Performance by implementação

Resumo

*Em algumas das execuções foi necessário ajustar a configuração de size-pool.

	Este método estourou o limite de RAM		Uso de RAM mediano, mas não controla o fluxo de escrita.		Uso de RAM mediano, mas tem bom controle de escrita.		Uso de RAM alto, mas para arquivos <4MB sem vantagem.		Este método estourou o limite de RAM		Baixo uso de RAM, mas requer controle na escrita dos arquivos.	
ROUND	**No streaming Buffer all data In Memory**		**Streaming (On Demand)**		**Streaming + Buffer In Memory (4KB)**		**Streaming + Buffer In Memory (4MB)**		**Streaming + Buffer In Memory (8MB)**		**No streaming Save all data In Temp File**	
	TEMPO (SEC) 1 MB	**TEMPO (SEC) 10 MB**	**TEMPO (SEC) 1 MB**	**TEMPO (SEC) 10 MB**	**TEMPO (SEC) 1 MB**	**TEMPO (SEC) 10 MB**	**TEMPO (SEC) 1 MB**	**TEMPO (SEC) 10 MB**	**TEMPO (SEC) 1 MB**	**TEMPO (SEC) 10 MB**	**TEMPO (SEC) 1 MB**	**TEMPO (SEC) 10 MB**
AVG	1.16	8.82	1.24	8.89	1.14	9.03	1.29	9.08	1.34	9.10	1.46	9.09
AVG 5MB	4.99		5.06		5.09		5.18		5.22		5.28	
MAX	2.73	19.42	3.45	21.61	2.55	23.45	3.87	24.50	4.01	24.31	15.31	22.01
MIN	0.37	3.52	0.27	3.07	0.36	2.59	0.38	2.86	0.37	2.53	0.35	2.51

Test Azure Upload java

Uso dos recursos da máquina

Uso limitado a 512m de memória (geralmente este é o tamanho dos containers)

*Talvez seja interessante avaliar o aumento para no mínimo 1GB, isso permitiria aumentar a quantidade de operações simultâneas. (Ou o tamanho dos arquivos)

***Para medir o uso dos recursos do servidor foi utilizado VisualVM.**

***A cada operação foi performado o Garbage Collector**

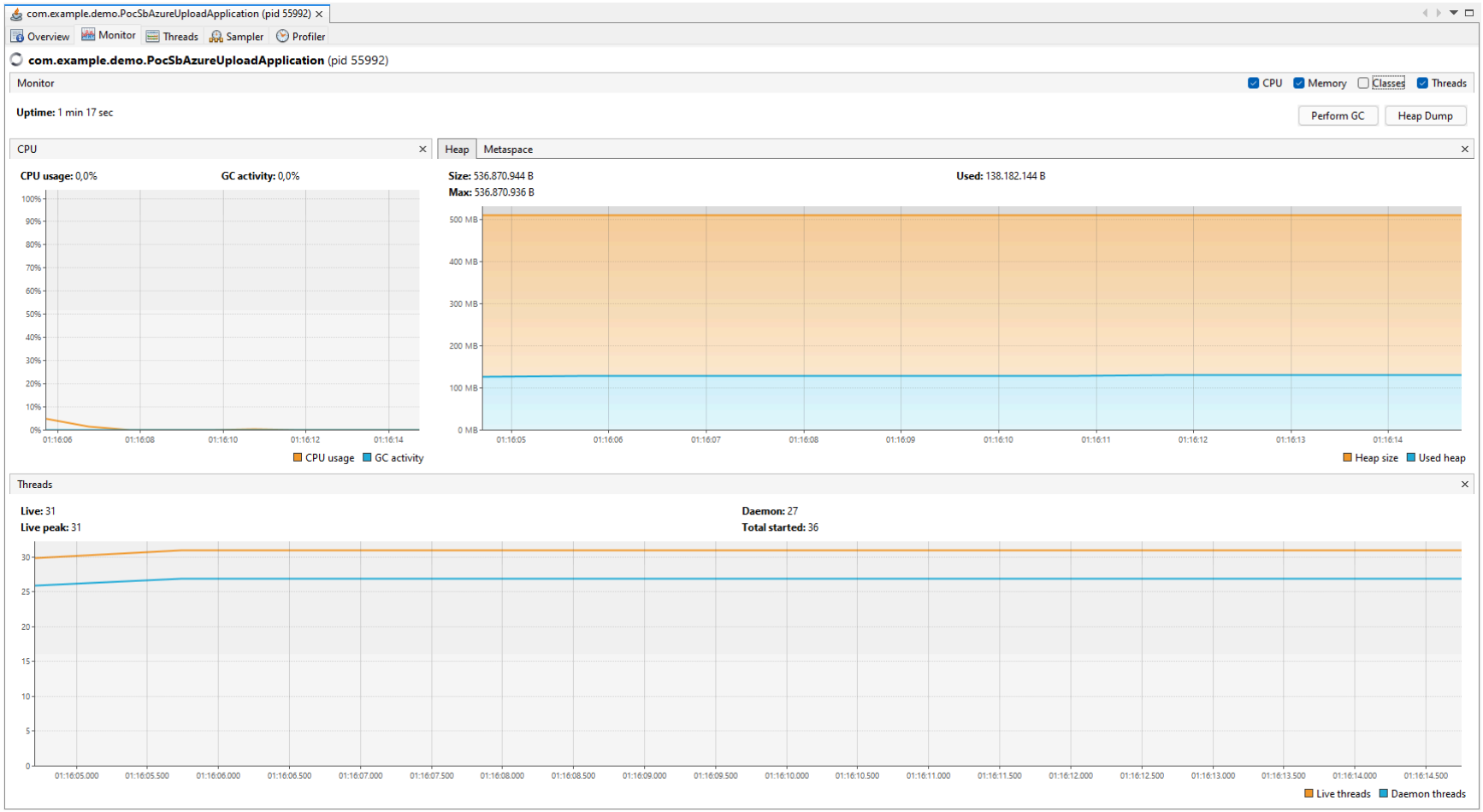
Para todas as operações foi utilizado a configuração de *AsyncExecutor*, portanto é necessário ajustar a quantidade de *Min/MaxWorkers* e *Min/MaxWorkerQueue* para que a aplicação não ultrapasse o limite de memória disponível para a JVM; Para todos os testes a seguir, as configurações de worker foram as seguintes:

```
@Configuration  no usages
public class ExecutorConfig {

    @Bean()  no usages
    public Executor asyncExecutor()
    {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(20);
        executor.setMaxPoolSize(25);
        executor.setQueueCapacity(100);
        executor.setThreadNamePrefix("AsynchThread-");
        executor.setAllowCoreThreadTimeOut(true);
        executor.setKeepAliveSeconds(20);
        executor.initialize();

        return executor;
    }
}
```

Conforme a configuração ao lado, para os métodos que implementam este executor, as operações estarão limitadas a no máximo 25 operações em concorrência (***setMaxPoolSize***), assim que ultrapassar o máximo de operações assíncronas serão enfileiradas para aguardar sua inicialização e destas no máximo 100 (***setQueueCapacity***) poderão ser enfileiradas. Levando isso em conta, se chegarem 200 operações em simultâneo 125 serão acatadas, as demais 75 irão "estourar" uma exceção do tipo ***TaskRejectedException***.

***Nota:** As operações acatadas (do exemplo acima 125) não vão parar ou deixar de executar caso a função principal receba uma exceção!

Application StartUP



Implementations

No Streaming File

Esta é uma implementação não confiável, nem mesmo nos testes foi possível obter sucesso em todas as execuções. Pois a quantidade de memória utilizada pela aplicação é determinada pelo tamanho dos arquivos de upload da aplicação.

Dada isso, não seguirei com exemplos devido ao descarte desta implementação.

Save in temp file

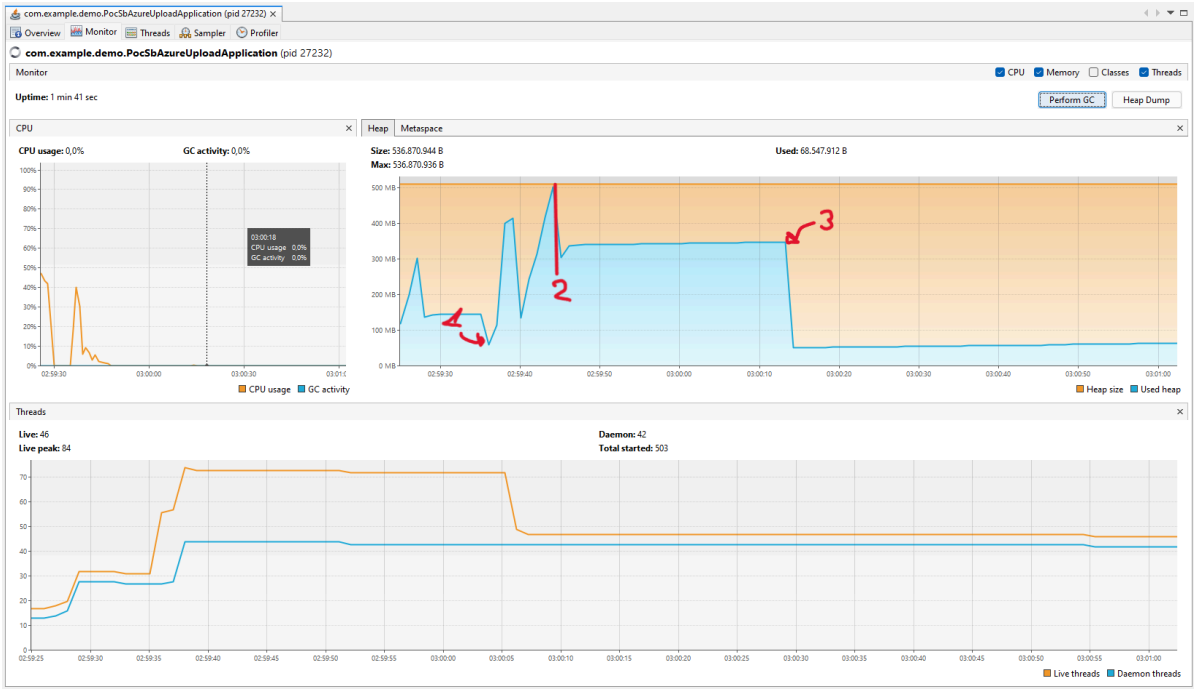
Esta é uma implementação que visava utilizar baixa memória, neste caso o consumo de memória é o disponível para I/O do sistema. Dessa forma conforme disponibilidade para escrita do arquivo o download é realizado.

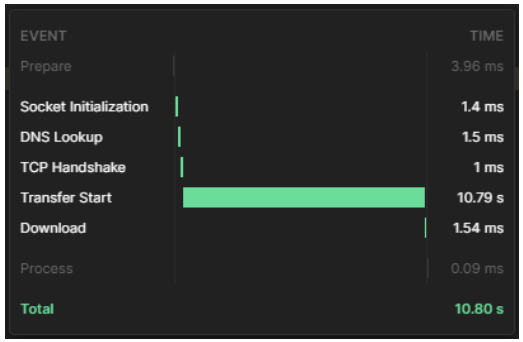
*Nota 1: Esta implementação não é tão recomendada pois precisa ter pleno controle dos arquivos temporários, em caso de rodar a aplicação em containers o problema é menor pois a cada restar os arquivos temporários são apagados, mas em implementação de servidores IaaS é possível que o arquivo fique salvo/perdido.

*Nota 2: Há risco também no container, mas no momento onde o POD fica sem memória ele é reiniciado. Não é uma solução mas a aplicação não para.

Arquivos de 1MB

Como é possível visualizar há um pico de memória, mas este não gera exceção devido ao sistema utilizar somente o disponível para escrita.



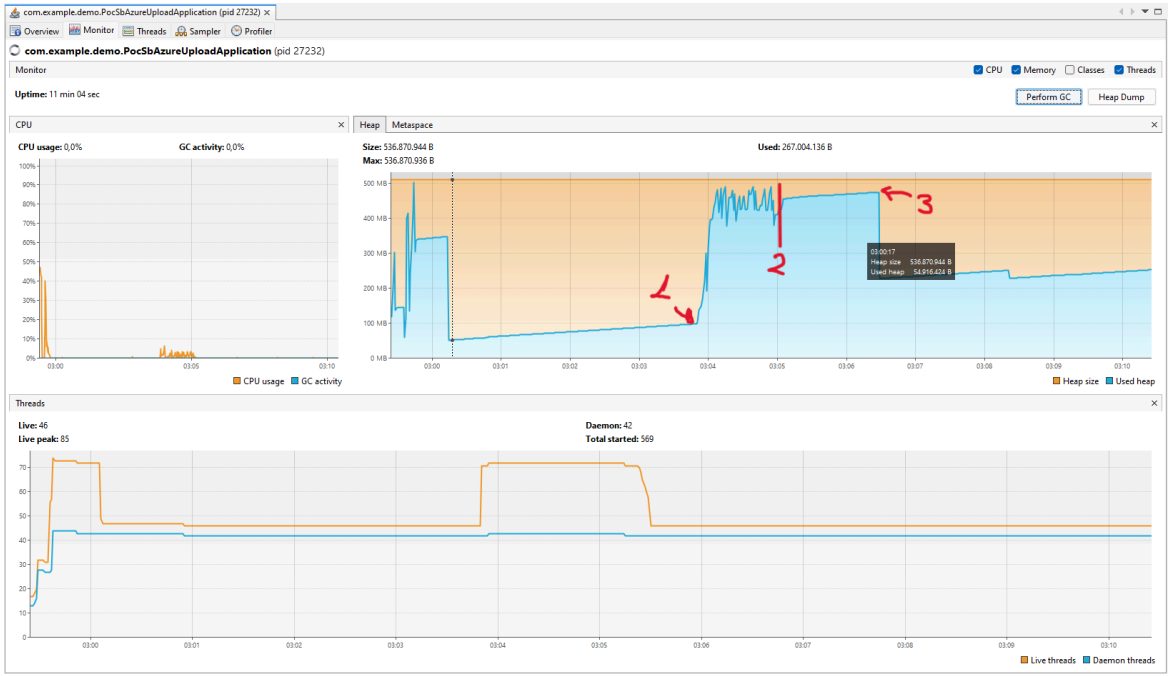


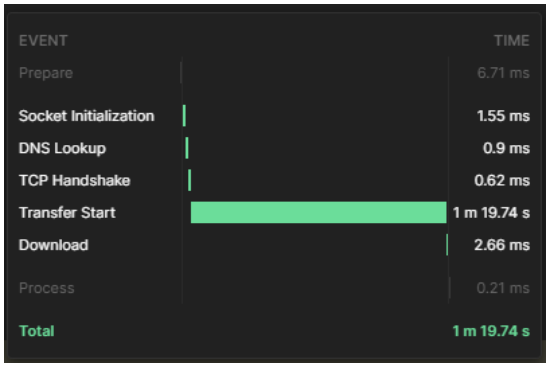
1. Início da operação
2. Aproximadamente o fim da operação
3. Execução do GC manualmente

Tempo: 10.6737162**s**
Média por arquivo: 2.030**s**
Pico de memória: 529.307**b**

Arquivos de 10MB

Conforme pode ser observado, para arquivos de 10MB o comportamento se repete.
Podemos concluir também que não surge uma grande vantagem, afinal é necessário realizar toda a escrita do arquivo localmente para assim iniciar o processo de upload.





1. Início da operação
2. Aproximadamente o fim da operação
3. Execução do GC manualmente

Tempo: 1**m** 19.722022**s**
Média por arquivo: 15.391**s**
Pico de memória: 516.771**b**

Upload Streaming File

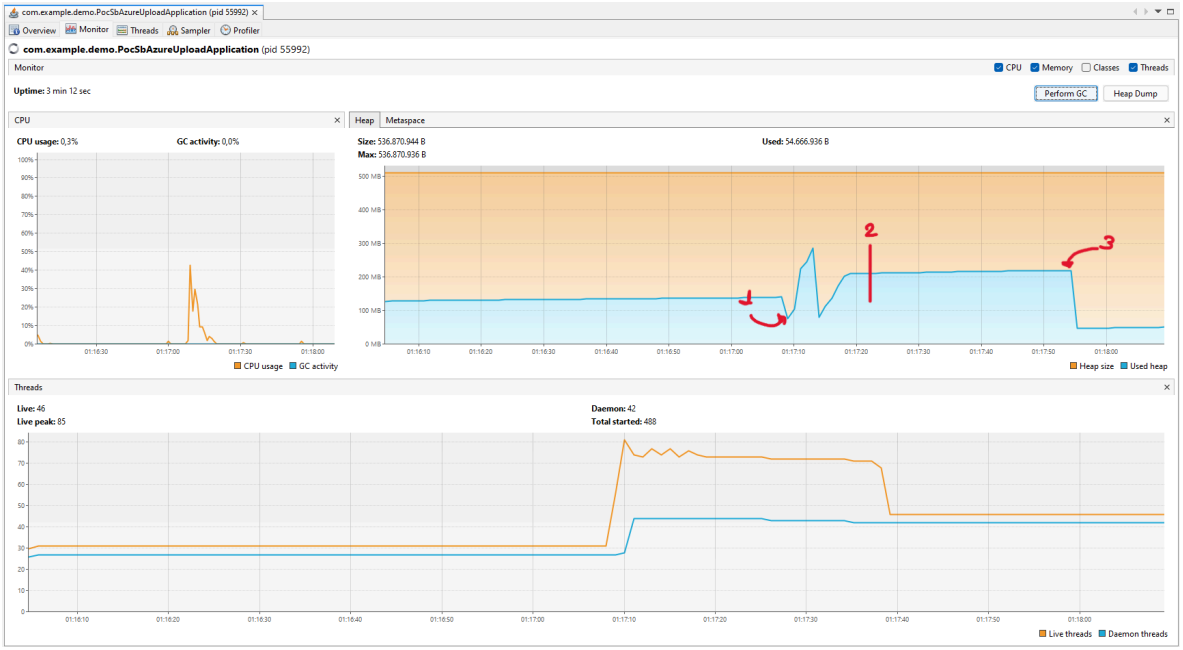
Para performar essas operações foi utilizada a implementação de streaming do arquivo, isto é, enquanto o download era realizado o arquivo era enviado para a Azure.

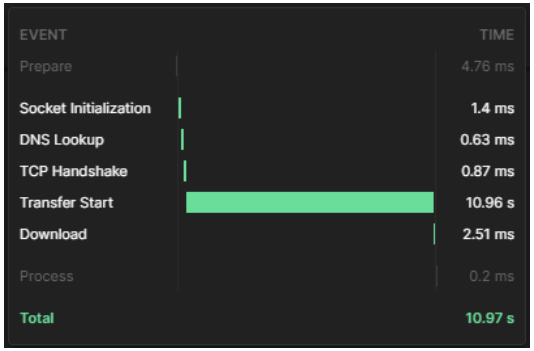
*Nota 1: Neste método o “arquivo” é salvo em memória (buffer), sendo assim a cada ciclo de leitura do inputstream (body da requisição) é “gasto” aproximadamente o tamanho do buffer.

*Nota 2: Foram realizadas 125 operações de upload “simultâneas”. **devido ao controle de thread-pool nem todas vão iniciar de imediato!*

Buffer de 4KB

Arquivos de 1MB

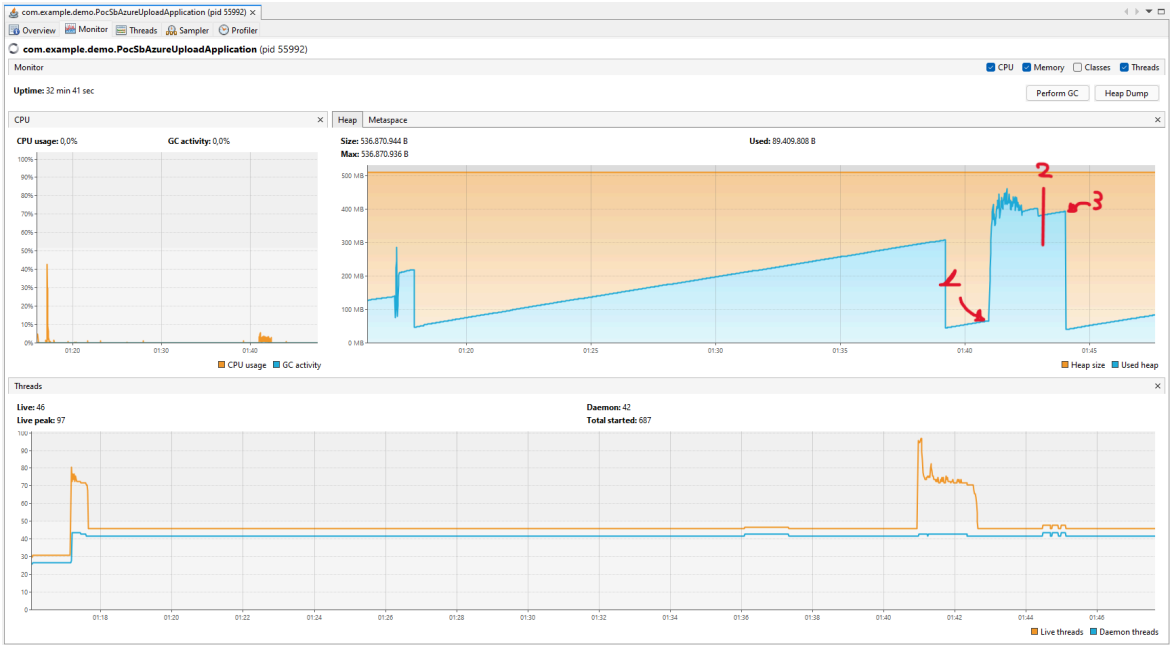




1. Início da operação
2. Aproximadamente o fim da operação
3. Execução do GC manualmente

Tempo: 10.6762119**s**
Média por arquivo: 1.556**s**
Pico de memória: 300.711**b**

Arquivos de 10MB



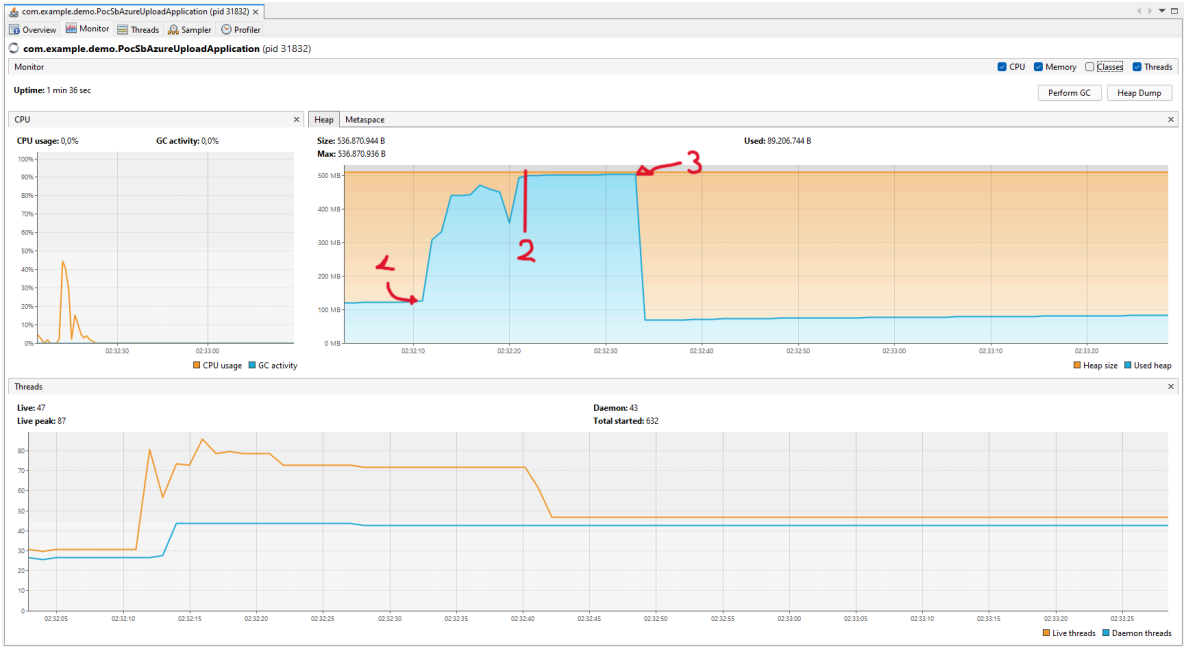
1. Início da operação
2. Aproximadamente o fim da operação
3. Execução do GC manualmente

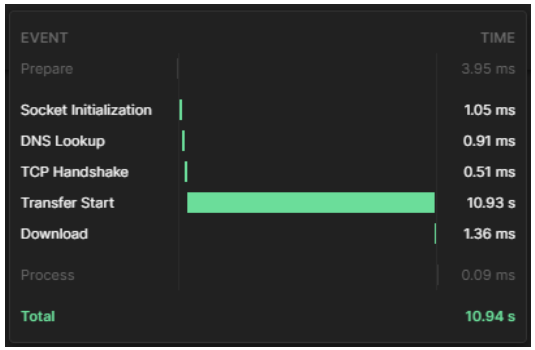
Tempo: 1**m** 21.1770179**s**
Média por arquivo: 15.721**s**
Pico de memória: 486.858**b**

Buffer de 4MB

Arquivos de 1MB

Para os casos de arquivos com 1MB nao vemos vantagem alguma, pelo contrário a performance deteriorou.
Devido ao alto volume de memória perdida o tempo de upload piorou. *Além da memória usada de forma indevida





1. Início da operação
2. Aproximadamente o fim da operação
3. Execução do GC automaticamente

Tempo: 10.6058811**s**
Média por arquivo: 2.022**s**
Pico de memória: 496.289**b**

Arquivos de 10MB

Não completos! Devido ao leak de memória muitas das operações estouraram o limite de memória.

```
java.lang.OutOfMemoryError: Java heap space
```

Buffer de 8MB

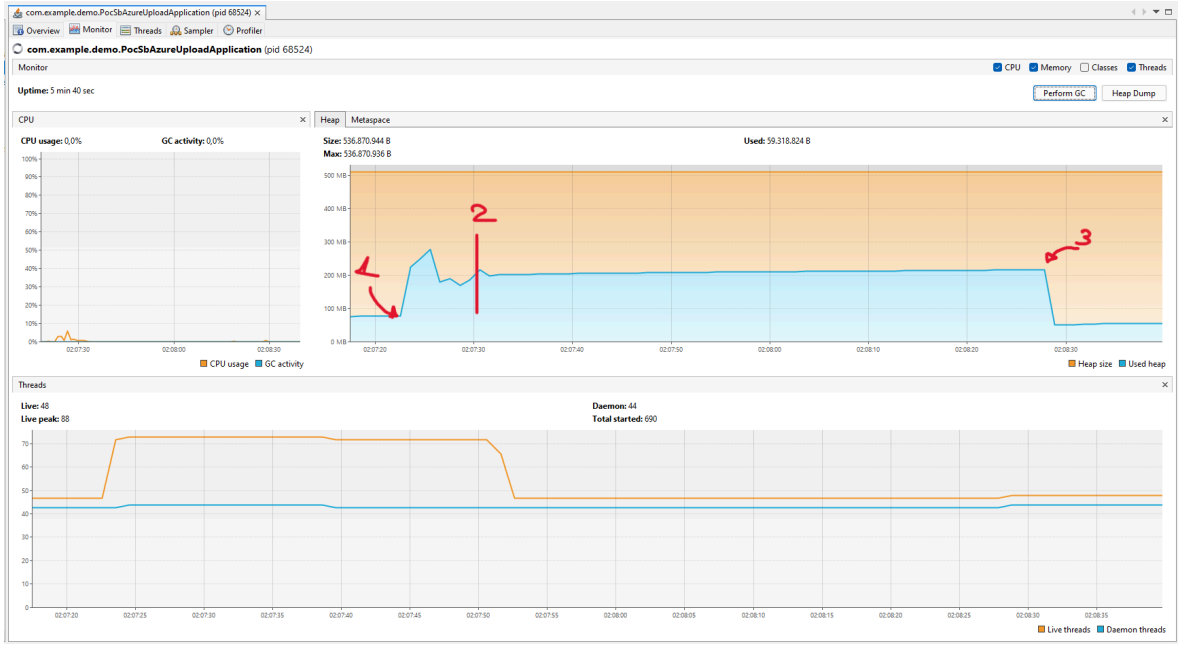
Devido ao Buffer de 4MB já não ser um sucesso, não será necessário concluir este!

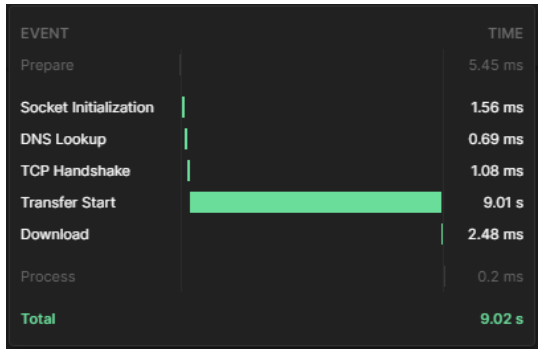
Buffering on Demand

Nesta implementação o uso do buffer será o tamanho do arquivo!

*Nota: Há um risco para ambientes onde o *throughput* e tamanho do arquivo não é controlado. Para a configuração realizada na aplicação é possível atingir o envio de 125 arquivos de 10MB sem estourar o limite de memória (subindo de 25 em 25 arquivos). Mas no mundo real sabemos que não é uma verdade, se realizado um teste de arquivos com 20MB a configuração de max-pool-size de 25 o resultado será de exceção por atingir o máximo de memória disponível. Caso adotada uma configuração menor que 25 é possível que os arquivos < 10MB sofram impacto, pois o *throughput* será menor.

Arquivos de 1MB

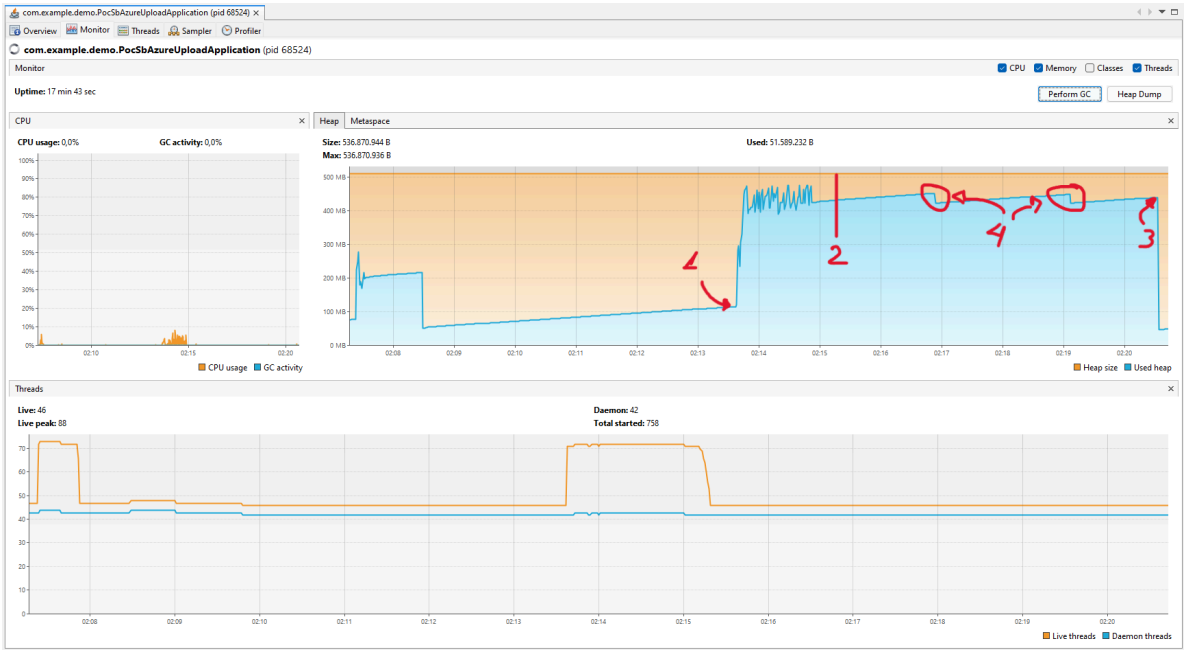


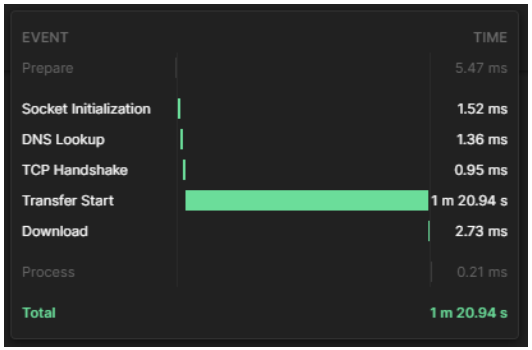


1. Início da operação
2. Aproximadamente o fim da operação
3. Execução do GC manualmente

Tempo: 9.0066449**s**
Média por arquivo: 1.686**s**
Pico de memória: 293.794**b**

Arquivos de 10MB





1. Início da operação
2. Aproximadamente o fim da operação
3. Execução do GC manualmente
4. Execução do GC automática

Tempo: 1**m** 20.908463**s**
Média por arquivo: 15.660**s**
Pico de memória: 499.453**b**

Conclusion

Dado os testes demonstrados aqui, é possível afirmar que a melhor implementação é a de streaming do arquivo. Isto é decorrente do uso da memória em buffer, o qual deixamos de certa forma a cargo da GC de quando limpar os resquícios já não utilizados mais, nos abstendo da necessidade de controles externos e maior capacidade de processamento simultâneo.

Mas não somente a implementação como também o conjunto de configurações ao redor da implementação também são importantes, é necessário avaliar a melhor configuração dado o ambiente e os arquivos que serão trafegados, dessa forma é possível realizar em harmonia o upload do arquivo, tendo a melhor performance para o cenário a ser executado.

Dessa forma, podemos considerar que a quantidade de execuções simultâneas está atrelado diretamente a quantidade de recurso disponível, mas também não se pode ignorar a capacidade das aplicações que proveem os arquivos assim como as de escrita dos mesmos.