**Tema 10 - Arquitetura de sistemas distribuídos**

Um sistema distribuído é formado por uma coleção de computadores independentes que operam conjuntamente para se apresentar como uma única entidade coerente aos usuários. Este arranjo envolve diversos elementos significativos: primeiramente, os componentes do sistema são computadores autônomos que variam desde mainframes até nós em redes de sensores, sem premissas específicas sobre o tipo de hardware ou a natureza da conexão entre eles.

Sistemas distribuídos operam em computadores e redes autônomas, o que implica que diferentes partes do sistema podem funcionar em redes distintas, em hardwares diferentes, com sistemas operacionais variados e até linguagens de programação diferentes. Os protocolos de rede facilitam a comunicação entre essas entidades computacionais de maneira transparente. Essa transparência é essencial em sistemas distribuídos, pois deve ocultar dos usuários a distribuição física ou virtual das aplicações, fazendo com que pareçam operar como um único sistema coerente. Sem essa característica, o acesso ao sistema distribuído seria complexo. Além disso, sistemas distribuídos frequentemente enfrentam situações onde múltiplos acessos concorrentes a um mesmo recurso ocorrem simultaneamente.

Além disso, é importante que os usuários e aplicações interajam de forma consistente e uniforme com o sistema, independentemente da localização física da interação. Para facilitar a integração de computadores e redes heterogêneas e manter uma visão unificada do sistema, os sistemas distribuídos são frequentemente estruturados através de uma camada de software conhecida como middleware, exibida na Figura 1. Esta camada fica posicionada logicamente entre a camada de aplicação, que inclui usuários e suas aplicações, e a camada de infraestrutura, que abrange os sistemas operacionais e as funcionalidades básicas de comunicação. Essa organização permite que o sistema distribuído funcione de maneira integrada e eficaz, ocultando a complexidade subjacente aos usuários.

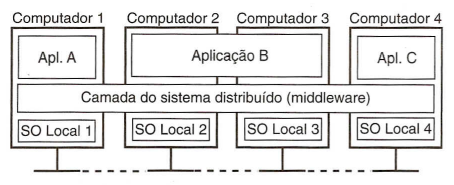


Figura 1: Sistema distribuído organizado como middleware. A camada de middleware se estende por várias máquinas e oferece a mesma interface a cada aplicação.

Nesse contexto, a arquitetura de sistemas distribuídos envolve tanto a organização dos componentes de software quanto a sua realização física nas máquinas. Fundamentalmente, a arquitetura de software de um sistema distribuído define como os componentes de software interagem e são posicionados em máquinas específicas, resultando na arquitetura final do sistema.

Estilos arquitetônicos são definidos considerando os componentes, suas conexões, a troca de dados entre eles e como esses elementos são integrados para formar um sistema completo. Um componente é uma unidade modular com interfaces de entrada e saída claramente especificadas, permitindo sua substituição sem impactos ao ambiente. Um conector, por sua vez, facilita a comunicação ou cooperação entre componentes, sendo essencial para a integração do sistema. Dentre os estilos arquitetônicos mais relevantes para sistemas distribuídos, destacam-se:

1. **Arquitetura em Camadas:** os componentes do software são divididos em camadas hierárquicas, de tal modo que cada camada tenha um propósito bem definido e que o acoplamento entre elas seja baixo. Isso significa que a camada superior conhece apenas a camada imediatamente inferior, que fornece serviços através de uma interface. Um tipo específico de arquitetura em camadas muito comum é chamado de arquitetura de 3 camadas, que divide a aplicação nas camadas de (1) apresentação, responsável pela interface com o usuário; (2) negócio, responsável pelas regras de negócio da aplicação; (3) dados, responsável pela gestão dos dados.

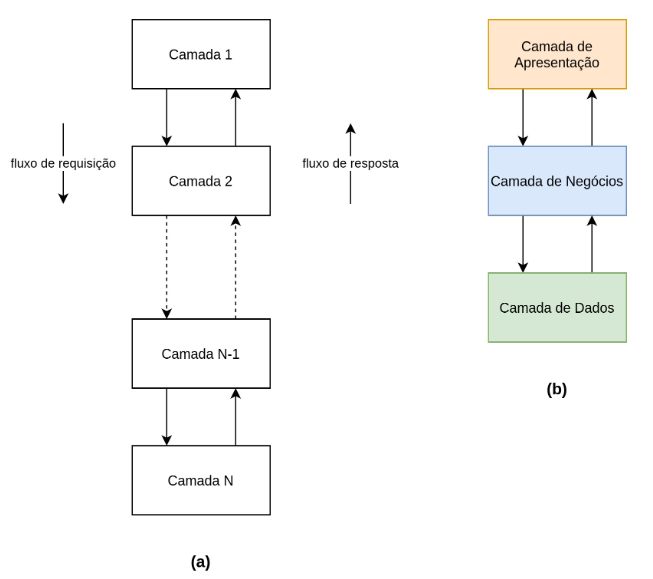


Figura 1:(a) Arquitetura com N camadas. (b) Arquitetura de 3 camadas.

1. **Arquitetura Baseada em Objetos:** Nesta abordagem, também conhecida como orientada a serviços, os componentes são tratados como objetos que interagem por meio de chamadas de procedimento remoto. Esta arquitetura é flexível e permite a comunicação entre objetos sem regras rígidas de interação.

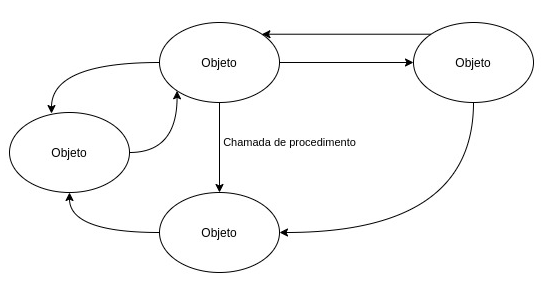


Figura 2: Arquitetura baseada em objeto

1. **Arquiteturas Centradas em Dados:** Caracteriza-se por um espaço de dados compartilhado que os componentes utilizam para interagir. Exemplificando, a Web utiliza arquiteturas centradas em dados, onde o protocolo HTTP e seus verbos (GET, POST, PUT, DELETE) manipulam arquivos de hipermídia. O serviço Amazon S3 é um exemplo de plataforma que suporta esta arquitetura, otimizando o armazenamento e recuperação de dados.

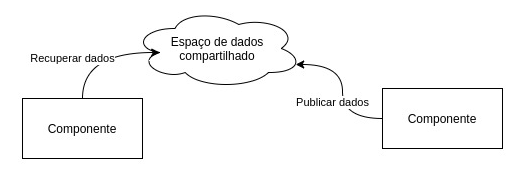


Figura 3: Arquitetura centrada em dados

1. **Arquiteturas Baseadas em Eventos:** nesse estilo arquitetural, os processos se comunicam através de eventos. Um evento é qualquer alteração de estado importante para o domínio do sistema. Quando um sistema detecta um evento, ele pode representá-lo em forma de mensagem e publicá-lo em um canal para que outros sistemas possam reagir a ele. O sistema que publica a mensagem é chamado de produtor e não necessita conhecer aqueles sistemas que podem reagir aos eventos. Determinados sistemas podem estar interessados em apenas alguns dos eventos, os quais pedem ao canal para serem avisados sempre que um evento desse tipo acontece. Esses sistemas são chamados subscritores de um evento. É comum que um sistema realize os dois papeis. A grande vantagem desse estilo arquitetural é o baixo acoplamento gerado entre os componentes do sistema distribuído.

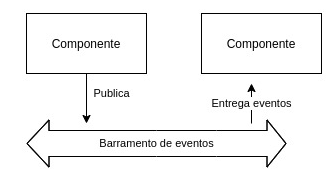


Figura 3: Arquitetura baseada em eventos

As arquiteturas de sistema em sistemas distribuídos são categorizadas em três grupos principais: Centralizada, Descentralizada e Híbrida.

A arquitetura centralizada é um modelo onde os componentes são categorizados em funções de cliente e servidor. O cliente faz solicitações, e o servidor processa essas solicitações. Essa configuração é especialmente prevalente em sistemas que seguem a arquitetura de três camadas, que pode ser configurada de várias maneiras. Por exemplo, a camada de apresentação pode ser hospedada em uma aplicação cliente separada que se comunica com um servidor responsável pela lógica de negócios. Alternativamente, tanto a apresentação quanto a lógica de negócios podem coexistir na mesma aplicação, que por sua vez interage com um servidor de banco de dados para manipulação de dados. Essa estrutura facilita tanto a manutenção quanto a escalabilidade, mas também concentra o processamento, criando um potencial ponto único de falha. A comunicação entre cliente e servidor pode ocorrer de duas formas principais:

1. **Protocolo Simples Sem Conexão:** Neste modo, o cliente envia uma requisição ao servidor empacotando os dados em uma mensagem. O servidor processa a requisição, empacota o resultado e envia de volta ao cliente. Este método é eficiente quando não há perda de mensagens, mas a recuperação de falhas torna-se complexa se a mensagem se perder, especialmente se repetições da mesma operação poderem causar efeitos diferentes.
2. **Protocolo Confiável Orientado a Conexão:** Este protocolo é geralmente utilizado em comunicações de longa distância e baseia-se em conexões TCP/IP confiáveis, onde uma conexão é estabelecida antes de qualquer transmissão de dados. Embora confiável, esse método pode ser custoso, especialmente se as mensagens trocadas forem pequenas.

Considerando que muitas aplicações cliente-servidor visam a dar suporte ao acesso de usuários ao banco de dados, muitos implementam o estilo arquitetônico em camadas:

* **Camada de Interface de Usuário:** Esta camada gerencia toda a interação com o usuário, incluindo exibição de dados e entrada de usuário. É a camada que o usuário vê e interage diretamente.
* **Camada de Processamento:** Contém as operações específicas da aplicação que processam as entradas do usuário, executam a lógica de negócios, e fazem chamadas para a camada de dados para recuperar ou armazenar informações.
* **Camada de Dados:** Esta camada lida com o armazenamento, recuperação e gestão dos dados. Ela opera diretamente com o banco de dados ou outra forma de armazenamento de dados, isolando essas funções das outras camadas.

Essas camadas podem ser distribuídas fisicamente em diferentes máquinas para melhorar a escalabilidade e o balanceamento de carga. Por exemplo, uma configuração comum em arquiteturas de três camadas pode incluir: máquinas clientes, que hospedam apenas a camada de interface do usuário, focando na apresentação e interação com o usuário; e máquinas servidoras, que gerenciam a lógica de negócios e a camada de dados, processando a lógica da aplicação e gerenciando o armazenamento de dados, respectivamente.

As arquiteturas centralizadas, adotam uma distribuição vertical de componentes, posicionando diferentes funções lógicas em máquinas distintas. Essa configuração permite uma clara separação de responsabilidades, com a interface do usuário geralmente gerenciada pelos clientes e o processamento de dados pelo servidor. No entanto, arquiteturas modernas frequentemente exploram a distribuição horizontal, em que tanto clientes quanto servidores podem ser divididos fisicamente em múltiplas unidades que operam de forma logicamente equivalente, cada uma processando integralmente sua porção de dados.

Diferente da abordagem centralizada, a arquitetura descentralizada caracteriza-se por essa distribuição horizontal. Nesse modelo, elementos como servidores ou mesmo as camadas de uma arquitetura de três camadas podem ser replicados em várias máquinas. Essa replicação visa equilibrar a carga de trabalho e aumentar a disponibilidade do sistema. Um exemplo clássico dessa estratégia é encontrado nas redes peer-to-peer (P2P), onde cada nó na rede executa tanto funções de cliente quanto de servidor. Esta abordagem não só facilita uma distribuição equitativa dos recursos computacionais, mas também aumenta a robustez e a tolerância a falhas do sistema devido à sua descentralização.

Além disso, em uma arquitetura descentralizada, a falha de um nó ou componente tem um impacto menor sobre o sistema total, pois os dados e responsabilidades estão distribuídos por várias unidades autônomas. Isso contrasta com sistemas centralizados, onde um ponto de falha pode comprometer todo o sistema. Em termos de escalabilidade, a descentralização permite adicionar mais nós à rede sem necessariamente aumentar a carga sobre os componentes centrais, o que facilita a expansão conforme a demanda cresce.

A descentralização inerente às arquiteturas peer-to-peer (P2P) permite que todos os processos em um sistema distribuído operem de forma equivalente, executando tanto funções de cliente quanto de servidor. Esse modelo é comumente referido como "servente", onde cada processo age simultaneamente como cliente e servidor, resultando em interações simétricas entre processos. Esta simetria é crucial, pois define a estrutura organizacional dos processos dentro de uma rede de sobreposição. Nesta rede, os nós representam os processos e as ligações entre eles são canais de comunicação, frequentemente baseados em conexões TCP, que não permitem comunicação direta com processos arbitrários, necessitando que as mensagens sejam transmitidas através dos canais disponíveis.

Dentro das redes P2P, distinguem-se duas principais arquiteturas: estruturadas e não estruturadas. As arquiteturas estruturadas são caracterizadas por uma organização determinística da rede de sobreposição, comumente utilizando uma tabela de hash distribuída (DHT). Neste sistema, tanto os itens de dados quanto os nós do sistema são identificados por chaves aleatórias. O elemento central dessas arquiteturas é um método eficiente e determinístico que associa a chave de um dado diretamente ao identificador de um nó específico. Quando um item de dado é consultado, o sistema retorna diretamente o endereço de rede do nó responsável por esse dado.

Por outro lado, as arquiteturas P2P não estruturadas dependem de algoritmos aleatórios para estabelecer a rede de sobreposição. Cada nó mantém uma lista de vizinhos formada de maneira mais ou menos aleatória, e os dados são distribuídos nos nós sem uma ordem específica. Isso implica que a busca por dados específicos necessita "inundar" a rede com solicitações de busca, propagando a consulta através de nós até encontrar o dado desejado. Em tais arquiteturas, os nós frequentemente atualizam suas listas de vizinhos — também conhecidas como visões parciais — trocando informações sobre outros nós da rede, o que ajuda a manter a rede resiliente e adaptativa, apesar de sua organização menos rígida em comparação com as estruturas DHT.

Por fim, a arquitetura híbrida mescla os princípios das arquiteturas centralizadas e descentralizadas. Em sistemas híbridos, geralmente se inicia com uma configuração cliente-servidor para estabelecer conexões iniciais entre os nós, criando uma base sólida para interações subsequentes. Após essa fase inicial, a arquitetura pode transitar para um modelo descentralizado, o que permite uma distribuição mais equitativa da carga e aumenta a resiliência do sistema frente a falhas. A versatilidade dos sistemas híbridos se destaca pela sua capacidade de adaptar-se dinamicamente às demandas variáveis, promovendo uma escalabilidade eficaz e uma gestão de recursos mais flexível.

Dentro desta categoria, encontramos os Servidores de Borda, que são posicionados estrategicamente na fronteira entre a rede corporativa e a internet, como ocorre com muitos provedores de serviço de internet (ISPs). Estes servidores gerenciam o acesso dos usuários finais à internet, funcionando como um ponto de entrada e controle, o que pode melhorar significativamente a segurança e a eficiência do tráfego de dados.

Transitando para o contexto dos Sistemas Distribuídos Colaborativos, esses sistemas muitas vezes começam como cliente-servidor, mas rapidamente adotam um modelo descentralizado que facilita a colaboração direta entre nós. Um exemplo notável é o BitTorrent, um sistema de compartilhamento de arquivos baseado em P2P. No BitTorrent, um nó cliente inicia uma busca por um arquivo, identificado como 'F'. A busca é processada inicialmente por um Servidor Web, que direciona a solicitação para um servidor de arquivos .torrent correspondente. Este, por sua vez, consulta um nó rastreador que mantém uma lista de nós armazenando partes do arquivo 'F'. O rastreador então coordena o download ao instruir esses nós a iniciar a transferência do arquivo para o cliente.

Continuando a discussão sobre sistemas distribuídos, o middleware desempenha um papel crucial ao atuar como uma camada intermediária entre as aplicações e as plataformas distribuídas. Na prática, verifica-se que os sistemas de middleware frequentemente adotam estilos arquitetônicos específicos para facilitar a comunicação e a gestão de processos. Por exemplo, muitos sistemas de middleware optam por um estilo arquitetônico baseado em objetos, exemplificado pelo Common Object Request Broker Architecture (CORBA), que permite a interação entre objetos distribuídos em diferentes sistemas operacionais. Por outro lado, sistemas como o TIB/Rendezvous oferecem um estilo baseado em eventos, facilitando a reação a mudanças de estado ou eventos específicos em tempo real.

Embora a escolha de um estilo arquitetônico particular para o middleware possa simplificar significativamente o desenvolvimento de aplicações, ela também pode restringir a flexibilidade dos desenvolvedores. Uma desvantagem dessa abordagem é que o middleware pode não se alinhar perfeitamente com os requisitos específicos ou a visão original do desenvolvedor. Uma solução mais flexível é projetar sistemas de middleware que sejam facilmente configuráveis, adaptáveis e personalizáveis. Isso permite que o middleware se ajuste às necessidades específicas da aplicação, oferecendo uma base mais robusta para o desenvolvimento de soluções eficientes e sob medida para desafios específicos enfrentados nas aplicações distribuídas. Um exemplo prático da aplicação dessa flexibilidade é a utilização de interceptores em sistemas distribuídos.

Conceitualmente, um interceptor é apenas uma construção de software que interrompe o fluxo usual de controle e permite a execução de outro código (específico da aplicação). Tornar os interceptores genéricos pode exigir um esforço de implementação substancial, como ilustrado em Schmidt et al. (2000), e não é claro se, nesses casos, a generalidade deve ser preferida em detrimento da aplicabilidade restrita e simplicidade. Além disso, em muitos casos, ter apenas capacidades de interceptação limitadas pode melhorar a gestão do software e do sistema distribuído como um todo. Para exemplificar, considere a interceptação suportada em muitos sistemas distribuídos baseados em objetos. A ideia básica é simples: um objeto A pode chamar um método que pertence a um objeto B, enquanto este último reside em uma máquina diferente de A. Essa invocação de objeto remoto é realizada em uma abordagem de três etapas:

1. O objeto A é oferecido por uma interface local que é exatamente a mesma que a interface oferecida pelo objeto B. A simplesmente chama o método disponível nesta interface.
2. A chamada de A é transformada em uma invocação de objeto genérica, tornada possível por meio de uma interface de invocação de objeto geral oferecida pelo middleware na máquina onde A reside.
3. Finalmente, a invocação de objeto genérico é transformada em uma mensagem que é enviada através da interface de rede de nível de transporte oferecida pelo sistema operacional local de A.

Após o primeiro passo, a chamada específica para um método em um objeto B é transformada em uma chamada genérica com uma referência ao método e aos parâmetros necessários. Se o objeto B estiver replicado em várias instâncias, cada uma dessas réplicas deve ser invocada para garantir a consistência e disponibilidade. Aqui entra o papel dos interceptores, que facilitam este processo ao replicar a chamada para cada instância de B. A grande vantagem é que o objeto que faz a chamada, A, não precisa saber da existência de múltiplas réplicas de B, e nem o middleware precisa de componentes adicionais para gerenciar essa replicação. Essa tarefa é realizada de forma transparente pelo interceptor de requisição, que é configurado para estar ciente da replicação de B.

Quando uma chamada é feita a um objeto remoto, ela deve ser enviada pela rede. Isso envolve a interface de rede do sistema operacional local. Um interceptor de nível de mensagem pode auxiliar nesse ponto, facilitando a transferência da invocação para o objeto de destino. Por exemplo, se o parâmetro da chamada envolver uma grande quantidade de dados, pode ser vantajoso fragmentar esses dados em partes menores para que sejam remontados no destino. Essa abordagem pode melhorar tanto o desempenho quanto a confiabilidade da transferência. O middleware, por sua vez, não precisa estar ciente dessa fragmentação; o interceptor de nível mais baixo gerencia a comunicação com o sistema operacional local de forma transparente, garantindo uma transmissão eficiente.

Os interceptores adaptam o middleware às mudanças contínuas no ambiente de execução das aplicações distribuídas, como mobilidade, variação na qualidade das redes, falhas de hardware e consumo de bateria. Tradicionalmente, as aplicações não são responsáveis por estas adaptações; ao invés, essa tarefa é integrada ao middleware devido às significativas influências ambientais. Existem três técnicas principais para adaptação:

1. **Separação de preocupações:** Tradicionalmente, sistemas são modularizados separando funcionalidades básicas das funcionalidades extras (como segurança e desempenho). No entanto, isolar completamente essas funcionalidades extras é desafiador, e a integração de preocupações transversais é explorada pelo desenvolvimento de software orientado a aspectos.
2. **Reflexão computacional:** Refere-se à capacidade de um programa inspecionar e adaptar seu comportamento em tempo real. Técnicas reflexivas foram incorporadas em linguagens como Java, proporcionando flexibilidade para modificações durante a execução. Essas técnicas também são aplicadas em alguns sistemas de middleware, embora sua eficácia em grandes sistemas distribuídos ainda precise ser comprovada.
3. **Design baseado em componentes:** Facilita a adaptação por meio da composição e configuração de componentes, podendo ser feita estaticamente ou dinamicamente. A vinculação tardia é uma técnica chave, permitindo flexibilidade operacional em tempo de execução. Pesquisas continuam para automatizar a seleção da melhor implementação de um componente durante a execução, mas a complexidade aumenta em sistemas distribuídos, especialmente quando a substituição de um componente pode impactar outros.

As arquiteturas de software para sistemas distribuídos, especialmente middleware, enfrentam desafios significativos de complexidade devido à necessidade de equilibrar a transparência na distribuição com requisitos específicos das aplicações. Essa demanda resultou no desenvolvimento de soluções altamente flexíveis, embora complexas. Adicionalmente, sistemas distribuídos precisam adaptar-se a mudanças ambientais, como falhas de hardware ou ataques de segurança, de forma autônoma. Isso implica a habilidade de atualizar ou substituir componentes durante a operação contínua, minimizando a necessidade de intervenção humana. Tal capacidade de auto adaptação é essencial para manter a continuidade e eficiência do serviço.

A autogestão em sistemas distribuídos é desafiadora no contexto de middleware. Esses sistemas empregam mecanismos de controle de retroalimentação de alto nível, conhecidos como Computação Autonômica ou Sistemas Autoadaptativos, projetados para autogerenciamento, incluindo auto-reparo, autoconfiguração e auto-otimização. Essas capacidades permitem ao sistema distribuído ajustar-se automaticamente a variações de carga, reparar falhas, mudar configurações e otimizar recursos em tempo real, sem intervenção externa.

Os sistemas auto-gerenciáveis frequentemente utilizam loops de controle de feedback para adaptações automáticas. Esses sistemas são estruturados com base em componentes que precisam ser gerenciados, influenciados por parâmetros controláveis e entradas incontroláveis, como distúrbios do ambiente, que podem causar comportamentos inesperados. A organização de tais sistemas se baseia em três elementos essenciais:

1. **Monitoramento:** O sistema requer monitoramento constante, onde diferentes aspectos são medidos. No entanto, medir comportamentos, como atrasos de ida e volta na internet, pode ser complexo devido às variações e dependências do que está sendo medido.
2. **Análise de Feedback:** Esta é a parte central do loop de controle, contendo algoritmos que analisam as medições em comparação com valores de referência para decidir sobre adaptações necessárias.
3. **Mecanismos de Influência:** Diversos mecanismos podem ser utilizados para influenciar diretamente o comportamento do sistema, como posicionar réplicas, alterar prioridades de agendamento ou redirecionar solicitações. O componente de análise precisa estar ciente desses mecanismos e seus efeitos esperados no comportamento do sistema, desencadeando um ou mais deles para observar posteriormente o efeito.

O loop de controle de feedback também se aplica ao gerenciamento manual de sistemas, onde o componente de análise é substituído por administradores humanos. No entanto, para gerenciar adequadamente qualquer sistema distribuído, esses administradores precisam de equipamentos de monitoramento adequados e mecanismos eficazes para controlar o comportamento do sistema.

Os sistemas distribuídos representam uma complexa interação de componentes autônomos que trabalham conjuntamente para fornecer uma funcionalidade unificada ao usuário, ocultando sua distribuição física através de tecnologias como middleware. Este facilita a comunicação e gestão de processos entre diferentes sistemas operacionais e plataformas, adaptando-se a uma variedade de estilos arquitetônicos, desde arquiteturas em camadas até orientadas a eventos. Essencialmente, as arquiteturas de sistemas distribuídos podem ser centralizadas, descentralizadas ou híbridas, cada uma oferecendo diferentes vantagens em termos de escalabilidade, gerenciamento de carga e tolerância a falhas.

A integração e gerenciamento eficaz desses sistemas requer uma abordagem proativa e autônoma, utilizando técnicas como computação autônoma para adaptar-se a mudanças no ambiente operacional. Isso inclui não apenas auto-reparo e autoconfiguração, mas também a otimização contínua de recursos para manter a eficiência operacional. O desafio persistente na arquitetura de sistemas distribuídos é equilibrar a complexidade inerente com a necessidade de flexibilidade e transparência na distribuição, garantindo que os sistemas sejam robustos, seguros e adaptáveis às exigências dinâmicas e evolutivas das aplicações modernas.

**Referências**

Livro Sistemas Distribuídos Princípios e Paradigmas - Tanembaum (capítulo 1 e 2)