

Segurança Informática e nas Organizações

Professor:
João Paulo Barraca

Gestão de direitos digitais

Hugo Paiva, 93195
Luís Valentim, 93989



DETI
Universidade de Aveiro
30-12-2020

Índice

1	Introdução	2
2	Funcionamento Geral	3
3	Cifra das comunicações	10
3.1	Protocolo	10
3.1.1	Diagrama	10
3.1.2	Implementação	10
3.2	Troca de chaves	13
3.2.1	Diagrama	13
3.2.2	Implementação	14
3.3	Cifra das comunicações e Validação da integridade	18
3.4	Efemeridade das chaves	21
3.5	Rotação de chaves	21
4	Autenticação e Isolamento	24
4.1	Autenticação do Servidor	24
4.1.1	Diagrama	24
4.1.2	Implementação	25
4.1.3	Infraestrutura de Chaves Públicas	25
4.1.4	Validação da Cadeia de Certificação	25
4.1.5	Autenticação Desafio-Resposta	28
4.2	Autenticação do Cliente	30
4.2.1	Diagrama	30
4.2.2	Implementação	31
4.3	Licenças e Autenticação das mesmas	37
4.3.1	Diagrama	37
4.3.2	Implementação	38
4.4	Proteção do conteúdo guardado no servidor	42
4.4.1	Encriptação	42
4.4.2	Desencriptação	43
5	Conclusão	45
6	Bibliografia	46

1 Introdução

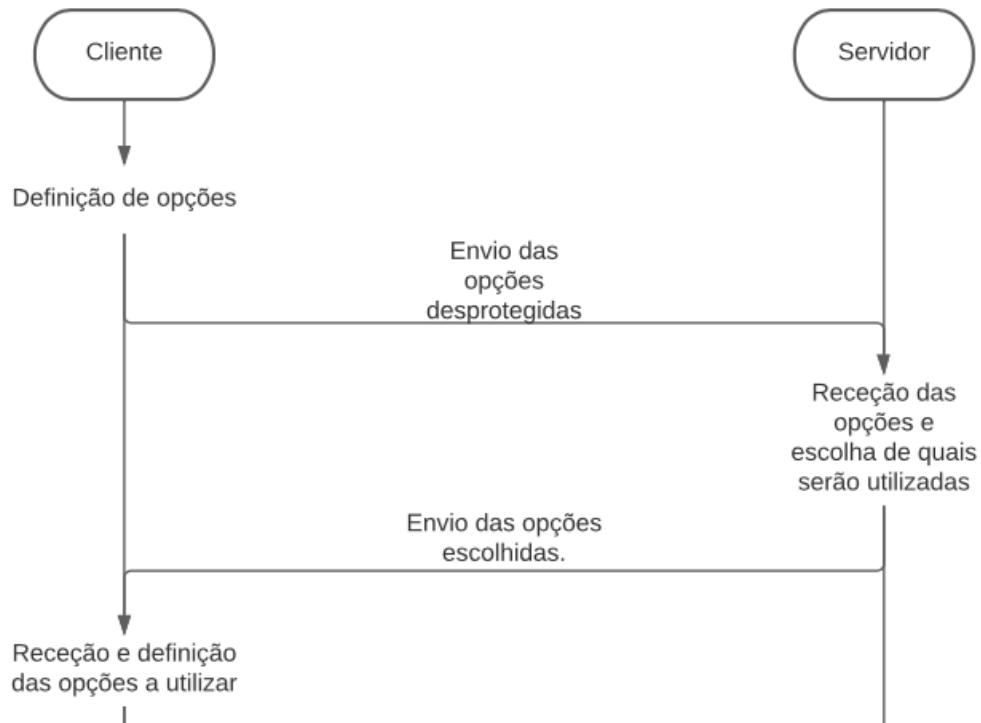
Este trabalho tem como objetivo o desenvolvimento de mecanismos de proteção de dados e comunicações num serviço de reprodução de música cliente-servidor, permitindo uma comunicação segura.

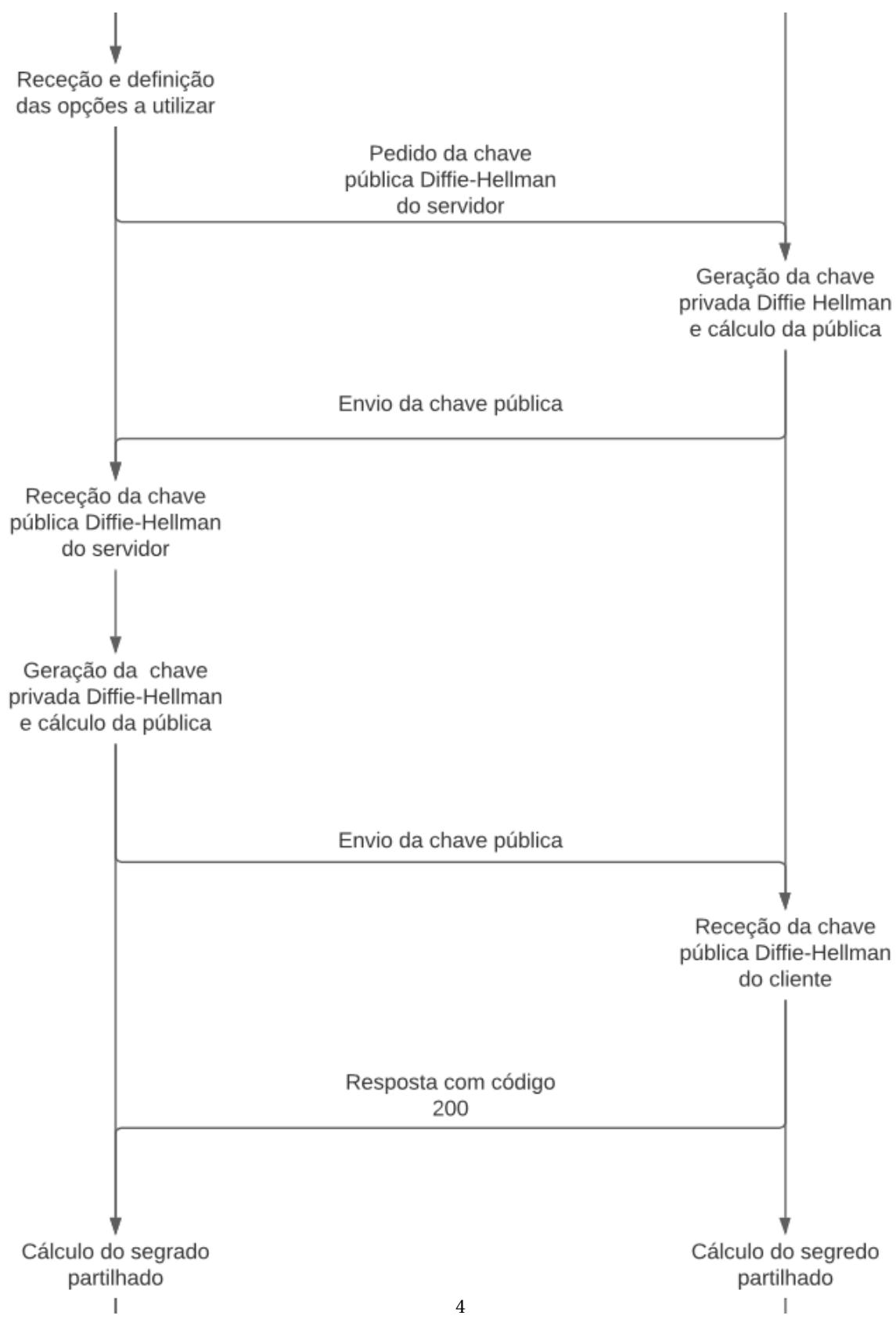
Neste sentido, foi fornecida a estrutura base do serviço com algumas funções relativas aos protocolos de comunicação desprotegidos já implementadas, permitindo a reprodução de músicas sem qualquer tipo de segurança ou autenticação, tendo sido necessário desenvolver as capacidades de proteção da troca de informação, de autenticação, etc.

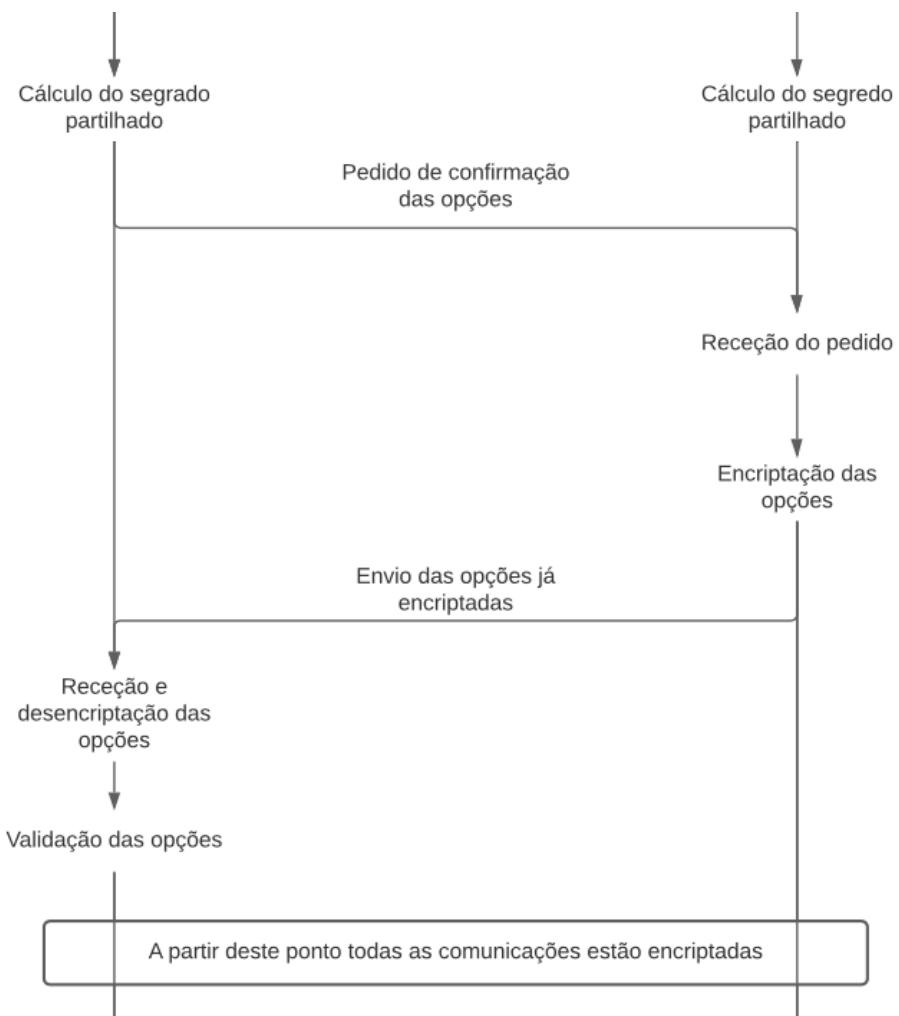
Este relatório visa explicar os protocolos escolhidos para cada problema, a razão das escolhas e a forma como os mesmos foram implementados, focando-se na descrição da implementação relativa à segurança.

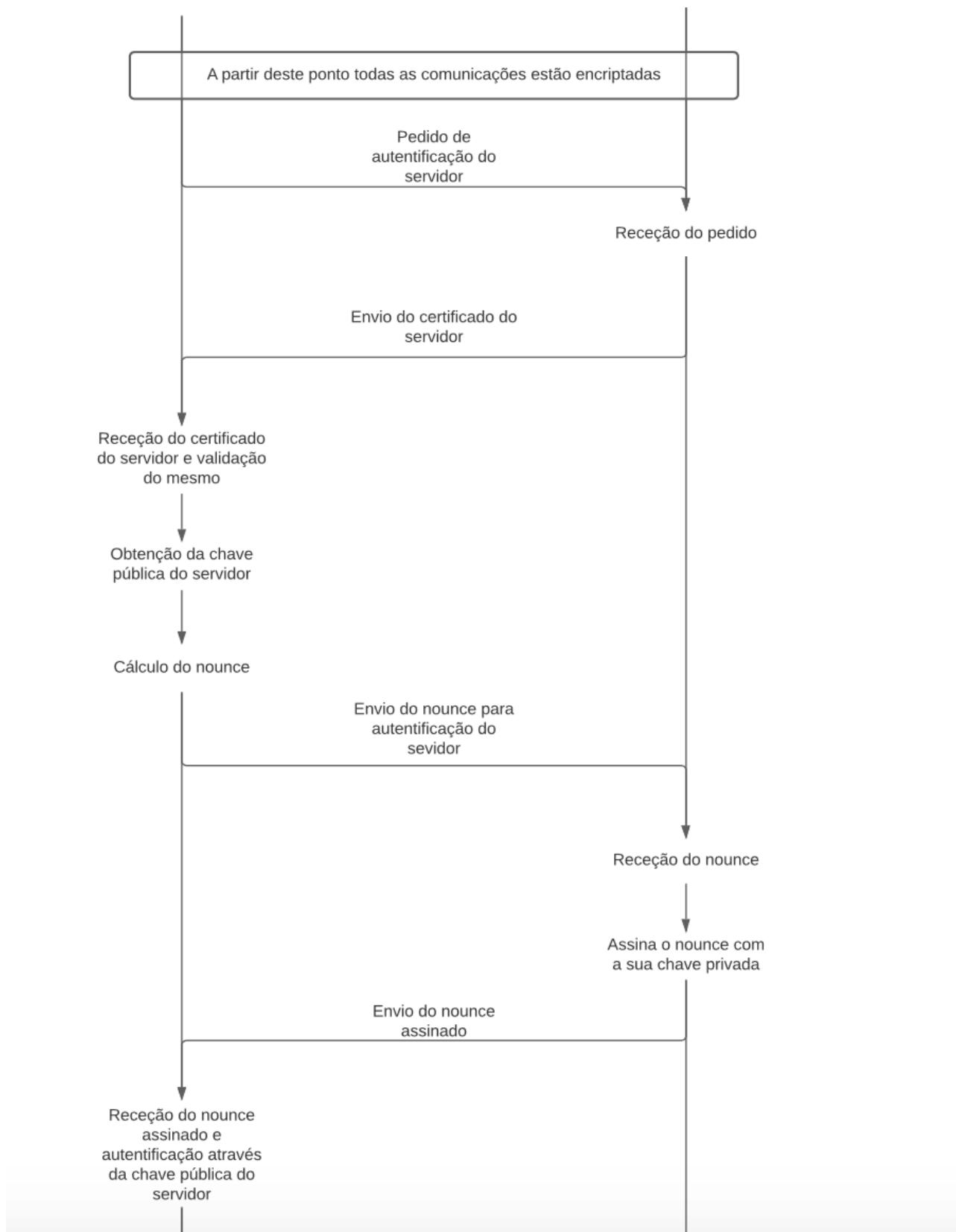
2 Funcionamento Geral

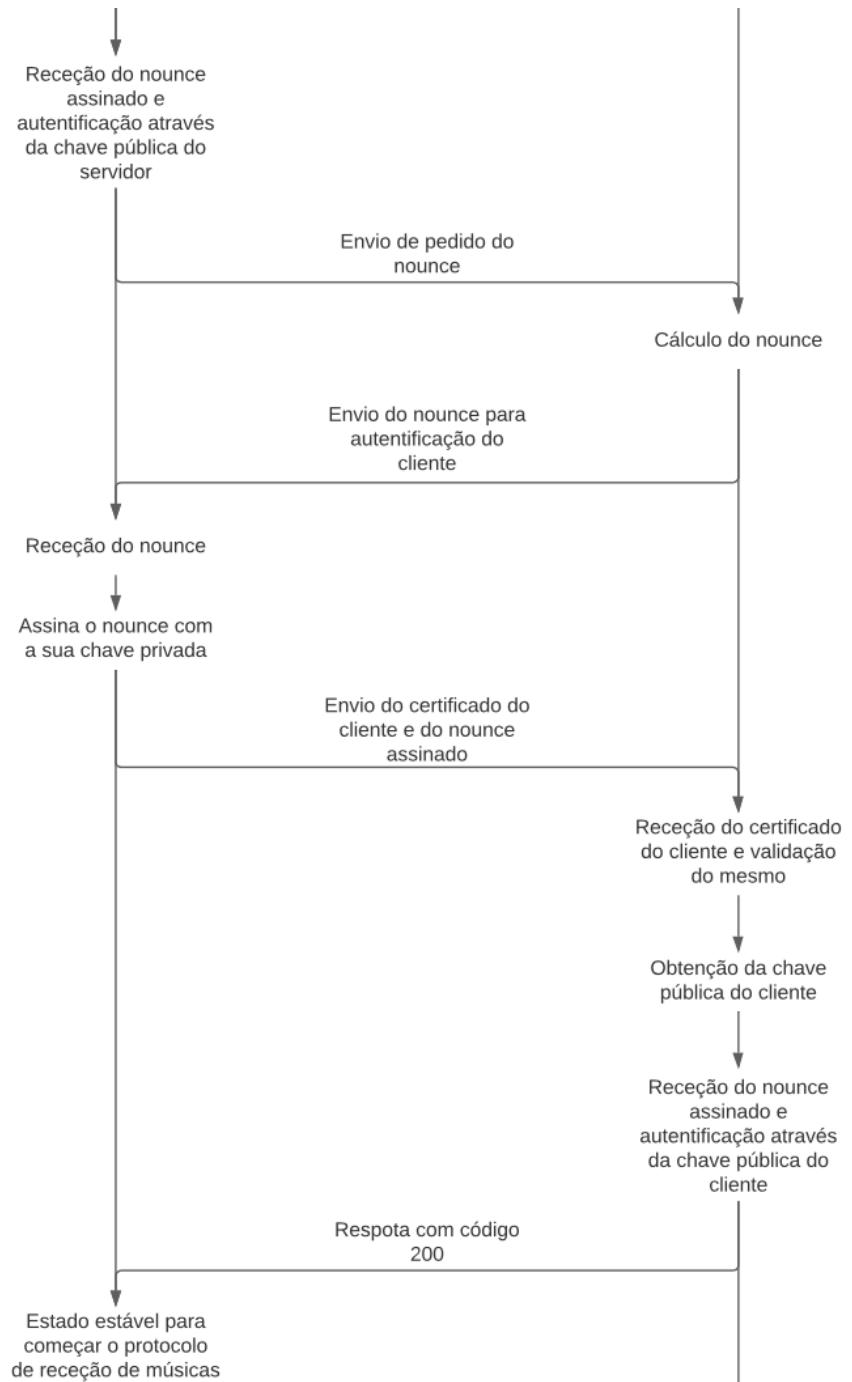
De seguida encontra-se na forma de diagrama o funcionamento geral da solução desenvolvida. Cada secção do mesmo será discutida posteriormente.

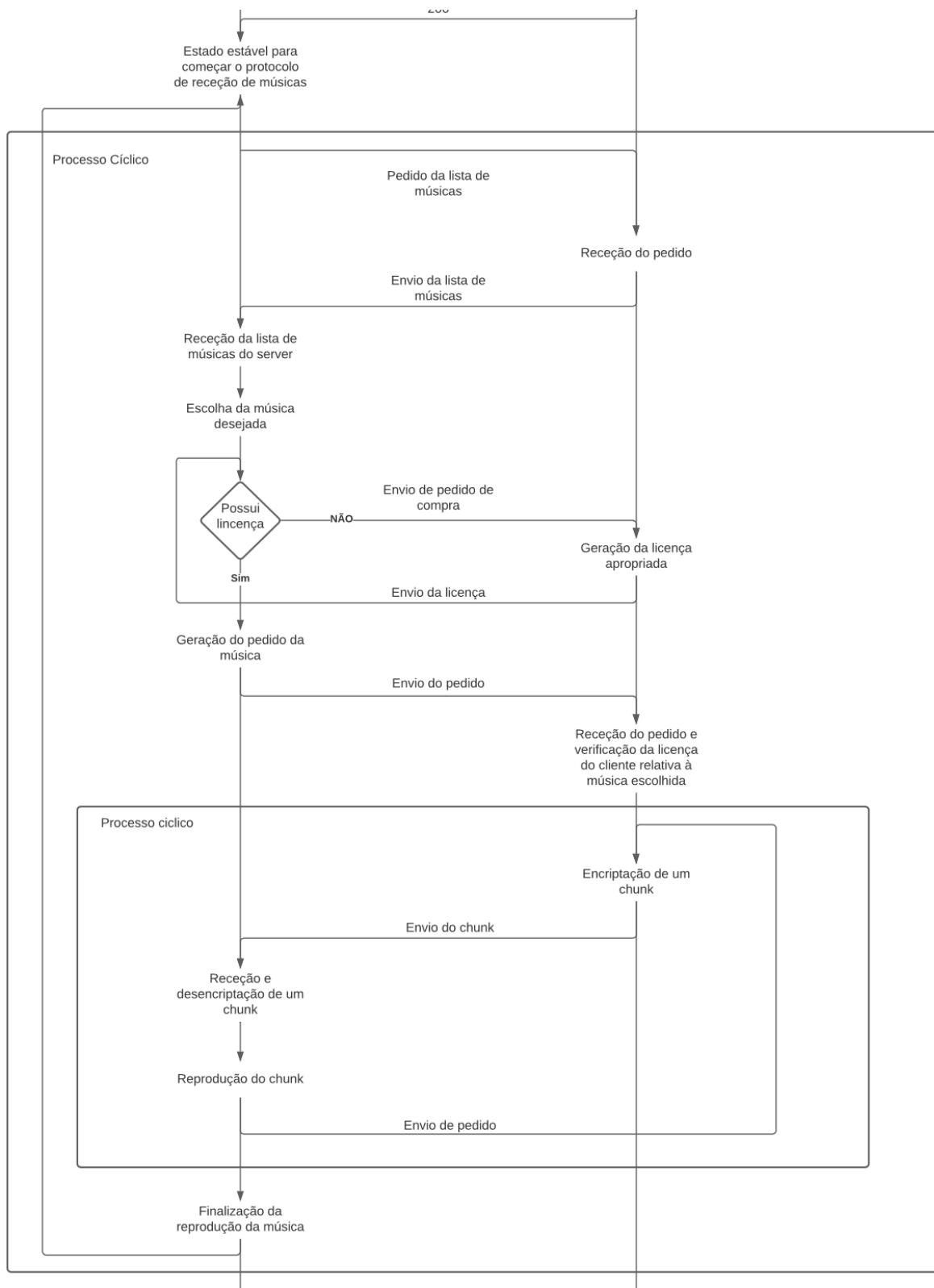












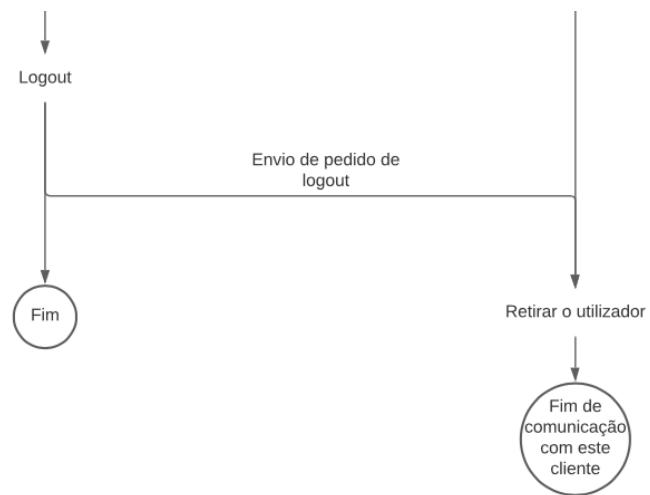
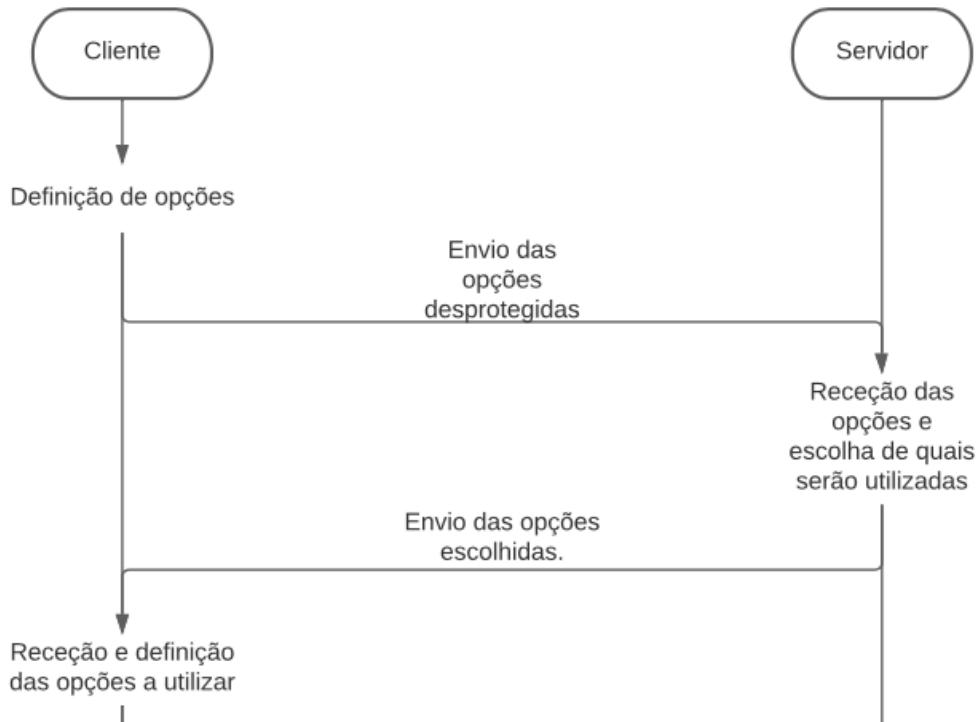


Figure 1: Funcionamento geral da solução

3 Cifra das comunicações

3.1 Protocolo

3.1.1 Diagrama



3.1.2 Implementação

Com o objetivo de cifrar as comunicações cliente-servidor, foi definido um protocolo que dita qual o tipo de cifra, o algoritmo que a mesma usa, o modo de cifra e qual o algoritmo de *hash* utilizado para gerar assinaturas e outros dados.

Optando pela Cifra Simétrica para as comunicações, foi escolhido um leque de opções de entre alguns modos de cifra, algoritmos de *hash* e algoritmos de cifra que posteriormente é utilizado para escolher de forma aleatória um deles para utilização na comunicação. As opções disponíveis são:

- **Algoritmos de Cifra** - AES e *Camellia*
- **Algoritmos de hash** - SHA256 e SHA3_256
- **Modos de Cifra** - CTR, OFB e CFB

Em ambas as aplicações, Cliente e Servidor, o leque de opções é definido da seguinte forma:

```
ALGORITHMS_OPTIONS = [algorithms.AES, algorithms.Camellia]
HASHES_OPTIONS = [hashes.SHA256, hashes.SHA3_256]
CIPHER_MODES_OPTIONS = [modes.CTR, modes.OFB, modes.CFB]
```

Figure 2: Leque de opções mencionado

Ao executar o cliente, a primeira comunicação com o servidor é o envio das opções. De entre o leque de opções de cada tipo, é gerado aleatoriamente o número de opções enviado para o servidor, podendo, por exemplo, para as opções de algoritmos de cifra, enviar ambos os algoritmos ou apenas um deles. Após esta decisão, são escolhidos aleatoriamente os algoritmos, sem repetição, até chegar ao número gerado anteriormente, podendo, portanto, enviar apenas o algoritmo *AES* ou apenas o *Camellia* ou ambos. Para os modos de cifra e algoritmos de *hash* a escolha é feita de maneira semelhante:

```
# Escolha de opções

# Opções de algoritmos de cifra
# 0. AES
# 1. Camellia
print("Generating Algorithms Options...")
algs = []
num_opt_alg = random.randint(1, len(ALGORITHMS_OPTIONS))
for i in range(0, num_opt_alg):
    rand = random.randint(0, len(ALGORITHMS_OPTIONS)-1)
    while rand in algs:
        rand = random.randint(0, len(ALGORITHMS_OPTIONS)-1)
    algs.append(rand)

# Opções de hash
# 0. SHA256
# 1. SHA3_256
print("Generating Hash Options...")
hashes = []
num_opt_hash = random.randint(1, len(HASHES_OPTIONS))
for i in range(0, num_opt_hash):
    rand = random.randint(0, len(HASHES_OPTIONS)-1)
    while rand in hashes:
        rand = random.randint(0, len(HASHES_OPTIONS)-1)
    hashes.append(rand)

# Opções de modo de cifra
# 0.CTR
# 1.OFB
# 2.CFB
print("Generating Cipher Mode Options...")
modes = []
num_opt_modes = random.randint(1, len(CIPHER_MODES_OPTIONS))
for i in range(0, num_opt_modes):
    rand = random.randint(0, len(CIPHER_MODES_OPTIONS)-1)
    while rand in modes:
        rand = random.randint(0, len(CIPHER_MODES_OPTIONS)-1)
    modes.append(rand)

available_options = {'algorithms': algs,
                     'hashes': hashes, 'modes': modes}
```

Figure 3: Escolha do leque de opções para enviar ao servidor

Ao enviar este leque de opções, o cliente identifica-se através do *header Authorization* com um *uuid* gerado previamente, passando a utilizá-lo em todas as comunicações como seu identificador. O servidor, ao receber estas opções, escolhe aleatoriamente quais pretende utilizar e responde ao cliente com essa informação, ficando essa informação guardada em ambas as aplicações:

```
def do_post_protocols(self, request):
    """
    Receção dos protocolos para a cifra
    """
    response = json.loads(request.content.read())

    selected_algorithm = random.choice(response['algorithms'])
    selected_hash = random.choice(response['hashes'])
    selected_mode = random.choice(response['modes'])

    id = request.getHeader('Authorization')

    CLIENT_INFO[id] = {}
    CLIENT_INFO[id]['options'] = {}
    CLIENT_INFO[id]['options']['selected_algorithm'] = selected_algorithm
    CLIENT_INFO[id]['options']['selected_hash'] = selected_hash
    CLIENT_INFO[id]['options']['selected_mode'] = selected_mode

    request.setResponseCode(200)
    request.responseHeaders.addRawHeader(
        b"content-type", b"application/json")
    return json.dumps(
        {
            'selected_algorithm': selected_algorithm,
            'selected_hash': selected_hash,
            'selected_mode': selected_mode,
        },
    ).encode('latin')
```

Figure 4: Escolha das opções finais por parte do servidor

```
print("\nContacting Server\n")
print("Sending all Cipher related options")

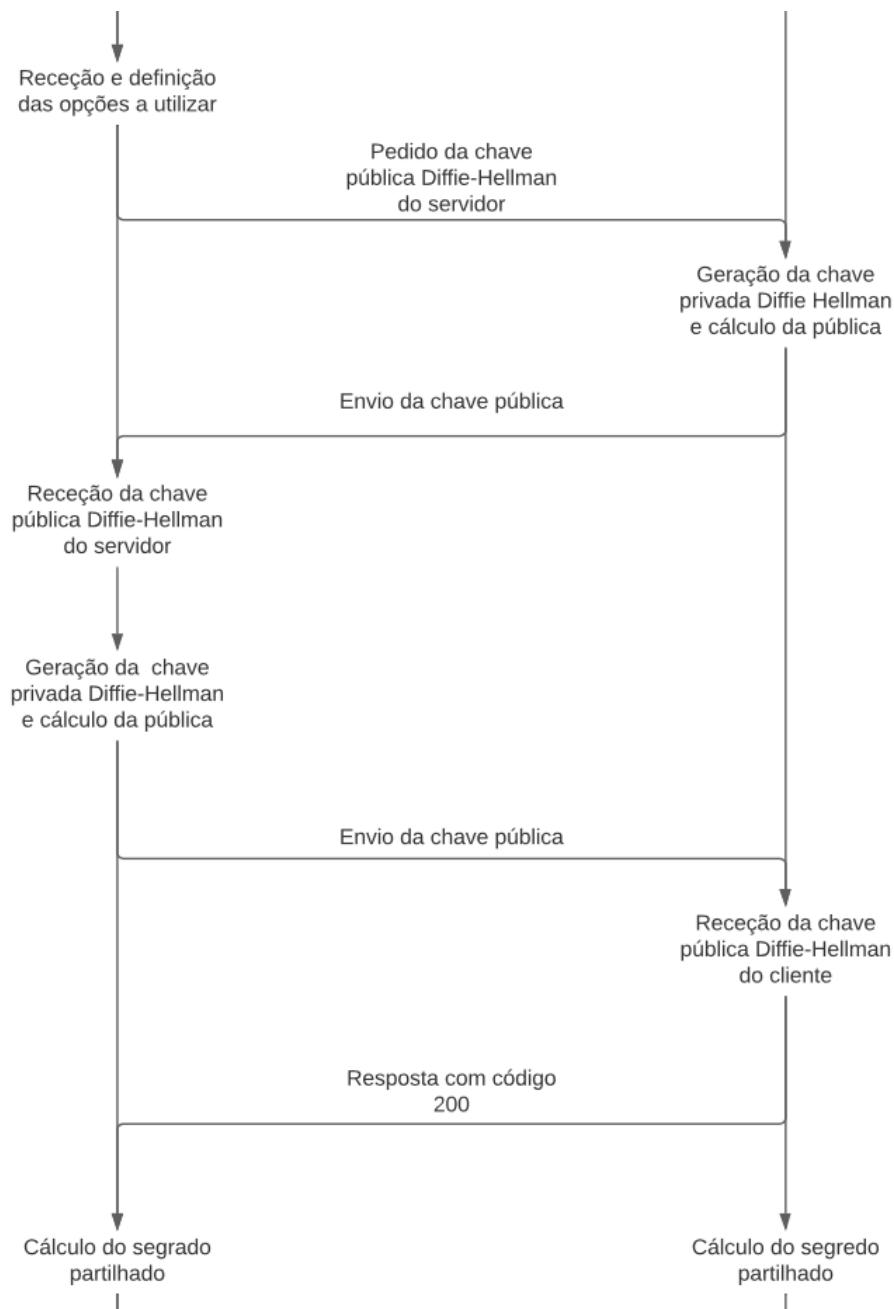
# Envio das opções relativas à cifra
req = requests.post(f'{SERVER_URL}/api/protocols', data=json.dumps(
    available_options.encode('latin')), headers={"content-type": "application/json", "Authorization": self.id})

if req.status_code != 200:
    print('\u27e8\u27e8\u27e8\u27e8'+f'Error sending Cipher related options'+'\u27e9\u27e9\u27e9\u27e9')
    quit()
else:
    response = req.json()
    self.selected_algorithm = response['selected_algorithm']
    self.selected_hash = response['selected_hash']
    self.selected_mode = response['selected_mode']
```

Figure 5: Informação do protocolo guardada no cliente

3.2 Troca de chaves

3.2.1 Diagrama



3.2.2 Implementação

Para ser possível cifrar as mensagens após a troca do protocolo é necessário a geração e troca de chaves simétricas. Como a comunicação está em aberto foi necessário recorrer a um algoritmo que permite a geração de um segredo partilhado em um canal inseguro, como é o caso, sem problemas de segurança, tal como foi abordado nas aulas teóricas com o algoritmo *Diffie-Hellman*. Para evitar os problemas de um ataque *Man-In-The-Middle* que este algoritmo tem, foi utilizada a forma efêmera do mesmo, disponível no *cryptography.io*.

Foram trocadas duas chaves, uma para ser utilizada na criptografia das comunicações e outras para ser utilizada na geração das assinaturas das comunicações com base em *HMAC*. Para gerar as chaves foram criadas duas funções no cliente utilizando as implementações do *cryptography.io*:

```
def dh_digest_key(self):
    ''' Troca de novas chaves simétricas para utilização ao criar digests nas comunicações '''
    private_key = ec.generate_private_key(ec.SECP384R1())
    req = requests.get(f'{SERVER_URL}/api/digest_key', headers={"Authorization": self.id})
    if req.status_code == 200:
        response = req.json()

        if self.message_key and self.digest_key:
            digest = response['digest'].encode('latin')
            server_public_key = response['key'].encode('latin')
            iv = response['iv'].encode('latin')

            if self.verify_digest(self.digest_key, self.selected_hash, server_public_key, digest):
                server_public_key, decryptor_var = self.decryptor(
                    self.selected_algorithm, self.selected_mode, self.message_key, server_public_key, iv)

                server_public_key = binascii.a2b_base64(server_public_key)
            else:
                print('\033[31m'+ "Data integrity of communication violated"+'\033[0m')
                quit()
        else:
            server_public_key = binascii.a2b_base64(
                response['key'].encode('latin'))

        loaded_public_key = serialization.load_pem_public_key(
            server_public_key,)

        shared_key = private_key.exchange(ec.ECDH(), loaded_public_key)

        digest_key = HKDF(
            algorithm=hashes.SHA256(),
            length=32,
            salt=None,
            info=b'handshake data',).derive(shared_key)

    else:
        print('\033[31m'+ "Error trading DH symmetric keys for communication digests"+'\033[0m')
        quit()

    public_key = private_key.public_key()

    serialized_public = public_key.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo)

    serialized_public = binascii.b2a_base64(serialized_public)

    if self.message_key and self.digest_key:
        serialized_public, encryptor_cypher, iv = self.encryptor(self.selected_algorithm, self.selected_mode, self.message_key, serialized_public)

        serialized_public_digest = self.digest(self.digest_key,
                                              self.selected_hash, serialized_public)

        response = {'key': serialized_public.decode('latin'), 'digest': serialized_public_digest.decode('latin'), 'iv': iv.decode('latin')}
    else:
        response = {'key': serialized_public.decode('latin').strip()}

    req = requests.post(f'{SERVER_URL}/api/digest_key', data=json.dumps(
        response).encode('latin'), headers={"content-type": "application/json", "Authorization": self.id})

    if req.status_code != 200:
        print('\033[31m'+ "Error trading DH symmetric keys for communication digests"+'\033[0m')
        quit()

    self.digest_key = digest_key
```

Figure 6: Geração do segredo partilhado para utilizar no *HMAC*, no cliente

```

def dh_message_key(self):
    """ Troca de novas chaves simétricas para cifrar as comunicações """
    private_key = ec.generate_private_key(ec.SECP384R1())

    req = requests.get(f'{SERVER_URL}/api/key', headers={"Authorization": self.id})
    if req.status_code == 200:
        response = req.json()

        if self.message_key and self.digest_key:
            digest = response['digest'].encode('latin')
            server_public_key = response['key'].encode('latin')
            iv = response['iv'].encode('latin')
            if self.verify_digest(self.digest_key, self.selected_hash, server_public_key, digest):
                server_public_key, decryptor_var = self.decryptor(
                    self.selected_algorithm, self.selected_mode, self.message_key, server_public_key, iv)

                server_public_key = binascii.a2b_base64(server_public_key)
            else:
                print('\033[31m'+Data integrity of communication violated'+'\033[0m')
                quit()
        else:
            server_public_key = binascii.a2b_base64(
                response['key'].encode('latin'))

        loaded_public_key = serialization.load_pem_public_key(
            server_public_key,)

        shared_key = private_key.exchange(ec.ECDH(), loaded_public_key)

        message_key = HKDF(
            algorithm=hashes.SHA256(),
            length=32,
            salt=None,
            info=b'handshake data').derive(shared_key)
    else:
        print('\033[31m'+Error trading DH symmetric keys for communication'+'\033[0m')
        quit()

    public_key = private_key.public_key()

    serialized_public = public_key.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo)

    serialized_public = binascii.b2a_base64(serialized_public)

    if self.message_key and self.digest_key:
        serialized_public, encryptor_cypher, iv = self.encryptor(self.selected_algorithm, self.selected_mode, self.message_key, serialized_public)

        serialized_public_digest = self.digest(self.digest_key,
                                              self.selected_hash, serialized_public)

        response = {'key': serialized_public.decode('latin'), 'digest': serialized_public_digest.decode('latin'), 'iv': iv.decode('latin')}
    else:
        response = {'key': serialized_public.decode('latin').strip()}

    req = requests.post(f'{SERVER_URL}/api/key', data=json.dumps(
        response), encode('latin'), headers={"content-type": "application/json", "Authorization": self.id})

    if req.status_code != 200:
        print('\033[31m'+Error trading DH symmetric keys for communication'+'\033[0m')
        quit()
    self.message_key = message_key

```

Figure 7: Geração do segredo partilhado para utilizar na criptografia das comunicações, no cliente

Do lado do servidor o processo é relativamente parecido sendo que, de acordo com a utilização do cliente, primeiro é gerada uma chave pública para ser enviada e posteriormente é recebida a chave pública do cliente, gerando o segredo partilhado que será a chave a utilizar:

```
def get_key(self, request):
    ''' Envio de chave pública do servidor para Diffie Hellman '''
    private_key = ec.generate_private_key(ec.SECP384R1())

    id = request.getHeader('Authorization')

    if request.path == b'/api/digest_key':
        CLIENT_INFO[id]['dh_digest_private_key'] = private_key
    elif request.path == b'/api/key':
        CLIENT_INFO[id]['dh_private_key'] = private_key
    else:
        request.setResponseCode(401)
        request.responseHeaders.addRawHeader(b"content-type", b"text/plain")
        return b''

    public_key = private_key.public_key()

    serialized_public = public_key.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo)

    serialized_public = binascii.b2a_base64(serialized_public)

    if 'message_key' in CLIENT_INFO[id] and 'digest_key' in CLIENT_INFO[id]:
        serialized_public, encryptor_cypher, iv = encryptor(CLIENT_INFO[id]['options']['selected_algorithm'], CLIENT_INFO[id]
                                                          ['options']['selected_mode'], CLIENT_INFO[id]['message_key'], serialized_public)

        serialized_public_digest = generate_digest(CLIENT_INFO[id]['digest_key'],
                                                    CLIENT_INFO[id]['options']['selected_hash'], serialized_public)

        response = {'key': serialized_public.decode('latin'), 'digest': serialized_public_digest.decode('latin'), 'iv': iv.decode('latin')}

    else:
        response = {'key': serialized_public.decode('latin')}

    request.responseHeaders.addRawHeader(
        b"content-type", b"application/json")
    return json.dumps(response).encode('latin')
```

Figure 8: Geração da chave pública para enviar ao cliente, no servidor

```

def post_key(self, request):
    ''' Receção de chave pública do cliente para Diffie Hellman '''
    response = json.loads(request.content.read())
    id = request.getHeader('Authorization')

    if 'message_key' in CLIENT_INFO[id] and 'digest_key' in CLIENT_INFO[id]:
        digest = response['digest'].encode('latin')
        client_public_key = response['key'].encode('latin')
        iv = response['iv'].encode('latin')

        if verify_digest(CLIENT_INFO[id]['digest_key'], CLIENT_INFO[id]['options']['selected_hash'], client_public_key, digest):
            client_public_key, decryptor_var = decryptor(
                CLIENT_INFO[id]['options']['selected_algorithm'], CLIENT_INFO[id]
                ['options']['selected_mode'], CLIENT_INFO[id]['message_key'], client_public_key, iv)
            client_public_key = binascii.a2b_base64(client_public_key)
        else:
            print('\x033[31m'+f'Data integrity of communication violated for user {id}'+'\x033[0m')
            request.setResponseCode(401)
            request.responseHeaders.addRawHeader(b"content-type", b"text/plain")
            return b''

    else:
        client_public_key = binascii.a2b_base64(
            response['key'].encode('latin'))

    loaded_public_key = serialization.load_pem_public_key(
        client_public_key,)

    shared_key = None

    if request.path == b'/api/digest_key' and 'dh_digest_private_key' in CLIENT_INFO[id]:
        shared_key = CLIENT_INFO[id]['dh_digest_private_key'].exchange(
            ec.ECDH(), loaded_public_key)
    elif request.path == b'/api/key' and 'dh_private_key' in CLIENT_INFO[id]:
        shared_key = CLIENT_INFO[id]['dh_private_key'].exchange(
            ec.ECDH(), loaded_public_key)
    else:
        request.setResponseCode(401)
        request.responseHeaders.addRawHeader(b"content-type", b"text/plain")
        return b''

    MESSAGE_KEY = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=None,
        info=b'handshake data',).derive(shared_key)

    if request.path == b'/api/digest_key':
        CLIENT_INFO[id]['digest_key'] = MESSAGE_KEY

    elif request.path == b'/api/key':
        CLIENT_INFO[id]['message_key'] = MESSAGE_KEY

    request.setResponseCode(200)
    request.responseHeaders.addRawHeader(b"content-type", b"text/plain")
    return b''

```

Figure 9: Geração do segredo partilhado de acordo com a chave pública recebida do cliente, no servidor

Visto que a troca de chaves também irá acontecer já depois do protocolo da cifra estar definido, como será explicado posteriormente, estas funções permitem enviar a chave pública em aberto quando o protocolo ainda não está definido mas, também enviam cifrada, caso o protocolo e as chaves já estejam definidas. Mais uma vez, toda esta troca tem como base o *uuid* do cliente referido anteriormente e que é utilizado pelo servidor para guardar as informações necessárias no dicionário *CLIENT_INFO*.

3.3 Cifra das comunicações e Validação da integridade

Tendo o protocolo e as chaves trocadas, o servidor e o cliente passam a comunicar de forma encriptada. Em ambas as aplicações foram criadas funções para encriptar, desencriptar e validar a integridade de mensagens. A implementação da função de encriptação do lado do cliente é a única que difere da do lado do servidor pois não necessita de permitir encriptar novamente a partir do *encryptor* de uma encriptação anterior, sendo que isto é utilizado para permitir o fluxo correto do *iv* e outras propriedades, dependendo do modo de cifra. Esta diferença cai no facto de que o cliente apenas cifra mensagens que não são divididas em vários blocos, ao contrário do servidor que cifra os vários blocos das músicas, necessitando de guardar e manter o fluxo das propriedades daquela cifra em específico para não ser necessário enviar um novo *iv* em todos os blocos encriptados da música.

As implementações das funções são as seguintes:

```
def decryptor(self, algorithm, mode, key, data, iv=None, decryptor=None):
    ''' Decifra de dados de acordo com o protocolo escolhido '''
    if decryptor:
        data = decryptor.update(data)
        return data, decryptor
    else:
        cipher = Cipher(ALGORITHMS_OPTIONS[algorithm](
            key), CIPHER_MODES_OPTIONS[mode](iv))
        decryptor = cipher.decryptor()
        data = decryptor.update(data)
        return data, decryptor
```

Figure 10: Função de desencriptação, igual em ambas as aplicações

```
def digest(self, key, hash, data):
    ''' Geração de digest para verificação de integridade com recurso a HMAC '''
    h = hmac.HMAC(key, HASHES_OPTIONS[hash]())
    h.update(data)
    return h.finalize()
```

Figure 11: Função que gera uma espécie de *digest*, uma assinatura, utilizando uma chave, permitindo validar a integridade e autenticidade de dados, tal como referido na especificações do *HMAC*

```
def verify_digest(self, key, hash, data, digest):
    ''' Verificação da integridade com recurso a HMAC e ao digest gerado previamente '''
    h = hmac.HMAC(key, HASHES_OPTIONS[hash]()
    h.update(data)
    try:
        h.verify(digest)
        return True
    except InvalidSignature:
        return False
```

Figure 12: Função que permite verificar a assinatura dos dados passados, passando também a respectiva assinatura

```
def encryptor(self, algorithm, mode, key, data):
    ''' Cifra de dados de acordo com o protocolo escolhido '''
    iv = os.urandom(ALGORITHMS_OPTIONS[algorithm].block_size // 8)
    cipher = Cipher(ALGORITHMS_OPTIONS[algorithm](
        key), CIPHER_MODES_OPTIONS[mode](iv))
    encryptor = cipher.encryptor()
    ct = encryptor.update(data) + encryptor.finalize()
    return ct, encryptor, iv
```

Figure 13: Função que permite encriptar dados no cliente

```
def encryptor(algorithm, mode, key, data, encryptor=None, iv=None, last=True):
    ''' Cifra de dados de acordo com o protocolo escolhido '''
    if encryptor:
        ct = encryptor.update(data)
        if last:
            ct += encryptor.finalize()
        return ct, encryptor, iv
    else:
        iv = os.urandom(ALGORITHMS_OPTIONS[algorithm].block_size // 8)
        cipher = Cipher(ALGORITHMS_OPTIONS[algorithm](
            key), CIPHER_MODES_OPTIONS[mode](iv))
        encryptor = cipher.encryptor()
        ct = encryptor.update(data)
        if last:
            ct += encryptor.finalize()
        return ct, encryptor, iv
```

Figure 14: Função que permite encriptar dados no servidor

Aliando todas estas funções é possível cifrar as comunicações, tal como é feito para verificar que o protocolo da cifra, escolhido inicialmente, não foi forjado entretanto:

```
# Verificação da opções relativas à cifra
req = requests.get(f'{SERVER_URL}/api/protocols', headers={"Authorization": self.id})
if req.status_code != 200:
    print("\033[31m"+Error getting Cipher related options"+\033[0m")
    quit()
else:
    response = req.json()
    digest = response['digest'].encode('latin')
    options = response['options'].encode('latin')
    iv = response['iv'].encode('latin')

    if self.verify_digest(self.digest_key, self.selected_hash, options, digest):
        options, decryptor_var = self.decryptor(
            self.selected_algorithm, self.selected_mode, self.message_key, options, iv)

        options = json.loads(options.decode('latin'))
        if(self.selected_algorithm != options['selected_algorithm'] or self.selected_hash != options['selected_hash'] or self.selected_mode != options['selected_mode']):
            print("\033[31m"+Cipher related options different from the server"+\033[0m")
            quit()
        else:
            print('\033[1m'+Protocol integrity verified"+\033[0m")
    else:
        print("\033[31m"+Data integrity of communication violated"+\033[0m")
        quit()
```

Figure 15: Verificação no cliente que o protocolo da cifra, trocado anteriormente, não foi forjado

Para a verificação anterior, o cliente pede ao servidor o protocolo da cifra que o mesmo tem guardado. O envio do protocolo é feito da seguinte forma:

```
def do_get_protocols(self, request):
    ''' Envio dos protocolos para a cifra '''
    id = request.getHeader('Authorization')

    options = json.dumps(
        CLIENT_INFO[id]['options']).encode('latin')

    options, encryptor_cypher, iv = encryptor(CLIENT_INFO[id]['options']['selected_algorithm'], CLIENT_INFO[id]
                                                ['options']['selected_mode'], CLIENT_INFO[id]['message_key'], options)

    options_digest = generate_digest(CLIENT_INFO[id]['digest_key'],
                                     CLIENT_INFO[id]['options']['selected_hash'], options)

    request.responseHeaders.addRawHeader(
        b"content-type", b"application/json")
    return json.dumps({'options': options.decode('latin'), 'digest': options_digest.decode('latin'), 'iv': iv.decode('latin')}).encode('latin')
```

Figure 16: Envio do protocolo da cifra definido anteriormente e guardado pelo servidor

Todas as comunicações a partir deste momento são realizadas de forma similar, permitindo-as ser **cifradas** e seguras relativamente a atacantes durante esta sessão.

Para aumentar a segurança das comunicações e evitar a reutilização de chaves, foram também implementadas duas soluções:

3.4 Efemeridade das chaves

- **A primeira incide na implementação da efemeridade das chaves.** Para isto, sempre que o cliente termina a aplicação, novamente com recurso ao *uuid* gerado anteriormente, é enviado um pedido ao servidor, onde são eliminadas todas as informações associadas ao cliente geradas nesta sessão, com exceção das licenças, que serão discutidas posteriormente:

```
print("...")  
  
while True:  
    selection = input("Select a media file number (q to quit). If a song with no access is selected, a new license will be requested: ")  
    if selection.strip() == 'q':  
        req = requests.get(f'{SERVER_URL}/api/logout', headers={"Authorization": self.id})  
        if req.status_code == 200:  
            print("All done!")  
            sys.exit(0)
```

Figure 17: Envio do pedido de *logout* ao selecionar a opção de sair da aplicação no cliente

```
def logout(self, request):  
    ''' Término de sessão de um utilizador, eliminando os seus dados (exceto licenças) '''  
    id = request.getHeader('Authorization')  
    del CLIENT_INFO[id]  
    request.setResponseCode(200)  
    request.responseHeaders.addRawHeader(b"content-type", b"text/plain")  
    return b''
```

Figure 18: Informação associada ao cliente eliminada ao receber o pedido de *logout* no servidor

3.5 Rotação de chaves

- **A segunda incide na rotação das chaves** utilizadas na cifra à medida que o cliente obtém novos blocos da música. Para tal, foi utilizada uma solução baseada no algoritmo **Double Ratchet** utilizado em aplicações como o *Signal*. Em suma, o funcionamento deste algoritmo dita que cada mensagem tem de ter uma chave diferente, utilizando para o efeito funções de derivação de chaves, evitando a possibilidade de a partir de uma chave gerar chaves anteriores devido às propriedades destas funções. Além disso, sempre que possível, deverá fazer-se uma nova troca de chaves utilizando o algoritmo *Diffie-Helman* já abordado anteriormente. Isto é feito para evitar o caso em que alguma chave é interceptada e tem-se acesso à função de derivação, o que permitiria gerar todas as chaves a partir daí.

A solução aplicada foi a utilização de uma simples função de síntese aplicada às chaves sempre que se recebe um novo bloco de música e, a cada 5 blocos recebidos, a geração de novas chaves utilizando o algoritmo *Diffie-Helman* mas, utilizando as comunicações cifradas, mantendo todo o secretismo e segurança do início ao fim do programa. A implementação foi a seguinte:

```

# Get data from server and send it to the ffplay stdin through a pipe
for chunk_id in range(media_item['chunks']):
    req = requests.get(
        f'{SERVER_URL}/api/download?id={media_item["id"]}&chunk={chunk_id}', headers={"Authorization": self.id})
    if req.status_code != 200:
        print('\033[31m'+"Error getting chunk of song"+'\033[0m')
        quit()
    chunk = req.json()

    if chunk_id == 0:
        iv = chunk['iv'].encode('latin')

    digest = chunk['digest'].encode('latin')

    data = chunk['data'].encode('latin')

    if not self.verify_digest(self.digest_key, self.selected_hash, data, digest):
        print('\033[31m'+"Data integrity of communication violated"+'\033[0m')
        quit()

    if decryptor_var:
        data, decryptor_var = self.decryptor(
            self.selected_algorithm, self.selected_mode, self.message_key, data, decryptor=decryptor_var)
    else:
        data, decryptor_var = self.decryptor(
            self.selected_algorithm, self.selected_mode, self.message_key, data, iv)

    data = binascii.a2b_base64(data)

    if chunk_id % 5 == 0 and chunk_id != 0:
        self.dh_digest_key()
        self.dh_message_key()
    else:
        self.message_key = self.simple_digest(
            self.selected_hash, self.message_key)
        self.digest_key = self.simple_digest(
            self.selected_hash, self.digest_key)

    try:
        proc.stdin.write(data)
    except:
        kill_needed = False
        break
    if kill_needed:
        time.sleep(5)
        proc.kill()

```

Figure 19: Rotação de chaves (parte assinalada) à medida que se obtém novos blocos de música no cliente

```

# Open file, seek to correct position and return the chunk
if SONGS[media_item['file_name']]:
    data = SONGS[media_item['file_name']][offset:offset+CHUNK_SIZE]
    data = binascii.b2a_base64(data)
    encryptor_cypher = CLIENT_INFO[id]['encryptor']

    data, encryptor_cypher, iv = encryptor(CLIENT_INFO[id]['options']['selected_algorithm'], CLIENT_INFO[id]
        | ['options']['selected_mode'], CLIENT_INFO[id]['message_key'], data, encryptor_cypher, last=chunk_id == totalchunks)

    digest_data = generate_digest(CLIENT_INFO[id]['digest_key'],
        CLIENT_INFO[id]['options']['selected_hash'], data)

    if chunk_id != totalchunks:
        CLIENT_INFO[id]['encryptor'] = encryptor_cypher
    else:
        CLIENT_INFO[id]['encryptor'] = None

    response = {
        'media_id': media_id,
        'chunk': chunk_id,
        'data': data.decode('latin'),
        'digest': digest_data.decode('latin')
    }

    if chunk_id == 0:
        response['iv'] = iv.decode('latin')

    # Double Ratchet
    if chunk_id % 5 != 0 or chunk_id == 0:
        CLIENT_INFO[id]['message_key'] = generate_simple_digest(
            CLIENT_INFO[id]['options']['selected_hash'], CLIENT_INFO[id]['message_key'])
        CLIENT_INFO[id]['digest_key'] = generate_simple_digest(
            CLIENT_INFO[id]['options']['selected_hash'], CLIENT_INFO[id]['digest_key'])

    request.responseHeaders.addRawHeader(
        b"content-type", b"application/json")
    return json.dumps(
        response, indent=4
    ).encode('latin')

```

Figure 20: Rotação de chaves (parte assinalada) à medida que se obtém novos blocos de música no servidor

```

def simple_digest(self, hash, data):
    ''' Geração de um digest simples '''
    digest = hashes.Hash(HASHES_OPTIONS[hash]())
    digest.update(data)
    return digest.finalize()

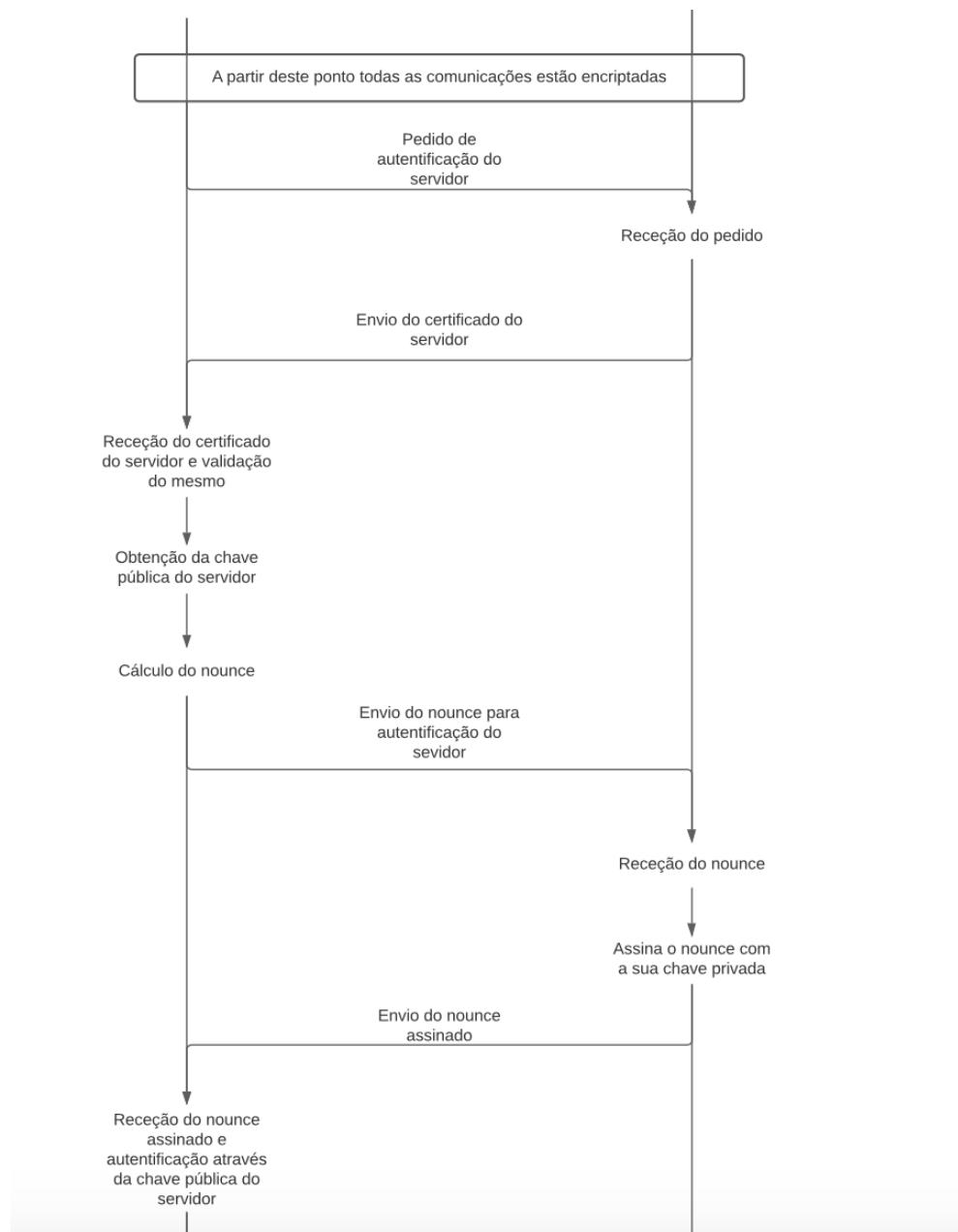
```

Figure 21: Função de síntese aplicada às chaves que utiliza o algoritmo de *hash* do protocolo escolhido

4 Autenticação e Isolamento

4.1 Autenticação do Servidor

4.1.1 Diagrama



4.1.2 Implementação

4.1.3 Infraestrutura de Chaves Públicas

Para autenticar o servidor foi criada uma Infraestrutura de chaves públicas personalizada para este caso. A base de dados utilizada no programa *xca* para gerar toda esta Infraestrutura encontra-se na pasta *server_certs* bem como todos os certificados e chaves privadas geradas para o efeito. Em suma, foi gerado o seguinte:

- **Certificado Root CA** desta Infraestrutura
- **Chave privada do Certificado Root CA** desta Infraestrutura
- **Certificado do Servidor** assinado pela *Root CA*
- **Chave privada do Certificado do Servidor**

4.1.4 Validação da Cadeia de Certificação

Com a cifra das comunicações definida e estes certificados gerados, o próximo passo do cliente é autenticar o servidor. Para tal, o cliente faz um pedido ao servidor para obter o certificado do servidor e validar a cadeia de certificação.

```
# Verificação da autenticidade do servidor
req = requests.get(f'{SERVER_URL}/api/auth', headers={"Authorization": self.id})
if req.status_code == 200:
    response = req.json()
    digest = response['digest'].encode('latin')
    certificate = response['certificate'].encode('latin')
    iv = response['iv'].encode('latin')

    if self.verify_digest(self.digest_key, self.selected_hash, certificate, digest):
        certificate, decryptor_var = self.decryptor(
            self.selected_algorithm, self.selected_mode, self.message_key, certificate, iv)
    else:
        print('\033[31m'+"Data integrity of communication violated"+'\033[0m')
        quit()

    certificate = binascii.a2b_base64(certificate)
    self.server_cert = x509.load_pem_x509_certificate(certificate)

    if not self.full_cert_verify(self.server_cert, self.root_ca_cert):
        print('\033[31m'+"Could not validate server certificate"+'\033[0m')
        quit()
```

Figure 22: Pedido do certificado ao servidor e validação no cliente

```

def get_auth(self, request):
    ''' Envio do certificado do servidor '''
    id = request.getHeader('Authorization')
    certificate = binascii.b2a_base64(
        SERVER_CERT.public_bytes(Encoding.PEM))

    certificate, encryptor_cypher, iv = encryptor(CLIENT_INFO[id]['options']['selected_algorithm'],
                                                CLIENT_INFO[id]['options']['selected_mode'],
                                                CLIENT_INFO[id]['message_key'],
                                                certificate)

    certificate_digest = generate_digest(CLIENT_INFO[id]['digest_key'],
                                          CLIENT_INFO[id]['options']['selected_hash'],
                                          certificate)

    response = {'certificate': certificate.decode('latin'),
                'digest': certificate_digest.decode('latin'),
                'iv': iv.decode('latin')}
    request.responseHeaders.addRawHeader(b"content-type", b"application/json")
    return json.dumps(response).encode('latin')

```

Figure 23: Envio do certificado do servidor, no servidor

A validação da cadeia de certificação do servidor no cliente apenas utiliza o certificado do servidor e o certificado da entidade emissora, a *Root CA*, carregado ao executar o cliente. Após verificar que o segundo certificado é de uma *Root CA* e é da entidade emissora do certificado do servidor, são verificados três pontos:

- **Validade da data do certificado**
- **Validade da assinatura do certificado**
- **Validade dos *purposes*** - Neste caso verificando se o objetivo do certificado é para *TLS Web Server Authentication*

Escolheu-se, para esta Infraestrutura, não se criar listas de revogação de certificados, uma vez que não acrescentariam grandes benefícios e o funcionamento destas seriam testadas com a integração de um *hardware token*. A implementação do processo descrito é a seguinte:

```

def full_cert_verify(self, cert, issuer_cert):
    """ Verificação da validade do certificado do servidor de acordo com a data, propósito e assinatura, dando o certificado da entidade emissora (Root CA) """
    if cert.issuer == issuer_cert.issuer and issuer_cert.issuer == issuer_cert.subject:
        if self.verify_date(cert) and self.verify_purpose(cert) and self.verify_signatures(cert, issuer_cert):
            return True
        else:
            print('\033[31m'+"Can't verify certificate integrity"+'\033[0m')
    else:
        print('\033[31m'+"Can't chain to Root CA"+'\033[0m')
    return False

def verify_purpose(self, cert):
    """ Verificação do propósito do certificado do servidor """
    if ExtendedKeyUsageOID.SERVER_AUTH in cert.extensions.get_extension_for_class(X509.ExtendedKeyUsage).value:
        return True
    else:
        return False

def verify_signatures(self, cert, issuer_cert):
    """ Verificação da assinatura de um certificado, dando o certificado da entidade emissora """
    issuer_public_key = issuer_cert.public_key()
    try:
        issuer_public_key.verify(
            cert.signature,
            cert.tbs_certificate_bytes,
            PKCS1v15(),
            cert.signature_hash_algorithm,
        )
        return True
    except InvalidSignature:
        return False

def verify_date(self, cert):
    """ Verificação da validade da data de um certificado """
    if datetime.now() > cert.not_valid_after:
        return False
    else:
        return True

```

Figure 24: Processo de validação da cadeia de certificação do certificado do servidor, no cliente

4.1.5 Autenticação Desafio-Resposta

Para o resto da autenticação do servidor recorreu-se ao conjunto de protocolos de segurança de comunicações, **Autenticação Desafio-Resposta**, abordado nas aulas teóricas. No fundo, o funcionamento assenta no envio de um **Desafio** por uma entidade, sendo que a outra entidade terá de enviar uma **Resposta Válida** para se autenticar.

Utilizando a Autenticação Desafio-Resposta, a autenticação do servidor é terminada com o envio de uma *nounce*, um número arbitrário apenas utilizado uma vez, por parte do cliente, representando o **Desafio**, e o servidor por sua vez responde com essa *nounce* assinada com a sua chave privada, de modo a que esta assinatura seja validada por parte do cliente, autenticando o servidor:

```
    print('\033[31m'+ "Could not validate server certificate"+'\033[0m')
    quit()

server_nounce = os.urandom(32)

nounce = binascii.b2a_base64(server_nounce)

nounce, encryptor_cypher, iv = self.encryptor(self.selected_algorithm, self.selected_mode, self.message_key, nounce)

nounce_digest = self.digest(self.digest_key,
                            self.selected_hash, nounce)

response = {'nounce': nounce.decode('latin'), 'digest': nounce_digest.decode('latin'), 'iv': iv.decode('latin')}

req = requests.post(f'{SERVER_URL}/api/auth', data=json.dumps(
    response).encode('latin'), headers={"content-type": "application/json", "Authorization": self.id})

if req.status_code == 200:
    response = req.json()
    digest = response['digest'].encode('latin')
    signed_nounce = response['signed_nounce'].encode('latin')
    iv = response['iv'].encode('latin')

    if self.verify_digest(self.digest_key, self.selected_hash, signed_nounce, digest):
        signed_nounce, decryptor_var = self.decryptor(
            self.selected_algorithm, self.selected_mode, self.message_key, signed_nounce, iv)
    else:
        print('\033[31m'+ "Data integrity of communication violated"+'\033[0m')
        quit()

    signed_nounce = binascii.a2b_base64(signed_nounce)

try:
    self.server_cert.public_key().verify(
        signed_nounce,
        server_nounce,
        asymmetric.padding.PSS(
            mgf=asymmetric.padding.MGF1(
                primitives.hashes.SHA256()),
            salt_length=asymmetric.padding.PSS.MAX_LENGTH
        ),
        primitives.hashes.SHA256()
    )
    print('\033[1m'+ "Authenticated server"+'\033[0m')
except InvalidSignature:
    print('\033[31m'+ "Server could not be authenticated"+'\033[0m')
    quit()

else:
    print('\033[31m'+ "Error getting signed nounce for server authentication"+'\033[0m')
    quit()
```

Figure 25: Autenticação Desafio-Resposta aplicada no final da autenticação do servidor, no cliente

```

def post_auth(self, request):
    ''' Receção de uma nounce para posterior envio permitindo a autenticação do servidor '''
    id = request.getHeader('Authorization')
    response = json.loads(request.content.read())
    digest = response['digest'].encode('latin')
    nounce = response['nounce'].encode('latin')
    iv = response['iv'].encode('latin')

    if verify_digest(CLIENT_INFO[id]['digest_key'], CLIENT_INFO[id]['options']['selected_hash'], nounce, digest):
        nounce, decryptor_var = decryptor(
            CLIENT_INFO[id]['options']['selected_algorithm'], CLIENT_INFO[id]
            ['options']['selected_mode'], CLIENT_INFO[id]['message_key'], nounce, iv)

        nounce = binascii.a2b_base64(nounce)

        signed_nounce = binascii.b2a_base64(sign(nounce))

        signed_nounce, encryptor_cypher, iv = encryptor(CLIENT_INFO[id]['options']['selected_algorithm'], CLIENT_INFO[id]
            ['options']['selected_mode'], CLIENT_INFO[id]['message_key'], signed_nounce)

        signed_nounce_digest = generate_digest(CLIENT_INFO[id]['digest_key'],
            CLIENT_INFO[id]['options']['selected_hash'], signed_nounce)

        response = {'signed_nounce': signed_nounce.decode(
            'latin'), 'digest': signed_nounce_digest.decode('latin'), 'iv': iv.decode('latin')}

        request.setResponseCode(200)
        request.responseHeaders.addRawHeader(
            b"content-type", b"application/json")
        return json.dumps(response).encode('latin')

    print('\033[31m'+f'Data integrity of communication violated for user {id}"+'\033[0m')
    request.setResponseCode(401)
    request.responseHeaders.addRawHeader(b"content-type", b"text/plain")
    return b''

```

Figure 26: Reposta do pedido de assinatura da *nounce* para a Autenticação Desafio-Resposta aplicada no final da autenticação do servidor, no servidor

```

def sign(bytes):
    ''' Assinatura de bytes com a private key do servidor '''
    with open("../server_certs/server-localhost_pk.pem", "rb") as f:
        private_key = serialization.load_pem_private_key(f.read(), None)

    signature = private_key.sign(
        bytes,
        asymmetric.padding.PSS(
            mgf=asymmetric.padding.MGF1(hashes.SHA256()),
            salt_length=asymmetric.padding.PSS.MAX_LENGTH
        ),
        hashes.SHA256()
    )

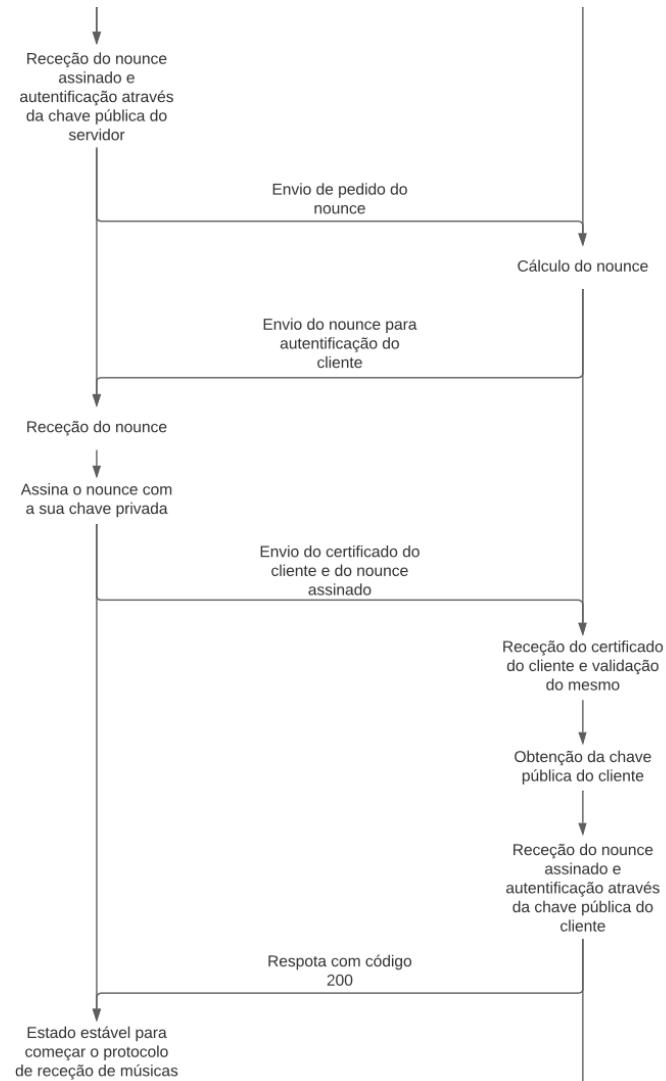
    return signature

```

Figure 27: Função de assinatura de dados com a chave privada do servidor, no servidor

4.2 Autenticação do Cliente

4.2.1 Diagrama



4.2.2 Implementação

Leitura do Cartão de Cidadão

Para o servidor autenticar o cliente, escolheu-se utilizar um *hardware token*, o cartão de cidadão. Ao iniciar a aplicação do cliente é verificada a presença de um cartão de cidadão e, caso esteja presente, é feito o carregamento inicial de diversos dados usados posteriormente. Isto tudo acontece na função `get_cc`:

```
def get_cc(self):
    """ Verificação da presença do cartão de cidadão e carregamento de informação necessária ao programa """
    try:
        # Localização para Linux
        lib = '/usr/local/lib/libpteidpkcs11.so'

        # Localização para MacOs
        if sys.platform.startswith('darwin'):
            lib = '/usr/local/lib/libpteidpkcs11.dylib'
        pkcs11 = PyKCS11.PyKCS11Lib()
        pkcs11.load(lib)

        # Listar o primeiro slot com um token
        slot = pkcs11.getSlotList(tokenPresent=True)[0]

    except:
        print('\x033[31m'+"Citizen card not present"+'\x033[0m')
        quit()

    # Filtrar atributos
    all_attr = list(PyKCS11.CKA.keys())
    all_attr = [e for e in all_attr if isinstance(e, int)]

    self.session_cc = pkcs11.openSession(slot)

    try:
        for obj in self.session_cc.findObjects():
            attr = self.session_cc.getAttributeValue(obj, all_attr)
            attr = dict(zip(map(PyKCS11.CKA.get, attr), attr))

        # Obtenção do certificado de autenticação
        if attr['CKA_LABEL'] == 'CITIZEN AUTHENTICATION CERTIFICATE':
            if attr['CKA_CERTIFICATE_TYPE'] != None:
                self.cert_cc = x509.load_der_x509_certificate(
                    bytes(attr['CKA_VALUE']))
            # Número do cartão de cidadão
            self.user_cc = self.cert_cc.subject.get_attributes_for_oid(NameOID.SERIAL_NUMBER)[0].value

            self.private_key_cc = self.session_cc.findObjects(
                [(PyKCS11.CKA_CLASS, PyKCS11.CKO_PRIVATE_KEY), (PyKCS11.CKA_LABEL, 'CITIZEN AUTHENTICATION KEY')])[0]
    except:
        print('\x033[31m'+"Invalid Citizen card"+'\x033[0m')
        quit()

    print("Correct reading of the Citizen Card\n")
    return
```

Figure 28: Leitura inicial do cartão de cidadão por parte do cliente

É guardado em memória o certificado relativo ao cartão de cidadão, a sessão, a chave privada e o número do cartão.

Autenticação Desafio-Resposta e Validação da Cadeia de Certificação

Com isto feito, já depois da autenticação do servidor, volta-se a utilizar a Autenticação Desafio-Resposta, referida anteriormente, para o servidor validar o cliente. Para tal, é feito um pedido ao servidor de modo a obter uma *nounce* gerada pelo mesmo, sendo esta assinada com a chave privada do cartão de cidadão no cliente, colocando-se as credenciais necessárias para tal. Com a *nounce* assinada, é enviado ao servidor o certificado do cartão de cidadão e a *nounce* assinada de modo a permitir a autenticação do cliente.

```
# Autenticação do Utilizador
req = requests.get(f'{SERVER_URL}/api/cc_auth', headers={"Authorization": self.id})
if req.status_code == 200:
    response = req.json()
    digest = response['digest'].encode('latin')
    nounce = response['nounce'].encode('latin')
    iv = response['iv'].encode('latin')

    if self.verify_digest(self.digest_key, self.selected_hash, nounce, digest):
        nounce, decryptor_var = self.decryptor(
            self.selected_algorithm, self.selected_mode, self.message_key, nounce, iv)
    else:
        print('\033[31m'+Data integrity of communication violated+'\033[0m')
        quit()

    nounce = binascii.a2b_base64(nounce)

    try:
        mechanism = PyKCS11.Mechanism(PyKCS11.CKM_SHA1_RSA_PKCS, None)

        signed_nounce = bytes(self.session_cc.sign(
            self.private_key_cc, nounce, mechanism))
    except:
        print('\033[31m'+Unable to sign with citizen card+'\033[0m')
        quit()
    else:
        print('\033[31m'+Error getting nounce for user authentication+'\033[0m')
        quit()

    certificate = binascii.b2a_base64(self.cert_cc.public_bytes(Encoding.PEM))
    signed_nounce = binascii.b2a_base64(signed_nounce)

    certificate, encryptor_cypher, certificate_iv = self.encryptor(self.selected_algorithm, self.selected_mode, self.message_key, certificate)

    certificate_digest = self.digest(self.digest_key,
                                    self.selected_hash, certificate)

    signed_nounce, encryptor_cypher, signed_nounce_iv = self.encryptor(self.selected_algorithm, self.selected_mode, self.message_key, signed_nounce)

    signed_nounce_digest = self.digest(self.digest_key,
                                    self.selected_hash, signed_nounce)

    response = {'certificate': certificate.decode('latin'), 'certificate_digest': certificate_digest.decode('latin'), 'certificate_iv': certificate_iv.decode('latin'),
    'signed_nounce': signed_nounce.decode('latin'), 'signed_nounce_digest': signed_nounce_digest.decode('latin'), 'signed_nounce_iv': signed_nounce_iv.decode('latin')}

    req = requests.post(f'{SERVER_URL}/api/cc_auth', data=json.dumps(
        response).encode('latin'), headers={"content-type": "application/json", "Authorization": self.id})

    if req.status_code == 200:
        print('\033[1m'+Authenticated user+'\033[0m')
    else:
```

Figure 29: Processo de autenticação do cliente, no cliente

Para validar a cadeia de certificação do certificado do cartão de cidadão, o servidor, ao inicializar, necessita de carregar certificados intermédios e de raiz, bem como listas de revogação do governo e de outras entidades utilizadas para validar estes certificados. Para tal, foram utilizadas as funções `get_chain` e `get_crls`, sendo que a primeira tem a responsabilidade de carregar os certificados intermédios e de raiz e a segunda tem a responsabilidade de carregar as listas de revogação, previamente obtidas nos endereços presentes na Bibliografia. De notar que, devido ao trabalho ter sido testado em *MacOS* e os certificados raiz, neste SO, serem guardados de maneira diferente, optou-se ao invés de carregar os certificados na localização `/etc/ssl/certs`, como habitualmente, carregar o certificado raiz da cadeia de certificação do cartão de cidadão utilizado, guardado previamente.

```

def get_chain():
    ''' Obtenção de certificados para futura construção da cadeias de certificados para autenticar o cliente '''
    for root, dirs, files in os.walk("../cc_certs/"):
        for filename in files:
            if filename != '.DS_Store':
                certificate = load_cert("../cc_certs/" + filename)
                if verify_date(certificate):
                    CC_CERTS[certificate.subject] = certificate

    for root, dirs, files in os.walk("../root_certs/"):
        for filename in files:
            if filename != '.DS_Store':
                certificate = load_cert("../root_certs/" + filename)
                if verify_date(certificate):
                    ROOT_CERTS[certificate.subject] = certificate

def get_crls():
    ''' Obtenção de crls para futura autenticação do cliente '''
    for root, dirs, files in os.walk("../cc_crl/"):
        for filename in files:
            if filename != '.DS_Store':
                CRLS.append(load_crl("../cc_crl/" + filename))

```

Figure 30: Processo carregamento dos certificados intermédia, raiz e listas de revogação no servidor

Com toda esta informação carregada previamente, é possível validar a cadeia de certificação do certificado recebido e validar a *nounce* assinada:

```
def post_cc_auth(self, request):
    ''' Receção de nounce e certificado para autenticar cliente '''
    responseCode = 401
    response = json.loads(request.content.read())
    id = request.getHeader('Authorization')
    certificate_digest = response['certificate_digest'].encode('latin')
    certificate = response['certificate'].encode('latin')
    certificate_iv = response['certificate_iv'].encode('latin')
    signed_nounce_digest = response['signed_nounce_digest'].encode('latin')
    signed_nounce = response['signed_nounce'].encode('latin')
    signed_nounce_iv = response['signed_nounce_iv'].encode('latin')

    if verify_digest(CLIENT_INFO[id]['digest_key'], CLIENT_INFO[id]['options']['selected_hash'], certificate, certificate_digest):
        certificate, decryptor_var = decryptor(
            CLIENT_INFO[id]['options']['selected_algorithm'], CLIENT_INFO[id]
            ['options']['selected_mode'], CLIENT_INFO[id]['message_key'], certificate, certificate_iv)
    else:
        print('\033[31m'+f'Data integrity of communication violated for user {id}'+'\033[0m')
        request.setResponseCode(responseCode)
        request.responseHeaders.addRawHeader(b"content-type", b"text/plain")
        return b''

    if verify_digest(CLIENT_INFO[id]['digest_key'], CLIENT_INFO[id]['options']['selected_hash'], signed_nounce, signed_nounce_digest):
        signed_nounce, decryptor_var = decryptor(
            CLIENT_INFO[id]['options']['selected_algorithm'], CLIENT_INFO[id]
            ['options']['selected_mode'], CLIENT_INFO[id]['message_key'], signed_nounce, signed_nounce_iv)
    else:
        print('\033[31m'+f'Data integrity of communication violated for user {id}'+'\033[0m')
        request.setResponseCode(responseCode)
        request.responseHeaders.addRawHeader(b"content-type", b"text/plain")
        return b''

    certificate = binascii.a2b_base64(certificate)

    signed_nounce = binascii.a2b_base64(signed_nounce)

    client_cert = x509.load_pem_x509_certificate(certificate)

    chain = get_cc_chain(client_cert)

    nounce = CLIENT_INFO[id]['cc_nounce']

    if chain and nounce:
        if full_chain_cert_verify(chain, client_cert, True):
            try:
                client_cert.public_key().verify(
                    signed_nounce,
                    nounce,
                    asymmetric.padding.PKCS1v15(),
                    hashes.SHA1()
                )
            except InvalidSignature:
                print('\033[31m'+f'Invalid signed nounce'+'\033[0m')
                pass

    request.setResponseCode(responseCode)
    request.responseHeaders.addRawHeader(b"content-type", b"text/plain")
    return b''
```

Figure 31: Processo de autenticação do cliente, no servidor

Ao validar a cadeia de certificação, o servidor cria a própria cadeia de certificação do certificado do cartão de cidadão, recorrendo à função `get_cc_chain`, que percorre todos os certificados, verificando se algum deles é a entidade emissora do certificado passado e terminando quando encontra um certificado raiz, criando assim a cadeia.

```
def get_cc_chain(cert, chain={}):
    """ Construção da cadeia de certificados de um dado certificado proveniente do cartão de cidadão """
    chain[cert.subject] = cert

    if cert.issuer == cert.subject and cert.issuer in ROOT_CERTS:
        return chain
    elif cert.issuer in ROOT_CERTS:
        return get_cc_chain(ROOT_CERTS[cert.issuer], chain)
    elif cert.issuer in CC_CERTS:
        return get_cc_chain(CC_CERTS[cert.issuer], chain)

    return False
```

Figure 32: Criação da cadeia de certificação do certificado do cartão de cidadão

Existindo esta cadeia de certificação, o servidor verifica que todos os certificados estão válidos, de forma semelhante à feita pelo cliente mas, desta vez, verificando também com as listas de revogação carregadas. Neste caso, o objetivo dos certificados terá de ser do tipo *digital_signature* no certificado do cartão de cidadão, visto que é utilizado para verificar assinaturas sem ser de certificado e, do tipo *key_cert_sign* nos restantes, visto que são utilizados para verificar assinaturas de outros certificados:

```
def full_chain_cert_verify(chain, cert, first=False):
    """ Verificação da validade do certificado do cliente de acordo com a data, propósito e assinatura, repetindo o processo para todos os certificados da sua cadeia """
    issuer_cert = chain[cert.issuer]
    if verify_date(cert) and verify_purpose(cert, first) and verify_crls(cert) and verify_signatures(cert, issuer_cert):
        if cert.issuer == issuer_cert.issuer:
            return True
        else:
            return full_chain_cert_verify(chain, issuer_cert)
    else:
        print('\x033[31m'+ "Can't verify certificate integrity"+'\x033[0m')
        return False

def verify_crls(cert):
    """ Verificação da validade do certificado de acordo com as crls carregadas """
    for crl in CRLS:
        if crl.get_revoked_certificate_by_serial_number(cert.serial_number):
            return False
    return True

def verify_purpose(cert, is_cc=False):
    """ Verificação do propósito de um certificado (varia consoante o tipo) """
    if is_cc:
        if cert.extensions.get_extension_for_class(x509.KeyUsage).value.digital_signature:
            return True
    else:
        if cert.extensions.get_extension_for_class(x509.KeyUsage).value.key_cert_sign:
            if cert.extensions.get_extension_for_class(x509.BasicConstraints).value.ca:
                return True
    return False

def verify_signatures(cert, issuer_cert):
    """ Verificação da assinatura de um certificado, dando o certificado da entidade emissora """
    issuer_public_key = issuer_cert.public_key()
    try:
        issuer_public_key.verify(
            cert.signature,
            cert.tbs_certificate_bytes,
            PKCS1v15(),
            cert.signature_hash_algorithm,
        )
        return True
    except InvalidSignature:
        return False

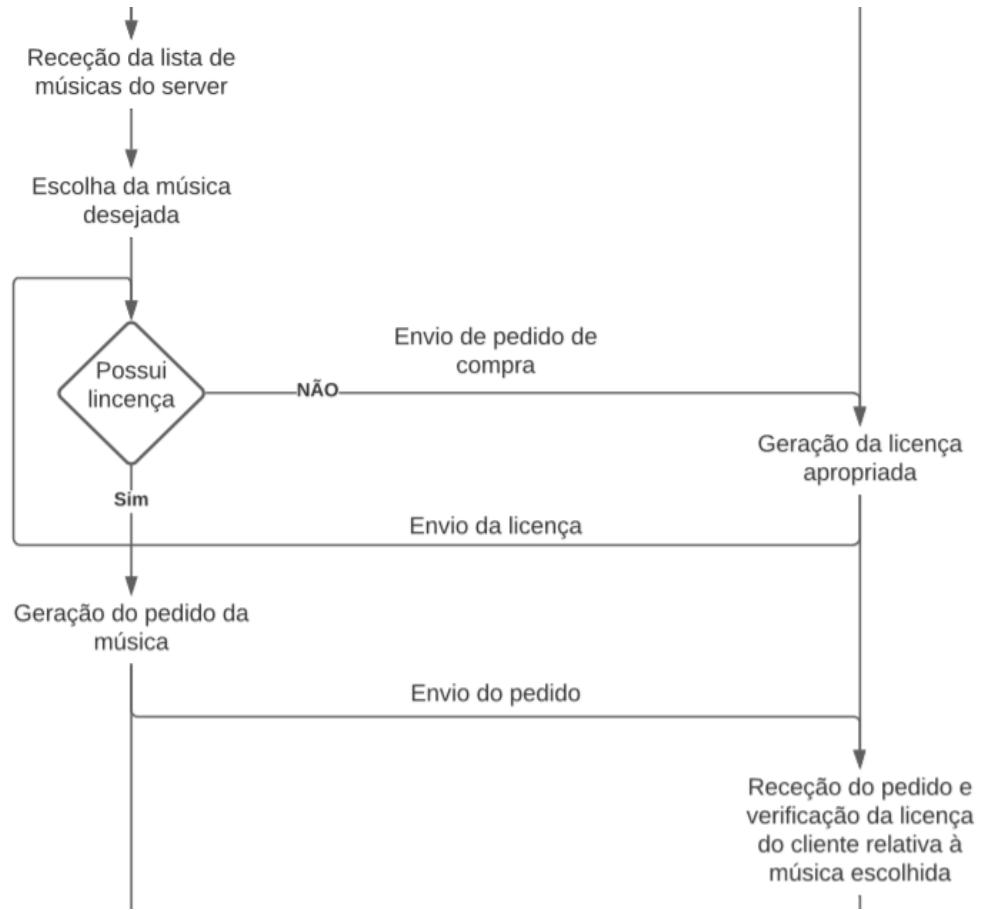
def verify_date(cert):
    """ Verificação da validade da data de um certificado """
    if datetime.now() > cert.not_valid_after:
        return False
    else:
        return True
```

Figure 33: Validação da cadeia de certificação

Estando este processo concluído em conformidade, o cliente passa a estar autenticado para com o servidor.

4.3 Licenças e Autenticação das mesmas

4.3.1 Diagrama



4.3.2 Implementação

Ultrapassadas todas as medidas de seguranças mencionadas anteriormente, o programa pode finalmente mostrar ao cliente as músicas disponíveis e, caso não tenha acesso, permitir a compra de músicas através da geração de uma licença com tempo limitado.

Para isso, é feito um pedido ao servidor de modo a listar as músicas disponíveis. Nesta listagem é também apresentado se o cliente possui acesso à música ou não, de acordo com as licenças geradas anteriormente e com as informações provenientes da autenticação do cliente:

```
MEDIA CATALOG
0 - Sunny Afternoon - Upbeat Ukulele Background Music - Access: False
-----
Select a media file number (q to quit). If a song with no access is selected, a
new license will be requested:
```

Figure 34: Exemplo de listagem de músicas ao executar o programa do cliente

Toda esta informação é apresentada com a seguinte implementação:

```
# Obter lista de músicas
while True:
    media_list = None
    while True:
        req = requests.get(f'{SERVER_URL}/api/list', headers={"Authorization": self.id})
        if req.status_code == 200:
            print("Got Server List")
        else:
            print('\033[31m'+"Error getting server list"+'\033[0m')
            continue

        response = req.json()
        digest = response['digest'].encode('latin')
        media_list = response['media_list'].encode('latin')
        iv = response['iv'].encode('latin')

        if self.verify_digest(self.digest_key, self.selected_hash, media_list, digest):
            media_list, decryptor_var = self.decryptor(
                self.selected_algorithm, self.selected_mode, self.message_key, media_list, iv)

            media_list = json.loads(media_list.decode('latin'))['media_list']
        else:
            print('\033[31m'+"Data integrity of communication violated"+'\033[0m')
            quit()

    # Present a simple selection menu
    idx = 0
    print("\nMEDIA CATALOG")
    for item in media_list:
        print(f'{idx} - {media_list[idx]["name"]} - Access: {media_list[idx]["has_access"]}')
    print("-----")

    while True:
        selection = input("Select a media file number (q to quit). If a song with no access is selected, a new license will be requested: ")
        if selection.strip() == 'q':
            req = requests.get(f'{SERVER_URL}/api/logout', headers={"Authorization": self.id})
            if req.status_code == 200:
                print("All done!")
                sys.exit(0)

        if not selection.isdigit():
            continue

        selection = int(selection)
        if 0 <= selection < len(media_list):
            break
```

Figure 35: Obter lista de músicas no cliente

Caso o cliente não tenha permissão para ouvir uma certa música, ao seleccionar essa música, irá fazer um pedido ao servidor para gerar uma nova licença, sendo que o servidor gerará uma licença com validade de 1 hora, permitindo ao utilizador ouvir a música na próxima hora, como será explicado posteriormente. Esta licença contém, também, no final, separado por *b"-"*, a assinatura da mesma feita pelo servidor. A validade da licença depende da assinatura, sendo que ao receber a licença, o cliente confirma se a assinatura é válida antes de a guardar em *client/licenses*. Apesar da licença guardada pelo utilizador não ser utilizada, permite que o servidor não consiga refutar a mesma em casos extremos:

```
# Obter licença para música
if not media_list[selection]['has_access']:
    req = requests.get(f'{SERVER_URL}/api/get_music?id={media_list[selection]["id"]}', headers={"Authorization": self.id})
    if req.status_code == 200:
        print()
        print('\033[1m'+f"Access granted to the song '{media_list[selection]['name']}' for 1 hour"+'\033[0m')
        print()
        response = req.json()
        digest = response['digest'].encode('latin')
        license = response['license'].encode('latin')
        iv = response['iv'].encode('latin')

        if self.verify_digest(self.digest_key, self.selected_hash, license, digest):
            license, decryptor_var = self.decryptor(
                self.selected_algorithm, self.selected_mode, self.message_key, license, iv)
            decoded_license = license.split(b"-")
            signature = base64.b64decode(decoded_license[1])
            decoded_license = base64.b64decode(decoded_license[0])

            try:
                # Verificação da assinatura
                self.server_cert.public_key().verify(
                    signature,
                    decoded_license,
                    asymmetric.padding.PSS(
                        mgf=asymmetric.padding.MGF1(
                            primitives.hashes.SHA256()),
                        salt_length=asymmetric.padding.PSS.MAX_LENGTH
                    ),
                    primitives.hashes.SHA256()
                )
                decoded_license = json.loads(decoded_license.decode('latin'))
                with open("./licenses/"+str(decoded_license['serial_number_cc'])+"_"+str(decoded_license['media_id'])+"_"+str(decoded_license['date_of_expiration'])+".txt", "wb") as f:
                    f.write(license)
                    f.close()
            except InvalidSignature:
                print("\033[1m"+f"Music license is not valid!"+'\033[0m')
                quit()
            else:
                print('\033[31m'+f"Data integrity of communication violated"+'\033[0m')
                quit()
        else:
            print('\033[31m'+f"Error getting song license"+'\033[0m')
    else:
        break
```

Figure 36: Obter licença de músicas no cliente

No servidor, é gerada essa licença e guardada em *server/licenses*, sendo que o nome do ficheiro é constituído pelo *serial_number_cc*, *media_id* e *date_of_expiration* para permitir uma verificação prévia à abertura do ficheiro e verificação da assinatura.

A licença é um objeto JSON com as seguintes propriedades:

- **serial_number_cc** - O número do cartão de cidadão, que é guardado no servidor durante a autenticação do utilizador em comunicação cifrada
- **media_id** - O identificador da música em questão
- **date_of_expiration** - Data até quando a licença é válida

```

def get_license(self, request):
    ''' Geração e envio de uma licença para uma dada música e utilizador com validade de 1 hora '''
    id = request.getHeader('Authorization')
    media_id = request.args.get(b'id', [None])[0].decode('latin')

    date = datetime.now() + timedelta(hours=1)
    new_license = {'serial_number_cc': CLIENT_INFO[id]['serial_number_cc'],
                  'media_id': media_id,
                  'date_of_expiration': date.strftime("%Y-%m-%d-%H-%M-%S")}

    new_license = json.dumps(new_license).encode('latin')

    signature = sign(new_license)

    result = base64.b64encode(new_license) + \
             b"-" + base64.b64encode(signature)

    with open("./licenses/" + str(CLIENT_INFO[id]['serial_number_cc']) + "_" + str(media_id) + "_" + str(date.strftime("%Y-%m-%d-%H-%M-%S")) + ".txt", "wb") as f:
        f.write(result)
        f.close()

    result, encryptor_cypher, iv = encryptor(CLIENT_INFO[id]['options']['selected_algorithm'], CLIENT_INFO[id]
                                              ['options']['selected_mode'], CLIENT_INFO[id]['message_key'], result)

    result_digest = generate_digest(CLIENT_INFO[id]['digest_key'],
                                    CLIENT_INFO[id]['options']['selected_hash'], result)

    request.responseHeaders.addRawHeader(
        b"content-type", b"application/json")
    return json.dumps({'license': result.decode('latin'), 'digest': result_digest.decode('latin'), 'iv': iv.decode('latin')}).encode('latin')

```

Figure 37: Geração licença de músicas no servidor

Com a licença gerada, quando o cliente ouvir uma música, o servidor verifica em todos os pedidos de cada *bloco* da música se o cliente continua a ter uma licença válida:

```

def do_download(self, request):
    ''' Envio de um determinado chunk de uma música caso o utilizador tenha licença '''
    id = request.getHeader('Authorization')
    logger.debug(f'Download: args: {request.args}')

    media_id = request.args.get(b'id', [None])[0]
    logger.debug(f'Download: id: {media_id}')

    # Check if the media_id is not None as it is required
    if media_id is None:
        request.setResponseCode(400)
        request.responseHeaders.addRawHeader(
            b"content-type", b"application/json")
        return json.dumps({'error': 'invalid media id'}).encode('latin')

    # Convert bytes to str
    media_id = media_id.decode('latin')

    # Search media_id in the catalog
    if media_id not in CATALOG:
        request.setResponseCode(404)
        request.responseHeaders.addRawHeader(
            b"content-type", b"application/json")
        return json.dumps({'error': 'media file not found'}).encode('latin')

    # Check license
    if not license_check(media_id, id):
        request.setResponseCode(401)
        request.responseHeaders.addRawHeader(
            b"content-type", b"application/json")
        return json.dumps({'error': 'License not found'}).encode('latin')

```

Figure 38: Verificação (parte assinalada) da licença ao enviar *blocos* de músicas, no servidor

A verificação das licenças do lado do servidor percorre todas as licenças guardadas, faz uma verificação inicial segundo o nome do ficheiro e, em caso positivo, abre o ficheiro e verifica as informações no objeto *JSON* e a validade da assinatura:

```
def license_check(media_id, user_id):
    """ Verificação da validade das licenças de obtenção de músicas para uma dada música e cliente, retornando True caso o cliente tenha pelo menos uma licença válida """
    has_access = False
    for root, dirs, files in os.walk("./licenses/"):
        for filename in files:
            filename_split = filename.split(".")
            # Verificação inicial com o nome do ficheiro
            if filename_split[1] == media_id and CLIENT_INFO[user_id]['serial_number_cc'] == filename_split[0] and datetime.strptime(filename_split[2].split('.')[0], "%Y-%m-%d-%H-%M-%S") > datetime.now():
                with open("./licenses/" + filename, "rb") as f:
                    content = f.read()
                    content = content.split("\n")
                    license = base64.b64decode(content[0])
                    signature = base64.b64decode(content[1])
                    try:
                        # Verificação da assinatura
                        SERVER_CERT.public_key().verify(
                            signature,
                            license,
                            asymmetric.padding.PSS(
                                mgf=asymmetric.padding.MGF1(
                                    hashes.SHA256()),
                                salt_length=asymmetric.padding.PSS.MAX_LENGTH
                            ),
                            hashes.SHA256()
                        )

                        license = json.loads(license.decode('latin'))
                        # Verificação da própria licença
                        if license['media_id'] == media_id and CLIENT_INFO[user_id]['serial_number_cc'] == license['serial_number_cc'] and datetime.strptime(license['date_of_expiration'], "%Y-%m-%d-%H-%M-%S") > datetime.now():
                            has_access = True
                            return has_access
                    except InvalidSignature:
                        pass
    return has_access
```

Figure 39: Função de verificação da validade de licenças dada uma música e um utilizador, no servidor

4.4 Proteção do conteúdo guardado no servidor

4.4.1 Encriptação

No âmbito da proteção dos ficheiros a nível do servidor foi desenvolvido um *script* independente, *file_encrypt.py* em */server*, o qual tem que ser executado antes de iniciar o servidor caso as músicas ainda não estejam cifradas.

Através deste *script*, é efetuada a encriptação dos ficheiros segundo uma cifra híbrida e são guardadas as versões protegidas no diretório */server/encrypted_catalog*, sendo ainda realizada a geração de um documento de texto com a informação que o servidor necessita para proceder à desencriptação, nomeadamente, nome do ficheiro, chave simétrica utilizada para encriptação das músicas e *iv* utilizado.

Primeiramente é percorrido o diretório */server/catalog* e para cada música é efetuada uma encriptação com chave simétrica, segundo o algoritmo *AES* e modo de cifra *OFB*, utilizando uma chave e *iv* aleatórios. Após a cifra os novos documentos são guardados em */server/encrypted_catalog*.

```
for root, dirs, files in os.walk("./catalog/"):
    for filename in files:
        # Evita os ficheiros gerados pelo MacOS
        if filename != '.DS_Store':
            block_size = algorithms.AES.block_size // 8
            iv = os.urandom(block_size)

            key = os.urandom(32)

            cipher = Cipher(algorithms.AES(
                key), modes.OFB(iv))
            encryptor = cipher.encryptor()

            f = open("./catalog/" + filename, "rb")
            f2 = open("./encrypted_catalog/" +
                      filename.split(".")[0] + ".bin", "wb")
            counter = 0
            content = f.read(block_size)

            # Cifra da música recorrendo a uma Cifra Simétrica com chave aleatória
            while True:
                if len(content) < block_size:
                    ct = encryptor.update(content) + encryptor.finalize()
                    break
                else:
                    ct = encryptor.update(content)

                f2.write(ct)

                counter += 1
                f.seek(counter*block_size)
                content = f.read(block_size)

            f.close()
            f2.close()
```

Figure 40: Processo de encriptação das músicas.

Posteriormente é gerado um documento de texto *file_info.txt* no diretório */server* que, para cada música, contém o nome do ficheiro, a chave de encriptação cifrada através da chave pública do servidor e o *iv* utilizados na cifra simétrica dessa música. Desta forma foi utilizada cifra simétrica para cifrar os ficheiros e cifra assimétrica para cifrar a chave simétrica, juntando o melhor dos dois mundos.

```

9   f = open("../server_certs/server-localhost_pk.pem", "rb")
10  private_key = serialization.load_pem_private_key(f.read(), None)
11
12  public_key = private_key.public_key()
13
14 # Ficheiro com as informações relativas às músicas cifradas, nomeadamente nome do ficheiro,
15 # chave de encriptação e iv
16 f_info = open("file_info.txt", "ab")
17
18
19      # Cifra Assimétrica da chave aleatória usada para cifrar a música permitindo apenas acesso pelo servidor
20      encrypted_key = public_key.encrypt(
21          key,
22          padding.OAEP(
23              mgf=padding.MGF1(algorithm=hashes.SHA256()),
24              algorithm=hashes.SHA256(),
25              label=None
26          )
27      )
28
29
30      # Primeiros 128 bits da linha alocados para o nome do ficheiro, recorrendo a padding
31      filename = bytes(filename, encoding='utf8')
32      diff = 128-len(filename)
33      filename += bytes([diff]*diff)
34
35      f_info.write(filename+encrypted_key+iv)
36
37 f_info.close
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71

```

Figure 41: Geração do documento *file_info.txt*

4.4.2 Desencriptação

Ao iniciar o servidor é chamada a função *decrypt_catalog()*, que irá por sua vez proceder à leitura da informação contida em *file_info.txt*, desencriptará as chaves simétricas utilizadas através da chave privada do servidor e poderá de seguida desencriptar os ficheiros das músicas e guardá-los em memória *RAM* onde estarão seguros, mais exatamente no dicionário SONGS.

Esta solução simplificada foi adotada no âmbito do projeto uma vez que não compromete a segurança visto que a memória *RAM* é volátil, no entanto reconhecemos que numa situação real representaria um problema a nível de escalabilidade do serviço, podendo ser interessante aplicar outro modo de cifra, como o *CBC*, apenas necessitando do bloco anterior ao que se pretende para ser possível desencriptar.

```

def decrypt_catalog():
    """ Obtenção das músicas decriptadas para variáveis (memória RAM) """
    private_key = None
    file_name = None
    decrypted_key = None
    iv = None

    try:

        with open("../server_certs/server-localhost_pk.pem", "rb") as server_cert_file:
            private_key = serialization.load_pem_private_key(
                server_cert_file.read(), None)

        info_file = open('file_info.txt', 'rb')

        for line in info_file.readlines():
            file_name = line[0:128-line[127]].decode('utf-8')
            encrypted_key = line[128:384]
            iv = line[384:416]

            decrypted_key = private_key.decrypt(
                encrypted_key,
                asymmetric.padding.OAEP(
                    mgf=asymmetric.padding.MGF1(algorithm=hashes.SHA256()),
                    algorithm=hashes.SHA256(),
                    label=None
                )
            )

        for root, dirs, files in os.walk("./encrypted_catalog/"):
            for filename in files:
                if filename != '.DS_Store':
                    if filename.split('.')[0] == file_name.split('.')[0]:

                        block_size = algorithms.AES.block_size // 8

                        with open("./encrypted_catalog/" + filename, mode='rb') as encrypted_song_file:
                            counter = 0
                            content = encrypted_song_file.read(block_size)

                            cipher = Cipher(algorithms.AES(
                                decrypted_key), modes.OFB(iv))
                            decryptor = cipher.decryptor()

                            SONGS[file_name] = bytearray()

                            while True:
                                if len(content) < block_size:
                                    SONGS[file_name] += bytearray(
                                        decryptor.update(content) + decryptor.finalize())
                                    break
                                else:
                                    SONGS[file_name] += bytearray(
                                        decryptor.update(content))

                                counter += 1
                                encrypted_song_file.seek(counter*block_size)
                                content = encrypted_song_file.read(block_size)

    except:
        print('\033[31m'+"Error loading encrypted songs"+'\033[0m')
        quit()

    return

```

Figure 42: Função decrypt_catalog()

5 Conclusão

Para terminar, pensa-se que, de acordo com as metas estabelecidas pelo docente, o trabalho foi bem sucedido. Desde a comunicação segura, até às diversas autenticações, foi possível solidificar e aprofundar o conhecimento adquirido durante as aulas práticas e teóricas.

A implementação baseou-se nas questões relativas à segurança pelo que algumas verificações menos importantes e/ou manutenção de código limpo podem estar em falta.

6 Bibliografia

- [1] http://cryptowiki.net/index.php?title=The_Double_Ratchet_Algorithm
- [2] <https://www.youtube.com/watch?v=9s02qdTci-s>
- [3] <https://www.ecce.gov.pt/certificados/>
- [4] <https://pki.cartaoeconomia.pt>
- [5] <https://cryptography.io/en/latest/index.html>

Para além destes *websites*, foi utilizada muita informação proveniente dos slides e guiões das aulas teóricas e práticas.