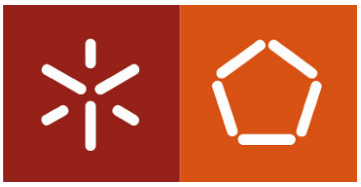

Universidade do Minho

Mestrado em Engenharia Informática

Gestão e Segurança de Redes (2022/23)

Simão Cunha (a93262)

Braga, 25 de junho de 2023

Conteúdo

1 Estratégias

Neste projeto da UC de Gestão e Segurança de Redes, explicarei as estratégias escolhidas para os dois trabalhos práticos.

1.1 TP1

1.1.1 Leitura do ficheiro de configuração

De forma a implementar este requisito, criei um ficheiro de configuração com os requisitos do enunciado, i.e., número de linhas ou colunas da matriz, chave mestra, intervalo (em ms) de atualizações de matrizes, tempo máximo do armazenamento da informação na matriz (em segundos), número máximo de entradas na tabela e porta de atendimento UDP. De seguida, criei uma estrutura de dados *Configurations* que lê este ficheiro e, utilizando expressões regulares, atribui os valores lidos às variáveis desta classe. Além disto, esta classe também contém mais duas variáveis de instância *hard coded*, que representam o intervalo do código ASCII para os caracteres da matriz *fm*.

1.1.2 Geração de chaves

De forma a permitir a geração de chaves, criei uma diretoria *keys/* que contém todos os módulos necessários para este requisito. Neste caso, possuo duas matrizes M_1 e M_2 com os primeiros K bytes e com os últimos K bytes, respetivamente - o valor de K já estará registado na classe *configurations* referida anteriormente. Uma vez que optei por utilizar uma matriz *fm*, apenas vou ter as matrizes Z_a, Z_b e Z_s através das regras definidas no enunciado. No final, obterei a matriz Z necessária para a geração da chave. Num módulo à parte (*update_matrix.py*), irei atualizar a matriz Z de T em T ms (T consta no ficheiro de configuração e, por conseguinte, na classe *configurations*). De seguida, a partir desta matriz Z, gerarei a chave juntamente com a matriz *fm* criada com K caracteres.

NOTA: As funções utilizadas nesta diretoria irão ser utilizadas no agente, mas explicarei isso mais à frente no relatório.

1.1.3 Managers

Os managers serão os clientes que efetuarão pedidos ao agente (o servidor neste paradigma cliente-servidor) e, posteriormente, receber respostas dessa entidade. Para tal, criar-se-á uma classe *Manager* que possui 3 variáveis de instância: *AGENT_HOST* que se refere ao endereço IP do manager - será o *localhost* (ou 127.0.0.1) pois tanto os clientes como o servidor executarão na mesma máquina; *AGENT_PORT*, que se refere à porta da ligação com o servidor (será a mesma que a usada pelo agente); e *BUFFER_SIZE*, que corresponderá ao tamanho alocado para o socket de comunicação.

Um manager só poderá efetuar dois tipos de pedidos: *set()* e *get()*. Para pedir ao servidor que crie uma chave, o manager terá de enviar um pedido *set()* utilizando os OIDs do grupo *data*, do objeto referente à tabela, do objeto referente ao *keyVisibility* e ao OID reservado 0 - por outras palavras, a sequência terá de ser 3.2.6.0. Nesta situação, o cliente não terá de enviar um novo valor a colocar no objeto referido na sequência de OIDs. Caso se trate de um pedido *set()* para um valor existente na MIB, o agente terá de atribuir um novo valor, mas esse terá de ser do mesmo tipo que o definido para o objeto em questão (caso contrário, o

servidor enviará uma resposta de erro - irei falar mais detalhadamente sobre o tratamento de erros mais a frente).

Por outro lado, caso se se trate de um pedido de *get()*, o raciocínio na construção da PDU a ser enviada para o agente é análogo ao de pedidos de *set()*.

Exemplos de comandos efetuados (em Windows)

- Sintaxe abstrata de um comando:

```
python .\manager.py [password_do_cliente] [set/get] [ID_do_pedido]
[sequência_de_oids] [novo_valor - se pedido = set (para alterar valor)]
```

- para criar uma chave (comando SET):

```
python .\manager.py password set 1 3.2.6.0
```

- para alterar valor (comando SET):

```
python .\manager.py password set 4 2.1 42
python .\manager.py password set 7 3.2.4.2 novo_valor
```

- para obter valor (comando GET):

```
python .\manager.py password get 12 2.1
```

Construção da PDU

Sabendo que argumentos um cliente usa na linha de comandos, a PDU de envio de pedido é simples de se entender:

PDU do pedido	
security_level	0
n_security_parameters_number	1
n_security_parameters_list	[password do cliente]
request_id	valor inserido pelo cliente
primitive_type	1 = get; 2 = set
instance_elements_size	comprimento de *instance_elements_list*
instance_elements_list	sequência de OIDs inseridas pelo cliente + novo valor caso seja um pedido de set()
error_elements_size	[ ]
error_elements_list	0

Observação: Embora neste TP não exista segurança, decidi incluir uma password no cliente de forma a poder distinguir clientes diferentes no campo da PDU *n_security_parameters_list* - existe a assunção de que nenhum cliente possui a mesma password, não existindo essa verificação. A utilidade desta password irá ser explicada na próxima secção.

1.1.4 Agente

Neste trabalho prático, o agente terá o papel de servidor, respondendo a pedidos dos seus clientes/gestores. Para tal, foi criada uma classe *Agent*, que possuirá 5 variáveis de instância: *n_lock*, que será responsável por fazer o controlo de concorrência da variável *n_updated_times* que irá ser partilhada por várias threads (entenda-se que um cliente irá ser conectado por uma thread); *n_updated_times* - refere-se ao número de vezes em que a matriz Z foi atualizada; *bufferSize* será o tamanho alocado para a comunicação entre sockets; *HOST* será o endereço local (*127.0.0.1*); *current_time* será a variável responsável por contar o tempo desde o início da execução do agente - apenas começará a contar o tempo depois de, na função *main* desta classe, o agente ser criado, ler o ficheiro de configuração, preencher os valores iniciais da MIB e criar o ficheiro de registo de todos os clientes que se conectaram ao agente, com o intuito de apenas refletir o tempo em que o agente apenas fica a espera de pedidos dos managers.

Tal como já referido, é o agente o responsável por ler o ficheiro de configuração e carregar os dados para a estrutura de dados reservada para esse efeito, preencher os valores iniciais da MIB e criar o ficheiro de registo de todos os clientes já conectados.

Assim, depois de efetuar as tarefas iniciais, irá ser lançada uma thread para cada cliente conectado, mas, de forma a simplificar a elaboração do trabalho prático, o agente irá devolver a resposta de forma sequencial. Por outras palavras, vários clientes irão poder estar conectados ao servidor, mas as respostas irão ser devolvidos através de uma política FIFO - o primeiro cliente a conectar-se ao servidor será o primeiro a receber a resposta e, consequentemente, desligar-se do servidor.

Em cada thread lançada, o agente irá descodificar a PDU recebida pelo manager e calcular o tempo em segundos desde que o agente foi inicializado. De seguida, irá verificar se esse cliente enviou um pedido com um *request_id* igual há pelo pelos V segundos (este valor V consta no ficheiro de configuração) - caso tenha enviado há menos de V segundos, o servidor rejeita este pedido e envia uma mensagem de erro, terminando a sua ligação com esse cliente; caso contrário, esse cliente será registado ao ficheiro de registo. De seguida, verifica se a lista de OIDs inserida pelo cliente é válida - apenas verifico se não envia uma lista com mais de 4 OIDs: se for inválido, envio mensagem de erro e a ligação é terminada; caso contrário, volto a verificar que tipo de pedido se trata: *set()* para criação de chave, *set()* para alteração de valor ou *get()* para obtenção de valor. Em qualquer um dos tipos de pedidos, vou fazendo acessos sucessivos às estruturas de dados e, caso falhe em algum, envio imediatamente uma mensagem de erro ao cliente, ignorando eventuais outros erros que possam existir.

Construção da PDU

O formato da PDU de resposta do servidor/agente será o seguinte:

PDU de resposta	
security_level	0
n_security_parameters_number	0
n_security_parameters_list	[ ]
request_id	valor inserido pelo cliente
primitive_type	0 = response
instance_elements_size	comprimento de *instance_elements_list*
instance_elements_list	sequência de OIDs inseridas pelo cliente + valor consultado do comando get()/ *keyId* da chave criada do comando set()
error_elements_size	[string que contém um código de erro (se aplicável)]
error_elements_list	comprimento de *error_elements_size*

Códigos de erro

Nesta secção irei descrever quais os erros detetados pela minha aplicação. O código de erro é devolvido numa string representativa, embora o mais correto passa-se por devolver um inteiro que referencia-se um erro pré-catalogado

- Erro #1 (`"MIB FULL"`): Pedido de criação de chave mas MIB não suporta a adição de mais chaves
- Erro #2 (`"NON EXISTENT GROUP"`): Grupo acedido da MIB não existe
- Erro #3 (`"NON EXISTENT VALUE (Sys/Conf)"`): Objeto não existe no grupo System ou Config
- Erro #4 (`"NON EXISTENT ENTRY"`): Entrada da MIB a aceder não existe
- Erro #5 (`"NON EXISTENT OBJECT IN ENTRY"`): Campo da entrada acedida na tabela não existe
- Erro #6 (`"VALUE WITH DIFFERENT TYPE"`): Valor a adicionar não é do mesmo tipo que o estipulado para o objeto
- Erro #7 (`"PRIMITIVE NOT SUPPORTED"`): Primitiva efetuada não corresponde a get ou set
- Erro #8 (`"OIDs len INCORRECT"`): Número de OIDS incorretos
- Erro #9 (`"SAME REQUEST_ID SENT IN V SECONDS"`): Manager não pode enviar o mesmo request id dentro de V segundos

Ficheiro de registo de clientes

Embora a criação deste ficheiro fosse sugerida para aplicar no TP2, decidi implementar já nesta fase pois é com recurso a este ficheiro que verifico se o cliente envia 2 pedidos com igual *request_id* dentro de um intervalo de tempo em segundos pré-definido.

No final do TP1, este ficheiro estará no formato JSON e em *plain text* (tendo em mente que para o TP2 este ficheiro terá de estar encriptado), onde cada entrada terá como chave a password do cliente (forma que encontrei para distinguir clientes), e o valor dessa entrada conterá o IP do cliente, a porta conectada pelo cliente, um ID que será atribuido automaticamente pelo programa (autoincrementável e apenas disponível na classe *Clients*), a sua password (informação que é redundante, mas não prejudica o funcionamento da classe) e uma lista de pedidos efetuados, onde cada elemento terá o ID do pedido e o timestamp em segundos.

clients.json

```
{
    "password": {
        "client_id": "1",
        "client_ip": "127.0.0.1",
        "client_port": 2048,
        "password": "password",
        "requests": [
            {
                "request_id": "1",
                "timestamp": 12
            }
        ]
    },
    "ola": {
        "client_id": "200",
        "client_ip": "127.0.0.1",
        "client_port": 2048,
        "password": "ola",
        "requests": [
            {
                "request_id": "200",
                "timestamp": 14
            }
        ]
    }
}
```

Figura 1: Exemplo de ficheiro de registo de clients (para o TP1)

1.2 TP2

Para este TP2, utilizei a abordagem sugerida pelo professor na última aula presencial de Gestão e Segurança de Redes: utilizei mecanismos criptográficos base que dependam apenas de segredos/chaves simétricas (ou chaves privadas).

Para tal, o cliente/manager precisará de uma *password* (que já tinha adicionado para o TP1) e de um ID (ou *username*) - embora o cliente já tivesse este ID, era atribuído automaticamente pela classe *Clients*, era autoincrementável e, tal como referido, não trazia nenhuma informação útil para o TP1. De seguida, será calculada o checksum a partir de uma string concatenada constituída pelo username do cliente e da password. De seguida, os valores do username e do checksum irão encriptados na PDU na secção `n_security_parameters_list`, mas a password irá em plain text, o que pode ser considerado uma grave falha de segurança. Optei por deixá-la em texto visível uma vez que esta palavra passe irá ser útil para desencriptar a password e o username no lado do agente. Do lado do servidor/agente, este irá receber estes três parâmetros de segurança e, juntamente com a série de verificações que já fazia

no TP1, passará também a verificar se aquele utilizador é realmente quem diz ser (se o checksum calculado é o mesmo que o checksum recebido na PDU) e também irá verificar se um dado cliente coloca a sua password e não uma outra qualquer - uma password válida é igual à primeira inserida na primeira ligação do cliente ao servidor. Além disso, o ficheiro de registo de clients está encriptado com a password do servidor, mas adicionei a possibilidade de desencriptar esse ficheiro de forma a um humano poder analisá-lo.

1.2.1 Cálculo do checksum

De forma a calcular o checksum de uma string em Python, utilizei a biblioteca *hashlib*, recorrendo a uma função de hash SHA-256, cuja documentação e respetivo código de implementação poderá ser consultado em [1].

1.2.2 Encriptação/desencriptação de strings

De forma a encriptar/desencriptar strings em Python, adaptei o código de [2] de forma a ter uma função que me encriptasse/desencriptasse uma dada string utilizando uma palavra-passe. Para tal, utilizei a função de derivação de chaves (que na documentação consta o facto de ser uma função "perigosa") PBKDF2HMAC [3] para utilizar uma chave de 32 bytes a partir da palavra passe do servidor ou do cliente sabidas de antemão, uma vez que utilizei o mecanismo de encriptação Fernet, que impede que uma string seja lida sem a utilização de uma chave (*URL-safe base64-encoded 32-byte key*[4]). Além disso, também adaptei estas funções de encriptar/desencriptar strings em Python de forma a aplicar estes mecanismos a dicionários que irão ser escritos num ficheiro JSON.

1.2.3 Correções à classe `Clients`

Tal como referido no TP1, esta classe contém um dicionário com todos os clientes alguma vez conectados ao agente. Será útil para determinar se um cliente enviou 1 ou mais pedidos com um dado *request_id* dentro de um intervalo de tempo pré-definido e, para este TP2, será útil para determinar se o manager está a inserir a password correta, i.e., a primeira password inserida na primeira ligação ao agente. Além disso, cada entrada do dicionário foi alterada: a chave passará a ser o *client_id/username* do gestor inserido em vez de ser a sua password. Por último, o conteúdo do ficheiro JSON de registo de clientes estará encriptado. Mais tarde, o agente poderá desencriptar este ficheiro usando a sua password (assim como encripta aquando do início da sua execução).

clients.json.encripted

```
gAAAAABkmIAq2k-eDLv5PWZ4rYdNY-
Oq8wronhLthIe_BPDqdJEh4UWP7y_arUv0K_CElDD8tfke0ctT2s0TduUXX1GBPHmZ3pf3hGjCTN1z55XmNj1tEWlnaT
o0HSiAQJIZ8d8BY-
jcKKxlkqIkbIjL_aK0mU_mrC0REcJNpKtv0v0OkHwPwi5kyu2r_U2Q96SVVW_Z56CO4iNSDeSVFkf3C3bJ-FvPoqiZ25-
njnpQ1Gw3a3oVMXI_UrEg_2VVPyGHeLz-rBkC6A-
skWapjPIXP4beLe3PMPYXI9i2Bg3bOonddZH_20C82LDWOwj3aqItqHJDLbwplWkJeZHgn-eef8DjOrQGG7Z-
f9ka1BjVc-jxbvxeeJbkcohsuAefTUCObT5D3eT_xyxjLi98_RvclvUCrxNvUNzFqK8O9o8D_feS0bHVmw-
wVoPxyPzHTXKDqHjDLjZtQo-dFO9tTU6KMs25o3h_6sHN3e12-R1Aa1u6kJomFyu-16iW7lCv4sCXSevJijU0uz-
H7xlbBUvoXYUZTJBGZbXqICL-kgkfriZcCR-
iFdNQTiYizzejs7sBJRNxgEbHQvwrauwdusdTUEM2Wz5loDOjAnI_Wsn13-VdnQ9t3oUDVuAvP-E2faYUm7uYDkCI
```

Figura 2: Exemplo de ficheiro de registo de clients encriptado (para o TP2)

clients.json

```
{
    "username1": {
        "client_ip": "127.0.0.1",
        "client_port": 2048,
        "password": "password1",
        "requests": [
            {
                "request_id": "1",
                "timestamp": 11
            }
        ]
    },
    "username2": {
        "client_ip": "127.0.0.1",
        "client_port": 2048,
        "password": "password2",
        "requests": [
            {
                "request_id": "42",
                "timestamp": 13
            }
        ]
    },
    "username3": {
        "client_ip": "127.0.0.1",
        "client_port": 2048,
        "password": "password3",
        "requests": [
            {
                "request_id": "1",
                "timestamp": 17
            }
        ]
    }
}
```

Figura 3: Exemplo de ficheiro de registo de clients desencriptado (para o TP2)

1.2.4 Manager

Para este TP2, o gestor terá de ser alterado de forma a incluir o seu *username*/*client_id*. Para tal, será adicionado um novo argumento à linha de comandos. De seguida, irá calcular o seu checksum através da função já referida anteriormente e encriptará o seu *client_id*

usando a palavra-passe do utilizador e o *checksum* usando a palavra-passe do utilizador. Posteriormente, estes valores serão adicionados à lista `n_security_parameters_list` da PDU pela ordem *client_id*, *password* e *checksum*. Por defeito, o valor de *security_level* será 1, contrariamente ao valor 0 tomado no TP1.

NOTA: O cliente terá numa variável guardada a palavra passe do servidor, o que é algo anormal, mas não encontrei outra forma do cliente utilizar a palavra passe do servidor para encriptar o *client_id*.

Exemplos de comandos efetuados (em Windows) - atualização

- Sintaxe abstrata de um comando:

```
python .manager.py [client_id]
[set/get] [ID_do_pedido]
[sequência_de_oids] [novo_valor - se pedido = set (para alterar valor)]
```

- para criar uma chave (comando SET):

```
python .manager.py username password set 1 3.2.6.0
```

- para alterar valor (comando SET):

```
python .manager.py username password set 4 2.1 42
python .manager.py username password set 7 3.2.4.2 novo_valor
```

- para obter valor (comando GET):

```
python .manager.py username password get 12 2.1
```

Construção da PDU - atualização

A PDU não sofreu muitas alterações em relação TP1 - apenas foram adicionadas elementos de segurança. A cinzento irão estar os elementos inalterados e a preto vão estar os elementos adicionais. Sabendo que argumentos um cliente usa na linha de comandos, a PDU de envio de pedido é simples de se entender:

PDU do pedido	
security_level	1
n_security_parameters_number	comprimento de *n_security_parameters_list*
n_security_parameters_list	[*client_id* encriptado, password do cliente, *checksum* encriptado]
request_id	valor inserido pelo cliente
primitive_type	1 = get; 2 = set
instance_elements_size	comprimento de *instance_elements_list*
instance_elements_list	sequência de OIDs inseridas pelo cliente + novo valor caso seja um pedido de set()
error_elements_size	[ ]
error_elements_list	0

1.2.5 Agente

O primeiro passo a tomar no agente foi a elaboração da sua password. Adicionei mais uma variável de instância à classe *Configurations* e o agente irá ler o valor a partir daí.

De seguida, extraio os valores do *client_id*, password do cliente e checksum enviado pelo mesmo através da PDU.

Posteriormente, a primeira verificação que o utilizador irá fazer é verificar se o checksum recebido na PDU é o correto. Para tal, calculo o checksum através do *client_id* e a password do cliente (uma vez que tenho todos os dados necessários para efetuar esse cálculo) e comparo com o valor contido na PDU. Caso sejam o mesmo, a série de verificações continua. Caso contrário, foi criado um novo código de erro que significa que o checksum não corresponde ao recebido, enviando a resposta para o cliente com esse código.

Por último, de forma a que um dado cliente não possa enviar outras palavras-passe que não a sua, foi adicionada uma verificação na função já existente `can_send_same_requestID()` - entenda-se que o cliente tem de inserir a primeira palavra-passe inserida na primeira ligação ao cliente. Caso tenha enviado uma password errada, foi adaptado o código de erro #9 de forma a indicar ao cliente que também poderá ocorrer o erro de password errada.

Neste TP, o servidor terá duas funções: receber pedidos de gestores e adicionar cada registo no ficheiro JSON que estará encriptado - a escrita de cada entrada do ficheiro está encriptada; desencriptar apenas o ficheiro JSON - o objetivo desta funcionalidade será para apenas consultar o conteúdo do ficheiro e precisará de inserir a sua palavra-passe de forma a poder desencriptar o conteúdo:

- formato abstrato do comando:

  ```
  python .\agent.py [password_do_agente] [flag_de_encriptação]
  ```

- inicializar agente com ficheiro de registo de clientes encriptado:

  ```
  python .\agent.py server_gsr -encrypt
  ```

- Agente apenas desencripta conteudo do ficheiro JSON com a password:

  ```
  python .\agent.py server_gsr -decrypt
  ```

Construção da PDU - atualização

Para este TP, a PDU não mudou drasticamente, sendo que apenas foi adicionados os valores relativos a segurança. Semelhantemente à PDU do gestor, a cinzento estará apresentado os elementos inalterados e a preto os valores atualizados.

PDU de resposta	
security_level	1
n_security_parameters_number	comprimento de *n_security_parameters_list*
n_security_parameters_list	[*client_id*, password e checksum inseridos pelo gestor]
request_id	valor inserido pelo cliente
primitive_type	0 = response
instance_elements_size	comprimento de *instance_elements_list*
instance_elements_list	sequência de OIDs inseridas pelo cliente + valor consultado do comando get()/ *keyId* da chave criada do comando set()
error_elements_size	[string que contém um código de erro (se aplicável)]
error_elements_list	comprimento de *error_elements_size*

Códigos de erro

Neste TP adicionei um novo código de erro e adaptei um dos códigos já existentes para simbolizar 2 erros numa só mensagem (que não é o ideal).

- Erro #1 (`"MIB FULL"`): Pedido de criação de chave mas MIB não suporta a adição de mais chaves
- Erro #2 (`"NON EXISTENT GROUP"`): Grupo acedido da MIB não existe
- Erro #3 (`"NON EXISTENT VALUE (Sys/Conf)"`): Objeto não existe no grupo System ou Config
- Erro #4 (`"NON EXISTENT ENTRY"`): Entrada da MIB a aceder não existe
- Erro #5 (`"NON EXISTENT OBJECT IN ENTRY"`): Campo da entrada acedida na tabela não existe
- Erro #6 (`"VALUE WITH DIFFERENT TYPE"`): Valor a adicionar não é do mesmo tipo que o estipulado para o objeto
- Erro #7 (`"PRIMITIVE NOT SUPPORTED"`): Primitiva efetuada não corresponde a get ou
- Erro #8 (`"OIDs len INCORRECT"`): Número de OIDS incorretos
- Erro #9 (`"SAME REQUEST_ID SENT IN V SECONDS| WRONG PASSWORD"`): Manager não pode enviar o mesmo request id dentro de V segundos ou a password inserida não é igual à primeira
- Erro #10 (`"WRONG CHECKSUM"`): Checksum mostra que cliente não é quem diz ser

2 Definição da MIB

De forma a implementar a MIB que consta no enunciado, irei utilizar hierarquia de classes. O primeiro passo foi criar uma estrutura de dados *MIB* que contém um dicionário para os 3 grupos da MIB: *system*, *config* e *data*. Cada chave do dicionário será o OID para o respetivo

grupo - as chaves estão *hard coded*, ou seja, não faço o parse das linhas da MIB que me indicam quais os OIDs que se referem a cada grupo.

Investigando, agora, os valores deste dicionário, observamos que temos uma classe *System*, que contém um dicionário de objetos referentes ao grupo System da MIB - onde faço o parse através de uma expressão regular. Também contém valores (como K, TTL, timestamps, ...) que serão necessários para atribuir valores a esses objetos (mais tarde explicarei como esta atibuição de valores funcionará).

De igual modo, também existe uma classe Config, cujo procedimento de criação de objetos é análogo à do grupo System, mas desta vez é aplicado ao grupo Config - também é utilizado uma expressão regular para dar parse dos objetos da MIB, os objetos também serão adicionados a um dicionário e também contém valores (como *master_key* ou *fst_ascii_code*) que serão passados aos objetos deste grupo da MIB.

Por último, irei falar do grupo *data* da MIB. De forma a implementar este grupo, terei a classe *MIB_Keys* que contém os objetos do grupo *data*, mas, neste caso, não optei por utilizar expressões regulares e criei os objetos de forma *hard coded*, uma vez que possuem informação pouco útil - a tabela e o número de chaves válidas irão estar noutras classes e terão esta informação. Assim, um dos objetos do grupo *data* (reforço que todos os objetos tanto do grupo *data* como dos outros grupos serão guardados em dicionários, uma vez que o acesso a estes elementos é imediado - de complexidade $\theta(1)$) conterá a tabela onde guardamos as entradas dos clientes/*managers* que façam pedidos ao agente. Quando o agente adiciona uma entrada, ele verificará se existem entradas expiradas, ou seja, entradas cujo *keyExpirationTime* foi atingido, removendo-as da tabela; de seguida, criará um tuplo `(entrada na tabela, timestamp em segundos)` e adicionará este valor ao dicionário de entradas na tabela, cuja chave será o ID da chave gerada (*keyId).*

Quanto à implementação dos objetos da MIB, o raciocício foi semelhante ao já efetuado com os grupos *system* e *config*: faço parse dos objetos através de uma expressão regular e preencho os valores da classe *MIB_Object* com estes valores obtidos no *parsing.* Além disso, de forma a permitir que cada objeto guardasse um valor, criei uma variável de instância *value*, permitindo que os managers consigam fazer pedidos de *set()* e de *get()* a estes objetos.

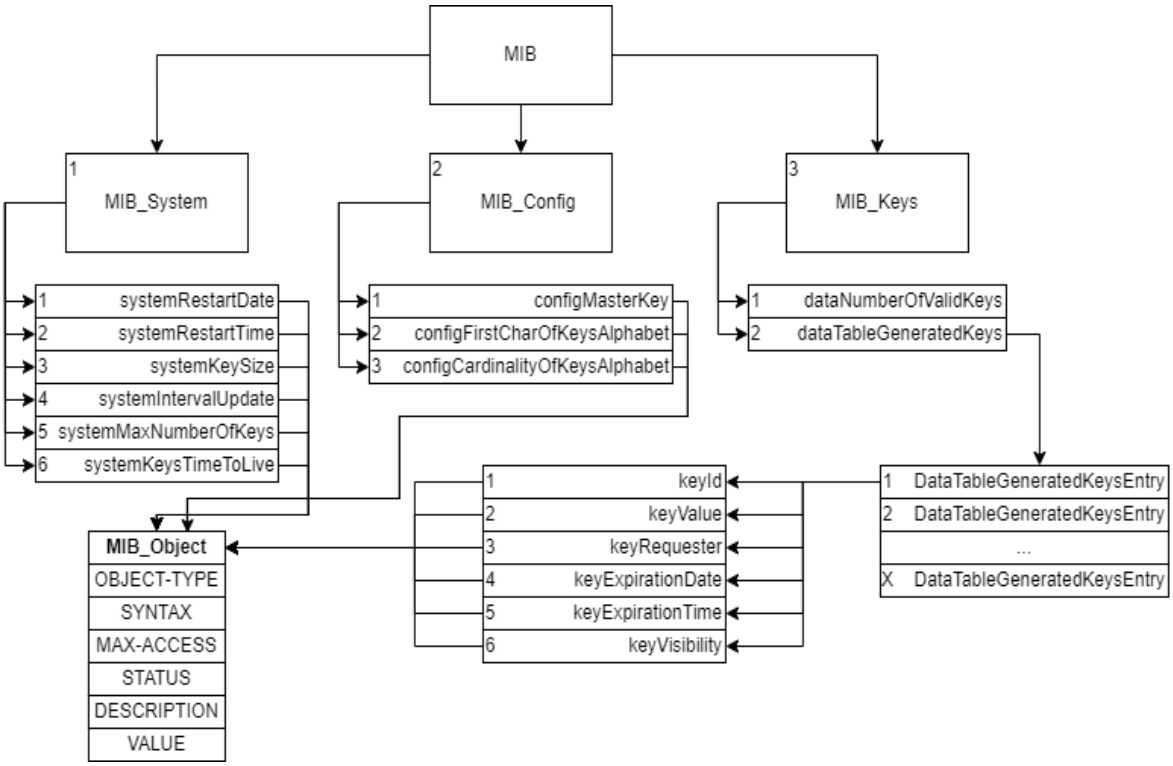

Figura 4: Esquema da estruturação da MIB

Segue, abaixo, um exemplo de uma tabela criada com pedidos de managers enviados ao agente:

keyId	keyValue	KeyRequester	keyExpirationDate	keyExpirationTime	keyVisibility
1	JQbo&^f"	127.0.0.1	1687366195	64195	0
2	u/lut0uY	127.0.0.1	1687366196	64196	0
3	G/$/(j&l	127.0.0.1	1687366202	64202	0
4	7b79C⌈Fb	127.0.0.1	1687366208	64208	0

Figura 5: Exemplo de tabela

Note-se que o campo *keyId* é um valor inteiro autoincrementável que identifica uma entrada na tabela; *keyValue* representa a chave criada através das matrizes; *keyRequester* representa o endereço IP do cliente; *keyExpirationDate* representa o timestamp da altura em que a entrada foi criada (apenas considerando o ano, mês e dia) em segundos; *keyExpirationTime* representa o mesmo timestamp de *keyExpirationDate*, mas agora considerando a hora, o minuto e segudo do timestamp (este valor estará em segundos); o valor *keyVisibility*, por defeito, estará a 0.

3 Funções/classes implementadas

3.1 TP1

3.1.1 configurations.py

```
class Configurations:
    regex_n_matrix        = r'K:\s*(\d+)'
    regex_master_key      = r'M:\s*(.+)'
    regex_update_itv      = r'T:\s*(\d+)'
    regex_max_store_time  = r'V:\s*(\d+)'
    regex_n_max_entries   = r'X:\s*(\d+)'
    regex_port            = r'Port:\s*(\d+)'

    def __init__(self, filename):
        with open(filename, 'r') as f:
            f_lines = f.readlines()

        self.n_matrix = int(self.get_value(''.join(f_lines), self.regex_n_matrix))
        self.master_key = self.get_value(''.join(f_lines), self.regex_master_key)
        self.update_interval = int(self.get_value(''.join(f_lines), self.regex_update_itv))
        self.max_store_time = self.get_value(''.join(f_lines), self.regex_max_store_time)
        self.n_max_entries = int(self.get_value(''.join(f_lines), self.regex_n_max_entries))
        self.port = int(self.get_value(''.join(f_lines), self.regex_port))
        self.min = 33
        self.max = 126
```

Esta classe é a responsável por ler as configurações iniciais expressas no ficheiro de configuração. Este possui uma configuração por linha do tipo `[variável] = [valor]`. Assim, utiliza várias expressões regulares para ler os dados e atribuir os respetivos valores às variáveis de instância, que serão úteis para os outros módulos do programa. Além disso, a classe também possui duas variáveis *min* e *max* que determinam o intervalo que os caracteres ASCII da matriz *fm* pode tomar.

3.1.2 MIB/main.py

```
class MIB:
    def __init__(self, filename, K, updating_interval, max_keys, ttl, master_key,
                 fst_ascii_code, number_of_chars):
        self.dictionary = {}
        self.dictionary[1] = system.MIB_System(filename, K, updating_interval,
                                               max_keys, ttl)
        self.dictionary[2] = config.MIB_Config(filename, master_key,
                                               fst_ascii_code, number_of_chars)
        self.dictionary[3] = keys.MIB_Keys()
```

Esta classe será responsável por representar a MIB. Contém um dicionário onde cada chave será o OID do grupo (valor que está *hard coded*) e cada valor será uma estrutura de dados que representa um grupo.

3.1.3 matrix.py

```
def create_m1_m2(n,master_key)
def create_matrix_Za(n,m1)
def create_matrix_Zb(n, m2)
def create_matrix_Zs(n, S)
def create_matrix_Z(n,Za,Zb,Zs,fm):
def get_matrix(n,master_key, fm_matrix, S)
```

Este módulo será responsável pela criação da matriz Z. A primeira função devolverá um tuplo onde o primeiro membro possuirá uma lista com os primeiros n bytes e, no segundo membro, ficarão os últimos n bytes. A segunda função gerará a matriz Z_a com a primeira lista da função anterior e a terceita função gerará a matriz Z_b com a segunda lista do tuplo. De seguida, irá ser criada a matriz Z_s, cujos elementos serão aleatórios no intervalo [33,126] será criada a matriz Z, com recurso às matrizes criadas anteriormente. A última função será responsável por juntar todas as funções anteriormente criadas.

3.1.4 update_matrix.py

```
def first_update_Z(n, Z)
def second_update_Z(n, Z)
def update_matrix_Z(n, Z, timestamp):
    first_update_Z(n, Z)
    second_update_Z(n, Z)
    time.sleep(timestamp/1000)
```

Este módulo será responsável por atualizar a matriz Z de T em T ms, aplicando a algoritmia requerida para cada atualização à matriz Z.

3.1.5 pdu.py

```
class PDU:
    def __init__(self, request_id, primitive_type, instance_elements_size,
                 instance_elements_list,
                 error_elements_size, error_elements_list, security_level=0,
                 n_security_parameters_number=0, n_security_parameters_list=[]):
        self.security_level = security_level
        self.n_security_parameters_number = n_security_parameters_number
        self.n_security_parameters_list = n_security_parameters_list
        self.request_id = request_id
        self.primitive_type = primitive_type
        self.instance_elements_size = instance_elements_size
        self.instance_elements_list = instance_elements_list
        self.error_elements_size = error_elements_size
        self.error_elements_list = error_elements_list
```

Esta classe será responsável pela PDU que será transmitida entre os gestores e o agente. Possui várias variáveis de instância com o nome autoexplicativo e, neste TP, os parâmetros de segurança serão todos definidos a 0 e/ou [ ]. No entanto, tal como explicarei na secção

seguinte, foi necessário incluir uma password por parte dos gestores de forma a poder distinguir diferentes clientes e o campo *n_security_parameters_list* foi escolhido - nas PDUs de resposta, estes parâmetros de segurança irão estar todos com valores *default* nulos.

3.1.6 client_registry.py

```
class Client_Registration:
    def __init__(self, client_ip, client_port, client_id, password):
        self.client_ip = client_ip
        self.client_port = client_port
        self.client_id = client_id
        self.password = password
        self.requests = []

    def add_request(self, request_id, timestamp):
        self.requests.append([request_id, timestamp])

class Clients:
    def __init__(self):
        self.clients = {}
        self.start_time = int(time.time())
        self.initialize_json_file()
```

Estas duas classes serão utilizadas para registar todos os clientes que se conectaram ao agente. Inicialmente, tinha pensado apenas em criar estas duas classes para o TP2, mas, devido à necessidade de impedir que um cliente envie dois pedidos com o mesmo *request_id*, foram criadas desde já - para o TP2, deverá ser apenas necessário adicionar mais variáveis de instância.

Assim, a classe *Client_Registration*, conterá o IP do cliente, a porta conectada pelo cliente, um ID autoincrementado automaticamente, a sua password e uma lista de pedidos efetuados por esse cliente - cada lista irá ser um tuplo do tipo `(request_id, timestamp em segundos)`. Este timestamp em segundos refere-se ao número de segundos que passaram desde que o agente foi inicializado (na prática, refere-se ao início da instância de *Clients*, mas a diferença de tempo é insignificante, pois trata-se de uma questão de milissegundos).

Com isto, a classe *Clients* terá um dicionário de clientes, cuja chave será a password de um certo cliente e o valor será um objeto da classe *Client_Registration*; a sua inicialização levará ao início de um relógio; todos os clientes serão registados num ficheiro JSON, que, neste TP1, se encontra em *plain text*.

Observação: Existe a assunção de que clientes diferentes não possuem a mesma password, o que na realidade pode não acontecer, não sendo esta a melhor prática a adotar no projeto. Além disso, a variável *client_id* é redundante, uma vez que não traz informação útil, mas também não prejudica o funcionamento do programa.

3.1.7 manager.py

```
class Manager:
    def __init__(self):
        self.AGENT_HOST  = 'localhost'
```

```
4           self.AGENT_PORT  = 2048
5           self.BUFFER_SIZE = 8192
```

De forma a implementar o gestor, criei a classe *Manager*, que contém o endereço IP a usar - tanto gestores como agentes irão correr na nossa máquina, pelo que vão usar o mesmo IP, a porta a usar na ligação - terá de ser a mesma que a do agente de forma a que os sockets possam comunicar, e o tamanho alocado para que os sockets possam transportar a informação das PDUs. De forma a executar o gestor, será necessário escrever vários argumentos consoante o intuito do pedido, que podem ser consultados na secção 1.1.3.

3.1.8 agent.py

```
1   class Agent:
2
3       def __init__(self):
4           self.n_lock           = threading.Lock()
5           self.n_updated_times  = 0
6           self.bufferSize       = 8192
7           self.HOST             = 'localhost'
8           self.current_time     = None
9
10      def handle_request(self, sock, data, addr, F, mib, client_registry)
```

Esta classe será responsável por implementar o agente, que receberá pedidos de gestores e será responsável por enviar respostas. Este possui algumas variáveis de instância já explicadas na secção 1.1.4. Tal como explicado anteriormente, será lançada uma thread por cada cliente conectado, mas a resposta será atribuída de forma sequencial e não concorrentemente - o trabalho a executar por cada thread será efetuado na função `handle_request()`. Aqui, o agente recebe a PDU do gestor, calcula o timestamp do número de segundos em que o cliente está a executar (esta variável é partilhada por todas as threads, pelo que será necessário aplicar concorrência através de `threading.Lock()`), obter a password do cliente, verificar se não enviou um pedido com um mesmo *request_id* dentro de um intervalo de tempo, verificar se a lista de OIDs enviada é válida - a verificação aqui feita não é totalmente rigorosa pois apenas verifico se o comprimento da lista recebida de instâncias é menor que um determinado número (o que é verdade para a maior parte das situações), mas não verifico se os OIDs que o gestor escreve realmente existem nos respetivos grupos da MIB (ou seja, para um pedido com os OIDs [7,3,2,6], esta função valida-o pois o comprimento da lista é menor que 4, mas deveria invalidar pois o OID 7 não existe na MIB). Além disso, também verifica que tipo de pedido foi feito pelo gestor, procedendo à respetiva lógica de resposta. Esta lógica consiste em acessos sucessivos às estruturas de dados da MIB já referidas anteriormente e, caso haja algum erro - por exemplo, aceder a um valor do dicionário que não existe - será reportado imediatamente uma mensagem de erro, ignorando outros possíveis erros que possam existir no pedido do cliente.

3.2 TP2

3.2.1 security/checksum.py

```
def get_checksum(message, key):
    if isinstance(message, bytes):
        message = message.decode('utf-8')
    if isinstance(key, bytes):
        key = key.decode('utf-8')
    mensagem = message + key
    hash_object = hashlib.sha256(mensagem.encode())
    checksum = hash_object.hexdigest()
    return checksum
```

O ficheiro *checksum.py* contém a função responsável por calcular o checksum de uma mensagem. Neste caso, a mensagem irá ser uma string concatenada do *client_id* com a palavra passe do client. Note-se que os dois argumentos necessários para o cálculo devem estar no formato string e não em bytes.

3.2.2 security/encrypted_data.py

```
def derive_key_from_password(password):
    salt = b'salt_value'
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        iterations=100000
    )
    key = kdf.derive(password.encode())
    return key

def encrypt_string(string, password):
    key = derive_key_from_password(password)
    fernet_key = base64.urlsafe_b64encode(key)
    fernet = Fernet(fernet_key)
    encMessage = fernet.encrypt(string.encode())

    return encMessage

def decrypt_string(string, password):
    key = derive_key_from_password(password)
    fernet_key = base64.urlsafe_b64encode(key)
    fernet = Fernet(fernet_key)
    decMessage = fernet.decrypt(string).decode()

    return decMessage

def encrypt_file(file_path, password)
def decrypt_file(file_path, password)
```

Este módulo está responsável pela encriptação/desencriptação das mensagens. De forma a utilizar como chave de encriptação as palavras-passe do servidor ou de cada cliente, foi

necessário derivar uma chave Fernet através da função `derive_key_from_password()`. De seguida, encripto/desencript a mensagem que desejar. Também permito aplicar mecanismos criptográficos a ficheiros usando as funções *encrypt_string()* e *decrypt_string()* anteriormente definidas, que serão aplicadas ao ficheiro JSON como já referido.

3.2.3 client_registry.py - atualizações

```
class Clients:
    def __init__(self, filename, server_password):
        #variáveis de instância já existentes
        self.initialize_json_file(server_password)

    def initialize_json_file(self, server_password)

    def add_client(self, client_id, client_ip, client_port,
                   request_id, password, server_password):
        #Resto do código
        self.save_to_json(server_password)


    def save_to_json(self, server_password):
        data = {}
        for client_id, client in self.clients.items():
            data[client_id] = {
                'client_ip': client.client_ip,
                'client_port': client.client_port,
                'password': client.password,
                'requests': []
            }
            for request in client.requests:
                request_id, timestamp = request
                data[client_id]['requests'].append({
                    'request_id': request_id,
                    'timestamp': timestamp
                })

        encrypted_data = enc.encrypt_string(json.dumps(data), server_password)

        with open(self.filename, 'wb') as file:
            file.write(encrypted_data)

    def can_send_same_requestID(self, client_id, request_id, max_time, client_password):
           if client_id not in self.clients:
               return True
           client = self.clients[client_id]
           if client.password == client_password:
               #Código já existente
           else:
               return False
```

Neste módulo, a classe *Clients* sofreu algumas alterações: a chave de cada entrada do dicionários *clients* passa a ser o username/*client_id* do gestor; cada entrada do ficheiro JSON será escrita de forma encriptada recorrendo à password do servidor; na função *can_send_same_requestID()* verifico o *request_id* do gestor apenas se a password inserida por ele for a já registada no

ficheiro JSON - caso não seja, o erro que desta função viria passa a incluir o erro de password errada.

3.2.4 manager.py - atualizações

```
class Manager:
    def __init__(self):
        #variáveis de instância já existentes


if __name__ == "__main__":
    if len(sys.argv) < 6:
        #Erro caso não se insira o número de argumentos corretos
    else:

        # Atribuição dos argumentos às respetivas variáveis
        client_id = sys.argv[1]
        password = sys.argv[2]
        action = sys.argv[3]
        request_id = sys.argv[4]
        version_numbers = sys.argv[5]

        #Código já existente

    mn = Manager()

    #Calculo do checksum do cliente usando a sua password
    password_from_server = "server_gsr"
    checksum = checksum.get_checksum((client_id).encode(), password)

    n_security_parameters_list = []

    #Client_id encriptado com a password do servidor
    client_id = enc.encrypt_string(client_id, password_from_server)
    n_security_parameters_list.append(client_id)

    n_security_parameters_list.append(password)

    #Checksum encriptado com a password do cliente
    checksum = enc.encrypt_string(checksum, password)
    n_security_parameters_list.append(checksum)
    n_security_parameters_number = len (n_security_parameters_list)

    #Preparação da PDU com os novos elementos
    #Envio da PDU e código já existente
```

Tal como já referido, o gestor passará a ter uma palavra passe. Para isso foi adicionado mais um argumento à lista de argumentos inserida na linha de comandos. Além disso, foi calculado o checksum com recurso à password do agente que está guardada numa variável - não é a melhor forma de o fazer, mas não encontrei outro método para o gestor saber a password do agente. De seguida, encripta o seu username e o checksum calculado, adiciona-os na PDU e envia para o agente, ficando à espera de uma resposta.

3.2.5 agent.py - atualizações

```
if __name__ == "__main__":

    # Leitura do ficheiro
    F = configurations.Configurations("config.conf")

    if len(sys.argv) == 3 and sys.argv[1] == F.server_password
                          and sys.argv[2] == "-encrypt":
        #Código já existente

        enc.encrypt_file("clients.json", sys.argv[1])
        client_registry = client_registry.Clients("clients.json", F.server_password)

        #Código já existente
    elif len(sys.argv) == 3 and sys.argv[1] == F.server_password
                            and sys.argv[2] == "-decrypt":
        enc.decrypt_file("clients.json", F.server_password)
    else:
        print("Password inválida!")
```

No excerto acima, o agente terá duas funcionalidades: a primeira consiste apenas na adição do facto de ter o ficheiro JSON encriptado e, para tal, terá de escrever a password correta e terá de usar a flag `-encrypt`; já a segunda consiste em simplesmente desencriptar o ficheiro JSON já criado e com conteúdo, recorrendo, novamente, à sua palavra-passe e à flag `-decrypt`.

De forma a esclarecer o trabalho executado pelo o agente em cada ligação com um gestor, segue-se um excerto em pseudo-código de todo o trabalho executado:

```
def handle_request():
    #Codigo anterior
    #Obtenção dos dados de segurança do TP2: client_id, password e checksum
    if(valida_checksum() == True):
        if(cliente_pode_enviar_pedido_com_certo_req_ID() == True):
            client_registry.add_cliente_ao_ficheiro_json()
            if(valida_lista_de_oids_da_pdu_recebida() == True):
                if criacao_de_chave_na_tabela() == True:
                    #Código de criação de chaves
                    else:
                        envia_PDU_de_erro("MIB FULL")
                elif agente_fez_get() == True:
                    try:
                        grupo_acedido = devolve_grupo_da_mib(oids[0])
                    except Exception as _:
                        envia_PDU_de_erro("NON EXISTENT GROUP")
                    if (get_para_grupos_system_ou_config() == True):
                        try:
                            valor_do_objeto =
                                devolve_valor_do_objeto_do_grupo_system_ou_config(oids[1])
                        except Exception as _:
                            envia_PDU_de_erro("NON EXISTENT VALUE (Sys/Conf)")
                    elif (get_para_objetos_da_tabela() == True):
                        try:
                            valor_do_campo_da_entrada =
                            devolve_valor_do_campo_da_entrada_da_tabela_procurada(oids[2])
```

```
                        except Exception as _:
                            envia_PDU_de_erro("NON EXISTENT ENTRY")
                        try:
                            valor_do_objeto_da_entrada =
                                devolve_valor_do_objeto_da_entrada(oids[3])
                        except Exception as _:
                            envia_PDU_de_erro("NON EXISTENT OBJECT IN ENTRY")
                    #Não foi detetado erro nenhum
                    pdu_resposta = preparacao_pdu_resposta()
                    envia_pdu_pelo_socket(pdu_resposta)
                elif agente_fez_set() == True:
                    try:
                        grupo_acedido = devolve_grupo_da_mib(oids[0])
                    except Exception as _:
                        envia_PDU_de_erro("NON EXISTENT GROUP")
                    if (get_para_grupos_system_ou_config() == True):
                        try:
                            objeto_do_grupo =
                                devolve_objeto_do_grupo_system_ou_config(oids[1])
                        except Exception as _:
                            envia_PDU_de_erro("NON EXISTENT VALUE (Sys/Conf)")
                        try:
                            valor_a_adicionar_a_PDU = set_do_novo_valor(novo_valor)
                        except Exception as _:
                            envia_PDU_de_erro("VALUE WITH DIFFERENT TYPE")
                    elif (get_para_objetos_da_tabela() == True):
                        try:
                            entrada_da_tabela =
                                devolve_entrada_da_tabela_a_aceder(oids[1])
                        except Exception as _:
                            envia_PDU_de_erro("NON EXISTENT ENTRY")
                        try:
                            campo_da_entrada = devolve_campo_da_entrada(oids[2])
                        except Exception as _:
                            envia_PDU_de_erro("NON EXISTENT OBJECT IN ENTRY")
                        try:
                            valor_a_adicionar_a_PDU =
                                set_do_novo_valor_no_objeto_da_tabela(novo_valor)
                        except Exception as _:
                            envia_PDU_de_erro("VALUE WITH DIFFERENT TYPE")
                    #Não foi detetado erro nenhum
                    pdu_resposta = preparacao_pdu_resposta()
                    envia_pdu_pelo_socket(pdu_resposta)
                else:
                    envia_PDU_de_erro("PRIMITIVE NOT SUPPORTED")
            else:
                envia_PDU_de_erro("OIDs len INCORRECT")
        else:
            envia_PDU_de_erro("SAME REQUEST_ID SENT IN V SECONDS | WRONG PASSWORD")
    else:
        envia_PDU_de_erro("WRONG CHECKSUM")

```

4 Conclusões

Em traços gerais, considero que o trabalho prático contém as funcionalidades pedidas nos dois enunciados, embora hajam alguns aspetos negativos, que já foram referidos ao longo do relatório.

No TP1, gostaria de ter implementado uma outra forma de distinguir um cliente sem ser através pela sua password, uma vez que na vida real, clientes diferentes podem ter a mesma password (mas dificilmente isso acontece). Além disso, deveria ter sido mais rigoroso em algumas verificações que faço em funções *booleanas*, tal como a verificação da lista de OIDs inserida ser válida apenas pelo número de elementos inseridos. Também deveria ter mais cuidado com a escrita do ficheiro de registo de clientes conectados, pois é anormal uma entrada para um cliente ter como chave a sua password e não um identificador como o *client_id*.

No TP2, considero que consegui implementar as ideias sugeridas pelo docente na última aula presencial, apesar de ter alguns pontos onde o risco de segurança é maior: password em *plain text* e a password do servidor guardada numa variável do código do gestor. Consegui, também, retificar alguns pontos do TP1, nomeadamente o ficheiro de registo de clientes, que enste momento já traz alguma informação útil para o servidor.

Como trabalho futuro, poderia tentar corrigir os pontos fracos que fui identificando ao longo do relatório. Além disso, será interessante impedir clientes consultarem entradas (seja com pedidos GET ou SET) que não foram criadas por si e ser mais restritivo na alteração dos valores dos objetos da MIB. Ou seja, quaisquer clientes podem alterar os valores dos grupos *system* ou *config* quando apenas o agente deveria alterar estes valores.

Bibliografia

[1] Documentação função de hash SHA-256 em Python: `https://docs.python.org/3/library/hashlib.html` (consultado em jun. 2023)

[2] Website GeeksForGeeks para encriptação/desencriptação de strings em Python: `https://www.geeksforgeeks.org/how-to-encrypt-and-decrypt-strings-in-python/` (consultado em jun. 2023)

[3] Documentação PBKDF2HMAC: `https://cryptography.io/en/latest/hazmat/primitives/key-derivation-functions/` (consultado em jun. 2023)

[4] Documentação Fernet: `https://cryptography.io/en/latest/fernet/` (consultado em jun. 2023)