Relatório

Planejamento:

O ambiente utilizado consistiu em containers Docker executando os serviços AirlinesHub, Exchange, Fidelity e IMDTravel, todos rodando em um computador com processador Intel i5-12400F (e o cooler que vem junto com ele). As requisições de carga foram geradas com a ferramenta **K6**.

Dois tipos de teste foram selecionados: **teste de capacidade**, para obter a reação do programa ao decorrer de cada vez mais requisições, e **teste de pico**, para observá-lo sob aumentos bruscos de carga. Esses testes foram escolhidos porque são os mais abrangentes e interessantes em questões de resultados.

Como parte do planejamento para começar os testes, defini como primeiro objetivo descobrir o limite de capacidade do programa sendo executado na minha máquina. Experimentando, com muita pena do processador, pude chegar em um valor aproximado de **200 requisições por segundo** antes de atingir o ponto de saturação, em que as requisições não tinham mais chance de sucesso.

Métricas de Desempenho à serem analisadas

- **Latência média (avg)**
- **Latência p95 (95º percentil)**
- **Taxa de falhas (http_req_failed)**
- **Throughput (req/s)**

Métricas de Disponibilidade à serem analisadas

- **Taxa de sucesso das requisições**
- **Uptime funcional durante o teste de pico**

Algo importante a mencionar é o comportamento do timeout do **“/sell”**, que lança uma falha após 2 segundos sem resposta.

Quando a falha ocorre, todas as requisições pelos próximos 10 segundos falham por causa do timeout. Assim, como em grande parte do teste o “/sell” está com a latência ativada, especialmente nos momentos com um número elevado de requisições por segundo, a taxa de sucesso é extremamente baixa.

Para contornar isso e fazer com que os testes realmente tivessem valores significativos, separei em 3 variações:

1. **Sem tolerância a falhas**.

2. **Com tolerância a falhas**, em que o timeout ainda lança erro.

3. **Com tolerância a falhas**, onde o timeout acontece, mas não lança erro; a compra é considerada um fracasso internamente, mas a requisição é tratada como sucesso.

Uma outra nota antes de mostrar os resultados dos testes, é que como os serviços não são executados paralelamente, e assim, compartilham o processador e disputam por recursos, acontecia muito de elas, em alta carga, monopolizar todo o cpu, fazendo com que só um serviço fosse executado até acabar todas as suas requisições.

Teste de capacidade

```
const tempoStr = "10s";

export const options = {
  scenarios: {
    carga: {
```

```
      executor: "ramping-arrival-rate",
      startRate: 5,          // começa com 5 req/s
      timeUnit: "1s",
      preAllocatedVUs: 20,   // VUs pré-alocados
      maxVUs: 10000,          // limite máximo de VUs
      stages: [
        { duration: tempoStr, target: 1 },
        { duration: tempoStr, target: 10 },
        { duration: tempoStr, target: 100 },
        { duration: tempoStr, target: 250 }, // vai subindo a RPS
        { duration: tempoStr, target: 0 },
      ],
    },
  },
  thresholds: {
    http_req_duration: ["p(95)<5500"], // meta: 95% das reqs < 5.5s
    http_req_failed: ["rate<0.05"],    // meta: < 5% de falhas
  },
};


         /\      Grafana   /‾‾/
    /\  /  \     |\  __   /  /
   /  \/    \    | |/ /  /   ‾‾\
  /          \   |   (  |  (‾)  |
 / __________ \  |_|\_\  \_____/

     execution: local
        script: Testes/testeDeCapacidade.js
        output: -

     scenarios: (100.00%) 1 scenario, 10000 max VUs, 1m20s max duration
(incl. graceful stop):
              * carga: Up to 250.00 iterations/s for 50s over 5 stages
(maxVUs: 20-10000, gracefulStop: 30s)
```

Resultados:

Não tolerante à falhas:

```
  THRESHOLDS

    http_req_duration
    ✗ 'p(95)<5500' p(95)=19.14s

    http_req_failed
    ✗ 'rate<0.05' rate=99.26%


      TOTAL RESULTS

          checks_total.......: 1893   23.656725/s
          checks_succeeded...: 0.73%  14 out of 1893
          checks_failed......: 99.26% 1879 out of 1893

          ✗ status é 200
            ↳  0% — ✓ 14 / ✗ 1879

          HTTP
          http_req_duration..............: avg=9.83s  min=18.27ms
med=7.55s max=1m0s   p(90)=18.09s p(95)=19.14s
            { expected_response:true }...: avg=4.3s   min=18.27ms
med=5.01s max=5.01s p(90)=5.01s  p(95)=5.01s
          http_req_failed................: 99.26% 1879 out of 1893
          http_reqs......................: 1893   23.656725/s

          EXECUTION
          dropped_iterations.............: 1284   16.046083/s
          iteration_duration.............: avg=10.83s min=1.01s
med=8.55s max=1m1s   p(90)=19.09s p(95)=20.14s
          iterations.....................: 1893   23.656725/s
          vus............................: 458    min=4
max=1267
          vus_max........................: 1294   min=20
max=1294

          NETWORK
          data_received..................: 557 kB 7.0 kB/s
          data_sent......................: 473 kB 5.9 kB/s
```

```
      running (1m20.0s), 00000/01294 VUs, 1893 complete and 458
interrupted iterations
```

É um resultado bastante interessante, por não ter timeout, fica a cargo do k6 decidir que a requisição falhou por omissão/latência com seu “gracefulStop”, e como a maioria das requisições falharam por esse motivo, o tempo está altíssimo. Além disso, algo que vale a pena comentar, o fidelity crashou depois de apenas 13 bonificações, fazendo com que todas as requisições que vieram depois falhassem.

Tolerante à falhas com timeout sendo erro:

```
  THRESHOLDS

    http_req_duration
    ✓ 'p(95)<5500' p(95)=4.02s

    http_req_failed
    ✗ 'rate<0.05' rate=99.60%


  TOTAL RESULTS

    checks_total.......: 2772   50.652474/s
    checks_succeeded...: 0.39%  11 out of 2772
    checks_failed......: 99.60% 2761 out of 2772

    ✗ status é 200
      ↳  0% — ✓ 11 / ✗ 2761

    HTTP
    http_req_duration..............: avg=4.01s min=2.05s med=4s
max=6.59s p(90)=4.01s p(95)=4.02s
      { expected_response:true }...: avg=4.97s min=3.01s med=4.96s
max=6.59s p(90)=6.55s p(95)=6.57s
    http_req_failed................: 99.60% 2761 out of 2772
    http_reqs......................: 2772   50.652474/s

    EXECUTION
    dropped_iterations.............: 862    15.751238/s
    iteration_duration.............: avg=5.01s min=3.05s med=5s
max=7.59s p(90)=5.02s p(95)=5.02s
    iterations.....................: 2772   50.652474/s
    vus............................: 12     min=4            max=870
    vus_max........................: 876    min=20           max=876

    NETWORK
    data_received..................: 817 kB 15 kB/s
    data_sent......................: 555 kB 10 kB/s
```

```
running (0m54.7s), 00000/00876 VUs, 2772 complete and 0 interrupted
iterations
carga ✓ [======================================] 00000/00876 VUs  50s
003.54 iters/s
ERRO[0055] thresholds on metrics 'http_req_failed' have been crossed
```

Por fim, apesar do tempo médio das requisições ser bem menor e do teste ser concluído mais rápido pela existência do timeout, a taxa de sucesso foi bem parecida com quando executado na versão não tolerante, requisições que ainda podiam ser concluídas, apesar de que com grande latência, acabaram sendo canceladas pelo timeout se tornando erros. Fazendo com que assim tivesse mais erros do que sua versão sem tolerância.

Tolerante à falhas, com timeout não jogando erro:

```
  ▉ THRESHOLDS

    http_req_duration
    ✓ 'p(95)<5500' p(95)=3.02s

    http_req_failed
    ✓ 'rate<0.05' rate=0.13%


  ▉ TOTAL RESULTS

    checks_total.......: 3002   55.998979/s
    checks_succeeded...: 99.86% 2998 out of 3002
    checks_failed......: 0.13%  4 out of 3002

    ✗ status é 200
      ↳  99% — ✓ 2998 / ✗ 4

    HTTP
    http_req_duration..............: avg=2.22s min=9.23ms med=2s
max=5.02s p(90)=3.01s p(95)=3.02s
      { expected_response:true }...: avg=2.22s min=9.23ms med=2s
max=5.02s p(90)=3.01s p(95)=3.02s
    http_req_failed................: 0.13%  4 out of 3002
    http_reqs......................: 3002   55.998979/s

    EXECUTION
    dropped_iterations.............: 632    11.789259/s
    iteration_duration.............: avg=3.22s min=1s     med=3.01s
max=6.03s p(90)=4.01s p(95)=4.02s
    iterations.....................: 3002   55.998979/s
    vus............................: 7      min=4          max=642
    vus_max........................: 651    min=20         max=651

    NETWORK
    data_received..................: 893 kB 17 kB/s
    data_sent......................: 601 kB 11 kB/s
```

```
running (0m53.6s), 00000/00651 VUs, 3002 complete and 0 interrupted
iterations
carga ✓ [======================================] 00000/00651 VUs  50s
003.54 iters/s
```

Com os timeout não sendo considerados erros, a grande maioria se configura um sucesso.

Sendo as únicas requisições falhas sendo motivadas pelas tentativas de obter informações do voo, que após 4 tentativas falhas lança erro no sistema.

Teste de Spike

```
export const options = {
  scenarios: {
    pico: {
      executor: "ramping-arrival-rate",
      startRate: 20,
      timeUnit: "1s",
      preAllocatedVUs: 50,   // VUs reservados
      maxVUs: 5000,
      stages: [
        { duration: "10s", target: 20 },

        { duration: "5s", target: 250 },

        { duration: "15s", target: 5 },

        { duration: "5s", target: 250 },

        { duration: "10s", target: 0 },
      ],
    },
  },
  thresholds: {
    http_req_failed: ["rate<0.05"],
  },
};



         /\      Grafana   /‾‾/
    /\  /  \     |\  __   /  /
   /  \/    \    | |/ /  /   ‾‾\
  /          \   |   (  |  (‾)  |
 / __________ \  |_|\_\  \_____/

     execution: local
        script: Testes/testeDeSpike.js
        output: -

     scenarios: (100.00%) 1 scenario, 5000 max VUs, 1m15s max duration
(incl. graceful stop):
              * pico: Up to 250.00 iterations/s for 45s over 5 stages
(maxVUs: 50-5000, gracefulStop: 30s)
```

Resultados:

Não tolerante à falhas:

```
  ▉ THRESHOLDS

    http_req_duration
    ✗ 'p(95)<5500' p(95)=18.21s

    http_req_failed
    ✗ 'rate<0.05' rate=99.18%


  ▉ TOTAL RESULTS

    checks_total.......: 2931   39.065581/s
    checks_succeeded...: 0.81%  24 out of 2931
    checks_failed......: 99.18% 2907 out of 2931

    ✗ status é 200
      ↳  0% — ✓ 24 / ✗ 2907

    HTTP
    http_req_duration..............: avg=4.58s min=4.13ms med=8.99ms
max=1m0s  p(90)=9.78s  p(95)=18.21s
      { expected_response:true }...: avg=2.3s  min=8.57ms med=17.57ms
max=5.01s p(90)=5.01s  p(95)=5.01s
    http_req_failed................: 99.18% 2907 out of 2931
    http_reqs......................: 2931   39.065581/s

    EXECUTION
    dropped_iterations.............: 1133   15.101093/s
    iteration_duration.............: avg=5.15s min=1s     med=1.01s
max=1m1s  p(90)=10.53s p(95)=16.72s
    iterations.....................: 2908   38.759027/s
    vus............................: 633    min=19           max=1145
    vus_max........................: 1168   min=50           max=1168

    NETWORK
    data_received..................: 836 kB 11 kB/s
    data_sent......................: 712 kB 9.5 kB/s
```

```
running (1m15.0s), 0000/1168 VUs, 2908 complete and 633 interrupted
iterations
pico ✓ [======================================] 0633/1168 VUs  45s
003.54 iters/s
ERRO[0075] thresholds on metrics 'http_req_duration, http_req_failed'
have been crossed
```

Mais uma vez o tempo médio de duração foi altíssimo, com uma taxa de sucesso minúscula.

Tolerante à falhas com timeout jogando erro:

```
  ▉ THRESHOLDS

    http_req_duration
    ✓ 'p(95)<5500' p(95)=4.01s

    http_req_failed
    ✗ 'rate<0.05' rate=99.86%


  ▉ TOTAL RESULTS

    checks_total.......: 3836   77.145798/s
    checks_succeeded...: 0.13%  5 out of 3836
    checks_failed......: 99.86% 3831 out of 3836

    ✗ status é 200
      ↳  0% — ✓ 5 / ✗ 3831

    HTTP
    http_req_duration..............: avg=3.99s    min=3.13ms   med=4s
max=4.04s    p(90)=4.01s    p(95)=4.01s
      { expected_response:true }...: avg=874.46ms min=740.96ms
med=883.78ms max=975.49ms p(90)=958.48ms p(95)=966.99ms
    http_req_failed................: 99.86% 3831 out of 3836
    http_reqs......................: 3836   77.145798/s

    EXECUTION
    dropped_iterations.............: 838    16.853018/s
    iteration_duration.............: avg=4.99s    min=1s       med=5s
max=5.04s    p(90)=5.01s    p(95)=5.01s
    iterations.....................: 3836   77.145798/s
    vus............................: 12     min=12           max=884
    vus_max........................: 887    min=50           max=887

    NETWORK
    data_received..................: 1.1 MB 23 kB/s
    data_sent......................: 768 kB 15 kB/s
```

```
running (0m49.7s), 0000/0887 VUs, 3836 complete and 0 interrupted
iterations
pico ✓ [======================================] 0000/0887 VUs  45s
003.54 iters/s
ERRO[0050] thresholds on metrics 'http_req_failed' have been crossed
```

Novamente a taxa de sucesso é semelhante à versão sem tolerância, porém com uma taxa média de duração muito menor.

Tolerante à falhas com timeout não jogando erro:

```
  ▉ THRESHOLDS

    http_req_duration
    ✓ 'p(95)<5500' p(95)=4.02s

    http_req_failed
    ✗ 'rate<0.05' rate=92.93%


  ▉ TOTAL RESULTS

    checks_total.......: 3821   76.845434/s
    checks_succeeded...: 7.06%  270 out of 3821
    checks_failed......: 92.93% 3551 out of 3821

    ✗ status é 200
      ↳  7% — ✓ 270 / ✗ 3551

    HTTP
    http_req_duration..............: avg=3.96s min=10.2ms med=4s
max=7.13s p(90)=4.01s p(95)=4.02s
      { expected_response:true }...: avg=3.35s min=10.2ms med=3.06s
max=7.13s p(90)=5.22s p(95)=5.68s
    http_req_failed................: 92.93% 3551 out of 3821
    http_reqs......................: 3821   76.845434/s

    EXECUTION
    dropped_iterations.............: 853    17.154974/s
    iteration_duration.............: avg=4.96s min=1.01s  med=5s
max=8.13s p(90)=5.01s p(95)=5.02s
    iterations.....................: 3821   76.845434/s
    vus............................: 12     min=12           max=884
    vus_max........................: 889    min=50           max=889

    NETWORK
    data_received..................: 1.1 MB 23 kB/s
    data_sent......................: 765 kB 15 kB/s
```

```
running (0m49.7s), 0000/0889 VUs, 3821 complete and 0 interrupted
iterations
pico ✓ [======================================] 0000/0889 VUs  45s
003.54 iters/s
ERRO[0050] thresholds on metrics 'http_req_failed' have been crossed
```

Apesar de o pico de RPS ser o limite estipulado, provavelmente devido à algumas variáveis no consumo do cpu (e bastante calor), acabou acontecendo o problema de um único endpoint monopolizar os processos, e quando ele fazia uma requisição que, por sua vez, era alvo de timeout, ele falhava diversas requisições até acabar suas atividades. Por fim, nenhum dos 3 modos conseguiu se recuperar do primeiro pico.

Nota:

Dias após as realizações dos testes, realizei uma limpeza em todo o computador, e executando os testes novamente, obtive resultados bem diferentes.

```
WARN[0034] Request Failed                                error="Post
\"http://localhost:3001/buyTicket\": EOF"


  ▉ THRESHOLDS

    http_req_duration
    ✓ 'p(95)<5500' p(95)=3.01s

    http_req_failed
    ✓ 'rate<0.05' rate=0.22%


  ▉ TOTAL RESULTS

    checks_total.......: 4087   81.474933/s
    checks_succeeded...: 99.77% 4078 out of 4087
    checks_failed......: 0.22%  9 out of 4087

    ✗ status é 200
      ↳  99% — ✓ 4078 / ✗ 9

    HTTP
    http_req_duration..............: avg=2.22s min=522.29µs med=2s
max=5.05s p(90)=3s    p(95)=3.01s
      { expected_response:true }...: avg=2.21s min=7.37ms   med=2s
max=5.05s p(90)=3s    p(95)=3.01s
    http_req_failed................: 0.22%  9 out of 4087
    http_reqs......................: 4087   81.474933/s
```

```
    EXECUTION
    dropped_iterations.............: 587    11.70193/s
    iteration_duration.............: avg=3.22s min=1s       med=3s
max=6.05s p(90)=4.01s p(95)=4.01s
    iterations.....................: 4087   81.474933/s
    vus............................: 1      min=1         max=629
    vus_max........................: 635    min=50        max=635

    NETWORK
    data_received..................: 1.2 MB 24 kB/s
    data_sent......................: 818 kB 16 kB/s




running (0m50.2s), 0000/0635 VUs, 4087 complete and 0 interrupted
iterations
pico ✓ [======================================] 0000/0635 VUs  45s
003.54 iters/s
```

O primeiro teste teve resultado uma taxa de sucesso altíssima, com uma requisição ao /buyticked falhando.

```
  ▉ THRESHOLDS

    http_req_duration
    ✓ 'p(95)<5500' p(95)=4s

    http_req_failed
    ✗ 'rate<0.05' rate=49.03%


  ▉ TOTAL RESULTS

    checks_total.......: 3922   78.430442/s
    checks_succeeded...: 50.96% 1999 out of 3922
    checks_failed......: 49.03% 1923 out of 3922

    ✗ status é 200
      ↳  50% — ✓ 1999 / ✗ 1923
```

```
    HTTP
    http_req_duration..............: avg=3.08s min=9.41ms med=4s
max=5.02s p(90)=4s p(95)=4s
      { expected_response:true }...: avg=2.19s min=9.41ms med=2s
max=5.02s p(90)=3s p(95)=3.01s
    http_req_failed................: 49.03% 1923 out of 3922
    http_reqs......................: 3922   78.430442/s

    EXECUTION
    dropped_iterations.............: 753    15.058165/s
    iteration_duration.............: avg=4.08s min=1s     med=5s
max=6.02s p(90)=5s p(95)=5s
    iterations.....................: 3922   78.430442/s
    vus............................: 1      min=1            max=795
    vus_max........................: 798    min=50           max=798

    NETWORK
    data_received..................: 1.2 MB 23 kB/s
    data_sent......................: 785 kB 16 kB/s




running (0m50.0s), 0000/0798 VUs, 3922 complete and 0 interrupted
iterations
pico ✓ [======================================] 0000/0798 VUs  45s
003.54 iters/s
ERRO[0050] thresholds on metrics 'http_req_failed' have been crossed
```

E o segundo teste com uma taxa de sucesso menor - sobrevivendo ao primeiro pico, porém não conseguindo se recuperar do segundo -, apesar de muito maior que os resultados dos testes com refrigeração inferior.

Conclusão:

A latência do /sell foi, sem dúvidas, o fator de maior impacto nos testes. Como a janela total de execução era de aproximadamente 1 minuto e a latência artificial eram 10 segundos, bastava um número relativamente alto de requisições simultâneas para garantir repetidamente o evento de alta latência. Em consequência, durante boa parte do teste o endpoint permanecia em falha.

Outra falha de grande impacto foi o crash do Fidelity, que no modo sem tolerância à falhas tinha muito impacto na taxa de sucesso: a partir do momento em que o serviço caía, não era mais possível concluir nenhuma requisição com sucesso.

O método de tolerância do buscarVoo também se mostrou ineficiente em cenários de carga elevada. Como o projeto é single-threaded, o loop de tentativas consecutivas monopolizava o event loop do serviço, impedindo o /flight de receber e processar as novas requisições. Isso resultava em uma situação de bloqueio, onde o sistema não conseguia se recuperar e acabava lançando falhas sucessivas.

Após a limpeza do computador, otimizando o fluxo de ar e permitindo assim, o processador operar em uma frequência maior. Os resultados dos testes se mostraram inegavelmente superiores, já que a maior velocidade de processamento causou com que as requisições, que são executadas consecutivamente e não paralelamente, fossem realizadas mais rapidamente, fazendo com que o acúmulo do /buyTicket não fosse suficiente para causar o problema de monopolização do buscarVoo.

Por fim, a conclusão dos métodos de tolerância a falhas é que, apesar do tempo de execução ser menor e algumas falhas não apresentarem mais problemas. O fato dos serviços não executarem suas funções paralelamente faz com que a existência de um timeout, acabe se tornando mais prejudicial do que benéfico em situações de sobrecarga. Mesmo com a boa responsividade e proteção à falhas do fidelity e do exchange, o imdtravel monopolizava o processador em situações de sobrecarga, de forma que o airlineshub não é capaz de responder à tempo por inanição, e quando responde pode estar em alta latência, o estado qual permanecia por maior parte do tempo, fazendo com que a versão tolerante à falhas que lança erro no timeout tenha menor taxa de sucesso que a versão não tolerante, que ainda teria a possibilidade de concluir as requisições com alta latência com sucesso. Por fim, as formas de melhorar esse sistema seria por uma melhora no hardware ou, mais eficientemente, melhorar o código para que os serviços fossem executados paralelamente.