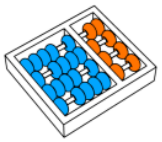

Implementação de um Sistema Distribuído para a Solução Smart Parking

André Luis R. Gouvêa *Lucas B. A. Farias*
Tiago P. Dall'Oca *Juliana Freitag Borin*

Relatório Técnico - IC-PFG-24-08
Projeto Final de Graduação
2024 - Julho

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Implementação de um Sistema Distribuído para a Solução Smart Parking

André Luis R. Gouvêa
Lucas B. A. Farias
Tiago P. Dall'Oca
Juliana Freitag Borin

Resumo

Neste trabalho, apresentamos uma extensão do projeto Smart Parking, um sistema de monitoramento de vagas de estacionamento implementado no campus da Unicamp. O objetivo foi desenvolver uma solução tecnológica eficiente para a gestão de estacionamentos, aumentando a conveniência dos usuários e otimizando o uso das vagas disponíveis. Utilizamos tecnologias como MongoDB para gerenciamento de dados, MQTT para comunicação entre dispositivos IoT e React para o front-end, criando um sistema modular, escalável e de fácil manutenção.

A arquitetura segue práticas de projeto de software, garantindo baixo acoplamento e alta coesão entre componentes. Testes unitários com Jest asseguraram a confiabilidade do sistema, validado mediante simulações controladas contendo múltiplos estacionamentos.

1. Introdução

Smart Parking [1] é uma das aplicações desenvolvidas com a ambição de utilizar conceitos de IoT (Internet das Coisas) teóricos e aplicá-los em projetos reais dentro do Campus da Unicamp [2]. Ele apresenta uma solução para diminuir o tempo necessário para motoristas conseguirem encontrar um local para estacionar, utilizando-se técnicas de Deep Learning para analisar imagens de câmeras a fim de obter o número de vagas ocupadas em um estacionamento automaticamente, e o disponibilizar em um totem próximo.

Atualmente, existem dois bolsões de estacionamento no campus da Unicamp que estão sendo monitorados por sistemas de câmeras, demonstrando a viabilidade e aplicabilidade do projeto em uma situação real. No entanto, para ampliar e aprimorar essa iniciativa, faz-se necessário desenvolver uma arquitetura de software robusta, escalável e modular, de forma que o software seja capaz de prover o serviço independente do aumento do número de estacionamentos.

Neste contexto, o presente trabalho propõe a criação de uma aplicação baseada em microsserviços, continuando e expandindo o projeto Smart Parking. A escolha por uma arquitetura de microsserviços se justifica pela necessidade de flexibilidade e escalabilidade que este tipo de solução oferece. Ao contrário dos monólitos, que podem se tornar inflexíveis e difíceis de manter à medida que crescem, os microsserviços permitem que diferentes componentes do sistema sejam desenvolvidos, implantados e escalados de forma independente.

Fazendo-se uma revisão da literatura acerca de microsserviços, o objetivo do trabalho é criar uma aplicação capaz de atender os requisitos de um sistema de monitoramento de estacionamento inteligente. A pesquisa abrangerá o estudo de artigos científicos e livros sobre tecnologias de comunicação e armazenamento de dados que auxiliarão no desenvolvimento do projeto tendo como base as melhores e mais consolidadas práticas de programação.

2. Objetivos

Nosso projeto teve como objetivo desenvolver e avaliar uma solução prática para o monitoramento de vagas do estacionamento universitário da Unicamp, o já mencionado Smart Parking. Nos propomos adotar tecnologias de comunicação e armazenamento de dados que permitissem um modelo de sistema que fosse ao mesmo tempo simples, no sentido de não trazer complexidades incidentais desnecessárias, como também facilitasse subsequentes iterações de desenvolvimento, seja para manutenção, aprimoramento da

lógica de negócio ou ainda para o escalonamento do sistema em questão. Para alcançar este objetivo, propusemos a implementação de um sistema composto por um Back-End robusto, um banco de dados MongoDB e uma interface de usuário amigável.

Assim sendo, nossos objetivos específicos foram:

1. **Desenvolver um Back-End integrado com um broker de mensagens**: Implementar um Back-End que utilize um broker de mensagens para gerenciar a comunicação entre os dispositivos IoT que monitoram as vagas de estacionamento e o sistema central. O Back-End deve ser capaz de processar e armazenar os dados recebidos em um banco de dados, garantindo a confiabilidade e a integridade das informações, e o broker adotado deve ter aderência com os 'publishers' (dispositivos IoT).
2. **Implementar uma interface de comunicação para consulta de dados**: Desenvolver uma interface de comunicação que permita a consulta dos dados de ocupação das vagas de estacionamento em tempo real. Essa interface deve estar disponível para se comunicar com o Front-End e, portanto, deve levar em consideração as necessidades específicas desse tipo de comunicação.
3. **Simular o envio de atualizações de ocupação de vagas**: Implementar uma simulação de envio de dados pelos dispositivos IoT, testando a capacidade do sistema de processar e exibir informações atualizadas em tempo real. Esta simulação é importante para validar a funcionalidade do sistema em um cenário operacional.
4. **Desenvolver uma aplicação Front-End**: Criar uma interface de usuário que seja intuitiva e fácil de usar, permitindo que os usuários visualizem a disponibilidade das vagas de estacionamento de maneira rápida e eficaz.
5. **Avaliar a escalabilidade e a modularidade do sistema**: Verificar a capacidade do sistema de ser expandido para monitorar um número maior de vagas ou ser aplicado em outros contextos. A arquitetura deve ser modular, permitindo que componentes como o broker e o Back-End possam ser escalados ou modificados independentemente, conforme necessário.

Este projeto então não apenas tem por objetivo desenvolver uma solução técnica para um problema prático, mas também explorar as práticas em design de software, arquitetura de sistemas e a integração de diferentes tecnologias. Ao final deste estudo, esperamos ter um protótipo funcional e aprendizados sobre a implementação de soluções IoT.

3. Metodologia

A metodologia adotada neste trabalho está estruturada para permitir uma análise abrangente e uma implementação eficaz de um sistema de monitoramento de vagas de estacionamento, utilizando uma combinação de tecnologias IoT, comunicação via broker de mensagens e armazenamento de dados. As fases do projeto são delineadas como segue:

1. **Avaliação Teórica das Tecnologias**: Antes da implementação prática, foi realizada uma revisão teórica para selecionar as tecnologias e ferramentas mais adequadas para cada parte do sistema:
    - **Revisão de Literatura**: Investigação das tecnologias de broker de mensagens, bancos de dados e frameworks de desenvolvimento Front-End e Back-End, focando em suas capacidades, limitações e casos de uso ideais.
    - **Seleção de Tecnologias**: Com base nos critérios de desempenho, escalabilidade, facilidade de implementação e integração, as tecnologias mais apropriadas foram escolhidas para o Back-End, broker de mensagens, banco de dados e Front-End.
2. **Modelagem do Sistema**:
    - **Definição dos Requisitos**: Identificação detalhada das necessidades dos usuários finais e dos requisitos funcionais e não funcionais do sistema, garantindo que o design atende tanto às necessidades operacionais quanto às expectativas de experiência do usuário.
    - **Design de arquitetura**: Elaboração da arquitetura do sistema, desenhando um layout que facilite futuras expansões ou modularizações, incluindo diagramas de fluxo de dados e arquitetura de componentes.
3. **Implementação do Sistema**:
    - **Desenvolvimento do Back-End e Broker de Mensagens**: Implementação do Back-End utilizando a tecnologia de broker de mensagens escolhida, configurando a interação eficiente entre dispositivos IoT e o servidor central.
    - **Configuração do Banco de Dados**: Instalação e configuração do MongoDB, estabelecendo uma estrutura de dados otimizada para consultas rápidas e eficientes.
    - **Criação do Front-End**: Desenvolvimento de uma interface de usuário clara e responsiva, que apresente as informações de forma intuitiva e acessível.
4. **Simulação e Testes**:
    - **Simulação de Envio de Dados**: Realização de testes controlados para simular o envio de dados pelos dispositivos IoT, avaliando a robustez e a resposta do sistema em condições variadas.
    - **Testes de Usabilidade e Funcionalidade**: Verificação da interface do

usuário e das funcionalidades do sistema para garantir que todos os componentes operem conforme o esperado em cenários reais.

5. **Conclusões**:
    - Registro detalhado das etapas de desenvolvimento, desafios enfrentados e soluções adotadas, seguido de uma análise dos resultados obtidos, lições aprendidas e considerações a respeito das possibilidades para futuros projetos e estudos.

Ao final deste projeto, é esperado não apenas apresentar um protótipo funcional do sistema de estacionamento inteligente, mas também fornecer uma análise das tecnologias e abordagens adotadas.

4. Fundamentação Teórica

De modo a ser guiados pelas boas práticas de criação de microsserviços, buscamos literatura: Challenges When Moving from Monolith to Microservice Architecture, Kalske et al. [3], no livro Monolith to Microservices, Sam Newman [4], e em artigos e blogs, a chamada *grey literature*, para auxiliar em questões técnicas de implementação.

4.1 Domain-Driven Design (DDD)

Normalmente, recomenda-se começar a desenhar a arquitetura seguindo padrões de Domain-Driven Design (DDD) [5]. Este conceito propõe que, numa fase inicial, sejam identificados os atores e entidades do sistema e, em seguida, criadas as relações entre eles para obter uma visão mais ampla e geral. Segundo o livro Monolith to Microservices, de Sam Newman, ao decompor um sistema monolítico existente, é essencial possuir uma forma de decomposição lógica, e é aqui que o DDD pode ser extremamente útil. O desenvolvimento de um modelo de domínio ajuda a definir os limites dos serviços e a priorizar a decomposição do sistema, proporcionando uma estrutura clara e lógica para a transição para microsserviços.

4.2 Evitando Anti-Patterns

Um dos aspectos cruciais ao se desenvolver aplicações microsserviços é, corretamente delimitar responsabilidades entre os componentes do macro-sistema e distribuir responsabilidades entre esses módulos para torná-los coesos em si [9]. Isso significa que, quando se constrói um microsserviço com mais funcionalidades do que o planejado ou com bordas mal definidas, ele se torna muito suscetível a formar um

acoplamento rígido com outros módulos. ‘Acoplamento rígido’ (*tight coupling)* é o termo usado para quantificar o quão dependentes dois módulos diferentes são entre si, sendo o objetivo torná-los o mais independentes possível (*loose coupling*). De acordo com Newman, identificar e criar contextos delimitados (*bounded contexts*) é crucial para definir os limites dos microsserviços e evitar problemas de acoplamento apertado.

4.3 Database per Service Pattern

Outro importante padrão utilizado como base para a confecção do trabalho foi *Database per Service Pattern*. Ele dita que cada microsserviço possui seu próprio banco de dados, e este deve ser o único responsável por se comunicar com o banco de dados. Isso significa que outros microsserviços terão de usar uma API mantida por este serviço para acessar o dado requerido, promovendo a independência entre os microsserviços e um ponto focal para limitar o acesso a informações supérfluas conforme o serviço que fez a requisição (segurança). Em nossa implementação, portanto, para que dados sejam inseridos e lidos, haverá de serem criadas interfaces claras de comunicação entre o componente que gerencia o banco de dados e os demais, seja entre os produtores de dados (sensores) quanto da leitura que é feita do mesmo para fins de apresentação (na forma do Front-End para o usuário), de forma que tais componentes não façam acesso direto, mas sim um acesso intermediado.

5. Modelagem da Solução

Primeiramente, foi feita uma análise dos requisitos e regras de negócios que eram esperados do sistema, para conseguir abstrair os maiores problemas e identificar as melhores ferramentas capazes de solucioná-los.

A fim de oferecer uma boa experiência ao usuário, definiram-se os seguintes requisitos:

- O Back-End deve ser capaz de receber e armazenar dados de múltiplos estacionamentos ao mesmo tempo.
- O Front-End deve oferecer ao usuário uma resposta visual à quantidade de estacionamentos cadastrados, assim como seu número total de vagas e o número de vagas disponíveis no momento para cada estacionamento.
- Há um foco na rapidez e eficácia na comunicação entre o dispositivo IoT e o Back-End assim como entre o Back-End e o Front-End.

- Dados armazenados em um banco capaz de fazer buscas de maneira rápida e eficiente, levando-se em conta que os dados devem refletir o mais próximo possível da realidade do estacionamento selecionado.
- Testes unitários para as funcionalidades mais críticas certificando-se de que continuam funcionando corretamente.

5.1 Diagrama da Arquitetura

Levando tais requisitos em consideração, modelou-se a arquitetura apresentada na Figura 1:

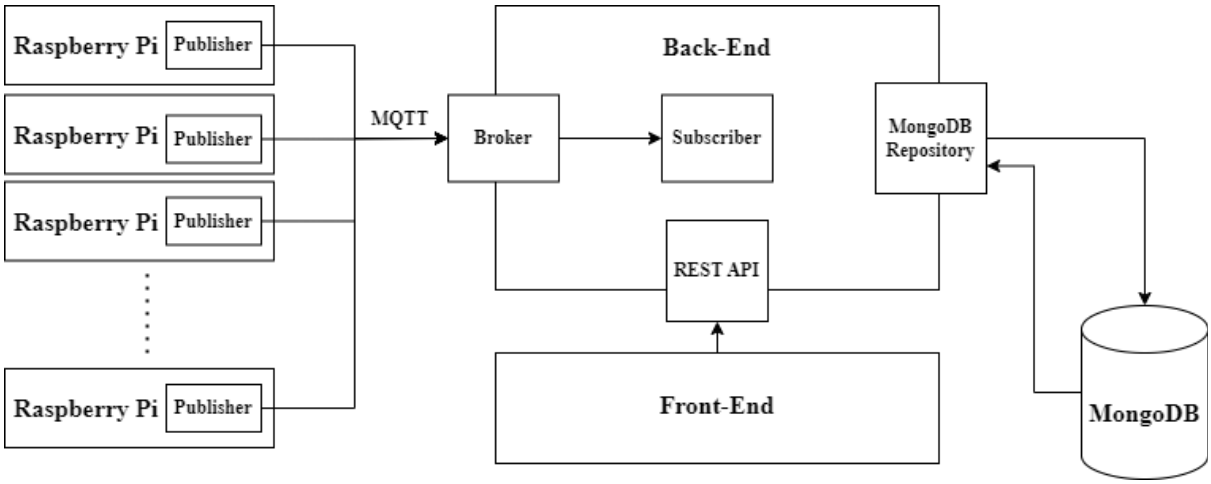

Figura 1: Diagrama da Arquitetura

Na imagem acima, vê-se a representação da arquitetura, levando em conta os componentes distribuídos **Raspberry Pi**, **Back-End, Front-End** e **MongoDB**.

O Raspberry Pi representa o algoritmo Deep Learning que faz a computação do número de vagas, considerando-se as imagens das câmeras de um estacionamento, e depois publica o número de vagas para o Back-End através do protocolo MQTT (o qual será discutido mais adiante). A ideia é fornecer um tipo de comunicação que seja capaz de, tanto desacoplar os serviços, quanto facilitar na escalabilidade do número de publishers.

O Back-End é responsável por administrar as mensagens que chegam através do módulo Subscriber. Para isso, haverá uma lógica de redirecionamento das mensagens, para que, após decodificadas, elas possam ser corretamente armazenadas na *collection* que representa o estacionamento em questão. A identificação do estacionamento (*estacionamento_id*) está incluída no tópico em que o publisher publica as mensagens.

O Front-End possuirá uma tela inicial, onde haverá algumas informações acerca do tema da aplicação, e também a opção de visualizar os estacionamentos já cadastrados. Por

meio de um botão ‘Estacionamentos’ será possível visualizar a lista de Estacionamentos, com algumas informações gerais, como número total de vagas, descrição e *id*. Ao escolher um dos itens da lista, o usuário é direcionado para uma nova tela, onde receberá uma representação visual do atual estado do estacionamento selecionado. Para isso, o Front-End faz uma operação de GET (REST) para o Back-End, que retornará o valor mais recente do MongoDB.

O MongoDB é o banco de dados não relacional escolhido para armazenar todas as informações de número total de vagas e timestamps do momento em que elas foram coletadas. O Back-End será o único serviço capaz de acessá-lo, mantendo assim toda a implementação do Repository em um só lugar.

5.2 Banco de Dados

Para o banco de dados do projeto, optou-se por um banco NoSQL, devido à sua compatibilidade com a ideia de um estacionamento receber diversos valores homogêneos, que poderiam ser modelados em uma única tabela. Não havendo, por exemplo, a necessidade de criar relações complexas entre Entidades (Estacionamento e Vagas), toda a complexidade poderia ser facilmente desenhada através de coleções (collections). Isso reduziria o esforço de busca, alteração e escalabilidade da aplicação no futuro.

Dentre os bancos não-relacionais, foram analisadas algumas opções:

PostgreSQL: Um banco de dados confiável para grandes quantidades de dados, com suporte para esquemas de dados não relacionais. É robusto e amplamente utilizado, mas sua estrutura acaba por ser mais rígida do que a natureza esperada dos dados de IoT.

XTDB: Focado em resolver problemas temporais com esquema bitemporal e gerenciamento de histórico. É poderoso para aplicações que exigem controle temporal rigoroso, mas pode ser complexo para nosso caso.

CouchDB: Também baseado em documentos e usa JSON para armazenar dados. É projetado para trabalhar bem com a web e possui capacidades de replicação integradas úteis para distribuir dados entre diferentes locais. Tem suporte para Node.js através de várias bibliotecas, mas possui uma comunidade muito menor.

MongoDB: Um banco de dados não relacional popular, baseado em documentos, que oferece alto desempenho, alta disponibilidade e fácil escalabilidade. Consegue armazenar dados em documentos semelhantes ao JSON, o que é muito útil para armazenar diferentes tipos de dados dos sensores e as informações de disponibilidade das vagas. Possui um driver bem suportado para Node.js e uma grande comunidade mantenedora.

Dessa forma, a escolha do MongoDB acabou por se mostrar ser a mais apta e promissora para o tema do projeto, refletindo a necessidade de um banco de dados flexível,

escalável e de alto desempenho, garantindo que o sistema de monitoramento de vagas de estacionamento atenda aos requisitos definidos.

5.3 Transmissão de Dados

Para garantir uma comunicação robusta entre os dispositivos IoT e o sistema central, o protocolo MQTT (Message Queuing Telemetry Transport) foi escolhido. Essa escolha se justifica por sua leveza, confiabilidade, escalabilidade e simplicidade, características essenciais para o projeto em questão.

Leveza e Eficiência: É otimizado para redes com baixa largura de banda, ideal para dispositivos IoT com recursos limitados. Isso garante um consumo mínimo de energia e evita sobrecarga da rede.

Escalabilidade e Flexibilidade: É altamente escalável, suportando muitos dispositivos IoT simultâneos sem comprometer o desempenho. Isso permite a expansão do sistema para monitorar mais estacionamentos no futuro.

Simplicidade e Facilidade de Implementação: Apresenta uma sintaxe simples, facilitando a integração com os dispositivos IoT e o sistema central. Isso reduz o tempo de desenvolvimento e garante a manutenção eficiente do sistema.

Para a transmissão de dados para o Front-End, adotamos o padrão de **API REST**, utilizado para expor os dados do MongoDB ao Front-End. Essa provavelmente foi uma das escolhas mais triviais, já que o padrão é amplamente utilizado na indústria e é bastante familiar para os integrantes do grupo.

5.4 Biblioteca para o Front-End

ReactJS é uma popular biblioteca de JavaScript de código aberto utilizada como base no desenvolvimento de aplicações de single-page ou mobile. É usada para construir interfaces de usuário e permite um desenvolvimento mais rápido. É fácil de aprender e possui uma comunidade extensa que conta com diversas grandes empresas. Ele trabalha com o Virtual DOM, o que melhora o desempenho e possui ligação unidirecional, o que facilita o trabalho para os desenvolvedores [7].

6. Desenvolvimento

Para atingir os objetivos desejados, o grupo se dividiu de forma que todas as grandes decisões do projeto fossem discutidas por todos, enquanto aspectos técnicos e específicos da implementação foram segregados como explicitado na Tabela 1:

André	Back-End (MQTT, Interface Banco de Dados)
Tiago	Back-End (Conteinerização, handler, testes, Interface Front-End)
Lucas	Front-End (Consumo da API, Apresentação Visual)

Tabela 1: Participações

6.1 Desenvolvimento do Back-End

A Figura 2 apresenta a interface de como os dados foram planejados para serem armazenados dentro do banco de dados:

```
export interface Estacionamento {
  id: string;
  desc: string;
  total_vagas: number;
}

export interface EstacionamentoVagas {
  estacionamento_id: string;
  vagas_ocupadas: number;
  timestamp_local_coleta: Date;
  timestamp_ack_server: Date;
}
```

Figura 2: Estrutura das *collections* do MongoDB

Optou-se por definir duas interfaces separadas: uma para o Estacionamento e outra para as Vagas de Estacionamento.

A interface Estacionamento abrange as características imutáveis da entidade, incluindo um identificador único (representado por uma *string*), uma descrição (desc) para armazenar informações sobre a localização e características específicas, e o total de vagas (*total_vagas*), que corresponde à capacidade máxima disponível do estacionamento.

A interface EstacionamentoVagas, por sua vez, é responsável por armazenar os dados que são frequentemente atualizados e alimentados pelas análises de Machine Learning implementadas no projeto Smart Parking [2]. O *estacionamento_id* funciona como uma chave estrangeira, conectando esta interface à entidade Estacionamento através do *id*. O atributo *vagas_ocupadas* representa o número de vagas ocupadas, atualizado periodicamente para cada estacionamento e exibido ao cliente. O *timestamp_local_coleta*

indica o momento em que os dados foram coletados, e o *timestamp_ack_server* registra a data de confirmação do recebimento dessas informações pelo servidor.

6.1.1 Padrão: Abstração por Repositório

A adoção do padrão de Abstração por Repositório foi uma decisão estratégica para modularizar o acesso aos dados e aumentar a manutenibilidade do sistema. Implementamos funções que encapsulam as operações de banco de dados, como inserções (INSERT) e consultas (SELECT). Isso possibilita que, por exemplo, a implementação do banco de dados seja substituída se necessário. Com isso, qualquer alteração nas operações de banco de dados pode ser realizada em um único local, minimizando o impacto nas várias partes do sistema que dependem desses dados.

6.1.2 Injeção de dependências e gerenciamento de ciclo de vida

A arquitetura do back-end foi projetada para maximizar a modularidade e facilitar a manutenção através do uso eficaz da injeção de dependências e gerenciamento de ciclo de vida dos componentes. Este design permite que cada componente funcione de forma independente e, ao mesmo tempo, coesa com outros componentes do sistema.

1. **Modularidade**: Cada componente, como o *HttpApiServerComponent*, *MqttClientComponent*, e *MqttBrokerComponent*, é tratado como um módulo isolado que pode ser desenvolvido, testado e atualizado de forma independente. Isso reduz a complexidade do sistema, facilitando a gestão de cada parte sem interferir nas outras.
2. **Separando os Ciclos de Vida**: Gerenciando o ciclo de vida de cada componente de forma independente (inicialização e término controlados explicitamente), o sistema pode garantir que os recursos sejam alocados e liberados, garantindo a estabilidade e a disponibilidade do sistema.
3. **Explicitando a Interdependência**: A injeção de dependências explicita as relações entre os componentes. Por exemplo, o *HttpApiServerComponent* depende do *MongoClient* para funcionamento. Isso torna as dependências entre os módulos claras e gerenciáveis, reduzindo o acoplamento e facilitando mudanças ou substituições de componentes.

As bibliotecas de gerenciamento de estado do Clojure, como *Integrant* e *Component* de Stuart Sierra foram uma inspiração no quesito de gerenciamento de ciclo de vida e injeção de dependências. A biblioteca *Component*, por exemplo, cria um sistema de componentes reutilizáveis onde cada componente gerencia seu próprio estado e dependências. Similar ao *MqttBrokerComponent* ou *HttpApiServerComponent*, cada

componente pode ser iniciado e parado independentemente, facilitando a gestão de recursos e o desenvolvimento isolado (como é exemplificado na Figura 3).

```
async function main() {
  const client = new MongoClient(uri);
  await client.connect();

  // ...

  const httpServer = new HttpApiServerComponent({ client });
  const mqttClient = new MqttClientComponent({ url:
'mqtt://localhost:1883', estacionamentoIds: estacionamentoIds,
estacionamentoVagasCollection });
  const mqttBroker = new MqttBrokerComponent();

  const gracefulShutdown = async () => {
      await httpServer.stop();
      await mqttClient.stop();
      await mqttBroker.stop();
      await client.close();
      process.exit(1);
  }

  try {
      await httpServer.start();
      await mqttClient.start();
      await mqttBroker.start();

      process.on('SIGINT', gracefulShutdown);
      process.on('SIGTERM', gracefulShutdown);

  } catch (error) {
      console.error('Error starting components:', error);
      await gracefulShutdown();
  }
}
```

Figura 3: Injeção de dependência e gerenciamento de ciclo de vida de cada componente

6.1.3 Conteinerização dos Serviços

Utilizamos a ferramenta *Docker* para a conteinerização dos serviços, o que proporciona uma série de vantagens, como isolamento de ambiente, replicação fácil de serviços, e a garantia de que o software se comporte de maneira consistente em diferentes ambientes de desenvolvimento, teste e produção.

6.1.4 Testes com Jest

Adotamos o *Jest* como nossa ferramenta de testes unitários, devido à sua vasta capacidade de integração com projetos *JavaScript/TypeScript*. Os testes unitários nos dão mais segurança de que o código funciona conforme esperado, nos dando maiores garantias que cada aspecto do código funciona em unidade, além de também facilitar a migração desses testes para outros microsserviços, se necessário. Cada componente do nosso sistema tem seu conjunto de testes, que podem ser executados de maneira isolada, garantindo que cada parte do sistema esteja boa individualmente.

6.2 Desenvolvimento do Front-End

O desenvolvimento do front-end do sistema Smart Parking Unicamp foi guiado pela intenção de criar uma interface de usuário moderna, eficiente e modular, que pudesse se integrar adequadamente com uma arquitetura baseada em microsserviços. O objetivo deste módulo do projeto é fornecer aos usuários uma aplicação prática e intuitiva para consultar a situação dos estacionamentos cadastrados em tempo real, de forma simples (Figura 4 mostra o design minimalista do front-end). Escolhemos o React.js para esta tarefa, devido às suas características que facilitam a construção de aplicações escaláveis e de fácil manutenção.

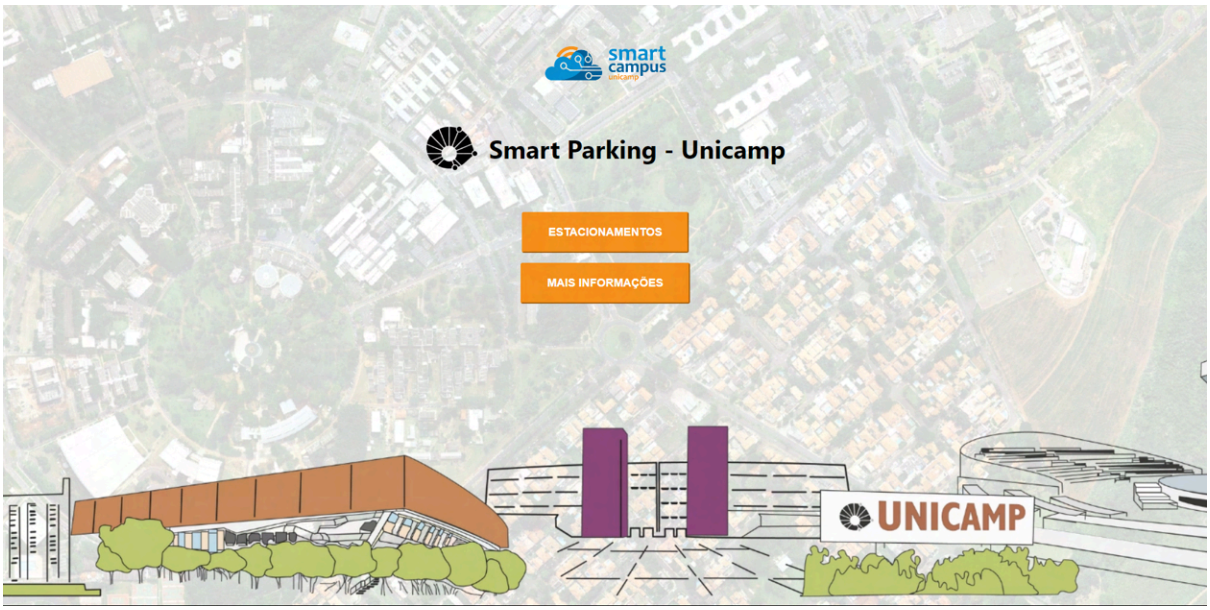

Figura 4: Página inicial do Smart Parking

6.2.1 Arquitetura

Conforme destacado no artigo sobre a integração do React com microsserviços [6], "No contexto de microsserviços, o React permite a construção de aplicações com componentes isolados que podem ser desenvolvidos e atualizados de forma independente. Isso é essencial para manter a modularidade e a capacidade de escalar a aplicação de forma eficiente. Cada componente pode ser integrado como um serviço separado, facilitando a manutenção e a atualização contínua da interface do usuário". A seguir, discutimos como estas e outras características do React contribuíram para essa decisão e como elas se relacionam com os princípios dos sistemas distribuídos:

Componentização, Reutilização e Desacoplamento: Como dito anteriormente, o React permite a construção de componentes isolados e reutilizáveis, facilitando a manutenção e evolução da aplicação. Essa abordagem reflete o conceito central dos microsserviços, onde cada serviço é uma unidade autônoma que pode ser modificada sem impactar outras partes do sistema. A arquitetura modular do React promove um alto nível de desacoplamento, garantindo que mudanças em um componente não afetem os demais.

- **Exemplo Prático**: No nosso projeto, cada tela da aplicação foi implementada como um componente separado, como *Home.tsx*, *Estacionamentos.tsx* e *EstacionamentoInfo.tsx*. Cada componente encapsula sua própria lógica e estado, o que permite que sejam desenvolvidos e testados de forma independente.

Gerenciamento de Estado Local: Utilizando hooks como *useState* e *useEffect* (Figura 5), o React facilita o gerenciamento de estado local dentro dos componentes. Esse gerenciamento de estado local permite que cada componente mantenha seu próprio estado, evitando dependências complexas.

- **Exemplo Prático**: Em *Estacionamentos.tsx*, utilizamos o *useState* para gerenciar a lista de estacionamentos que é carregada pela função *getEstacionamentos*. Isso permite que o componente mantenha e atualize seu estado de forma independente de outros componentes.

```
useEffect(() => {
 async function fetchData() {
   try {
     const estacionamentoData = await getEstacionamentoByID(id);
     const vagasData = await getVagasByID(id);
     setEstacionamento(estacionamentoData[0]);
     setVagas(vagasData);
     setListaDeCarros(ListadeCarros(estacionamentoData[0],
vagasData));
   } catch (err) {
     setError('Erro ao obter dados do estacionamento.');
   } finally {
     setLoading(false);
   }
 }
 fetchData();
}, [id]);
```

Figura 5: Utilização do *useEffect* no gerenciamento de estado local

Renderização Eficiente com o Virtual DOM: O Virtual DOM do React melhora a eficiência da renderização da aplicação, atualizando apenas os componentes que sofreram mudanças (Na figura 6, vê-se a tela *EstacionamentoInfo* em que apenas a representação visual das vagas ocupadas é atualizada conforme a resposta da chamada feita ao back-end, *getEstacionamentoByID*). Em um sistema distribuído, onde a latência e a eficiência são preocupações constantes, essa característica ajuda a manter a performance da interface do usuário. Além disso, o React facilita a criação de componentes reutilizáveis da interface, que exibem informações dinâmicas e se atualizam conforme necessário [8].

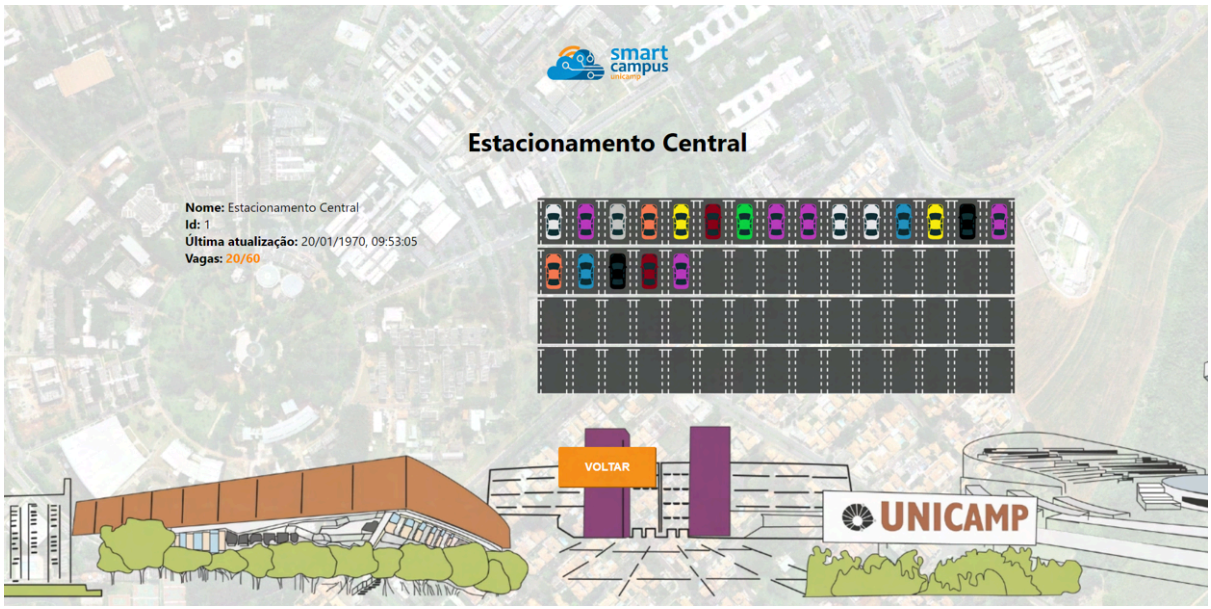

Figura 6: Tela de detalhes de um estacionamento

6.2.2 Comunicação com o Back-End

No contexto de sistemas distribuídos, a comunicação eficiente entre o front-end e os microsserviços do back-end é essencial. Implementamos essa comunicação utilizando APIs RESTful, que permitem que o front-end requisitar dados de maneira assíncrona, garantindo que a interface do usuário esteja sempre atualizada com as informações mais recentes fornecidas pelos serviços do back-end.

- **Exemplo Prático**: No arquivo *service.tsx*, definimos funções assíncronas como *getEstacionamentos*, *getEstacionamentoByID* (Figura 7) e *getVagasByID*, que fazem chamadas HTTP para os endpoints dos microsserviços. Utilizamos o fetch para realizar essas chamadas, o que permite que a aplicação obtenha dados em tempo real.

```
async function getEstacionamentoByID(id: string | undefined):
Promise<IEstacionamento[]> {
  const response = await
fetch(`http://localhost:3000/estacionamentos/${id}`);
  if (!response.ok) {
     throw new Error(`Erro ao obter estacionamento com ID
${id}.`);
  }
```

```
  const data: IEstacionamento[] = await response.json();
  return data;
}
```

Figura 7: Chamada assíncrona para o Back End

Essa abordagem assíncrona é compatível com o React, projetado para lidar com atualizações de dados em tempo real de maneira eficiente. A utilização deste tipo de chamada garante que a interface do usuário permaneça responsiva, mesmo quando está aguardando dados de múltiplos serviços distribuídos.

6.2.3 Desafios

- **Gerenciamento de Estado Global**: Embora o gerenciamento de estado local seja simples com React, a coordenação entre múltiplos estados locais e a necessidade de um estado global coerente pode se tornar complexa. Ferramentas adicionais, como Redux, podem ser consideradas para gerenciar estados globais em aplicações maiores, como exemplo, caso fosse decidido expandir o sistema para múltiplos campi da Unicamp.
- **Consistência de Dados**: Garantir a consistência dos dados entre o front-end e os múltiplos microsserviços do back-end é um desafio constante. Mecanismos robustos de sincronização e manejo de erros são essenciais para manter a integridade e a confiabilidade da aplicação, o que novamente pode não aparentar um grande problema para a situação atual do sistema, porém visando casos em que poderíamos ampliar a gama de recursos do front-end, realizando mais requisições para diferentes microsserviços, esta seria uma questão que precisaria ser devidamente analisada.

7. Conclusões e Trabalhos Futuros

A implementação de um protótipo de estacionamento inteligente com uma arquitetura integrando tanto o front-end em React.js quanto o back-end com MongoDB, MQTT e uma API REST, demonstrou ser uma solução satisfatória. A flexibilidade do schema de dados proporcionada pelo MongoDB e a comunicação eficiente e aderente a dispositivos IoT do MQTT demonstraram ser escolhas apropriadas para lidar com as interações frequentes entre os dispositivos conectados ao servidor.

Além disso, o desenvolvimento do front-end utilizando React.js encaixou bem com os princípios de microsserviços e sistemas distribuídos. Sua arquitetura modular, a

eficiência na renderização e a capacidade de comunicação assíncrona com o back-end foram fundamentais para proporcionar uma interface de usuário intuitiva. No entanto, ficou evidente a necessidade de algumas mudanças para garantir a consistência de dados e a confiabilidade no gerenciamento de estados globais, considerando especialmente o potencial crescimento no número de microsserviços que o front-end precisaria gerenciar.

7.1 Melhorias e Expansão Futura

Para melhorar ainda mais a robustez e escalabilidade do sistema, foram identificadas várias áreas que poderiam ser exploradas em trabalhos futuros:

- **Desacoplação do MQTT Broker**: Isso permitiria que, mesmo com o back-end indisponível, o broker pudesse manter os dados mais recentes enviados pelos sensores, assegurando a integridade das informações até que o sistema fosse restabelecido.
- **Criação de Endpoints para Cadastro de Novos Estacionamentos**: Implementar uma interface administrativa no front-end que permita a usuários autorizados adicionar novos estacionamentos à rede. Isso facilitaria a expansão e gestão do sistema, integrando novas funcionalidades de forma transparente.
- **Melhorias na Gestão de Estado Global**: Explorar tecnologias de gerenciamento de estado, como Redux ou Context API do React, para melhorar a consistência e confiabilidade do estado global do front-end.
- **Automação de Testes e Integração Contínua**: Desenvolver testes automatizados mais abrangentes e usar integração contínua para minimizar riscos de defeitos e interrupções, garantindo que novas atualizações e funcionalidades sejam testadas antes de sua integração.

Essas melhorias não apenas aumentariam a usabilidade do aplicativo de Estacionamento Inteligente da Unicamp, mas também forneceriam uma base mais sólida para futuras expansões e adaptações do sistema para atender a demandas emergentes ou mudanças no ambiente operacional.

Referências

[1] SmartParking. *SmartParking A smart solution using Deep Learning*. url: https://smartcampus.prefeitura.unicamp.br/pub/artigos_relatorios/PFG_Joao_Victor_Estacionamento_Inteligente.pdf

[2] Smart Campus. *Site da Smart Campus da Unicamp*. url: https://smartcampus.prefeitura.unicamp.br/

[3] Kalske, M., Mäkitalo, N., Mikkonen, T. (2018). Challenges When Moving from Monolith to Microservice Architecture. In: Garrigós, I., Wimmer, M. (eds) Current Trends in Web Engineering. ICWE 2017. Lecture Notes in Computer Science(), vol 10544. Springer, Cham. https://doi.org/10.1007/978-3-319-74433-9_3

[4] Sam Newman. *Monolith to Microservices*. 1. ed. Sebastopol: O'Reilly Media, 2019.

[5] Millet, Scott; Tune, Nick (2015). Patterns, Principles, and Practices of Domain-Driven Design.

[6] Davide Taibi, Valentina Lenarduzzi, Claus Pahl, and Andrea Janes. 2017. Microservices in agile software development: a workshop-based study into issues, advantages, and disadvantages. In Proceedings of the XP2017 Scientific Workshops (XP '17). Association for Computing Machinery, New York, NY, USA, Article 23, 1–5. https://doi.org/10.1145/3120459.3120483

[7] C. Ritwik, A. Sandeep, ReactJS and front-end development. Int. Res. J. Eng. Technol. (IRJET) 07(04), | Apr (2020). e-ISSN: 2395-0056

[8] NANDA, Satyasai Jagannath; YADAV, Rajendra Prasad; GANDOMI, Amir H. *Data Science and Applications: Proceedings of ICDSA 2023, Volume 3*. Springer Singapore, 2024.

[9] SearchMyExpert. (2023). React and Microservices: A Perfect Match. url: https://blog.searchmyexpert.com/react-microservices/