Instituto Superior de Engenharia
Politécnico de Coimbra

Disponibilidade e Desempenho

Disponibilidade e Desempenho em Aplicações Web Modernas

Pedro Miguel Neves Martins
a2021135054(@isec.pt)

Licenciatura em Engenharia Informática
Ramo de Redes e Administração de Sistemas
Instituto Superior de Engenharia de Coimbra
Instituto Politécnico de Coimbra

Coimbra, 1 de novembro de 2023

Conteúdo

Lista de Figuras

Lista de Acrónimos

CIA Confidencialidade, integridade e disponibilidade

2FA autenticação por 2 fatores

DDoS *Distributed denial of service*

SPOF *Single Point of Failure*

SPOFs *Single Point of Failure*

ACID *Atomicity Consistency Isolation Durability*

2PC *Two Phase Commit Protocol*

3PC *Three Phase Commit Protocol*

SGBD Sistemas de Gestão de Base de Dados

DNS *Domain Name System*

IP *Internet Protocol*

HTTP *Hypertext Transfer Protoco*

HTTPS *Hyper Text Transfer Protocol Secure*

DoS *Denial of Service*

APIs *Application Programming Interface*

AWS *Amazon Web Services*

Capítulo 1

Introdução

Na era digital na qual estamos inseridos, as aplicações *web* tornaram-se cruciais nas nossas pesquisas *online*. A disponibilidade e o desempenho são dois pilares fundamentais que definem o sucesso ou o insucesso de uma aplicação *web*. A disponibilidade refere-se à capacidade de um serviço estar sempre acessível aos seus utilizadores, independentemente de onde estejam e do dispositivo que utilizem. Já o desempenho engloba a rapidez e a eficiência com que o esse mesmo serviço responde às pesquisas dos utilizadores/clientes, proporcionando uma experiência fluida.

Este relatório tem como objetivo explorar a complexidade da garantia de disponibilidade e desempenho em aplicações *web* modernas.

Capítulo 2

Conceitos de Disponibilidade

2.1 Disponibilidade

Qualquer organização hoje em dia tem a responsabilidade de proteger os dados de todos os seus utilizadores e fornecer-lhes recursos para responderem ás suas necessidades. O conceito de "disponibilidade"está diretamente relacionado com os conceitos de "confidencialidade"e "integridade"(Confidencialidade, integridade e disponibilidade (CIA)).

A confidencialidade tem como objetivo impedir que informações protegidas ou confidenciais sejam acedidas por individuos não autorizados. Para isso devem ser adotadas medidas de segurança para proteção de ficheiros ou outras informações de modo a impossibilitar o acesso a terceiros, como por exemplo a criptografia dos dados usando autenticação por 2 fatores (2FA) e verificação biométrica. Já a integridade impede que os dados sejam alterados por entidades não autorizadas. É recomendado a adoção de algumas medidas para evitar comprometer a integridade das informações, entre as quais a criptografia, sistema de *backup*, *software* para deteção de erros, etc.

A disponibilidade garante que as informações/dados estão disponiveis a entidades autorizadas e permite-lhes o acesso eficiente aos recursos necessários para o bom funcionamento de uma organização. É posta em causa quando há um tempo de inatividade na rede, seja por falha de *hardware* ou *software* ou mesmo quando há um ataque *Distributed denial of service* (DDoS) a uma organização. Para garantir a disponibilidade, as organizações geralmente implementam medidas como sistemas de *backup*, monitoramento de rede, implementar *firewalls* para mitigar possiveis ataques e educar os colaboradores para boas práticas de segurança. [19] [8]

2.2 SPOF (*Single Point of Failure*)

Os *Single Point of Failure* (SPOFs) correspondem a componentes ou pontos de uma rede que, em caso de falha, podem comprometer o bom funcionamento do sistema, pois deles podem resultar tempo de inatividade na rede, perda de informações e interrupção das operações em curso. Muitos dos potenciais SPOFs são componentes de um *data center*. Para os identificar, é recomendado uma análise cuidada e periódica ao *data center* para monitoramento e análise de riscos para rastrear o desempenho e disponibilidade dos componentes.

Alguns exemplos de SPOFs são:

- **Servidor Principal**. Se uma rede depender de um único servidor principal, a falha desse mesmo servidor pode resultar na inoperância de todos os serviços.
- ***Switch***. Em caso de falha de um switch, todos os servidores ligados a esse *switch* ficarão inacessiveis para o resto dos elementos da rede.
- ***Load Balancer* único**. Se houver apenas um único *Load Balancer* na rede, em caso de falha, todo o tráfego deixará de ser transmitido corretamente.

A prevenção de SPOFs é crucial para garantir a melhor *performance* dos sistemas na rede. Algumas das melhores práticas para prevenir SPOFs envolvem implementar redundância (*backups* que podem assumir quando o componente principal falhar) em servidores, NAS e outros dispositivos de armazenamento, etc; usar *Load Balancers* para fornecer tráfego entre vários servidores, evitando que os mesmos fiquem sobrecarregados e, assim, aumentado o seu desempenho; sensibilizar e treinar a equipa de colaboradores para estarem preparados para responder a situações de falha e recuperações de sistemas. [13] [3] [17]

Capítulo 3

Replicação de estado em sistemas Distribuidos

3.1 ACID (*Atomicity Consistency Isolation Durability*)

A siga ACID representam um conjunto de propriedades essenciais em sistemas de gestão de bases de dados para lhe assegurar a sua confiabilidade e integridade. Refere-se às propriedades de transação (sequência de operações que são tratadas como uma única ação; incluem inserção, exclusão e consulta de dados) de uma base de dados. [15] [16]

- ***Atomicity***. Garante que cada transação seja tratada como uma única unidade atómica, ou seja, ela é totalmente concluída ou então toda a transação falhará e todas as alterações até o momento serão desfeitas (*roolback*).

- ***Consistency***.Garante que apenas dados válidos, sem violar nenhuma das restrições sejam escritos no banco de dados. Se uma transação tentar desrespeitar as regras, ela é desfeita e a base de dados permanece inalterada. Crucial para manter a integridade dos dados.

- ***Isolation***. Garante que a transação seja isolada de todas as outras, processadas de maneira segura e independente ao mesmo tempo, sem interferência. Esta propriedade impede problemas de leituras, que podem ocorrer quando as transações não são devidamente isoladas.

- ***Durability***. Garante que, uma vez que uma transação seja concluida, ela permanecerá no sistema e e sobreviverá a falhas do mesmo, como falta de energia. As alterações feitas por uma transação são armazenadas de maneira que possam ser recuperadas, mesmo em caso de falha do sistema.

3.2 2PC (*Two Phase Commit Protocol*)

O protocolo 2PC é um algoritmo que garante a atomicidade das transações em bases de dados. Este protocolo está dividido em duas fases: **fase de preparação** e **fase de confirmação**.

3.2.1 Fase de Preparação

O coordenador envia uma mensagem de preparação para todos os nós participantes.

Cada nó verifica se pode executar a transação com sucesso. Se puder, responde com uma mensagem, avisando que está pronto. Se houver algum tipo de falha ou outro problema, responde com uma mensagem, avisando que não está pronto.

O coordenador recolhe todos as mensagens dos nós e analisa-as. Se todos os nós confirmaram que estão prontos, o coordenador inicia a transação. Se algum nó avisar que não está pronto ou nem querer responder á solicitação do coordenador, o mesmo decide cancelar a transação e envia uma mensagem de aborto a todos os nós.

3.2.2 Fase de Confirmação

Em caso de confirmação de todos os nós, o coordenador envia uma mensagem de *commit* a todos os nós.

Cada nó participante executa a ação decidida pelo coordenador. Se recebeu uma mensagem de *commit*, confirma a transação. Se recebeu uma mensagem *abort*, anula a transação.

Após concluir a ação (*commit* ou *abort*), cada nó participante envia uma confirmação ao coordenador.

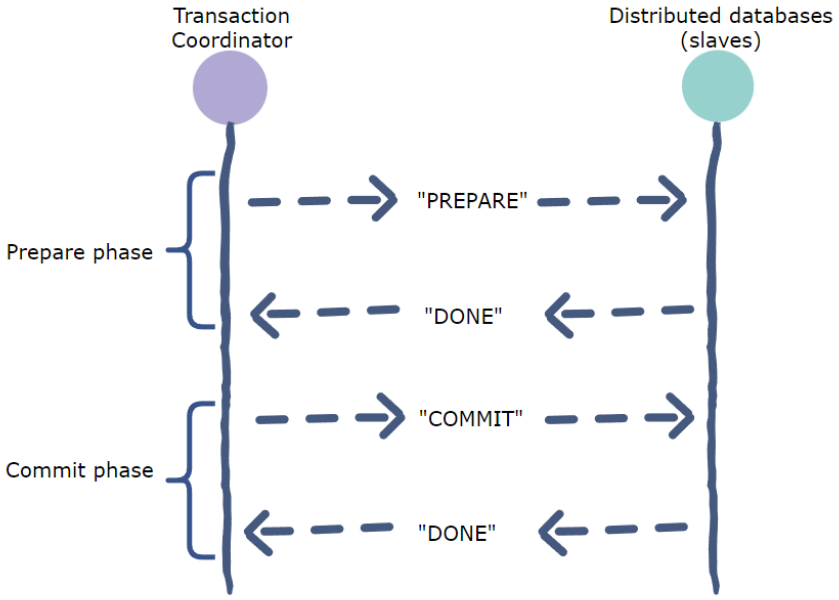

Figura 3.1: Transação bem sucedida, usando o protocolo 2PC

O 2PC garante que todos os nós participantes confirmam ou cancelam a transação. No entanto, ele tem algumas desvantagens, incluindo possiveis bloqueios (por exemplo, a falha de um nó ou até do coordenador da transação, bloqueia todo o processo até que o mesmo se recupere).

Na atualidade, o protocolo 2PC é útil em sistemas distribuidos como sistemas financeiros onde a atomicidade das transações é fundamental para garantir que as transações sejam bem-sucedidas antes de qualquer movimentação e em plataformas de comércio eletrónico para compras *online* onde é necessário que as transações sejam confirmadas com sucesso. [5] [4]

3.3 3PC (*Three Phase Commit Protocol*)

O 3PC é um protocolo usado em bases de dados para garantir que uma transação seja confirmada ou abortada em todos os nós participantes. É uma melhoria em relação ao de duas fases, com o objetivo de reduzir as hipóteses de bloqueios e *deadlocks* que podem ocorrer em diversas situações com o 2PC. O 3PC consiste em três fases: ***Wait***, ***Pre-Commit*** e ***Commit***.

3.3.1 Fase de (*Wait*):

Nesta fase, o coordenador envia uma solicitação de *PREPARE* para todos os participantes. Cada participante, ao receber a solicitação, verifica se pode confirmar a transação. Se o participante puder confirmar, ele responde com uma mensagem *VOTE COMMIT* ao coordenador, senão envia uma mensagem de *VOTE ABORT*.

3.3.2 Fase de *Pre-Commit*:

Assim que o coordenador recolhe as mensagens *VOTE COMMIT* de todos os participantes, ele entra na fase de *Pre-Commit*. Porém, caso alguma das mensagens seja *VOTE ABORT*, ele envia uma mensagem de aborto a todos e cancela a transação. Partindo do principio que correu tudo bem, o coordenador envia uma mensagem de *PREPARE TO COMMIT* para todos os elementos. Quando os participantes recebem essa mensagem, eles respondem com um *READY TO COMMIT* ao coordenador.

3.3.3 Fase de *Commit*:

Se todos os participantes reconheceram a mensagem de *PREPARE TO COMMIT*, o coordenador envia uma mensagem de *GLOBAL COMMIT* para todos os participantes. Ao receber essa mensagem, cada participante confirma a transação e liberta todos os recursos associados a ela.

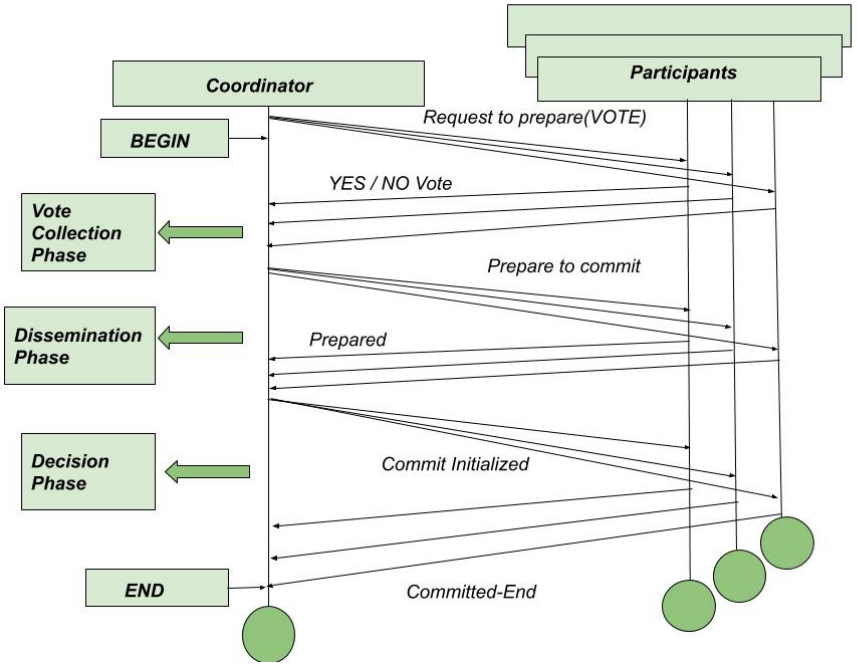

Figura 3.2: Transação bem sucedida, usando o protocolo 3PC

No 2PC, se um participante não tiver certeza se deve confirmar ou abortar, ele pode bloquear logo a transação. No 3PC, a fase de *Pre-Commit* permite que os participantes se preparem para confirmar ou abortar, reduzindo as chances de bloqueio, tornando-se assim a principal vantagem em relação ao 2PC.

No entanto, é importante considerar algumas desvantagens. Por exemplo, o protocolo não garante que uma transação será sempre confirmada ou abortada, uma vez que falhas de rede ou até mesmo do coordenador podem levar a resultados indeterminados.[7] [6]

3.4 Tolerância a falhas em sistemas de base de dados distribuídos

Quando há três Sistemas de Gestão de Base de Dados (SGBD) distribuídos responsáveis por replicar a mesma base de dados, a probabilidade de ocorrer vários cenários de falha aumenta, podendo afetar a disponibilidade e a integridade dos dados. Alguns dos cenários de falha envolvem:

- **Falhas de *hardware***. Podem incluir falhas de disco rígido e/ou memória, por exemplo. Nisto, podem resultar indisponibilidade dos dados replicados pelos SGBD distribuidos.

- **Falhas de *software***. Problemas de software nos SGBD podem levar a interrupções nas operações de replicação dos dados ou mesmo comprometer-lhes a sua integridade.

- **Falhas de rede**. Interrupções na conectividade entre os SGBD podem causar problemas na replicação de dados, uma vez que se um dos servidores deixar de se comunicar com os outros, a sincronização dos dados provavelmente será comprometida.

- **Falhas de Coordenador**. Em sistemas onde o coordenador é responsável por dirigir muitas operações, ele pode ficar sobrecarregado, ou seja, a latência das operações irá aumentar e o sistema pode não atender às expectativas de desempenho.

- **Falhas nos nós**. Perdas de conectividade entre os nós podem resultar na divisão de rede, onde parte dos nós não consegue se comunicar com os outros, podendo afetar a replicação dos dados. Além disso, num sistema distribuído, a carga entre os nós nem sempre é regular. Por exemplo, se um nó estiver sobrecarregado, pode ficar mais lento ou até mesmo falhar, afetando o desempenho geral do sistema.

O 2PC tem a vantagem de saber responder a algumas situações de falha, tais como **falhas do coordenador** e **falhas de nós participantes**.

Em relação à falha do coordenador, se o mesmo falhar durante a coordenação das transações (fase de confirmação), os nós participantes podem detetar essa falha após um período de *timeout* (falha do coordenador) e têm a capacidade de decidir, por conta própria, se devem confirmar ou abortar a transação, evitando que os nós fiquem bloqueados durante largos periodos de tempo.

Quando se trata de uma falha de nó participante durante a fase de preparação, o coordenador pode simplesmente reenviar a mensagem de preparação quando o nó estiver novamente disponível. Por exemplo, caso um nó falhe após ter votado *commit*, a transação só vai ser aplicada quando o nó recuperar, de acordo com as orientações previamente dadas pelo coordenador.

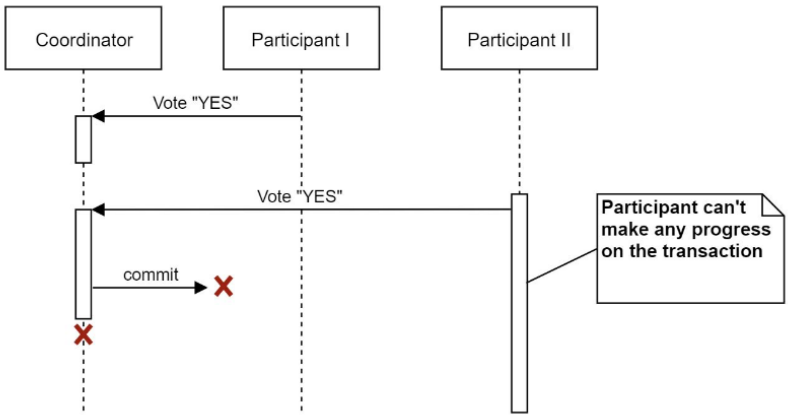

Figura 3.3: Falha do coordenador e da rede

Apesar do 2PC ser eficaz em algumas situações de falha, também tem limitações e não é capaz de resolver todos os cenários de falha. Alguns deles são:

- ***Deadlocks***. O 2PC pode levar a situações de bloqueio indefinido, onde o coordenador e os nós participantes ficam à espera uns dos outros. Se o coordenador falhar após enviar a mensagem de preparação, os nós participantes podem ficar num estado indefinido, onde não sabem se devem confirmar ou abortar a transação. Antes de ser enviada a mensagem de confirmação, os nós podem abortar a transação (como foi dito acima), caso detetem uma falha após um período de *timeout* e, assim, são impedidos de continuar a transação.

- **Falhas na rede**. Se houver interrupções na rede durante a execução do 2PC, os nós participantes podem não receber mensagens do coordenador, o que pode levar (mais uma vez) a atrasos ou até mesmo comprometer a conclusão da transação.

- **Mensagens perdidas**. Se uma mensagem for perdida, os nós participantes podem ficar em estados indefinidos e a transação pode ser comprometida.

- **Falha de um nó participante**. Se um nó participante falhar após votar *prepare*, não poderá ser receber a mensagem de confirmação, enquanto não se recuperar, ou seja, nesse caso, todos os outros nós podem ficar esperando periodos de tempo à espera pela resposta do nó que falhou, e caso este não se recupere toda a transação é abortada.

[9]

Capítulo 4

Arquiteturas de *software* modernas

A arquitetura de *software* é a estrutura que define como os diferentes componentes de um sistema interagem e funcionam juntos. A arquitetura de software, conforme definida por Mark Richards e Neil Ford no livro *"Software Architecture Fundamentals"*, envolve quatro elementos essenciais:

- **Estrutura de *software***. Refere-se à organização dos componentes de um sistema, incluindo como eles se interligam e interagem.

- **Decisões de arquitetura**. São escolhas críticas feitas durante o processo de *design* que influenciam a estrutura e o funcionamento do sistema.

- **Características de arquitetura**. São propriedades do sistema (desempenho, escalabilidade, segurança e flexibilidade) essenciais para atender aos requisitos do mesmo.

- **Principios de *design***. Diretrizes gerais que orientam o processo de *design*, ajudando a garantir que a arquitetura seja eficaz e que cumpra os objetivos do sistema.

4.1 Evolução das aplicações

O *design* de aplicações evoluiu ao longo do tempo devido a uma série de fatores que tornaram essencial acomodar a capacidade de serviços escalarem e evoluírem de forma mais simples e regular.

As principais motivações para essa evolução são:

- **Escalabilidade**. Com o aumento do número de utilizadores, as aplicações modernas precisam de ser capazes de escalar para lidar com cargas de trabalho variáveis, o que não acontecia nas arquiteturas tradicionais, que muitas vezes não conseguiam lidar eficazmente com picos de tráfego. Por exemplo, a computação em nuvem revolucionou a forma como as aplicações são implementadas e geridas, tornando a escalabilidade mais acessível e eficiente. Esta motivação é fundamental num ambiente de desenvolvimento de *software* moderno.

- **Necessidades dos utilizadores**. Torna-se comum as espectativas dos utilizadores serem cada vez mais elevadas em relação à sua experiência de utilização e ao desempenho das aplicações, logo as mesmas necessitam de ser flexíveis o suficiente para se adaptarem a estas mudanças de necessidade.

- **Ciclos de atualizações mais curtos**. A necessidade de lançar atualizações de *software* com maior frequência, muitas vezes em resposta a problemas de segurança, novos recursos ou melhorias de desempenho, requer um *design* que facilite a implementação rápida e regular de mudanças. A segurança, por exemplo, é uma preocupação constante, e as aplicações modernas precisam de ser projetadas para serem facilmente e rapidamente atualizadas para corrigir vulnerabilidades e enfrentar ameaças que surjam ou que possam vir a surgir, evitando que o utilizador fique com uma experiência de utilização não tão boa por largos periodos de tempo.

[10]

4.2 Arquiteturas mais populares

As arquiteturas de *software* mais populares dependem do contexto de uso e das necessidades específicas de cada *software*. Nas seguintes secções, serão apresentadas algumas das arquiteturas mais populares.

4.2.1 Arquitetura em Camadas

A arquitetura em camadas é um padrão comum no desenvolvimento de sistemas de *software*, que tem como objetivo organizar o código e a funcionalidade do próprio *software* em camadas distintas, cada uma com uma determinada responsabilidade.

Cada camada tem uma função clara na aplicação e abstrai o trabalho necessário para atender a uma solicitação de negócios específica, ou seja, cada camada concentra-se apenas no que tem que fazer sem se preocupar com o comportamento das outras camadas.

As camadas que constituem esta arquitetura são:

- **Apresentação**. Interface do sistema. Lida com a interação do utilizador, recolhe e exibe informações, e fornece uma maneira dos utilizadores interagirem com o *software*. Isso pode incluir formulários, páginas *web*, etc.
- **Negócio**. Contém a lógica central do sistema. Processa e gere dados e, também, regras de negócios.
- **Persistência**. Responsável por interagir com sistemas de base de dados. Realiza operações de leitura, escrita e manipulação de dados.
- **Base de Dados**. Camada onde os dados são armazenados. Responsável pela criação de tabelas, consultas e gestão de dados.

Camada fechada

Cada camada da arquitetura é considerada uma camada fechada, ou seja, uma camada na qual os componentes de uma camada superior só podem comunicar com os componentes da camada imediatamente abaixo dela, e não com camadas que estão além dessa camada. A ideia principal por trás desta tecnologia é criar uma separação entre as camadas, impedindo que informações fluam diretamente entre camadas distantes, promovendo uma maior coesão dentro de cada camada.

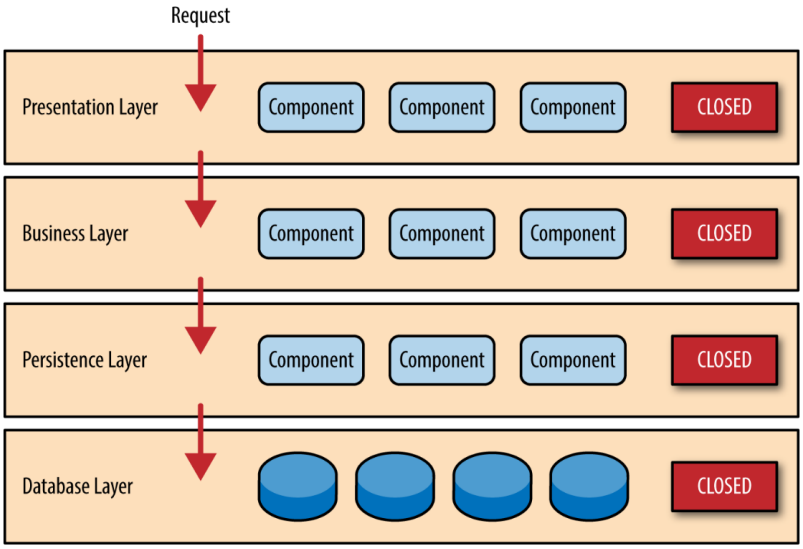

Figura 4.1: Pedido de acesso

Camada isolada

O conceito de camadas de isolamento implica que as mudanças feitas numa camada da arquitetura afetam diretamente componentes noutras camadas. As alterações são limitadas aos componentes dentro da camada em questão e, possivelmente, a uma camada associada. Por exemplo, se permitirmos que a camada de apresentação tenha acesso direto à camada de persistência, qualquer alteração feita no SQL dentro da camada de persistência afetará tanto a camada de negócios quanto a camada de apresentação, resultando num sistema altamente acoplado, com muitos componentes dependentes uns dos outros, o que irá tornar a aplicação mais dificil de modificar.

Camada aberta

Enquanto as camadas fechadas facilitam as camadas de isolamento e, portanto, ajudam a isolar a mudança dentro da arquitetura, há momentos em que faz sentido que certas camadas sejam abertas. Uma camada aberta é projetada de forma a permitir a comunicação e o acesso a funcionalidades específicas por várias camadas do sistema, sem restrições rígidas de isolamento. São frequentemente usadas quando é necessário permitir a comunicação direta e a colaboração entre partes específicas do sistema, ao mesmo tempo em que se mantém um certo controlo neste mesmo processo.

Quando existem funcionalidades ou serviços que precisam ser acedidos por várias partes do sistema, como classes de utilizadores, serviços de auditoria e/ou serviços de

registro, criar uma camada de serviços compartilhados pode ser uma abordagem eficaz, pois permite que diferentes partes do sistema acedam esses serviços de forma controlada, quando necessário.

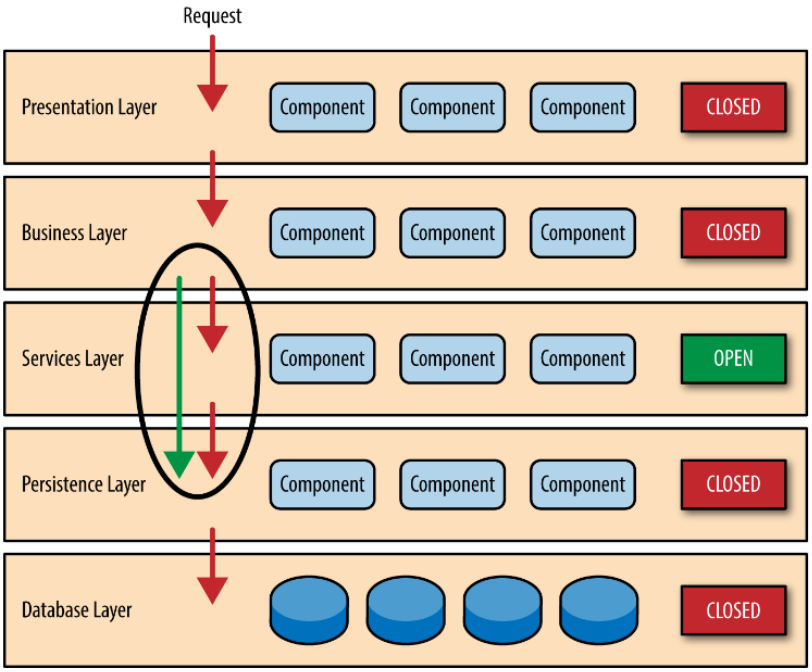

Figura 4.2: Pedido de acesso e camada aberta

Funcionamento

- **Solicitação na camada de apresentação**. O utilizador faz uma solicitação e a camada verifica se os dados fornecidos pelo utilizador são válidos.
- **Encaminhamento para a camada de negócios**. A solicitação é enviada para a camada de negócios, onde ocorre o processamento central (realização de cálculos, processamento de dados, etc).
- **Acesso á camada de persistência**. Caso seja necessário, a camada de negócios interage com a camada de persistência para aceder à base de dados. Esta, por sua vez, é responsável por ler e escrever os dados.
- **Resposta ao Utilizador**. A camada de persistência envia os resultados à camada de negócios, que por sua vez fornece a resposta final à camada de apresentação. Esta apresenta ao utilizador a resposta ao pedido do mesmo.

[20] [12]

4.2.2 Arquitetura Cliente-Servidor

A arquitetura cliente-servidor é um modelo utilizado em redes de computadores e sistemas de *software*. Nesta arquitetura, os sistemas são divididos em duas partes distintas: o cliente e o servidor.

- **Cliente**. O cliente é a parte do sistema que interage diretamente com o utilizador. Pode ser um programa de *software*, uma aplicação *web*, etc. O cliente solicita serviços ao servidor.
- **Servidor**. O servidor é a parte central do sistema que fornece serviços solicitados pelo cliente. Quando recebe uma solicitação, processa-a e fornece a resposta apropriada ao cliente. Os servidores geralmente são computadores ou sistemas de alto desempenho, projetados para lidar com várias solicitações de clientes simultaneamente.

Arquitetura de 1 camada

A arquitetura de 1 camada engloba várias camadas, como a camada de apresentação, de negócios e de base de dados, que são unificadas por meio de um pacote de *software* especializado. Os dados geralmente residem no dispositivo local ou numa unidade compartilhada. Basicamente, a solicitação de recursos, o processamento de dados e a resposta são tratados aqui nesta mesma camada.

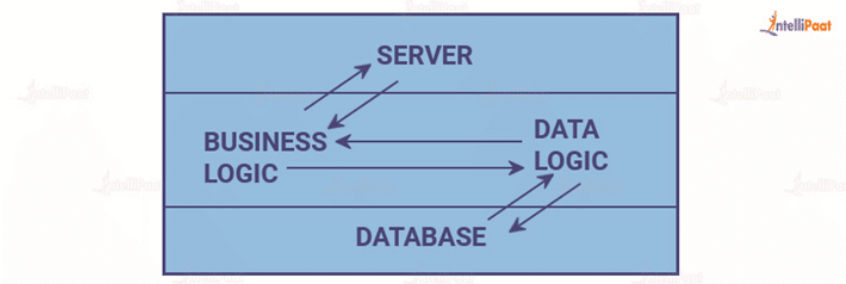

Figura 4.3: Arquitetura de 1 camada

Arquitetura de 2 camadas

Nesse modelo, o cliente é responsável pela apresentação da interface do utilizador e pela lógica de negócios. Já o servidor é responsável por armazenar dados, processar solicitações e fornecer acesso a recursos, como base de dados. A principal vantagem da arquitetura de 2 camadas em relação à de 1 camada é a separação de preocupações e a divisão de responsabilidades entre o cliente e o servidor.

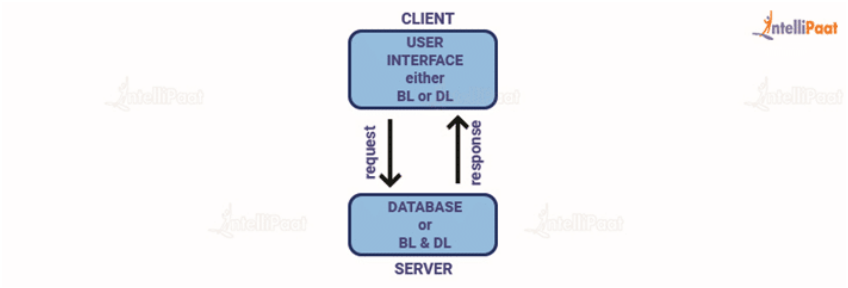

Figura 4.4: Arquitetura de 2 camadas

Arquitetura de 3 camadas

Esta arquitetura divide a aplicação em três camadas distintas: a camada de apresentação (cliente), a camada de lógica de negócios (servidor intermediário) e a camada de dados (base de dados). O cliente lida com a interface do utilizador e a interação com o mesmo. O servidor intermediário gere a lógica de negócios e as regras de processamento. Por fim, na camada de base de dados, os dados são armazenados e recuperados em resposta às solicitações da camada de lógica de negócios. Essa arquitetura é muito usada em aplicações *web*, onde a escalabilidade e a manutenção dos serviços é crucial.

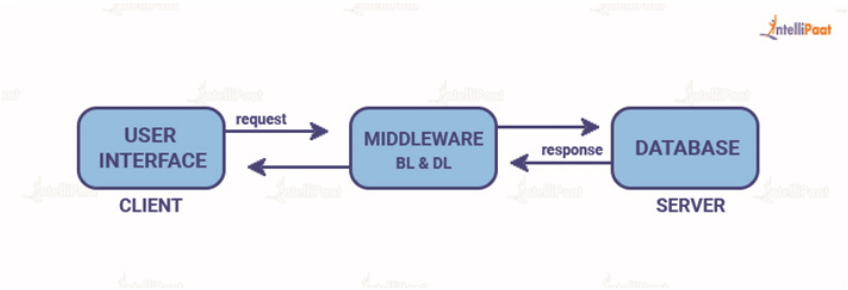

Figura 4.5: Arquitetura de 3 camadas

Arquitetura de nível N

Esta arquitetura permite que cada função seja organizada como uma camada separada, abrangendo a apresentação, o processamento das aplicações e a gestão das funcionalidades dos dados.

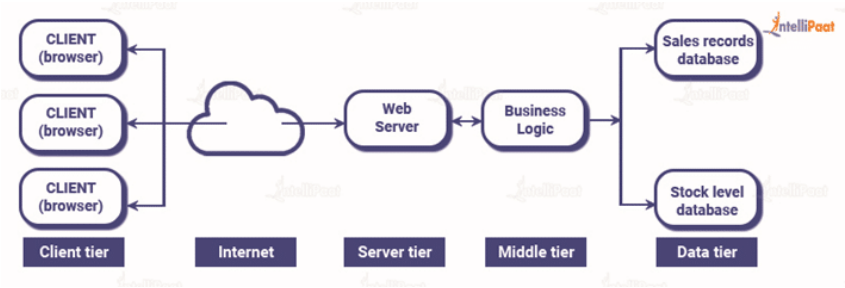

Figura 4.6: Arquitetura de N camadas

Funcionamento

Abaixo está um exemplo de todo o processo de uma arquitetura cliente-servidor.

- O utilizador introduz o endereço do site no navegador, e o navegador envia um pedido ao servidor de *Domain Name System* (DNS).
- O servidor DNS tem a tarefa de procurar e obter o endereço *Internet Protocol* (IP) associado a um servidor *web* e, em seguida, tomar uma decisão com base nesse endereço IP.
- Após receber a resposta do servidor DNS, o navegador envia um pedido *Hypertext Transfer Protoco* (HTTP) ou *Hyper Text Transfer Protocol Secure* (HTTPS) ao endereço IP do servidor *web*, que foi fornecido pelo servidor DNS.
- Uma vez que o pedido é recebido, o servidor começa a enviar os ficheiros essenciais do site necessários para a visualização.
- Finalmente, os ficheiros são processados pelo navegador e o *site* é apresentado ao utilizador.

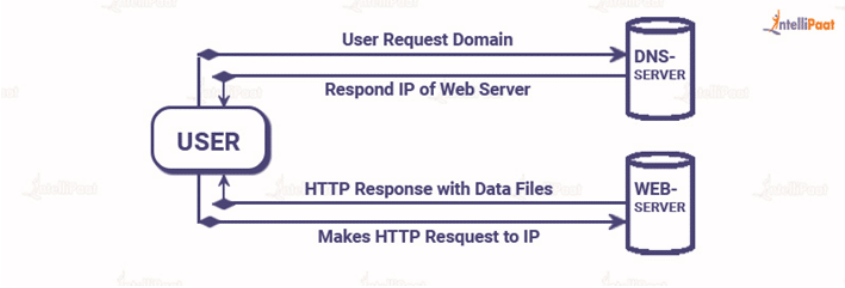

Figura 4.7: Arquitetura Cliente-Servidor

Vantagens

- **Escalabilidade**. A capacidade de aumentar ou diminuir os recursos do servidor conforme a quantidade de solicitações, torna a arquitetura cliente-servidor escalável, que é importante para lidar com picos de tráfego e/ou crescimento futuro.
- **Gestão Centralizada**. O servidor atua como um ponto central de controlo, o que facilita a implementação de políticas de segurança, atualizações de *software* e manutenção do sistema.
- **Partilha de recursos**. Vários clientes podem compartilhar recursos, facilitando o uso eficiente dos mesmos.
- **Padronização**. Permite o uso de protocolos de comunicação padrão, como HTTP ou HTTPS, facilitando a integração com diferentes sistemas e plataformas.

Desvantagens

- **Risco de falha de servidor**. Se o servidor ficar inativo, toda a arquitetura será interrompida.
- **Custos de infraestrutura**. A implementação e manutenção de servidores e da infraestrutura podem ser dispendiosas.
- **Sobrecarga de rede**. Muitos utilizadores ao mesmo tempo podem causar o problema do congestionamento do tráfego.
- **Possibilidade de ataques**. Os servidores são propensos a ataques *Denial of Service* (DoS). Além disso, os pacotes de dados podem ser falsificados ou modificados durante a transmissão de dados de um cliente ao servidor ou vice-versa.

[11] [14]

4.2.3 Arquitetura de Microserviços

A Arquitetura de microserviços é uma abordagem de *design* de *software* que envolve o desenvolvimento de uma aplicação como um conjunto de microserviços, independentes e altamente especializados. Cada serviço é projetado para executar uma função específica e é responsável por uma parte do funcionamento da aplicação. Esses serviços são interconectados por meio de *Application Programming Interface* (APIs) e geralmente são executados em *containers* ou ambientes virtualizados.

Principios

- **Microserviços independentes**. Cada microserviço é autónomo e pode ser desenvolvido, implementado e escalado de forma independente, sem afetar o resto do sistema.
- **Comunicação por APIs**. Os microserviços comunicam-se uns com os outros por meio de APIs, que definem os formatos de mensagens para as interações entre serviços.
- **Escalabilidade**. A escalabilidade é facilitada, uma vez que se pode aumentar ou diminuir a quantidade de instâncias de um microserviço, conforme a quantidade de solicitações, sem afetar os outros serviços.
- **Liberdade Tecnológica**. Cada microserviço pode ser implementado numa tecnologia diferente, se for apropriado para a tarefa. Isso permite escolher a melhor ferramenta para o trabalho.
- **Resiliência**. A arquitetura de microserviços promove a resiliência, uma vez que uma falha num microserviço não afeta todo a aplicação.
- **Gestão de dados**. Cada microserviço pode ter a sua própria base de dados, permitindo escolher o sistema de armazenamento de dados mais adequado para as suas necessidades

Conceitos importantes da arquitetura

.

- ***Microservices***. Representam a divisão da aplicação em serviços independentes que executam funções específicas. Essa divisão é crucial para a arquitetura de microserviços.

- ***Containers***. São frequentemente usados na arquitetura de microserviços para distribuir serviços de maneira consistente. Para gerir um grande número de *containers* em execução, muitas vezes são usadas ferramentas como o *Amazon Elastic Container Service*, que fornece recursos de escalabilidade e alta disponibilidade, que são cruciais em arquiteturas de microserviços.

- ***Cloud-Oriented***. A arquitetura de microserviços é frequentemente adotada em ambientes de nuvem, como *Amazon Web Services* (AWS), *Azure* ou *Google Cloud*. A escalabilidade e a flexibilidade oferecidas por esses ambientes são compatíveis com os princípios desta arquitetura.

- **Armazenamento e base de dados**. Em vez de usar uma única base de dados centralizada, cada microserviço pode ter a sua própria base de dados, podendo escolher a tecnologia que melhor atende às suas necessidades. Por exemplo, o *Amazon ElastiCache* melhora o desempenho dos serviços, possibilitando a recuperação de dados a partir de caches de memória rápidos e fáceis de gerir. Existem outras plataformas da *Amazon* que permitem o acesso rápido e seguro a todos os dados dos serviços, tais como *Amazon RDS* e *Amazon DynamoDB*.

- ***Load balancing***. Os serviços de balanceamento de carga são essenciais para distribuir o tráfego entre os microserviços, garantindo alta disponibilidade e escalabilidade.

- **Monitoramento e *logging***. Numa arquitetura de microserviços, a capacidade de monitorar o desempenho e rastrear problemas é fundamental. Soluções de monitoramento, como *AWS CloudTrail* (monitoramento de APIs) e *Amazon CloudWatch* (monitoramento de aplicações).

- ***DevOps***. A cultura *DevOps* é altamente compatível com a arquitetura de microserviços. Envolve a colaboração entre equipas de desenvolvimento e operadores de *software* para criar um ambiente de desenvolvimento e operações mais eficiente e colaborativo, permitindo que as organizações entreguem *software* de alta qualidade de forma mais rápida e confiável, automatizando tarefas repetitivas, como compilações de código, testes de monitoramento, etc.

[2]

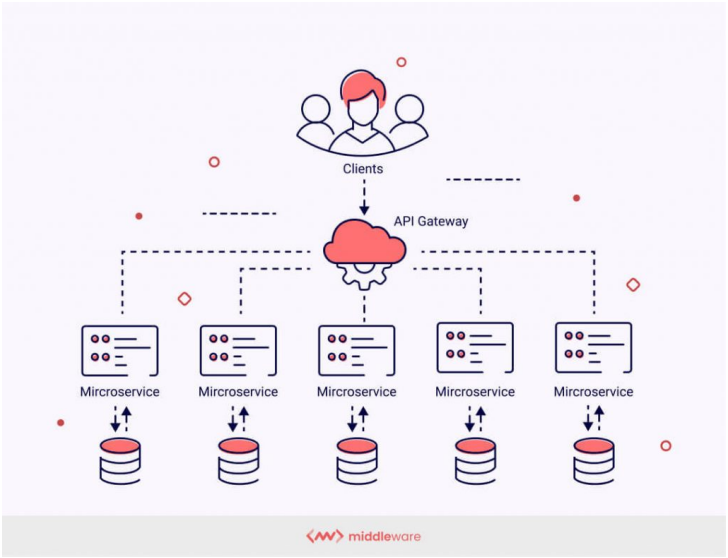

Figura 4.8: Arquitetura de Microserviços

4.2.4 Arquitetura baseada em eventos

Uma arquitetura orientada a eventos é um padrão de *design* de software que permite detetar eventos como transações, consultas de *sites* e agir sobre eles em tempo real. O fluxo desta arquitetura é executado por eventos e permite realizar alguma ação em resposta a um evento.

Um exemplo comum de uso desta arquitetura é em sistemas de comércio eletrónico, onde eventos podem representar ações do utilizador, como adicionar um item ao carrinho de compras e fazer um pedido. Os eventos são gerados e recebidos pelo consumidor de eventos, permitindo (neste caso) a atualização do *stock* e o processamento do pagamento.

Conceitos importantes da arquitetura

- **Evento**. Um evento é uma notificação assíncrona que indica que algo aconteceu num sistema. Pode ser qualquer ação ou mudança de estado que seja relevante para o sistema, como a criação de um novo utilizador, solicitação da redefinição de uma senha, a realização de uma compra, etc.
- **Produtor de eventos**. Parte do sistema responsável por gerar eventos quando ocorre uma ação ou mudança de estado. Por exemplo, uma aplicação pode gerar um evento sempre que um utilizador faz *login*.
- **Consumidor de Eventos**. Recebe e responde aos eventos gerados pelos produtores. Os consumidores executam ações com base nos eventos recebidos (atualizar

dados, enviar notificações, entre outros).

- ***Router* de eventos**. Infraestrutura que filtra e distribui os eventos. Atua como intermediário entre o produtor e o consumidor de eventos. O *router* elimina a necessidade de comunicação direta entre serviços produtores e consumidores. Além disso, pode definir políticas que definem que utilizadores e recursos têm permissão para aceder aos seus dados.

Benefícios

- **Resposta em tempo real**. A arquitetura baseada em eventos permite que os sistemas reajam a eventos em tempo real, o que é valioso em cenários onde a detecção e ação imediata são críticas, como sistemas de monitoramento de segurança, notificações em tempo real, entre outros.
- **Escalabilidade**. À medida que a carga de eventos aumenta, é possível adicionar novos consumidores para lidar com os eventos, sem afetar a geração dos mesmos, proporcionando uma maior agilidade no sistema.
- **Desacoplamento**. Além da escalabilidade, os serviços desta arquitetura são menos acoplados, ou seja não têm uma forte dependência uns dos outros. Isso facilita a expanção ou a remoção de um serviço em caso de falha, sem afetar todo o sistema.
- **Auditoria**. A arquitetura orientada a eventos facilita a auditoria dos mesmos. No *router* de eventos, como foi dito acima, é permitido restringir o acesso a certas informações por parte de um determinado utilizador.

Funcionamento da arquitetura

O fluxo desta arquitetura ocorre da seguinte forma:

- Um evento ocorre, desencadeado por uma ação do utilizador. Alguns eventos comuns (já visto acima) são a adição de um item ao carrinho de compras, realização de uma compra num *site*, a solicitação da redefinição de uma senha, entre outros.
- O produtor de eventos (aplicação *web*, por exemplo) gera um evento e publica no *router* de eventos que filtra o evento e envia aos consumidores preparados para lidar com este tipo de evento.

- Cada consumidor de eventos processa o evento de acordo com sua lógica de negócios, que pode envolver a execução de determinadas ações, atualizações de dados, etc.
- A resposta ao evento é enviada ao produtor de eventos que lida com o utilizador. Normalmente, a resposta é um determinado comportamento ou consequência do evento produzido pelo utilizador.

[18] [1]

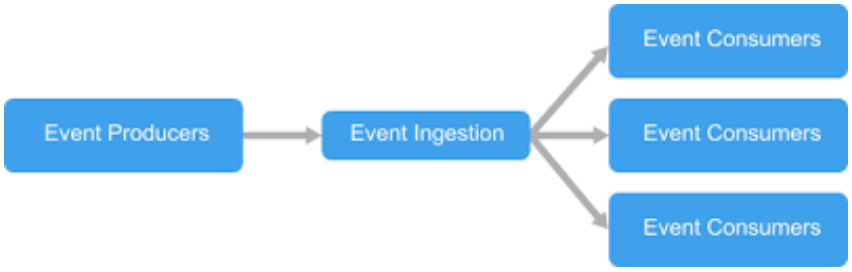

Figura 4.9: Arquitetura baseada em eventos

Referências

[1] amazon. *event-driven-architecture*. URL: https://aws.amazon.com/pt/event-driven-architecture/.

[2] amazon. *microservices*. URL: https://aws.amazon.com/pt/microservices/.

[3] avinetworks. *single point of failure*. URL: https://avinetworks.com/glossary/single-point-of-failure/.

[4] dremio. *Two Phase Commit Protocol*. URL: https://www.dremio.com/wiki/two-phase-commit-protocol/.

[5] educative.io. *Two Phase Commit Protocol*. URL: https://www.educative.io/answers/what-is-the-two-phase-commit-protocol.

[6] exploredatabase.com. *Three Phase Commit Protocol*. URL: https://www.exploredatabase.com/2018/03/three-phase-commit-3pc-protocol-in-distributed-database-transactions.html.

[7] geeksforgeeks. *Three Phase Commit Protocol*. URL: https://www.geeksforgeeks.org/three-phase-commit-protocol/.

[8] getsmarteye. *confidentiality, integrity, availability basics of information security*. URL: https://getsmarteye.com/confidentiality-integrity-availability-basics-of-information-security/.

[9] Ricardo Alexandre Gonçalo Amador. *Sistemas Distribuidos e Tolerancia a Falhas*. URL: https://www.di.ubi.pt/~pprata/sdtf/Ti_DBdistribuidasGoncaloRicardo.pdf.

[10] Magnus Larsson Hongyu Pei Breivold Ivica Crnkovic. *A systematic review of software architecture evolution research*. URL: https://www.sciencedirect.com/science/article/abs/pii/S0950584911001376.

[11] intellipaat. *what-is-client-server-architecture*. URL: https://intellipaat.com/blog/what-is-client-server-architecture/.

[12] oreilly. *Layered Architecture*. URL: https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html.

[13] parallels. *single point of failure 2*. URL: https://www.parallels.com/blogs/ras/single-point-of-failure-2/.

[14] simplilearn. *what-is-client-server-architecture-article*. URL: https://www.simplilearn.com/what-is-client-server-architecture-article.

[15] techopedia. *atomicity consistency isolation durability acid database management system*. URL: https://www.techopedia.com/definition/23949/atomicity-consistency-isolation-durability-acid-database-management-system.

[16] techtarget. *ACID*. URL: https://www.techtarget.com/searchdatamanagement/definition/ACID.

[17] techtarget. *single point of failure*. URL: https://www.techtarget.com/searchdatacenter/definition/Single-point-of-failure-SPOF.

[18] tibco. *what-is-event-driven-architecture*. URL: https://www.tibco.com/reference-center/what-is-event-driven-architecture.

[19] tutorialspoint. *availability in information security*. URL: https://www.tutorialspoint.com/availability-in-information-security.

[20] Priyal Walpita. *Software Architecture Patterns — Layered Architecture*. URL: https://priyalwalpita.medium.com/software-architecture-patterns-layered-architecture-a3b89b71a057.

Capítulo 5

Anexos

5.1 Meta 1

Em resumo, ao discutir conceitos como disponibilidade e SPOF, exploramos a importância de garantir a boa *performance* dos sistemas, especialmente em cenários em que a falha dos mesmos não é uma opção. Ao mesmo tempo, a compreensão dos princípios ACID destacou a necessidade de manter a confiabilidade e a integridade dos dados em aplicações críticas. Os protocolos 2PC e 3PC desempenham um papel fundamental na gestão de transações, com o 2PC sendo amplamente utilizado, embora apresente desvantagens no que toca a bloqueios inesperados. Já o 3PC, surgiu como uma solução que aborda algumas das limitações do 2PC, proporcionando maior robustez.

5.2 Meta 2

Em suma, na meta 2 foi falado da evolução do *design* de aplicações para atender às crescentes demandas por escalabilidade, flexibilidade e eficiência. Foi também falado das arquiteturas mais populares nos dias que correm, como a arquitetura em camadas que, como vimos, essa abordagem divide a aplicação em componentes lógicos, facilitando a sua manutenção. A arquitetura cliente-servidor, por sua vez, introduziu a ideia de distribuição de funções entre clientes e servidores, melhorando a interatividade. A arquitetura de microserviços decompõe a aplicação em serviços independentes e altamente especializados, permitindo escalabilidade e manutenção mais eficazes. Por fim, a arquitetura baseada em eventos trouxe a ideia de sistemas reativos, em que eventos desencadeiam ações, proporcionando uma abordagem altamente eficiente para lidar com fluxos de dados em tempo real.

5.3 Alterações no documento

O documento sofreu algumas alterações na meta 2, em relação á meta 1. Tentei adaptar o mesmo à *template* fornecida pelo professor. Algumas das formatações de capitulos, secções e subsecções foram também alteradas (foram adicionados capitulos ao relatório). Além disso, foi adicionada uma página de acrónimos na meta 2.