapped I/O*, basta endereçar os controladores dos dispositivos para poder trocar informações com eles. Dessa forma, outro esquema foi desenvolvido para atender a essa demanda: *Direct Memory Access* (DMA). Mas, para usá-lo, o sistema operacional precisa que o hardware possua o controlador de DMA. O mesmo possui acesso direto ao barramento do sistema independente da CPU. E, para que a CPU consiga controlá-lo, existem vários registradores que podem ser lidos e escritos. Esses são: registrador de endereçamento de memória, registrador contador de bytes e um ou mais registradores de controle. Sendo que os registradores de controle especificam a porta de *I/O* em uso, a direção da transferência (leitura ou escrita para o dispositivo), a unidade de transferência e o número de bytes a serem transferidos (TANENBAUM, 2003). Seu funcionamento é programado pela CPU inserindo dados em seus registradores. Após sua inicialização toda transferência é controlada pelo o mesmo, não dependendo mais da CPU. O DMA emite uma interrupção de sistema quando a operação de *I/O* é finalizada, logo a CPU saberá do término da mesma (TANENBAUM, 2003). A FIG. 5 ilustra um sistema que utiliza um controlador de DMA, os registradores de controle do DMA foram omitidos afim de simplificar. Basicamente, o DMA e a CPU se comunicam para sincronizar a utilização do barramento principal, e o controle do dispositivo de *I/O*, bem como a transferência dos dados, permanece a cargo do DMA.

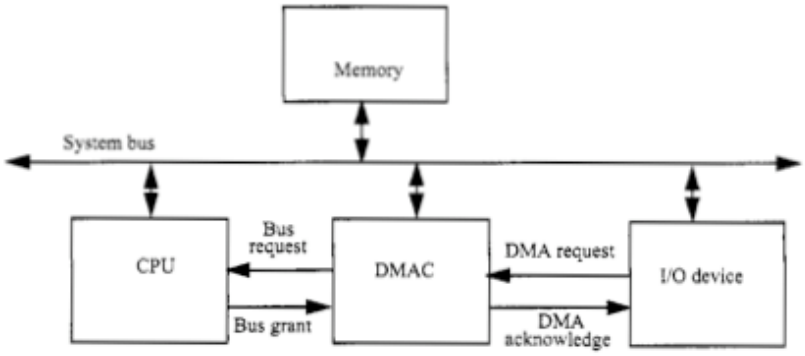

Figura 5 - Representação de um sistema de input/output utilizando DMA
Fonte: EL-Ghazawi (2003)

Assim, o DMA retira o peso da CPU de transferir os dados entre os dispositivos de *I/O* e a memória principal, mas ainda, continuam sendo programados pela CPU. Para minimizar esse problema, existem os co-processadores de *I/O*, que podem realizar instruções especializadas de *I/O*, além de algumas outras instruções padrões, como aritmética. Portanto, os co-processadores de *I/O* são processadores especializados que possuem a função de conduzir uma operação de *I/O* de forma eficiente (EL-GHAZAWI, 2003).

Contudo, existem dois tipos de chamadas de operações de *I/O* possíveis: *I/O* não bloqueante (ou assíncrono) e *I/O* bloqueante. Uma chamada de sistema bloqueante faz com que a execução da aplicação seja suspensa. Com isso, a aplicação é movida da fila de execução do sistema operacional para uma fila de espera. Somente, após o término da chamada de sistema é que a aplicação volta para a fila de pronto, onde é possível ser elegível para retomar a execução recebendo os valores solicitados pela chamada de sistema (SILBERSCHATZ *et al,* 2004*)*. Já, uma chamada de sistema não bloqueante não irá interromper a execução por um período de tempo indeterminado, ao invés disso, ela irá retornar rapidamente com um valor de retorno que indica quantos bytes foram transferidos. Ainda, uma alternativa a essa chamada não bloqueante é a utilização de chamadas assíncronas que retornam imediatamente. Por isso, a aplicação continuará executando seu código sendo que uma função de *callback* é executada para tratar o término da chamada assíncrona quando finalizada (SILBERSCHATZ *et al,* 2004*)*.

2.2 Desempenho

No desempenho de um sistema, operações de *input/output* são importantes, uma vez que impõem demanda pesada sobre a CPU para executar o código do *driver* do dispositivo e continuar escalonando os processos de forma eficiente enquanto bloqueiam e desbloqueiam. Conforme visto na Seção 1.1, ao bloquear e desbloquear um processo todo o seu estado precisa ser salvo para que, ao voltar à execução, o mesmo possa continuar do ponto onde foi interrompido. Ainda, quando um processo é bloqueado ou desbloqueado, é necessária que a CPU realize uma troca de contexto. Tais trocas de contexto geradas pressionam a CPU e seus *caches* de hardware e, com isso, diminuindo o desempenho do sistema como um todo (SILBERSCHATZ *et al,* 2004). Além disso, quando uma operação de *input/output* é executada, acontece a transferência de dados entre os controladores e a memória física, sobrecarregando o barramento da memória. Isso ocorrerá novamente durante as cópias entre os *buffers* do *kernel* e o espaço de dados da aplicação. Ao sobrecarregar o barramento com transferência de dados, acarretará uma ociosidade em algum outro subsistema que esteja precisando utilizá-lo (SILBERSCHATZ *et al,* 2004).

Entretanto, mesmo que os computadores atuais possam tratar milhares de interrupções por segundo, o tratamento continua sendo uma tarefa que demanda alto processamento. A cada interrupção gerada é preciso realizar uma mudança de estado, tratar a interrupção e ainda restaurar o estado (SILBERSCHATZ *et al,* 2004). Devido ao crescimento explosivo da internet, novos serviços estão surgindo e deixaram de ser *sites* que apresentam apenas

conteúdo estático, mas sim, abrangem uma grande variedade de categorias, incluindo negociação de ações, transmissões ao vivo, e-commerce, serviços de mensagem instantânea, download p2p e hospedagem de aplicações. Com, essa nova classe de serviços dinâmicos faz-se necessário um número significante de processamento e recursos de *input/output* para processar cada requisição em tempo viável para o usuário final. (WELSH, 2003).

É possível perceber que esses novos tipos de serviços apresentam páginas altamente personalizadas de acordo com cada usuário e, com a disseminação da web social, onde os usuários são responsáveis por gerarem conteúdos (GRUBER, 2008), as operações de *input/output* começam a ficar mais frequentes. A maneira mais utilizada de se armazenar toda essa informação gerada é através de um Sistema Gerenciador de Banco de Dados (SGDB) (KORTH *et al,* 2001). Basicamente, um SGDB consiste em uma coleção de programas para criar, manter e manipular um banco dados. Sendo que esses dados serão gravados em um dispositivo de armazenamento. Com esses novos serviços sendo oferecidos via *web,* o acesso aos SGBDs aumentou consideravelmente, pois toda informação é gravada ou lida nesses bancos de dados (KORTH *et al,* 2001). Consequentemente, ao aumentar o número de acesso a esses SGDBs também aumenta o número de operações de *input/output* realizadas.

Nesse contexto, o sistema *web* `Odrible` foi escolhido para ilustrar uma situação onde várias operações de *input/output* causaram demora para responder às requisições dos usuários. Por ser um portal relacionado a conteúdo de futebol, o número de acesso aumenta quando há campeonatos importantes acontecendo. No período da copa do mundo de 2010, os acessos ao portal cresceram exponencialmente. O portal utiliza um SGDB para armazenar os dados das partidas e dos usuários e, ao entrar no portal, o usuário visualizava os dados relacionados às partidas que aconteceram ou que ainda iriam acontecer. Dessa forma, várias operações de *I/O* eram realizadas a fim de buscar os dados necessários, e novos usuários não eram atendidos quando o sistema estava saturado de requisições a serem processadas.

2.3 Considerações Finais

As operações de *I/O* apresentam um alto custo computacional para todo o sistema. Elas exigem um esforço intenso por parte da CPU para executar o código do *driver* do dispositivo e continuar escalonando os processos de forma eficiente enquanto os processos bloqueiam e desbloqueiam.

Segundo Silberschatz (2004), existem dois tipos de chamadas de operações de *I/O* possíveis: *I/O* não bloqueante (ou assíncrono) e *I/O* bloqueante, sendo esta última a mais utilizada. As chamadas de I/O bloqueantes fazem com que a aplicação seja movida da fila de

execução do sistema operacional para um fila de espera. Assim, a aplicação só voltará a ficar elegível para retomar a execução quando a operação de *I/O* for completada. Esse tipo de alternativa em uma Arquitetura *Thread Based* faz com que as requisições de novos usuários não sejam atendidas quando todos os *threads* estiverem ocupados.

Para minimizar este problema, é possível utilizar chamadas de operações de *I/O* assíncronas que retornam imediatamente fazendo com que a aplicação possa continuar executando seu código enquanto a operação de *I/O* não é finalizada. E, assim que for finalizada, uma função de *callback* é executada.

Nesse contexto, ao realizar chamadas de operações de *I/O* assíncronas em uma Arquitetura *Evented Based* é possível continuar processando novas requisições. Em outras palavras, a aplicação continuará atendendo à novas requisições mesmo quando as operações de *I/O* estiverem sendo realizadas, pois são feitas de forma assíncrona. Diante do entendimento do custo de tais operações de *I/O* e a sua aplicabilidade em Arquiteturas *Evented I/O*, o próximo capítulo aborda em detalhes tal arquitetura, tais como seu contexto de utilização, diretrizes e padrões utilizados para desenvolvimento.

CAPÍTULO 3 – Arquitetura *Evented I/O*

Com o crescimento explosivo da Internet, novos serviços estão surgindo. Esses deixaram de ser *sites* que apresentam apenas conteúdo estático e passaram a abranger uma grande variedade de categorias, incluindo negociação de ações, transmissões ao vivo, e-commerce, serviços de mensagem instantânea, download p2p e hospedagem de aplicações. Assim, esses serviços oferecidos pela *web* atualmente demandam um número significativo de processamento e recursos de *input/output* para processar cada requisição (WELSH, 2002).

Nesse contexto, a grande utilização de operações de *input/output* apresenta um alto custo computacional para todo o sistema. De acordo com a Seção 2.2, a cada operação de *input/output* executada, uma troca de contexto é realizada, diminuindo o desempenho do sistema como um todo.

Contudo é preciso minimizar o tempo gasto com essas operações e diminuir o tempo de resposta para o usuário final. Dessa forma, a Arquitetura *Evented I/O*, que tem como característica a realização de *I/O* de forma não bloqueante, visa minimizar esse problema.

Este capítulo apresenta um detalhamento dessa arquitetura. Assim, na Seção 3.1 é apresentado seu contexto de utilização; na Seção 3.2 apontam-se as diretrizes para o desenvolvimento utilizando essa arquitetura inclusive descrevendo o padrão *Reactor*; na Seção 3.3 é apresentado um sistema de exemplo; por fim, na Seção 3.4 são apresentadas as considerações finais do capítulo.

3.1 Contexto de Utilização

Geralmente quando se tenta criar serviços que irão escalar para milhares de usuários, ou seja, atender a milhares de requisições existem dois fatores que influenciam no desempenho: computação e *I/O* (SEEGER *et al*, 2010).É conhecido que *I/O* são operações de entrada ou saída de dados em dispositivos conectados, os quais podem ser placa de rede, disco rígido ou qualquer outro que não esteja diretamente conectada à memória RAM ou à CPU.

Devido ao crescimento explosivo da Internet, novos serviços estão surgindo. Esses serviços deixaram de ser *sites* que apresentam apenas conteúdo estático e passaram a abranger uma grande variedade de categorias, incluindo negociação de ações, transmissões ao vivo, e-commerce, serviços de mensagem instantânea, download p2p e hospedagem de aplicações (WELSH, 2002). Contudo, esses serviços passaram a gerar uma alta demanda por armazenamento e apresentação de dados. Ainda, a maneira mais utilizada de se armazenar e disponibilizar toda essa informação gerada é através de um Sistema Gerenciador de Banco de

Dados (SGBD), que utiliza um dispositivo de armazenamento para persistência do mesmo (KORTH *et al,* 2001). Assim, consequentemente o número de operações de *input/output* necessária para atender a cada requisição do usuário irá aumentar.

As operações de *input/output* apresentam um alto custo computacional para todo o sistema. Essas operações, quando utilizadas de forma bloqueante, geram trocas de contexto que precisam ser executas pela CPU. Também pressionam a CPU e seus *caches* de hardware diminuindo o desempenho do sistema como um todo afirma Silberschatz (2004). Nesse contexto, aplicações que fazem grande uso de operações de *input/output* podem ter seu desempenho melhorado utilizando a Arquitetura *Evented I/O*, pois essas ficam em um *loop* eterno processando eventos de forma não bloqueante (WELSH, 2002). A FIG. 6 ilustra um sistema que faz grande uso de operações de *input/output*. Basicamente, existem vários usuários realizando requisições para o sistema e a cada requisição o sistema poderá ser solicitado a entregar algum arquivo que está localizado no disco, disponibilizar alguma informação que está gravada no SGBD ou, ainda, realizar a comunicação com um serviço externo. É importante mencionar que todas essas operações são consideradas operações de *input/output*.

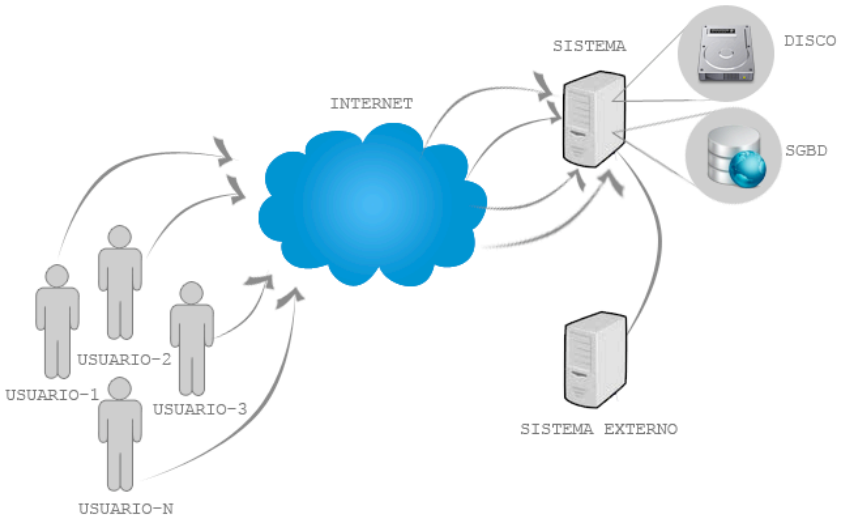

Figura 6 - Representação de um sistema que faz grande uso de operações de input/output
Fonte: Do autor.

Existem os sistemas que demandam grande quantidade de computação e realizam um número menor de operações de *input/output*. Computação, de acordo com a Seção 1.3, é a parte da aplicação que utiliza a maior quantidade disponível de CPU para realizar tarefas como: compressão de imagens, tradução de textos, ordenação de dados, cálculos matemáticos, etc. Dessa forma, o *loop* eterno de eventos ficaria bloqueado realizando essas operações (SEEGER *et al*, 2010). Como ao bloquear o *loop* eterno de eventos, os benefícios

proporcionados pela Arquitetura *Evented I/O* deixam de existir, não se recomenda a utilização dessa arquitetura para aplicações que demandam grande quantidade de computação.

3.2 Diretrizes para o desenvolvimento de arquiteturas baseadas em *evented I/O*

A Arquitetura *Evented I/O* é uma arquitetura *Evented Based*, ou seja, seu principio de funcionamento tem como característica um *loop* eterno que fica processando eventos. Basicamente, Arquiteturas *Evented Based* são organizadas em volta do processamento de eventos. Assim, quando uma operação não pode ser completada imediatamente por ter que esperar por algum evento (ex: a chegada de algum pacote pela interface de rede ou o término de uma operação de transferência de dados do disco) uma função de *callback* é vinculada a mesma, que será executada assim que a operação for completada (DABEK *et al*, 2002).

Assim, a Arquitetura *Evented I/O* faz o uso de interfaces de *I/O* não bloqueantes que operam de forma dividida. Ao realizar uma operação de *I/O*, a mesma irá retornar imediatamente, evitando o bloqueio e, quando a operação for finalizada, a função de *callback* vinculada é executada (WELSH, 2002). Portanto, para que seja possível executar operações de *I/O* de forma não bloqueante o Sistema Operacional precisa ter suporte a essas interfaces.

Nesse contexto, uma chamada de sistema operacional, Unix/POSIX, bastante utilizada para esse tipo de operação é `select()`[4] (WELSH, 2002). Essa chamada de sistema permite monitorar múltiplos descritores de arquivo[5], esperando até que um deles fique pronto para alguma operação de *I/O*. Já no sistema operacional Windows é possível usar *I/O Completion Ports* para atingir esse mesmo resultado.

Dessa forma, a FIG. 7 demonstra baseado no principio de funcionamento da Arquitetura *Evented I/O*, o *loop* eterno que fica processando eventos. De forma detalhada, primeiro registram-se as funções de *callback* desejadas para as operações de escrita e leitura. O *loop* eterno começa a processar e, quando algum evento ocorrer, a função de *callback* vinculada a esse evento é executada e após o seu termino o *loop* principal volta a sua execução. Assim, o código executado pelas funções de *callback* deve ser curto para evitar

[4]select() : é uma chamada de sistema operacional e uma API (Application Programming Interface) para Unix/POSIX que permite examinar o status de descritores de arquivos em canais abertos de input/output. Disponível em: <http://en.wikipedia.org/wiki/Select_(Unix)>. (Acesso em: 09 nov. 2011)

[5]Descritores de Arquivo: é um índice para uma estrutura de dados que contém os detalhes de todos os arquivos abertos. Descritor de arquivo pode também se referir aos dispositivos de memória, soquetes, etc. Disponível em: <http://pt.wikipedia.org/wiki/Descritor_de_arquivo>. (Acesso em: 09 nov. 2011)

bloquear o *loop* principal e garantir que todas as requisições sejam processadas de forma justa (WELSH, 2002).

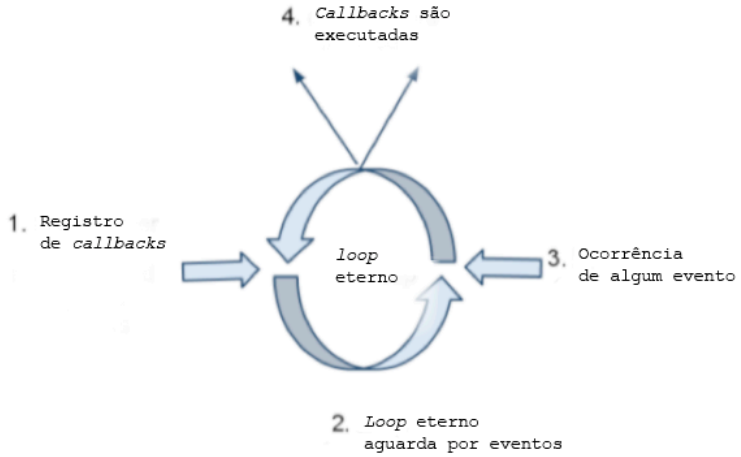

Figura 7 - Demonstração do loop eterno
Fonte: SEEGER *et al*, 2010

Segundo Seeger *et al* (2010), é importante salientar as seguintes diretrizes a serem seguidas ao desenvolver sistemas que farão o uso da Arquitetura *Evented I/O*.

1 – evitar *loops* que executam sobre uma grande coleção de objetos;

2 – evitar operações que demandam grande quantidade de processamento;

3 – evitar executar operações de input/output de forma síncrona;

4 – evitar executar chamadas a *sleep*/pause.

Resumidamente, todas as operações que fazem grande uso de processamento devem ser evitadas, pois existe a possibilidade do *loop* ser bloqueado durante sua execução devido a sua alta demanda de processamento.

Nesse contexto, utiliza-se o padrão *Reactor* para implementar o *loop* eterno que ficará processando os eventos. Esse padrão é explicado na seção seguinte, mas basicamente ele facilita o desenvolvimento de sistemas que utilizam a Arquitetura *Evented Based*, pois ajuda a desacoplar mecanismos independentes do sistema de funcionalidades especificas do mesmo (SCHMIDT, 1994).

3.2.1 Padrão *Reactor*

O padrão *Reactor* é útil para gerenciar um *loop* eterno que fica processando eventos e executando funções de *callback* vinculadas aos mesmos. Assim, esse padrão facilita a

construção de sistemas baseados em Arquiteturas *Evented Based*, pois ajuda a desacoplar as funcionalidades do sistema com seus mecanismos independentes (SCHMIDT, 1994).

Nesse contexto, ao se utilizar o padrão *Reactor*, os desenvolvedores podem se concentrar em funcionalidades especificas do sistema ao invés de funções de baixo nível que irão ficar processando eventos e disparando chamadas para as funções de *callback* vinculadas (SCHMIDT, 1994).

Assim, a FIG. 8 demonstra a organização da estrutura de classes em uma arquitetura que faz a utilização do padrão *Reactor*. Basicamente, segundo Schmidt (1994), o trabalho de cada classe é:

`Reactor:` define uma interface para registrar, remover e executar objetos `EventHandler`. A implementação dessa interface fornece uma série de mecanismos independentes da aplicação. Esses mecanismos são: controlar eventos e executar as funções de *callback* vinculadas.

`EventHandler`: especifica uma interface para despachar métodos de *callback* definidos por objetos que são registrados como *handle* de eventos.

`ConcreteEventHandle`: Implementa os métodos de *callback* que processam eventos de uma maneira especifica na aplicação.

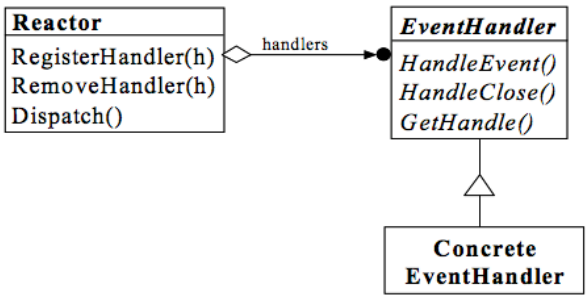

Figura 8 - Estrutura de classes do padrão *Reactor*
Fonte: SCHMIDT (1994)

Nesse contexto, os *handlers* de eventos são utilizados para tratar eventos quando necessário e são associados com os *handlers* do sistema operacional, pois estão associados a fonte de onde um evento pode ocorrer (por exemplo, portas de *I/O* e soquetes). Tais *handlers* precisam ser vinculados ao objeto `Reactor` para que o mesmo possa utilizá-los quando algum evento ocorrer. Deste modo, o método `GetHandler` é utilizado pelo objeto `Reactor` para obter o *handler*. Quando algum evento vinculado ao *handler* ocorrer, o objeto `Reactor` irá utilizar o *handler* obtido através do método `GetHandler` como uma forma de localizar e despachar para a função de *callback* apropriada segundo Schmidt (1994).

3.3 Sistema de Exemplo

Para exemplificar as diretrizes citadas na Seção 3.2 foi elaborado um sistema de exemplo conforme ilustrado na FIG. 9. Esse sistema tem como característica a escrita e leitura de um valor em um banco de dados não relacional chamado Redis[6]. Basicamente, o usuário acessa um URI para registrar um valor no Redis e obtém uma resposta de confirmação após 10 segundos, e para ler esse valor, uma URI diferente é ser acessada e o valor é mostrado na tela.

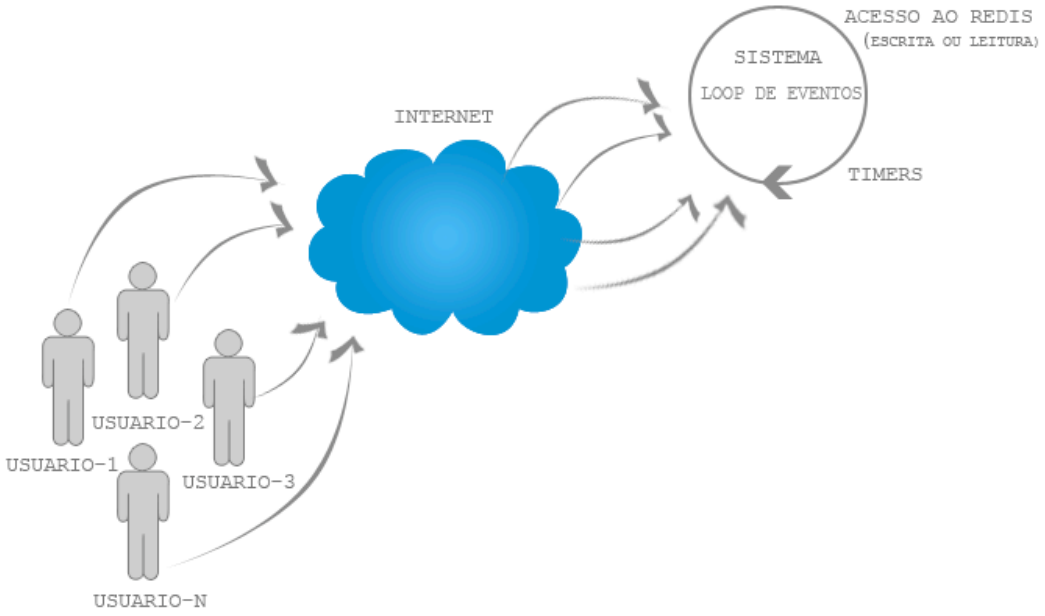

Figura 9 - Demonstração de um sistema que utiliza a Arquitetura *Evented I/O*
Fonte: Do autor.

Para abstrair a complexidade das funcionalidades que são necessárias que os sistemas operacionais implementem (interfaces assíncronas) e manter o foco no principio de funcionamento, foi utilizado um *framework* baseado em *evented I/O*. Esse *framework*, EventMachine[7], é escrito em Ruby e implementa o padrão *Reactor* para processar eventos. Além disso, foi utilizada uma biblioteca que utiliza interfaces assíncrona, escrita em Ruby, para escrita e leitura assíncrona no *Redis*. Para construir o sistema para *web*, foi utilizado o

[6]Redis: Redis é um projeto open source que armazena dados na estrutura chave – valor. Disponível em:<http://redis.io/topics/introduction>. Acesso em 12 nov. 2011

[7]EventMachine: É biblioteca para programadores Ruby, C++ e java. Permite a utilização de operações de input/output baseado em eventos usando o padrão Reactor. Disponível em: <https://github.com/eventmachine/eventmachine/wiki>. Acesso em: 12 nov. 2011.

framework Sinatra[8]. Para atender a essas requisições, foi utilizado um servidor que funciona de maneira assíncrona, processando eventos. Esse servidor, chamado Thin[9], também é escrito em Ruby.

A FIG. 10 demonstra o diagrama de classe do sistema. Basicamente existem 3 classes, `MyApp`, `Sinatra::Base` e `EventMachine::Hiredis::Client`.

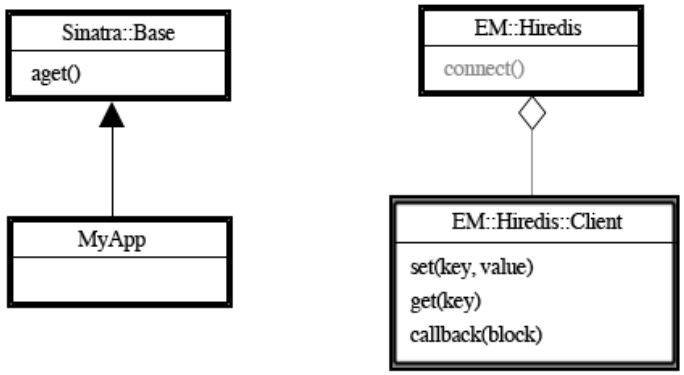

Figura 10 - Diagrama de classes do sistema de exemplo
Fonte: Do autor.

A classe `Sinatra::Base` faz parte do *framework* Sinatra. Essa classe implementa o método `aget`, através desse método é possível definir URI que estará disponível para acesso via HTTP. A classe `MyApp` herda de `Sinatra::Base`. Assim, duas chamadas ao método `aget` é realizada dentro da classe `MyApp`. Essas chamadas definem as duas URI possíveis de serem acessadas: /set e /get. É passado um bloco para as chamadas ao método `aget` dentro da classe `MyApp`. O código dentro desse bloco será executado quando o usuário acessar alguma das duas URI possíveis.

Assim, de acordo com a FIG. 11, a primeira chamada ao método `aget` é para definir a URI /set.

[8]Sinatra: É uma Domain Specific Language (DSL) para criar aplicações web em Ruby com o mínimo de esforço. Disponível em: < http://www.sinatrarb.com/intro>. Acesso em: 12 nov. 2011.
[9]Thin: É um servidor web escrito em ruby que utiliza a biblioteca EventMachine para realizar I/O baseado em eventos. Disponível em: <http://code.macournoyer.com/thin/>. Acesso em: 12 nov. 2011.

```
27  aget '/set' do
28    retorno = ""
29    EM.add_timer(10) do
30      env['async.callback'].call [
31        200,
32        {'Content-Type' => 'text/html'},
33        ["desculpe por fazer voce esperar mas o retorno foi: #{retorno}"]
34      ]
35    end
36
37    redis = EM::Hiredis.connect
38    redis.set('foo', 'bar').callback do
39      retorno = "setado no redis!"
40    end
41  end
```

Figura 11 - Demostração de código
Fonte: Do autor.

O código, que está dentro do bloco passado como parâmetro, realiza uma chamada ao método `add_timer` (linha 29 da FIG. 11) da classe `EM` passando como argumento o tempo em segundos e um bloco. Esse bloco será utilizado como uma função de *callback* para ser executada quando o tempo solicitado tiver sido atingido. Ainda, é feita uma chamada ao método `connect` da classe `EM:Hiredis` (linha 33 da FIG. 11). Esse método retorna uma instância da classe `EM::Hiredis::Client`. Logo após é feita uma chamada ao método `set` (linha 34 da FIG. 11), na instancia da classe `EM::Hiredis::Client`, e uma chamada encadeada ao método `callback` para registrar uma função de *callback* para ser chamada quando o método `set` acabar sua operação. Analogamente, retirando a definição do *timer* e trocando a chamada do método `set` para `get` para realizar a leitura assíncrona de dados no Redis, os mesmos passos são realizados para a segunda chamada ao método `aget` para definir a URI /get. Resumidamente, ao acessar a URI /set o usuário irá aguardar 10 segundos e será armazenado um valor no Redis. O *timer* executa de forma não bloqueante, pois o valor da variável retorno é setado em uma operação que acontece após a chamada de definição do timer. Já ao acessar a URI /get será executada um operação de leitura assíncrona ao Redis e, o valor lido, será apresentado ao usuário.

Nesse contexto, o *loop* eterno de eventos não é bloqueado, pois foram seguidas todas as diretrizes citadas na Seção 3.2. Não acontece interações sobre uma grande coleção de objetos, a operação executada não demanda grande poder de processamento e as chamadas de *input/output* são executadas de forma assíncrona, além das funções de *callback* vinculadas executam operações rápidas para evitar o bloqueio do *loop* eterno. Ainda, um *timer* não

bloqueante foi adicionado para exemplificar a utilização dessa funcionalidade em uma Arquitetura *Evented I/O*.

3.4 Considerações Finais

Ao analisar qual operação é mais executada em um sistema e concluir que são as operações de *input/output*, é interessante conhecer a Arquitetura *Evented I/O*. A mesma, por utilizar de interfaces não bloqueantes, faz com que o desempenho de sistemas que realizam grande quantidade de operações de *input/output* fique mais eficiente. Ainda, por utilizar um *loop* eterno que fica processando eventos, o *overhead* na criação de novos *threads* e a troca de contexto constante é evitado.

Nesse contexto, o padrão *Reactor* é utilizado para implementar e controlar o *loop* eterno de eventos. Esse padrão facilita a construção de sistemas baseados em Arquiteturas *Evented Based*, pois ajuda a desacoplar as funcionalidades do sistema com seus mecanismos independentes.

Assim, um sistema de exemplo é construído para demostrar os reais ganhos atingidos utilizando a Arquitetura *Evented I/O*. Esse sistema, abordado no próximo capítulo, tem como característica realizar operações de *input/output*.

CAPÍTULO 4 – ESTUDO DE CASO

Para ilustrar os benefícios proporcionados pela Arquitetura *Evented I/O* em um sistema *web*, são contrastada duas versões da aplicação MyIO. Na primeira versão utilizou-se a arquitetura *Evented I/O* e na segunda a arquitetura *Thread Based*. Isso indica que ambos os sistemas contêm a mesma funcionalidade, contudo com diferentes arquiteturas.

Na arquitetura *Evented I/O* todo o sistema realiza tarefas de forma assíncrona e as funções de *callback* vinculadas aos eventos são curtas. Dessa forma, ao realizar alguma operação de *input/output* a mesma irá retornar imediatamente e, ao terminar, a função de *callback* vinculada será executada. A operação realizada pela função de *callback* não bloqueia o *loop* eterno, pois ela executa apenas operações simples. Ainda, o sistema segue todas as diretrizes citadas na Seção 3.2.

Já na arquitetura *Thread Based,* o sistema utiliza interfaces bloqueantes para realizar operações de *input/output*. Assim, ao realizar alguma operação de *input/output*, o sistema terá que aguardar seu término para poder continuar processando a requisição do usuário. Ao bloquear, a única saída do sistema para atender a novas requisições é iniciar um novo *thread*. Dessa forma, esse tipo de arquitetura usualmente não consegue atender a muitas requisições simultâneas uma vez que iniciar novos *threads*, de acordo com a Seção 1.1, é uma operação que demanda custo computacional.

O restante deste capítulo está organizado como a seguir: na Seção 4.1 é apresentado o sistema MyIO; a Seção 4.2 apresenta a análise de desempenho entre as duas versões do sistema MyIO descrevendo a metodologia utilizada, os resultados obtidos e os riscos à validade do estudo; por fim, a Seção 4.3 apresenta as considerações finais.

4.1 Sistema MyIO

O sistema MyIO realiza a leitura e busca sequencial de valores em um arquivo localizado no disco. Basicamente, o usuário acessa uma URI para que a busca seja realizada e seja retornado um resultado. O arquivo pesquisado contém 4 KB de conteúdo, a busca realizada é a procura de uma letra que se encontra na última posição desse arquivo. Dessa forma, o sistema realiza operações de *input/output* em um arquivo de dados assim como a maioria dos sistemas *web* atuais. A linguagem Ruby foi utilizada para o desenvolvimento.

A primeira versão do sistema MyIO, cujo o código fonte pode ser encontrado no Anexo I, desenvolvido utilizando a arquitetura *Evented I/O* realiza todas as operações de forma assíncrona. O servidor web *Thin*, utilizado neste estudo, utiliza a biblioteca *EventMachine* para atender as requisições dos usuários de forma assíncrona e para realizar operações de *input/output* de forma não bloqueante. O método `aget`, utilizado para definir a URI de acesso, trabalha de forma assíncrona. Basicamente, o usuário acessa a URI /perf definida pelo método `aget` para realizar a busca sequencial no arquivo de dados. A busca é feita de forma não bloqueante, todo o arquivo é processado de forma assíncrona utilizando funções de *callback* vinculadas ao *loop* de eventos. Nesse contexto, primeiro é definida a função de *callback* que será executada quando a próxima iteração do *loop* for realizada (linha 11 do Anexo I) e, em seguida, essa função é vinculada a esse evento (linha 25 do Anexo I). A leitura do arquivo está sendo realizada dentro da função de *callback* associada ao *loop* principal de eventos. A biblioteca *EventMachine* implementa o padrão *Reactor* para controlar o *loop* eterno que ficará processando eventos. Dessa forma, tudo acontece dentro do *loop* eterno de forma assíncrona com o processamento realizado por *callbacks* vinculadas aos eventos. Ainda, não é preciso se preocupar com mecanismos independentes do sistema, pois de acordo com a Seção 3.2.1, o padrão *Reactor* ajuda a desacoplar esses mecanismo das funcionalidades do sistema.

A segunda versão do sistema MyIO, cujo o código fonte pode ser encontrado no Anexo II, foi desenvolvida utilizando a arquitetura *Thread Based*. Para atender as requisições dos usuários foi utilizado o servidor *web Mongrel*. As interfaces utilizadas pelo *web server* não são assíncronas e todo o processamento das requisições são atendidas por vários *threads*. A leitura do arquivo foi feita de forma tradicional, ou seja, bloqueante. Basicamente, é feita a leitura do arquivo (linha 08 do Anexo II) e logo em seguida faz-se uma busca sequencial em cada linha do arquivo (linha 10 do Anexo II) até o valor procurado ser encontrado. O mesmo arquivo e o valor procurado utilizados na primeira versão do sistema MyIO também foram utilizados nessa versão. A versão utilizada do *framework Sinatra* também é implementado de forma síncrona.

Dessa forma, o Quadro 2 apresenta algumas informações técnicas relacionadas ao sistema MyIO nas duas versões. É demonstrada a quantidade de linhas de código, número de classes, bibliotecas externas utilizadas bem como o tempo de desenvolvimento gasto em cada versão.

	MyIO*EventedBased*	MyIO*ThreadBased*
LOC	**33**	**21**
Número de classes	**1**	**1**
Bibliotecas externas	**2**	**1**
Tempo de desenvolvimento	**4 horas**	**2 horas**

Quadro2– Sistema *MyIO*
Fonte: Do autor

Não é gerada muita diferença no número de classes, de linhas de código, bibliotecas externas e no tempo de desenvolvimento. A real intenção deste estudo é demonstrar a melhoria no desempenho.

4.2 Análise do desempenho

A fim de demonstrar o real ganho proporcionado pela Arquitetura *Evented I/O* em um sistema que tem como principal característica a utilização de operações de *input/output* foram realizados testes comparando as duas arquiteturas.

Diante disso, na Seção 4.2.1 é apresentada a metodologia utilizada para realizar esse comparativo; na Seção 4.2.2 são apresentados os resultados obtidos; por fim, na Seção 4.2.3 são apresentados os riscos de validade do estudo.

4.2.1 Metodologia

Para realizar a análise de desempenho as seguintes atividades foram executadas:

1 – A aplicação foi implantada em um servidor com processador XEON de 04 processadores com 512 MB de RAM e com 20 GB de espaço em disco, sob o sistema operacional Ubuntu 10.04.3 LTS (*Long Term Support*). É importante mencionar que somente os serviços indispensáveis ao teste de desempenho estavam ativados e somente as bibliotecas estritamente necessárias foram instaladas.

2 – Para realizar os testes de desempenho da aplicação desenvolvida foi utilizada a ferramenta Siege[10]. Essa ferramenta é executada via linha de comando e permite

[10]Siege: Ferramenta utilizada para realizar testes de desempenho em sistemas web. Disponível em:

analisar o desempenho das requisições realizadas. Os argumentos aceitos permitem definir a quantidade de requisições que serão disparadas para o servidor e dessas requisições quantas serão realizadas de forma concorrente. Diante disso, utilizou-se o parâmetro –c100 para disparar 100 requisições simultâneas ao servidor e o parâmetro –t1M para limitar o tempo do teste em 1 minuto. O mesmo servidor utilizado para o sistema utilizando a arquitetura *Evented I/O* também foi utilizado para a Arquitetura *Thread Based*.

3 – Em busca de maior validade estatística, os testes foram replicados 31 vezes. Por isso, para cada um dos atributos testados foi calculado a média aritmética com um intervalo de confiança de 99%.

4.2.2 Resultados

Após a realização dos experimentos, com uma confiança de 99% o Quadro 03 apresenta os resultados obtidos nas duas arquiteturas. É importante mencionar que os valores apresentados são as médias aritméticas seguida do grau de confiança.

	MyIO *Evented Based*	MyIO *Thread Based*
Tempo de resposta (s)	**0.03** ± 0.05	**0.14**± 0.01
Transações por segundo	**186.85**± 0.01	**150.65**± 0.05
Número de transações efetivadas	**11.152,32**± 0.01	**9.314,61**± 0.05

Quadro3 - Demonstração de resultados comparativos
Fonte: Do autor

Por utilizar de interfaces não bloqueantes a arquitetura *Evented I/O* apresentou um tempo de resposta quase 5 vezes menor do que a arquitetura *Thread Based*. Na arquitetura *Thread Based*, de acordo com a Seção 1.1, ao aumentar o número de *threads* aumenta também o *overhead* na troca de contexto e consequentemente o desempenho do sistema cai. Já na Arquitetura *Evented I/O*, de acordo com a Seção 3.2, uma das diretrizes que se deve seguir é não bloquear o *loop* principal de eventos para que o mesmo possa atender a maior quantidade de eventos possíveis. Ainda, de acordo com a Seção 1.2, por utilizar poucos

<http://www.joedog.org/index/siege-manual#a02>. Acesso em: 12 nov. 2011.

threads a performance do sistema não sofre com as trocas de contexto, consequentemente um maior número de transações é atendido.

Assim, o número de transações por segundo atendido foi 24% maior na Arquitetura *Evented I/O* por essa arquitetura utilizar poucos *threads* e não sofrer com a troca de contexto do sistema operacional. Ainda, conforme citado na Seção 1.3, operações de *input/output* demandam um custo considerável do sistema computacional, ao esperar essas operações finalizarem a quantidade de transações realizadas por segundo diminui. Ao realizar toda o trabalho de forma assíncrona, a Arquitetura *Evented I/O* foi possível de executar um maior número de transações por segundo.

Consequentemente, ao realizar um maior número de transações por segundo e apresentar um menor tempo de resposta, a Arquitetura *Evented I/O* também foi possível de executar aproximadamente 24% mais transações que na Arquitetura *Thread Based*. Já a arquitetura *Thread Based* sofreu com o *overhead* na troca de contexto e com as operações de *input/output* sendo executadas de forma síncrona. De acordo com a Seção 2.2, operações de *input/output* levam um tempo computacional considerável para serem realizadas, na arquitetura *Thread Based* todo esse tempo gasto não é aproveitado para processar novas requisições. Assim, o número total de transações efetivadas é menor.

4.2.3 Riscos a validade do estudo

As versões das bibliotecas utilizadas pelo sistema operacional Ubuntu foram padrões de instalação, mas poderiam ser utilizadas bibliotecas específicas que apresentariam um melhor desempenho. Além disso, não foram ajustadas variáveis de *tunning* no Ruby a fim de não privilegiar alguma arquitetura. Por fim, o tempo de desenvolvimento da versão *Thread Based* foi menor em função da versão *Evented Based* ter sido desenvolvida *a priori* e com isso houve um grande reaproveitamento de código fonte.

4.3 Considerações Finais

Ao identificar que as operações mais utilizadas pelo sistema são operações de *input/output*, é valido considerar a utilização da Arquitetura *Evented I/O*. Conforme apresentado na Seção 4.2.2, o desempenho dessa arquitetura é superior ao desempenho da Arquitetura *Thread Based*.

Por utilizar de interfaces não bloqueantes, todas as operações de *input/output* realizada são feitas de forma assíncrona. Assim, conforme citado na Seção 2.2, operações de *input/output* demandam um tempo computacional para serem completadas. Um ponto importante é que na Arquitetura *Evented I/O*, esse tempo despendido para realizar operações de *input/output* é compensado por meio da execução de outras tarefas.

Por funcionar através de um *loop* eterno de eventos que utiliza poucos *threads*, o *overhead* na troca de contexto é diminuído fazendo com que o desempenho de todo o sistema seja melhor, quando muitas operações de *input/output* são realizadas, comparado com a Arquitetura *Thread Based*.

Assim, ao utilizar sistemas que fazem grande uso de operações de *input/output* em que o tempo de resposta, quantidade de requisições processadas por segundo e número total de transações seja importante, a utilização da Arquitetura *Evented I/O* é fortemente indicada.

CONCLUSÃO

Os serviços oferecidos via Internet deixaram de apresentar apenas páginas estáticas e passaram a oferecer páginas cada vez mais customizadas e dinâmicas. De acordo com a Seção 3.1, esses serviços começaram a abranger uma grande variedade de categorias que demandam cada vez mais armazenamento e disponibilidade de informações.

Assim, o tempo de resposta e a capacidade de atender a todos os clientes passaram a ser características indispensáveis para esses novos tipos de sistema. Segundo a Seção 3.1, toda a informação armazenada e disponibilizada por esses serviços estão armazenadas em um SGBD que utiliza como meio de armazenamento o disco rígido. Ainda, segundo a Seção 2.2, operações de *input/output* demandam um alto custo computacional para serem realizadas. Dessa forma, ao se identificar que operações de *input/output* são bastante usada pelo sistema, precisa-se pensar em uma arquitetura que minimize os efeitos gerados por tais operações.

Como um primeiro passo dessa monografia foi contrastada a Arquitetura *Thread Based* e a Arquitetura *Evented Based* a fim de descrevê-las e compará-las. Em resumo, a Arquitetura *Thread Based* tem como característica a criação de um novo *thread* a cada requisição. Por se comportar dessa maneira, segundo a Seção 1.1, tal arquitetura apresenta um grande *overhead* na troca de contexto e na utilização de memória. Em contrapartida, a Arquitetura *Evented Based*, minimiza o *overhead* tanto nas operações de troca de contexto quanto na utilização de memória. Isso é possível porque a Arquitetura *Evented Based* utiliza um *loop* eterno, com poucos *threads*, que fica processando eventos.

Ainda, a maneira de se programar utilizando a arquitetura *Evented Based* é um pouco diferente, pois o fluxo não é mais continuo, i.e. assíncrono. Desse modo, o processamento ocorre através de funções de *callback* que são vinculadas a eventos e, quando esses eventos ocorrem, tais funções são executadas.

Como um segundo passo dessa monografia buscou-se conceituar operações de *I/O* bem como demonstrar o desempenho de tais operações. Dessa forma, segundo a Seção 2.1, operações de *input/output* são operações que transferem informações entre a memória principal do computador e o mundo externo. Ainda, segundo a Seção 2.2, esse tipo de operação demanda um grande custo computacional para ser realizada uma vez que demanda um alto processamento pela CPU para executar o código do *driver* do dispositivo e continuar escalonando os processos de forma eficiente enquanto bloqueiam e desbloqueiam.

Ainda nesse contexto, existem dois tipos de operações de *I/O*: *I/O* não bloqueante (ou assíncrono) e *I/O* bloqueante, sendo essa última a mais utilizada. Esse último tipo de chamada faz com que a aplicação tenha que aguardar o término da operação de *I/O* para poder voltar a executar. Para minimizar essa espera é possível utilizar operações de *I/O* assíncronas que retornam imediatamente. Uma função de *callback* é vinculada a operação de *I/O* e essa função será executada quando a operação for finalizada. Assim, em uma Arquitetura *Evented Based* é possível utilizar chamadas de *I/O* assíncronas e continuar processando novas requisições sem precisar esperar que a operação termine.

Após o entendimento dos custos de uma operação de *I/O* e a aplicabilidade de operações de *I/O* assíncronas em uma Arquitetura *Evented Based*, o terceiro passo dessa monografia busca apresentar a Arquitetura *Evented I/O*. Assim, descreveu-se o contexto de utilização, as diretrizes para desenvolvimento, o padrão *Reactor* e um sistema de exemplo. Em suma, a ideia geral consiste no fato de que quando operações de *input/output* são as operações mais utilizadas em um sistema é interessante considerar o uso da Arquitetura *Evented I/O*. Essa arquitetura, por basear seu processamento em eventos, utiliza interfaces não bloqueantes. Isso indica que em sistemas que fazem grande uso de operações de *I/O* o desempenho fica mais eficiente, pois essas operações retornam imediatamente. Ademais, é possível continuar realizando outras operações, pois assim que a operação de *I/O* finalizar, a função de *callback* vinculada será executada.

Ainda, essa arquitetura utiliza o padrão *Reactor* para implementar e controlar o *loop* eterno de eventos. Esse padrão facilita a construção dos sistemas, pois ajuda a desacoplar as funcionalidades do sistema com seus mecanismos independentes. No entanto, ao construir sistemas utilizando tal arquitetura é necessário seguir as diretrizes abordadas na Seção 3.2, tais como: evitar *loops* que executam sobre uma grande coleção de objetos, evitar operações que demandam grande quantidade de processamento, evitar executar operações de *input/output* de forma síncrona e evitar executar chamadas a *sleep*/pause. Enfim, ao seguir tais diretrizes é possível atingir o desempenho proporcionado pela Arquitetura *Evented I/O*.

No intuito de demonstrar os reais ganhos proporcionados pela Arquitetura *Evented I/O*, o quarto e último passo desta monografia foi apresentar um estudo de caso em que é construído um mesmo sistema, porém utilizando a Arquitetura *Evented I/O* e a Arquitetura *Thread Based*. Resultados demonstraram que a Arquitetura *Evented I/O* em sistemas em que as operações de *input/output* são bastante utilizadas, é consideravelmente melhor que a Arquitetura *Thread Based*. Isso é explicado pelo fato da Arquitetura *Evented I/O* utilizar interfaces não bloqueantes e utilizar um *loop* eterno de eventos que utiliza poucos *threads*.

Por utilizar interfaces não bloqueantes, a Arquitetura *Evented I/O* consegue processar novas requisições enquanto as operações de *input/output* estão sendo executadas. Ainda, por utilizar um *loop* eterno de eventos com poucos *threads*, o *overhead* na troca de contexto é evitado e a utilização de memória fica mais baixa. Assim, o número de requisições atendidas e a quantidade de transações executadas por segundo é cerca de 24% maior na Arquitetura *Evented I/O*. Além disso, por utilizar interfaces não bloqueantes, a arquitetura *Evented I/O* apresenta um tempo de resposta 5 vezes mais baixo do que a Arquitetura *Thread Based*.

Por fim, a partir de todo o estudo desenvolvido nesta monografia, pode-se afirmar que sistemas que fazem grande uso de operações de *input/output* que desejam eficiência – curto tempo de resposta a uma requisição – e ainda atendimento a uma grande quantidade de requisições simultâneas, devem utilizar a Arquitetura *Evented I/O*.

REFERÊNCIAS

BOVET, Daniel P.; CESATI, Marco. **Understanding the Linux Kernel**. 3. ed. Sebastopol: O'Reilly, 2005.

DABEK, Frank; ZELDOVICH, Nickolai; KAASHOEK, David M.; MORRIS, Robert. **Event-driven programming for robust software**. Proceedings of the 10th workshop on ACM SIGOPS European workshop: beyond the PC**,** Saint-Emilion, 2002.

EL-GHAZAWI, Tarek; FRIEDER, Gideon. **Input-output operations**. Encyclopedia of Computer Science**.** 4. ed. John Wiley and Sons Ltd. Chichester 2003.

GRUBER, Tom. **Collective knowledge systems**: Where the Social Web meets the Semantic Web, Web Semantics: Science, Services and Agents on the World Wide Web, Volume 6, Fevereiro 2008.

JAMIL, G. L. ; BAX, M. P. Serviços WEB e a Evolução dos Serviços da Internet. **Datagramazero** (Rio de Janeiro), Belo Horizonte, v. 1, n. 1, p. 1-1, 2001

KORTH, Henry F.; SILBERSCHATZ, Abraham. **Database system concepts**. 4 ed. Nova Iorque: McGraw-Hill, 2001.

LI, Chuanpeng; DING, Chen; SHEN, Kay. **Quantifying The Cost of Context Switch**. Proceedings of the 2007 workshop on Experimental computer science**,** San Diego, California, 2007.

NÚÑEZ, Angel; NOYÉ, Jacques; GASIUNAS, Vaidas. **Declarative definition of contexts with polymorphic events**. International Workshop on Context-Oriented Programming**,** Genova, 2009.

SEEGER, Marc; HdM, Stuttgart. **Event-Driven I/O A hands-on introduction**. 2010. Disponível em: <http://blog.marc-seeger.de/assets/papers/seeger-aysnc_io.pdf>. Acesso em: 24 nov. 2011.

SILBERSCHATZ, Abraham; GALVIN, Peter B.; GAGNE, Greg. **Sistemas operacionais:** com java. Rio de Janeiro: Elsevier, 2004.

TANENBAUM, Andrew S. **Sistemas operacionais modernos.** 2. ed. São Paulo: Prentice Hall, 2003.

SCMMIDT, Douglas C, 1995. **Reactor An Object Behavioral Pattern for Concurrent Event Demultiplexing and Dispatching**. Disponivel em: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.9570>. Acesso em: 01 de set. 2011.

TERRA, Ricardo. **Conformação Arquitetural utilizando Restrições de Dependência entre**

Módulos. 2009. 73f. Dissertação (Pós-Graduação em Ciência da Computação) - Curso de Pós-graduação em Ciência da Computação, Pontifícia Universidade Católica de Minas Gerais, Belo Horizonte.

WELSH, Matthew David. **An Architecture for Highly Concurrent, Well-Conditioned Internet Services**. 2002. 175 f. Tese Doutorado em Ciência da Computação - Universidade da Califórnia em Berkeley, California. 2002

ANEXOS

Anexo I – MyIO *Evented Based*

```
1 require 'rubygems'
2 require 'sinatra/async'
3 require 'eventmachine'
4
5 class MyApp< Sinatra::Base
6       register Sinatra::Async
7
8       set :server, 'thin'
9
10 aget '/perf' do
11leitura = EM.Callback do |value|
12     leitura_feita = IO.read("testfile", 1, value)
13      ifleitura_feita == 'c'
14          env['async.callback'].call [
15              200,
16              {'Content-Type' => 'text/html'},
17              ["retorno"]
18            ]
19      else
20        EM.next_tickdo
21          leitura.call(value+2)
22        end
23      end
24    end
25    EM.next_tickdo
26      leitura.call(0)
27    end
28
29  end
30
31 end
32
33 MyApp.run!
```

Anexo II- MyIO *Thread Based*

```
1 require 'rubygems'
2 require 'sinatra'
3
4 class MyApp< Sinatra::Base
5
6  set :server, 'mongrel'
7
8  get '/perf' do
9File.open("testfile", "r") do |infile|
10      loop do
11        if infile.gets == "c\n"
12          break
13        end
14      end
15    end
16    body "achou c"
17  end
18
19 end
20
21 MyApp.run!
```