SISTEMAS DISTRIBUIDOS

# TICKETSAFE

*Uma solução eficiente para eventos*

# ENTREGA 3 - FINAL

*Fabio Ofugi*
*Paulo Victor*
*Rafael Melo*
*Rilbert Teixeira*
*Vinicius Soares*

# REDIS

- **Banco open source que armazena dados em memória:**

Todos os dados do Redis residem na memória principal do seu servidor, em contraste com a maioria dos sistemas de gerenciamento de banco de dados que armazenam dados em disco ou SSDs. Ao eliminar a necessidade de acessar discos, bancos de dados na memória, como o Redis, evitam atrasos de tempo de busca e podem acessar dados com algoritmos mais simples que usam menos instruções de CPU. Operações comuns exigem menos do que 10 milissegundos para serem executadas.

Possui natureza single-thread, ou seja, processa uma operação por vez, mesmo que receba varias. Porém a partir da versão 6.0 utiliza multi-threads que realizam I/O (entrada e saída) de forma assíncrona em background.

Uma requisição é processada em 4 estágios:

- 1 - Lendo a requisição do socket/cliente.
- 2 - Traduzindo/parsing.
- 3 - Processando.
- 4 - Respondendo ao cliente.

Os dois primeiros estágios são delegados para outras threads e ficam aguardando para que a main thread prossiga com o processamento e posterior resposta.

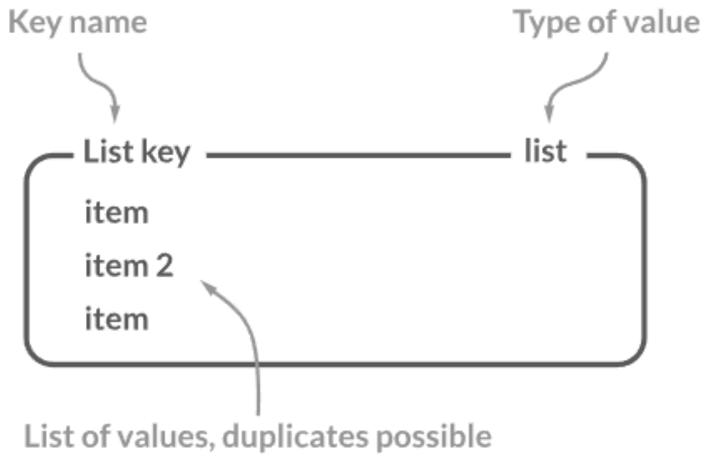
# REDIS

- **Casos de uso do REDIS**

- Gerenciamento de sessões (Time To Live).
- Classificações em tempo real (Sorted Set).
- Filas (atômica).
- Chat e sistema de mensagens (PUB/SUB) (Streams).
- Monitoramento de sensores.
- Event sourcing (rastrear ações de usuários, clicks, etc)

# REDIS

- **Estruturas de dados (List)**




## Listas

Redis Lists mantém coleções de elementos de string classificados de acordo com sua ordem de inserção. Empurre ou retire itens de ambas as extremidades, corte com base em deslocamentos, leia itens individuais ou múltiplos ou encontre ou remova itens por valor e posição. Você também pode fazer chamadas de bloqueio para transferências de mensagens assíncronas.

Redis para ingestão rápida de dados →

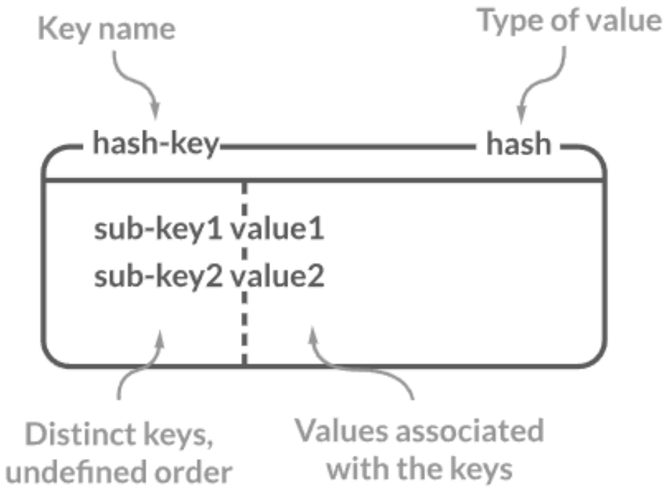
# REDIS

- **Estruturas de dados (Hash)**

## Hashes

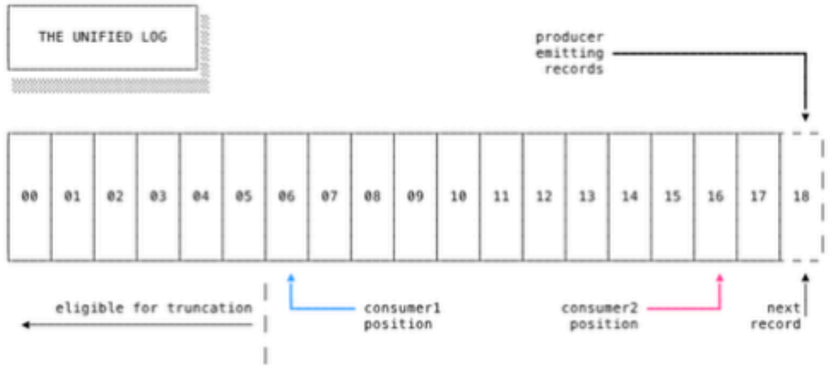
Um hash Redis é um tipo de dado que representa um mapeamento entre um campo de string e um valor de string. A estrutura Redis Hashes armazena um conjunto de pares de campo-valor projetados para não ocupar muito espaço, tornando-os ideais para representar objetos de dados. Ele fornece a capacidade de adicionar, buscar ou remover itens individuais, buscar o hash inteiro ou usar um ou mais campos no hash como um contador.

Crie e implante seu contêiner Redis com docker →



# REDIS

- **Estruturas de dados (Stream) - Broker em memória**




## Fluxos

O Redis Streams é uma estrutura de dados incrivelmente poderosa para gerenciar fluxos de dados de alta velocidade (como uma fila de mensagens). Com particionamento, replicação e persistência prontos para uso, ele pode capturar e processar milhões de pontos de dados por segundo em latência de submilissegundos. O Redis Streams é baseado em uma implementação eficiente de radix-tree (um algoritmo em que cada nó que é o único filho é mesclado com o pai), o que torna as consultas de intervalo e pesquisa extremamente rápidas. Ele conecta produtores e consumidores com chamadas assíncronas e oferece suporte a grupos de consumidores.

Tutorial: Como construir aplicativos usando fluxos Redis →

# PROBLEMAS IMPORTANTES

*POSSÍVEIS FORMAS DE TRATAR*

- **Concorrência e condição de corrida**

- **Operações atômicas simples:**
*Comandos únicos para incrementar/decrementar o valor em um hash; adicionar/remover um elemento de uma lista; calcular interseção, união e diferença de conjuntos; ou obter o membro com a classificação mais alta em um conjunto classificado.*

**- Transações:**
*Enfileiramento de mais de um comando simples para execução como um bloco atômico (cancelamento em casos de falhas).*

**- Scripts Lua:**
*Construção de um bloco de programa que é enviado ao servidor REDIS e executado como um único comando atômico.*

**- Locks explícitos:**
*Semáforos distribuídos para garantir o bloqueio controlado a determinado recurso compartilhado.*

# PROBLEMAS IMPORTANTES

*POSSÍVEIS FORMAS DE TRATAR*

- **Operações atômicas simples:**

```
try {
  // publica uma mensagem ou requisição no canal de STREAM para os workers disponiveis
  const result = await redisClient.xadd('reservas_pendentes', '*', 'eventoId', eventoId, 'userId', userId, 'quantidade', quantidade);
```

- **Transações:**

```
// Função para processar a reserva expirada
const processExpiredReservation = async (reservationKey) => {
    const [_, eventoId, userId, timestamp, quantidade] = reservationKey.split(':');

    // Iniciar uma transação no Redis
    const transaction = redisClient.multi();

    // Incrementa a quantidade de ingressos disponíveis no evento
    transaction.incrby(`evento:${eventoId}:ingressosDisponiveis`, quantidade);
    console.log(`Quantidade de ingressos a ser atualizada para o evento ${eventoId}: +${quantidade}`);

    // Publica uma notificação para cada quantidade de ingressos
    // possibilitando que varios usuarios que estejam esperando na fila de espera sejam atendidos
    for (let i = 0; i < quantidade; i++) {
        const message = `reserva_cancelada:${eventoId}:${userId}:${timestamp}:${quantidade}`;
        transaction.publish('reservas_solicitadas', message);
      }

    // Executa a transação no Redis
    await transaction.exec();
};
```

# PROBLEMAS IMPORTANTES

*POSSÍVEIS FORMAS DE TRATAR*

**- Scripts Lua:**

```
const script = `
-- Verifica e realiza claim de mensagens seguras para processamento

local eventoKey = KEYS[1]
local lockKey = KEYS[2]
local quantidade = tonumber(ARGV[1])
local reservationKey = ARGV[2]
local expireTime = tonumber(ARGV[3])
local waitingListKey = KEYS[3]
local waitingRequest = ARGV[4]
local streamKey = KEYS[4]
local idRequest = ARGV[5]
local groupName = ARGV[6]

-- Verifica a disponibilidade de ingressos
local quantidadeDisponivel = tonumber(redis.call('GET', eventoKey))

if quantidadeDisponivel and quantidadeDisponivel >= quantidade then
    -- Decrementa e cria a reserva
    redis.call('DECRBY', eventoKey, quantidade)
    redis.call('HSET', reservationKey, 'pagamento_efetuado', 'false')
    redis.call('EXPIRE', reservationKey, expireTime)

    -- Registra a mensagem como processada
    redis.call('DEL', lockKey)
    redis.call('XACK', streamKey, groupName, idRequest)
    return 1
else
    -- Mensagem em fila de espera
    redis.call('LPUSH', waitingListKey, waitingRequest)
    redis.call('DEL', lockKey)
    redis.call('XACK', streamKey, groupName, idRequest)
    return 0
end
`;
```

**- Locks explícitos:**

```
while (true){
    const result = await redisClient.xreadgroup(
        'GROUP', groupName, consumerName,
        'COUNT', 1,
        'BLOCK', 5000, // Bloqueia por até 5 segundos aguardando novas mensagens, prossegue assim que uma m
        'STREAMS', streamKey, '>'
    );
    if (result && result.length > 0) {
        for (const [stream, messages] of result) {
            for (const [idRequest, fields] of messages) {
                const lockKey = `lock:${idRequest}`;

                const lockAcquired = await redisClient.set(lockKey, idRequest, 'NX', 'EX', lockTTL);

                if (lockAcquired) {
                    await callProcessStream(idRequest, fields, lockKey);
                } else {
                    console.log(`O item ${idRequest} já está sendo processado.`);
                }
            }
        }
    } else {
        await sleep(1000); // Espera 1 segundo antes de tentar novamente
    }
}
```

# PROBLEMAS IMPORTANTES

*POSSÍVEIS FORMAS DE TRATAR*

- **Justiça**

- Ids únicos de entrada no Stream para cada requisição de usuário (mesmo que sejam no mesmo milissegundo).

- É possível informar manualmente o timestamp como parâmetro do lado do cliente (evitar injustiças de ordem em caso de atrasos na rede ou falhas entre o cliente e o servidor redis).

### Entry IDs

The entry ID returned by the XADD command, and identifying univocally each entry inside a given stream, is composed of two parts:

```
<millisecondsTime>-<sequenceNumber>
```

The milliseconds time part is actually the local time in the local Redis node generating the stream ID, however if the current milliseconds time happens to be smaller than the previous entry time, then the previous entry time is used instead, so if a clock jumps backward the monotonically incrementing ID property still holds. The sequence number is used for entries created in the same millisecond. Since the sequence number is 64 bit wide, in practical terms there is no limit to the number of entries that can be generated within the same millisecond.

The format of such IDs may look strange at first, and the gentle reader may wonder why the time is part of the ID. The reason is that Redis streams support range queries by ID. Because the ID is related to the time the entry is generated, this gives the ability to query for time ranges basically for free. We will see this soon while covering the XRANGE command.

# PROBLEMAS IMPORTANTES

*POSSÍVEIS FORMAS DE TRATAR*

- **Lentidão**

- Escalar quantidade de instancias do serviço responsável por incluir as requisições no Stream.

- Escalar quantidade de workers responsáveis por processar as requisições do Stream .

- Uso de pipelines, transactions e Lua scripts a fim de eliminar o Round-trip time (RTT).

- Em casos de alta demanda de processamento no servidor Redis considerar a utilização do Redis Cluster com divisão em shards e balanceamento de requisições entre instâncias Redis por região, utilizando o modelo de arquitetura mestre-réplica.

- **Persistência**

- **RDB (Redis Database):** Cria snapshots do banco de dados periodicamente. São arquivos pequenos e compactos tornando a recuperação rápida, no entanto, uma desvantagem é que esses arquivos podem ser corrompidos em caso de uma queda de energia repentina.

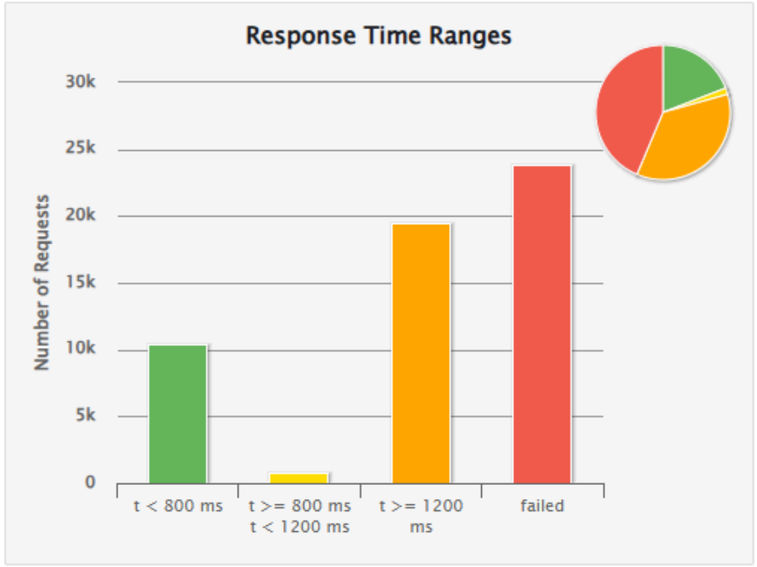
- **AOF (Append Only File):** Anexa todas as operações e comandos recebidos pelo servidor em um arquivo, esses comandos podem ser executados novamente para reconstrução do banco. São mais duráveis e não sofrem corrompimento, no entanto dependendo da política de sincronização podem ser muito grandes e a recuperação no momento de recriar o banco ser mais demorada.

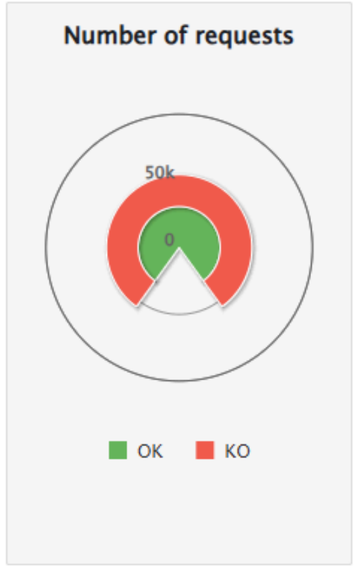
- **Postgresql :** Banco relacional durável para guardar as informações de reservas que foram de fato confirmadas.

# TESTES DE CARGA

- **Gatling (ramping)** **1 instância de reserveService (10% cpu quota)**

## ReservaSimulation




**Gatling Version**

**Version:** 3.12.0

**Released:** 2024-09-09

**Run Information**

**Date:** 2024-11-10 03:48:54 GMT

**Duration:** 4m 53s

**Description:** —

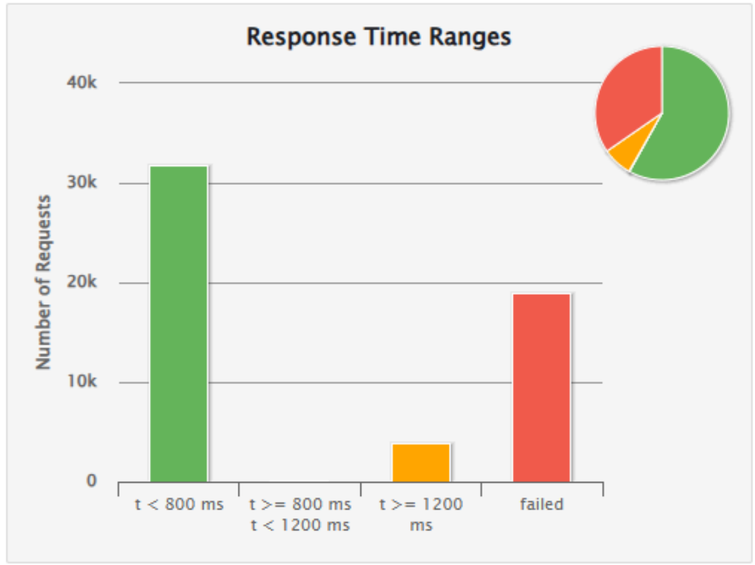
Expand all groups | Collapse all groups

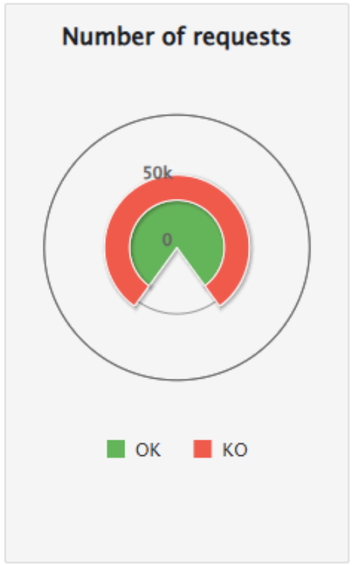
| Requests | Executions | | | | | Response Time (ms) | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Total | OK | KO | % KO | Cnt/s | Min | 50th pct | 75th pct | 95th pct | 99th pct | Max | Mean | Std Dev |
| All Requests | 54633 | 30776 | 23857 | 43.67 | 185.83 | 1 | 21681 | 61741 | 85802 | 94898 | 103118 | 32469 | 31000 |
| Reservar Ingresso | 54633 | 30776 | 23857 | 43.67 | 185.83 | 1 | 21680 | 61741 | 85802 | 94898 | 103118 | 32469 | 31000 |

# TESTES DE CARGA

- Gatling (ramping) 5 instâncias de reserveService (10% cpu quota)

## ReservaSimulation




**Gatling Version**

**Version:** 3.12.0

**Released:** 2024-09-09

**Run Information**

**Date:** 2024-11-12 20:11:02 GMT

**Duration:** 5m 13s

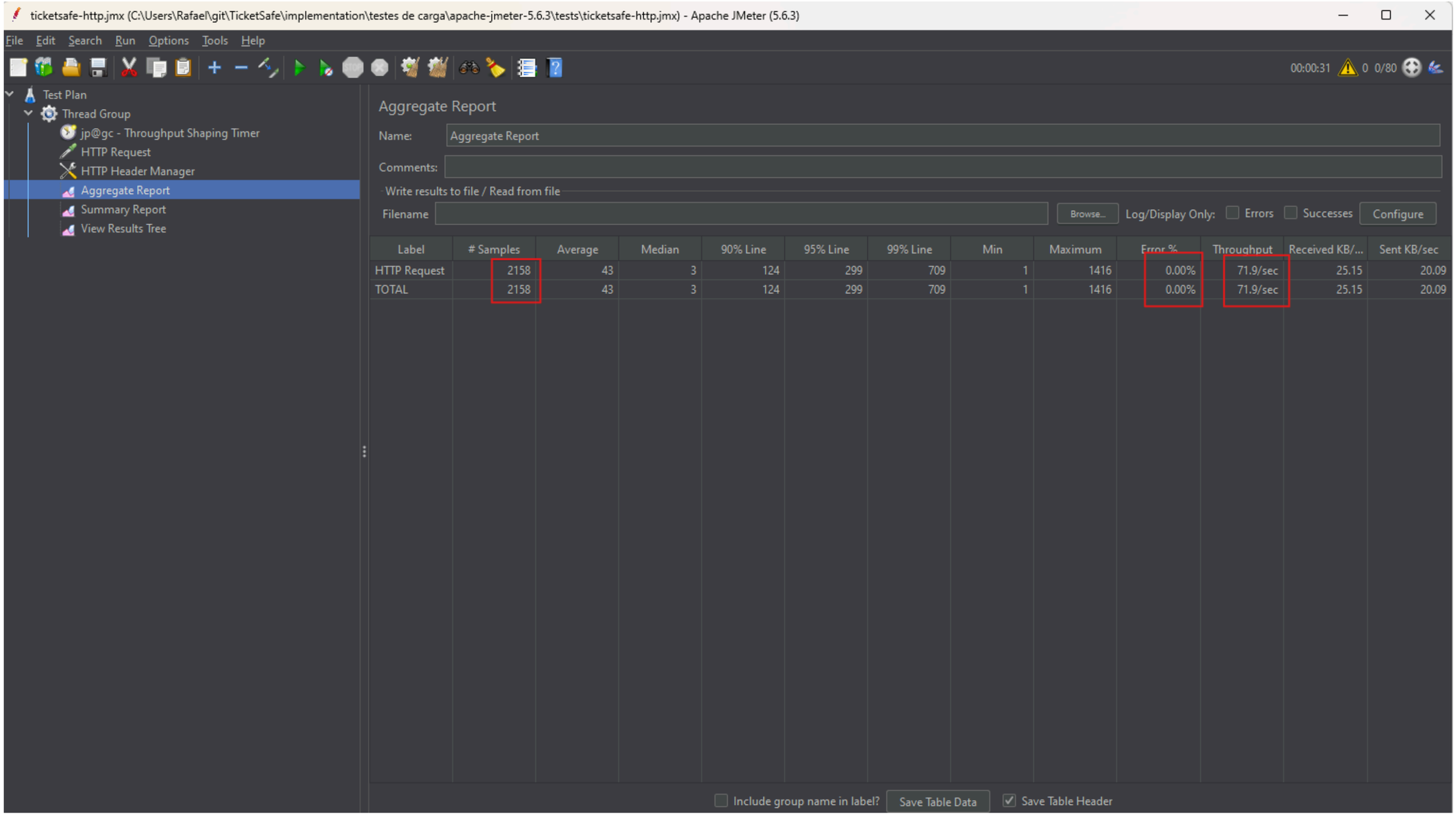
**Description:** —

Expand all groups | Collapse all groups

| Requests | Executions | | | | | Response Time (ms) | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Total | OK | KO | % KO | Cnt/s | Min | 50th pct | 75th pct | 95th pct | 99th pct | Max | Mean | Std Dev |
| All Requests | 54640 | 35715 | 18925 | 34.64 | 174.01 | 1 | 14 | 55189 | 91103 | 100967 | 108025 | 24998 | 34195 |
| Reservar Ingresso | 54640 | 35715 | 18925 | 34.64 | 174.01 | 1 | 14 | 55189 | 91103 | 100967 | 108025 | 24998 | 34195 |

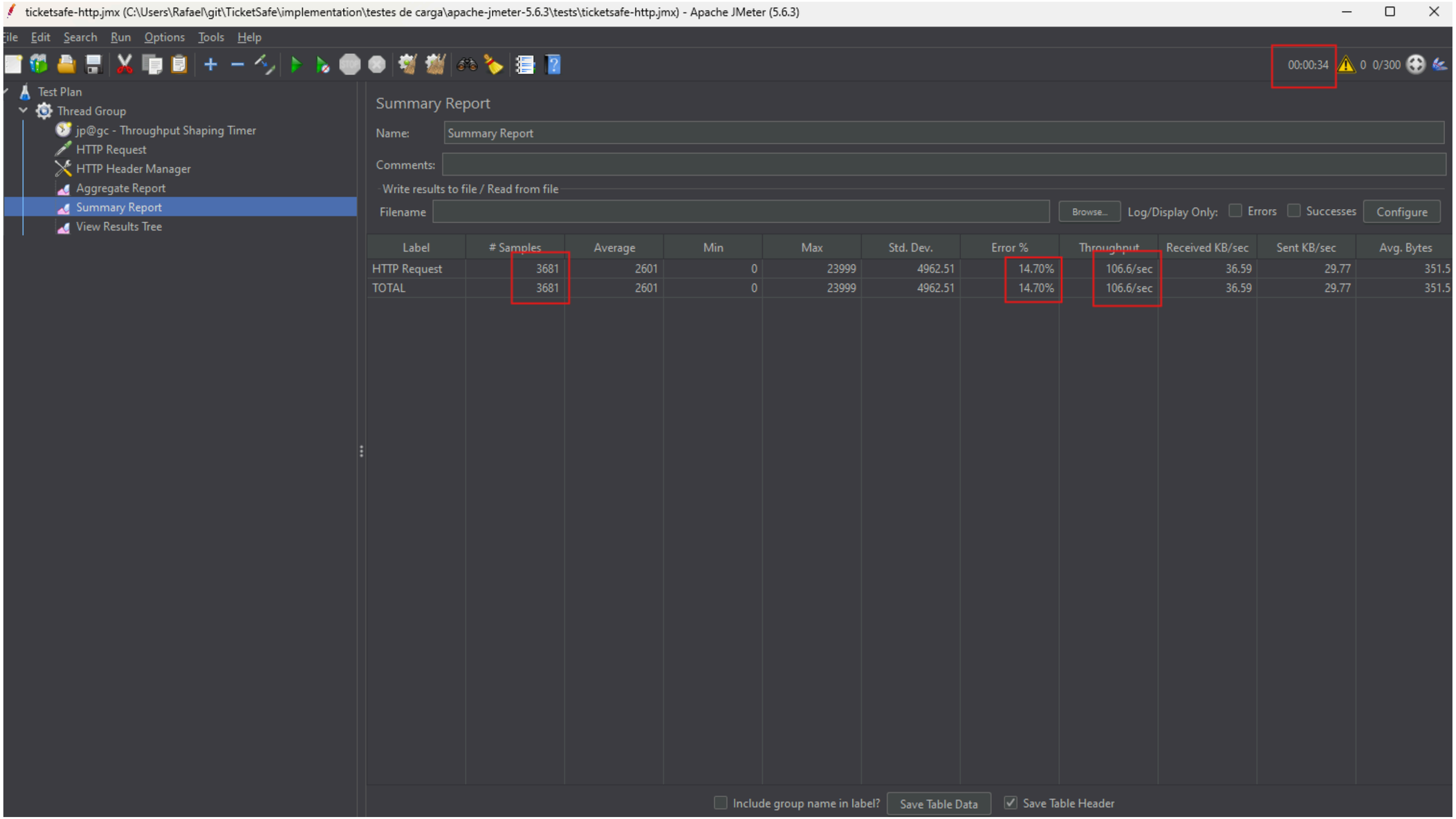
# TESTES DE CARGA

- Jmeter (constante) **1 instância 70 RPS**



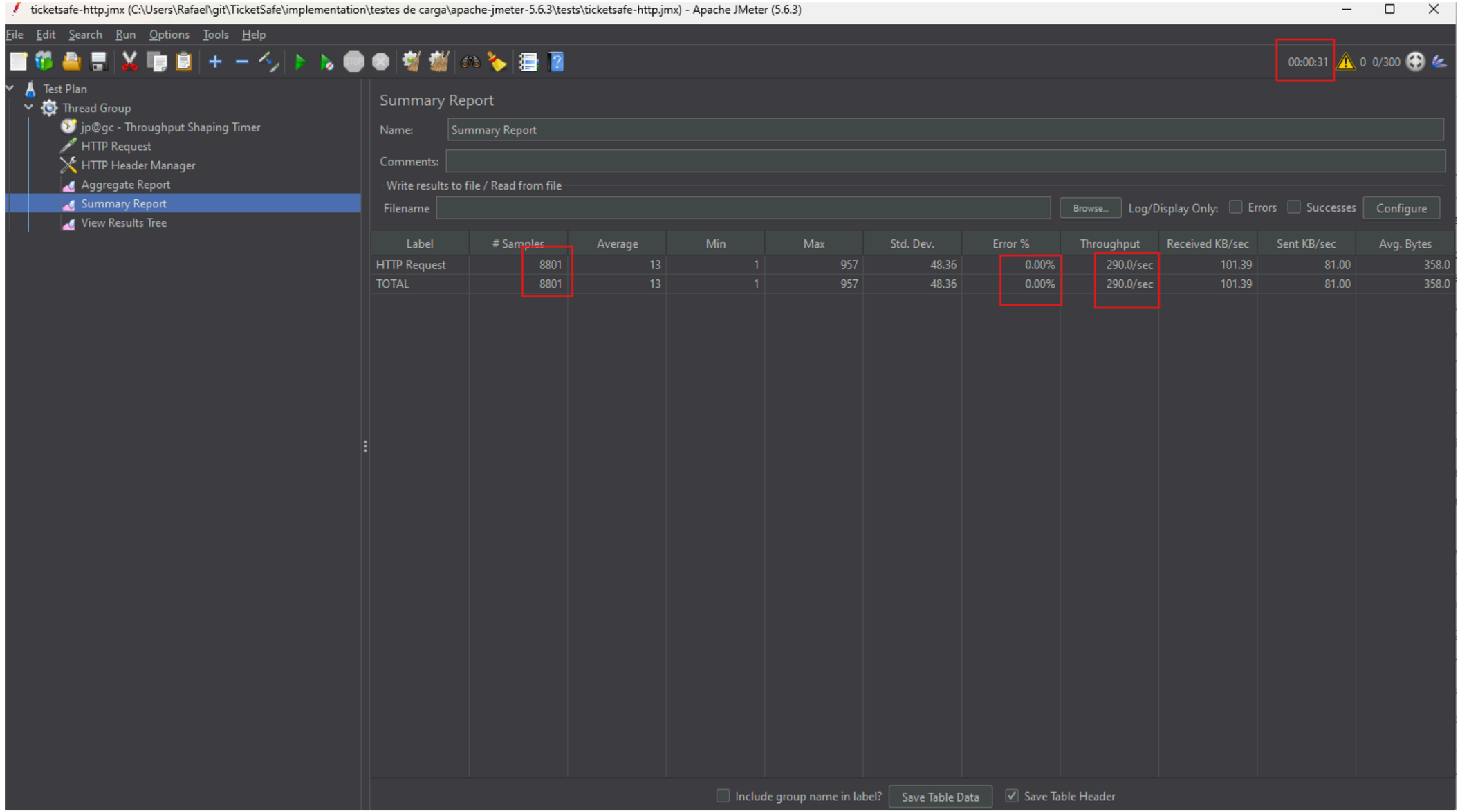
# TESTES DE CARGA

- Jmeter (constante) **1 instância 300 RPS**



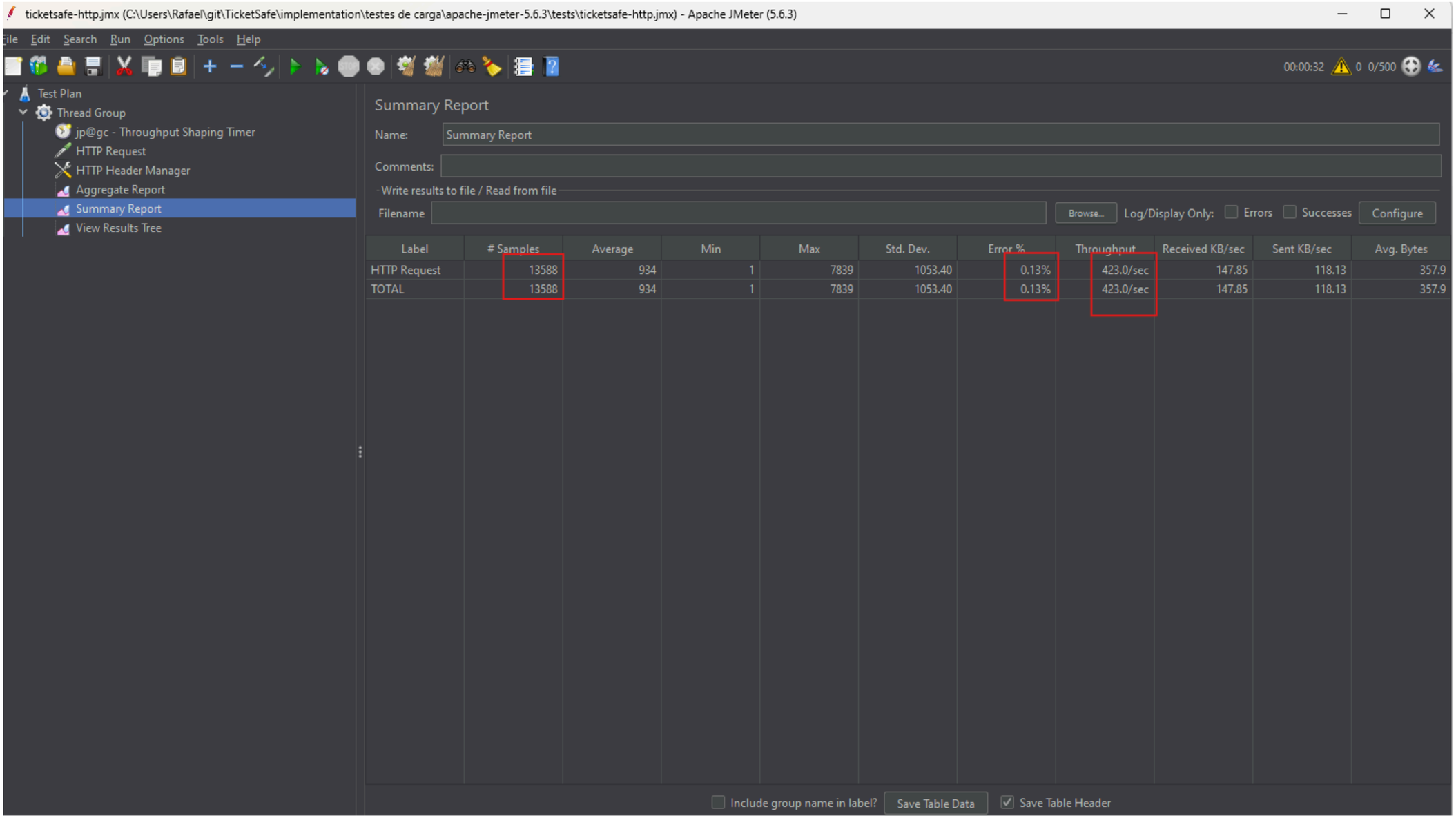
# TESTES DE CARGA

- Jmeter (constante) **5 instâncias 300 RPS**



# TESTES DE CARGA

- Jmeter (constante) **5 instâncias 500 RPS**



# OBRIGADO!

https://github.com/Sistemas-Distribuidos-UFG-2024-2/TicketSafe

*Fabio Ofugi*
*Paulo Victor*
*Rafael Melo*
*Rilbert Teixeira*
*Vinicius  Soares*

SISTEMAS DISTRIBUIDOS