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

Trabalho Prático - Parte 4

Gemmarium

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

Professora:
Thais Regina de Moura Braga Silva

21 de julho 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), mudando da API de sockets para o middleware gRPC [3]. Documentamos então o impacto que o middleware teve no projeto, salientando as facilidades e dificuldades, além de relatar uma das funcionalidades planejadas do sistema que não foi entregue anteriormente. Na Seção 2 são mostradas as diferenças da implementação. Na Seção 3, apresentamos uma visão geral do sistema e sua arquitetura após uso do middleware, além das telas da funcionalidade adicionada. Por fim, na Seção 4 damos nossas considerações finais.

2 Desenvolvimento e Implementação

2.1 Adaptação ao gRPC

Em relação à adaptação ao gRPC, foi necessário retirar a implementação de sockets feita na parte anterior do trabalho e incluir certos componentes relacionados ao middleware, tais como os Protocol Buffers que serão mostrados a seguir. Além disso, agora é responsabilidade do middleware cuidar da representação externa de dados, empacotamento, dentre outros.

```
service Auth {
    rpc signup(SignupRequest) returns (SignupResponse) {}
    rpc auth(stream AuthRequest) returns (stream AuthResponse) {}
}

message SignupRequest {
    string username = 1;
    bytes key = 2;
}

message SignupResponse {
    string id = 1;
    string error = 2;
}

message AuthRequest {
    string id = 1;
    bytes secret = 2;
}

message AuthResponse {
    bytes secret = 1;
    bytes token = 2;
    string error = 3;
}
```

Algoritmo 1: Protocol Buffers do Cofre.

```
service Forge {
    rpc gem (GemRequest) returns (GemResponse) {}
    rpc fuse (FusionRequest) returns (FusionResponse) {}
}

message GemRequest {
    bytes token = 1;
}

message GemResponse {
    bytes gem = 1;
    string error = 2;
    uint32 wait = 3;
}

message FusionRequest {
    bytes token = 1;
    string peerid = 2;
    repeated bytes gems = 3;
}

message FusionResponse {
    repeated bytes gems = 1;
    string error = 3;
}
```

Algoritmo 2: Protocol Buffers da Forja.

```
service Trade {
    rpc trade (TradeEvent) returns (TradeResponse) {}
    rpc update (TradeEvent) returns (TradeResponse) {}
    rpc accept (TradeEvent) returns (TradeResponse) {}
    rpc reject (TradeEvent) returns (TradeResponse) {}
    rpc fuse (TradeEvent) returns (TradeResponse) {}
    rpc gems (TradeGems) returns (TradeResponse) {}
}

message TradeEvent {
    string peerid = 1;
    string peername = 2;
    uint32 port = 3;
    repeated string wanted = 4;
    repeated string offered = 5;
}

message TradeGems {
    string sender = 1;
    repeated bytes gems = 2;
}

message TradeResponse {
    bool ack = 1;
    string error = 2;
}
```

Algoritmo 3: Protocol Buffers do Cliente.

No que diz respeito aos Protocol Buffers, tem-se o que está representado nos Algoritmos 1, 2 e 3. O Algoritmo 1 mostra aqueles relacionados ao servidor do Cofre, o Algoritmo 2 mostra aqueles relacionados ao servidor da Forja e o Algoritmo 3 aqueles relacionados ao cliente.

Esses arquivos, reconhecidos pela extensão `.proto`, foram colocados numa mesma pasta no projeto e compilados usando a ferramenta `protoc`, gerando classes em Python que foram então copiadas para as respectivas pastas de cada processo que as utilizaria. Cada uma das partes marcadas como `service` são transformadas em duas classes: uma com o sufixo `Servicer`, que são herdadas pelas classes `Endpoints` para implementar o lado servidor, e outra com o sufixo `Stub`, a qual possibilita ao lado cliente instanciar e realizar a invocação remota de métodos. Em todos os métodos, assinalados com a palavra-chave `rpc`, há a passagem de apenas uma requisição, esperando receber apenas uma resposta, com exceção do método `auth` no Algoritmo 1. A palavra-chave `stream` tanto no parâmetro quanto no retorno indica um fluxo bidirecional de troca de mensagens, que foi necessário para esse método, como explicado na Seção 2.2.

Já as partes marcadas como `message` são compiladas em classes de mesmo nome, e substituem o formato criado na parte anterior do trabalho para o formato de mensagens trocadas no sistema. Cada `message` é uma estrutura que armazena um valor de cada um dos campos especificados, podendo estar vazios. Alguns campos foram marcados com `repeated` para que se pudesse trocar um vetor daquele tipo, como é o caso do `FusionRequest.gems` ou `TradeGems.gems`, que precisam guardar uma lista de gemas, representadas como bytes cada uma.

É importante salientar que houve uma exceção na adaptação ao gRPC. A funcionalidade de busca na rede local foi mantida com o uso de sockets, visto que utiliza broadcast com UDP, o que não era possível fazer com o gRPC.

2.2 Autenticação e Segurança

Pensando nos modelos de segurança do canal e dos processos modelados e implementados nas entregas anteriores, foram necessárias algumas alterações ao utilizar um middleware. O gRPC possui seu próprio mecanismo de segurança para a comunicação, de forma que as chaves públicas e privadas que usávamos anteriormente não possuem um formato compatível. Consideramos implementar esse recurso, porém não encontramos uma forma prática e programática de gerar e registrar as chaves e certificados de autenticação adequados ao gRPC. Assim, decidimos manter apenas o processo de autenticação já estabelecido, aproveitando parcialmente a implementação anterior, usando os algoritmos Curve25519 [1] e Ed25519 [2] da biblioteca PyNaCl.

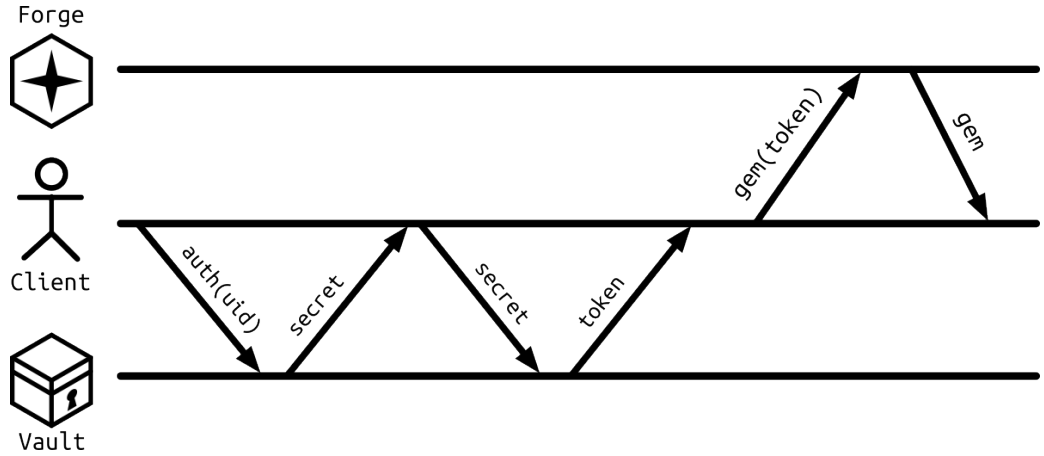


Figura 2: Diagrama de comunicação do processo de autenticação para requisição com a Forja.

Fizemos, na verdade, uma melhoria no processo de autenticação, de forma a deixar o servidor do Cofre inteiramente responsável por essa funcionalidade. O atual fluxo é apresentado na Figura 2, e nesse exemplo é mostrado logo antes de uma requisição com a Forja, que requer autenticação. Antes do Cliente efetivamente fazer uma requisição de seu interesse, ele deve

contatar o servidor do Cofre para obter um token de autenticação, que nada mais é que um objeto contendo o identificador do usuário, seu nome, e uma data de expiração do token. Definimos 60 segundos de validade para o token.

Como vemos no diagrama, temos a mesma troca de mensagens de um número aleatório entre cliente e servidor, que atesta sua identidade através da criptografia assimétrica. Ao final da troca, o token é gerado e assinado pela chave pública do Cofre, de forma que qualquer outro participante do sistema pode conferir se o token foi de fato gerado pelo Cofre.

O Cliente pode então fazer outras requisições com esse token, junto com qualquer outro dado necessário para as mesmas. Neste exemplo, o usuário quer solicitar uma nova gema. O servidor da Forja, ao receber o token, consegue obter o identificador do usuário e verificar se o token já expirou ou não. Se tiver expirado, o servidor se recusa a atender a requisição. Caso contrário, o mesmo prossegue como esperado.

2.3 Fluxo de troca

Com a substituição de sockets pelo middleware, o fluxo de troca das gemas entre os clientes foi alterado. Apesar de haver a opção de realizar o método com troca bidirecional nos protobuffers (`stream` tanto na parte de envio quanto na parte de receber), o que deixaria a implementação mais parecida com a anterior que utilizou sockets, optamos por não fazer isso, pois a sintaxe para realizar o fluxo bidirecional, em Python pelo menos, é pouco prática e confusa. Ou pior, no caso específico das trocas talvez não seria possível realizar a implementação de fato. Assim, ao invés de manter uma mesma conexão, apenas armazenamos o `TradeStub` ao longo de uma mesma troca, e cada requisição é independente uma da outra.

2.4 Geração de conteúdo

Antes mesmo de passar para o gRPC, foi terminada a implementação da funcionalidade de fusão das gemas trocadas. A ideia é que essa seja uma forma alternativa de obter gemas, que só é alcançada se dois usuários estiverem realizando uma troca, incentivando essa interação.

Normalmente as gemas são obtidas de forma aleatória, sendo sorteadas pelo servidor da Forja. O servidor obtém as informações de todas as gemas do arquivo `forge/res/gems.json`, sendo que algumas delas só podem ser obtidas por meio do mecanismo de fusão, marcadas pelo atributo `"rarity"` com valor zero. As gemas que podem ser obtidas por fusão possuem uma lista de "materiais". Por exemplo, a gema chamada granada é obtida pela fusão do rubi e da safira (Fig. 3).

A princípio as fusões podem parecer arbitrárias, mas na verdade tanto a escolha das gemas disponíveis quanto a lista de materiais de cada fusão é inspirada na série de desenho animado Steven Universe, criado pela animadora, compositora e ativista Rebecca Sugar. Nessa série, cada gema é uma personagem, e elas possuem a habilidade de se fundirem em personagens mais fortes. Assim, cada gema de fusão no nosso sistema corresponde a uma gema que aparece na série. A lista completa das gemas e das fusões disponíveis pode ser lida no arquivo JSON supracitado, mas para os usuários do sistema, a ideia seria descobrir por tentativa e erro quais gemas geram fusões ou não.

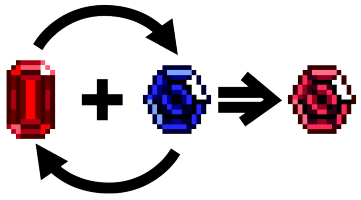


Figura 3: Exemplo de fusão, com o rubi (esquerda) e a safira (meio) gerando a granada (direita).

Para implementar essa funcionalidade, usamos um esquema parecido com a aceitação de uma troca, como mostra a Figura 4. Durante uma troca, ao invés de clicar no botão "aceitar", um usuário pode clicar no botão "fusão", tornando visível ao seu par que ele deseja tentar uma fusão usando as gemas que estão atualmente sendo ofertadas. O outro usuário pode fazer o mesmo, de forma que ambos vão mandar para a Forja uma requisição de fusão, a qual se torna, a partir desse ponto, a mediadora da troca.

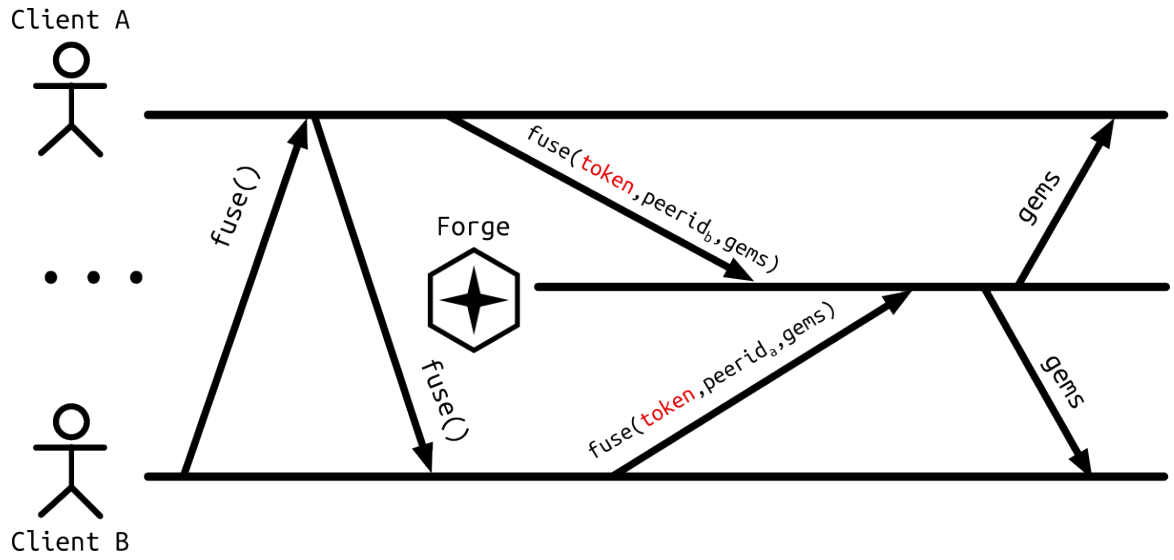


Figura 4: Diagrama de comunicação do processo de fusão.

Por simplicidade, estamos omitindo as mensagens de confirmação que sempre são trocadas entre os clientes. Como podemos ver, destacado de vermelho, ambos os usuários também já devem estar com seus tokens de autenticação, obtido através de uma comunicação prévia com o servidor do Cofre, também omitida do diagrama.

Cada requisição de fusão é atendida por threads diferentes no servidor da Forja, através do método `fuse` (Alg. 4), e deve haver uma colaboração entre elas, o que gera desafios. Em primeiro lugar, para que as threads saibam a qual par de usuários a fusão se refere, cada requisição deve informar o identificador do outro usuário. Em seguida, não há garantia de ordem de qual requisição será atendida primeiro, então ambas as threads devem atualizar o estado da fusão (linha 7) e esperar pela outra thread fazer o mesmo (Alg 5, linha 6).

```
1  def fuse(self, request, context):
2      uid, username = self.ctrl.auth(request.token)
3      if not uid:
4          return FusionResponse(error="AuthError")
5      peerid, gems = request.peerid, [g for g in request.gems]
6      try:
7          self.ctrl.update_fusion_request(uid, peerid, uid, username, gems)
8      except:
9          self.ctrl.remove_fusion_request(uid, peerid)
10         return FusionResponse(error="InvalidGems")
```

Algoritmo 4: Trecho inicial do método `fuse` do `ForgeEndpoint`.

Quando ambos os "lados"da fusão são preenchidos com as gemas dos usuários, as threads são liberadas para terminar de atender as requisições. Pode haver o caso em que há uma falha em algum dos clientes e sua requisição nunca chegue ao servidor, portanto essa espera tem uma janela de timeout, que estabelecemos arbitrariamente como 60 segundos (Alg. 5, linha 4).

A última thread que atender uma requisição fica responsável por sortear uma possível fusão entre as gemas oferecidas por ambos os usuários, calcular quais serão as gemas usadas para criá-la, e registrar o resultado da fusão numa estrutura de dados compartilhada. Poderia haver nesse processo uma condição de corrida, mas usamos o mecanismo de *lock* para evitá-la, impedindo que mais de uma thread esteja na mesma região de código que altera a estrutura, no método `update_fusion_request` do `ForgeCtrl`.

Quando enfim todos esses problemas são resolvidos, a Forja responde cada requisição com

```
    t0, ok = time(), False
    try:
        while time() - t0 <= 60:
            req = self.ctrl.get_fusion_request(uid, peerid)
            if req.is_complete() and req.has_fusion_set():
                ok = True
                break
    except:
        pass
    if not ok:
        return FusionResponse(error="Timeout")
```

Algoritmo 5: Trecho intermediário do método `fuse` do `ForgeEndpoint`.

```
    req = self.ctrl.get_fusion_request(uid, peerid)
    fused, others = req.fusion
    gems = [fused, *others[uid]] if fused else others[uid]
    self.ctrl.remove_fusion_request(uid, peerid, uid)
    return FusionResponse(gems=gems)
```

Algoritmo 6: Trecho final do método `fuse` do `ForgeEndpoint`.

as gemas trocadas (Alg. 6). Se não houver uma fusão disponível, o servidor apenas completa a troca, sem alterar as gemas. Se houver uma fusão, no entanto, o servidor consome as gemas usadas como material, envia uma cópia da fusão para cada usuário e envia também quaisquer gemas que não foram utilizadas no processo, caso houver. A Forja sempre prioriza consumir a menor quantidade quanto possível de gemas, além de garantir que vai gastar pelo menos uma gema de cada usuário, mas buscando equilibrar essa quantidade entre os dois lados. A remoção dessa estrutura compartilhada também segue o mesmo cuidado com o mecanismo de trava, de forma que nenhuma thread apaga o resultado da fusão antes que a outra thread possa enviá-la ao seu respectivo cliente.

3 Resultados

Para o usuário final do sistema, as mudanças implementadas nesta etapa do trabalho referentes ao middleware são, em geral, imperceptíveis. No entanto, para a experiência do programador, e supondo uma futura manutenção do sistema, usar o gRPC trouxe resultados muito positivos. Todos os métodos e classes dedicados apenas para ajustar os detalhes da comunicação em rede e fazê-la acontecer corretamente foram dispensados, incluindo questões como cálculo de buffer de recebimento de mensagens, empacotamento e desempacotamento, fechamento de conexões, criação de threads, entre outros. Tudo isso é resolvido pelo middleware.

A Figura 5 mostra um exemplo dessa simplificação, através do diagrama de classes do processo do Cofre. As classes `Database` e `User` se mantiveram totalmente inalteradas, a classe `AuthCtrl` ganhou novos métodos para lidar com o novo processo de autenticação, e a classe que mais mudou de fato foi a `AuthEndpoint`. Anteriormente, essa classe herdava de uma classe abstrata comum aos dois servidores, cujos métodos eram apenas voltados para tratar das questões dos sockets. Isso se tornou desnecessário, e agora `AuthEndpoint` herda de uma classe (omitida no diagrama) gerada pelo gRPC através dos Protocol Buffers.

Uma situação semelhante ocorreu com as classes da `Forja`, na qual a classe `ForgeEndpoint` foi reduzida drasticamente. A `ForgeCtrl` continuou praticamente a mesma, porém agora os métodos referentes à funcionalidade de fusão funcionam corretamente. Vale ressaltar que esses métodos ocupam cerca de metade do espaço do diagrama e do arquivo fonte, mostrando a

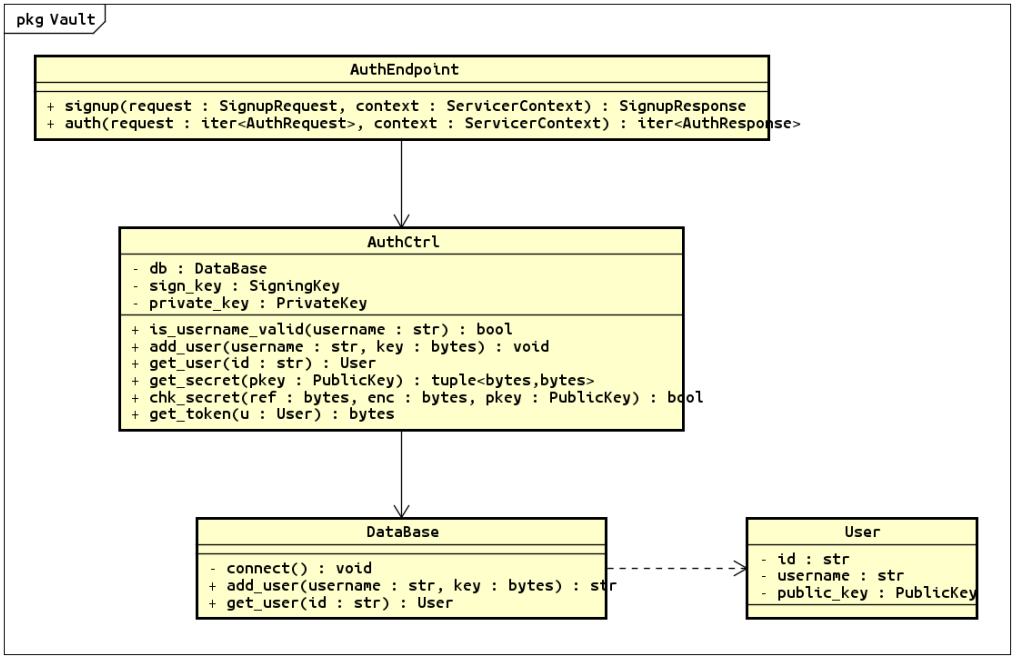


Figura 5: Diagrama de classes do Cofre após uso do gRPC.

complexidade de toda a lógica de escolher corretamente as fusões e as gemas usadas de material em cada uma.

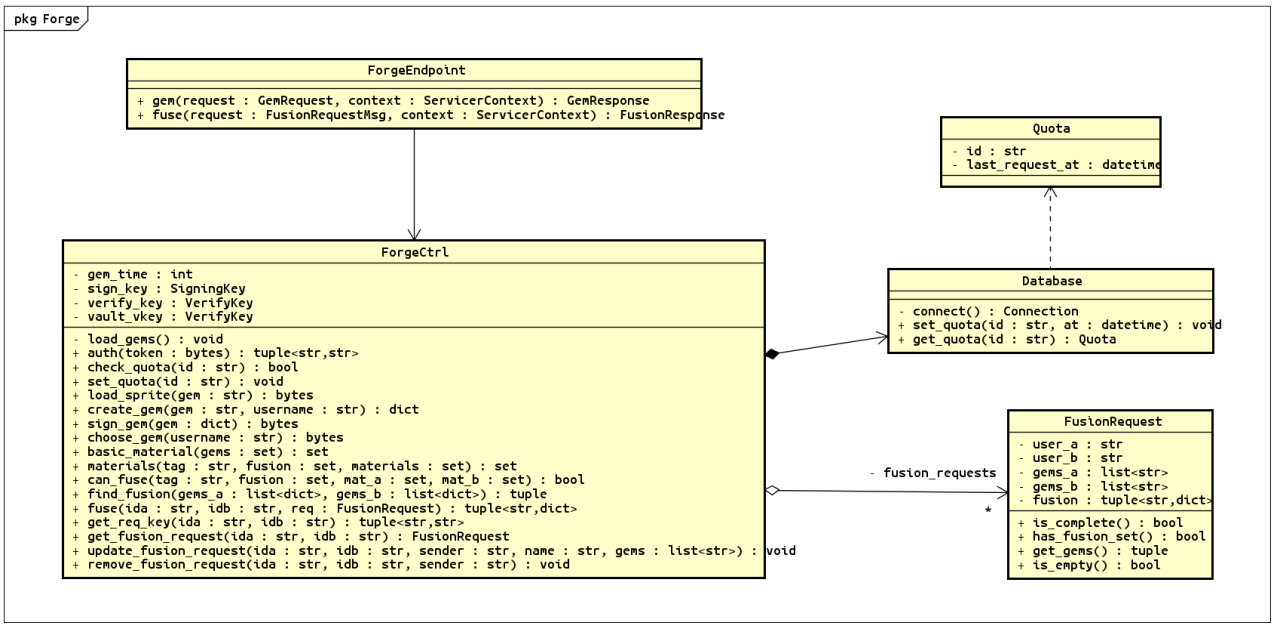


Figura 6: Diagrama de classes da Forja após uso do gRPC.

Por último nos diagramas, temos o mais complexo de todos, do processo Cliente (Fig. 7). Apesar de grande, também sofreu poucas alterações. Primeiramente, por causa da alteração do fluxo de autenticação, optamos por deixar o `ProfileEndpoint` responsável por adquirir o token de autenticação, tanto por uma questão de coesão das responsabilidades da classe, quanto pelo fator discutido na Seção 2.3, da sintaxe pouco prática de lidar com fluxo bidirecional de mensagens do gRPC em Python. Isso acabou criando uma dependência para as classes `CollectionEndpoint` e `TradeEndpoint` em relação ao `ProfileEndpoint`, mas valeu a pena.

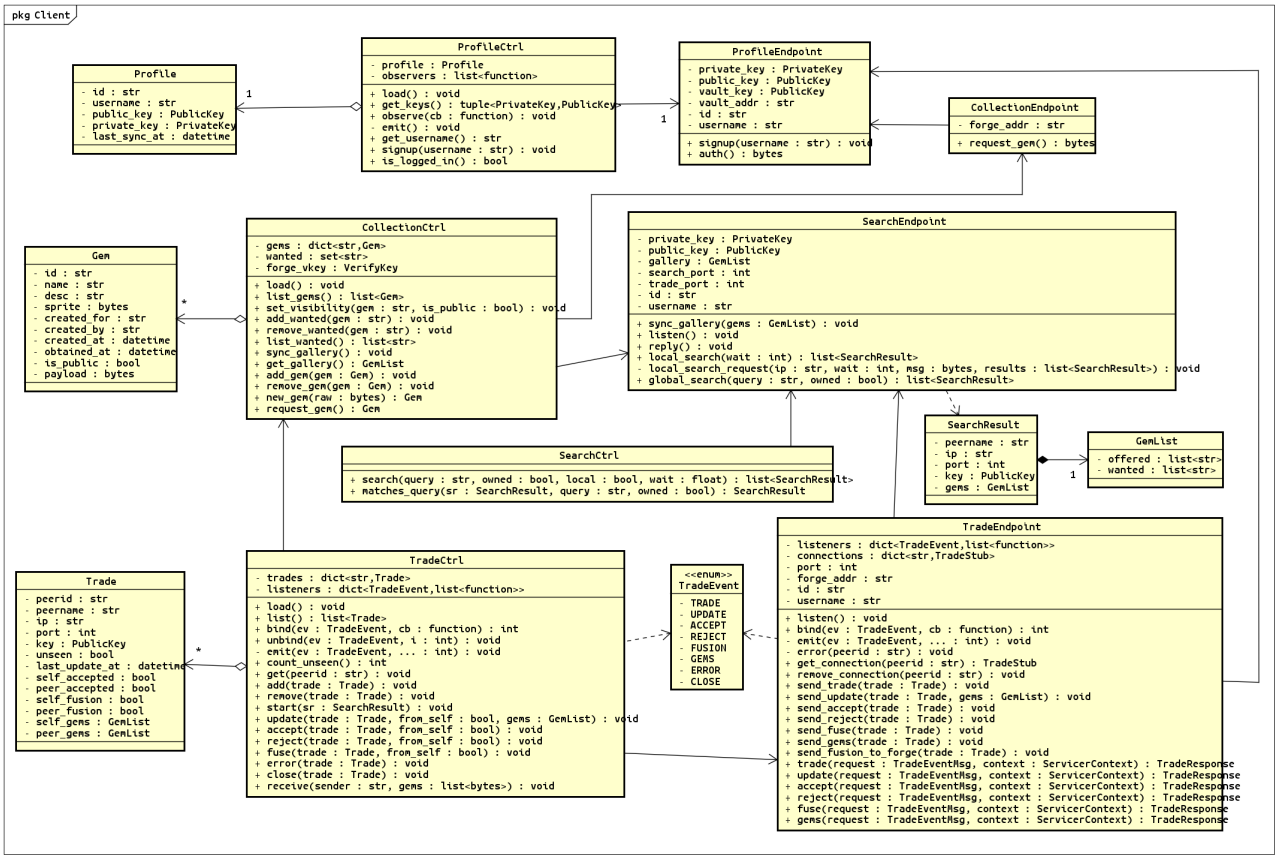


Figura 7: Diagrama de classes do Cliente após uso do gRPC.

Outra mudança notável é que agora a classe `TradeEndpoint` está na verdade maior do que antes. Isso se deve ao fato de que anteriormente, um único método recebia e confirmava todas as solicitações do processo de troca, disparando as callbacks correspondentes da camada lógica superior (das classes controladoras). Agora, com o gRPC, tornou-se obrigatório que cada método definido no Protocol Buffer se tornasse um método da classe. Apesar de não haver mais a noção explícita de conexão, como também discutido na Seção 2.3, mantivemos o atributo `connections`, mas dessa vez para armazenar os *stubs*, para que não fosse necessário instanciá-los toda vez que qualquer alteração mínima do estado da troca fosse executada.

Considerando que praticamente nada foi necessário alterar das classes além das `Endpoint`s em cada processo, percebemos que o uso de camadas lógicas para desacoplar cada responsabilidade do sistema facilitou em muito a incorporação do gRPC.

Por fim, mostramos o mecanismo de fusão sendo usado na prática. Supondo que temos dois usuários que já se encontraram na rede local através da funcionalidade de busca e já estão negociando uma troca (Figura 8). Em cada par de capturas de tela, vamos chamar o lado de esquerdo de Cliente A, e o lado direito de Cliente B.

Se o Cliente B clica no botão mais à direita da barra superior, o botão de fusão, sua tela fica marcada de roxo (Figura 9a), e seu par da troca visualiza essa mudança, sabendo que há a intenção de tentar uma fusão (Figura 9b). Como podemos ver, Cliente A está ofertando uma ametista, enquanto Cliente B está ofertando uma pérola, portanto essas duas gemas é que serão usadas na fusão, se houver.

Como já mostrado na entrega anterior, qualquer um dos usuários poderia rejeitar a troca e todo o processo seria cancelado. Se, por outro lado, o usuário do Cliente A estiver satisfeito com as condições da troca e achar que vale a pena tentar a fusão entre as gemas ofertadas por ambos, ele pode também clicar no botão de fusão, tornando as telas de ambos marcadas de roxo até que o servidor da Forja responda com o resultado. Neste caso, existia sim uma fusão

(a) Cliente A durante uma troca.

(b) Cliente B durante uma troca.

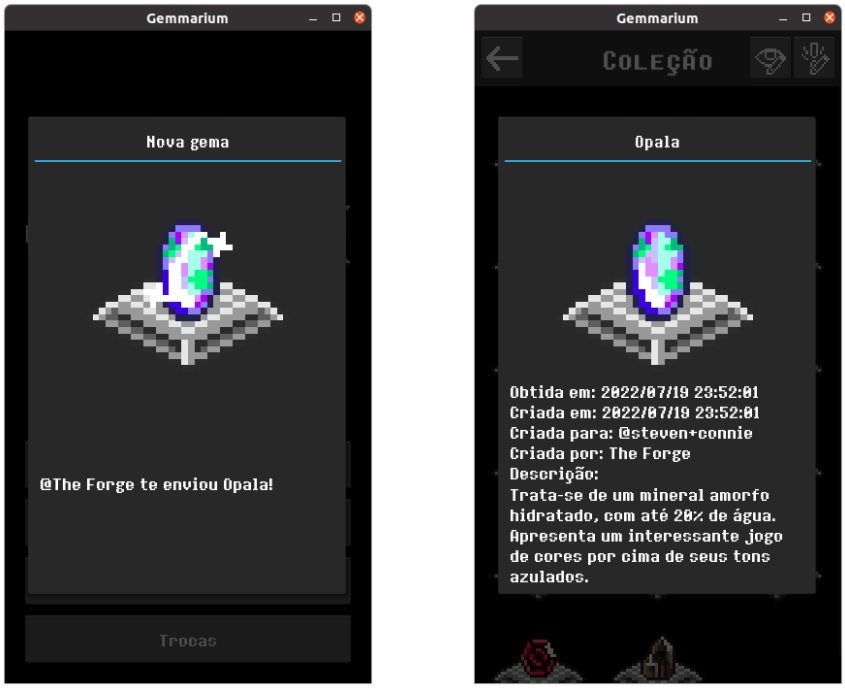
Figura 8: Tela de ambos os processos, Cliente A e Cliente B, no meio de uma troca.

(a) Cliente A visualiza a intenção de fusão de seu par.

(b) Cliente B pede para tentarem uma fusão.

Figura 9: Início de uma tentativa de fusão.

entre essas gemas, resultando na opala, mostrada na Figura 10a. Aqui foi mostrada apenas uma tela, pois ambos os processos vão ver exatamente a mesma informação, visto que a gema resultante da fusão é copiada para ambos os usuários participantes da troca.



(a) Resultado da fusão, idêntica para ambos os processos.

(b) Visualização da gema obtida por fusão.

Figura 10: Gema obtida pela fusão.

Um detalhe interessante é que o fato da gema ter sido obtida por fusão fica marcada em seus metadados. Ao abrir a coleção e visualizar a gema, o usuário pode ver que os nomes de usuário dos envolvidos na troca fica registrado. Neste exemplo, o usuário no Cliente A era `steven` e o usuário em B era `connie`, como mostra a Figura 10b, ao observar o campo "Criado para". A ordem dos nomes é determinada pela ordem dos identificadores dos usuários, apenas coincidiu com a ordem de apresentação das capturas de tela.

4 Conclusão

Através deste trabalho, tivemos a oportunidade de experimentar na prática alguns conceitos importantes apresentados tanto na disciplina. Em primeiro lugar, não conhecíamos bem o paradigma de chamada de procedimento remoto (RPC) e de invocação de método remoto (RMI). Foi interessante conhecer e usar essa tecnologia, observando que, de fato, ela cumpre o que propõe, permitindo uma sintaxe muito parecida com a de invocação de métodos locais, sem preocupações com detalhes da comunicação em rede.

Além disso, a modelagem inicial do sistema, antes da implementação, auxilia muito a manutenção do sistema, visto que foi consideravelmente fácil trocar a tecnologia subjacente à camada lógica de comunicação, modelada na segunda entrega do trabalho e implementada na terceira. Ao mesmo tempo, as camadas superiores permaneceram praticamente inalteradas.

Por fim, houve também algumas dificuldades específicas do trabalho apresentado, tanto pelo fato de haver comunicação P2P, quanto pela escolha da linguagem. A comunicação par-a-par exigiu que criássemos os *protobufs* também para o processo Cliente, visto que ele atua também como servidor durante as trocas de itens. Já a escolha da linguagem tornou pouco prático o uso de um dos recursos específicos do gRPC, de fluxo bidirecional de mensagens, porém, felizmente, esse ponto não se tornou um grande problema.

Referências

[1] Daniel J Bernstein. Curve25519: new diffie-hellman speed records. In *International Workshop on Public Key Cryptography*, pages 207–228. Springer, 2006.

[2] Jacqueline Brendel, Cas Cremers, Dennis Jackson, and Mang Zhao. The provable security of ed25519: theory and practice. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1659–1676. IEEE, 2021.

[3] gRPC. gRPC: A high performance, open source universal RPC framework, 2022.