

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

1	DESCRIÇÃO DO PROBLEMA	1
1.1	Justificativa	1
2	OBJETIVOS	2
2.1	Objetivo Geral	2
2.2	Objetivos Específicos	2
3	METODOLOGIA	3
4	FERRAMENTAS UTILIZADAS	4
4.1	Áreas Envolvidas	4
5	DESCRIÇÃO DA IMPLEMENTAÇÃO	5
5.1	Adaptive DSD	5
5.1.1	Funcionamento	5
5.1.2	Algoritmo	6
6	CRONOGRAMA	12
7	CONCLUSÃO	13
	REFERÊNCIAS	14

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 *containerização*. De modo resumido, esse conceito pode ser descrito, segundo (FERNANDES, 2018), 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 se deseja 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, quando se está aprendendo a utilizar 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 é encarregado.

1.1 Justificativa

Baseando-se nos motivos descritos no capítulo 1 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*

3 METODOLOGIA

Sempre que almeja-se alcançar um objetivo ao final de uma tarefa, trabalho ou projeto, naturalmente o indivíduo 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, (STOODI, 2018) “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 (JUNIOR, 2017) “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-organizar-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 uma boa organização e um resultado que atendeu às expectativas iniciais.

4 FERRAMENTAS UTILIZADAS

4.1 Áreas Envolvidas

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. Onde 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 rede. É caracterizado como adaptativo por não depender e nem restringir o número de máquinas na rede, necessitando, no mínimo uma máquina para 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 de um processo na máquina.

5.1.1 Funcionamento

O algoritmo possui duas listas, que possuem de tamanho o número de máquinas conectadas à rede, as listas são: o vetor TESTED_UP, que irá guardar na posição da máquina atual, o índice da máquina testada que possui funcionamento normal; o vetor STATE, que armazena o estado das máquinas, tendo inicialmente o valor FALHO para todas e, caso uma máquina tenha seu funcionamento correto confirmado, esta receberá o valor NORMAL no vetor. A 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 à ser testada retornar uma resposta de funcionamento correto, a máquina que está realizando o teste atualiza os dados 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 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 os dados da rede, a máquina receptora irá prosseguir com a distribuição de informações.

5.1.2 Algoritmo

O algoritmo *Adaptive-DSD* inicia sua execução criando uma conexão em uma porta por 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 por *socket* com a porta.

```

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 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 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 distribuição 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  def ReceberRequisicao(conexao):
2      print("Aguardando mensagem...")
3      msg = ReceberResposta(conexao)
4

```

```

5  if msg == "start":
6      IniciarTeste(conexao)
7  elif msg == "check":
8      RealizarVerificacao(conexao)
9  elif msg == "keepTest":
10     ContinuarTeste(conexao)
11  elif msg == "keepInfo":
12     ManterInformacao(conexao)
13  elif msg == "info":
14     RetornaInformacao(conexao, False)

```

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

```

1  def IniciarTeste(conexao):
2      global maquinas
3      global tested_up
4      global state
5
6      EnviarResposta(conexao, "OK")
7      json_maquinas = ReceberResposta(conexao)
8      maquinas = json.loads(json_maquinas)
9
10     tested_up = ["-1"] * len(maquinas)
11     state = ["FALHO"] * len(maquinas)
12
13     print("maquinas: " + str(maquinas))
14     print("tested_up: " + str(tested_up))
15     print("state: " + str(state))
16
17     index = maquinas.index(ip_host+":"+str(porta_host))
18     state[index] = "NORMAL"
19     if index == len(maquinas)-1:
20         index = 0
21     TestarMaquina(index)

```

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 retornar a confirmação de funcionamento, a máquina atual procede em enviar as informações existentes à máquina testada, primeiramente enviando a mensagem 'keepTest' para informar a outra máquina a dar continuidade no teste. Caso a máquina testada retornar erro,

esta será marcada com um 'X' no vetor TESTED_UP e como 'FALHO' no vetor STATE. Ao chegar na última máquina da lista, o método **DistribuirInformacao** é executado.

```

1 def TestarMaquina(index):
2     global maquinas
3     global tested_up
4     global state
5     global ip_host
6     global porta_host
7
8     index_maquina = maquinas.index(ip_host+":"+str(porta_host))
9     while index < len(maquinas):
10         if index_maquina != index and tested_up[index] != "X":
11             host, porta = maquinas[index].split(":")
12             time.sleep(0.5)
13             maquina = CriarConexao(host, porta)
14
15             msg = EnviarInformacao(maquina, "check")
16             if msg == "OK":
17                 time.sleep(0.5)
18                 maquina = CriarConexao(host, porta)
19                 EnviarInformacao(maquina, "keepTest")
20
21                 tested_up[index_maquina] = str(index)
22                 state[index] = "NORMAL"
23
24                 json_maquinas = json.dumps(maquinas)
25                 json_tested = json.dumps(tested_up)
26                 json_state = json.dumps(state)
27
28                 EnviarInformacao(maquina, json_maquinas)
29                 EnviarInformacao(maquina, json_tested)
30                 EnviarInformacao(maquina, json_state)
31
32                 maquina.close()
33                 break
34             else:
35                 print("maquina com falha: " + str(index))
36                 tested_up[index] = "X"
37                 state[index] = "FALHO"
38                 index = index + 1
39         else:
40             index = index + 1
41
42     if index == len(maquinas):
43         print("Iniciar segundo ciclo...")

```

O método **ContinuarTeste** recebe as informações coletadas até o momento no teste, enviando uma confirmação a cada envio. Após isto, uma verificação é realizada para direcionar o teste para próxima máquina da lista, ou para iniciar a distribuição das informações, caso não haja mais máquinas não testadas na rede.

```

1 def ContinuarTeste (conexao):
2     global maquinas
3     global tested_up
4     global state
5
6     EnviarResposta (conexao , "OK")
7
8     json_maquinas = ReceberResposta (conexao)
9     maquinas = json.loads (json_maquinas)
10    EnviarResposta (conexao , "OK")
11
12    json_tested = ReceberResposta (conexao)
13    tested_up = json.loads (json_tested)
14    EnviarResposta (conexao , "OK")
15
16    json_state = ReceberResposta (conexao)
17    state = json.loads (json_state)
18    EnviarResposta (conexao , "OK")
19
20    index = 0
21    segundo_ciclo = True
22    while index < len (tested_up):
23        if tested_up [index] == "-1":
24            segundo_ciclo = False
25            break
26        index = index + 1
27
28    if segundo_ciclo:
29        print ("Iniciar segundo ciclo ...")
30        DistribuirInformacao ()
31    else:
32        index_maquina = maquinas.index (ip_host+":"+str (porta_host))
33        if index_maquina == len (maquinas) -1:
34            index_maquina = 0
35        TestarMaquina (index_maquina)

```

O método **ManterInformacao** recebe as informações do teste finalizado, enviando uma confirmação a cada envio. Por fim, a máquina envia estas informações à próxima máquina com status 'NORMAL'.

```

1 def ManterInformacao(conexao):
2     global tested_up
3     global state
4     global porta_host
5     global ip_host
6     EnviarResposta(conexao, "OK")
7
8     json_tested = ReceberResposta(conexao)
9     tested_up = json.loads(json_tested)
10    EnviarResposta(conexao, "OK")
11
12    json_state = ReceberResposta(conexao)
13    state = json.loads(json_state)
14    EnviarResposta(conexao, "OK")
15
16    index_host = maquinas.index(ip_host+": "+str(porta_host))
17    indexMaquina = int(tested_up[index_host])
18    if index_host < len(maquinas)-1 and indexMaquina > index_host:
19        time.sleep(0.5)
20        host, porta = maquinas[int(indexMaquina)].split(":")
21        maquina = CriarConexao(host, porta)
22
23        EnviarInformacao(maquina, "keepInfo")
24        RetornaInformacao(maquina, True)
25        maquina.close()

```

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, se não, irá apenas enviar as informações, sem esperar por uma resposta de confirmação.

```

1 def RetornaInformacao(conexao, verificacao):
2     global tested_up
3     global state
4

```

```
5     json_tested = json.dumps(tested_up)
6     json_state = json.dumps(state)
7
8     if verificacao:
9         EnviarInformacao(conexao, json_tested)
10        EnviarInformacao(conexao, json_state)
11    else:
12        EnviarResposta(conexao, json_tested)
13        EnviarResposta(conexao, json_state)
14    conexao.close()
```

6 CRONOGRAMA

7 CONCLUSÃO

Conclusão aqui.

REFERÊNCIAS

FERNANDES, A. **O que é Containerização de aplicação?** 2018. Disponível em: <<https://vertigo.com.br/o-que-e-containerizacao-de-aplicacao/>>. Acesso em: 28 de setembro de 2019. Citado na página 1.

JUNIOR, C. **Scrum: o que é sprint e como executá-lo?** 2017. Disponível em: <<https://www.projectbuilder.com.br/blog/scrum-o-que-e-sprint-e-como-executa-lo/>>. Acesso em: 14 de novembro de 2019. Citado na página 3.

STOODI. **O QUE É METODOLOGIA?** 2018. Disponível em: <<https://www.stoodi.com.br/guias/dicas/o-que-e-metodologia/>>. Acesso em: 14 de novembro de 2019. Citado na página 3.