UNIVERSIDADE FEDERAL DE GOIÁS

ENGENHARIA DE SOFTWARE

SOFTWARE CONCORRENTE E DISTRIBUIDO – INF0298

Discente:

Adriel Lenner Vinhal Mori

Docente:

Vagner Jose Do Sacramento Rodrigues

ATIVIDADE DE PESQUISA – 16 jun 2023

Goiânia

2023

1. Middleware

O middleware é um componente crucial no campo da informática, desempenhando um papel fundamental na integração e na comunicação entre diferentes sistemas e aplicativos. Atuando como uma camada intermediária entre o sistema operacional e as aplicações, o middleware permite a troca de dados e serviços de forma eficiente e transparente. Ele oferece um conjunto de funcionalidades e abstrações que simplificam a complexidade dos sistemas distribuídos, facilitando a interconexão e a interoperabilidade entre diferentes componentes. Além disso, o middleware fornece recursos para garantir a segurança, a escalabilidade e o desempenho dos sistemas, contribuindo para a confiabilidade e o bom funcionamento das aplicações distribuídas. Em resumo, o middleware desempenha um papel crucial na construção de infraestruturas robustas e na implementação de soluções eficazes no contexto da informática.

2. Paralelismo

No domínio de sistemas distribuídos, o paralelismo desempenha um papel crucial na melhoria do desempenho e na eficiência das operações. O paralelismo refere-se à capacidade de executar múltiplas tarefas simultaneamente, distribuindo o processamento entre diferentes nós ou unidades de processamento em um sistema distribuído. Esse conceito é especialmente relevante em cenários onde grandes volumes de dados e tarefas intensivas em computação precisam ser processados de forma rápida e eficiente. O paralelismo pode ser aplicado em diferentes níveis de granularidade, desde a execução de threads simultâneas dentro de um único nó até a distribuição de tarefas em uma rede de nós interconectados. Além de melhorar o desempenho, o paralelismo também pode fornecer tolerância a falhas e capacidade de escalabilidade em sistemas distribuídos, permitindo que eles se adaptem à carga de trabalho crescente. No entanto, o desenvolvimento e a implementação efetiva de algoritmos e estratégias paralelas em sistemas distribuídos exigem considerações cuidadosas de sincronização, comunicação e balanceamento de carga, a fim de aproveitar ao máximo o potencial de paralelismo e obter resultados significativos em termos de desempenho e eficiência.

3. Aplicações IO-intensive e CPU-intensive

Imagem ilustrando a operação de CPU-intensive

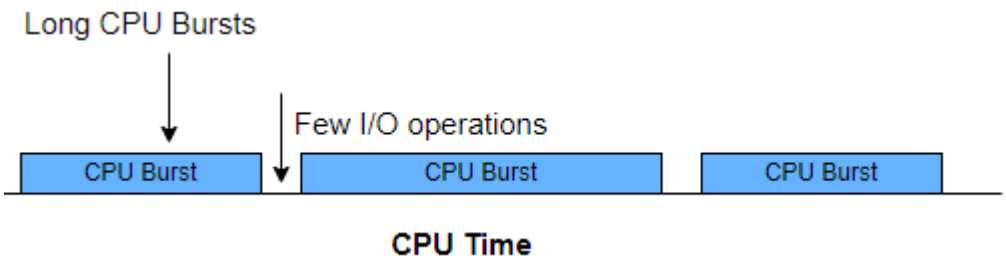
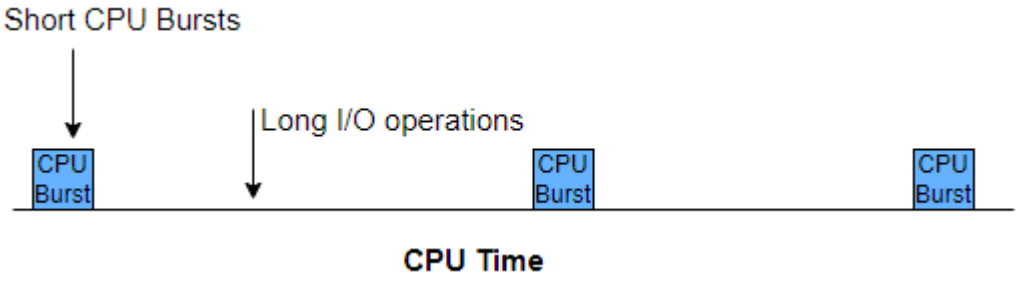


Imagem ilustrando a operação de I/O-intensive



As aplicações IO-intensive são aquelas em que a maior parte do tempo de execução é gasta na realização de operações de entrada e saída, como leitura e gravação de arquivos, acesso a bancos de dados ou comunicação com dispositivos externos. Essas aplicações tendem a ter um uso intensivo de recursos de armazenamento e rede, com a necessidade de lidar eficientemente com operações assíncronas, gerenciamento de buffers e otimização de transferência de dados. Por outro lado, as aplicações CPU-intensive são aquelas que requerem um alto poder de processamento para a execução de cálculos complexos e intensivos em CPU. Essas aplicações são comumente encontradas em áreas como modelagem e simulação, processamento de imagens e vídeos, criptografia, entre outros. Elas exigem que o processador execute algoritmos computacionalmente intensivos, realizando operações matemáticas ou lógicas complexas em grandes volumes de dados.

A distinção entre aplicações IO-intensive e CPU-intensive é importante para o projeto e a otimização de sistemas e infraestruturas. Para aplicações IO-intensive, a eficiência no gerenciamento de recursos de armazenamento, rede e operações assíncronas é fundamental. Já para aplicações CPU-intensive, é crucial ter uma capacidade de processamento adequada e otimização de algoritmos para garantir um desempenho eficiente.

Compreender a natureza das aplicações IO-intensive e CPU-intensive é essencial para a seleção de hardware adequado, alocação de recursos e estratégias de otimização. O conhecimento sobre essas características auxilia na criação de soluções eficientes e no dimensionamento correto de sistemas, garantindo um desempenho satisfatório e uma melhor utilização dos recursos disponíveis.

4. Thread Safe

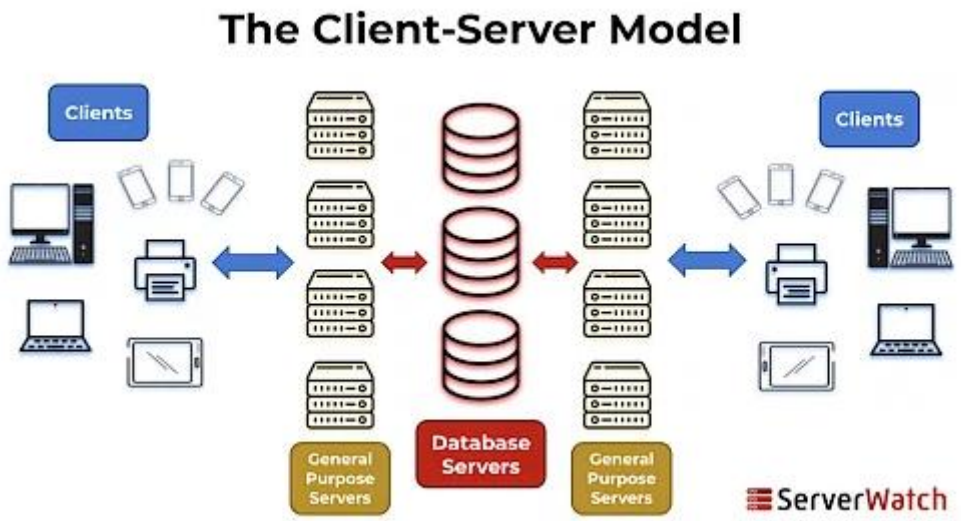
Thread Safe é um conceito importante para garantir a integridade e a consistência dos dados em ambientes concorrentes e multithread. Uma aplicação ou biblioteca é considerada thread safe quando pode ser utilizada por várias threads simultaneamente, sem corromper os dados compartilhados ou gerar resultados inconsistentes. Isso é particularmente relevante em sistemas onde múltiplas threads executam operações simultaneamente, como em aplicações de servidor ou em processamento paralelo. Para alcançar a thread safety, é necessário implementar mecanismos adequados de sincronização e controle de acesso aos recursos compartilhados, como variáveis, estruturas de dados e arquivos. Esses mecanismos incluem o uso de mutexes, semáforos, locks, entre outros, para garantir a exclusão mútua e evitar condições de corrida. Além disso, é necessário considerar a coerência de cache, a ordem de execução e outros aspectos específicos do ambiente de execução para garantir a corretude dos

resultados. Ao projetar e desenvolver aplicações e bibliotecas thread safe, os desenvolvedores podem assegurar que as operações concorrentes sejam executadas de forma correta e consistente, evitando problemas de race conditions e garantindo a estabilidade e a confiabilidade do software em ambientes multithread.

5. Memória compartilhada

No contexto da computação concorrente e distribuída, a memória compartilhada desempenha um papel fundamental na comunicação e na coordenação entre processos ou threads que executam em diferentes unidades de processamento. A memória compartilhada permite que múltiplos processos ou threads acessem um espaço de memória comum, possibilitando a troca de dados e a sincronização de operações. Essa abordagem é particularmente útil em cenários onde é necessário compartilhar informações em tempo real e coordenar o trabalho de múltiplos participantes. No entanto, o uso adequado da memória compartilhada requer a implementação cuidadosa de mecanismos de sincronização para evitar condições de corrida e garantir a consistência dos dados compartilhados. Além disso, é importante considerar as questões de coerência de cache e latência de acesso à memória em sistemas distribuídos, onde a memória compartilhada pode estar fisicamente distribuída em diferentes nós. Com uma correta utilização da memória compartilhada, é possível obter ganhos significativos em desempenho e eficiência em sistemas concorrentes e distribuídos, permitindo a colaboração e a execução paralela de tarefas complexas em tempo real.

6. Arquitetura cliente-Servidor



Conhecidos como redes A arquitetura Cliente-Servidor é uma abordagem amplamente utilizada no campo da informática para projetar e implementar sistemas distribuídos. Nesse modelo, os sistemas são divididos em dois componentes principais: o cliente e o servidor. O cliente é responsável por solicitar serviços ou recursos ao servidor, enquanto o servidor é encarregado de atender essas solicitações e fornecer os serviços solicitados. A comunicação entre o cliente e o servidor ocorre por meio de troca de mensagens, seguindo um protocolo estabelecido.

Essa arquitetura oferece uma série de vantagens, como a centralização do processamento e do armazenamento de dados no servidor, permitindo que os clientes sejam mais leves e se concentrem na interface do usuário. Além disso, a separação entre cliente e

servidor facilita a escalabilidade e a manutenção do sistema, uma vez que é possível adicionar ou atualizar servidores sem afetar os clientes.

A arquitetura Cliente-Servidor é aplicada em uma ampla gama de sistemas e serviços, desde aplicações web e serviços de nuvem até sistemas de gerenciamento de banco de dados e redes corporativas. Ela fornece uma estrutura flexível e escalável para a construção de soluções distribuídas, permitindo a colaboração eficiente entre diferentes componentes e facilitando o compartilhamento de recursos e informações em tempo real. No entanto, é importante destacar que a arquitetura Cliente-Servidor também apresenta algumas limitações, como a dependência do servidor para o fornecimento de serviços e a necessidade de uma conexão estável entre cliente e servidor. Além disso, o desempenho do sistema pode ser afetado por gargalos no servidor, o que requer uma atenção cuidadosa no dimensionamento e na configuração dos recursos do servidor.

No geral, a arquitetura Cliente-Servidor é uma abordagem amplamente adotada e comprovada para projetar e implementar sistemas distribuídos, permitindo a interação eficiente entre clientes e servidores, possibilitando a construção de aplicações robustas, escaláveis e de alto desempenho.

7. Peer-to-Peer Achitecture

A arquitetura ponto a ponto (arquitetura P2P) é uma arquitetura de rede de computadores comumente usada na qual cada estação de trabalho, ou nó, tem os mesmos recursos e responsabilidades. Muitas vezes é comparado e contrastado com a arquitetura cliente/servidor clássica, na qual alguns computadores são dedicados a servir outros.

P2P também pode ser usado para se referir a um único programa de software projetado para que cada instância do programa possa atuar como cliente e servidor, com as mesmas responsabilidades e status.

As redes P2P têm muitas aplicações, mas a mais comum é para distribuição de conteúdo. Isso inclui publicação e distribuição de software, redes de entrega de conteúdo, streaming de mídia e peercasting para streams multicasting, o que facilita a entrega de conteúdo sob demanda. Outras aplicações envolvem redes de ciência, redes, pesquisa e comunicação. Até mesmo o Departamento de Defesa dos EUA começou a pesquisar aplicações para redes P2P para estratégias modernas de guerra de rede.

8. Modelo Fundamentais em sistemas Distribuídos:
    a. Modelo de interação

    O modelo de interação descreve como os componentes em um sistema distribuído se comunicam e trocam informações entre si. Ele define os protocolos, formatos de mensagem e mecanismos de comunicação utilizados para garantir a troca de dados confiável e eficiente. O modelo de interação também aborda questões como sincronização, consistência e concorrência, para garantir que as operações ocorram em uma ordem adequada e que a consistência dos dados seja mantida em um ambiente distribuído.

    b. Modelo de Falha

    O modelo de falha lida com as possíveis falhas que podem ocorrer em um sistema distribuído. Ele descreve os tipos de falhas que podem afetar os componentes,

como falhas de hardware, falhas de rede ou falhas de software, e como o sistema deve se recuperar dessas falhas. O modelo de falha também aborda a tolerância a falhas, que envolve a capacidade do sistema de continuar funcionando mesmo na presença de falhas em seus componentes.

c. Modelo de Segurança

O modelo de segurança trata da proteção dos dados e recursos em um sistema distribuído. Ele define políticas e mecanismos de segurança para garantir a confidencialidade, integridade e disponibilidade dos dados. Isso inclui a autenticação de usuários, o controle de acesso, a criptografia e a detecção de atividades maliciosas. O modelo de segurança também aborda a proteção contra ameaças, como ataques de negação de serviço, invasões e roubo de dados.

9. ScheduledExecutorService

O ScheduledExecutorService é uma interface do Java que estende o ExecutorService e permite a execução agendada de tarefas em um ambiente concorrente. Ele oferece métodos flexíveis para agendar tarefas em momentos específicos ou em intervalos regulares, sendo útil para tarefas repetitivas ou agendadas, como atualizações periódicas e coleta de métricas. Essa interface simplifica o gerenciamento de tarefas agendadas, fornecendo controle preciso sobre o tempo de execução e possibilitando o cancelamento e tratamento de exceções. Em resumo, o ScheduledExecutorService é uma ferramenta poderosa para agendar e executar tarefas programadas em aplicações concorrentes.

Exemplicação a partir de um pseucódigo atualizado a partir da atividade da ContaBancária.

```
import java.util.concurrent.*;

public class BankApplication {
    private ScheduledExecutorService executorService;
    private Account account;

    public BankApplication() {
        executorService = Executors.newScheduledThreadPool(1);
        account = new Account(1000); // saldo inicial de 1000 unidades monetárias
    }

    public void start() {
        // Agendar tarefa para execução a cada 30 dias
```

```
    executorService.scheduleAtFixedRate(new MonthlyInterestTask(), 0, 30, TimeUnit.DAYS);
}

public void stop() {
    // Parar a execução das tarefas agendadas e encerrar o executorService
    executorService.shutdown();
}

private class MonthlyInterestTask implements Runnable {
    @Override
    public void run() {
        // Calcular e adicionar juros mensais à conta
        double interestRate = 0.05; // taxa de juros mensal de 5%
        double interest = account.getBalance() * interestRate;
        account.deposit(interest);
        System.out.println("Juros mensais adicionados à conta: " + interest);
    }
}

private class Account {
    private double balance;

    public Account(double initialBalance) {
        this.balance = initialBalance;
    }

    public double getBalance() {
        return balance;
    }

    public void deposit(double amount) {
```

```
            balance += amount;
        }
    }

    public static void main(String[] args) {
        BankApplication bankApp = new BankApplication();
        bankApp.start();

        // Aguardar alguns meses para demonstrar a execução periódica da tarefa
        try {
            Thread.sleep(90 * 24 * 60 * 60 * 1000); // Aguardar 90 dias (3 meses)
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        bankApp.stop();
    }
}
```

REFERÊNCIAS:

Rouse, M. (2023, 22 de junho). Peer-to-Peer Architecture. Blog de Techpedia. Disponível em https://www.techopedia.com/definition/454/peer-to-peer-architecture. Acessado em: 20 jun 2023

Rodrigues, L.; Guerraoui, R. Introduction to reliable distributed programming. Springer, 2006. ISBN 9783540288459.

Ghemawat, Sanjay et al. The Google File System. In: Symposium on Operating Systems Principles (SOSP). ACM, 2003.