Universidade Federal de Viçosa
Instituto de Ciência Exatas e Tecnológicas
CCF 355 - Sistemas Distribuídos e Paralelos

Trabalho Prático - Parte 5

Gemmarium

Grupo:
Henrique de Souza Santana Matrícula: 3051
Pedro Cardoso de Carvalho Mundim Matrícula: 3877

Professora:
Thais Regina de Moura Braga Silva

5 de agosto de 2022

1 Introdução

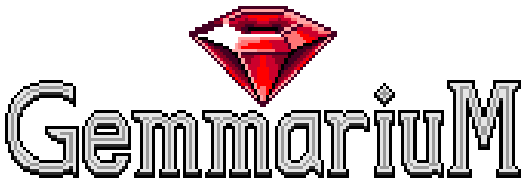

Figura 1: Logotipo do sistema.

Neste entrega do trabalho prático, adaptamos a implementação já existente do nosso sistema distribuído, o *Gemmarium* (Fig. 1), para um middleware de serviços Web, utilizando a biblioteca Flask [1], e seguindo o padrão REST na medida do possível. Na Seção 2 são mostradas as diferenças da implementação. Na Seção 3, apresentamos uma visão geral do sistema e sua arquitetura após uso de serviços. Por fim, na Seção 4 damos nossas considerações finais.

2 Desenvolvimento e Implementação

Como supracitado, nesta última etapa do trabalho adaptamos novamente as implementações para outro paradigma de comunicação, o de serviços Web. Dentre as ferramentas disponíveis, escolhemos trabalhar com a biblioteca Flask para elaborar os processos servidores, e a biblioteca *requests* para realizar as requisições. Essa escolha foi feita com base na familiaridade com a ferramenta, e na simplicidade.

2.1 Padrão REST

Utilizamos o Flask para implementar os processos servidores do sistema seguindo (até certo ponto) o padrão REST. Assim, a representação intermediária de dados volta a ser em JSON, porém com este middleware fazendo todo o trabalho de empacotamento.

O padrão REST sugere orientações específicas com relação ao formato das mensagens e a semântica por trás do uso dos métodos do HTTP (GET, POST, etc.) [2]. Serviços Web chamados RESTful respeitam essas orientações, geralmente especificando as interfaces com URLs que apontam para um objeto ou um conjunto deles, usando os métodos HTTP para especificar qual operação está sendo feita sobre esse(s) objeto(s) e usando o corpo das requisições POST ou PUT para descrever o novo estado.

No entanto, tendo em mãos um sistema cuja implementação já estava toda pronta sem o padrão REST em mente, optamos por não segui-lo a risca. No servidor da Forja e do Cliente, o padrão foi seguido, mas não no servidor do Cofre, como abordado em detalhes na Seção 2.2. Associamos rotas e métodos HTTP aos métodos das classes do sistema, como listado a seguir.

- `AuthEndpoint`:
  - `POST /signup` – realiza cadastro;
  - `GET /auth` – solicita token de autenticação;
  - `POST /auth` – informa chave secreta para obter token de autenticação.
- `ForgeEndpoint`:
  - `GET /gem` – solicita uma nova gema;
  - `POST /fusion` – envia dados de gemas para fusão com outro usuário.
- `TradeEndpoint`
  - `PUT /trade/<peerid>` – atualiza o estado de uma troca;

– `DELETE /trade/<peerid>` – rejeita uma troca;

– `POST /gems` – envia gemas para o outro usuário.

2.2 Implementação com Flask

Abordando agora detalhes específicos da implementação, em primeiro lugar, tivemos uma escolha quanto ao uso da biblioteca. Normalmente, ao usar Flask, são usados *decorators*, indicados pelo caractere `@` (Alg. 1), para associar as interfaces dos serviços às funções. No entanto, para reaproveitar a arquitetura do sistema, que usa de orientação a objetos e injeção de dependência, esses *decorators* foram utilizados diretamente para transformar funções que são passadas como parâmetro, alterando seus comportamentos, como mostra o Algoritmo 2. Esse código mostrado é o que de fato foi usado.

```
@app.route("/signup", methods=["POST"])
def signup():
    # ...
```

Algoritmo 1: Implementação comum em Flask.

```
    def __init__(self, ctrl: AuthCtrl):
        self.ctrl = ctrl
        self.app = Flask(__name__)
        self.app.route("/signup", methods=["POST"])(self.signup)
        self.app.route("/auth", methods=["GET", "POST"])(self.auth)
        self.session = {}
        self.__lock = threading.Lock()
```

Algoritmo 2: Construtor da classe `AuthEndpoint`, no servidor do Cofre.

As mensagens foram todas adaptadas, com exceção das de autenticação, pois estas dependiam do fluxo de solicitação, envio do número secreto e por fim obtenção do token de autenticação. Ou seja, cada lado manda duas mensagens (duas requisições-respostas). Para esta implementação, utilizamos a mesma URL, porém com um método GET e um POST. O GET é utilizado para obter o número secreto, enquanto que o POST é utilizado para mandar de volta o que número descriptografado, retornando enfim um token de autenticação. Essa troca não segue o padrão REST, mas foi a maneira mais simples que encontramos para realizá-la. A Figura 2 ilustra mais uma vez esse fluxo, ressaltando as URLs e métodos nas setas que indicam as mensagens trocadas.

Além disso, nas implementações com soquetes e gRPC, a conexão era aberta, a troca de mensagens era realizada de forma livre, encerrando em seguida a requisição. Já na implementação com Web services, foi necessário gerenciar explicitamente um registro de conexões ativas, armazenando qual é o número secreto que cada usuário deve responder. Caso a requisição POST não chegue, existe também um time out para retirar essa informação.

3 Resultados

Diferentemente das entregas anteriores, nesta não houve nenhuma mudança visualmente perceptível, portanto não há novas telas a serem exibidas. Além disso, o diagrama de classes se manteve praticamente inalterado, havendo mudanças sutis nas assinaturas dos métodos das classes `Endpoint`. Isso por si só já é um resultado notável, de que a separação em camadas lógicas para o projeto se provou muito útil. Para colocar em perspectiva, foram gastas cerca de

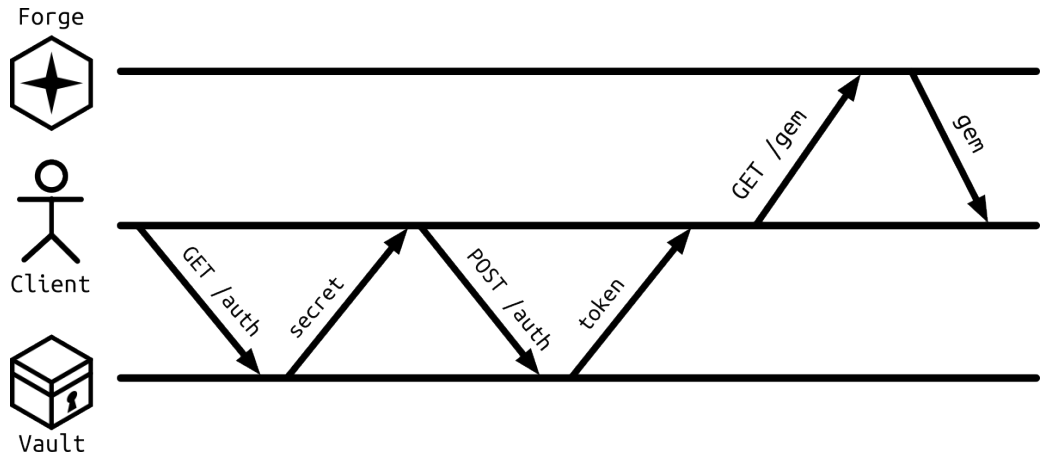

Figura 2: Diagrama de comunicação para solicitar gema.

4h para retirar as implementações do gRPC e inserir as do Flask. Isso foi influenciado também, claro, pelo conhecimento prévio com a biblioteca.

Uma outra comparação que podemos fazer entre as implementações é da sua complexidade. Podemos aproximar a complexidade do código pela quantidade de linhas gastas para implementá-lo. Observando apenas as classes da camada de comunicação, temos que a versão do trabalho com sockets gastou 602 linhas no total, com gRPC foram 532 linhas, e com Flask foram 496. Outras métricas poderiam ser avaliadas, mas essas já são indicativo da maior facilidade que conseguimos com o uso de middlewares.

4 Conclusão

Através deste trabalho, concluimos todo o trajeto do sistema com diferentes paradigmas. Traçando um paralelo com as partes anteriores do trabalho, podemos concluir que a utilização de middlewares foi de grande ajuda. Apesar disso, no caso específico de Web services, acreditamos que não fez tanto sentido tal implementação no contexto deste sistema pois existem partes que não precisavam ser muito desacopladas, especialmente por ser peer-to-peer. Mas no geral, foi interessante sua utilização.

Em relação às vantagens e desvantagens de cada implementação temos que com soquetes a vantagem está ligada à flexibilidade, visto que pudemos esolher todos os detalhes de implementação. Por exemplo, apenas com sockets que foi possível fazer o modelo de segurança completo. Os middlewares forneciam mecanismos de segurança, mas não pudemos reaproveitar nossa própria implementação. A desvantagem dos sockets está ligada à sua complexidade, visto que, como é possível escolher cada detalhe, também é necessário implementar cada um deles.

Quanto ao gRPC, sua vantagem está em retirar do programador a preocupação com os detalhes da comunição, enquanto que sua desvantagem está atrelada à curva de aprendizado, pois era uma ferramenta que não era conhecida por nós, contendo algumas sintaxes menos convenientes.

Por fim, em relação aos Web services, temos a vantagem de ser muito utilizado e já tínhamos familiaridade. A desvantagem foi em relação à limitação apenas por requisição-resposta, não tendo conexões que poderiam trocar mais de uma mensagem.

Em relação às questões de modularidade, em todas as implementações foi consideravelmente fácil alternar as implementações de comunicação, graças à divisão em camadas lógicas e toda a modelagem feita anteriormente.

Referências

[1] Flask, web development, one drop at a time. https://flask.palletsprojects.com/en/2.2.x/, 2022. Acesso em: 5 ago. 2022.

[2] Alex Rodriguez. Restful web services: The basics. *IBM developerWorks*, 33:18, 2008.