Introdução a Programação Baseada em Eventos

Daniel de Angelis Cordeiro

Aula para a disciplina de
MAC 438 – Programação Concorrente

21 de junho de 2005

Ou “Como evitar o uso do que aprendemos em Concorrente em busca de escalabilidade” :-)

Daniel de Angelis Cordeiro

Aula para a disciplina de
MAC 438 – Programação Concorrente

21 de junho de 2005

Sumário

- Introdução
- Os 4 problemas clássicos de desempenho
- Arquiteturas escaláveis
- SO e Escalabilidade

Um pouco do meu trabalho de mestrado:

- Descrição do projeto
- Arquitetura do Quake
- Resultados Preliminares
- Plano de Trabalho

Introdução

- Por que não conseguimos lidar com dezenas de milhares de clientes com os computadores atuais? Se usarmos uma máquina de 1 GHz, 2 Gb de memória RAM e uma conexão Gigabit para atender 20.000 requisições simultâneas, teríamos 50 KHz, 100 Kbytes e 50 Kbits/s para o tratamento de cada requisição. **Não é suficiente?**

- Será que é exagero?
  - Efeito *slashdot*;
  - Página da CNN durante 11/9 [30.000 req./s].

Os 4 “arqui-inimigos” do desempenho de servidores

- Alocação de Memória: repetidas alocações e desalocações de memória podem prejudicar o desempenho. Pré-alocação ou *lookaside lists* (coloque os objetos que não serão utilizados em uma lista ao invés de desalocá-los e reutilize-os) resolvem o problema;
- Cópia de Dados: duplicação de dados dentro do programa (comum entre os programas orientados a objetos) ou entre o programa e o SO;
- **Troca de Contexto**;
- **Contenção de *locks*.**

Meu servidor Web

```
si.sin_family = PF_INET;
inet_aton("127.0.0.1", &si.sin_addr);
si.sin_port = htons(80);
bind(fd, (struct sockaddr*)si, sizeof si);
listen(fd);

while ((cfd=accept(fd,(struct sockaddr*)si,sizeof si)) != -1)
{
    read_request(cfd); /* read(cfd,...) até ler "\r\n\r\n" */
    write(cfd, "200 OK HTTP/1.0\r\n\r\n"
               "Minha página várzea.", 19+20);
    close(cfd);
}
```

Meu servidor Web (II)

- Parece eficiente, mas **é tosco!**
- O processo pode travar nas seguintes operações:
  - **accept()**: mas tudo bem, se travar é porque não tem nenhuma requisição para atender;
  - **read()**: apesar de accept() indicar que há uma requisição a ser processada, os dados podem estar incompletos;
  - **write()**: bloqueia se os *buffers* do SO estiverem cheios.
- Sem contar que o servidor não implementa o protocolo HTTP.

Arquiteturas Escaláveis

Aplicações que disponibilizam serviços para um grande número de usuários precisam multiplexar o tratamento de diversas requisições simultaneamente. Apresentaremos algumas das arquiteturas utilizadas para tal finalidade:

- Multi-process (MP);
- Multi-threaded (MT);
- Single process event-driven (SPED);
- Asymmetric multi-process event-driven (AMPED);
- Staged event-driven architecture (SEDA).

Arquitetura Multi-process

- Nesta arquitetura, todas as etapas são executadas, seqüencialmente, por um único processo.
- Quando um processo executa uma operação bloqueante, outro processo é automaticamente eleito pelo sistema operacional para usar a CPU.
- Como múltiplos processos podem ser utilizados (tipicamente algo entre 20 e 200 processos), requisições são atendidas simultaneamente.
- Não é necessário realizar sincronização entre processos. Entretanto é mais difícil realizar otimizações que dependam de memória global.

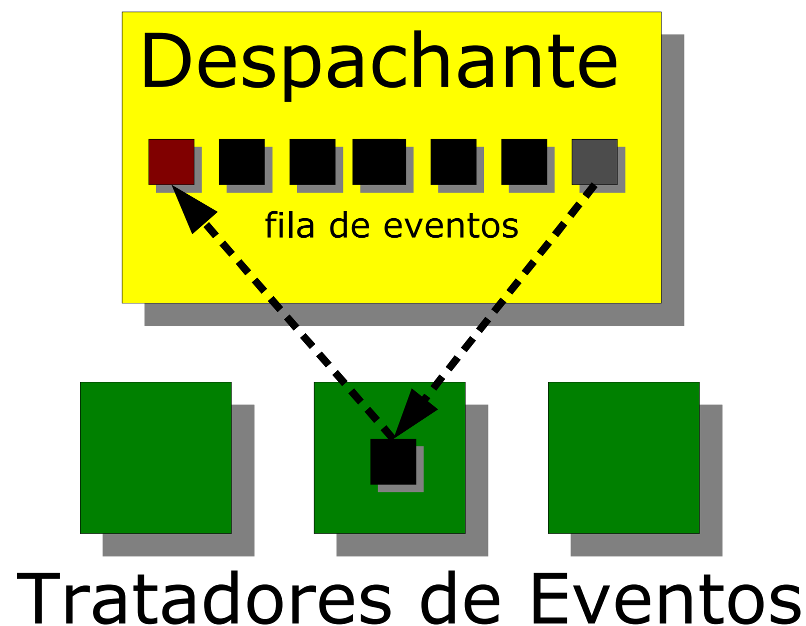
Arquitetura Multi-threaded

- A arquitetura MT emprega múltiplas *threads* independentes que utilizam um espaço de endereçamento compartilhado. Cada *thread* executa todas as etapas de uma requisição.
- A diferença em relação a arquitetura MP é que neste modelo é mais fácil realizar otimizações que necessitem de memória compartilhada (por exemplo, um cache de URLs válidas em servidores web).
- Entretanto, é necessário utilizar controles de acesso a memória compartilhada e é imprescindível que o sistema operacional forneça suporte a *threads.*

Arquitetura Single-process event-driven

- Utiliza um único processo, implementado utilizando o paradigma de programação baseada em eventos.

- O fluxo de execução é controlado pelos eventos ocorridos;
- Possui uma única linha de execução (não há concorrência);
- Não há preempção nos tratadores de eventos.

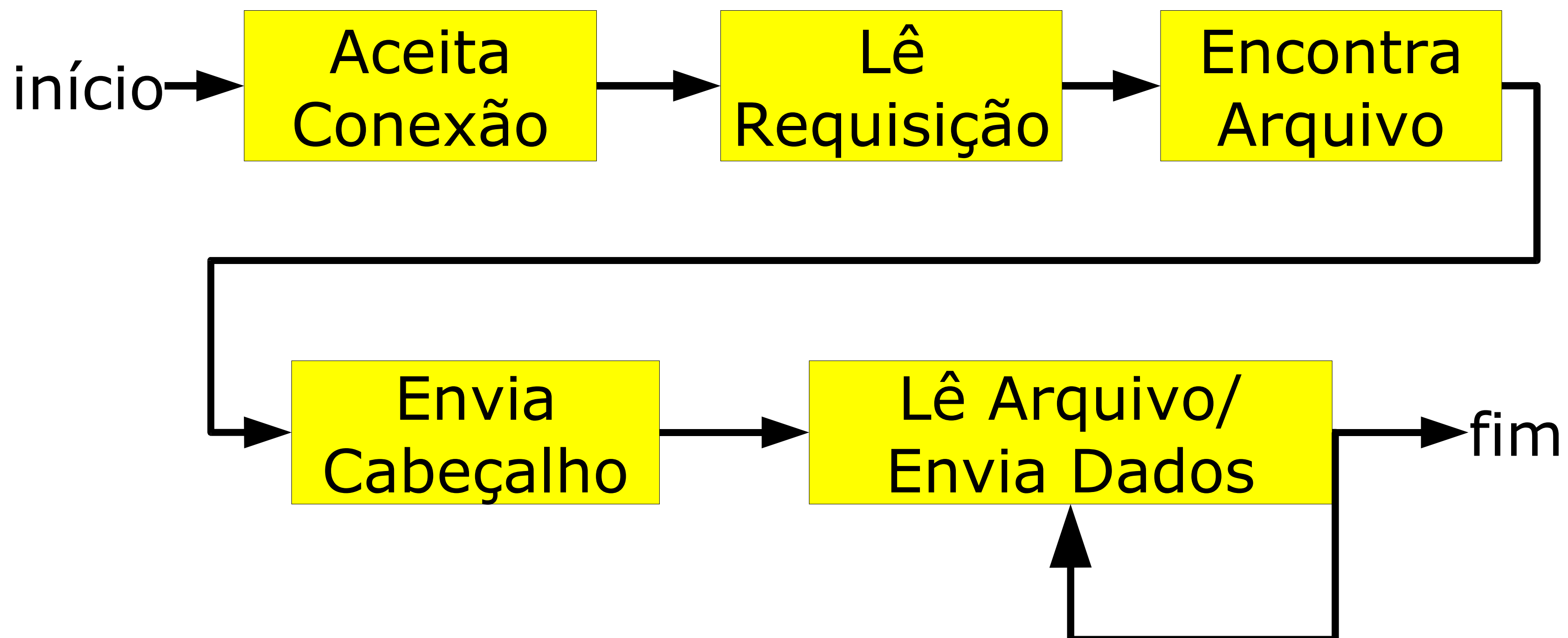


Arquitetura SPED (cont.)

- O servidor utiliza operações não-bloqueantes para realizar operações de E/S assíncronas (utilizando *select()*, *poll()*, *epoll()*, etc.).
- A arquitetura SPED pode ser vista como uma máquina de estados. Em cada iteração, o servidor realiza um *select()* para determinar se alguma operação de E/S foi concluída e, se foi concluída, gera um novo evento que será tratado pelo despachante de eventos.
- Não necessita de nenhuma forma de sincronização, mas pode apresentar problemas em SOs que não implementam operações não-bloqueantes adequadamente.

Exemplo de servidor baseado em eventos

- O exemplo clássico de implementação de um servidor baseado em eventos é o servidor web.
- Um servidor web pode ser dividido da seguinte forma:



Vocês não ficam emocionados?

Ao ver que a arquitetura SPED é mais eficiente do que a multi-*threaded* e, como bônus, livra os desenvolvedores (vocês, que estão se descabelando tentando implementar o melhor EP da turma) dos *locks,* monitores e de todos os problemas que aprendemos a resolver em Programação Concorrente?

:-)

Arquitetura Asymmetric Multi-Process Event-Driven

- A arquitetura AMPED combina a abordagem baseada em eventos da arquitetura SPED com múltiplos processos (ou *threads*) ajudantes.
- Quando uma operação potencialmente bloqueante precisa ser realizada, o despachante utiliza um ajudante existente ou cria um novo. Assim que a operação termina, o ajudante notifica o despachante utilizando canal de comunicação inter-processos (ex: pipe).
- Idealmente o término do tratamento de um evento assíncrono produz um novo evento.

Staged Event-Driven Architecture

- A arquitetura SEDA divide um programa baseado em eventos em conjunto de estágios interligados por filas de eventos. Cada estágio é composto por uma fila de eventos, um despachante de eventos e um *pool* de *threads*.
- Um controle de admissão de novos eventos em cada um dos estágios permite a adaptação do servidor em caso de sobrecarga.
- Cada estágio determina dinamicamente o tamanho mais adequado para seu *pool* de *threads* ajudantes.

SO e Escalabilidade

- Historicamente os sistemas operacionais oferecem aos processos uma visão “virtual” dos recursos: o processo se utiliza dos recursos como se fosse o único processo existente.
- Por conta disso os SOs atuais não permitem que as aplicações influenciem as decisões tomadas nos gerenciamentos de recursos – escondendo das aplicações o fato de que os recursos são limitados e compartilhados com outros processos.
- Vários trabalhos visam a criação de extensões do SO para torná-lo mais adequado para a criação de aplicações escaláveis.

SO e Escalabilidade (II)

Dentre os trabalhos estudados que analisam o impacto da interação entre SO e aplicação, podemos citar:

- *Scheduler Activations*;
- *accept()able Strategies for Improving Web Server Performance;*
- *Lazy Asynchronous I/O for Event-Driven Servers*
- *User-level connection tracking;*
- *sendfile();*
- /dev/epoll;
- K42.

E aí, vamos falar de joguinhos?

Motivação

- Redes de computadores velozes e confiáveis propiciam a disponibilização de serviços a um grande número de usuários.
- Para que os usuários possam acessar simultaneamente esse serviço é necessário entender como programá-los de forma a atingir escalabilidade.
- A escalabilidade de aplicações científicas foi bastante estudada. Mas pouco se sabe sobre escalabilidade de aplicações comerciais.

Motivação (II)

- Em particular, pouco se estudou sobre jogos de computadores, apesar de seu interesse teórico e comercial.
- A criação de jogos interativos maciçamente multi-usuários é um desafio. Atualmente apenas jogos do tipo RPG – cuja relação temporal entre o acontecimento dos eventos e sua conseqüências é menos rígida – possuem versões maciçamente multi-usuários (são os chamados MMORPG).

Objetivo

- Utilizaremos o jogo interativo, multi-usuário, Quake – disponibilizado publicamente pela Id Software sob a licença GPL – para estudar a escalabilidade dessa classe de aplicações.
- Nosso objetivo é abstrair o *workload* do servidor do Quake em ambientes multi-processados, caracterizar o uso das CPUs sob tal carga e investigar possíveis melhorias de aproveitamento do processamento disponível utilizando os recursos dos sistemas operacionais Linux e K42.

Arquiteturas de Jogos Comerciais

Pouco é divulgado sobre a arquitetura dos jogos comerciais. Alguns estudos indicam que a maioria utiliza-se das seguintes arquiteturas:

- Cliente/servidor, terminal burro;
- *Shards* e servidores distribuídos;
- *Peer-to-peer*;
- Puramente baseado em banco-de-dados.

Arquitetura do Quake

- Quake é um jogo de ação interativo, multi-usuário, desenvolvido e distribuído pela *Id Software*. Seu lançamento, em 31 de maio de 1996, foi considerado um marco na história dos jogos pois introduziu grandes avanços tecnológicos em jogos 3D.
- A arquitetura utilizada é a cliente/servidor. Os clientes conectam-se a um servidor centralizado, que é responsável por sincronizar e simular as ações dos clientes. Os clientes são responsáveis por representar graficamente o estado atual do jogo e pela interação com os usuários.

Arquitetura do Quake (II)

O processamento do servidor é dividido em quadros (*frames*) que consistem das seguintes tarefas:

- **remoção de clientes inativos**: verifica os clientes que deixaram de comunicar-se com o servidor e os remove da sessão;
- **evolução da simulação**: evolui o modelo de simulação física (velocidade, aceleração, posicionamento de objetos, etc.) e o modelo do jogo (IA, efeitos de luz, som, etc.);

Arquitetura do Quake (II)

- **processamento de novas mensagens**: o servidor processa todas as mensagens enviadas pelos clientes (tipicamente será uma mensagem do tipo `MOVE`), calcula a nova posição das entidades e marca quais entidades potencialmente foram afetadas pela ação;
- **construção e envio de respostas**: o servidor envia mensagens UDP apenas para os clientes que enviaram alguma mensagem no quadro anterior. Para cada cliente, o servidor envia dados sobre os elementos alterados presentes na área de atuação do jogador.

Resultados Preliminares

- Modificações no código do Quake:
  - Automatização do cliente para que fosse possível a realização de testes com um número maior de usuários;
  - Remoção do limite artificial de 32 clientes por sessão. Atualmente conseguimos manter 160 jogadores simultâneos em uma sessão.

Resultados Preliminares (II)

- Caracterização do uso de CPU da versão seqüencial do servidor do Quake com o auxílio do *GNU Profiler* (*gprof*) e do *Linux Trace Toolkit*:
  - 17,5% do tempo foi gasto com a simulação do modelo físico, 53,5% do tempo foi utilizado para o processamento de mensagens dos clientes, 27% para a construção das mensagens de resposta para os clientes e os outros 2% foram utilizados para outras tarefas.
  - O processo utiliza a CPU durante aproximadamente 90% do tempo. O restante do tempo é utilizado por chamadas de sistema **`select()`** e **`socketcall()`**.

Plano de Trabalho

- Implementação de um programa que abstraia a carga de trabalho (*workload*) do Quake;
- Investigação de paralelização com distribuição dinâmica;
- Condução de experimentos no Linux e K42;
- Investigação de alternativas e possibilidade de melhorias nos serviços fornecidos pelo sistema operacional;
- Redação da dissertação.

Referências

- Texto para o meu exame de qualificação do mestrado: http://www.ime.usp.br/~danielc/qualificacao.pdf

- E duas páginas muito boas sobre o assunto:
  - *The C10K problem*: http://www.kegel.com/c10k.html
  - *High-Performance Server Architecture*: http://pl.atyp.us/content/tech/servers.html