Resumo de Sistemas Distribuídos

(com base no livro Sistemas Distribuídos do Tim Kindberg e anotações das aulas)

Definição do que é um sistema distribuído:

Um sistema distribuído é aquele no qual os componentes localizados em computadores interligados em rede se comunicam e coordenam suas ações apenas passando mensagens.

Ex: Internet, Intranet e Computação Móvel.

Middleware: É a camada de software que fornece uma abstração de programação, assim como mascaramento da heterogeneidade das redes, do hardware, de sistemas operacionais e linguagens de programação adjacentes, oferecendo um modelo computacional uniforme parar ser usado pelos programadores de serviços e de aplicativos distribuídos.

Aqui estão algumas características:

Heterogeneidade: Devem ser construídos a partir de uma variedade de redes, sistemas operacionais, hardware e linguagens de programação diferentes. Os protocolos de comunicação.

Sistemas abertos: Os sistemas distribuídos devem ser extensíveis

Segurança: Criptografia é essencial para proteger dados sigilosos, entretanto, ainda há problemas com ataques de negação de serviço.

Escalabilidade: O sistema distribuído é considerado escalável se o custo de adição de usuários for constante no quesito recursos a serem adicionados. Os algoritmos utilizados devem ser compartilhados e evitar gargalhos desempenho com o recebimento de dados de forma hierárquica.

Tratamento de falhas: Deve haver tratamento para os erros do sistema

Concorrência: Em um ambiente concorrente cada recurso deve ser projetado para manter a consistência nos estados dos seus dados.

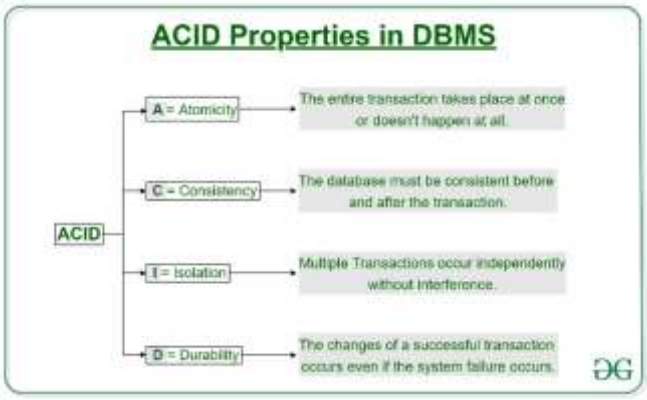
Transparência: O objetivo é tornar certos aspectos da distribuição invisíveis para o programador de aplicativos, para que este se preocupe apenas com o projeto de seu aplicativo.

Em uma computação distribuída, há consumo de recursos (estes recursos podem ser tanto de hardware, como de software).

As informações distribuídas passam por uma raid de clusters transferindo dados ao banco de dados, guardando arquivos.

Banco de dados relacional VS Banco de dados não relacional.

Banco relacional



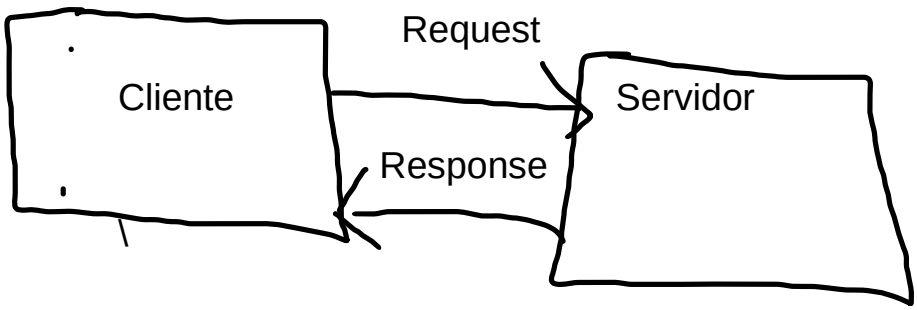
Banco não relacional:

Possui uma velocidade maior em realizar as suas querys.

Resumo das arquiteturas

1. Cliente-servidor: Um servidor fornece serviços aos clientes.
2. Multicamada: Os componentes do sistema são organizados em camadas com diferentes responsabilidades.
3. Peer-to-peer: Todos os computadores no sistema desempenham funções semelhantes.
4. Híbrido: Combinação de diferentes modelos para atender às necessidades específicas da aplicação.

Arquitetura Cliente – Servidor



O que é o HTTP?

HTTP é um protocolo que é composto por um Header, Blanckline e o Body. A porta utilizada é a 80, já o HTTPS é a 443.

Curiosidade: O Header é o que define a linguagem da resposta que será enviada pelo servidor.

Portas abaixo de 1000 são para as intranets.

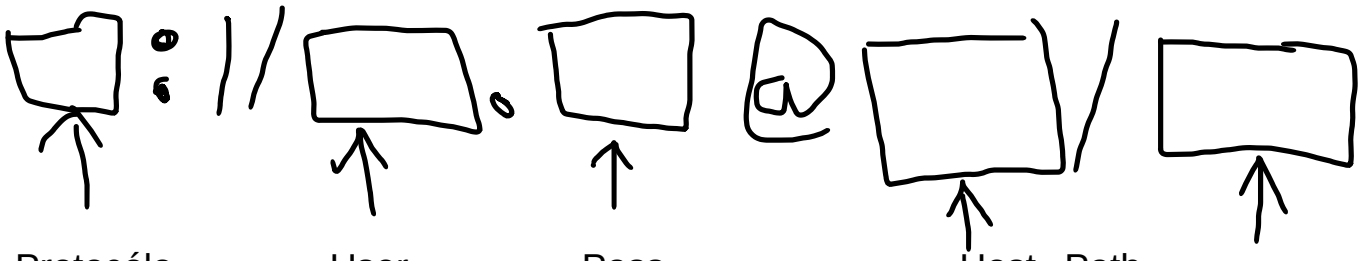
Estruturas de:

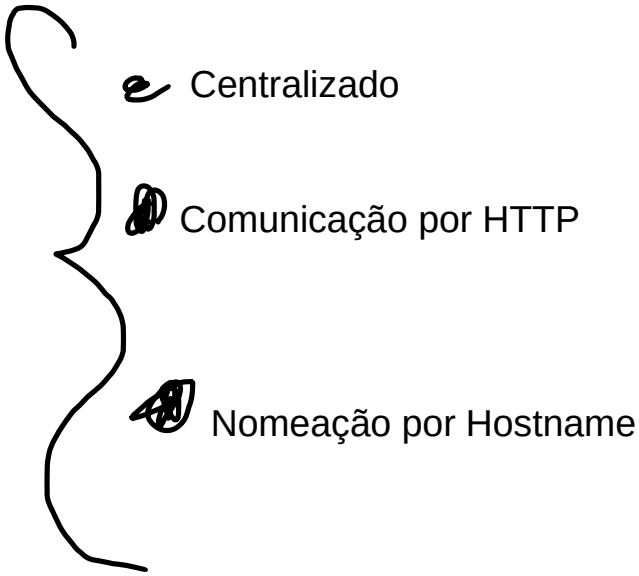
Mensagem : Request/ Response

Request: Get/Post

Get: Header + Blanckline + Body

Gramática da URL:





Estruturas monolíticas vs Microserviços

Estruturas Monolíticas

Definição:

Uma aplicação monolítica é construída como uma única unidade indivisível. Todos os componentes do software, como a interface de usuário, lógica de negócio e acesso a dados, estão integrados em um único programa.

Vantagens:

1. Simplicidade de Desenvolvimento: Facilita o desenvolvimento inicial, pois todos os componentes estão em um único código base.
2. Desempenho: A comunicação interna é rápida e eficiente porque tudo está dentro do mesmo processo.
3. Deploy Simples: Apenas um único artefato precisa ser implantado, simplificando o processo de deploy.

Desvantagens:

- Manutenção Complexa: À medida que a aplicação cresce, torna-se mais difícil de manter e modificar sem causar efeitos colaterais.
- Escalabilidade Limitada: Difícil escalar apenas partes específicas da aplicação. Normalmente, toda a aplicação deve ser escalada.
- Implantações Rígidas: Qualquer mudança, mesmo pequena, requer a reinstalação da aplicação inteira, o que pode ser arriscado e demorado.

Microserviços

Definição:

Uma arquitetura de microserviços divide a aplicação em pequenos serviços independentes, cada um executando um único propósito ou funcionalidade. Esses serviços se comunicam através de APIs.

Vantagens:

1. Escalabilidade: Cada serviço pode ser escalado independentemente, permitindo otimização de recursos.
2. Desenvolvimento Ágil: Equipes podem desenvolver, testar e implantar serviços de forma independente, acelerando o ciclo de desenvolvimento.

3. Resiliência: Falhas em um serviço não afetam necessariamente outros serviços, aumentando a robustez da aplicação.
4. Flexibilidade Tecnológica: Cada serviço pode ser desenvolvido com a tecnologia mais adequada para sua funcionalidade específica.

Desvantagens:

- Complexidade de Gerenciamento: A coordenação entre diversos serviços aumenta a complexidade operacional, incluindo o gerenciamento de deploys, monitoramento e debugging.
- Comunicação entre Serviços: A comunicação entre serviços pode introduzir latência e exigir estratégias de tolerância a falhas.
- Sobrecarga de Rede: A comunicação entre serviços geralmente ocorre via rede, o que pode aumentar a sobrecarga e a latência.

Comparação

- Desempenho: Monolíticos tendem a ter um desempenho interno melhor devido à ausência de comunicação via rede, mas podem enfrentar gargalos em sistemas grandes. Microserviços, embora possam enfrentar latências de rede, permitem otimizações específicas.
- Escalabilidade: Microserviços oferecem uma escalabilidade mais granular e eficiente, enquanto monolíticos requerem escalabilidade uniforme.
- Manutenção: A modularidade dos microserviços facilita a manutenção e a evolução contínua da aplicação, ao contrário dos monolíticos que podem se tornar complexos e difíceis de modificar.
- Ciclo de Desenvolvimento: Microserviços suportam um ciclo de desenvolvimento mais ágil e independente, comparado aos monolíticos que podem ser mais lentos devido à integração mais rígida.

Em resumo, a escolha entre estruturas monolíticas e microserviços depende das necessidades específicas do projeto, da escala da aplicação, e das preferências e competências da equipe de desenvolvimento. Enquanto estruturas monolíticas podem ser mais adequadas para aplicações menores e mais simples, arquiteturas de microserviços são geralmente preferidas para sistemas grandes e complexos que requerem alta escalabilidade e flexibilidade.

MPI

A MPI (Interface de Passagem de Mensagens) é um paradigma de programação para computadores paralelos que facilita a comunicação entre processos em diferentes máquinas. Ela define uma interface padrão e funcionalidade para uma ampla gama de recursos de comunicação, tornando o desenvolvimento de programas paralelos mais portátil e eficiente.

Objetivo Principal: Facilitar a comunicação e a coordenação entre processos que estão distribuídos em diferentes máquinas em um ambiente de computação paralela.

Contexto de Uso: É amplamente utilizado em supercomputadores e clusters para aplicações científicas e de engenharia que requerem grande poder computacional.

Mensagens Bufferizadas:

Definição: As mensagens são armazenadas temporariamente em um buffer até que o destinatário esteja pronto para recebê-las.

Funcionamento: Quando um processo envia uma mensagem, ela é colocada em um buffer, permitindo que o processo remetente continue a executar sem esperar que o destinatário leia a mensagem.

Vantagem: Aumenta a eficiência, pois o remetente não precisa esperar a disponibilidade do destinatário. Adequado para sistemas onde a comunicação é frequente e a sincronização estrita não é necessária.

Desvantagem: Pode haver sobrecarga de memória se o buffer encher, exigindo gerenciamento adicional para lidar com buffers cheios.

Mensagens Não Bloqueantes:

Definição: O envio ou recebimento de mensagens ocorre sem que o processo precise esperar pela conclusão da operação.

Funcionamento: O processo que envia ou recebe a mensagem continua a execução imediatamente após iniciar a operação de envio ou recebimento, independentemente do estado do destinatário ou remetente.

Vantagem: Melhora a responsividade e permite que processos realizem outras tarefas enquanto aguardam a comunicação.

Desvantagem: Requer mecanismos para verificar o estado das mensagens, como callbacks ou polling, aumentando a complexidade do código.

Mensagens Não Bufferizadas:

Definição: As mensagens não são armazenadas em um buffer intermediário; o envio e recebimento ocorrem diretamente entre os processos.

Funcionamento: O remetente deve esperar até que o destinatário esteja pronto para receber a mensagem, e vice-versa.

Vantagem: Simplicidade na implementação e no gerenciamento, pois não há necessidade de gerenciar buffers.

Desvantagem: Pode levar a atrasos e ineficiências, pois processos podem ficar ociosos esperando uns pelos outros.

Mensagens Bloqueantes:

Definição: O processo que envia ou recebe a mensagem fica bloqueado (em espera) até que a operação de comunicação seja concluída.

Funcionamento: O remetente espera até que o destinatário esteja pronto para receber a mensagem, ou o destinatário espera até que uma mensagem esteja disponível para leitura.

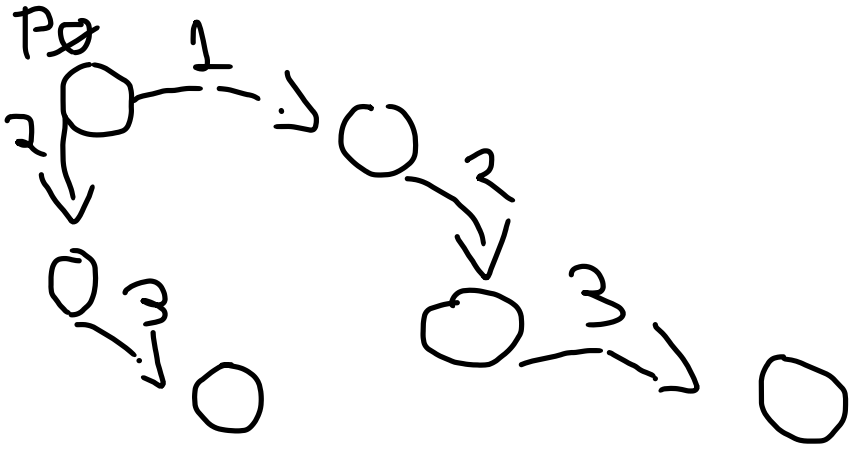
Vantagem: Garante sincronização estrita entre os processos, útil em sistemas que requerem comunicação ordenada e sincronizada.

Desvantagem: Pode levar a baixa utilização de recursos, pois processos podem ficar bloqueados esperando pela comunicação, reduzindo a eficiência geral do sistema.

Solução trivial para MPI:

Dividir o conjunto de dados pela quantidade de processos.

Per-to-per:



RMI

RMI, ou Remote Method Invocation, é um protocolo para criar aplicativos distribuídos em Java. Ele permite que objetos em diferentes máquinas virtuais Java se comuniquem e interajam como se estivessem no mesmo local.

Servidor: Executa os objetos remotos que podem ser acessados por clientes.

Cliente: Obtem referências a objetos remotos no servidor e invoca seus métodos.

Registro RMI: Armazena referências a objetos remotos e as fornece aos clientes.

Stub: Representa um objeto remoto no cliente e traduz chamadas de método em solicitações RMI.

Vantagens:

Carregamento Dinâmico de Código: O código das classes dos objetos remotos pode ser carregado dinamicamente na máquina virtual Java do cliente, permitindo a execução de novas tarefas sem reinicialização do servidor.

Transparência de Rede: A comunicação entre objetos remotos parece com chamadas de método Java regulares, abstraindo a complexidade da rede.

Flexibilidade: O RMI pode ser usado para diversos tipos de aplicações distribuídas, desde simples clientes-servidor até sistemas complexos baseados em agentes.

Coletivas:

Scatter: Envia e espalha os dados

Gather: Recebe os pedacinhos dos processos. Tem que ter o recolhimento pelo processo raíz das importações

Reduce: Pega conjuntos de dados e reduz para um dado final. Utiliza hash maps para poder realizar esta redução(Usado em Big Data).

Sincronização e relógios:

UTC = Universal Time Coordinated

Protocolo mais utilizado para sincronizar relógios: NTP

Relógios lógicos:

Prover a identificação da ordem dos eventos

Não se os relógios estão próximos do tempo real.

Relógios físicos:

Para cenários onde os relógios devem ser apenas iguais e não devem se desviar do tempo real.

Problemas:

Não pode regredir, no máximo diminuir a frequência do clock.

Em uma rede é difícil medir com precisão com o delay da rede.

Atraso na rede mal calculada para computar o tempo certo nos ajustes dos relógios.

Algoritmo de Berkeley

O algoritmo de Berkeley é um método de sincronização de relógios em sistemas distribuídos. Esse algoritmo é utilizado para alinhar os relógios de todos os computadores em uma rede distribuída, garantindo que todos os nós tenham aproximadamente o mesmo horário.