Computação Distribuída (Web services)

Prof Alcides/Prof Mario

Laboratório

Sistema de Controle de Telescópio Espacial Compartilhado (SCTEC)

Vocês foram contratados para desenvolver o núcleo do sistema de agendamento do novo telescópio espacial acadêmico. Cientistas do mundo todo competirão por tempo de observação neste equipamento, que custa milhões de dólares por hora para operar. Nosso desafio não é apenas criar uma API de agendamento, mas garantir que ela seja justa, consistente, à prova de falhas em um ambiente de altíssima concorrência.

Primeiro vamos entender os conceitos e as ferramentas que nos permitirão construir a solução.

Fundamentos teóricos

O que é um Web Service?

Um Web Service é um sistema projetado para permitir a comunicação entre diferentes máquinas através de uma rede. Ele expõe funcionalidades que podem ser requisitadas por outras aplicações, independentemente da linguagem ou sistema operacional.

O que são APIs e o Padrão REST?

- API (Application Programming Interface): É um contrato, um conjunto de regras que permite que diferentes aplicações se comuniquem. A API define quais funcionalidades estão disponíveis, que dados são necessários e o que será recebido como resposta.
- REST (REpresentational State Transfer): É um estilo de arquitetura para a criação de Web Services.

API RESTful

1. Arquitetura Cliente-Servidor: Cliente (quem pede) e servidor (quem responde) são separados.
2. Stateless (sem estado): Cada requisição contém toda a informação necessária para ser processada. O servidor não guarda o "contexto" do cliente.
3. Interface uniforme: Esta é a restrição mais importante do REST.
    - Recursos são identificados por URIs: /telescopios/hubble, /agendamentos/123.
    - Uso dos Métodos HTTP: GET (ler), POST (criar), PUT (atualizar), DELETE (remover).

- Representações: Os recursos são trocados em um formato como JSON.
- HATEOAS (Hypermedia as the Engine of Application State): Este é o nível mais alto de maturidade REST. Significa que a resposta do servidor não deve conter apenas os dados do recurso, mas também links para as próximas ações possíveis que podem ser executadas sobre aquele recurso.

Ferramentas: Flask e Node.js

- Flask (Python): É um "microframework" para desenvolvimento web em Python. Usaremos o Flask para construir nosso Serviço de Agendamento. Ele é excelente para criar APIs RESTful de forma rápida e limpa, cuidando da lógica de negócio principal e da comunicação com o banco de dados. Ele será o "cérebro" do nosso sistema.
- Node.js (JavaScript): É um ambiente de execução de JavaScript no lado do servidor. Sua principal característica é a arquitetura orientada a eventos e I/O (Entrada/Saída) não-bloqueante. Isso o torna extremamente eficiente para tarefas que envolvem muitas conexões simultâneas ou tempo de espera, como gerenciar locks. Usaremos o Node.js para construir nosso Serviço Coordenador, o "porteiro" rápido e eficiente do nosso sistema.

Os três desafios centrais

1. **Condição de corrida (exclusão mútua):** O que acontece se dois astrônomos, um em São Paulo e outro em Tóquio, conseguirem agendar o telescópio para o mesmo slot das 03:00 às 03:05 UTC? A garantia de que apenas um comando de agendamento pode ser processado por vez para um mesmo horário é a definição de exclusão mútua neste contexto.
2. **A Ilusão do tempo (sincronização de relógio):** Se o relógio do astrônomo de São Paulo está 2 segundos adiantado e o de Tóquio está 1 segundo atrasado, como o sistema pode determinar quem clicou em "reservar" primeiro? Um sistema distribuído sem uma noção de tempo unificada é caótico e não confiável.
3. **A memória do sistema (logging):** Se um agendamento for contestado, como podemos provar quem o fez e quando? Se o sistema falhar, como podemos rastrear a sequência de eventos que levou ao erro? Em nosso sistema, o logging servirá para:
    - Logging de aplicação (depuração): Registros de eventos de baixo nível que ajudam os desenvolvedores a entenderem o fluxo do código. Ex: "Conexão com o banco de dados estabelecida", "Requisição recebida na rota /agendamentos".
    - Logging de auditoria (negócio): Uma trilha de eventos importantes do ponto de vista do negócio. É um registro imutável de quem fez o quê e quando. Ex: "Cientista 'Marie Curie' (ID: 7) criou o

agendamento (ID: 123) para o horário '2025-12-01T03:00:00Z'". Isso é fundamental para resolver disputas.

Arquitetura e Tecnologias

- Serviço de Agendamento (API Principal):
  - Linguagem/Framework: Python 3.9+ com Flask.
  - Banco de Dados: SQLite (para simplicidade inicial) com SQLAlchemy.
  - Responsabilidade: Gerenciar a agenda, os cientistas e a persistência dos dados. É a fonte autoritativa do estado da agenda.
- Serviço de Travamento e Sincronia (Coordenador):
  - Linguagem/Framework: Node.js 18+ com Express.js.
  - Responsabilidade: Atuar como um coordenador centralizado para garantir a exclusão mútua (serviço de "lock") e, posteriormente, ajudar na disseminação de eventos em tempo real.

Preparação do Ambiente (Setup Inicial)

Antes da primeira entrega, vocês devem preparar suas máquinas.

1. Instalar as Ferramentas:
   - Instale Python 3.9+.
   - Instale Node.js 18+.
   - Instale Docker Desktop.
   - Um cliente de API como Postman ou Insomnia.
2. Estrutura de pastas do projeto (sugestão):

```
sctec-projeto/
├── servico-agendamento/  (Python/Flask)
│   ├── venv/
│   ├── app.py
│   └── requirements.txt
└── servico-coordenador/    (Node.js/Express)
    ├── node_modules/
    ├── server.js
    └── package.json
```

3. Configuração Inicial (Serviço de Agendamento):
   # Dentro da pasta /servico-agendamento
   python -m venv venv

```
source venv/bin/activate  # No Windows: venv\Scripts\activate
pip install Flask Flask-SQLAlchemy
#salvar a lista de todas as bibliotecas instaladas
pip freeze > requirements.txt
```

4. Configuração inicial (Serviço Coordenador):
```
# Dentro da pasta /servico-coordenador
npm init -y
npm install express
```

Como os serviços colaboram

Antes de iniciar, é crucial entender a "divisão de trabalho" em nossa arquitetura de microserviços. Teremos dois sistemas independentes que conversam entre si para atingir um objetivo comum. Pense neles como o Cérebro e o Porteiro de uma operação crítica.

1. **O Cérebro - Serviço de Agendamento (Python/Flask):**
    - Responsabilidade: Lidar com toda a lógica de negócio complexa. Ele sabe o que é um cientista, o que é um agendamento e como persistir essas informações no banco de dados. Ele é a fonte de verdade sobre o "estado" da agenda do telescópio.
    - Em Ação: Quando recebe um pedido para criar um agendamento, ele é responsável por validar os dados, verificar as regras de negócio e, finalmente, salvar o registro no banco de dados.
2. **O Porteiro - Serviço Coordenador (Node.js/Express):**
    - Responsabilidade: Executar uma única tarefa de forma extremamente rápida e eficiente: controlar o acesso. Ele não sabe nada sobre telescópios ou cientistas. Sua única função é dizer "pode passar" ou "espere, tem alguém na sua frente". Ele gerencia uma fila invisível para garantir que apenas uma operação crítica aconteça por vez.
    - Em Ação: Ele expõe uma API simples para "travar" (lock) um recurso e depois "destravar" (unlock). Sua natureza leve e orientada a eventos (Node.js) é perfeita para essa tarefa de alta concorrência.

O fluxo de uma requisição: Sincronização e Concorrência

Para que o agendamento seja justo, o sistema resolve dois desafios em duas etapas:

Etapa 1: Sincronização do relógio (garantindo um pedido justo)

Imagine que o relógio do computador de um cientista em São Paulo está 2 segundos adiantado.

1. Busca da Verdade: Antes de permitir o agendamento, o cliente (navegador) faz uma requisição GET /time ao Cérebro (Flask).
    1. [LOGGING no Cérebro (Flask):] Um Log de Aplicação é gerado, ex: "INFO: Requisição recebida em GET /time do IP 1.2.3.4".
2. Cálculo: O Cérebro responde com seu tempo oficial (a "fonte da verdade"). O cliente (usando o Algoritmo de Cristian) calcula e ajusta seu relógio local, compensando a latência da rede.
3. Resultado: Agora, o cliente sabe a hora exata. Quando o cientista clica para reservar o horário 03:00:00Z, o timestamp enviado na requisição é o correto (baseado no tempo do servidor), e não o horário adiantado da máquina local (03:00:02Z).

Etapa 2: O Fluxo de uma Requisição de Agendamento Concorrente (Resolvendo a Disputa)

Agora, imagine que duas requisições (de São Paulo e de Tóquio), ambas com o timestamp sincronizado e justo (ex: 03:00:00Z), chegam ao Cérebro (Flask) quase simultaneamente. O fluxo para resolver a disputa será o seguinte:

1. Chegada: O Cérebro (Flask) recebe a primeira requisição POST /agendamentos (com timestamp já sincronizado).
    - [LOGGING no Cérebro (Flask):] Um Log de Aplicação é gerado, ex: "INFO: Requisição POST /agendamentos recebida para o recurso Hubble-Acad_2025-12-01T03:00:00Z”.
2. Pedido de Permissão (Exclusão Mútua): Antes de tocar no banco de dados, o Cérebro pausa e faz uma chamada de rede para o Porteiro (Node.js), dizendo: "Quero acesso exclusivo ao recurso telescopio-1_2025-12-01T03:00:00Z. Posso?".
    - [LOGGING no Cérebro (Flask):] Um Log de Aplicação é gerado, ex: "INFO: Tentando adquirir lock para o recurso Hubble-Acad_2025-12-01T03:00:00Z ".
3. Permissão Concedida: O Porteiro, vendo que ninguém mais pediu acesso a esse recurso, responde "Sim, pode passar" (HTTP 200 OK) e anota que o recurso agora está travado.
    - [LOGGING no Porteiro (Node.js):] Um log é gerado, ex: "Recebido pedido de lock para recurso Hubble-Acad_2025-12-01T03:00:00Z".
    - O Porteiro vê que o recurso está livre e o trava.
    - [LOGGING no Porteiro (Node.js):] Um log é gerado, ex: "Lock concedido para recurso Hubble-Acad_2025-12-01T03:00:00Z".

- O Porteiro responde "HTTP 200 OK" para o Flask.

4. Ação Crítica: Com a permissão em mãos, o Cérebro (Flask) executa sua lógica crítica: verifica a disponibilidade no banco de dados e salva o novo agendamento.
    - [LOGGING no Cérebro (Flask):] Um Log de Aplicação é gerado, ex: "INFO: Lock adquirido com sucesso para o recurso Hubble-Acad_2025-12-01T03:00:00Z".
    - O Cérebro salva o agendamento no banco de dados.
    - [LOGGING no Cérebro (Flask):] Este é o mais importante: um Log de AUDITORIA (em JSON) é gerado para provar a ação de negócio, ex: {"level": "AUDIT", "event_type": "AGENDAMENTO_CRIADO", "agendamento_id": 123, ...}.

5. Chegada da segunda requisição: Enquanto o Cérebro está ocupado, a segunda requisição (também com timestamp justo) chega. Ela também pede permissão ao Porteiro para o mesmo recurso.
    - [LOGGING no Cérebro (Flask):] Um Log de Aplicação é gerado, ex: "INFO: Requisição POST /agendamentos recebida para o recurso Hubble-Acad_2025-12-01T03:00:00Z".
    - O Cérebro (Flask) chama o Porteiro (Node.js) pedindo um lock para o mesmo recurso.
    - [LOGGING no Cérebro (Flask):] Um Log de Aplicação é gerado, ex: "INFO: Tentando adquirir lock para o recurso Hubble-Acad_2025-12-01T03:00:00Z".

6. Permissão Negada: O Porteiro verifica sua lista e vê que o recurso já está travado. Ele responde imediatamente "Não, espere. Recurso ocupado" (HTTP 409 Conflict). O Cérebro (Flask), ao receber essa resposta, rejeita a segunda requisição.
    - [LOGGING no Porteiro (Node.js):] Um log é gerado, ex: "Recebido pedido de lock para recurso Hubble-Acad_2025-12-01T03:00:00Z".
    - O Porteiro vê que o recurso já está travado.
    - [LOGGING no Porteiro (Node.js):] Um log é gerado, ex: "Recurso Hubble-Acad_2025-12-01T03:00:00Z já em uso, negando lock.".
    - O Porteiro responde "HTTP 409 Conflict" para o Flask.

7. Rejeição: O Cérebro (Flask) recebe o "Conflict".
    - [LOGGING no Cérebro (Flask):] Um Log de Aplicação é gerado, ex: "INFO: Falha ao adquirir lock para o recurso Hubble-Acad_2025-12-01T03:00:00Z, recurso ocupado.".
    - O Cérebro retorna o erro 409 ao segundo cliente (de Tóquio).

8. Liberação: Após o Cérebro (Flask) terminar sua operação com sucesso, ele envia um último comando ao Porteiro: "Obrigado, já terminei. Pode liberar o recurso". O Porteiro então remove a trava.
    - [LOGGING no Cérebro (Flask):] Um Log de Aplicação é gerado, ex: "INFO: Liberando lock para o recurso Hubble-Acad_2025-12-01T03:00:00Z".
    - [LOGGING no Porteiro (Node.js):] Um log é gerado, ex: "Recebido pedido de unlock para recurso Hubble-Acad_2025-12-01T03:00:00Z. Lock liberado.".

Essa colaboração em duas etapas garante que a Sincronização de Relógio (Etapa 1) torne os pedidos justos, e a Exclusão Mútua (Etapa 2) escolha apenas um vencedor entre esses pedidos justos. Ainda registra tudo no Logging (Etapa 3)

Entregas do projeto em etapas

Etapa 1: O Blueprint da API

- Objetivo: Compreender que um bom software começa com um bom design. Aprender a pensar em uma API como um contrato e a projetar a observabilidade do sistema desde o início, definindo o formato dos logs de auditoria.
- Descrição: Nesta fase, vocês atuarão como arquitetos de software, focados puramente na especificação.
- Passo a passo:
    1. Crie um arquivo chamado MODELOS.md para definir as entidades do sistema.
    2. Crie um arquivo chamado API.md para descrever cada endpoint, incluindo HATEOAS.
    3. Design de Log: No arquivo API.md ou em um novo LOGGING.md, defina a estrutura dos logs de auditoria. Pense em quais informações são cruciais para um registro imutável. Exemplo:

```
{
  "timestamp_utc": "2025-10-26T18:00:05.123Z",
  "level": "AUDIT",
  "event_type": "AGENDAMENTO_CRIADO",
  "service": "servico-agendamento",
  "details": {
    "agendamento_id": 123,
    "cientista_id": 7,
    "horario_inicio_utc": "2025-12-01T03:00:00Z"
  }
}
```

4. Logs de aplicação (exemplo)
   INFO:2025-10-26T18:00:04.500Z:servico-agendamento:Requisição recebida para POST /agendamentos
   INFO:2025-10-26T18:00:04.505Z:servico-agendamento:Tentando adquirir lock para o recurso Hubble-Acad_2025-12-01T03:00:00Z
   INFO:2025-10-26T18:00:05.120Z:servico-agendamento:Lock adquirido com sucesso
   INFO:2025-10-26T18:00:05.122Z:servico-agendamento:Iniciando verificação de conflito no BD
   INFO:2025-10-26T18:00:05.123Z:servico-agendamento:Salvando novo agendamento no BD

   INFO:2025-10-26T18:00:04.800Z:servico-agendamento:Requisição recebida para POST /agendamentos
   INFO:2025-10-26T18:00:04.805Z:servico-agendamento:Tentando adquirir lock para o recurso Hubble-Acad_2025-12-01T03:00:00Z
   INFO:2025-10-26T18:00:05.121Z:servico-agendamento:Falha ao adquirir lock, recurso ocupado

Critérios de Avaliação:

- **Arquivo MODELOS.md completo.**
- **Arquivo API.md detalhando endpoints e respostas HATEOAS.**
- **Seção ou arquivo definindo o formato padrão para os logs do sistema.**

Entrega 2: O Sistema inicial (e a prova da falha nos logs)

- Objetivo: Traduzir a especificação em código funcional, implementar uma API RESTful básica em Flask e usar os logs de aplicação e auditoria para provar visualmente a existência de uma condição de corrida.
- Passo a passo:
  1. Implemente o "Cérebro" (Serviço de Agendamento) em Flask/SQLAlchemy de forma simples.
  2. Implementação de logging:
     - Configure o módulo logging padrão do Python para escrever em um arquivo (app.log) e no console.
     - Adicione logs de aplicação (nível INFO) em pontos-chave: "Requisição recebida para POST /agendamentos", "Iniciando verificação de conflito no BD", "Salvando novo agendamento no BD".

        - Após salvar com sucesso no banco, emita um log de auditoria (nível WARNING ou um nível customizado AUDIT) no formato JSON definido na Entrega 1.
    3. Implemente a rota POST /agendamentos e os links HATEOAS.
    4. Crie um script de teste de estresse que dispare 10 requisições simultâneas.
- Como Validar o Sucesso:
    1. Execute o script de teste de estresse.
    2. Consulte o banco de dados e veja os múltiplos registros conflitantes.
    3. Examine o arquivo app.log. Você deverá ver claramente os logs de aplicação de várias requisições se entrelaçando e, mais importante, múltiplos logs de auditoria para a criação de agendamentos no mesmo horário, uma prova irrefutável da falha de negócio.
- **Critérios de avaliação:**
    - **Código funcional do serviço Flask com HATEOAS.**
    - **Script de teste que prova a condição de corrida.**
    - **Arquivo de log gerado que mostra tanto os logs de aplicação entrelaçados quanto os logs de auditoria duplicados para o mesmo evento de negócio.**

<u>Entrega 3: O Coordenador (e seus próprios registros)</u>

- Objetivo: Entender a arquitetura de microserviços, a comunicação inter-serviços e implementar o padrão de Coordenador Centralizado.
- Passo a passo:
    1. No servico-coordenador, implemente o servidor Express com os endpoints POST /lock e POST /unlock.
    2. Logging no Coordenador: Adicione logs de console (console.log) no serviço Node.js para registrar eventos importantes: "Recebido pedido de lock para o recurso X", "Lock concedido para o recurso X", "Recurso X já está em uso, negando lock", "Lock para o recurso X liberado".
    3. No serviço Flask, modifique a rota POST /agendamentos para chamar o serviço de lock antes da operação de banco de dados, garantindo a liberação do lock em um bloco try...finally.
    4. Logging da Coordenação: No serviço Flask, adicione logs de aplicação que registrem a comunicação: "Tentando adquirir lock

para o recurso X", "Lock adquirido com sucesso", "Falha ao adquirir lock, recurso ocupado".

- Como validar o sucesso:
    1. Inicie ambos os servidores.
    2. Execute o mesmo script de teste de estresse.
    3. O resultado deve ser uma requisição 201 Created e nove 409 Conflict. O banco de dados deve ter apenas um registro.
    4. Observe os terminais: o terminal do Node.js deve mostrar o log de um lock sendo concedido e liberado, e nove pedidos sendo negados. O log do Flask (app.log) deve mostrar as tentativas, um sucesso e as nove falhas, e, crucialmente, apenas um log de auditoria.
- **Critérios de avaliação:**
    - **Código funcional de ambos os serviços.**
    - **O teste de estresse prova que a exclusão mútua foi implementada.**
    - **Os logs de ambos os serviços refletem a lógica de coordenação correta (um lock concedido, múltiplos negados).**

Entrega 4: A fonte da verdade e o cliente inteligente

- Objetivo: Compreender a sincronização de tempo cliente-servidor e o HATEOAS.
- Passo a passo:
    1. Crie o endpoint GET /time no Flask.
    2. Crie a interface web (index.html) com JavaScript que implementa algum algoritmo para sincronizar o tempo.
    3. Faça o cliente usar os links HATEOAS da resposta para habilitar e executar a ação de cancelamento.
    4. Logando a nova interação: Adicione um log de auditoria para o evento de cancelamento no endpoint correspondente do Flask. O log de aplicação também deve registrar as chamadas ao endpoint /time.
- **Critérios de avaliação:**
    - **Endpoint /time funcionando e cliente web implementado.**
    - **O cliente sincroniza o tempo e usa HATEOAS para o cancelamento.**

- **Ao cancelar um agendamento, um novo log de auditoria com event_type: "AGENDAMENTO_CANCELADO" deve aparecer no app.log.**

Entrega 5: Orquestração com Docker (e logs centralizados)

- Objetivo: Aprender os fundamentos de conteinerização com Docker e orquestração com Docker Compose.
- Passo a passo:
    1. Crie um Dockerfile para o servico-agendamento (Python/Flask).
    2. Crie um Dockerfile para o servico-coordenador (Node.js/Express).
    3. Na raiz do projeto, crie um docker-compose.yml para definir e conectar os dois serviços.
    4. Modifique a URL de comunicação no código Flask para usar o nome do serviço do Docker Compose (ex: http://servico-coordenador:3000).
- Como validar o sucesso:
    1. Na raiz do projeto, execute docker-compose up --build. Os dois contêineres devem iniciar.
    2. Validação de Logs Orquestrados: Em um novo terminal, execute docker-compose logs -f. Isso mostrará o fluxo de logs de ambos os contêineres em tempo real e de forma entrelaçada.
    3. Use o Postman ou a interface web para interagir com a aplicação. Observe como as requisições geram logs no serviço Flask, que por sua vez geram logs no serviço coordenador, tudo visível em um único fluxo. O sistema deve funcionar perfeitamente.
- **Critérios de avaliação:**
    - **Arquivos Dockerfile e docker-compose.yml funcionais.**
    - **A aplicação completa é executável com docker-compose up.**
    - **A execução de docker-compose logs demonstra com sucesso a agregação dos logs dos dois microserviços, provando que a observabilidade do sistema distribuído está funcionando.**

Exemplo de código de teste de estresse do sistema

```
import requests
import threading
import time

# A URL do Serviço de Agendamento (Flask)
# Certifique-se de que o Flask esteja rodando nesta porta.
URL_AGENDAMENTO = "http://127.0.0.1:5000/agendamentos"

# Número de requisições simultâneas
NUMERO_DE_REQUISICOES = 10

# Os dados EXATOS que todas as requisições tentarão criar.
# É crucial que seja o mesmo payload para forçar a condição de corrida.
PAYLOAD_CONFLITANTE = {
    "cientista_id": 1,
    "horario_inicio_utc": "2025-12-01T03:00:00Z"
    # Adicione outros campos necessários conforme seu MODELOS.md
}

def fazer_requisicao_agendamento(thread_num):
    """
    Função que será executada por cada thread.
    Ela envia uma única requisição POST para o endpoint de agendamento.
    """
    print(f"[Thread {thread_num}]: Iniciando requisição...")
    try:
        response = requests.post(URL_AGENDAMENTO, json=PAYLOAD_CONFLITANTE)
        print(f"[Thread {thread_num}]: Resposta recebida! "
              f"Status Code: {response.status_code}, "
              f"Body: {response.text[:100]}...")

    except requests.exceptions.ConnectionError as e:
        print(f"[Thread {thread_num}]: Erro de conexão. O servidor Flask está rodando? Erro: {e}")
    except Exception as e:
        print(f"[Thread {thread_num}]: Erro inesperado: {e}")


if __name__ == "__main__":
    print(f"Disparando {NUMERO_DE_REQUISICOES} requisições simultâneas para
{URL_AGENDAMENTO}")
    print(f"Payload: {PAYLOAD_CONFLITANTE}\n")
    threads = []
    start_time = time.time()
    for i in range(NUMERO_DE_REQUISICOES):
        t = threading.Thread(target=fazer_requisicao_agendamento, args=(i+1,))
        threads.append(t)
        t.start()

    for t in threads:
        t.join()

    end_time = time.time()
    print(f"Tempo total: {end_time - start_time:.2f} segundos")
```

Como rodar o teste?

1. Salve o código como teste_estresse.py (fora das pastas dos serviços).
2. Para a Entrega 2:
    - Rode seu serviço Flask.
    - Execute este script (python teste_estresse.py).
    - Resultado esperado: Você verá no console que várias threads (talvez todas) receberão Status Code: 201 (Created). Ao verificar seu banco de dados, você verá múltiplos registros conflitantes para o mesmo horário.
3. Para a Entrega 3:
    - Rode seu serviço Flask e seu serviço Node.js.
    - Execute este mesmo script.
    - Resultado esperado: Você verá no console que apenas uma thread receberá Status Code: 201 (Created), e todas as outras (9) receberão Status Code: 409 (Conflict). No banco, haverá apenas um registro.