Resumo: Livro DSID

Capítulo 1: Introdução

Sistema distribuído: Coleção de elementos computacionais autônomos que aparece aos usuários como um sistema único e coerente.

Nós (ou Coleção de elementos computacionais autônomos):

Nós agem independentes um dos outros.

São criados para atingirem objetivos comuns, os quais são realizados pela troca de mensagens uns com os outros.

Reagem as mensagens recebidas, as quais são processadas e, então, passadas adiante no processod de comunicação.

Cada um tem sua própria noção de tempo, ou seja, não há um relógio global.

Participam de dois tipos de grupos, e gerir esses grupos pode ser muito difícil pela parte de controle de admissão:

- **Grupo aberto**: Qualquer nó pode se juntar ao grupo, isto é, qualquer nó pode enviar mensagens para outro nó do sistema.
- **Grupo fechado**: Somente nós do grupo são permitidos se comunicar uns com os outros e um mecanismo separado cuida da parte de admitir ou excluir um nó.

Um sistema distriuído é geralmente organizado como uma rede de overlay, neste caso o nó é um processo de software contendo uma lista de outros processos que ele pode enviar mensagens. Existe dois tipos de rede de overlay:

- **Overlay estruturado**: Cada nó tem uma lista de vizinhos bem definida com os quais ele pode se comunicar. Ex: os nós podem se organizar em anel lógico ou em árvore.
- **Overlay não-estruturado**: Cada nó tem um número de referências para selecionar aleatoriamente outros nós.

Sistema coerente e simples:

Deveria ter uma visão de sistema-único, ou seja, usuários finais não deveriam ter notar que eles estão lidando com o fato de que processos, dados, e controles estão dispersos na rede de computadores. Atingir este comportamento é muito difícil, por isso um sistema distribuído pode aparentar ser simples e coerente ao invés de ser de fato.

Um sistema distribuído é coerente se age da maneira esperada pelos usuários. Mais especificamente, a coleção de nós, deste sistema, age da mesma maneira como um todo, independente de onde, quando, e como se da a interação entre usuario e sistema.

Pela complexidade do sistema, haverá falhas em algumas partes do sistema e esconder essas falhas, pode ser particularmente muito difícil.

Middleware e sistemas distribuídos:

Middleware é a camada posta acima dos sistemas operacionais da máquina, a qual tem a responsabilidade de gerenciar os recursos, oferecendo à aplicação um compartilhamento e implementação eficiente dos recursos através da rede.

Serviços típicos oferecidos:

- **Comunicação**: Chamada de procedimento remoto (RPC) permite a aplicação invocar uma função que está implementada e executada em um computador remoto como se estivesse localmente disponível.
- **Transações**: Oferece suporte de transações atômicas ao estilo de ou realiza tudo ou nada.
- **Composição de serviço**: Ajuda a padronização de como os serviços web são acessados e providenciando o jeito de generar suas funções em uma ordem específica.
- **Confiabilidade**: O mesmo princípio das transações, garante que uma mensagem de um processo ou vai chegar para todos os processos ou para nenhum.

Objetivos do Sistema distribuído:

Suporte a compartilhamento de recursos:

- Facilitar aos usuários e aplicações acesso e compartilhamento de recursos.
- Famoso caso do BitTorrent nas redes P2P.

Fazer uma distribuição transparente:

- Processos e recursos estão fisicamente distribuídos através de múltiplos computadores (às vezes separados por longas distâncias).
- Tipos de transparência:
  - Acesso: Esconde a representação do dado em si e como o objeto é acessado.

        - Localização: Esconde onde o objeto está.
        - Relocação: Esconde que o objeto pode ser movido enquanto está sendo usado.
        - Migração: Esconde que o objeto pode ser movido.
        - Replicação: Esconde que o objeto é replicado.
        - Concorrência: Esconde que o objeto pode ser compartilhado por vários usuários.
        - Falha: Esconde a falha e recuperação de um objeto.
    - Fazer uma aplicação ser extremamente transparente pode não ser uma boa ideia. Ao aumentar o grau de transparência da aplicação o desempenho pode ficar comprometido.

Ser aberto:

- Sistema que oference componentes que podem ser facilmente usados ou integrados por outros sistemas.
- **Interomperabilidade**: Caracteriza que dois sistemas completamente diferentes, construidos por pessoas diferentes podem coexistir e trabalhar juntos.
- **Composibilidade**: Caracteriza que uma aplicação A pode ser executada, sem modificação, por uma aplicação B que implementa a mesma interface que A.
- **Extensibilidade:** Caracteriza um sistema que é fácil de adicionar componentes ou modificar os que existem sem haver complicações.
- Separar política de mecanismo

Ser escalável:

- **Dimensões de escalabilidade**:
    - **Tamanho**:

        Vários usuários e recursos podem ser adicionados ao sistema ssem nenhuma queda significante de desempenho.

        Usar servidores distribuídos, pois centralizar os processos pode consumir todo o poder computacional

    - **Geográfico**:

        Os usuários e recursos podem estar muito distantes um dos outros, porém o delay de comunicação dificilmente é percebido.

        Problema com comunicação síncrona: cliente bloqueia até que o servidor responda.

Comunicação não confiável

- **Administrativo**:

  É facilmente administrado mesmo que tenha vários administradores independentes.

  Problema maior com conflitos de polítcas com respeito ao uso de recursos (e pagamentos), gerenciamento e segurança.

- **Técnicas de Escalamento**:
  - **Escondendo latências de comunicação**:

    Aplica-se ao problema geográfico.

    Evite ficar esperando por respostas de requisições de serviços remotos o máximo que puder.

    Usar somente comunicação assíncrona.

    Em aplicações interativas, uma solução melhor é mover parte da comunicação do lado do servidor para o lado do cliente (JavaScript).

  - **Particionando e distribuindo**:

    Pegar um componente, dividí-lo em pedaços pequenos, e espalhá-los pelo sistema.

    Exemplo do DNS.

  - **Replicação**:

    Problemas de degradação de desempenho, usar replicação pode ser uma boa ideia.

    Aumento de disponibilidade do recurso.

    Ter uma cópia perto em questão geográfica diminui os problemas de latência na comunicação.

    Caching (parecido, mas diferente, decisão do cliente).

    Leva ao problema da inconsistência, uma atualização do recurso precisaria ser propagada por todas as cópias. Duas atualizações concorrentes geraria problemas de atualização em relação ao tempo, porém não há um relógio global.

Tipos de Sistemas Distribuídos:

- **Computação distribuída de alto desempenho**:
  - *Computação agrupada (cluster)*:

Ficou popular quando a razão do preço/desempenho de computadores pessoais e estações de trabalho melhorou.

É usada para programação paralela a qual um único programa está rodando em máquinas múltiplas ou paralelas.

Na maioria dos casos, os computadores agrupados possuem são iguais, possuem o mesmo sistema operacional e estão conectados através da mesma rede.

- *Computação em grade*:

  Diferente da computação agrupada, nenhuma suposição sobre hardware, sistema operacional, redes e etc é feita.

  O problema principal é que os recursos de diferentes organizações são trazidos juntos para permitir uma colaboração de um grupo de pessoas: organização virtual. Os recursos pertencentes a uma mesma organização virtual tem os direitos de acesso aos recursos providenciados por aquela organização.

  Recursos: servidores de computador, facilidades de armazenamento e banco de dados.

  Arquitetura:

  - Camada de fábrica:
    - Providencia interfaces (funções para pesquisar o estado e capacidades de um recurso) para um recurso local em um site específico.
  - Camada de conectividade:
    - Consiste na comunicação de protocolos apoiando as transações em grade que vão mandando múltiplos recursos.
  - Camada de recurso:
    - É responsável por gerenciar um único recurso. Usa as funções providenciadas pela camada de conectividade e chama diretamente a interface feita pela camada de fábrica para obter informações sobre o recurso.
  - Camada coletiva:
    - Lida com como coordenar múltiplos recursos.
  - Camada de aplicação:
    - Consiste de aplicações que mexe na organização virtual.

- *Computação em nuvem*:

  Baseado na computação utilizável: comprador paga por recurso.

  Arquitetura:

  - Hardware:
    - Formado apenas pelo o que controla o hardware: processadores, rotas.
    - São implementados no centros de dados
  - Infraestrutura:
    - Implanta técnicas virtuais providenciando uma infraestrutura consistindo de armazenamento virtual e recursos computacionais.
  - Plataforma:
    - É uma API específica do vendedor, a qual inclui chamadas para envio e e execução de um programa na nuvem do vendedor.
  - Aplicação:
    - As aplicações rodam nessa camada e são oferecidos ao usuários para customização.

  IaaS (camadas de hardware e infraestrutura)

  PaaS (camada de plataforma)

  SaaS (camada de aplicação)

- **Sistemas de Informação Distribuídos**:
  - Muita das soluções para middleware foram o resultado de trabalhar com infraestruturas que são fáceis de integrar aplicações com toda uma empresa de informações.
  - **Processamento de Informações Distribuídos**:

    Operações em banco de dados são carregados de primitivas chamadas de transações(begin transaction, end transaction, read, write, abort transaction, etc).

    Transações possuem as propriedades ACID:

    - **Atômicas**: A transação ocorre indivisivelmente mundo a fora.
    - **Consistentes**: Não viola as invariantes do sistema.
    - **Isoladas**: Transações concorrentes não intereferem umas as outras.
    - **Duráveis**: Uma vez que a transação é submetida, ela é permanente.

Uma transação pode possuir subtransações para lidar com diferentes servidores/banco de dados, porém, subtransações possuem um problema sutil:

- Uma transação começa várias subtransações em paralelo, e um dos commits faz o resultado estar visível para o pai desta transação. Se este pai abortar, o sistema inteiro será restaurado ao ponto anterior da transação mais alta ter começado.

Para lidar com o problema acima, o componente (middleware) que lida com múltiplas transações chama-se **Monitor de processamento de transações** ou **TP monitor**:

- Permite a aplicação acessar múltiplos servidores/banco de dados oferecendo um modelo de programação transacional. Basicamente ele coordena o comprometimento das subtransações seguindo o protocolo padrão **commit distribuido**.

- **Integração de aplicações corporativas**:

  Comunicação direta entre aplicações (lado cliente e lado servidor)

  RPC (chamadas de procedimento remoto): um componente da aplicação envia uma requisição para outro componente da aplicação fazendo uma chamada de procedimento local, o que resulta na requisição sendo empacotado como uma mensagem e mandada pro destinatário.

  RMI (invocação de método remoto): funciona da mesma maneira que o RPC, porém lida com objetos e não funções.

  MOM (mensagem orientada a middleware): Lida com o envio das mensagens de ambos os lados, sabendo como referenciar o lado destinatário.

- **Sistemas difusos**:

  A intenção é que estes sistemas estejam misturados em nossos ambientes: smartphones.

  A separação entre usuário e sistema é um pouco nebulosa. Este sistema é equipado de sensores que captam os comportamentos dos usuários e possuem uma caralhada de atuadores providenciando informações e feedbacks.

  Sistemas de computação ubiquo:

  - O usuário estará interagindo com o sistema continuamente.
  - Requerimentos:

- *Distribuição*:

  Dispositivos e os outros computadores que formam os nós estão todos conectados, os dispositivos podem estar próximos, porém as máquinas estarão longe da vista, talvez até operando remotamente.

- *Interação*:

  Usuários finais tem um papel importante no sistema, é necessária uma atenção especial a como os usuários interagem com o sistema. A interação entre o usuário e o sistema é através de uma ação implícita.

- *Consciência do contexto*:

  Consciencia da situação(localização, identidade, tempo e atividade) em que as interações estão sendo feitas.

- *Autonomia*:

  O sistema consegue tomar decisões e reagir de maneira autônoma e automática.

- *Inteligência*:

  Usam métodos e técnicas do campo da Inteligência Artificial (para lidar com entradas incorretas, reagir rapidamente a mudança de ambiente, lidar com decisões inesperadas e etc).

Sistemas de computação móvel:

- Implicância de wireless.
- Mudança constance: necessária localização geográfica, mas também pode ser necessário localizar a posição da rede (IP).

Redes de sensores:

- A rede consiste de 10 a centenas ou milhares de pequenos nós equipados com um ou mais dispositivos de sensores.
- Muitas destas redes utilizam comunicação wireless e os nós são muitas vezes carregados a bateria.

Capítulo 2: Arquiteturas

Estilos Arquiteturais:

- Os estilos arquiteturais são definidos em forma de componentes, o jeito que eles são conectados uns aos outros, os dados trocados entre componentes e como esses elementos são configurados em união em um sistema.

- Um **componente** é uma unidade modular com interfaces requisitadas e fornecidas de uma maneira bem definida, as quais podem ser trocadas dentro de um ambiente.
- Um **conector** é um mecanismo que media comunicação, coordenação ou cooperação entre componentes.
- **Arquitetura de camadas**:
  - Os componentes são organizados por camadas, o componente da camada Lj faz uma chamada para baixo para o componente da camada Li (i < j) e geralmente espera uma resposta.
  - Organização de camadas pura:

    Apenas chamadas para baixo.
  - Organização de camadas misturadas:

    Apenas chamadas para baixo, porém utiliza outras aplicações externas.
  - Organização de camadas utilizando chamadas para cima:

    Uma camada de baixo nível lida com chamadas para cima.
  - Protocolo de comunicação de camadas:

    Protocolo a ser usado na comunicação entre camadas iguais: TCP, UDP
  - Camadas da aplicação:

    Nível de interface da aplicação: Lida com a interação do usuário ou aplicação externa.

    Nivel de processamento: Contém o núcleo da funcionalidade da aplicação.

    Nível de dados: Opera em um banco de dados ou sistema de arquivos.
- **Arquiteturas baseado em objetos e orientados a serviços**:
  - Os componentes sãop os objetos que se conectam através de chamadas de procedimento.
- **Arquiteturas baseado em recursos**
  - Sistema distribuído como uma grande coleção de recursos, tais recursos podendo ser facilmente adicionadas ou removidos por componentes.
  - REST

    Identificação única dos recursos através de um esquema de nome simples.

Todos os serviços oferecem a mesma interface, consistindo ao máximo de quatro operações: PUT, GET, POST, DELETE

As mensagens enviadas para ou de um serviço são totalmente autodescritivas.

Após executar uma operação em um serviço, o componente esquece de tudo o que fez.

- **Arquiteturas de publicação e subscrição**
  - Ver o sistema como uma coleção de processos operando autonomamente.
  - Coordenação, nesse modelo, engloba a comunicação e cooperação entre processos
  - Tipos de coordenação:

    Direta: Processos acoplados referencialmente e temporariamente. (comunicação explícita, sabendo o nome)

    Caixa de email: Processos temporariamentes desacoplados, mas referencialmente acoplados.

    Baseado em evento: Processos referencialmente desacoplados, mas acoplados temporariamente. (publicar uma notificação)

    Espaço de dados compartilhados: Processos referencialmente e temporariamente desacoplados. (registros de dados)

Organização do middleware

- Duas formas de organização: Wrappers e Interceptos.
- Objetivo: Conquistar abertura.
- **Wrappers**:
  - Oferece uma interface aceitável à aplicação cliente.
  - Resolve o problema de interfaces incompatíveis.
  - Utiliza a extensibilidade para atingir a abertura
  - Muitos wrappers: problema de escalabilidade

    Fazer um Wrapper específico para B acessar os dados de A

    $O(N^2)$: N x (N - 1)
  - Solução: Utilizar Broker

    Componente centralizado que lida com todas as comunicações entre diferentes aplicações.

AS aplicações enviam as mensagens ao broker sobre o que elas precisam e o broker, tendo total conhecimento sobre as aplicações relevantes, contata as aplicações apropriadas.

Reduz o número para até 2N Wrappers (1 wrapper de contato entre aplicação e broker e vice-versa) = O(N).

- **Interceptors**:
  - Conceitualmente, um interceptor é uma construação de software que irá interromper o fluxo usual de controle e permitir outros códigos a serem executados.
  - Adapta o middleware para as necessidades específicas da aplicação.
  - Interceptors em sistemas distribuídos baseados em objetos:

    A chama um método de B que reside em uma outra máquina:

    - Objeto A oferece uma interface local exatamente igual a interface do objeto B.
    - A chamada pelo A é transformada em uma invocação genérica de objetos, isto é possível através da interface de invocação genéria de objeto oferecida pelo middleware onde A reside.
    - Por último, a invocação genérica de objeto é transformada em uma mensagem e enviada através da interface de rede do nível de transporte oferecido pelo S.O de A.
- Middleware modificáveis:
  - Wrappers e Interceptors são só maneiras de adaptar o middleware.
  - Mais do que deixar as aplicações reagirem as mudanças, essa tarefa é do middleware.
  - Fazer mudanças sem desligar tudo
  - Requer uma atenção muito especial.

Arquitetura de sistemas

- **Organização centralizada**:
  - Pensar em termos de clientes que fazem requisições ao servidores ajuda a entender e gerenciar sistemas complexos.
  - Arquitetura cliente-servidor simples:

    Cliente simplesmente empacota a entrada e envia para o servidor

Solucionar problema de mensagens perdidas: TCP/IP (apesar de ser bem ineficiente para conexões locais, em conexões não locais é muito eficiente).

Problema: como fazer uma distinção clara entre cliente e servidor?

- *Arquitetura multicamadas*:

  Distribuir os 3 níveis lógicos através de várias possibilidades numa aplicação cliente-servidor.

  Apenas dois tipos de máquinas:

  - Máquina cliente contendo apenas programas implementando (parte da) nivel de interface do usuário.
  - Máquina servidor contendo o resto, os programas implementado a parte de processamento e de dados.

  Arquitetura de duas camadas (fisicamente):

  - Cliente só possui um terminal dependente e parte da interface de usuário e a aplicação possui o controle remoto sobre a apresentação dos dados.
  - Toda a parte de interface do usuário reside no cliente, no cliente só há a partte de processamento dos dados a serem exibidos, a aplicação fica com o resto.
  - Cliente possui além da interface do usuário, uma parte da aplicação, como no caso da verificação de um formulário antes da sua validação.
  - Cliente possui total domínio sobre a interface e a aplicação e utiliza o servidor apenas para consulta no banco de dados. (banco)
  - Cliente utiliza além da interface e aplicação, parte do banco de dados, contruindo uma enorme cache de dados. (páginas da web recém-visitadas)

  Arquitetura de três camadas (fisicamente):

  - Programas que formam parte da camada de processamento são executadas por um servidor diferente, agindo assim como cliente. (sistema de monitoramento de transações)

- **Arquiteturas descentralizadas: Sistemas P2P**
  - Cliente e servidor são partes logicamente equivalentes, mas cada parte opera por sua própria conta.
  - Interação entre processos é simétrica.

- *Sistemas P2P estruturados*:

  Os nós (processos) são organizados em uma overlay que adere a uma topologia específica: anel, árvore binária, grade e etc para eficientizar a procura dos dados.

  Chave(item do dado) = Hash(valor do ítem do dado).

  A cada nó é atribuído um identificador para o mesmo conjunto dos valores de hash possíveis e cada nó é responsável por armazenar um dado associado a um conjunto específico de chaves. (DHT)

  Nó existente = achar(chave)

  Topologia é importante: Qualquer nó pode ser pedido para achar a chave dada.

- *Sistemas P2P destruturados*:

  Cada nó mantém uma lista de vizinhos

  Grafo aleatório: Uma aresta entre os nós u e v, existe com probabilidade P[{u,v}], que é a mesma para todas as arestas.

  As listas mudam constantemente.

  Dois modos de busca:

  - Flooding:
    - Um nó u simplesmente passa a requisição do dado para todos os seus vizinhos.
    - A mensagem é ignorada em caso do nó que possua o dado já foi visto
    - O nó que recebe a requisição olha internamente para checar os seus dados, caso tenha ele pode responder de duas maneiras:

      Reportar diretamente para o nó u

      Mandar de volta para quem o requisitou e este mandar até o u.
    - Caso não possua o dado, o nó requisitante repassa a requisição para todos os seus vizinhos.
    - A requisição pode ter um TTL (time to live), para evitar custo muito alto de flooding.
  - Random Walks:
    - Um nó u manda uma requisição para um nó aleatório v da sua lista de vizinhos, caso este não possua, ele faz o mesmo procedimento.

        - Muito lento, ao invés de 1 em 1, pode utilizar de n passos aleatórios concorrentes.
    - *Redes P2P organizadas hierarquicamente*:

      Super peer: nó que mantém uma lista de índices ou que haja como um broker.

      Nós comuns, agora chamados de peers fracos, conectam-se a algum super peer.
- **Arquiteturas Híbridas**
    - Class de sistemas distribuídos que combinam soluções de cliente-servidor com arquitetura descentralizada.
    - *Sistemas servidor de borda:*

      Sistemas são lançados “na beirada” da rede, tal beirada é formado na fronteira entre redes corporativas e a Internet (ISP).

      Sistemas de beirada são usados para ajudar na computação em nuvem.
    - *Sistemas colaborativos distribuídos:*

      Uma vez que um nó junta-se ao sistema ele pode usufruir totalmente do esquema descentralizado para colaboração.

      BitTorrent

      Garantir a colaboração. No caso do BitTorrent, forçar usuário a dar up nos arquivos ao invés de só baixar.

Examplo de Arquiteturas:

- **A rede de sistemas de arquivos**:
    - Cada servidor de arquivos providencia uma maneira padronizada de visualização.
    - Aos clientes é oferecido total transparência no acesso aos arquivos, entretanto, eles não percebem a atual localização dos arquivos.
- **A internet**:
    - *Sistemas baseados na web simples*:

      Arquiteturas simples de cliente-servidor.

      Cliente faz uma requisição através do browser para o servidor web, que processa essa requisição e a retorna para o cliente.
    - *Arquiteturas multicamadas*:

Utiliza um programa CGI (Common Gateway Interface) para buscar os dados e criar o html, repassando para o servidor que ao processar o html pré-renderizado pelo CGI, retorna ao cliente.

Capítulo 3: Processos:

Threads:

- Granularizar processos no memso formato que sistemas operacionais fazem nao é suficiente, porém fazer isso com as threads para controlar processos mostrou-se eficiente.
- *Contexto de processo:* Todas as informações sobre um processo.
- *Contexto de thread:* Apenas as informações do processador e outras para gerenciamento de thread.
- Uso de múltiplas threads traz muitos ganhos, porém é necessário todo um esforço intelectual.
- Uso de threads em sistemas não-distribuídos:
  - Sistema simples de 1 thread por processo: quando um bloco de sistema é executado o processo inteiro é bloqueado. (vantagem)
  - Sistemas multithreads ajudam no melhor uso de recurso, cada thread por núcleo (vantagem)
  - Implementação de threads:

    Construir uma lib de threads para o espaço de usuário:

    - Barato para criar e destruir threads.
    - Trocas de contexto podem ser feitas com poucas instruções.
    - Desvantagem em sistemas muitas threads para um processo, uma chamada de sistema bloquearia todas as threads pertencentes ao processo e outras threads neste processo, o que não deveria prevenir outras threads de executar.

    Deixar o kernel avisado das threads e alocá-las:

    - Todo o processo de criar e destruir envolve uma chamada de sistema que é mais custoso.
    - Troca de contexto entre threads pode ser mais custoso do que troca de contexto entre processos.

        - Problemas com o uso de libs pode ser lidado com o uso de uma thread por processo no kernel.
- Threads em sistemas distribuídos:
    - Permite o bloqueio de chamada de sistema sem bloquear todo o processo que a thread está rodando.
    - Clientes multithreads:

      Em contexto WEB, ao receber um arquivo HTML e ter algo a mostrar para o usuário, o browser configura várias threads para disponibilizar pedaços do arquivo, assim o usuário já tem uma resposta e o arquivo vai sendo carregado a medida que as threads vão terminando.

      Outro benefício, se o servidor recebe muita requisição (mesmo sendo multithread) o ganho anterior é minimo pela sobrecarga que há, porém com a replicação da página em vários servidores web apontando para o original faz browsers multithreads terem um ganho enorme, apontando as threads para diferentes réplicas.
    - Servidores multithreads:

      Em modelo de despachante/trabalhador, uma thread despacha uma requisição para threads trabalhadoras, enquanto uma faz uma requisição e espera pelo resultado (ficando assim ociosa e suspensa), outra thread trabalhadora pode assumir e ficar rodando, aumentando assim o ganho de perfomance.

      Em um sistema de thread simples em servidor resultaria em uma demora considerável, já que para cada requisição somente uma thread poderia estar trabalhando e enquanto ela estivesse ociosa nada poderia ser feito ou então quando a cpu ficasse ociosa.

      Outra alternativa: Grande máquina de estado em sistema de threads simples

        - Uma thread recebe a requisição e verifica se pode ser atendida só olhando a memória cache ou precisa olhar o disco.
        - Porém ao invés de fazer uma chamada de sistema, a thread agenda uma operação em disco assíncrona.
        - A thread armazena o status da requisição e vai verificando se não tem trabaho pendente, quando a leitura em disco anterior a thread recebe uma notificação do sistema operacional, no meio tempo ela olha a resposta e processa a resposta, fazendo uma chamada de sistema sem bloqueio para devolver a resposta ao cliente.

Multithread: Parelismo e chamadas de sistema com bloqueio (+- pra programar)

Threads simples: Sem paralelismo e chamadas de sistema com bloqueio (fácil)

Máquina de estado finita: Paralelismo e chamadas de sistema sem bloqueio (bem dificil)

Virtualização:

- Virtualização de recurso: Separação entre ter uma simples CPU e poder parecer que tem mais para estender os recursos.
- Princípio de virtualização:
  - Virtualização e sistemas distribuídos:

    Virtualização ajuda muito na redução da diversidade de plataformas e máquinas, deixando que cada aplicação rode em sua própria máquina virtual. (portabilidade e flexibilidade)

    Tipos de virtualização:

    - Sistemas oferecem 4 tipos de interface em três diferentes níveis:
      - Interface entre Software e Hardware

        Instruções privilegiadas, apenas o sistema operacional pode usar.

        Instruções comuns, qualquer programa pode usar.
      - Interface contendo chamada de sistema, oferecida pelo sistema operacional
      - Interface constituida por uma biblioteca de chamadas, API.
    - A essência da virtualização é imitar o comportamento das interfaces acimas.
    - Virtualização pode funcionar de duas maneiras:
      - Sistema essencialmente possuindo um conjunto de instruções abstratas que é para ser usada por aplicações. As instruções pode ser interpretadas (como no caso do JRE) ou emuladas para Windows/UNIX. (Máquina virtual de processo)
      - Outra abordadem é construir uma camada de blindagem em cima do hardware, mas oferecendo um conjunto de instruções completas do hardware como uma interface (Monitor de máquina

virtual nativa). Este monitor tera que providenciar e regular acessos a vários recursos.

- Ao invés de fazer todo o esforço do Monitor de máquina virtual nativa, um Monitor de máquina virtual hospedado irá rodar no topo do sistema operacional, podendo fazer uso das várias facilidades providenciadas pelo sistema operacional.

- Virtualização ajuda a isolar completamente a aplicação e seu ambiente, fazendo com que falhas não afetem completamente a máquina.

- Aplicação de máquinas virtuais para sistemas distribuídos:
  - Reside principalmente em computação em nuvem, que oferece:

    IaaS

    PaaS

    SaaS

  - Papel principal em IaaS, clientes irão pensar que estão comprando/alugando uma máquina física inteiramente dedicada, porém estão dividindo os recursos de uma máquina com outros clientes. (completo isolamento)

Clientes:

- Rede de interfaces de usuário:
  - Oferecer meios dos usuários se comunicarem com servidores remotos:

    Para cada serviço remoto o cliente terá um mieo específico de contatar o servidor através da rede.

    Oferecer um serviço genérico para contato remoto direto, utilizando uma interface conveniente.

- Software do lado do cliente para distribuição transparente:
  - Geralmente é lidado com um stub do cliente a partir de uma interface definida.
  - Esse stub é a mesma interface oferecida pelo servidor, porém esconde possíveis diferenças entre arquiteturas de máquinas e a comunicação.

Servidores:

- Problema de design geral:
  - Um servidor é basicamente um serviço implementado para vários clientes, sendo que cada servidor é organizado de maneira semelhante: ele espera

por uma requisição do cliente e garante que a requisição será entregue de volta, após isso espera um próxima requisição.

- Servidores interativos versus concorrentes:

  Servidores interativos:

  - O servidor que lida com a requisição e se preciso, devolve a requisição para o cliente requisitante.

  Servidor concorrente:

  - Ele não lida com a requisição em si, ele pega a requisição e direciona para alguma thread separada ou processo e espera pela próxima requisição.
  - A thread que lida com a requisição é quem devolve a resposta para o cliente.
  - Um servidor multithread é um servidor concorrente.

  Contactando um servidor: pontos finais

  - Todas as requisições dos clientes são feitas nos pontos finais (também chamadas de portas onde o servidor está rodando).
  - Como os clientes sabem os pontos finais de um servidor?
    - Uma abordagem é atribuir os pontos a serviços já conhecidos, como o FTP usa a porta 21 e HTTP a 80.
  - Com as portas os clientes só precisam saber o endereço de onde está rodando o servidor, serviços nomeados servem para este propósito.
  - Muitos serviços não tem um ponto final pré atribuído.
    - Uma solução é ter um daemon especial rodando em cada máquina do servidor. Ele mantém o rastro de cada ponto final de cadda servidor implementado no servidor co-alocado. Um cliente primeiro contata o daemon, requisitando o ponto final, e depois contata o servidor específico.
    - Outra solução é ter um super servidor ouvindo pra cada ponto final associado a um serviço específico. Quando uma requisição do cliente vem, o super servidor cria um servidor/processo para lidar com a requisição, este processo vai sair quando terminar a requisição.

  Interrompendo um servidor:

  - Como interromper um servidor?

        - O usuário sair abruptamente da aplicação cliente e reiniciar, passando a sensação de que nada aconteceu.
        - Outra abordagem é desenvolver no cliente e servidor uma possibilidade de mandar um fora da banda, que é um dado que é processado pelo servidor antes de qualquer coisa.

            Deixar o servidor ouvindo em um ponto final esta requisição e ouvindo, com baixa prioridade, em outro ponto final a requisição por onde os dados passam.

            Deixar passar o fora de banda na mesma requisição e conexão que é mandado os dados.

    Servidores com estado versus sem estado

    - Sem estado:
        - Não mantém nenhuma informação sobre o estado do cliente e consegue mudar seu próprio estado sem notificar nenhum cliente. (Servidor web)
        - O servidor tem a informação sobre o estado, mas esta informação é perdida.
        - Estado macio: mantém o estado do cliente, mas por um período de tempo, após esse tempo, o servidor “volta atrás” para o seu comportamento padrão.
    - Com estado:
        - Mantém informação persistente do cliente, ou seja, essa informação precisa ser explicitamente deletada pelo servidor. (servidor de arquivo que mantém cópias deste arquivo)
        - Se o servidor quebrar, terá que recuperar toda a sua tabela de informação, caso contrário não pode garantir informação atualizada. (dificil)

- Servidores objeto:
    - Não oferece serviço específico, oferece apenas meios de invocar objetos locais baseados nas requisições remotas do cliente.
    - Um servido de objetos age como um lugar onde os objetos residem.
    - Objetos são compostos de duas partes:

        Dados representando seu estado.

        Código para executar os seus métodos.

- Formas de invocar um objeto:

  Abordagem simples: Assumir que todos os objetos são parecidos e que há apenas um jeito de invocar um objeto. (abordagem geralmente inflexível)

  Melhor abordagem: Servidor suportar diferentes políticas:

  - Objeto transiente: Existe apenas enquanto o servidor existir, mas possivelmente por um período curto de tempo.
  - Desvantagem: ao criar um objeto pode levar um tempo, já que ele irá precisar dos recurcos do servidor.
  - Uma política plausível: as vezes instanciar todos os objetos transientes ao inicializar o servidor, mesmo que nenhum cliente esteja usando os objetos.

  Políticas de ativação: decisões de como invocar objetos

  Adaptador de objeto (object adapter/wrapper): agrupar objetos por política

  - Software implementando uma política de ativação específica.
  - São genéricos, vem apenas pra dar um help aos desenvolvedores. Necessita de configuração para uma política específica.

  Requisição cai no servidor, servidor repassa para o adaptador de objeto correto (existem vários diferentes no servidor) que então repassa para stub do objeto, no lado do servidor, invocar o método correto. Este stub é gerado a partir das definições da interface do objeto.

  Servo: pedaço de código que forma a implementação do objeto.

- Exemplo: Servidor Web Apache
  - Assume que todas as requisções vêm em TCP
  - Hook: Lugar que contém um grupo específico de funções.
  - As requisições são tratadas em fases, cada fase possuindo alguns hooks
  - Cada hook representa um grupo similar de ações que precisam ser ativadas como parte de processamento de uma requisição.
- Agrupamento de servidores:
  - Coleção de máquinas conectadas através da rede, onde cada máquina roda um ou mais servidores.
  - Clusters locais:

Clusters conectados via LAN, frequentemente oferecendo alta largura de banda e baixa latência

Organização geral:

- Geralmente é organizado em três camadas:
  - Interruptor lógico: onde as requisições dos clientes são roteadas.
  - Servidores rodando em alto potencial dedicados apenas à aplicação.
  - Servidores de processamento de dados, arquivos e banco de dados.
- As vezes a formação é em duas camadas, com as duas últimas camadas citadas acima sendo unidas em uma só. (serviço de streaming)
- Despachante de requisições (primeira camada):
  - Parte front-end
  - Oferece o ponto de entrada para o cluster de servidores, oferecendo um endereço de rede único.
  - Camada de transporte do interruptor:

    Um jeito: Configura uma conexão TCP para que todas as requisições e respostas passem por ele, então configura conexão TCP com o servidor selecionado para que a requisição do cliente seja despachada para este servidor e sua resposta seja repassada ao cliente. (Age como NAT)

    Outro jeito: As requisições passam por ele, porém a resposta vai do servidor diretamente ao cliente. (TCP handoff)

    Papel do interruptor é essencial na decisão de qual servidor irá lidar com a requisição.

- Clusters de área ampla (WAN):

  Problema de lidar com várias organizações pelo espalhamento de servidores é facilmente lidada com a computação em nuvem. (usando Amazon, Google,...)

  Despachante de requisições:

  - Escolher servidor mais próximo ao cliente.
  - Política de redirecionamento: escolher o servidor mais próximo.

        - Após selecionar o servidor, o expedidor (interruptor) terá que informar o cliente.
        - Serviços de redirecionamento:
            - DNS

                Nem sempre muito acurado, porém é fácil de implementar e transparente para o cliente

- Migração de código:
    - Em sistemas distribuídos, migrar código toma forma de migração de processos.
    - Razões para migrar código:

        Mover um processo de um lugar sobrecarregado para um lugar mais tranquilo, o desempenho é aumentado.

        Flexibilidade: particionamento da aplicação.
    - Migrações em sistemas heterogêneos:

        Lidar com Sistemas Operacionais e arquiteturas diferentes.

        Portabilidade: Java e Pascal

Capítulo 4: Comunicação

Fundações:

- Camadas de protocolos:
    - Comunicação se baseia em baixo nível
    - Modelo OSI:

        Desenhado para permitir que sistemas abertos se comuniquem.

        Dois tipos de serviço de conexão:

        - Conexão orientada ao serviço: Antes da troca de dados há um estabelecimento da conexão entre as duas partes
        - Serviços sem conexão: Não há um estabelecimento de conexão, só enviar a mensagem. (jogar um email na caixa de correio)
        - Camadas:
            - Física: bits

        - Link de dados: trata os erros das mensagens
        - Rede: repasse de mensagem
        - Transporte: estabelecimento da comunicação
        - Sessão: suporta sessão entre as aplicações
        - Apresentação: Descreve como são os dados
        - Aplicação: Qualquer outra coisa que sobrar.
      - A mensagem sai da aplicação com o header da aplicação e vai descendo, cada camada que passa adiciona seu header, a mensagem que chega vai tendo seus headers abertos pelas camadas correspondentes.
  - Protocolos de Middleware:

    DNS: associa um endereço pelo nome. (camada de aplicação)

    Protocolo de autorização.

    Protocolos de commit

    Protocolo de trancamento. (quando muitos processos tentam acessar um recurso)

    Middleware pega as camadas de sessão e apresentação e as une em uma só.

- Tipos de comunicação:
  - Persistente:

    A mensagem é armazenada pelo middleware enquanto for necessário para a sua transmissão.

  - Transiente:

    A mensagem é armazenada enquanto quem enviou e o destinatário estiverem rodando, caso contrário será descartado.

  - Assíncrona:

    Quem enviou segue a vida após a submissão, mesmo ela não tendo sido aceita.

  - Síncrona:

    Quem enviou é bloqueado enquanto a mensagem não foi aceita.

Chamada de procedimento remoto (RPC)

- Processo em uma máquina A chama um procedimento na máquina B, o processo de A fica suspenso e execução do procedimento toma lugar em B.
- A e B estão em máquinas, espaços de endereços e arquiteturas diferentes.
- Básica operação RPC:
  - Fazer uma chamada remoto parecer o máximo possível como uma chamada local: ser transparente.
  - Stub do cliente: empacota os parametros na mensagem e requisita envio para o servidor.
  - Stub do servidor: transforma a requisição em procedimento local
  - Passo a passo RPC:

    Procedimento de chamada do cliente chama o stub do cliente normal

    Stub do cliente constroi a mensagem e chama o sistema operacional local

    S.O do cliente chama S.O remoto

    S.O remoto passa mensagem pro stub do servidor

    Stub do servidor desempacota e chama o servidor

    Servidor faz seu trabalho e retorna o resultado pro stub

    Stub do servidor empacota a mensagem e chama S.O local

    S.O do servidor manda mensagem pro S.O do cliente

    S.O do cliente manda mensagem pro stub

    Stub desempacota a mensagem e passa pro cliente
- Passando parâmetro:
  - Empacotamendo de parâmetro (marshaling) para formatos neutros
  - ARM: big endian, Intel: little endian.
  - Transformar os dados para e de formatos independentes da rede e máquina
- Suporte de aplicação baseado em RPC:
  - Esconder um procedimento remoto requer que quem chamou e o destinatário acordem no formato da mensagem e que eles seguiram os mesmo passos em envios de dados complexos.
  - Ou seja, seguir o mesmo protocolo RPC:

    Um jeito:

        - Deixar o desenvolvedor especificar exatamente o que é preciso ser chamado remotamente, para quais lados cliente e servidor os stubs podem ser gerados

      Outro:

        - Deixar o RPC embutido em uma linguagem de programação

    - Geração de stubs:

      Necessário que cliente e servidor acordem sobre o formato/tipo da mensagem transmitida: encoding

      Acordar se usa TCP/IP, ou um serviço de datagrama não confiável para controle de erros como parte do protocolo RPC.

      Stubs para a mesma mensagem mas procedimentos diferentes, diferem apenas em suas interfaces.

      Interfaces em ambos os lados sobre o mesmo domínio de linguagem.

    - Suporte baseado em linguagem

      Muito mais simples.

      Libs mais conhecidas: RMI em java

  - Variações em RPC:

    - Quando o cliente faz uma RPC, ele é bloqueado, porém quando não há respostas a mostrar o bloqueio se torna desnecessário.
    - RPC assíncrono:

      Servidor responde ao cliente imediatamente após receber sua requisição.

      Com isso o cliente não fica bloqueado.

      Servidor responde antes de processar.

      Após ter uma resposta o servidor devolve um callback no lado do cliente

    - RPC multicast:

      Enviar um RPC pra um grupo de servidores e esperá-los processar para ter uma resposta e continuar

      A aplicação do cliente pode estar desavisada com o RPC sendo enviado para mais de um servidor

      Fica a gosto do desenvolvedor quando deixar o cliente continuar: após a primeira resposta, todas as respostas ou a maioria das respostas.

Comunicação orientada a mensagem (MOM):

- Trocar o bloqueio de resposta do cliente por mensagens
- Mensagens transientes simples com sockets
  - socket: Ponto final de comunicação o qual a aplicação pode escrever dados que são mandados para fora da rede e por onde dados que estão entrando podem ser lidos.
  - Operações do socket:

    socket: cria um novo ponto final

    bind: associa um endereço local ao recém criado socket. (IP local + porta)

    listen: Deixa o S.O saber o quanto de buffer reservar para ter um número máximo de requisições a aceitar.

    Accept: bloqueia quem chamou até a requisição de conexão chegar.

    Connect: tenta estabelecer uma conexão

    send e receive: enviar e receber dados

    close: deixar a conexão.
- Mensagens transientes avançadas:
  - Usando padrão de mensagens: ZeroMQ

    Várias aplicações de mensagens podem ser organizadas em padrões de comunicações para assim oferecer sockets mais "detalhados" a estes padrões.

    Comunicação assíncrona

    Pode tentar realizar uma conexão e enviar uma mensagem, mesmo que o destinatário não esteja rodando e pronto para receber a mensagem: fila de mensagem.

    Padrão requisição-resposta:

    - Cliente usa socket requisição
    - Servidor usa socket resposta
    - Retira a necessidade do servidor usar as operações listen a accept.

    Padrão publicar-subescrever:

    - O cliente subescreve a uma específica mensagem publicada pelo servidor
    - Mensagens em multicast, através de uma mensagem enviar para N clientes.

        - Se ninguém subescrever, a mensagem é perdida.

        Padrão pipeline:

        - Caracterizado pelo fato de que um processo que empurrar os resultados enquanto que outros processos querem puxá-los.
        - Empurrar não ligar para o puxar.
        - O puxar liga para o empurrar, primeiro empurra depois puxa.

    - A interface de passeio de mensagem:

        Designado para aplicações paralelas.

        Falhas e crash são fatais e não tem recuperação

        Operações:

        - MPI_bsend: adicionar uma mensagem que tá saindo para um buffer local
        - MPI_send: Mandar uma mensagem e esperar até que seja copiada pra um buffer local
        - MPI_ssend: Mandar uma mensagem e esperar até que a transmissão comece
        - MPI_sendrecv:Mandar e esperar por resposta
        - MPI_isend: Passar referencia para a mensagem que tá saindo, e continuar
        - MPI_issend: Passar referencia para a mensagem que tá saindo e esperar até que o recipiente comece.
        - MPI_recv: receber mensagem, bloquear se nao tem nenhuma
        - MPI_irecv: verificar se tem mensagem, mas não bloquear

- Comunicação persistente orientado a mensagens:
    - Middleware orientado a mensagens (MOM)
    - Modelo de enfilamento de mensagem:

        Inserir mensagens a filas especifícas, consequentemente essa mensagens serão entregues.

        Garantia de enfileiramento de mensagens, porém não há a respeiro de leitura, o que é determinado pelo comportamento do recipiente.

        Entregador e Recebedor se comportam de maneiras totalmente independentes, pois tem a fila como intermediário.

4 combinações de modelo:

- Servidor x Cliente | Passivo x Rodando
- Quem está rodando manda/recebe mensagem, que está passivo não recebe/envia mensangem

Operações:

- Put: Adicionar uma mensagem a uma fila específica. (não bloqueia)
- Get: Bloqueia até que a fila esteja não-vazia e remove a primeira mensagem.
- Poll: Verifica uma fila especifica por mensagem e remove a primeira mensagem (nunca bloqueia)
- Notify: Instalar um carinha para lidar quando uma mensagem é colocada em uma fila específica.

- Arquitetura geral de um sistema de enfileiramento de mensagem:

  A aplicação só coloca as mensagens em filas locais.

  Como endereçar o destinatário nessas filas.

  Mapeamento realizado para <host, porta> através de nomes

- Brokers de mensagens:

  Quem envia e quem recebe devem falar a mesma lingua: mesmo protocolo de mensagem

  Cada aplicação com um novo protoclo gera um aprendizado em todas as outras, logo, precisaria de um N x N conversores

  Conversor: broker

  Traduz mensagens que estão saindo para que o destinatário entenda.

- Comunicação multicast: