universidade de aveiro

Segurança 2020/2021

Secure, Multi-Player Online Domino Game
Relatório Final

Daniel Marques, Nº Mec: 85070
Eduardo Coelho, Nº Mec: 88867
Filipe Vale, Nº Mec: 85055
Tomás Freitas, Nº Mec: 84957

Janeiro 31, 2021

Conteúdo

1 Introdução

O objetivo deste trabalho será o desenvolvimento de um sistema que permite os utilizadores criarem e participarem em jogos de dominó online com garantia de integridade e segurança.

Este sistema será constituído por um Table Manager e um conjunto limitado de jogadores que terão que seguir um conjunto de medidas de segurança.

Os jogadores terão de se autenticar com um pseudónimo, ao qual, no fim de cada jogo, serão associados os pontos recolhidos.

O Table Manager terá que criar o baralho de peças de dominó e iniciar todo o processo de ocultação e cifragem. Todos os jogadores deverão participar na cifração e depois na distribuição deste stock, contribuindo para uma distribuição aleatória e para garantir que ninguém sabe qual é a mão (conjunto de peças de dominó) dos outros jogadores através da análise das peças que tirou do stock.

Antes de começar o jogo, os jogadores só poderão jogar as peças que tenham escolhido do stock (aquelas que têm na mão), não podendo usar mais nenhuma. E para que todos os jogadores cumpram isto, eles deverão comprometer-se com a sua mão antes do início do jogo.

O Table Manager será também o responsável pela evolução de todo o jogo e ele, em conjunto com os jogadores, poderão colaborar para apanhar um batoteiro, isto é, poderão verificar se uma dada peça foi jogada duas vezes ou caso tenha sido jogada numa situação onde não poderia.

No fim de cada jogo, deverá ser feita uma contabilização dos pontos recolhidos por cada jogador. Todos os jogadores terão que estar de acordo com esta contagem, podendo os pontos serem transferidos para uma identidade através do Cartão de Cidadão, guardando esta relação num ficheiro ’scores.json’.

2 Implementação

De modo a conseguir ter um sistema fluído de comunicação entre servidor e clientes foi usado sockets TCP o que permite uma abstração maior no fluxo de mensagens bem como já apresenta mecanismos de deteção de erros de transmissão, fazendo a retransmissão de pacotes ou mesmo a agregação de mensagens no mesmo pacote para poupar recursos e assim diminuir tempos de resposta. Com a ajuda da biblioteca select() do python conseguimos eliminar a necessidade de usar processos ou threads para servir e atender vários pedidos de vários clientes, simplificando a implementação do código.

2.1 Login e estabelecimento de sessões

No início do jogo, quando um Cliente se liga ao servidor, executa uma série de tarefas até conseguir jogar o jogo, entre estas o **Login**, o estabelecimento de **Sessões entre Cliente e Table Manager (C2S)** e de **Sessões entre Clientes (C2C)**.

2.1.1 Login

Iniciando o Cliente, este envia uma **Mensagem de Hello** para o servidor.

```
{    "action" :"hello"  }
```

Listing 1: Mensagem de Hello

O servidor após receber o pedido de hello do Cliente, responde com outra mensagem contendo o seu certificado **Mensagem de Login**.

```
{   "action": "login",
    "msg": "Welcome to the server, what will be your name?",
    "server_cert": x509_certificate  }
```

Listing 2: Mensagem de Login

Após a receção do certificado do servidor, o Cliente compara a chave pública do certificado com a chave pública que tem (localmente) do servidor e no caso de serem iguais comprova que o servidor é fidedigno.

Confiando no servidor, o Cliente compõe uma **Mensagem com a sua chave pública e pseudónimo** e cifra a mesma com a chave pública recebida pelo servidor, garantindo assim que só o detentor do certificado pode decifrar a mensagem.

```
{   "msg": pseud nimo, "pub_key": pub_key   }
```

Listing 3: Mensagem com a sua chave pública e pseudónimo

O Cliente responde então ao servidor com uma **Mensagem de Resposta de Login**

```
{   "action": "req_login","msg": mensagem_cifrada }
```

Listing 4: Mensagem de Resposta de Login

Já no lado do servidor e após decifrar, com a sua chave privada, a mensagem enviada pelo Cliente, este verifica se o pseudónimo já está a ser usado por alguém e em caso afirmativo desconecta o Cliente. O servidor associa então o Cliente ao Table Manager que guarda

então a socket, o pseudónimo e a chave pública do Cliente, confiando nela uma vez que veio cifrada com a sua chave pública.

O primeiro Cliente a trocar estas mesagens com sucesso com o servidor passa a ser o host do jogo, sendo este o único que pode mandar começar o jogo.

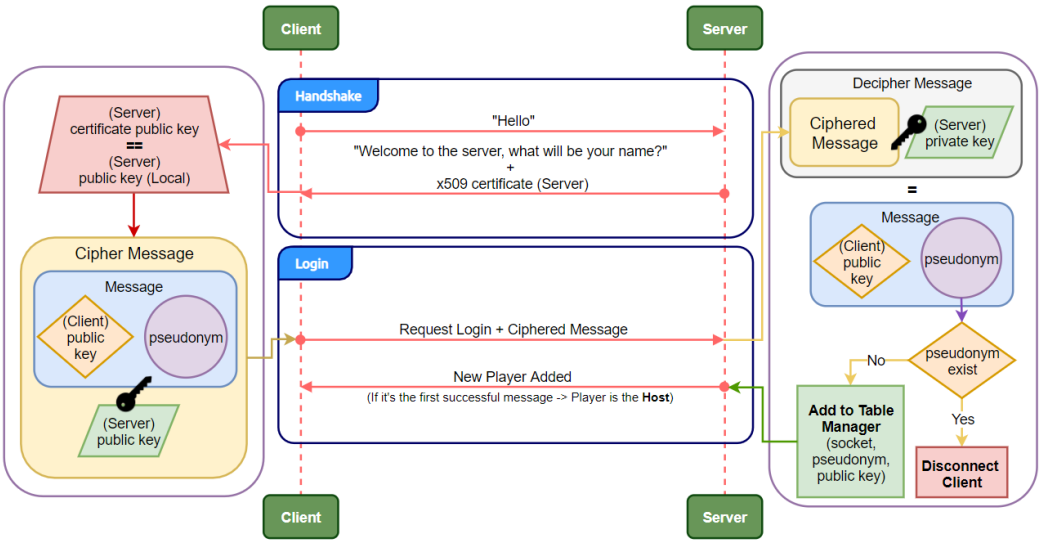

Figura 1: Login Process

2.1.2 Sessões entre Cliente e Table Manager (C2S)

Enquanto os Clientes aguardam que todos os jogadores se juntem ao Table Manager, este combina uma chave de sessão com cada jogador. Essa chave de sessão é uma chave simétrica derivada da chave partilhada gerada pelas negociações do Diffie-Hellman.

Para essas negociações o Table Manager compõe uma **Mensagem inicial de negociação Diffie-Hellman** onde insere uma chave pública Diffie-Hellman que se encontra cifrada com a chave pública do Cliente, garantindo assim que só o mesmo consegue saber qual a chave pública Diffie-Hellman.

```
{   "action": "server_dh_negotiations","dh_key": mensagem_cifrada   }
```

Listing 5: Mensagem inicial de negociação Diffie-Hellman

O Cliente recebe então a mensagem do servidor, decifra a chave pública Diffie-Hellman do servidor com a sua chave privada e envia para o Table Manager uma **Mensagem final de negociação Diffie-Hellman** onde insere uma chave pública Diffie-Hellman que se encontra cifrada com a chave pública do servidor, garantindo assim que só o mesmo consegue saber qual a chave pública Diffie-Hellman.

```
{   "action": "reply_dh_negotiations","dh_key": mensagem_cifrada   }
```

Listing 6: Mensagem final de negociação Diffie-Hellman

Para finalizar, o Table Manager recebe então a **Mensagem final de negociação Diffie-Hellman** e decifra a chave pública Diffie-Hellman do Cliente com a sua chave privada.

Após estas negociações tanto o Cliente como o Table Manager têm as chaves públicas Diffie-Hellman um do outro conseguindo assim gerar uma chave partilhada Diffie-Hellman.

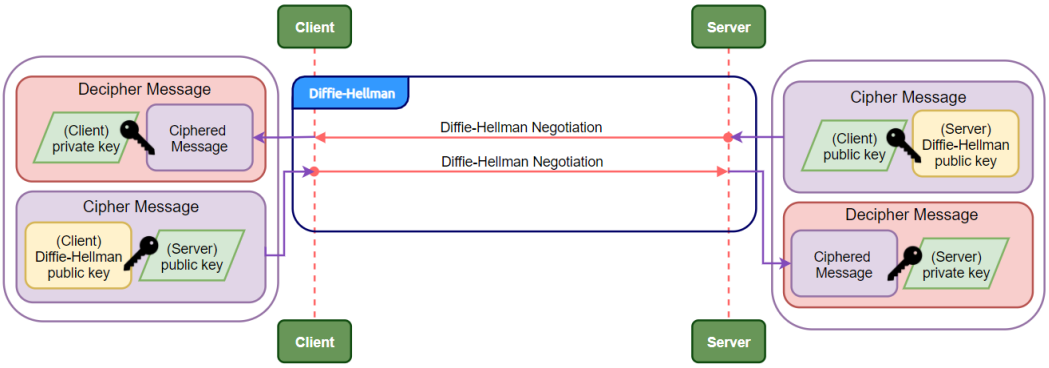

Figura 2: Negociações Diffie-Hellman

2.1.3 Sessões entre Clientes (C2C)

O estabelecimento de sessões entre Clientes é muito idêntico ao estabelecimento de sessões entre Cliente e Table Manager. Neste caso e devido os Clientes não conseguirem comunicar uns com os outros, usam o servidor como proxy e por isso, para evitar que o servidor saiba o que os Clientes estão a comunicar, as mensagens são cifradas entre Clientes. Como resultado final, o servidor a única coisa que sabe é que tem de reencaminhar uma mensagem para um Cliente.

Estas mensagens são cifradas usando as chaves públicas dos Clientes destino, tendo sido enviadas pelos próprios Clientes para o servidor e do servidor para os outros Clientes.

Essa chave de sessão também é uma chave simétrica derivada da chave partilhada gerada pelas negociações do Diffie-Hellman.

Para essas negociações cada Cliente compõe uma **Mensagem de negociação Diffie-Hellman para Clientes** onde insere uma chave pública Diffie-Hellman.

```
{   "action": "dh_negotiations", "msg": dh_pulicKey }
```

Listing 7: Mensagem de negociação Diffie-Hellman para Clientes

Por sua vez, essa **Mensagem de negociação Diffie-Hellman para Clientes** é cifrada usando a chave pública do Cliente destino impedindo assim o servidor de ler o seu conteúdo e envia a **Mensagem proxy para o Table Manager**.

```
{   "action": "send_to", "player_name": destination_player,
    "sent_by": current_player, "msg": mensagem_cifrada }
```

Listing 8: Mensagem proxy para o Table Manager

O Table Manager recebe então a **Mensagem proxy para o Table Manager** e reenvia para o Cliente destino alterando alguns campos da mensagem.

Para finalizar o Cliente destino recebe então a **Mensagem de negociação Diffie-Hellman para Clientes** e decifra a mesma usando a sua chave privada guardando a chave publica enviada pelo outro Cliente.

Uma vez que o estabelecimento de sessões entre Clientes acontece ao mesmo tempo, quando os Clientes recebem a **Mensagem de negociação Diffie-Hellman para Clientes** já têm as chaves públicas Diffie-Hellman um do outro conseguindo assim gerar uma chave partilhada Diffie-Hellman

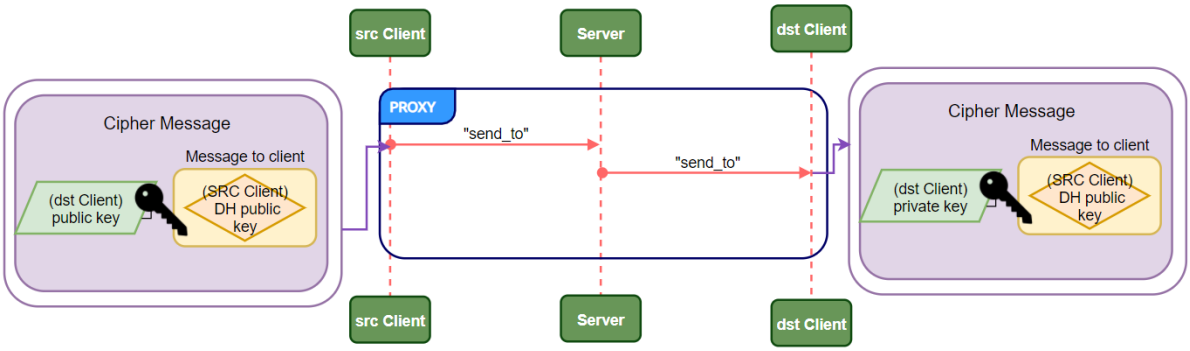

Figura 3: Sessões entre Clientes

2.2 Controlo de integridade, autenticidade e proteção de mensagens

Para garantir a integridade e autenticidade das mensagens enviadas pelo Table Manager e pelos Clientes usamos códigos de autenticação de mensagem baseados em hash (ou HMACs). Estes códigos permitem garantir que uma mensagem não foi adulterada por ninguém comparando o código agrupado na mensagem recebida, com o resultado de uma hash que recebe a mensagem e a chave de sessão que combinou com esse Cliente. Neste caso em questão a hash usada foi sha512.

No caso em que as mensagens são trocadas entre Clientes o controlo de integridade é feito em 2 niveis:

- na comunicação entre Cliente e servidor (proxy) onde o Cliente e o servidor fazem a verificação normal nas mensagens.
- na comunicação Cliente e Cliente onde os Clientes usam as suas chaves de sessão para validar que a mensagem não foi adulterada pelo servidor.

Para garantir a proteção das mensagens todas as chaves públicas e privadas são geradas segundo o algoritmo de geração de chaves assimétricas RSA. Apesar de ser recomendado usar chaves RSA de 2048 bits achamos que neste cenário basta usar chaves de 1024 para os Clientes, pois são geradas quando os Clientes são ligados e o tempo de segredo das mesmas é curto. A única chave de 2048 bits é a do servidor uma vez que os Clientes têm sempre acesso a ela estando guardada localmente para futura comparação com o certificado enviado pelo servidor.

No que toca a chaves simétricas usamos chaves de 32 bits, sendo as chaves de sessão obtidas através de uma função de derivação de chaves HKDF usando o algoritmo SHA256 e derivando com a chave resultante da negociação do Diffie-Hellman.

2.3 Protocolo de distribuição segura do deck

No início do programa é criado um deck composto por listas de dois argumentos: index e objeto Piece. O objeto Piece é construído através de dois valores: o valor da esquerda e o valor da direita. Estas combinações de valores são retiradas do ficheiro ’pieces’.

2.3.1 Pseudonymization Stage

```
Deck created
 [0,  4:1][1,  3:0][2,  2:1][3,  2:0][4,  3:3][5,  1:0][6,  3:1][7,  6:2][8,  6:6][9,  5:3][10,  6:4][11,  4:0]
[12,  3:2][13,  5:0][14,  5:4][15,  4:3][16,  5:5][17,  1:1][18,  6:1][19,  4:2][20,  5:1][21,  2:2][22,  6:3][
23,  5:2][24,  4:4][25,  6:5][26,  0:0][27,  6:0]
```

Figura 4: Criação inicial do deck

Com a criação do deck concluída, iniciamos a pseudonimização das peças presentes no mesmo (h(i,Ki,Ti)). Para facilitar este processo, em vez de utilizarmos o objeto Piece, transformamos num Integer através do seu método id(), que multiplica o valor da esquerda por 10 e soma ao valor da direita. Depois, ao criarmos um Ki, conseguimos através da biblioteca hashlib criar o pseudónimo da peça, guardando o tuplo (Ki,Ti) na lista pseudo_Table no index [i].

```python
for tile in range(0,len(self.game.deck.deck)):
    ix = int(self.game.deck.deck[tile][0])
    pwd = str(randint(00000, 99999))

    kdf = PBKDF2HMAC(hashes.SHA1(), 16, b'\x00', 1000, default_backend())
    Ki = kdf.derive(bytes(pwd, 'UTF -8'))

    self.pseudo_table[ix] = (Ki.hex(), self.game.deck.deck[tile][1].id(), False)

    dk = hashlib.sha256()
    dk.update(str(self.game.deck.deck[tile][1].id()).encode('utf-8'))
    dk.update(Ki)

    self.pseudo_deck.append([ix, dk.hexdigest()])
```

Figura 5: Criação dos pseudónimos com h(i,Ki,Ti)

Após este processo, o Table Manager terá uma pseudo_Table com os tuplos guardados e terá o deck resultante com as peças pseudonomizadas.

```
PSEUDO DECK:
[[0, 'dddff9f155d56f1a856dcf61a281b09b8c7eeb8a353066a34978345817863795'], [1, 'f973306afe5b9545bd476322205cbb7f
0a69076ed5b54e4ae36547f2f3b9e722'], [2, '6f9a29f137505e08bfa05b22df47573a086a76306e73ba73ba84ef526cc741bc'], [3
, '401691908c25bdf0dd073f2bf35401afeea77907290561751646 1c079465c053'], [4, '6134edbc8bd7ecb95f8d65bf5a789d52546
13c422350d7e3cf91b20c325b7b5b'], [5, '436ac0eacddc391a709f63a5472ea118cde3a938f49e9df59c4c7a6c577e5849'], [6, '
196b1cd52190ff5b853fcc6ed69d1bb50537eae411604e50a350b71d67b15c9a'], [7, '2de1386286980f2b9e02220e1b294e4c66cd4a
0127f63d05907a9d462585ad6b'], [8, '9995f5be3edd4bd44696f7839499c4cb5944075e53134083efab224dbce8f48a'], [9, 'fc1
c4fafefa14d009a30018596f13c8960ea3625a253fa4b87a9d23732e88172'], [10, '21d1773db831dea4d254a885f395d941e9a69ab4
72d49583ffc8be4340eac9eb'], [11, '247a2f8505566a0c284f07d2f0a04c5369ffe36cf2ff9b5475c07f96fea328c6'], [12, '9ae
6bc70dae0f4ac8937a0460cb00b071761ee63a6a9a1195d5a89ddf99d7c01'], [13, '76827159fe99b201745e62aa5a3e27b6c420de7e
c6ec176c4dfa5e0156ec949e'], [14, '94fa8773a8c0a6694ae8d416fca9e49e728f2f6fb2ad44c1e826251d429729a9'], [15, '41c
e44ba835e9d4d813eb1a13ba875546c99df06c84aec43549e6fde5a1efa0e'], [16, 'cf7e0a50c10b8f03e39fb8810f09a3b043c81929
80ab11719e95725dea96389a'], [17, 'd684959a7fe83860c73408b80299c46e8a0a035df00d687638794deb53d88192'], [18, '165
e2ee332eea499ae2c714be194dbfeeb0cf18979fbe9db550ec91eef91d65a'], [19, 'b1696c7d469214f4a75609726ddcb6c34e69dd86
0bb76be6d24b56a3ba3706cb'], [20, '918025d30df100261634b190cb1effd0b7a1ea641a37d9615ac0d7f0f429ea11'], [21, '918
b3929a7fe324f238f03aab3bcf97e1d00b228d3068f54ca789cb8d56cbb0b'], [22, 'eb851308a4f0aaff6d13c38f6bd0e2cd35d94a91
50bdfdf5c1204fad15e87691'], [23, 'c9096932cf5332eb67ec8b7c4f8a3d4418c3152d09a3c6af47a3575b3830a268'], [24, '6f5
094b59490dea8614674a1f75273cd1231e0c551ef993e7e7a4407af329c0d'], [25, '7f5172f900e104f9caf42a24c83ce107bd67883d
28f11cbc9adc92df5e650196'], [26, '1c7642c085f92b2981070c69a20022bbad3c781bb47b13f280468bb8ab26645c'], [27, '89c
1a2d3f0956e41c9f794f0948f2496596528f97f451aed4acffb6a766cef66']]
```

Figura 6: Deck após processamento, com peças pseudonomizadas

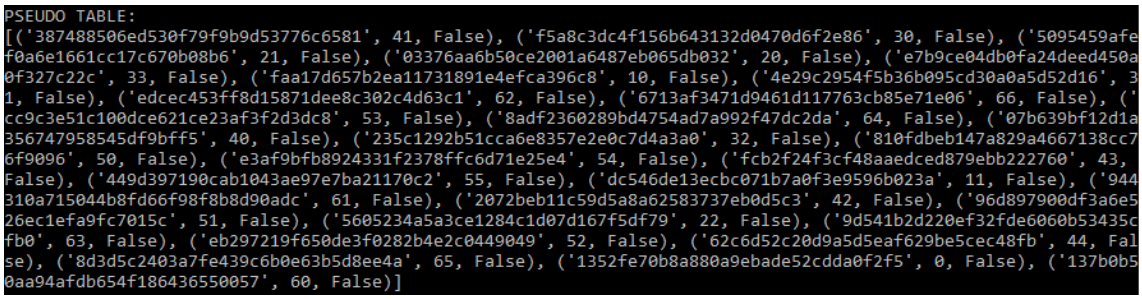

Figura 7: Tabela onde guardámos os tuplos (Ki,Ti)

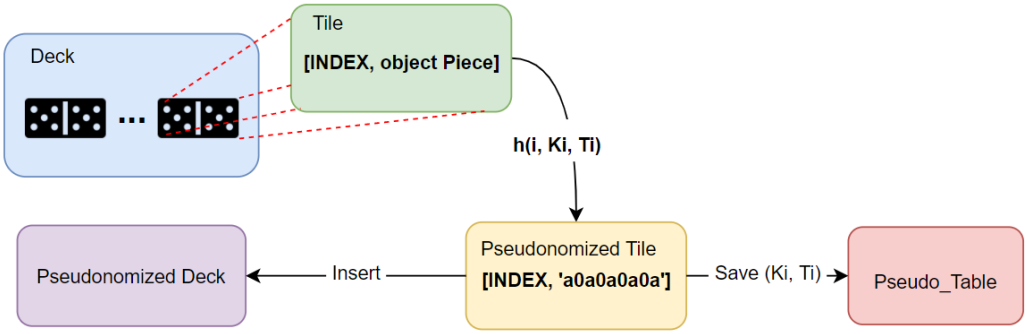

Figura 8: Pseudonymization Stage

2.3.2 Randomization Stage

O primeiro jogador a cifrar o deck recebe o deck que já tem as peças pseudonomizadas. Após cifrar todas as peças e dar shuffle, envia para o jogador seguinte, através da ação 'send_to_unencrypted' pois não é necessário cifrar a mensagem nesta fase, mas vai sempre com o hmac associado, para o recetor poder comprovar a integridade dos dados. O processo repete-se até ao último jogador, que após cifrar o deck, envia o resultado para o Table Manager.

Para cifrarmos a peça (lista [i, Pseudónimo]) transformamos em string e utilizamos a cifra AESGCM que utiliza uma key de 32 bytes, um nonce de 12 bytes e um IV sempre diferente.

Tendo 'ciphertext = nonce + AESGCM(key).encrypt(nonce, tile, b)', o Cliente guarda cada combinação de (key,ciphertext), ou seja, chave utilizada e peça resultante da cifração numa lista privada sua com a qual vai no futuro responder a pedidos de chaves.

No fim desta fase, o Table Manager adiciona outro index a todas as peças. Este novo index tem utilidade ao facilitar o bitcommitment por parte dos jogadores numa fase futura.

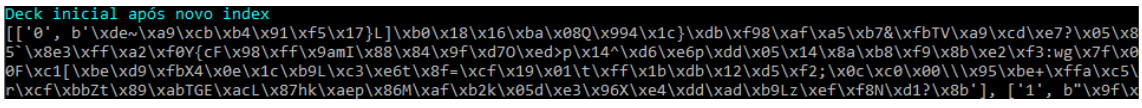

Figura 9: Exemplo da primeira peça após toda a encriptação e adição do novo index

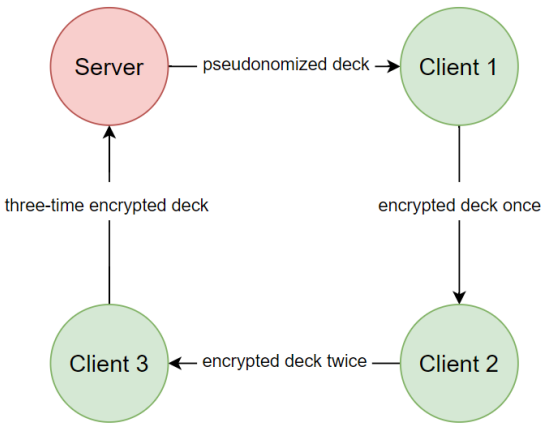

Figura 10: Randomization Stage

2.3.3 Selection Stage

O deck vai circular secretamente entre jogadores, que têm 5% de chance de retirar uma peça, 45% de trocar uma a 'n' peças (sendo 'n' o número de peças que tem na mão) e 50% de chance de passar sem realizar nenhuma ação.

A circulação secreta do deck é feita através do uso das session keys estabelecidas no início do programa, sendo que o server apesar de ver a mensagem com o deck que vai de um jogador para o outro, não consegue interpretar o seu conteúdo. A circulação aleatória é conseguida através do jogador escolher aleatóriamente um outro jogador para enviar o deck (cifrado com a session key).

O final desta fase acontece quando um dos jogadores repara que o deck tem um número de peças especifico que significa que todos os jogadores já têm a sua mão cheia. Com isto, tem uma pequena chance de informar o Table Manager que a fase terminou.

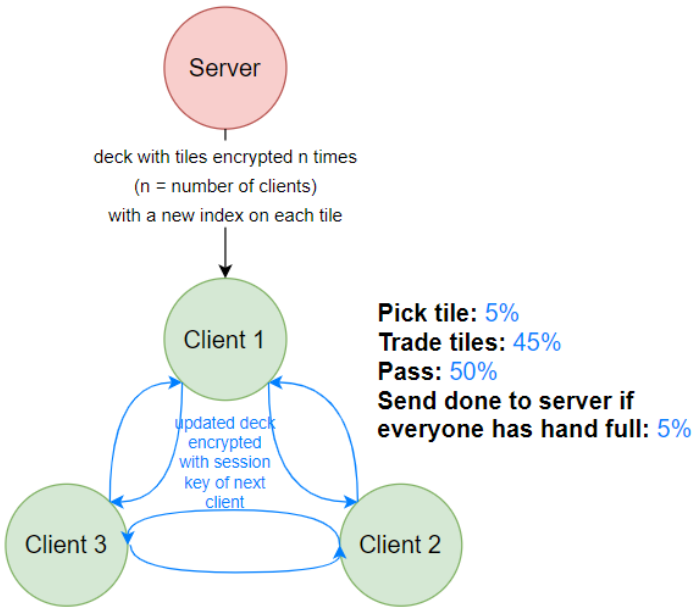

Figura 11: Selection Stage

2.3.4 Commitment Stage

Esta fase é realizada logo após a Selection Stage, onde cada jogador recebe o estado atual do deck e faz o bitcommitment utilizando os novos indexes adicionados ao deck no final da Randomization Stage.

```
for tile in self.player.hand:
    msg += tile[0] +','
self.bitcommitment_msg = msg[:(len(msg)-1)]
self.bitcommitment_key = "{0:0{1}x}".format(random.getrandbits(256), 64)
encoded_value = (self.bitcommitment_msg+self.bitcommitment_key).encode()
self.bitcommitment_res = sha256(encoded_value).hexdigest()
```

Figura 12: Exemplo do cálculo do bitcommitment enviado pelo jogador

Os jogadores guardam numa variável privada a mensagem e a chave que utilizaram para fabricar o seu bitcommitment, pois será útil no futuro quando for necessário encontrar um cheater após uma jogada não válida.

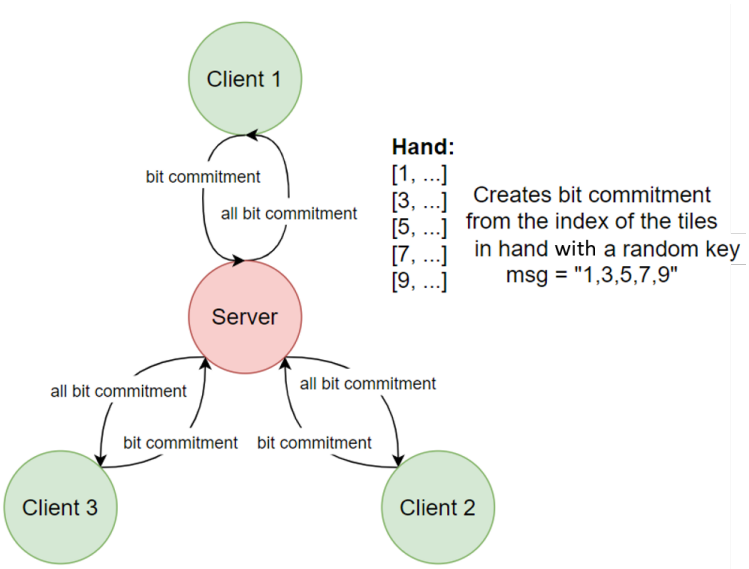

Figura 13: Commitment Stage

2.3.5 Revelation Stage

No início desta fase o último jogador a cifrar o deck irá juntar os pares de (chave utilizada, ciphertext resultante) das peças que não estão atualmente no deck, ou seja, que estão na mão de algum jogador.

Os jogadores seguintes utilizam os pares de chaves anteriores para decifrar o deck até conseguirem ver resultados da sua cifração passada e conseguirem também eles fornecer as chaves que utilizaram nesse momento.

Após isto, todos os jogadores sabem todas as chaves utilizadas para as peças atualmente em jogo, e assim, conseguem decifrar as peças na sua mão até verem o par [i, pseudónimo].

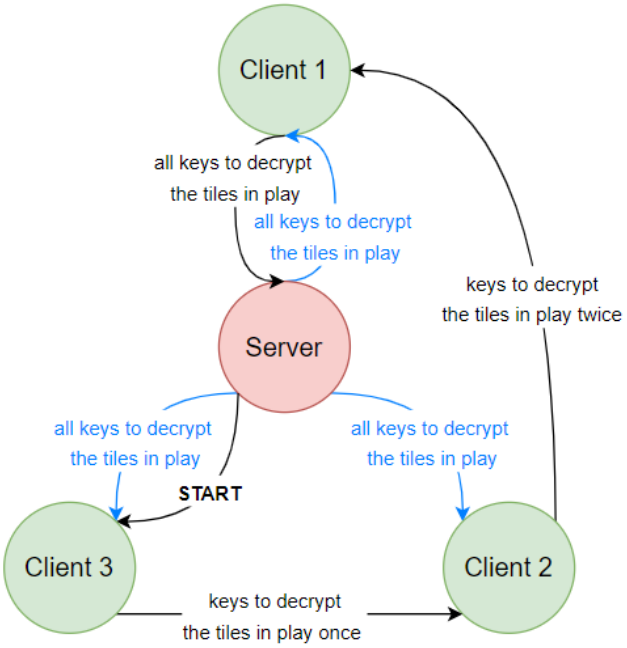

Figura 14: Revelation Stage

2.3.6 Tile de-anonymization preparation Stage

No início desta fase o Table Manager envia um array de 28 slots vazias para um jogador. Este jogador tem 5% de chance de adicionar uma chave pública ou 95% de chance de enviar o array para outro jogador aleatório, com o array devidamente cifrado com a session key combinada entre ambos.

```
privkeylist = [None] * 28
for index in self.indexes_to_deanon_save:
    pub_key, priv_key = security_utils.createAsymKeys()
    privkeylist[index] = [pub_key, priv_key]
```

Figura 15: Lista em que o jogador guarda o par chave pública/chave privada

Quando um dos jogadores detetar que o número de chaves públicas no array é igual ao número de jogadores multiplicado pelo número de peças máximo numa mão, tem 5% de chance de terminar esta fase e avisar o servidor, enviando o array no estado atual.

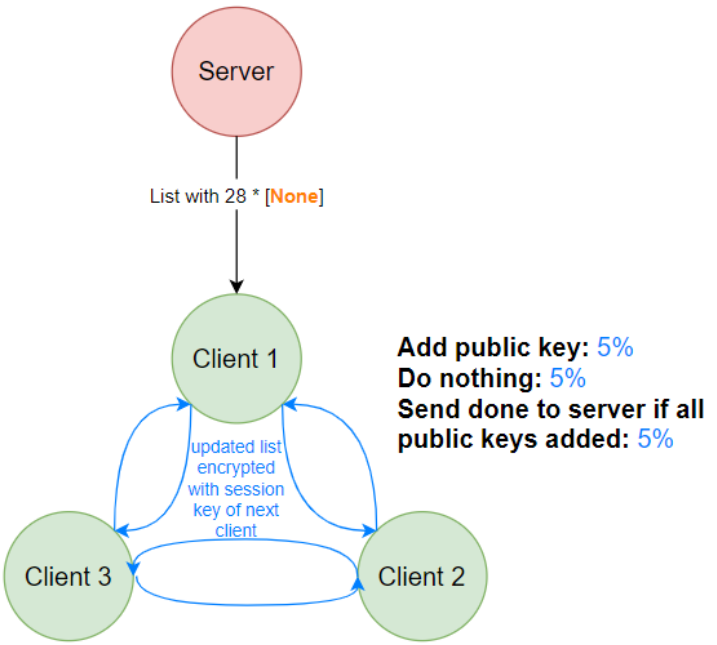

Figura 16: Tile de-anonymization preparation Stage

2.3.7 Tile de-anonymization Stage

Tendo o array com todas as chaves públicas, o Table Manager cifra os tuplos (Ki,Ti) correspondentes ao index onde a chave pública está, com essa mesma chave pública, e coloca o resultado no mesmo index do array. Ao terminar envia o resultado para todos os jogadores.

Os jogadores, como guardaram previamente os índices que preencheram e as chaves privadas que lhes permitem decifrar a chave pública que lá colocaram, vão ter finalmente acesso às suas peças, podendo verificar através do Ki recebido se o servidor não mentiu na peça dada.

Quando todos os jogadores se declararem prontos, é iniciado o jogo.

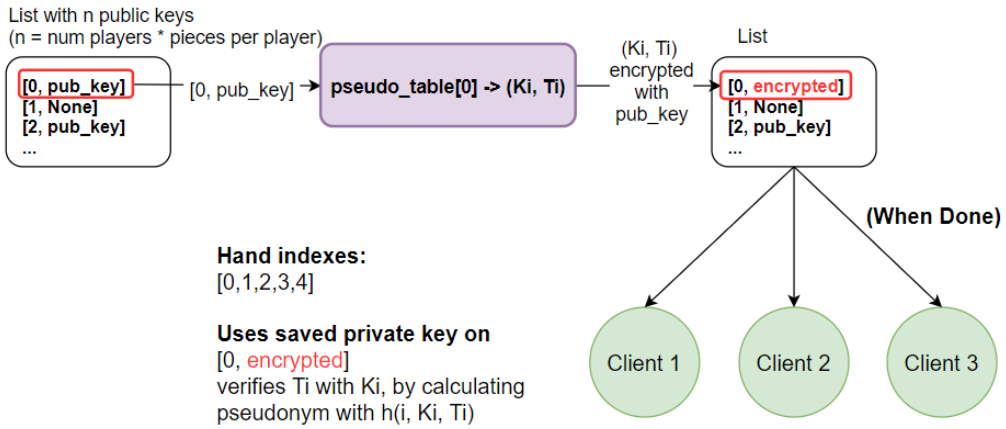

Figura 17: Tile de-anonymization Stage

2.4 Validação de peças jogadas durante o jogo por cada Cliente

Após cada jogada, todos os jogadores informam se a peça jogada estava na sua mão ou não. Caso não esteja, o Table Manager procede a mais duas validações:

- A peça jogada estar na Table
- A peça jogada ainda estar no stock

Como perguntamos a cada jogada se a peça estava na mão de outro jogador e verificamos se estava no stock, caso a peça jogada já se encontrar na mesa é imediatamente considerado que o último jogador a jogar é o cheater, pois se realmente tivesse a peça teria se queixado no momento em que foi jogada a primeira vez, ou o Table Manager teria detetado que a peça estava no stock.

Em relação ao outro caso de a peça jogada ainda estar no stock, na pseudo_Table do Table Manager (onde guardamos o par (Ki,Ti)), colocamos em vez de um tuplo, um triplo, sendo o terceiro argumento iniciado a False e alterado para True após alguem fazer um pedido de de-anonimização aquela peça. Assim, o Table Manager sabe exatamente quais são as peças que ainda estão no stock, pois são as peças que ainda têm o terceiro argumento a False.

Com isto, um cheater ao jogar uma peça que está no stock, o servidor irá imediatamente abortar o jogo e anúnciar o último jogador que efetuou a jogada como cheater.

2.5 Protestar contra batota

Quando um jogador protesta dizendo que a última peça jogada se encontra na mão dele o jogo é abortado e entram numa fase de descoberta do cheater, onde o jogador que acusa partilha a mensagem e a chave com que calcularam o seu bitcommitment na Commitment Stage.

Caso o jogador forneça um par mensagem,chave que, ao calcular um novo bitcommitment, este seja diferente do fornecido anteriormente, assumimos imediatamente que este jogador é o cheater, pois tinha intenções de falsificar informação.

Caso o par mensagem,chave resulte no bitcommitments fornecido anteriormente, é necessário ir ao deck inicial e decifrar e de-anonimizar as peças que têm os índices enviados na mensagem. Se um dos índices for a peça que foi jogada, significa que o acusador é inocente, e o outro é considerado o cheater.

Caso ao decifrarmos e de-anonimizarmos as peças dos índices fornecidos não encontrarmos a peça jogada, significa que ela pode ter sido tirada do stock durante o jogo pelo jogador e depois jogada por outro (cheater). Para encontrarmos o cheater, o Table Manager tem um game log em que guarda as peças que forneceu ao de-anonimizar peças tiradas do stock e a quem forneceu estas mesmas peças. Assim, através do log, o Table Manager conseguirá saber se o jogador tirou de facto essa peça do stock e o outro será considerado cheater.

Caso nenhum destes casos se confirmar, o acusador é considerado o cheater visto que protestou a jogada de uma peça que ele próprio não a tinha mostrando intenções de sabotar o jogo.

2.6 Possibilidade de fazer batota

É possível efetuar cheating com um Cliente ao executar o programa com a flag '-cheat'. Com isto, sempre que for a vez deste Cliente jogar, será abordado com uma escolha: jogar

automaticamente esse turno ou realizar um ato de cheating, isto é, decidir jogar uma certa peça, retirar uma peça do stock, passar a jogada ou forçar um protesto da próxima jogada. Caso escolha que quer decidir a peça que quer jogar, o jogo irá ocorrer normalmente se jogar uma peça da sua mão, ou não será validada caso jogue uma peça que esteja na mão de outro Cliente, na mesa ou no stock. Caso retire uma peça do stock o jogo continua normalmente e o jogador tem a oportunidade de jogar uma outra peça, se decidir passar e ainda houverem peças no stock o jogo é abortado pois existe a obrigatoriedade de retirar peças sempre que elas existam e, por último, se decidir forçar um protesto numa próxima jogada ele próprio vai ser acusado de ser o cheater pois protestou uma peça que se vem a provar que ele próprio não tem.

2.7 Contabilização dos pontos num jogo

Para calcular o número de pontos de cada jogador para um dado jogo, sempre que for jogada uma peça do dominó, é calculado o seu valor somando os valores das suas extremidade, por exemplo se a peça jogada for [3 • 2], o seu valor será 3 + 2 = 5, sendo incrementado esse valor à atual pontuação do jogador (score).

No fim do jogo, o jogador vencedor será o primeiro que não tiver peças na sua mão e consequentemente a sua pontuação será o valor total da soma das suas peças jogadas.

Por outro lado, a pontuação dos jogadores perdedores até ao fim do jogo é calculada usando o mesmo processo descrito acima mas terá que ser feita depois uma subtração do valor total das peças que possui na sua mão, isto é, caso um jogador chegue ao fim do jogo com uma pontuação de 19 mas tiver na sua mão as peças [4 • 1] e [2 • 3], a sua pontuação final será de 19 - ((4+1) + (2+3)) = 9.

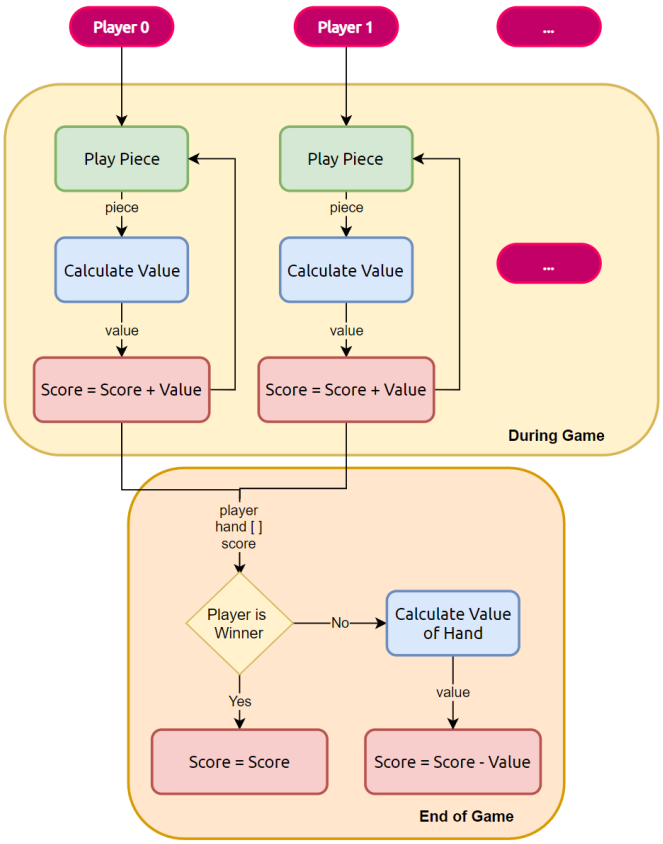

Figura 18: Contagem dos pontos para cada jogo

2.8 Guardar pontos associando a uma entidade através do Cartão de Cidadão

No fim de cada jogo, é colocada a questão "*Do you want to save your Score?*" onde cada jogador tem a hipótese de guardar os seus pontos ou não, respondendo com um '**y**' (yes) ou um '**n**' (no) respetivamente.

Após esta escolha e caso seja afirmativa, procede-se ao carregamento de todos os certificados do Cartão de Cidadão e das Listas de Revogação de Certificados para listas de tuplos, uma para os certificados Raiz, outra para os certificados de Autoridade e uma para os CRL's.

De seguida esses certificados e CRL's são adicionados ao X509 Store Context, para uma posterior validação do Cartão de Cidadão.

Feito isto, são listadas todas as Open Sessions relativas a todas as slots dos smartcards, onde caso não existir nenhum Cartão de Cidadão inserido é terminado o processo e nenhuma reclamação dos pontos é feita. Por outro lado caso exista algum Cartão de Cidadão inserido, é obtida uma lista das slots disponíveis, mostrando os nomes completos das slots, na qual o jogador pode escolher qual é a sua identidade.

Após isto, é obtido o certificado referente a esse Cartão de Cidadão, passando pela validação na cadeia de certificados de confiança e verificando se não pertence a nenhuma lista de revogação de certificados. Caso este certificado não passe no processo de validação, é considerado Inválido e termina a reivindicação dos pontos, por outro lado, caso seja Válido, é pedido o PIN de Autenticação ao jogador de modo o poder ser feito o Login na plataforma de autenticação do Cartão de Cidadão para que possa ser possível a assinatura dos dados a serem enviados.

Nesta fase, a garantia da identidade dos pontos recolhidos até ao momento por um dado pseudónimo é feita, primeiramente, através da assinatura dos dados (ficheiro JSON com o pseudónimo e os pontos recolhidos) com a chave privada do Cartão de Cidadão.

De seguida é enviado para o Table Manager uma mensagem contendo os dados, o certificado do Cartão de Cidadão e a assinatura dos dados gerada. Já no Table Manager, é verificada mais uma vez a validação do certificado na cadeia de confiança (o certificado pode ter sido adulterado), e caso seja válido é feita a verificação da assinatura.

Na verificação da assinatura, é extraída a chave pública do certificado e é gerado um contexto de verificação dessa chave com a assinatura e os dados. Caso a verificação seja inválida este processo termina e não é feita a reivindicação dos pontos, caso contrário, procede-se à reivindicação dos pontos.

Em primeiro lugar, na reivindicação dos pontos, é verificada a existência do ficheiro JSON (***scores.json***), onde a relação pseudónimo-pontos, o certificado do Cartão de Cidadão e a assinatura dos dados serão armazenados, garantindo assim a identidade. Caso não exista, é criado e é adicionada à lista de scores a nova entrada.

Caso exista, passamos para a segunda iteração, onde é verificado se o certificado do Cartão de Cidadão já existe associado a um pseudónimo nos scores, onde caso não exista é criado um novo score.

Caso contrário, é verificado se o pseudónimo já existe com o certificado acima associado, e caso exista, é feito uma atualização no score correspondente.

Caso não exista, significa que se está a tentar assinar os dados de um pseudónimo com um Cartão de Cidadão já utilizado, e assim é terminado o processo de reivindicação não guardando os pontos.

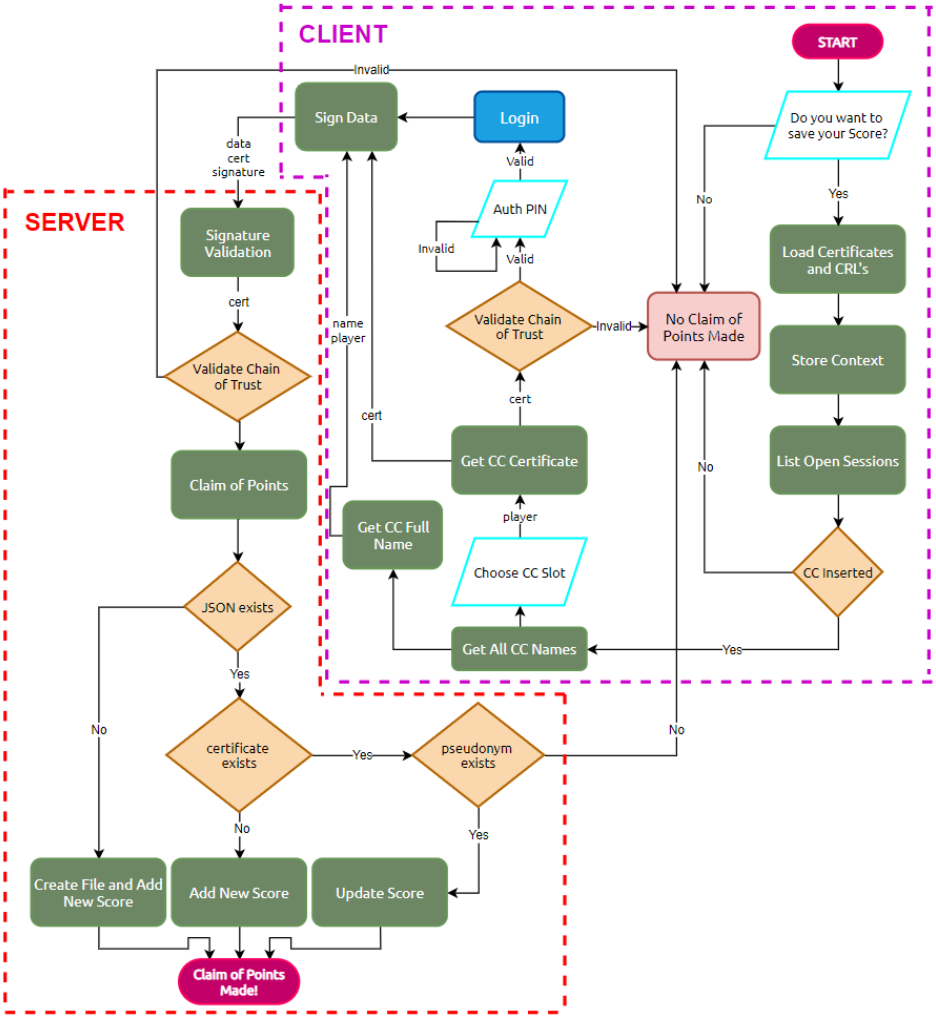

Figura 19: Reivindicação dos pontos com uma identidade fornecida por um Cartão de Cidadão

Figura 20: Estrutura do ficheiro JSON após a reivindicação dos pontos por um pseudónimo

2.9 Picking of stock tiles during a draw game

Em caso de o jogador não ter uma jogada possível e ainda existirem peças disponíveis no stock, o jogador irá retirar uma peça do stock, até encontrar uma com que possa jogar.

Ao retirar a peça do stock, ele vai pedir a todos os jogadores, por ordem inversa à cifração inicial do deck, a chave que decifra a peça que acabou de receber. Após ter todas

as chaves, ele decifra a sua peça chegando ao conjunto [i, pseudónimo].

Aqui faz um pedido ao servidor, que vai buscar o triplo guardado em pseudo_Table[i] e envia ao jogador que fez o pedido, ficando assim ele a saber qual é a peça que retirou do stock. Neste processo o servidor substitui este triplo por um triplo que tem o terceiro argumento a True, útil para poder não validar jogadas de peças que estão no stock.

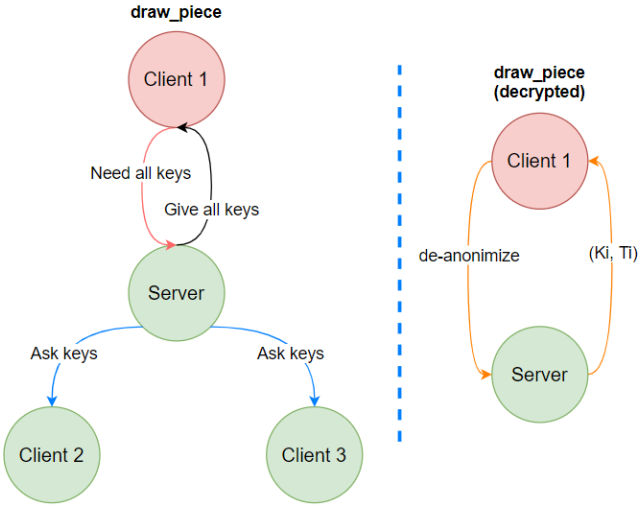

Figura 21: Picking of stock tiles during a draw game

3 Estrutura das mensagens JSON

Actions		

login	{
	"action": "login",
	"server_cert": <x509 certificate>
	}
		Cliente->Servidor com a sua chave pública cifrada com a chave pública recebida do Servidor na mensagem de "login"

req_login	{
	"action": "req_login",
	"pub_key": <chave pública cifrada>,
	"nickname": <pseudónimo escolhido>
	}
		Cliente->Servidor com a sua chave pública cifrada com a chave pública recebida do Servidor na mensagem de "login"

server_dh_negotiations	{
	"action": "server_dh_negotiations",
	"dh_key": <chave diffie-hellman cifrada>
	}
		Servidor->Cliente com a chave dh cifrada com a chave pública do cliente recebida na mensagem de "req$_{l}ogin$"

reply_dh_negotiations	{
	"action": "reply_dh_negotiations",
	"dh_key": <chave diffie-hellman cifrada>
	}
		Cliente->Servidor com a chave dh cifrada com a chave pública do Servidor

Apartir daqui os clientes já têm session key estabelecida com o Servidor e consequentemente todas as mensagens entre Cliente<->Servidor já vão com controlo de integridade, tendo sempre o {"hmac": hmac} e o {"sentby": <user>}.

As mensagens entre Cliente<->Cliente só vão ter este controlo de integridade após a fase "dh_sessions_finished", onde os clientes ficam com uma session key para cada outro cliente.

Actions		

receive_players_information	{
	"action": "receive_players_information",
	"players_info": <lista de listas [player.name, player.pub_key]>
	}
		Servidor->Cliente com algumas informações de todos os jogadores

dh_negotiations	{
	"action": "dh_negotiations",
	"dh_key": <chave diffie-hellman cifrada>
	}
		Cliente->Cliente com chave dh cifrada com a chave pública do cliente destino recebida na mensagem de "receive_players_information"

dh_sessions_finished	{
	"action": "dh_sessions_finished"
	}
		Cliente->Servidor para avisar que já tem todas as session keys combinadas com os outros clientes

waiting_for_host	{
	"action": "waiting_for_host"
	}
		Servidor->Cliente, host pode começar o jogo e os outros clientes esperam

start_game	{
	"action": "start_game"
	}
		Cliente(Host)->Servidor, host diz para começar

send_to	{
	"action": "encrypt_deck",
	"player_name": <player.name do destino>,
	"sent_by": <player.name>,
	"msg": <msg>
	}
		Cliente->Servidor, Cliente manda o Servidor redirecionar a 'msg' para o 'player_name' ('msg' está cifrada com a session key entre 'player_name' e 'sent_by')

send_to_unencrypted	{
	"action": "encrypt_deck",
	"player_name": <player.name do destino>,
	"sent_by": <player.name>,
	"msg": <msg>
	}
		Cliente->Servidor, Cliente manda o Servidor redirecionar a 'msg' para o 'player_name' ('msg' não está cifrada)

encrypt_deck	{
	"action": "encrypt_deck",
	"deck": <deck>
	}
		Servidor->Cliente(Host) na primeira chamada, e depois Cliente(Host)->Cliente através do "send_to_unencrypted"(pois não precisa de ser secreto)

encryption_done	{
	"action": "encryption_done",
	"deck": <deck>
	}
		Cliente->Servidor informa o Servidor que todos os jogadores já cifraram o deck

Actions		

get_initial_pieces	{
	"action":
	"get_initial_pieces",
	"deck": <deck>
	}
		Servidor->Cliente na primeira chamada, e depois Cliente->Cliente através do "send_to"(pois o deck tem de ir cifrado para ser secreto)

players_with_all_pieces	{
	"action":
	"players_with_all_pieces",
	"deck": <deck>
	}
		Cliente->Servidor com 5% de chance, quando deteta que já todos os clientes têm a mão cheia

get_bitcommitment	{
	"action":
	"get_bitcommitment"
	}
		Servidor->Cliente pedindo o bitcommitment da mão inicial

rcv_bitcommitment	{
	"action":
	"rcv_bitcommitment",
	"bitcommitment: <bitcommit usando sha256>
	}
		Cliente->Servidor enviando o bitcommitment pedido

rcv_info	{
	"action": "rcv_info",
	"all_bitcommitments: <lista de bitcommits>,
	"deck": <deck>
	}
		Servidor->Cliente enviando os bitcommitments de todos os clientes e o deck no seu estado atual

start_decrypt	{
	"action": "start_decrypt"
	}
		Cliente->Servidor dizendo que está pronto para a próxima fase

get_tilekeys	{
	"action": "get_tilekeys",
	"all_tilekeys": <lista de n listas (n = numplayers) com chaves que decifram peças em jogo>
	}
		Servidor->Cliente pedindo as chaves que decifram as peças que estão em jogo (que o cliente coloca na lista)

decrypt_hand	{
	"action": "decrypt_hand",
	"all_tilekeys": <lista de n listas (n = numplayers) com chaves que decifram peças em jogo>
	}
		Servidor->Cliente envia todas as chaves necessárias para o cliente decifrar a sua mão

Actions		

deanonymize	{
	"action": "deanonymize",
	"de_anon": <lista com 28 * [None] a ser preenchido por chaves públicas>
	}
		Cliente->Cliente através do "send_to"(pois o 'de_anon' tem de ir cifrado para ser secreto)

done_deanonymize	{
	"action":
	"done_deanonymize",
	"de_anon": <lista com as chaves públicas dos clientes>
	}
		Cliente->Servidor com 5% de chance quando deteta que já todos os outros adicionaram as suas chaves públicas

rcv_deanonymize	{
	"action":
	"rcv_deanonymize",
	"de_anon": <lista com os tuplos (Ki,Ti) cifrados com a chave pública>
	}
		Servidor->Cliente para poder decifar os índices onde colocou previamente uma chave pública e obter as peças da sua mão (verificando se o Servidor não mentiu através do cálculo de h(i,Ki,Ti) com o Ki fornecido)

ready_to_play	{
	"action": "ready_to_play"
	}
		Cliente->Servidor dizendo que está pronto para a próxima fase (começar a jogar)

rcv_game_propreties	{
	"action":
	"rcv_game_propreties",
	"deck": <deck>,
	"in_table": <peças na mesa (jogadas antes)>,
	"next_player":
	<player.name do jogar a jogar>
	}
		Servidor->Cliente atualizando o estado atual do jogo e dizendo qual é o jogador que tem a próxima ação

play_piece	{
	"action": "play_piece",
	"tile": <peças jogada>,
	"edge": <lado esquerdo ou direito da mesa>
	}
		Cliente->Servidor diz qual é a peça que quer jogar e qual o lado (esquerdo ou direito) em que quer por a peça

Actions		

pass_play	{
	"action": "pass_play"
	}
		Cliente->Servidor avisando que passou

validate	{
	"action": "validate",
	"status": <'ok' || 'not_ok'>
	}
		Cliente->Servidor dizendo ao servidor se protesta ou não (status 'not_ok' implica que a última peça jogada está, até prova em contrário, na mão do cliente que protestou)

draw_piece	{
	"action": "draw_piece",
	"deck": <deck atualizado>,
	"tile": <peça retirada do stock>
	}
		Cliente->Servidor dizendo ao servidor qual peça retirou do stock para começar o processo de decifrar e deanonimizar

get_drawntile_tilekeys	{
	"action":
	"get_drawntile_tilekeys",
	"tile": <peça retirada do stock>
	}
		Servidor->Cliente pede ao cliente a chave que decifra a peça retirada do stock

rcv_drawntile_tilekeys	{
	"action":
	"rcv_drawntile_tilekeys",
	"all_tilekeys": <lista com as n chaves que decifram a peça retirada do stock (n = numplayers)>
	}
		Servidor->Cliente envia ao cliente todas as chaves para ele decifrar a peça que retirou do stock

deanon_drawntile	{
	"action": "deanon_drawntile",
	"tile": <peça retirada do stock já decifrada>
	}
		Cliente->Servidor pede ao servidor para deanonimizar a peça

rcv_deanon_drawntile	{
	"action":
	"rcv_deanon_drawntile",
	"tuplo": (Ki,Ti)
	}
		Servidor->Cliente envia ao cliente o tuplo (Ki,Ti) para ele saber a sua peça e poder verificar com Ki. O cliente depois pode jogar, pegar uma peça do stock ou passar.

Actions		

end_game	{
	"action": "end_game ",
	"winner": <player.name>
	}
		Servidor->Cliente envia ao cliente nome do jogador que venceu o jogo (verifica após a última jogada ser validada por todos).

protest_all_tile	{
	"action": "protest_all_tile "
	}
		Servidor->Cliente pede ao cliente todas as chaves que usou para cifrar peças na Randomization Stage porque alguem protestou.

send_all_tile_keys	{
	"action":
	"send_all_tile_keys",
	"player.tilekeys": <lista de listas ([chave, ciphertext])>
	}
		Cliente->Servidor cliente envia todas as chaves que usou para encriptar todas as tiles no Randomization Stage.

disconnect	{
	"action": "disconnect",
	"msg": <string>
	}
		Servidor->Cliente desconecta os clientes.

4 Instruções de utilização

Para a nossa aplicação, existem dois modos de utilização:

1. Modo automático em que todo o jogo decorre sem o input do cliente.
2. Modo manual em que a cada jogada é requerido ao cliente a realização de uma de três ações já referidas previamente

Para cada um destes modos é possível o cliente escolher o seu nickname ou que seja gerado automáticamente pela aplicação.

Para correr o servidor é necessário executar:

```
$ python3 server.py [numclientes]
```

Para iniciar os clientes é necessário correr:

```
$ python3 client.py [--nick Nome] [--cheat]
```

Em que a flag -nick vai determinar a utilização de um nickname escolhido pelo utilizador e a flag -cheat vai determinar o grau de autonomia do cliente e a capacidade de fazer batota.

Bibliografia

[1] André Zúquete. Project Guide, 2020. URL http://sweet.ua.pt/andre.zuquete/Aulas/Seguranca/20-21/docs/project.pdf.
[Online; accessed 13-Outubro-2020].

[2] cryptography. Cryptography, 2021. URL https://github.com/pyca/cryptography.
[Online; accessed 15-Janeiro-2021].

[3] cryptography.io. Cryptography-X509, 2021. URL https://cryptography.io/en/2.7/x509/reference/.
[Online; accessed 15-Janeiro-2021].

[4] Jpfonseca. grabcrls.sh, 2021. URL https://github.com/Jpfonseca/Blockchain_auction_management/blob/master/src/grabcrls.sh.
[Online; accessed 15-Janeiro-2021].

[5] LudovicRousseau. PyKCS11, 2021. URL https://github.com/LudovicRousseau/PyKCS11.
[Online; accessed 15-Janeiro-2021].

[6] luminoso. PyKCS11, 2021. URL https://github.com/luminoso/chatsecure/blob/master/M2/PKCS11_Wrapper.py.
[Online; accessed 15-Janeiro-2021].

[7] PyKCS11. PyKCS11 samples codes, 2021. URL https://pkcs11wrap.sourceforge.io/api/samples.html.
[Online; accessed 20-Janeiro-2021].

[8] pyopenssl. PyOpenSSL, 2021. URL https://github.com/pyca/pyopenssl.
[Online; accessed 15-Janeiro-2021].

[9] A. Zúquete. *Segurança em redes informáticas*. FCA - Editora de Informática, 2013. ISBN 978-972-722-767-9.