

Sistema Distribuído: Lista de Compras

Jênifer Vieira Goedert¹

¹Acadêmica do Curso de Bacharelado em Ciência da Computação
do Instituto Federal Catarinense - Campus Rio do Sul, SC - Brasil

jenifergoedert10@gmail.com

Abstract. *This article presents the development of a shopping list application as a distributed system. The application allows users to add items to the list, view them, and perform calculations, using Python for implementation, MongoDB for data storage, and Docker for service management. The distributed system has proven to be a robust and efficient solution, demonstrating high availability, fault tolerance, and flexibility. The architecture ensures operational continuity even in failure scenarios, while Docker containers contribute to scalability and agile management. Despite these advantages, the implementation requires continuous attention to maintain consistency among services and databases.*

Resumo. *Este artigo apresenta o desenvolvimento de uma aplicação de lista de compras como um sistema distribuído. A aplicação permite ao usuário adicionar itens à lista, visualizá-los e calcular totais, utilizando a linguagem Python para a implementação, MongoDB para armazenamento de dados e Docker para a gestão dos serviços. Contudo, o sistema distribuído demonstrou alta disponibilidade, resiliência a falhas e flexibilidade. Sua arquitetura garante a continuidade operacional mesmo em situações de falha, enquanto os contêineres Docker facilitam a escalabilidade e gestão ágil. No entanto, a implementação exige atenção contínua para manter a consistência entre os serviços e bancos de dados e gerenciamento de latência.*

1. Introdução

A computação, em seu estágio inicial, era focada na centralização de seu processamento, e os principais atores desse modelo foram o *mainframe* e os terminais de acesso. Com o passar do tempo, foi proposta uma mudança dessa arquitetura centralizada e rígida, através da descentralização do processamento, quando surgiram, então, os primeiros sistemas distribuídos. Desde então, eles têm evoluído, sendo uma área da computação que se relaciona com o nosso dia a dia, por meio, por exemplo, de serviços e aplicações web, computação *peer-to-peer* e redes de sensores[AZEVEDO 2021].

No entanto, este artigo abordará a implementação e a estruturação de um sistema distribuído, organizado em camadas e utilizando o paradigma de comunicação RPC (Remote Procedure Call), com foco em APIs RESTful. Além disso, os serviços foram gerenciados por meio de contêineres Docker.

2. Fundamentação teórica

Nesta seção, serão apresentados o referencial teórico sobre sistemas distribuídos, e uma forma de organização desses sistemas por meio de uma arquitetura em camadas. Por fim, o paradigma de comunicação conhecido como Chamada de Procedimento Remoto (RPC) e o banco de dados MongoDB.

2.1. Sistemas distribuídos

Segundo [Coulouris 2007], um sistema distribuído é aquele no qual os componentes, localizados em computadores interligados por uma rede, se comunicam e coordenam suas ações apenas passando mensagens. De acordo com [Bezerra 2024], esses sistemas possuem as seguintes características:

- a) Compartilhamento de recursos.
- b) Concorrência.
- c) Inexistência de um relógio global.
- d) Falhas independentes.
- e) Comunicação e organização interna dos computadores ocorrem de forma oculta aos usuários.
- f) Interação entre usuários e aplicações ocorrem de maneira consistente e uniforme, independente do local da interação.
- g) Deve ser relativamente fácil de aumentar a escala de sistemas distribuídos.
- h) Sistema deve continuar disponível mesmo se algumas partes estiverem avariadas.

Exemplos de sistemas distribuídos incluem a Internet, as intranets (redes privadas geridas por organizações), e a computação móvel e ubíqua.

O compartilhamento de recursos é um forte motivo para a construção de sistemas distribuídos. Os recursos podem ser gerenciados por servidores e acessados por cliente, ou podem ser encapsulados como objetos e acessados por outros objetos cliente [Coulouris 2007].

Para [Bezerra 2024], um sistema distribuído deve ser interoperável e escalável, capaz de integrar diferentes plataformas e expandir seus recursos sem perder desempenho. Ele precisa garantir segurança, tratar falhas de forma eficaz e gerenciar a concorrência entre processos. Além disso, deve ser transparente, apresentando-se ao usuário como um sistema único, independentemente de sua complexidade interna. Diante disso, para lidar com essa complexidade existem diferentes tipos de arquitetura, dentre elas há a arquitetura em camadas [Bezerra 2024].

2.2. Arquitetura em camadas

A arquitetura em camadas, também conhecida como *layered architecture*, é um padrão arquitetural que organiza um sistema em camadas distintas, onde cada camada possui uma responsabilidade específica. É caracterizado por: a) Separação de responsabilidades em cada camada do sistema; b) Hierarquia de comunicação entre as camadas e c) Abstração das camadas superiores em relação às inferiores. A abordagem em camadas traz diversos benefícios, incluindo a separação de interesses, isolamento de mudanças, reutilização de lógicas independentes e testabilidade. Apesar dos benefícios, a abordagem multicamadas também pode apresentar desvantagens, tais como a complexidade excessiva e a latência decorrente do tráfego de dados por muitos componentes[DNC sd].

Nesse contexto, têm-se a arquitetura de três camadas. Para [IBM sd], ela é uma arquitetura de aplicativo de software estabelecida que organiza aplicativos em três camadas

de computação física e lógica: a camada de apresentação ou a interface com o usuário; a camada do aplicativo, na qual os dados são processados; e a camada de dados, na qual os dados associados ao aplicativo são armazenados e gerenciados, conforme figura 1. O principal benefício da arquitetura de três camadas é que devido ao fato de cada camada executar sua própria infraestrutura, cada camada pode ser desenvolvida simultaneamente por uma equipe de desenvolvimento separada e pode ser atualizada ou ajustada conforme necessário sem impactar as outras camadas. A comunicação entre essas camadas pode ocorrer de várias maneiras, incluindo o uso do paradigma de comunicação de alto nível conhecido como Chamada de Procedimento Remoto (RPC).



Figure 1. Arquitetura em três camadas

2.3. Chamada de Procedimento Remoto

A Chamada de Procedimento Remoto (RPC) é uma tecnologia que facilita a criação de programas distribuídos, permitindo que o usuário faça chamadas a procedimentos remotos como se fossem locais, independentemente das diferenças de arquitetura entre cliente e servidor[UEP 2024]. Uma forma de aplicar isso em projetos, é por meio do RESTful, uma API que segue os princípios do Representational State Transfer (REST) e oferece comunicação escalável e eficiente entre sistemas[AMA 2023]. O RESTful oferece versatilidade, facilidade de desenvolvimento e suporte a diversas linguagens e formatos de dados. APIs RESTful proporcionam escalabilidade, flexibilidade e independência tecnológica, permitindo que cliente e servidor evoluam separadamente. Entretanto, desvantagens incluem maior complexidade, dependência de conexão com a Internet e desempenho variável devido à dependência da rede e dos servidores [THE 2024].

2.4. MongoDB

O MongoDB é um sistema de banco de dados NoSQL orientado a documentos que utiliza uma estrutura flexível de armazenamento em JSON. Uma das principais características do MongoDB é a replicação de dados, que garante alta disponibilidade e tolerância a falhas. A replicação é implementada por meio de um conjunto de réplicas (*ReplicaSet*), que consiste em um grupo de instâncias do MongoDB (nós) que mantêm cópias idênticas dos dados[MON 2024].

Em um *ReplicaSet*, há um nó primário responsável por receber todas as operações de gravação, enquanto os nós secundários replicam os dados do primário de forma

assíncrona. Caso o nó primário falhe, um dos nós secundários é automaticamente eleito como o novo primário, garantindo a continuidade das operações. A configuração de um *ReplicaSet* pode incluir ainda um nó árbitro, que participa das eleições, mas não armazena dados[MON 2024].

Essa arquitetura oferece redundância, aumenta a disponibilidade dos dados e permite uma maior escalabilidade, distribuindo as operações de leitura entre os nós secundários. Além disso, a replicação de dados em diferentes servidores ou data centers melhora a localidade dos dados e oferece mecanismos para recuperação em caso de desastres[MON 2024].

3. Solução proposta

O sistema distribuído proposto é composto por um cliente, que atua como o *front-end* da aplicação, e um escalonador, responsável por alocar os serviços disponíveis em cada camada. O sistema conta com três serviços de negócios, que gerenciam as funções e regras do negócio, e três serviços de dados, dedicados ao acesso e gerenciamento de informações. Além disso, o sistema inclui três bancos de dados MongoDB com replicação de dados. A implementação de múltiplos serviços e bancos de dados visa garantir alta disponibilidade e proporcionar tolerância a falhas. Veja a estrutura e ciclo de requisições do sistema distribuído no diagrama a seguir.

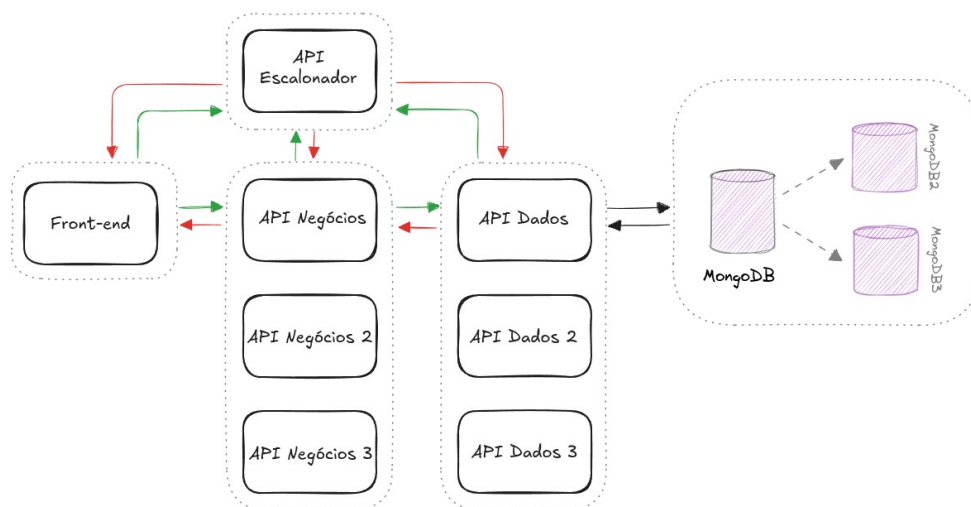


Figure 2. Estrutura do sistema distribuído

O processo se inicia na interface *front-end* da aplicação, que imediatamente solicita ao escalonador um serviço de negócios disponível, antes mesmo de exibir a interface ao usuário. O escalonador verifica a disponibilidade entre os três serviços configurados. Se um serviço estiver disponível, o escalonador retorna o endereço IP e a porta correspondentes ao *front-end*. Caso contrário, o escalonador informa que não há serviços disponíveis, mas apenas após esgotar todas as alternativas. Uma vez que o *front-end* obtém o IP e a porta de um serviço de negócios, ele os utiliza ao longo de todo o processo. Sempre que a camada de negócios precisar acessar um serviço de dados, por exemplo, para montar partes do objeto "Pessoa", ela novamente recorrerá ao escalonador para identificar um serviço disponível. Para operações de gravação de dados, como a gravação do

objeto "Pessoa" completo contendo nome, itens selecionados, quantidade total de itens e valor final, a camada de negócios também utilizará o escalonador para identificar um banco de dados MongoDB disponível. Uma vez gravados, os dados são replicados automaticamente para as outras duas réplicas do *ReplicaSet*, garantindo redundância e alta disponibilidade, fundamentos essenciais para a produção em sistemas distribuídos.

A codificação do sistema distribuído foi realizada utilizando a linguagem Python e estruturada em quatro camadas: cliente, escalonador, negócios e dados. Exceto pela camada cliente, todas as camadas contam com uma API dedicada. O escalonador realiza um papel importante no sistema, escalonando os serviços. Veja a seguir um *endpoint* da API escalonador para selecionar um banco de dados.


```
1 @escalonadorAPI.get("/seleciona_banco/")
2 async def seleciona_banco():
3     global bancos_disponiveis
4
5     if not bancos_disponiveis:
6         return {"message": "Banco de dados não está disponível."}
7
8     temp_banco = bancos_disponiveis[:]
9     resultado_final = {"message": "Não há um banco de dados disponível."}
10
11     for dado in temp_banco:
12         ip, porta = dado
13         print('Testando conexão ', ip, '->', porta)
14
15         conexao_bem_sucedida = await testar_conexao(ip, porta)
16
17         print('conexao_bem_sucedida >> ', conexao_bem_sucedida)
18         if conexao_bem_sucedida:
19             resultado_final = {"ip": ip, "porta": porta}
20             break
21
22     return resultado_final
```

Figure 3. Estrutura do sistema distribuído

Nesse trecho de código, o escalonador verifica se há bancos disponíveis na lista global `bancos_disponiveis`. Se a lista estiver vazia, retorna uma mensagem indicando que nenhum banco de dados está disponível. Caso contrário, percorre os bancos na lista, tentando estabelecer conexão com cada um. Se a conexão for bem-sucedida, retorna o IP e a porta do banco disponível. Se nenhuma conexão for estabelecida, retorna uma mensagem informando que não há banco disponível. As tentativas de conexão possuem um tempo limite configurado de 10 segundos. Da mesma forma, a seleção dos serviços também é realizada seguindo um processo semelhante.


Outra parte fundamental do projeto envolve a comunicação entre diferentes contêineres. Por exemplo, a API de dados faz uma solicitação ao escalonador para obter informações sobre um banco de dados disponível. Essa comunicação é ilustrada nas Figuras 4 e 5, que mostram o processo detalhado dessa interação. A função `obter_url_banco` é responsável por recuperar a URL de conexão com o banco de dados. Para isso é realizado uma requisição ao escalonador por meio da função

`seleciona_banco`, retornando IP e porta de acordo com os testes de disponibilidade realizados pelo escalonador. Em seguida, o retorno é verificado para confirmar se contém um IP e uma porta. Se essas informações estiverem presentes, o banco de dados é considerado disponível.



```
1 # Obtém banco de dados
2 async def obter_url_banco(servico: RequisicoesDados) -> str:
3     print('\nVerificando conexão com o banco de dados...')
4     try:
5         banco = await servico.seleciona_banco() # Requisição para o escalonador
6     except Exception as e:
7         raise HTTPException(status_code=500, detail=str(e))
8
9     # Verifica se a resposta contém as chaves 'ip' e 'porta', indicando uma conexão bem-sucedida
10    if isinstance(banco, dict) and 'ip' in banco and 'porta' in banco:
11        url_banco = f"mongodb://{banco['ip']}:{banco['porta']}"
12        print('Conectado ao banco ', url_banco)
13        return url_banco
14    else:
15        raise HTTPException(status_code=500, detail="Não foi possível obter um serviço/negócio disponível.")
```

Figure 4. API de dados solicita banco de dados disponível ao escalonador



```
1 async def seleciona_banco(self):
2     url_escalonador = "http://172.45.0.11:8012/seleciona_banco/"
3     async with httpx.AsyncClient() as client:
4         response = await client.get(url_escalonador, timeout=120)
5         return response.json()
```

Figure 5. Requisição solicitando o banco de dados ao escalonador

Para a construção deste sistema, foram utilizados contêineres Docker para isolar e gerenciar cada serviço. Foram criados contêineres para um cliente, três serviços de negócios, três serviços de dados e um serviço de escalonador, totalizando oito contêineres Python. Além disso, foram configurados três contêineres MongoDB com um *ReplicaSet* para garantir a alta disponibilidade e redundância dos dados. Todos os contêineres foram configurados para operar na mesma rede, facilitando a comunicação entre os serviços.

4. Aspectos positivos e negativos

A aplicação distribuída apresenta vários benefícios significativos. A alta disponibilidade e redundância são garantidas pela implementação de múltiplos serviços de negócios e dados, além do uso de um *ReplicaSet* MongoDB. Isso assegura que o sistema continue a operar mesmo em caso de falhas. O escalonador dinâmico contribui para uma distribuição eficiente das solicitações, otimizando o desempenho do sistema. A utilização de contêineres Docker facilita a gestão e a escalabilidade, isolando cada serviço e permitindo uma configuração mais flexível e replicável.

No entanto, a complexidade do sistema pode aumentar devido à arquitetura distribuída, o que torna a configuração e o gerenciamento mais desafiadores. A necessidade de verificar a disponibilidade e estabelecer conexões com vários serviços e bancos de dados introduz um pouco de latência em casos em que somente os últimos serviços das listas estão disponíveis. Manter a consistência e o estado entre os diferentes serviços e bancos

de dados também pode ser complicado, exigindo atenção especial para evitar problemas de sincronização.

5. Conclusão

Diante do sistema distribuído proposto, o mesmo demonstrou-se uma abordagem robusta e eficiente para a gestão de serviços e dados, destacando-se pela sua alta disponibilidade e resiliência a falhas. A utilização de múltiplos serviços de negócios e dados, juntamente com a implementação de um *ReplicaSet* MongoDB, assegura que o sistema mantenha a continuidade operacional mesmo em cenários de falha. A arquitetura baseada em contêineres Docker contribui para a escalabilidade e flexibilidade do sistema, permitindo um gerenciamento mais ágil e uma configuração replicável.

Nesse contexto, o escalonador desempenha um papel crucial na alocação dinâmica de serviços, otimizando a distribuição das solicitações e melhorando o desempenho geral. Porém, a complexidade inerente à arquitetura distribuída pode representar desafios significativos. A necessidade de monitorar e manter a consistência entre múltiplos serviços e bancos de dados requer uma atenção cuidadosa para evitar problemas de sincronização e latência. Além disso, a sobrecarga de recursos associada à execução de vários contêineres na mesma rede pode exigir ajustes adicionais para garantir a eficiência em contextos de maior escalabilidade.

Acesso ao projeto no GitHub: <https://github.com/Jenifer19IFC/sistema-distribuido-lista-compras>

References

- (2023). O que é uma api restful? Acesso em: 08 de abr. de 2024.
- (2024). Remote procedure call. Acesso em: 08 de abr. de 2024.
- (2024). Replication. Acesso em: 18 de ago. de 2024.
- (2024). Rest api: what it is, how it works, advantages and disadvantages. Acesso em: 09 de abr. de 2024.
- (s.d.). O que é arquitetura de três camadas? Acesso em: 12 de ago. de 2024.
- (s.d.). O que é arquitetura em camadas e como funciona? Acesso em: 12 de ago. de 2024.
- AZEVEDO, M. T. d. (2021). *Sistemas Distribuídos*. Senac.
- Bezerra, W. R. (2024). Características de sd. Acesso em: 09 de ago. de 2024.
- Coulouris, G. F. (2007). *Sistemas Distribuídos Conceitos e Projeto*. Bookman.