UNIVERSIDADE REGIONAL INTEGRADA DO ALTO URUGUAI E DAS MISSÕES
CAMPUS DE ERECHIM
DEPARTAMENTO DE ENGENHARIAS E CIÊNCIA DA COMPUTAÇÃO
CURSO DE CIÊNCIA DA COMPUTAÇÃO

JOÃO VITOR VERONESE VIEIRA
KELWIN KOMKA
VINICIUS EMANOEL ANDRADE

GerenciaDocker:
Sistema para gerenciar contêineres

ERECHIM - RS
2019

SUMÁRIO

LISTA DE ILUSTRAÇÕES

1 DESCRIÇÃO DO PROBLEMA

Em virtude da dinamicidade e agilidade necessárias em tarefas comuns para uma empresa de *software*, tal como a disponibilização de aplicações para clientes ou mesmo a configuração de ambiente para novos colaboradores na equipe de desenvolvimento, criou-se o conceito técnico de *conteinerização*. De modo resumido, esse conceito pode ser descrito, segundo (**??**), como "o processo de distribuir uma aplicação de *software* de maneira compartimentada, portátil e autossuficiente". Isto é, uma forma de criar um ambiente completo de qualquer aplicação desenvolvida e "empacotá-lo", para posteriormente distribuí-lo e utilizá-lo.

Dentro desse cenário e tendo em vista os diversos benefícios que essa prática traz aos seus utilizadores, diversas ferramentas foram criadas. No entanto, um *software* em específico acabou destacando-se como o mais utilizado quando deseja-se implantar essa tecnologia. O ***docker*** permite o gerenciamento completo de todos os *containers* criados e, devido às suas funcionalidades, ganhou notoriedade na comunidade.

No entanto, a utilização diária dessa ferramenta, geralmente realizada através de um terminal, pode se tornar uma tarefa desnecessária e até mesmo complicada, principalmente para um profissional iniciante, pois seu ambiente pode conter diversas especificidades (tal como vários *containers* executando em paralelo) que, especialmente nos primeiros contatos com a ferramenta, podem ser difíceis de serem implementadas.

Além disso, vale ressaltar que, se necessário, o usuário deve controlar a rede (*docker network*) em que os *containers* estão sendo executados, o que acaba gerando ainda mais dificuldades. Com isso, fica nítido que, apesar de ser um recurso que proporciona inúmeras vantagens aos usuários, ainda existe uma barreira de adoção à essa tecnologia, principalmente em um contexto organizacional, pois o profissional que se dispõe a aprender essa nova ferramenta, precisará conciliar esse aprendizado com a realização de todas as demais atividades tradicionais que ele já é encarregado atualmente.

1.1 Justificativa

Baseando-se nos motivos descritos no capítulo **??** e com o desejo de aprender mais sobre essa moderna ferramenta, o grupo considerou que um monitor *web* que abstraísse essas dificuldades de gerenciamento em uma interface intuitiva e amigável para o usuário seria, além de uma ferramenta útil para os profissionais que se enquadram nessa situação, um bom assunto para ser o tema deste projeto.

2 OBJETIVOS

2.1 Objetivo Geral

- Criar uma ferramenta que possibilite gerenciar os *containers* em execução na máquina

2.2 Objetivos Específicos

- Tornar a ferramenta flexível, permitindo que o usuário escolha o sistema operacional do *contêiner* (com 4 opções)
- Construir uma interface amigável e intuitiva para o usuário
- Executar corretamente o algoritmo (*Adaptive-DSD*), que detectará falha nos *containers*
- Criar um aplicativo complementar à solução, que será responsável por disparar uma notificação sempre que uma falha ocorrer e trará mais mobilidade para o usuário

3 METODOLOGIA

Sempre que almeja-se alcançar um objetivo ao final de uma tarefa, trabalho ou projeto, naturalmente o indíviduo ou equipe responsável por essa demanda irá adotar certas práticas que, no seu modo de ver, irão lhe ajudar a chegar mais assertivamente no resultado esperado. Essas práticas nada mais são do que a metodologia que será adotada para a execução da sua incumbência, isto é, (**??**) "a sistematização utilizada para alcançar um resultado".

Quando um trabalho é consideravelmente grande (tal como esse projeto, por exemplo), provavelmente ele será executado por várias pessoas e, naturalmente, quanto maior for a equipe, mais difícil será organizar as tarefas de cada integrante, bem como suas responsabilidades. Em virtude dessa complexidade de controle, ao longo do tempo, diversas metodologias de gerenciamento de projetos foram criadas, tais como: XP, Scrum, Kanban, Price2, Agile, entre outras, com o objetivo de auxiliar empresas e equipes profissionais a controlarem melhor suas demandas e entregas.

Apesar deste projeto ser um tanto complexo e a equipe realizadora do mesmo ser formada por três integrantes, nenhuma dessas metodologias foi implantada do início ao fim pois, no ponto de vista dos integrantes, seria um controle exagerado e perderia-se mais tempo seguindo as práticas indicadas do que realizando as tarefas propostas. Porém, isso não quer dizer que não houve organização no trabalho - pelo contrário -, quer dizer apenas que utilizou-se somente aquilo que seria útil para a equipe e que realmente seria implantado.

Um exemplo claro dessa "utilização sob demanda" é a *definição de prazos*. O Scrum, uma das metodologias citadas acima e bastante utilizada no ramo de desenvolvimento de *software*, por exemplo, tem como uma das suas principais características o *sprint*, que é (**??**) "uma forma de facilitar a divisão de um projeto em etapas ao longo do tempo". Ou seja, sua principal função é definir entregas esperadas ao longo da duração do projeto e, apesar de não utilizar o Scrum em sua essência, o grupo definiu prazos para as tarefas desde o início do projeto e procurou respeitá-las sempre que possível.

Da mesma forma, diversas outras práticas recomendadas por metodologias consagradas de gerenciamento de projetos foram utilizadas pelo grupo para que fosse possível chegar à versão final dessa solução. Alguns exemplos de práticas adotadas são: reuniões remotas através do *software Hangouts*; reuniões presenciais para definição de itens; organização das tarefas e atribuição de responsáveis utilizando o *software Trello*; versionamento do código da aplicação com o *GIT* e criação do documento final utilizando LaTeX.

Portanto, adotando essas práticas sempre que necessário e procurando respeitar os prazos definidos, a equipe conseguiu auto-orgarnizar-se e entregar suas obrigações sem maiores problemas. Além disso, é possível afirmar que a experiência foi agradável e não houve nenhum tipo de problema com gestão das tarefas para cada integrante, o que possibilitou atingir um resultado que atendeu às expectativas iniciais.

4 FERRAMENTAS UTILIZADAS

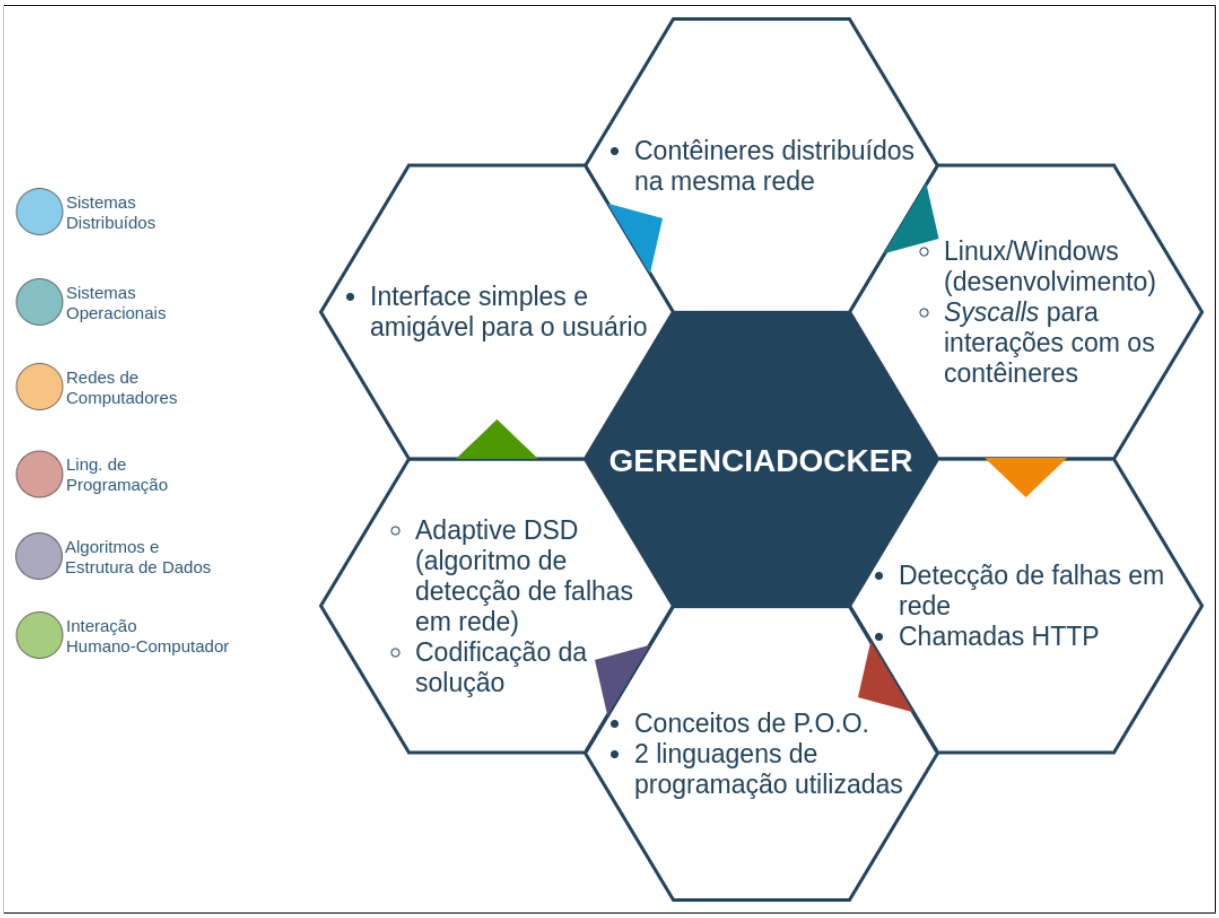
Esse capítulo tem por objetivo listar e definir, de forma resumida, todas as ferramentas que foram necessárias para que a versão final do projeto fosse criada. As ferramentas listadas abaixo abordam desde a criação do projeto e hospedagem do seu repositório em uma plataforma *on-line* até a documentação do mesmo e apresentação final:

- **Adaptive DSD:** algoritmo de diagnóstico/detecção de falhas em um sistema distribuído.
- **Bootstrap:** *kit* de ferramentas de código aberto para desenvolvimento com HTML, CSS e JS.
- **CSS:** linguagem que descreve o estilo de um documento HTML.
- **Docker:** ferramenta para facilitar a criação e execução de aplicação utilizando *containers*.
- **Firebase:** plataforma de desenvolvimento de aplicativos que funciona como *BaaS* (*Backend as a Service*).
- **GIT:** sistema de controle de versões distribuído.
- **GitHub:** plataforma *on-line* para armazenar o repositório do projeto.
- **HTML:** linguagem de marcação padrão para páginas da *web*.
- **Insomnia:** cliente multiplataforma para requisições *GraphQL* e *REST*.
- **JavaScript:** linguagem de programação.
- **Markdown:** linguagem simples de marcação em texto puro.
- **Mozilla, Chrome:** *browsers*.
- **Oh My Zsh:** *framework* de código aberto para gerenciar a configuração *zsh* do usuário.
- **Postman:** plataforma de colaboração para o desenvolvimento de *API* (Application Programming Interface).
- **PWA:** (*Progressive Web Applications*) é um tipo de *software* aplicativo fornecido pela *web*.
- **Python:** linguagem de programação.
- **React:** biblioteca JavaScript para criar interfaces de usuário.
- **Visual Studio Code:** editor de textos utilizado para a codificação da solução.
- **Visual Studio Code (Plugins):** extensões adicionadas ao editor de textos para facilitar o trabalho ou aumentar a produtividade, tais como:
  - *LaTex Workshop*
  - *React-Native/React/Redux snippets for es6/es7*
  - *Auto Import*
  - *JavaScript (ES6) code snippets*
  - *Live Share*

4.1 Áreas Envolvidas

Essa seção irá apresentar todas as disciplinas que foram envolvidas na realização deste trabalho, uma vez que o objetivo da matéria de **Laboratório de Desenvolvimento** é integrar os conhecimentos já vistos em outras cadeiras do curso e colocá-los em prática em um projeto que solucione ou aborde um problema real do cotidiano. No entanto, ao invés de escrever vários parágrafos que descreveriam cada uma das disciplinas e em qual parte do projeto seus conceitos foram necessários, o grupo optou por criar um diagrama:

Figura 1 – Diagrama de Relação: Disciplinas x Projeto



O objetivo deste diagrama é proporcionar uma visualização mais simples dessa relação "disciplinas x projeto", além de facilitar a leitura do documento como um todo. Como é possível ver na figura **??**, ao centro do diagrama está o projeto e, circundando-o, estão todas as tarefas realizadas durante sua execução. Por fim, para ilustrar à qual disciplina cada tarefa pertence, pode-se notar uma legenda na parte esquerda da figura.

5 DESCRIÇÃO DA IMPLEMENTAÇÃO

5.1 *Adaptive-DSD*

O *Adaptive-DSD* (*Adaptive Distributed System-Level Diagnosis*) é um algoritmo para diagnóstico em redes completamente conectadas e seu funcionamento é, ao mesmo tempo adaptativo e distribuído. Foi desenvolvido para que cada máquina que possua o algoritmo em execução possa realizar o teste e também ser testada por outras máquinas na mesma rede. É caracterizado como adaptativo por não depender e nem restrigir o número de máquinas na rede, necessitando, no mínimo, de uma máquina para realizar o teste. Para a execução dos testes, não é levado em consideração falhas na rede, pois o objetivo deste algoritmo é testar o processamento ou funcionamento específico dos processos nas máquinas.

5.1.1 Funcionamento

O algoritmo possui duas listas, que possuem o tamanho exato do número de máquinas conectadas à rede atual: o vetor TESTED_UP, que irá guardar na posição da máquina atual, o índice da máquina testada que possui funcionamento normal; e o vetor STATE, que armazena o estado das máquinas, atribuindo inicialmente o valor FALHO para todas e atualizando o valor de cada máquina para NORMAL sempre que a mesma tiver seu funcionamento correto confirmado. À cada rodada, os vetores são atualizados e enviados às outras máquinas na rede.

Na primeira rodada, uma máquina irá iniciar o teste seguindo a lista de máquina existentes e disponíveis na rede. Esta máquina irá percorrer a lista de máquinas e fará uma requisição de teste à próxima máquina da lista. No caso da máquina que será testada retornar uma resposta de funcionamento correto, a máquina que está realizando o teste atualiza os dados agregados até o momento e envia à máquina testada que, por sua vez, irá executar o mesmo processo com a máquina seguinte, até que todas as máquinas tenham sido testadas. Por outro lado, se a máquina testada retornar algum erro, será marcada como falha, e a máquina que está testando irá testar a próxima máquina da lista, até encontrar outra máquina com funcionamento normal ou até que a lista de máquinas disponíveis acabe.

A segunda rodada de testes será para atualizar as informações de todas as máquinas na rede sobre o estado de funcionamento de cada máquina. Inicialmente, a primeira máquina com funcionamento normal irá verificar na lista de máquinas, qual a próxima máquina funcionando, e irá enviar os dados da rede. Ao receber esses dados, a máquina receptora irá prosseguir com a distrbuição de informações.

5.1.2 Algoritmo

O algoritmo *Adaptive-DSD* inicia sua execução abrindo uma conexão em uma porta específica - através de um *socket* - e fica escutando esta porta até que uma conexão seja estabelecida por outra máquina. Ao final de toda requisição realizada por outra máquina, o algoritmo volta a escutar e aguardar uma nova conexão (também por *socket*) com a porta.

A linha 2 é utilizada para receber a informação do ip e porta da máquina que devem ser usados no *socket* e a linha 3 vincula estas informações ao *socket*. A linha 6 disponibiliza o *socket* para conexões na porta previamente configurada e, por fim, quando uma conexão for estabelecida, o método **ReceberRequisicao** é executado.

```
1    tcp = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
2    tupla = (ip_host, int(porta_host))
3    tcp.bind(tupla)
4    tcp.listen(1)
5
6    conexao, cliente = tcp.accept()
7    ReceberRequisicao(conexao)
```

O método **ReceberRequisicao** direciona o algoritmo para o fluxo requisitado pela máquina que está conectando com a máquina atual. As seguintes mensagens podem ser enviadas para realizar determinadas ações:

1. **“start”**: mensagem enviada pelo gerenciador para realizar uma verificação das máquinas. A máquina será considerada a primeira da lista (posição 0) e terá o estado definido como NORMAL. Ao receber esta mensagem, irá executar o método **IniciarTeste**.
2. **“check”**: mensagem enviada pela máquina que está realizando o teste no momento. Ao receber esta mensagem, a máquina atual irá realizar uma verificação de funcionamento e irá retornar se possui falha ou não.
3. **“keepTest”**: mensagem enviada pela máquina que está realizando o teste, caso a máquina atual esteja com funcionamento NORMAL. Informa a máquina atual para dar continuidade ao teste de funcionamento.
4. **“keepInfo”**: mensagem enviada por uma máquina na lista de máquinas com status NORMAL. Informa a máquina atual para manter as informações do teste realizado e prosseguir com a distribução da informação.
5. **“info”**: mensagem enviada pelo gerenciador para receber as informações do ultimo teste. A máquina atual irá retornar ao gerenciador o status de cada máquina da rede.

```
1     msg = ReceberResposta(conexao)
2     if msg == "start":
3         IniciarTeste(conexao)
4     elif msg == "check":
5         RealizarVerificacao(conexao)
6     elif msg == "keepTest":
7         ContinuarTeste(conexao)
8     elif msg == "keepInfo":
9         ManterInformacao(conexao)
10    elif msg == "info":
11        RetornaInformacao(conexao, False)
```

O método **IniciarTeste** inicia recebendo do gerenciador uma lista das máquinas, contendo o IP e porta de cada uma, para conexão. Cria as listas TESTE_UP e STATE e inicia o teste.

O método **TestarMaquina** percorre a lista de máquinas, executa o método **CriarConexao** para criar a conexão com a máquina a ser testada e envia a mensagem **"check"**. Caso a máquina testada retorne a confirmação de funcionamento, a máquina atual procede em enviar as informações existentes à máquina testada. A execução continua, primeiramente, enviando a mensagem **"keepTest"** (linha 5) para informar a outra máquina que esta deve dar continuidade ao teste. Essa, por sua vez, após a confirmação, envia o vetor de máquinas da rede (linha 6), o vetor TESTE_UP(linha 7) e o vetor STATE (linha 8). Caso a máquina testada retorne algum tipo de erro, esta será marcada com um "X" (linha 11) no vetor TESTED_UP e como "FALHO" no vetor STATE (linha 12). Ao chegar na última máquina da lista, o método **DistribuirInformacao** é executado.

```
1     maquina = CriarConexao(host, porta)
2     msg = EnviarInformacao(maquina, "check")
3     if msg == "OK":
4         maquina = CriarConexao(host, porta)
5         EnviarInformacao(maquina, "keepTest")
6         EnviarInformacao(maquina, json_maquinas)
7         EnviarInformacao(maquina, json_tested)
8         EnviarInformacao(maquina, json_state)
9         break
10    else:
11        tested_up[index] = "X"
12        state[index] = "FALHO"
13        index = index + 1
```

O método **ContinuarTeste** recebe as informações coletadas até o momento, enviando uma confirmação à cada requisição. Após isto, uma verificação é realizada para direcionar o teste para próxima máquina da lista ou para iniciar a distrbuição das informações, caso não haja mais máquinas não testadas na rede.

O método **ManterInformacao** recebe as informações do teste finalizado, enviando uma confirmação à cada envio. Por fim, a máquina envia estas informações à próxima máquina que possua o status "NORMAL". Por fim, o método **RetornaInformacao** retorna as informações do teste. Caso o parâmetro **verificacao** seja verdadeiro, ao enviar uma informação, a máquina irá esperar por uma resposta de confirmação; caso contrário, irá apenas enviar as informações, sem esperar por uma resposta de confirmação.

5.2 *API*

A utilização de uma *api* surgiu com a necessidade de um *backend* flexível, que se comunicasse com processos do Sistema Operacional, através de *syscalls* ou de *sockets* e também com uma página *web* (*frontend*) através do protocolo *HTTP*. Seu papel no projeto é funcionar como uma interface para o *frontend*, abstraindo, facilitando e fazendo uma "ponte" entre o que ele precisa e o que o Sistema Operacional deve fazer.

Devido à esses argumentos, foi decidido utilizar o *Flask*, que é um *framework* da linguagem *Python* e funciona como um *WebService*. A soma da praticidade do *Flask* (proporcionando toda a arquitetura em cima do protocolo *HTTP*) e a robustez do próprio *Python*, simplificou a conexão entre todas as partes do sistema e fez jus à escolha de termos uma *Api Rest*.

5.2.1 *Flask*

Como anteriormente mencionado, a *api* foi escrita em *Flask*, cujo, possui uma característica interessante, ele é um *MicroFramework*, ou seja, traz consigo somente o que é necessário para ser executado. O intuito dessa abordagem de *MicroFramework* é fazer com que o programador monte sua própria arquitetura e não sobrecarregue seu projeto.

Outro ponto importante a ser mencionado, é que a Api adota o padrão *MVC* (*Model*, *View* e *Controller*), insentando-se da camada de *View*, pois essa responsabilidade é do *frontend*. A adoção desse padrão acabou facilitando muito o desenvolvimento, pois proporcionou um *software* mais organizado e manuteível. O comportamento da *api* dentro desse escopo será detalhado no próximo subcapítulo.

5.2.2 Arquitetura

A declaração das rotas da *api* ficam dentro de suas *controllers*, sempre mantendo uma semântica, por exemplo, na *controller* **container** temos a rota **/container/iniciar**. As *controllers* servem para receber o *json* enviado pelo *frontend*, validá-lo e por fim enviar uma resposta. Vale ressaltar, na declaração delas é possível restringir quais verbos do protocolo *HTTP* são aceitos.

As *models* da aplicação tem a regra de negócio, as *syscalls* e a comunicação via *socket*. Para realizar as *syscalls* foi necessário utilizar o *subprocess*, uma biblioteca do *Python* que permite a execução e a captura dos resultados. Outra dependência importante no projeto é o *requests*, ele que permite o envio de um *post* para o *firebase*, que faz parte da arquitetura da *PWA*. Por fim, temos a conexão via *socket* com o *Adaptive-DSD*, nela é enviado um conjunto de *bytes* que fazem com que o algoritmo acima descrito seja executado, que por sua vez, retorna o estado atual dos containers.

5.3 *Frontend*

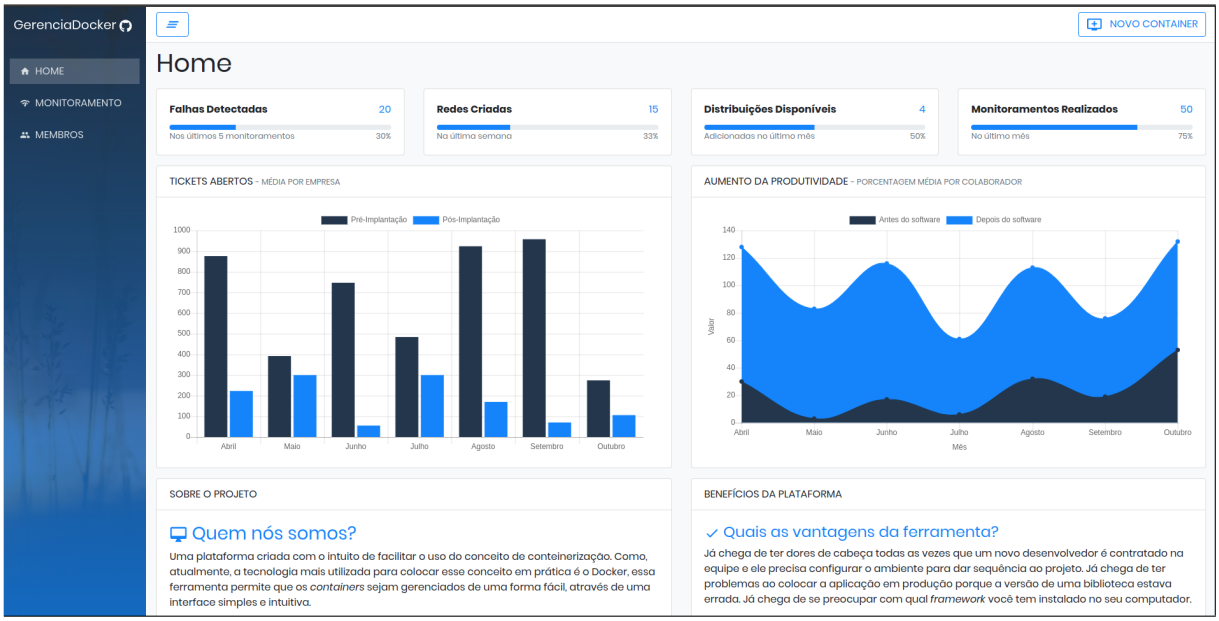
Desde o princípio do projeto, os integrantes do grupo optaram por modulizar a solução, encarregando cada parte com sua respectiva função, tal como: o *Adaptive-DSD* (subcapítulo **??**) seria responsável apenas por detectar falhas na rede; a *API* (subcapítulo **??**) faria a comunição entre os módulos; o *frontend* (subcapítulo atual) seria responsável por apresentar ao usuário uma tela simples e intuitiva e a *PWA* (subcapítulo **??**) é um complemento ao projeto, que servirá para dar mais mobilidade ao utilizador da plataforma.

Esta decisão possibilitou que o *frontend* fosse criado com foco total em proporcionar a melhor experiência possível para o usuário, consumindo e exibindo todas informações disponibilizadas pela *API*. Tendo em vista que o *Adaptive-DSD* será executado em intervalos de tempo, as páginas precisariam repassar esse dinamismo ao usuário e, buscando obter caraterística, toda a parte visual do projeto foi construída utilizando *React* - uma biblioteca JavaScript focada na criação de interfaces de usuário, desenvolvida pelo Facebook.

5.3.1 A Interface

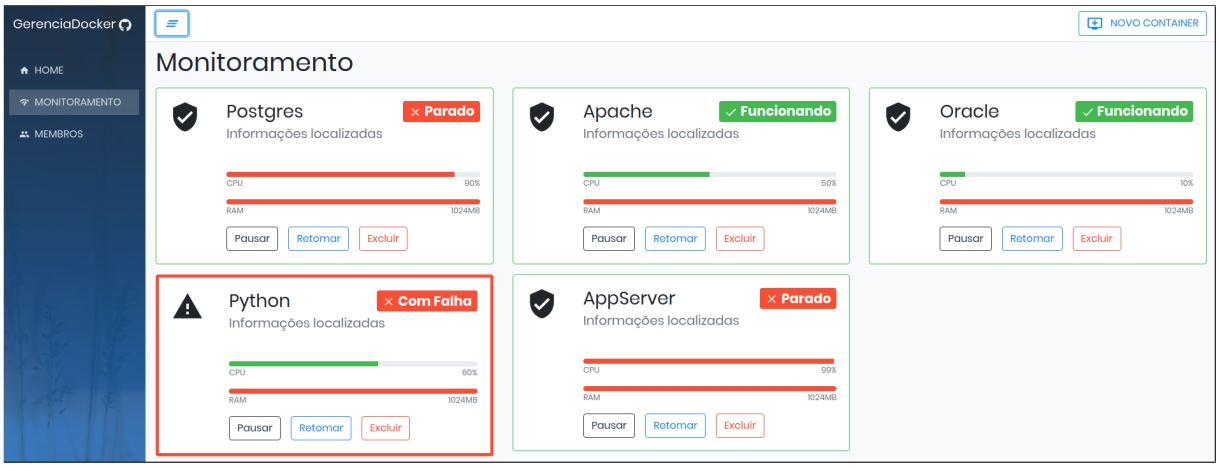
A biblioteca *React* realmente atendeu às expectativas e permitiu que as telas do sistema fossem criadas realmente conforme o pretendido. A interface, contudo, possui duas telas principais: a *home page*, que pode ser visualizada na figura **??**, além de ser uma tela de boas-vindas ao usuário, serve para exibir alguns dados importantes sobre as últimas utilizações do sistema e explanar, brevemente, quais são as vantagens que a plataforma pode proporcionar.

Figura 2 – *Home Page*



Já a segunda, que é a tela de monitoramento, é o local mais importante do sistema e pode ser acessado através do menu lateral esquerdo. É nesse ambiente que o usuário criará sua *docker network* e visualizará o estado atual de cada um dos *containers* que a compõe. Além disso, também é possível interagir com cada *contêiner* individualmente e utilizar opções como "Pausar", "Retomar" ou "Excluir", tal como pode ser visualizado na figura **??**.

Figura 3 – *Monitoramento*



A principal funcionalidade utilizada na tela de monitoramento são os *Hooks*, que permitem criar estados para armazenamento de dados e variáveis de forma rápida em uma função. Isto acelera o desenvolvimento, pois retira a necessidade da criação de classes para o funcionamento do *front*.

Os *Hooks* auxiliam o desenvolvedor a alcançar o principal objetivo do *React*, que é fazer uma atualização dinâmica e prática dos componentes presentes na página, reduzindo o consumo de recursos e complexidade, pois abstraem conceitos e métodos para facilitar a construção da página.

5.4 PWA

Os *Progressive Web Apps*, que recentemente vêm ganhando bastante popularidade e adeptos, podem ser definidos como (**??**) "uma aplicação *web* com tecnologias que permitem termos a experiência de uso muito próxima da oferecida pelos *mobile apps*". Isso quer dizer, basicamente, que ao acessar um *site*, o usuário terá a opção de adicionar um "atalho" para este no seu celular.

Porém, ao abrir esse atalho, a experiência para o usuário será de estar utilizando um aplicativo nativo, como se tivesse "instalado o *site*" em seu dispositivo. Isso porque ele não verá uma barra exibindo a *URL* que está sendo acessada e, além disso, funções nativas do celular - como acesso à câmera, geolocalização, aos contatos e *push notifications* - estarão em pleno funcionamento. Esse tipo de solução tem sido testada e utilizada por grandes empresas como Uber, Twitter e Facebook, por exemplo, em virtude de algumas vantagens que ela apresenta sobre os *apps* nativos, principalmente em dois pontos: custos e engajamento do cliente.

Primeiramente, no quesito despesas, utilizar um *PWA* permite que, com poucas alterações no site da empresa, ela disponibilize um "aplicativo" - o que economiza tempo de desenvolvimento e, consequentemente, dinheiro. Em segundo lugar, geralmente os *PWAs* são mais rápidos, ocupam menos espaço de armazenamento no dispositivo e, em casos reais, ficou comprovado que a facilidade em obter esse *app* gera mais conversão de clientes, conforme (**??**) "o Flipkart que é o maior e-commerce da Índia, eles decidiram fazer uma experiência mobile através de uma PWA e aumentaram a sua conversão em 70%".

Para resumir, sempre que deseja-se criar uma experiência agradável para o usuário, de forma rápida e com poucos custos, não sendo necessário implementar funcionalidades demasiadamente robustas, um *PWA* é a melhor opção. No caso desse projeto, portanto, que se enquadra perfeitamente nessas características, o grupo considerou essa como sendo a tecnologia ideal para criarmos um aplicativo.

5.4.1 Desenvolvimento

Após entender quais eram as tecnologias necessárias, realizar pesquisas sobre o assunto e realizar alguns cursos *on-line*, pode-se dizer que o desenvolvimento do aplicativo não foi tão complexo - até porque sua funcionalidade é bastante simples. No entanto, como em qualquer outra aplicação, a curva de aprendizado inicial foi o obstáculo mais relevante que teve de ser superado. Após passar por esta etapa, trabalhou-se com foco na parte visual (procurando deixá-la agradável e responsiva) e nos testes do *app*, pois um dos seus pontos mais atrativos é

o envio de notificações e, portanto, era obrigatório que isso que estivesse funcionando corretamente.

Ao fim da sua construção, pode-se dizer que esse aplicativo utilizou três recursos essenciais, sem os quais seu funcionamento seria inviável. Para que seja possível citar alguns detalhes técnicos, as seções a seguir serão destinadas especificamente a cada um deles: Firebase, GitHub e JavaScript.

5.4.1.1 Firebase

A plataforma, desenvolvida pela Google com o objetivo de tornar o desenvolvimento de aplicativos e sistemas mais fácil e rápida, é quem possibilita o envio das *push notifications* pois, na realidade, é ela quem gerencia o envio e recebimento desses avisos. Isso porque, toda vez que a *API* obter a informação de uma falha no container, ela enviará uma requisição *POST* para um servidor do Firebase e, somente então, o Firebase irá reconhecer quem deve receber essa mensagem e a entregará ao seu destinatário final.

Além desse grande auxílio, a plataforma possui um console completamente focado em escalar exponencialmente qualquer tipo de solução que a utilize, tal como permitir a utilização de um banco de dados de tempo real em poucos *clicks*. Essa grande variadade de ferramentas disponibilizadas pela mesma plataforma é um fato bastante positivo pois, caso o grupo decida evoluir ainda mais este projeto, diversos recursos atuais e de grande valia estariam disponíveis para serem utilizados facilmente.

5.4.1.2 GitHub

A plataforma de armazenamento de código teve dois papeis fundamentais para essa parte do projeto: o primeiro - e mais óbvio - é que o código foi versionado através da mesma, assim como os demais módulos da solução. No entanto, o segundo ponto é que a funcionalidade *GitHub Pages* permitiu disponibilizar esse aplicativo *on-line* e com um certificado HTTPS (requisito para o desenvolvimento de um *PWA*). Com isso, foi possível hospedar o aplicativo e realizar a demonstração através da própria plataforma, como é possível ver na figura **??**.

Figura 4 – *GitHub Pages*

5.4.1.3 JavaScript

É a linguagem de programação que foi utilizada para fazer todo o aplicativo. Além disso: é a responsável por criar os registros das falhas na tela dinamicamente, é nela que os *Service Workers* são escritos (*scripts* responsáveis por viabilizar o desenvolvimento das *PWAs*), também é através dela que a comunicação com o *firebase* é realizada e, por fim, é quem controla a inserção e leitura do *localStorage* (mecanismo do navegador que permite o armazenamento de informações).

O trecho de código abaixo apresenta o registro do *service worker*, que permite que o aplicativo seja utilizado mesmo sem conexão com a rede (*off-line*):

```
1    if ('serviceWorker' in navigator) {
2        navigator.serviceWorker
3            .register('service-worker.js')
4            .then(reg => console.log("[ServiceWorker] Registered..."))
5            [...]
```

5.4.2 O Aplicativo

Para tornar a solução ainda mais completa e próxima das necessidades do usuário, o grupo decidiu fazer um *app* simples (utilizando a tecnologia citada no subcapítulo **??**) que irá listar todas as ocorrências de falhas nos *containers*, além de disparar uma *push notification* no

dispositivo móvel do usuário. O objetivo principal deste aplicativo é servir como uma fonte de consulta para o usuário responsável pelo funcionamento da rede de *containers*.

Isso porque, em virtude de relacionar-se com diversas variáveis (velocidade da rede, infraestrutura, tamanho das equipes, entre outras), é possível que esse tipo de rede apresente pequenas falhas com o decorrer do tempo. Como é de conhecimento de todos, o profissional moderno precisa, além de realizar várias tarefas em paralelo, contar com uma certa mobilidade e, portanto, não seria agradável que um colaborador precisasse monitorar a tela do sistema em tempo integral para identificar quando ocorresse um problema.

Em razão disso, era necessário chamar a atenção dos responsáveis por manter a rede funcionando de alguma forma. Analisando as tecnologias disponíveis nos dias de hoje, chegou-se ao consenso que uma notificação no celular é, provavelmente, um dos melhores meios de chamar a atenção de alguém. Na figura **??**, é possível visualizar a interface da versão final do aplicativo:

Figura 5 – Aplicativo

5.5 Docker

O Docker é um pré-requisito para que a aplicação funcione, ou seja, ele deve estar presente no Sistema Operacional hospedeiro, pois é ele quem gerencia a criação dos containers e de suas respectivas networks. Apesar de ser apenas uma dependência do projeto, é necessário uma breve explicação de como ele é utilizado.

É através das syscalls vindas da api (mencionadas anteriormente) que o Docker é requisitado, basicamente, a api chama um comando específico dele e ele responde com uma saída

de sucesso ou falha. Vale ressaltar, a api funciona como uma interface web para o Docker, que por sua vez é uma interface para criação de LXC (Linux Containers).

5.5.1 Containers

A api possibilita a utilização de qualquer tipo de container, indiferente da sua distribuição Linux ou de qual processo está sendo executado dentro dele, porém, para que haja um monitoramento através do Adaptive-DSD, foi necessário a criação de alguns containers novos, todos eles tem o Adaptive-DSD rodando como processo principal. Esses containers estão divididos nas seguintes distribuições Linux: Ubuntu, Debian, Alpine e CentOs. Todos esses containers estão disponíveis no DockerHub (https://hub.docker.com/viniciusandd).

6 CRONOGRAMA

7 CONCLUSÃO

Conclusão aqui.