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

Trabalho - Parte 03

Gemmarium

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

Professora:
Thais Regina de Moura Braga Silva

2 de julho de 2022

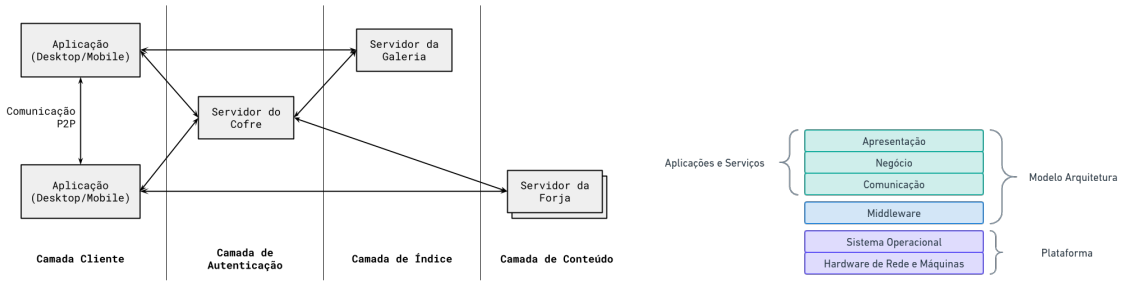
1 Introdução

Figura 1: Logotipo do sistema.

Nas três partes finais do trabalho prático, será desenvolvido o sistema distribuído em questão, que foi modelado anteriormente, três vezes. Assim, tivemos uma experiência de desenvolvimento primeiramente sem e com middleware. Na primeira parte, apresentada neste relatório, o sistema entitulado *Gemmarium*, foi desenvolvido utilizando apenas a API de sockets, sendo necessário configurar o uso da visão dos processos nos elementos arquitetônicos. Na Seção 2 são mostradas as principais decisões de implementação do projeto e modelagem. Na Seção 3, apresentamos os resultados obtidos com a implementação a partir das telas da interface gráfica. Por fim, na Seção 4 damos nossas considerações finais.

2 Desenvolvimento e Implementação

2.1 Arquitetura



(a) Camadas físicas.

(b) Camadas lógicas

Figura 2: Camadas da arquitetura.

Como estabelecido nas primeiras entregas, o sistema projetado seria dividido em camadas físicas e lógicas. A Figura 2a retoma essa divisão, porém após a implementação de fato, tendo identificado o desafio envolvido para a realizar trocas par-a-par na Internet, deixamos de lado a camada física da Galeria (Camada de Índice). As camadas do Cofre (Camada de Autenticação), da Forja (Camada de Conteúdo) e dos Clientes foram mantidas e realmente implementadas como processos separados.

A divisão em camadas lógicas (Fig. 2b) foi mantida como planejado: a camada de comunicação é responsável por lidar com todos os trâmites dos soquetes e da rede, a camada de negócio lida com a lógica do sistema, e a camada de apresentação mostra os dados para o usuário e permite interação. Esta última camada só se faz presente de fato nos processos Cliente, visto que nos servidores a única "apresentação"que ocorre são *prints* na saída padrão para ajudar a monitorar o fluxo de mensagens.

2.2 Ferramentas

Para o desenvolvimento do sistema, foi utilizada a linguagem Python em todos os processos, dada a maior familiaridade da dupla. Para implementar a interface gráfica do processo cliente,

foi escolhida a biblioteca Kivy [Kivy, 2022]. Como planejado, e como será mostrado em detalhes na Seção 2.4, um esquema de criptografia assimétrica foi utilizado, e para facilitar sua implementação, usamos a biblioteca PyNaCl [Authority, 2022] (pronuncia-se PySalt).

Por fim, para outras tarefas como multithreading, representação externa de dados, persistência de dados, e identificação de objetos, foram usadas as APIs padrões da linguagem Python de threads, JSON, SQLite3 e UUIDs, respectivamente. Como especificado, nesta entrega não usamos nenhum middleware para comunicação, portanto também a API padrão de sockets foi usada.

2.3 Representação externa de dados

Com o objetivo de padronizar a comunicação no sistema, foi utilizado JSON para empacotar e desempacotar os dados das mensagens, com um esquema específico. Toda mensagem foi estruturada como uma tupla de dois elementos, uma string que identifica o tipo de operação que a mensagem representa, e um dicionário contendo argumentos nomeados para as operações.

2.4 Segurança

Em relação ao modelo de segurança, utilizamos da criptografia para proteger os canais e processos. As mensagens que trafegam pela rede são criptografadas e através das chaves públicas conseguimos identificar com quem os processos estão conversando. Tendo em vista o funcionamento par-a-par do sistema e sabendo que os itens de interesse para realizar as trocas são digitais, também consideramos ser interessante garantir a autenticidade do conteúdo e de forma que qualquer Cliente possa fazer isso de forma autônoma.

A segurança da comunicação foi feita com o algoritmo Curve25519 [Bernstein, 2006], que gera pares de chaves públicas e privadas de exatos 32 bytes cada. Sempre que um processo precisa mandar uma mensagem para outro, o remetente usa sua chave privada e a chave pública de seu destinatário para criptografar a mensagem de forma que apenas ele possa descriptografá-la, e confirmando sua identidade. Quando o remetente não espera que seu destinatário o conheça (por exemplo, quando um usuário solicita cadastro), ele também envia sua própria chave pública logo antes dos bytes da mensagem em si, para que o destinatário possa ler suas mensagens e decidir se vai confiar naquela chave ou não.

Os servidores já são configurados para conhecerem entre si suas chaves públicas e usá-las na comunicação. Já os clientes, quando necessário se autenticar, precisam estar cadastrados no sistema com sua chave pública, e são identificados como já descrito na primeira entrega deste trabalho. Na Seção 2.6 o processo de autenticação é ilustrado.

No caso da autenticidade do conteúdo, foi usado o algoritmo Ed25519 [Brendel et al., 2021], que tem um formato diferente de chaves, representadas na biblioteca PyNaCl pelas classes `SigningKey` (privada) e `VerifyKey` (pública). Considerando que todas as gemas são criadas inicialmente pela Forja, basta que o servidor assine as gemas com sua chave privada, e que os clientes usem a chave pública bem conhecida da Forja para verificar as gemas, sem ser necessário trocar mais mensagens. Se houver algum erro na verificação, a gema não é autêntica.

2.5 Trocas de Mensagens

Modelamos as mensagens de acordo com o padrão estabelecido na Seção 2.3, listando as operações esperadas de acordo com o esquema a seguir. As letras C, V e F identificam o Cliente, o Cofre (Vault) e a Forja, respectivamente. Cada troca de mensagens é especificada da forma $[X \rightarrow Y](op, \{arg_1 : type_1, arg_2 : type_2\})$ que indica que espera-se que a entidade X envie a mensagem para Z, e a mensagem é identificada pela string op, contendo 0 ou mais argumentos, identificados por arg_i de tipos $type_i$1. Se uma seta $\xrightarrow{*}$ for usada, isso indica um broadcast. Um $*$ no nome da entidade indica que é uma mensagem que pode ser trocada de ou

para uma entidade qualquer. Um exemplo de valor para argumento pode ser mostrado também da seguinte forma: $[X \to Y](op, \{arg : str = "exemplo"\})$.

1. $[* \to *]$ `(error, {code:str})`: indica algum erro qualquer.
2. $[C \to V]$ `(signup, {username:str})`: solicita cadastro.
3. $[V \to C]$ `(ack, {id:str})`: confirma o cadastro, informando o ID único do usuário.
4. $[C \to F]$ `(request, {id:str})`: solicita uma nova gema.
5. $[F \to V]$ `(auth, {id:str})`: solicita dados de usuário para autenticação.
6. $[V \to F]$ `(user, {name:str, key:str})`: informa dados de usuário para autenticação.
7. $[F \to C]$ `(auth, {secret:int})`: envia um número aleatório usado no teste de autenticação.
8. $[C \to F]$ `(auth, {secret:int})`: retorna o mesmo número para confirmar identidade.
9. $[F \to C]$ `(gem, {gem:str})`: entrega dados de uma gema.
10. $[F \to C]$ `(error, {code:str="AuthError"})`: informa erro de autenticação.
11. $[F \to C]$ `(error, {code:str="QuotaExceeded", wait:int})`: informa quanto tempo em segundos o usuário tem que esperar para solicitar mais gemas.
12. $[C \xrightarrow{*} C]$ `(search, {})`: realiza descoberta na rede local.
13. $[C \to C]$ `(gallery, {id:str, username:str, port:int, wanted:list, offered:list})`: informa o nome das gemas que busca e nas quais possui interesse, e em qual porta espera receber trocas.
14. $[C \to C]$ `(trade, {peerid:str, peername:str, port:int})`: inicia um processo de troca.
15. $[C \to C]$ `(update, {wanted:list[str], offered:list[str]})`: atualiza o estado de uma troca.
16. $[C \to C]$ `(accept, {})`: aceita uma troca.
17. $[C \to C]$ `(reject, {})`: rejeita uma troca.
18. $[C \to C]$ `(fusion, {})`: indica que pretende tentar fusão.
19. $[C \to C]$ `(gems, {gems:list[str]})`: envia dados de gemas.
20. $[C \to C]$ `(ack, {})`: confirma alguma ação.

Como especificado, foi usado o TCP para todas as trocas de mensagens, com exceção das mensagens 12 e 13. Isso se deve ao fato de que os clientes se descobrem mutuamente na rede através para fazer broadcast é preciso usar o UDP, sem a noção de conexão que o TCP fornece.

Além disso, nas mensagens de tipo `gems`, como várias gemas podem ser passadas, é necessário informar, antes, o tamanho da mensagem, para que o destinatário da mensagem saiba o tamanho de buffer que deve ser usado. Essa informação vem logo depois dos bytes da chave pública e antes da mensagem em si.

Por fim, vale ressaltar que em todas as trocas de mensagens com os servidores, uma conexão atende apenas à uma requisição (que pode envolver mais de uma mensagem). Já o processo de troca entre os pares se dá através de duas conexões que se mantém abertas durante toda a troca. Apesar de uma única conexão TCP já ser bidirecional, essa escolha foi adotada para facilitar o fluxo de informações, de jeito que numa conexão apenas um cliente envia mensagens que vão atualizar o estado atual da troca, e o outro cliente apenas confirma o recebimento das mensagens, tendo também sua conexão própria para enviar dados.

2.6 Principais Implementações

Sendo o foco do trabalho a comunicação entre processos, alguns trechos de código são mostrados aqui, referentes à forma que as entidades do sistema se comunicam, além dos diagramas de classes do sistema como um todo.

O Algoritmo 1 mostra o método `listen` da classe `Server`, da qual ambos os servidores herdam, de forma que a implementação base da comunicação é compartilhada pelo Cofre e pela Forja. Como podemos ver, a cada nova conexão, é criada uma thread para atendê-la, e o servidor já volta a escutar por novas conexões. No caso das comunicações com os servidores, cada conexão atende apenas a uma requisição.

```
    def listen(self):
        try:
            with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
                s.bind(self.addr)
                s.listen()
                while True:
                    print("Listening...")
                    conn, addr = s.accept()
                    keep = addr in self.__trusted_hosts
                    worker = threading.Thread(
                        target=self.handle,
                        args=(conn, addr, keep)
                    )
                    worker.start()
        finally:
            s.close()
            self.close()
```

Algoritmo 1: Método `listen` da classe `Server`.

Falando especificamente de cada processo, primeiramente, o mais simples é o do Cofre, que apenas armazena a identidade dos usuários e suas chaves públicas. A Figura 4 mostra seu diagrama de classes. Um padrão adotado em todo o sistema é que as classes de sufixo `Ctrl` são da camada de negócio, e as de sufixo `Endpoint` são da camada de comunicação. Aqui vemos também os diversos métodos reaproveitados em ambos os servidores, com tarefas repetitivas como criptografar (`enc_msg`) e descriptografar (`dec_msg`) as mensagens, calcular e enviar tamanho de uma mensagem (`send_size`), entre outros.

Um exemplo de como o cliente interage com o servidor do Cofre é mostrado na Figura 3, na qual o cliente apenas envia seu nome e sua chave e o servidor confirma seu cadastro se tudo estiver correto.

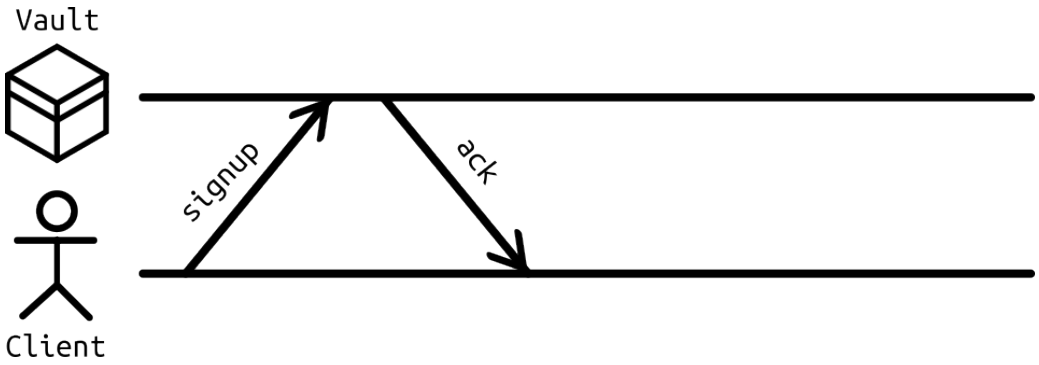


Figura 3: Diagrama de comunicação do processo de cadastro.

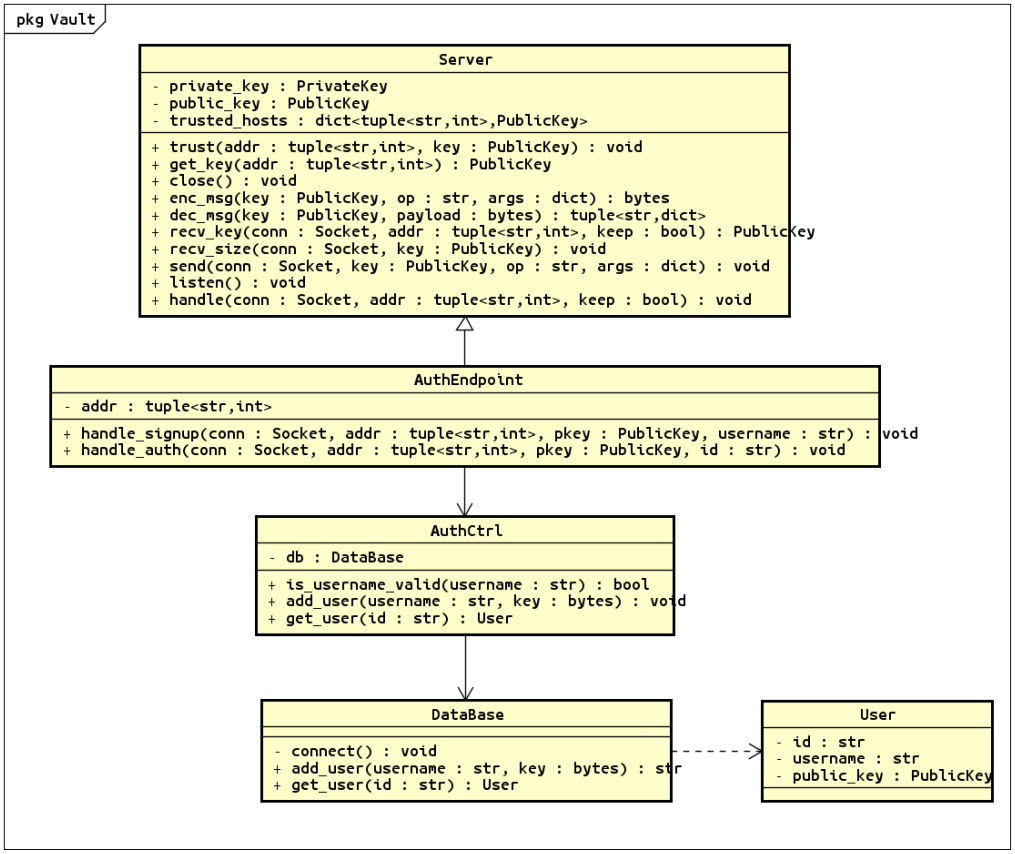


Figura 4: Diagrama de classes do Cofre.

Passando para a Forja, temos um diagrama mais elaborado (Fig. 6), visto que este processo possui mais responsabilidades. A Forja deve gerar as gemas, gerenciar uma cota de solicitação de cada usuário, e ainda mediar fusões de gemas quando dois clientes assim solicitam. Infelizmente, a funcionalidade de fusão não ficou pronta a tempo, e ainda não está funcionando corretamente.

A Figura 5 ilustra como funciona uma requisição de solicitação com a Forja, que necessita de autenticação, já que este processo precisa confirmar a identidade do Cliente para ajustar sua cota de solicitação. Primeiramente a requisição é recebida, o Forja entra em contato com o Cofre para adquirir os dados do suposto usuário, e então usa a chave pública cadastrada para enviar um número secreto ao Cliente. Se o Cliente for quem ele diz ser, ele vai possuir a chave privada correta e será capaz de responder de volta o mesmo número que a Forja enviou, confirmando sua identidade. Só então a Forja responde com os dados de uma gema sorteada, se estiver dentro da cota de solicitação do usuário.

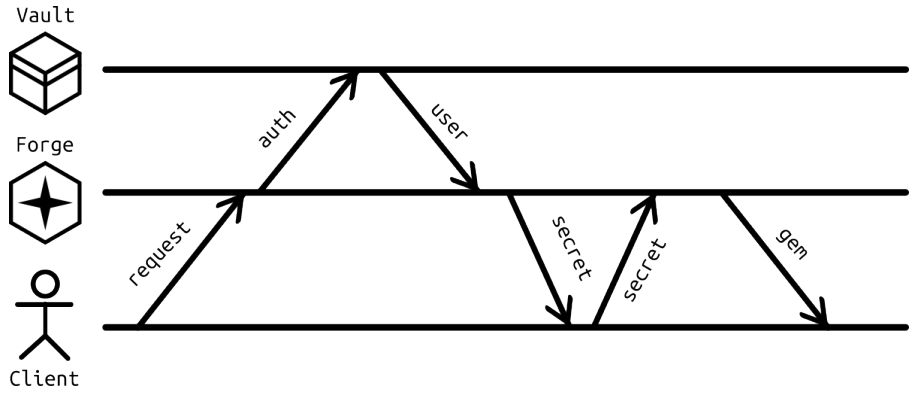


Figura 5: Diagrama de comunicação da requisição de gemas.

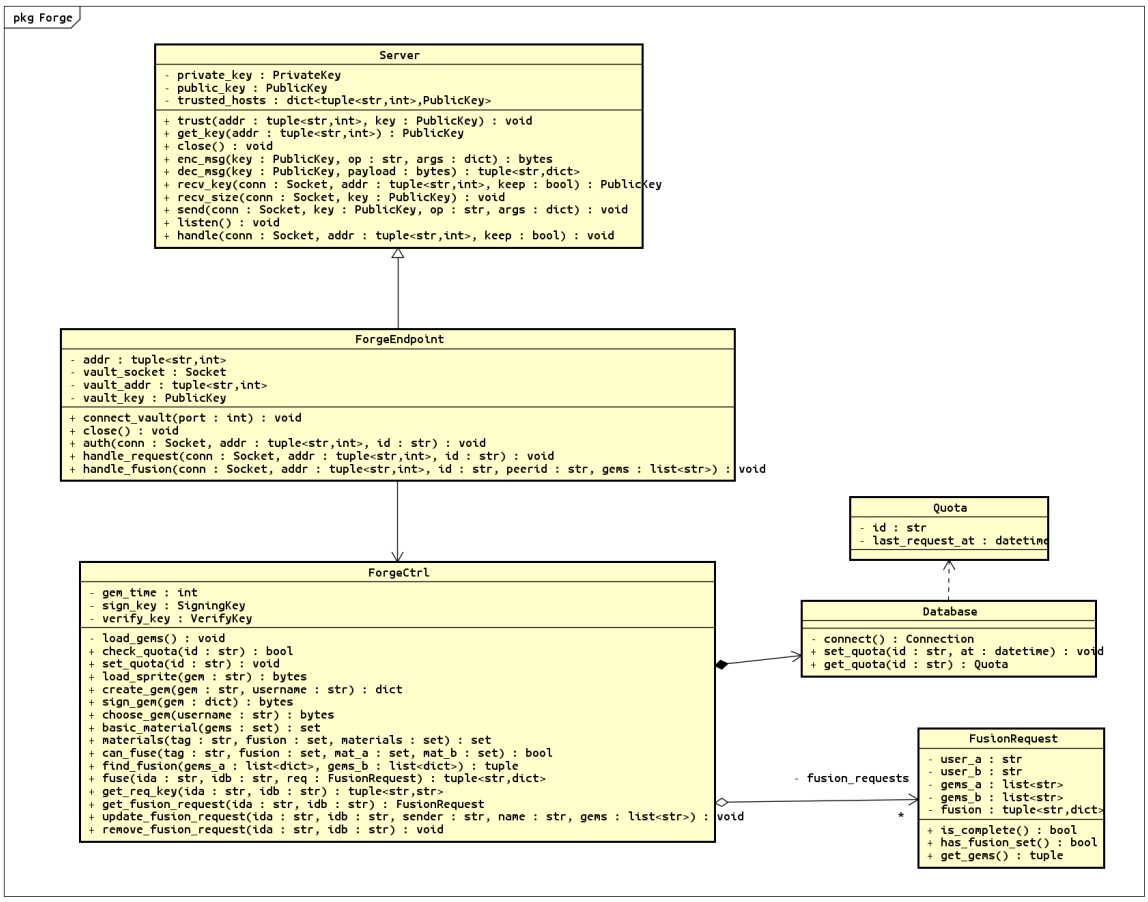


Figura 6: Diagrama de classes da Forja.

Por fim, temos o processo do Cliente, de longe o mais complexo, como pode ser visto em seu diagrama de classes (Fig. 7). Aqui a camada de apresentação foi omitida do diagrama, por motivos de simplicidade, mas a ordem hierárquica das camadas lógicas foi respeitada. As classes da camada de apresentação interagem apenas com as classes do negócio, e as classes do negócio fazem intermédio com as classes de comunicação.

Um grande desafio da implementação de um sistema P2P é capacitar o processo Cliente para ser capaz de iniciar trocas de mensagens, e também estar sempre a escuta de alguma conexão que estiver chegando. A thread principal do processo gerencia a interface gráfica e dispara novas threads que fazem requisições a outros processos, e ao mesmo tempo duas threads *daemon* ficam a escuta por conexões, uma thread no método `listen` do `SearchEndpoint` e outra no método `listen` do `TradeEndpoint`, que possuem estrutura muito parecida com a do Algoritmo 1.

A Figura 8 mostra como ocorre o processo de busca. Quando um cliente quer encontrar outros processos executando na rede local, ele dispara um broadcast direcionado a uma porta específica, bem conhecida pelo sistema distribuído. Os pares desse cliente que estiverem ouvindo, vão responder com suas respectivas galerias. Por se tratar de uma comunicação com UDP e de não haver garantia nenhuma de que algum cliente vai de fato responder às mensagens, o processo fica em espera por um certo tempo antes de parar de ouvir por novos datagramas (deixamos 1 segundo por padrão), e esse tempo é brevemente estendido se algum datagrama for recebido. Qualquer erro nessa comunicação é simplesmente ignorado.

No Algoritmo 2, vemos o método `handle` da classe `TradeEndpoint`. Após receber uma nova conexão, esse método é chamado, e o método por sua vez vai disparar todas as *callbacks* associadas com o tipo de mensagem recebida. Essas *callbacks* são registradas pelas classes da camada de negócio, e tratam-se de funções arbitrárias que repassam informações das mensagens. Da mesma forma, a classe `TradeCtrl` também fornece um mesmo mecanismo de *callbacks*, permi-

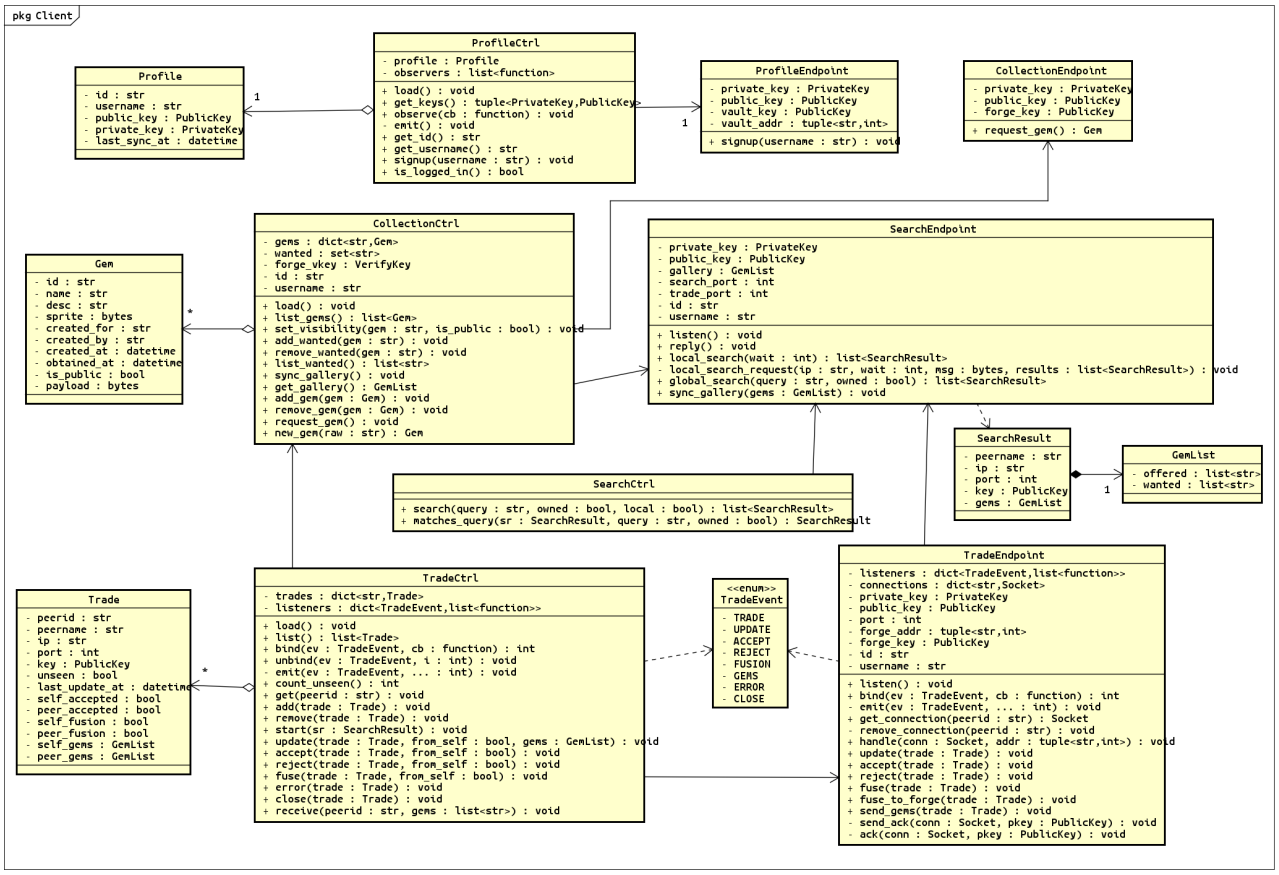


Figura 7: Diagrama de classes do Cliente.

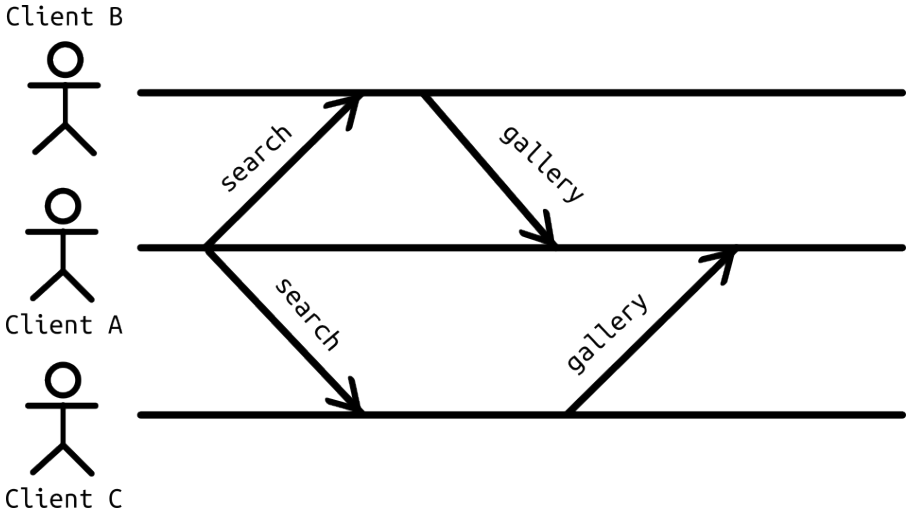


Figura 8: Diagrama de comunicação da busca local.

tindo que a camada de apresentação possa executar código associado a cada evento também. No Algoritmo 3, que é um trecho do construtor do `TradeCtrl`, podemos ver essas callbacks sendo registradas.

Colocando esses mecanismos em ação, estabelece-se o fluxo mostrado na Figura 9. Toda requisição é capaz de partir da interação do usuário desde a camada de apresentação do Cliente A, descer para a camada de negócio, que por sua vez usa a camada de comunicação, e as mensagens chegam na camada de comunicação do Cliente B. Graças às *callbacks* chamadas pelas threads de escuta em segundo plano, essas informações sobem da camada de comunicação

```python
    def handle(self, conn: socket.socket, addr):
        print(f"TradeEndpoint@{addr}: connected")
        peerid = None
        try:
            pkey = self.recv_key(conn)
            while True:
                print(f"TradeEndpoint@{addr}: waiting for msg")
                size = self.recv_size(conn, pkey)
                payload = conn.recv(size)
                op, args = self.dec_msg(pkey, payload)
                if peerid is None:
                    if op != 'trade':
                        raise BadTradeStart()
                    print(f"TradeEndpoint@{addr}: trade start")
                    for cb in self.__listeners[TradeEvent.TRADE]:
                        cb(ip=addr[0], key=pkey, **args)
                    peerid = args['peerid']
                    self.__send_ack(conn, pkey)
                    continue
                ev = TradeEvent[op.upper()]
                print(f"TradeEndpoint@{addr}: {ev}")
                for cb in self.__listeners[ev]:
                    cb(peerid=peerid,**args)
                self.__send_ack(conn, pkey)
                if ev == TradeEvent.REJECT:
                    self.__remove_connection(peerid)
                    break
        except BrokenPipeError:
            pass
            # ...
```

Algoritmo 2: Trecho do método `handle` da classe `TradeEndpoint`.

para a camada de negócio, que por fim repassa para a apresentação do usuário do Cliente B, quando solicitado.

Um usuário deve usar esses dois fluxos de mensagens para efetuar uma troca na prática: primeiro buscar por pares na rede, e propor trocas com quem estiver disponível. Cada troca ocorre apenas enquanto ambos os clientes estão em execução, ou seja, a troca é imediatamente considerada rejeitada e apagada se qualquer um dos dois pares encerrar a conexão (seja por falha, ou por vontade própria).

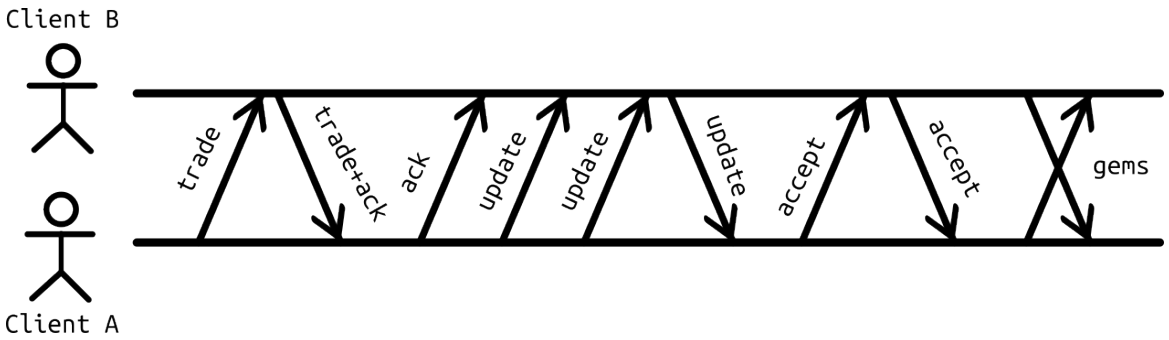


Figura 9: Diagrama de comunicação do processo de troca.

```
endp.bind(TradeEvent.TRADE,
    lambda ip, key, peerid, peername, port, **_:
        self.start_trade(SearchResult(peerid, peername, ip, port, key, GemList([], []), False),
        ↪  False))
endp.bind(TradeEvent.UPDATE,
    lambda peerid, wanted, offered, **_:
        self.update(self.get_trade(peerid), GemList(wanted, offered), False))
endp.bind(TradeEvent.ACCEPT, lambda peerid, **_: self.accept(self.get_trade(peerid), False))
endp.bind(TradeEvent.REJECT, lambda peerid, **_: self.reject(self.get_trade(peerid), False))
endp.bind(TradeEvent.FUSION, lambda peerid, **_: self.fuse(self.get_trade(peerid), False))
endp.bind(TradeEvent.GEMS, lambda peerid, gems, **_: self.receive(peerid, gems))
endp.bind(TradeEvent.ERROR, lambda peerid, **_: self.error(self.get_trade(peerid)))
endp.bind(TradeEvent.CLOSE, lambda peerid, **_: self.close(self.get_trade(peerid)))
```

Algoritmo 3: Trecho do construtor da classe `TradeCtrl`, registrando *callbacks*.

3 Resultados

Nesta Seção, apresentaremos os resultados da implementação, tendo como base a ordem na qual as telas do sistema são apresentadas para o usuário nos casos de uso previstos. Houve um mapeamento quase 1 para 1 nos casos de uso elaborados na entrega 1 do trabalho com as telas implementadas.

3.1 Cadastro e Menu

Na tela inicial é apresentado o logo do sistema, e o campo de preenchimento de nome de usuário para realizar o cadastro. Se o nome já estiver cadastrado será mostrado uma mensagem alertando o usuário, e o mesmo deverá tentar outro nome, como mostra a Figura 10a. Após o cadastro, o usuário é direcionado para a tela de menu. Nela estão presentes os seguintes botões: Coleção, Obter, Buscar e Trocas. Cada um deles levará às outras telas do sistema. A seguir, será comentado sobre cada uma delas. Na Figura 10b temos essa tela.

3.2 Coleção

Na tela da coleção são mostradas as gemas que o usuário possui até o momento. Além disso, é possível selecionar uma gema específica e verificar seus atributos. As Figuras 11a ilustram as telas relacionadas à coleção.

3.3 Gemas Oferecidas e Buscadas

Clicando no primeiro ícone lateral da tela anterior, é possível acessar a tela de gemas oferecidas. Nessa tela, temos quais gemas estão sendo oferecidas por outros usuários. É possível escolher aquelas que você deseja clicando no check box. As gemas marcadas ficarão visíveis para outros usuários quando fizerem uma busca na rede local. Na tela de gemas buscadas, acessada através do outro ícone, com sinal de exclamação, o usuário pode inserir nomes arbitrários de quais gemas ele possui interesse em obter. É intencional que o usuário não saiba quais gemas o sistema pode oferecer. Essa informação também fica visível para seus pares na rede ao fazer uma busca local, de forma que saibam o que oferecer para este usuário. A Figura 12 ilustra essas duas telas.

3.4 Obter Gemas

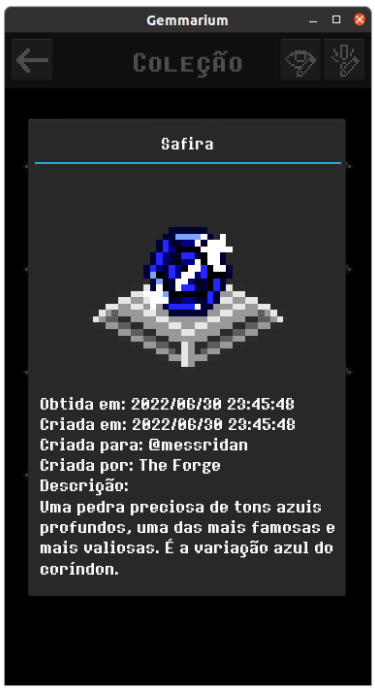
É na tela Obter (Fig. 13) que o usuário irá solicitar novas gemas com a Forja. Basta clicar no botão de solicitação, e a requisição será enviada. No caso do exemplo da figura, foi obtida

(a) Tela de cadastro.

(b) Menu inicial.

Figura 10: Telas iniciais de sistema.

(a) Todas as gemas possuídas.

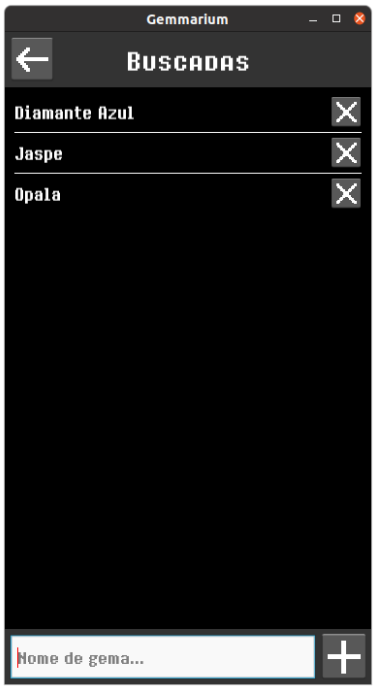


(b) Gema visualizada.

Figura 11: Tela da coleção.

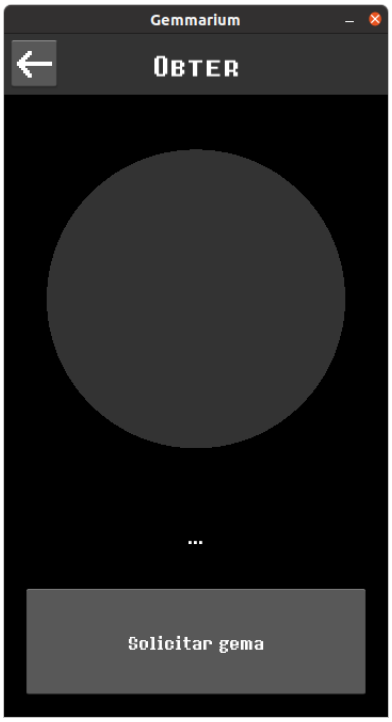
uma ametista (Fig. 13b). Caso o usuário tenha excedido sua cota, é apresentado um tempo de espera até que o usuário possa realizar mais trocas (Fig. 13c).

(a) Gemas oferecidas estão marcadas.



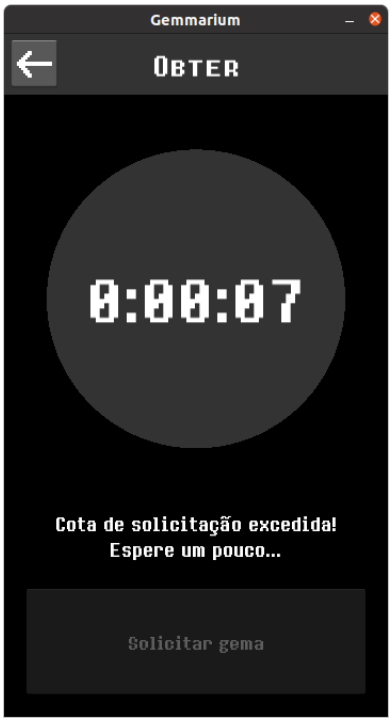
(b) Gemas buscadas podem ser inseridas.

Figura 12: Telas de gemas oferecidas e buscadas.



(a) Antes da solicitação.

(b) Gema recebida.



(c) Cota de solicitação excedida.

Figura 13: Tela de obter gemas.

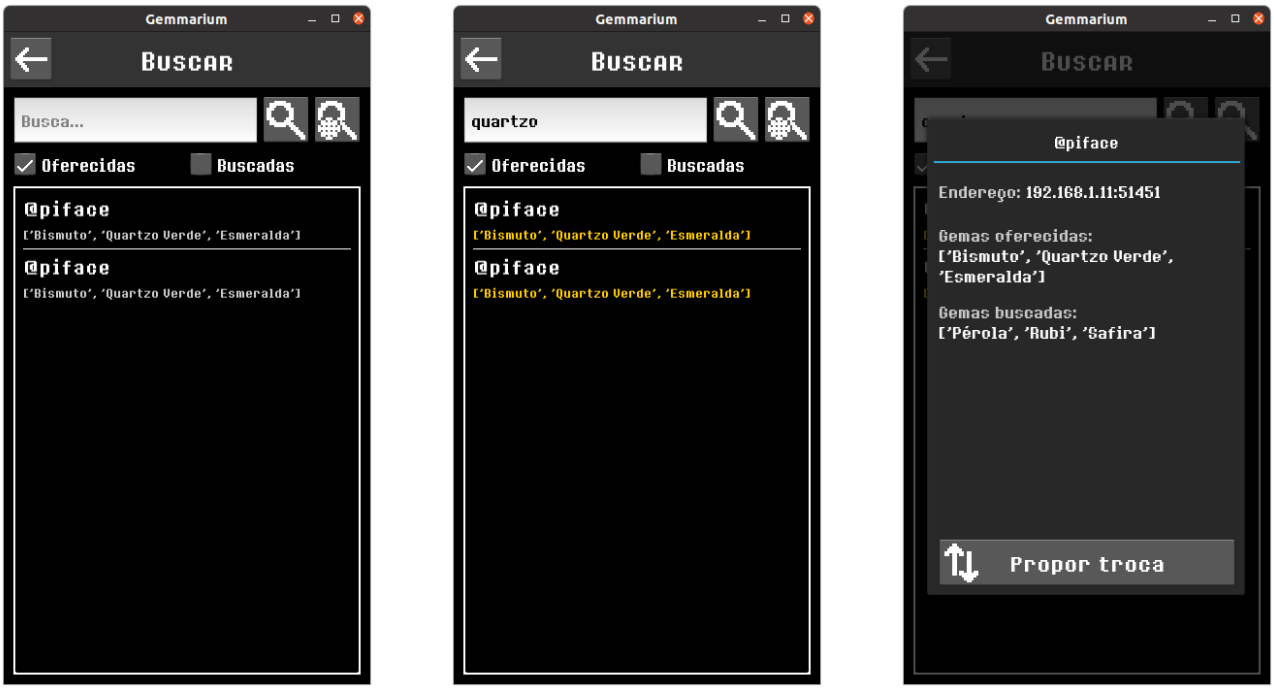
3.5 Busca

Na tela de busca (Fig. 14), temos um campo de texto para procurar usuários que possam possuir as gemas de interesse, tanto as oferecidas quantos as buscadas, limitadas por filtro.

Ao utilizar um termo na pesquisa que casa com as gemas do outro usuário, o texto fica

destacado de amarelo, conforme é mostrado na segunda tela. Importante salientar que, por ser uma execução na mesma máquina, os resultados ficaram duplicados. Mesmo que algum resultado não case com o filtro de busca, ele aparecerá nesta tela, porém com menor destaque e menor prioridade na lista.

Por fim, ao clicar num resultado de busca (Fig. 14c), temos um resumo, mostrando as gemas oferecidas e as gemas buscadas pelo usuário presente na rede local, de nome `@piface`. Estando tudo de acordo, é possível solicitar a troca no botão Propor troca.



(a) Busca sem termos.

(b) Busca com termos.

(c) Proposição de troca.

Figura 14: Tela de busca.

3.5.1 Tela de Trocas

Por fim temos as telas relacionadas às trocas. Primeiramente, vamos mostrar como fica o menu após receber uma troca Figura 15a, e a tela de trocas pendentes Figura 15b, a qual mostra a lista de trocas que ainda não foram concluídas com cada usuário.

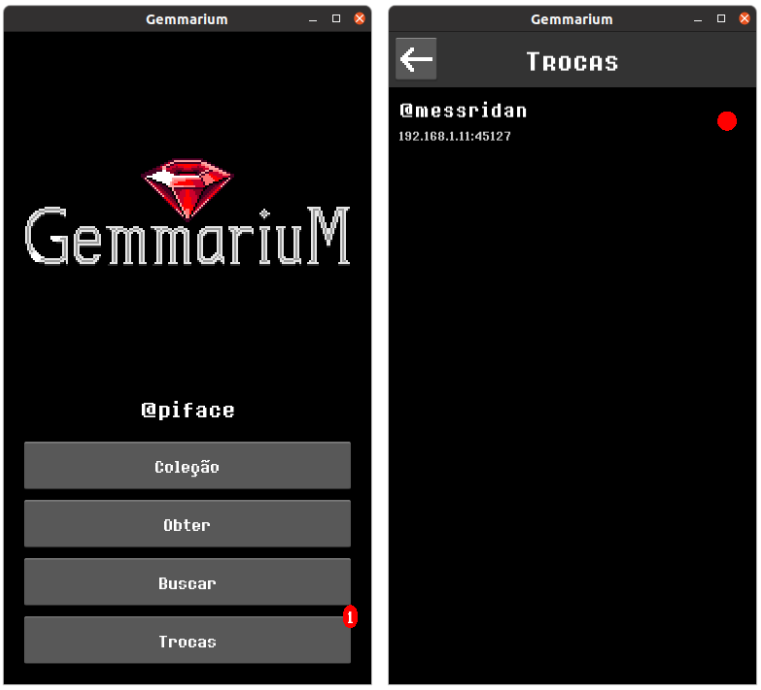
Em seguida, na Figura 16a, temos a tela de troca para o usuário A e, na Figura 16b, temos a tela de troca para o usuário B. Ambos estão acompanhando a mesma troca.

Na parte superior esquerda temos quais gemas estão sendo buscadas pelo par da troca, enquanto que na parte superior direita temos quais gemas esse par está oferecendo. Na parte inferior esquerda temos as gemas que o próprio usuário possui e, aquelas marcadas com check, são as gemas oferecidas por ele, aparecendo na parte superior direita para seu par. Por fim, na parte inferior direita estão as gemas que o próprio usuário deseja. Essas informações podem ser alteradas a vontade até que a troca se conclua.

Através dos botões superiores é possível atualizar de fato o estado da troca. O primerio botão envia as alterações feitas, o segundo aceita a troca, o terceiro a rejeita, e o quarto solicita uma fusão (funcionalidade ainda não implementa)

Em seguida, na Figura 17a e Figura 17b, é mostrado que ambos os usuários aceitaram a troca.

Na Figura Figura 18a, temos uma mensagem, caso uma troca seja rejeitada, enquanto que na Figura 18b e Figura 18c são mostradas as gemas que cada usuário recebe após realizar uma troca.



(a) Menu após receber troca. (b) Lista de trocas pendentes.

Figura 15: Telas do outro cliente ao receber uma solicitação de troca.

4 Conclusão

Com este trabalho foi possível aprofundar na implementação de um sistema distribuído com Sockets, aplicando os conceitos vistos até então na disciplina de forma prática, sem o auxílio de qualquer funcionalidade pronta feita por middleware ou afins.

As principais dificuldades foram relacionadas com o fato de o SD ser peer to peer, em que é preciso ter cautela, pois o cliente tem que estar preparado para agir como um servidor se houver necessidade. Outra dificuldade diz respeito ao uso do próprio sockets, o qual deu muita liberdade, mas com uma certa fadiga como por exemplo em questões de fechamento de conexão e tamanho de buffer.

Referências

[Authority, 2022] Authority, P. C. (2022). PyNaCl: Python binding to the libsodium library.

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

[Brendel et al., 2021] Brendel, J., Cremers, C., Jackson, D., and Zhao, M. (2021). The provable security of ed25519: theory and practice. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1659–1676. IEEE.

[Kivy, 2022] Kivy (2022). Kivy: Open source ui framework written in python, running on windows, linux, macos, android and ios.

(a) Estado inicial da troca para cliente A.

(b) Estado inicial da troca para cliente B.

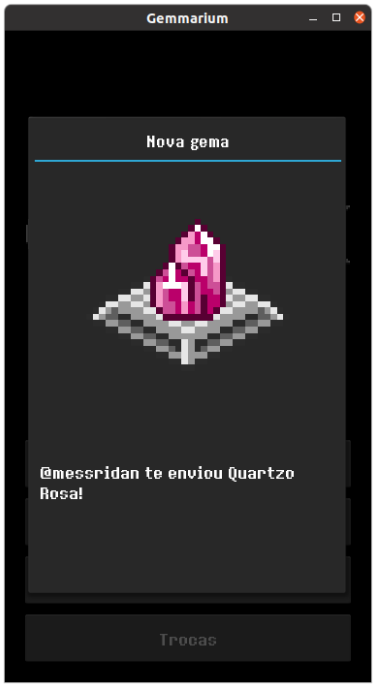
Figura 16: Tela de dois clientes lado a lado, decidindo sobre a troca. Ambos podem incluir ou retirar gemas.

(a) Cliente A aceitou a troca.

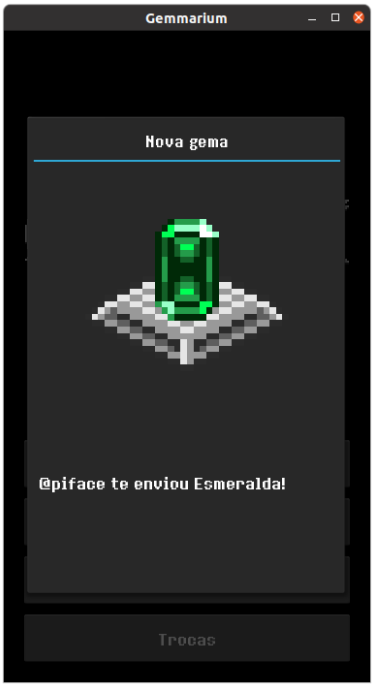
(b) Cliente B vê que A aceitou a troca.

Figura 17: Quando um cliente aceita a troca, ele não pode alterar seus dados.

(a) Quando uma troca é rejeitada, o outro cliente é notificado.



(b) Quando aceita, gemas são recebidas.



(c) Outro cliente também recebe as gemas.

Figura 18: Possíveis resoluções de uma troca.