



**Universidade de Brasília - UnB
Faculdade UnB Gama - FGA
Curso de Engenharia de Software**

**Lab03_distribuído - Construindo aplicações distribuídas
usando o paradigma publish-subscriber**

Fundamentos de Redes de Computadores - Turma A

Professor: Fernando William Cruz

**Autores: Bruno Carmo Nunes - 18/0117548
Marcos Vinícius R. da Conceição - 17/0150747**

**Brasília, DF
2021**



1. INTRODUÇÃO

O RabbitMQ é um broker leve e fácil de implementar nas premissas e na nuvem. Ele suporta múltiplos protocolos de mensagens. Ele pode ser também implementado em configurações distribuídas e federadas para encontrarem um alta escalabilidade e alta disponibilidade dos requisitos.

De uma forma resumida, o funcionamento do AMQP (Advanced Message Queuing Protocol) pode ser comparado a um sistema de correios conforme a figura 1: o cliente Publisher escrever uma mensagem e a encaminha para um exchange, que se assemelha a uma caixa de correio. A seguir, já no broker - que pode ser entendido como os Correios, por exemplo - esta mensagem é encaminhada de acordo com regras específicas (bindings) através de rotas a uma queue, que pode ser entendida como uma caixa postal, por exemplo. Finalmente, um cliente Consumer monitora a queue (caixa postal) para, então, ler a mensagem. [1]

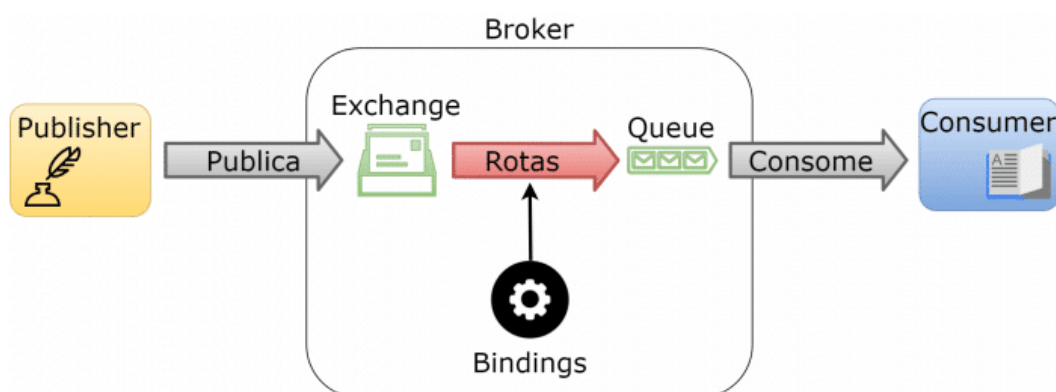


Figura 1- Diagrama do Funcionamento do Exchange tipo Direto no AMQP.

2. DESENVOLVIMENTO

2.1. Descrição da solução para 100 posições

Para solução inicial, pegamos o tutorial do Hello World! presente na lista de aprendizado na documentação do RabbitMQ feito em Python, tendo uma breve descrição do seu funcionamento abaixo:

I. Hello World!

Nesse exemplo, é somente implementado dois pequenos programas em Python; um produtor manda uma mensagem única e um consumidor recebe mensagens e exibe elas. No diagrama abaixo P é o produtor e C o nosso consumidor. A caixa vermelha no meio é o queue - buffer de mensagem que o RabbitMQ guarda pelo consumidor. O produtor manda mensagens pelo queue “hello”, e o consumidor recebe mensagens dele como mostrado na figura abaixo:

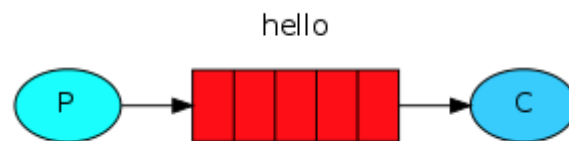


Figura 2 - Exemplo Hello World!

Sendo assim, conseguimos trabalhar nessa ideia de mandar as informações do tamanho do vetor para a fila na queue “hello”, tendo então um vetor de 100 posições. Ao receber, essa informação, o servidor então passa pelas informações do vetor e procura o valor máximo e mínimo dele. Depois disso, fizemos outra queue chamada “world” para retornar o valor máximo e mínimo para o cliente, e exibi-los no console. Conforme a figura abaixo:

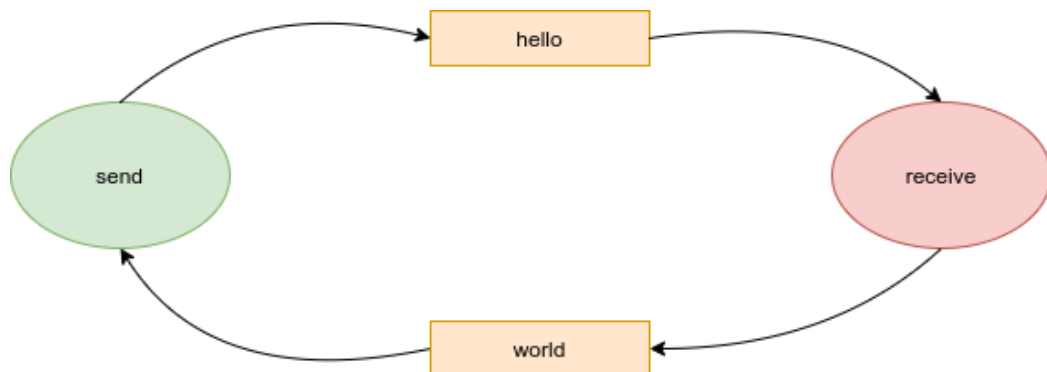
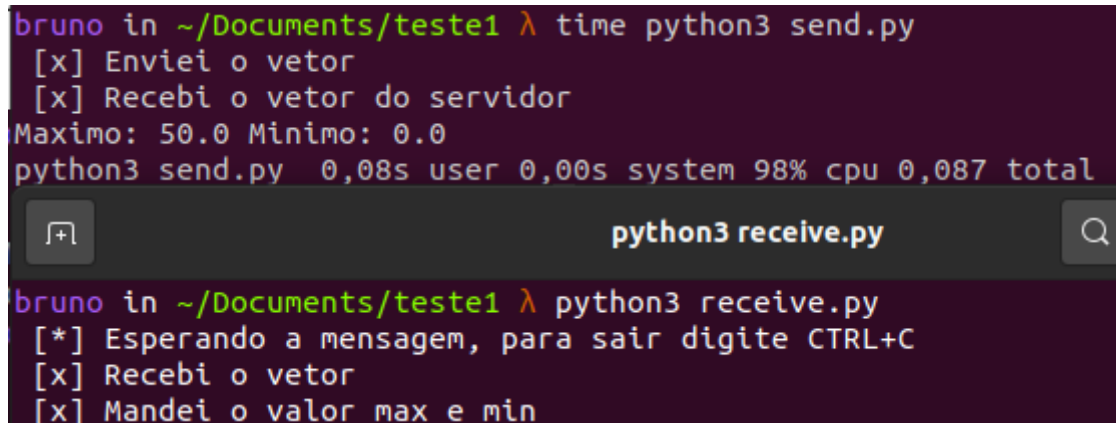


Figura 3 - Diagrama do problema

Executamos então o comando `“python3 receive.py”` para abrir o servidor(receive) esperando então o envio do vetor e posteriormente o comando `“python3 send.py”` para enviar o vetor pelo cliente(send). Também executamos o comando `“time”`

juntamente com o send para ver a duração da existência do processo. Analisamos na figura 4 que durou cerca de 0.087 segundos para executar os procedimentos.



```
bruno in ~/Documents/teste1 λ time python3 send.py
[x] Enviei o vetor
[x] Recebi o vetor do servidor
Maximo: 50.0 Minimo: 0.0
python3 send.py 0,08s user 0,00s system 98% cpu 0,087 total

python3 receive.py
[*] Esperando a mensagem, para sair digite CTRL+C
[x] Recebi o vetor
[x] Mandei o valor max e min
```

Figura 4 - Execução do cliente e servidor

2.2. Descrição da solução para um vetor de 1.000.000.000

Para a solução de um vetor de 1 bilhão tivemos problemas inicialmente para processar o tamanho da informação. Começamos então com 1 milhão de posições para tentar resolver o problema pois na primeira solução (figura 3), o computador não tem memória e capacidade de processamento para resolver um vetor tão grande. Com isso, o primeiro problema foi o transporte gigantesco de um vetor pela queue “hello”, onde por padrão cada queue pode transportar até 1MB de informação. Tivemos então duas opções, ou mandar menos informações ou aumentar o tamanho de armazenamento das queues. Sendo assim, por otimização do código resolvemos optar pela primeira opção. Dividimos então a informação do vetor por pacotes, tendo então a possibilidade de transformar por exemplo 4 MB de informação em 100 pacotes de 40 KB.

Depois de resolvido esse problema, encontramos outra complicação que foi o tamanho da memória em relação às posições. Onde quando calcula os 1 bilhões de posições o tamanho da memória do computador não foi o suficiente para armazenar essas informações pedindo até mais de 16GB de memória ram. Sendo assim, uma das maneiras de reduzir esse valor, foi reutilizando as posições já armazenadas, invés de armazenar todas as informações de uma só vez. Quando um pacote é

enviado, o cliente reaproveita a mesma posição utilizada na memória, por exemplo, um vetor de 100 posições invés de guardar todas as posições só podemos passar por 10 posições e enviá-las para a queue, e assim sucessivamente até enviarmos as 100 posições.

Porém, ao resolver esse problema encontramos um outro problema que foi o tempo para resolver o cálculo com mais de 10 milhões de posições. Pois quando é multiplicado por 10 (10 milhões para 100 milhões) o tempo também aumenta 10 vezes. Ao fator de 1 bilhão de posições chegamos a calcular quase 2000 segundos para executar todo o processo. Assim, uma das conclusões que tivemos foi distribuir parte do vetor calculado em tasks diferentes, usando então o tutorial Work queues, descrito abaixo.

II. Work queues

O primeiro tutorial é programas que enviam e recebem mensagens de uma queue nomeada. Nesse é criada uma Work Queue que vai ser usada para distribuir tarefas de tempo consumidos sobre múltiplos workers. A ideia por trás do Work Queues é evitar fazer tarefas de recursos intensivos imediatamente e ter que esperar ela para completar. Invés de agendar uma tarefa para ser feita posteriormente. Tem-se o encapsulamento de uma tarefa como mensagem e é enviada para uma queue. O processo do worker rodando por trás vai disparar as tarefas e eventualmente executar o processo. Quando você executa muitos trabalhadores as tarefas serão compartilhadas entre eles.

O conceito é especialmente útil para aplicações web onde é impossível se manusear uma tarefa complexa durante uma pequena abertura de pedido HTTP.

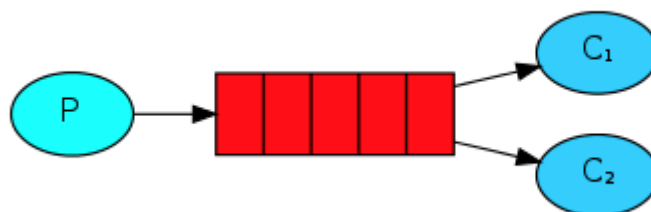


Figura 5 - Exemplo Work Queues

Depois de entender como funcionam os work queues, distribuimos as operações do cálculo e fixação dos vetores em processos iguais que executam partes dos vetores em posições diferentes. Por exemplo: Na figura 6, o send manda a posição inicial, a posição final, tamanho do vetor e quantidade de pacotes na fila task_queue. Sendo assim, um vetor de 1000 posições com 5 processos de tasks, temos então que cada task vai calcular 200 posições no vetor e envia o valor do vetor para o hello, onde o processo receive vai calcular o máximo e o mínimo dos vetores fornecidos. Depois de esperar todos os 5 processos acabar a operação, ele manda o resultado final para o processo inicial send, exibindo na tela os valores máximos e mínimos de todas as posições calculadas. Temos então uma maneira mais otimizada do algoritmo fornecer esses cálculos, onde calculamos o tempo sendo de 300 segundos para 1 bilhão de posições.

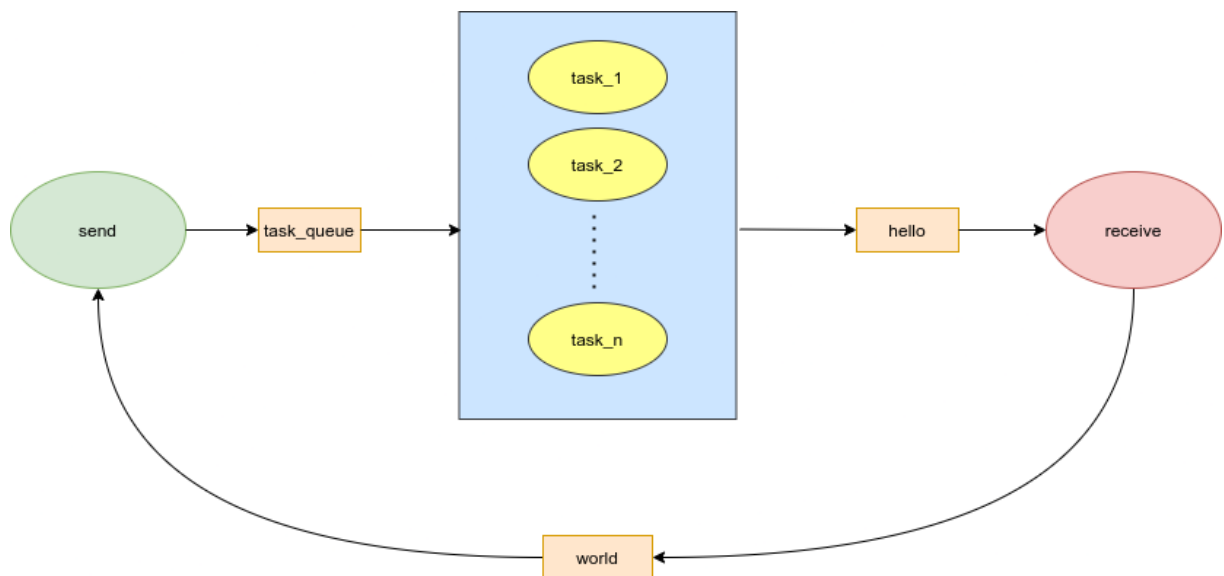
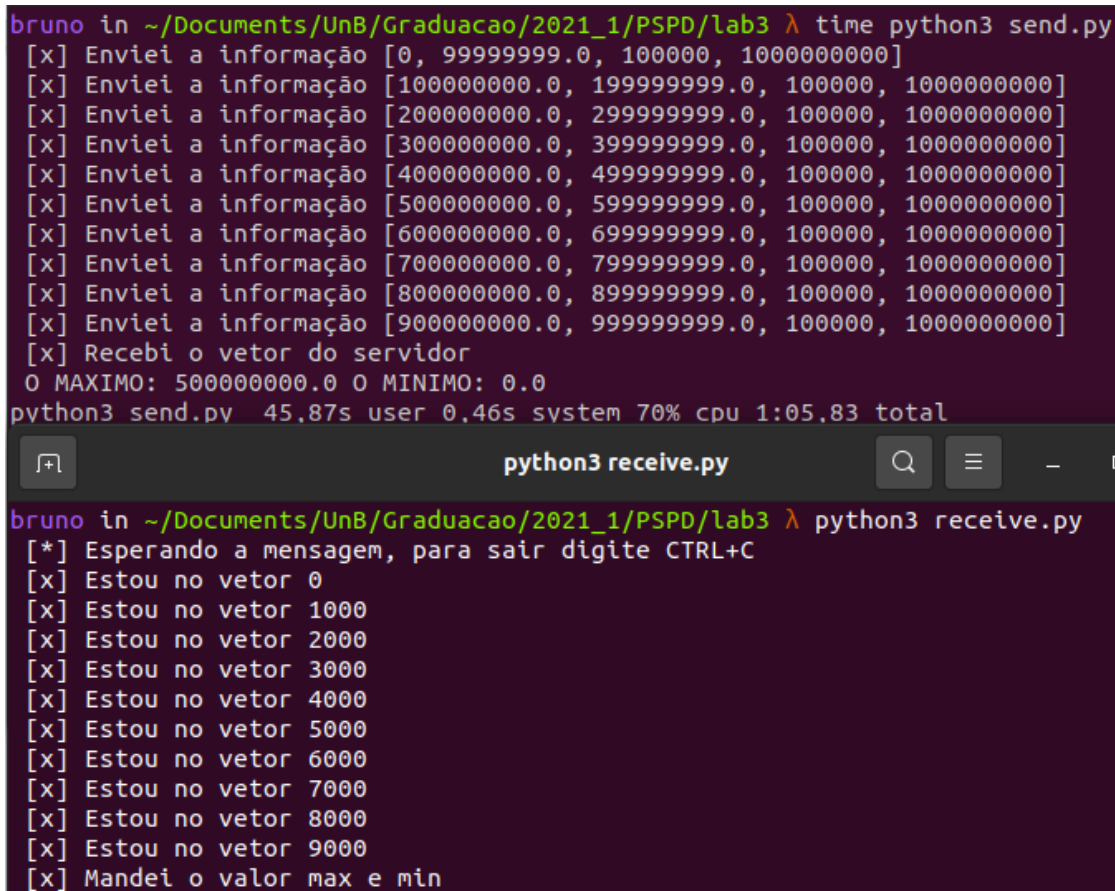


Figura 6 - Diagrama para solução de 1 bilhão de posições

Mas percebemos que um só processo (receive) para calcular todos os valores na fila “hello” demorava muito mais que os resultados das tasks, sendo que se todas as tasks resolviam em 2 minutos e o processo receive demorava o dobro para ler e descobrir o maior e menor de todos os vetores enviados a ele. Sendo assim, para otimizar mais uma vez essa solução, ordenamos o vetor calculado e realizamos o

cálculo de máximo e mínimo nas tasks que só mandavam dois valores para o receive (max, min). Por fim, tivemos um resultado da última versão na figura 7, onde mostra a duração de somente 65 segundos para a execução do cálculo para um vetor de 1 bilhão de posições.



```
bruno in ~/Documents/UnB/Graduacao/2021_1/PSPD/lab3 λ time python3 send.py
[x] Enviei a informação [0, 99999999.0, 100000, 1000000000]
[x] Enviei a informação [1000000000.0, 199999999.0, 100000, 1000000000]
[x] Enviei a informação [2000000000.0, 299999999.0, 100000, 1000000000]
[x] Enviei a informação [3000000000.0, 399999999.0, 100000, 1000000000]
[x] Enviei a informação [4000000000.0, 499999999.0, 100000, 1000000000]
[x] Enviei a informação [5000000000.0, 599999999.0, 100000, 1000000000]
[x] Enviei a informação [6000000000.0, 699999999.0, 100000, 1000000000]
[x] Enviei a informação [7000000000.0, 799999999.0, 100000, 1000000000]
[x] Enviei a informação [8000000000.0, 899999999.0, 100000, 1000000000]
[x] Enviei a informação [9000000000.0, 999999999.0, 100000, 1000000000]
[x] Recebi o vetor do servidor
O MAXIMO: 5000000000.0 O MINIMO: 0.0
python3 send.py 45.87s user 0.46s system 70% cpu 1:05.83 total
```

```
python3 receive.py
bruno in ~/Documents/UnB/Graduacao/2021_1/PSPD/lab3 λ python3 receive.py
[*] Esperando a mensagem, para sair digite CTRL+C
[x] Estou no vetor 0
[x] Estou no vetor 1000
[x] Estou no vetor 2000
[x] Estou no vetor 3000
[x] Estou no vetor 4000
[x] Estou no vetor 5000
[x] Estou no vetor 6000
[x] Estou no vetor 7000
[x] Estou no vetor 8000
[x] Estou no vetor 9000
[x] Mandei o valor max e min
```

Figura 7 - Medição do tempo em send.py e execução do servidor (máquina Bruno)

Fizemos comparações em máquinas separadas, onde na figura 4 e figura 7 foram realizadas numa FX-8350 de 4GHz, e nas figuras posteriores foram feitos num I7-8550U de 1.8Hz com turbo boost de 4GHz. Abaixo está mais detalhado os procedimentos tomados por cada arquivos.

```

marcos@marcos-HP-Laptop-15-bs1xx:~/Desktop/unb/UNB-1-202
1/Sistemas_Paralelos_e_Distribuidos/lab03$ time python3
send.py
[x] Enviei a informação [0, 99999999.0, 100000, 1000000
000]
[x] Enviei a informação [1000000000.0, 199999999.0, 1000
00, 10000000000]
[x] Enviei a informação [2000000000.0, 299999999.0, 1000
00, 10000000000]
[x] Enviei a informação [3000000000.0, 399999999.0, 1000
00, 10000000000]
[x] Enviei a informação [4000000000.0, 499999999.0, 1000
00, 10000000000]
[x] Enviei a informação [5000000000.0, 599999999.0, 1000
00, 10000000000]
[x] Enviei a informação [6000000000.0, 699999999.0, 1000
00, 10000000000]
[x] Enviei a informação [7000000000.0, 799999999.0, 1000
00, 10000000000]
[x] Enviei a informação [8000000000.0, 899999999.0, 1000
00, 10000000000]
[x] Enviei a informação [9000000000.0, 999999999.0, 1000
00, 10000000000]
[x] Recebi o vetor do servidor
0 MAXIMO: 5000000000.0 0 MINIMO: 0.0

real    2m24,124s
user    1m29,892s
sys      0m0,500s

```

Figura 8 - Terminal com dados send.py

A figura 8 mostra os dados que são enviados do send.py para os daemons (task.py) cada daemon recebe um pacote contendo a info um vetor de 4 posições onde:

info = [tamanhoInicial, tamanhoFinal, qtdNumVetor, tamanho]

- info [0] indica o início do vetor segmentado;
- info [1] indica o fim do vetor segmentado;
- info [2] indica a quantidade de elementos de cada vetor segmentado;
- info [3] indica o tamanho total do vetor completo;

Ao terminar o envio ele começa a esperar pela resposta que vai ser enviada pelo worker do receive.py em um vetorVolta de duas posições onde a primeira posição indica o menor valor global e a segunda posição o maior valor global do vetor de 1 bilhão de posições.


```
marcos@marcos-HP-Laptop-15-bs1xx:~/Desktop/unb/UNB-1-202
1/Sistemas_Paralelos_e_Distribuidos/lab03$ time python3
receive.py
[*] Esperando a mensagem, para sair digite CTRL+C
[x] Estou no vetor 0
[x] Estou no vetor 1000
[x] Estou no vetor 2000
[x] Estou no vetor 3000
[x] Estou no vetor 4000
[x] Estou no vetor 5000
[x] Estou no vetor 6000
[x] Estou no vetor 7000
[x] Estou no vetor 8000
[x] Estou no vetor 9000
[x] Mandei o valor max e min
^CInterrompido

real    4m38,686s
user    0m4,069s
sys     0m0,318s
```

Figura 9 - Terminal com dados do receive.py

Na figura 9 é mostrado em que posição a execução dos vetores segmentados está. Para um vetor de 1 bilhão de posições e 10000 sub-vetores de 100000 elementos.

Process Name	User	% CPU	ID	Memory	Disk read tot	Disk write tot	Disk read	Disk write	Priority
code	marcos	0,00	5932	229,9 MB	42,4 MB	221,2 kB	N/A	N/A	Normal
python3	marcos	8,16	40528	9,7 MB	N/A	N/A	N/A	N/A	Normal
python3	marcos	8,24	40529	9,7 MB	N/A	N/A	N/A	N/A	Normal
python3	marcos	7,78	40530	9,7 MB	N/A	N/A	N/A	N/A	Normal
python3	marcos	6,70	40532	9,7 MB	N/A	N/A	N/A	N/A	Normal
python3	marcos	6,20	40537	9,7 MB	N/A	N/A	N/A	N/A	Normal
python3	marcos	6,95	40531	9,7 MB	N/A	N/A	N/A	N/A	Normal
python3	marcos	7,12	40535	9,7 MB	N/A	N/A	N/A	N/A	Normal
python3	marcos	10,36	40536	9,7 MB	N/A	N/A	N/A	N/A	Normal
python3	marcos	10,24	40533	9,7 MB	N/A	N/A	N/A	N/A	Normal
python3	marcos	8,49	40534	9,7 MB	N/A	N/A	N/A	N/A	Normal
python3	marcos	0,00	40526	9,5 MB	N/A	N/A	N/A	N/A	Normal
python3	marcos	2,54	40462	9,5 MB	N/A	N/A	N/A	N/A	Normal

Figura 10 - Processos abertos ao executar os programas

Na figura 10 é possível verificar os dois programas (send.py e receive.py) juntamente com seus 10 workers que rodam como daemons pelo task.py. Um daemon é um programa Unix / Linux executado em segundo plano pronto para realizar uma operação quando necessário. Cada worker recebe um segmento do vetor total e executa a operação de preencher o vetor até o tamanho especificado, calcular o seu valor máximo e mínimo e adicionar em uma nova fila para o worker

final executar a verificação dos valores recebidos de todos os workers daemons para decidir qual o valor máximo e mínimo geral e enviá-lo para o send.

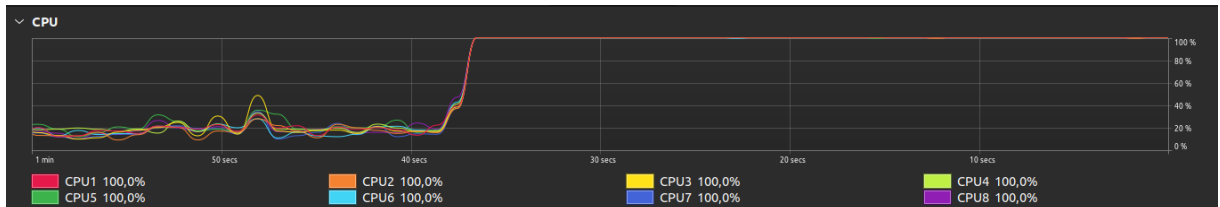


Figura 11 - Utilização da CPU ao inicializar envio dos 1 bilhão de dados

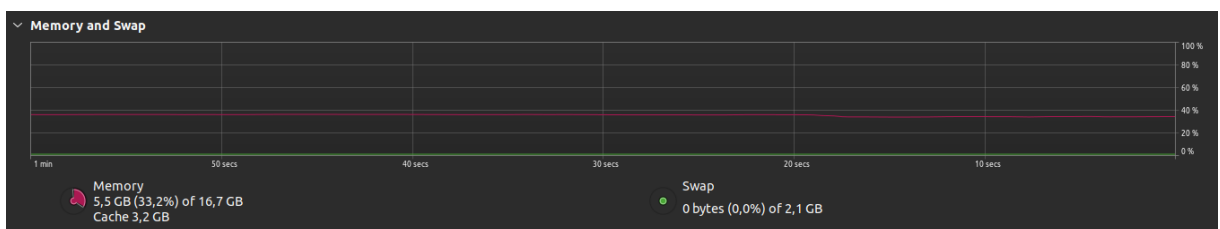


Figura 12 - Utilização de memória ao inicializar o programa

No algoritmo final conseguimos utilizar toda a capacidade de processamento da máquina como fica explícito na fig 0 para calcular da maneira mais rápida possível e ao mesmo tempo conseguimos resolver o problema da utilização da memória que se mantém estável durante toda a execução dos processos.

3. CONCLUSÃO

Com esse laboratório, além de conseguirmos encontrar uma solução sobre a utilização de um vetor pequeno por um broker, tivemos a oportunidade de distribuir várias tarefas para processos distintos, mostrando então um aumento na eficiência do algoritmo e da solução computacional. Teve então uma redução significativa do tempo de execução do projeto, onde inicialmente era de milhares de segundos para uma redução de no máximo 180 segundos e o mínimo de 65 segundos (depende do desempenho da máquina utilizada).

A solução computacional, mostrou também como utilizar otimizar o tamanho da informação nas filas(queues) do RabbitMQ proporcionando um aprendizado

aprimorado para os alunos em relação à AMQP. As maiores dificuldades do trabalho foram de otimizar a questão de um vetor de 1 bilhão de posições, pois os alunos tiveram que utilizar outros tutoriais além do inicial, e também resolver problemas em relação à memória e melhoria na performance do algoritmo. Uma das limitações do código é que não pode ser utilizado tamanhos ímpares de posições no vetor, pois podem acontecer erros inesperados.

4. REFERÊNCIAS

[1] RabbitMQ - <https://www.rabbitmq.com/getstarted.html> , acessado dia 2 de setembro de 2021.

[2] [What is remote desktop connection broker?](#) , acessado dia 2 de setembro de 2021.

[3] [Pika python wait for RabbitMQ Publisher if queue reach limit](#), acessado dia 4 de setembro de 2021.