Instituto Superior Técnico
Mestrado Integrado em Engenharia Aeroespacial
2º Semestre - 2020/2021

Relatório do projeto *Key-Value store*

Programação de Sistemas
Docentes:
João Nuno De Oliveira e Silva
André Gonçalves Mateus
Bruno Alexandre Morais Dias Ribeiro

23 de junho de 2021

Grupo 22 - Turno: L04

José Alberto Cavaleiro Henriques - 89684
Pedro Silva André - 89707

Conteúdo

Lista de Figuras

1 Objetivos

No âmbito da unidade curricular de Programação de Sistemas e de forma a aplicar os conhecimentos obtidos na mesma, um projeto de desenvolvimento de software foi realizado. Este consistia num servidor com capacidade de armazenar várias *key-value databases*, uma por grupo, que podiam ser acedidas e manipuladas por clientes. Além disto, o desenvolvimento de um servidor de autenticação tornou-se indispensável uma vez que a autenticação dos clientes era um dos objetivos. As especificações e requerimentos do projeto podem ser consultados na íntegra no enunciado do mesmo [1].

Sendo assim, o presente relatório tem como objetivo demonstrar o processo de design e desenvolvimento feito pelos alunos do grupo 22 de forma a criar um sistema que obedecesse aos requerimentos solicitados, bem como descrever dito sistema e justificar decisões tomadas acerca de parâmetros não especificados. O código desenvolvido pode ser encontrado em `https://github.com/PedroSAndre/PSis_20-21`.

2 Arquitetura

O sistema desenhado e criado foi desenvolvido em C com o objetivo de ser executado em sistemas operativos *UNIX*. Tal como já foi introduzido nos objetivos, este encontra-se dividido em três grandes sub-sistemas:

- **KVS-lib:** A biblioteca a partir da qual as aplicações que formarão os clientes serão desenvolvidas e que torna possível interagir com o servidor local (presente na mesma máquina) para introduzir, atualizar, obter, apagar ou registar *callback* de qualquer par *key-value* associado ao grupo a que a aplicação se conectou;
- **KVS-LocalServer:** Servidor central e local que irá conter todas as bases de dados *key-value*, irá aceitar ligações dos clientes e lidar com os seus pedidos. Também irá possuir uma interface que permite ao utilizador criar, eliminar e mostrar informação dos grupos assim como mostrar informação acerca das aplicações conectadas (incluindo as que se encontram desconectadas);
- **KVS-AuthServer:** Servidor que pode estar presente numa máquina distinta dos servidores locais e clientes que é responsável por gerar segredos para os grupos e autenticar aplicações que pretendam conectar-se a um dos mesmos. Para tal, precisa de guardar todos os grupos e respetivos segredos na sua memória, além de ser necessário atualizar os grupos sempre que um é eliminado por um servidor local.

De forma a tornar o mais evidente possível a função destes três subsistemas e a relação entre os mesmos, o diagrama do enunciado do projeto ([1]) foi expandido.

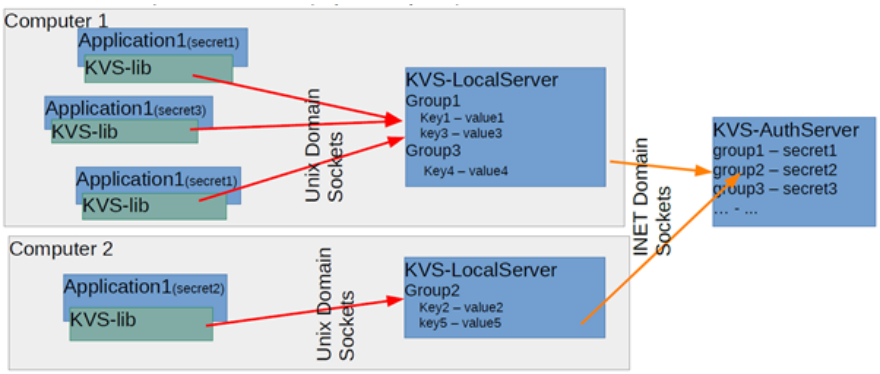

Figura 1: Estrutura geral do sistema quando ligado a várias aplicações, com destaque para os métodos de comunicação (*sockets*).[1]

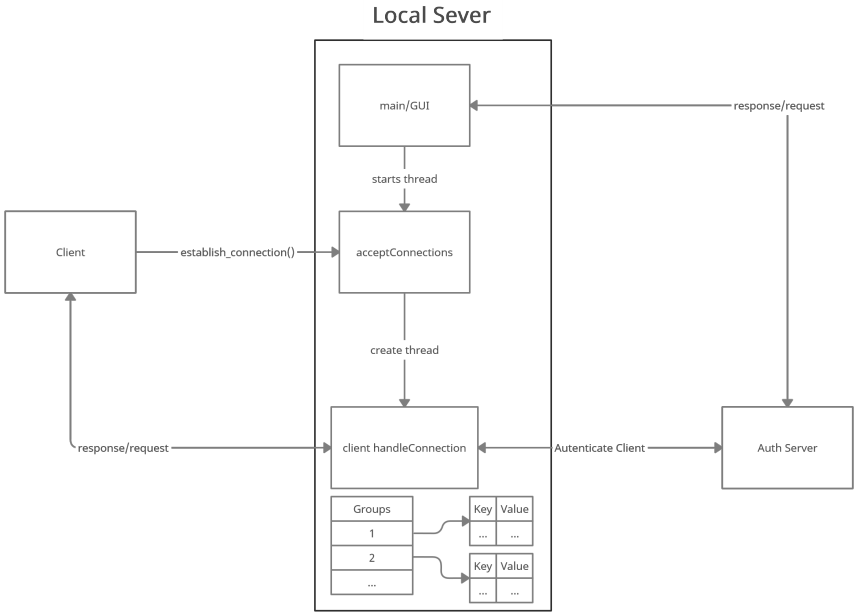

Figura 2: Estrutura específica do sistema quando é iniciada a ligação a um cliente, com destaque para a divisão dos vários sub-sistemas.

A forma como estes sub-sistemas se encontram divididos e a apresentação das suas estruturas serão abordados na próxima secção.

3 Implementação

Antes de iniciar a descrição de como cada sub-sistema foi desenhado e desenvolvido, destaca-se que o ficheiro *header* `Basic.h` foi utilizado em todos os ficheiros de código desenvolvido de forma a facilitar e uniformizar a inclusão de bibliotecas do sistema além de definir constantes necessárias para todo o projeto, como endereços, tamanhos máximos de strings e elementos e, por fim, *flags* de pedido e erro.

3.1 KVS-lib

O desenvolvimento da biblioteca a ser utilizada para gerar as aplicações encontra-se no ficheiro `KVS-lib.c` sendo o respetivo *header*, `KVS-lib.h`, fornecido pelo corpo docente da unidade curricular. Durante o desenvolvimento foi necessário utilizar uma variável global, `int client_sock`, de forma a guardar o *socket* após uma conexão estabelecida com sucesso, tornando assim possível posteriormente fazer pedidos ao servidor ou fechar a conexão. As funções desenvolvidas são as especificadas no enunciado, sendo que os erros que podem ser retornados e o seu significado encontram-se comentados no código antes do desenvolvimento de cada função.

Sendo assim, foi utilizado o paradigma de comunicação cliente-servidor, o que tornou o desenvolvimento de cada função além da `establish_connection` num processo de enviar o pedido pretendido para o servidor e os argumentos necessários e ler a resposta retornada, sempre verificando a ocorrência de erros, até mesmo para a função `register_callback` (cujo funcionamento tornar-se-á mais explícito na subsecção seguinte).

Apesar de o desenvolvimento de cada uma destas funções da biblioteca ter sido bastante intuitivo tendo em conta as especificações do enunciado, destaca-se que foi necessário ignorar o sinal `SIGPIPE` (*broken pipe*) que ocorria após uma quebra inesperada de ligação com o servidor local, através de `signal(SIGPIPE, SIG_IGN)` (ignora-se o sinal), de forma a que fosse possível através do retorno da função `write` notificar o utilizador/*developer* da aplicação que utiliza a biblioteca que a ligação caiu.

3.2 KVS-LocalServer

3.2.1 Estruturas de dados

O primeiro passo no planeamento do servidor central consistiu na definição das estruturas de dados. É importante referir que todas as strings exceptuando o valor (*data*) de cada chave, ou seja, chave, grupo e segredo, possuem limites de tamanho que podem ser alterados e encontram-se definidos em `Basic.h`, sendo que encontram-se definidos como 1024 inicialmente. Este valor nasce do facto de que cada *internet frame* tem um tamanho máximo de aproximadamente 1500 bytes, permitindo assim transmitir cada uma destas strings em apenas um destes *frames*, evitando perdas na mensagem.

De forma a guardar os *key-value*, decidiu-se que uma *hash table* iria obter o melhor compromisso entre rapidez de acesso e eficiência. Note-se que a tabela pode guardar infinitos elementos, visto que em caso de conflito, a respetiva entrada da tabela torna-se numa lista simplesmente ligada. Para tal, criou-se a estrutura `key_value` e os métodos correspondentes para inicializar e fazer *free* da tabela, inserir (ou atualizar caso a chave já exista na base de dados), obter e apagar elementos da mesma, além de permitir esperar por mudanças numa determinada chave (para o `register_callback`) e terminar todos os processos à espera de *callback* (através de `signal_all_callback` para o caso de ser necessário apagar o grupo, por exemplo, como será explicado mais à frente). Esta estrutura e respetivos métodos encontram-se em `key_value_struct.c` e `key_value_struct.h`. Ainda na estrutura `key_value`, é definida uma variável `mutex`, que irá ser utilizada para impedir problemas de sincronização (explicado em 5.2) e outra `cond` (*condition variable*) que será utilizada para implementar a funcionalidade de *callback* (explicado em 3.2.4).

Analogamente e pelas mesmas razões em `group_table_struct.c` e `group_table_struct.h` foi definida a estrutura `group_table` de forma a criar uma *hash table* responsável por guardar os pares grupo-tabela *key-value*. Destaca-se que todos os métodos são quase iguais aos da *hash table* descrita no parágrafo anterior, sendo que desta vez não existem variáveis nem métodos responsáveis por sincronização ou funcionalidades de *callback*.

Já para guardar os clientes, decidiu-se utilizar um vetor dinâmico (o que tornou necessário criar a variável global `int all_clients_connected` de forma a guardar o tamanho do vetor). Para tal, criou-se a estrutura `app_status`, em `app_status_struct.h` a partir da qual será constituído o vetor, responsável por guardar o ID do cliente e da thread que está a lidar com a ligação com o mesmo, o grupo a que se ligou o cliente e, finalmente, o tempo de conexão e desconexão (caso o cliente já sido corretamente desligado). Os métodos desenvolvidos em `app_status_struct.c` são responsáveis por adicionar um cliente, registar o tempo em que o cliente se desconectou, imprimir todos os clientes (presentes e passado) e, finalmente, esperar que todos os clientes de um determinado grupo desconectem-se.

Este último método é utilizado no caso de ser necessário eliminar um grupo, uma vez que a variável global `deleting_group` fica com o valor do grupo a eliminar o que faz com que todos os clientes terminem as suas conexões após *timeout* definido em `Basic.h` ou terminarem a presente tarefa (aqueles que estão à espera de *callback* também param de esperar graças ao método `signal_all_callback` descrito anteriormente). Contudo é necessário esperar que cada cliente termine a operação em que se encontra, daí ser necessário o método descrito.

3.2.2 Funções Auxiliares

Além das estruturas de dados referidas anteriormente, algumas funções auxiliares tiveram que ser definidas em `Localserver_aux.h` e desenvolvidas em `Localserver_aux.c`, de forma a facilitar a leitura do código e evitar repetição. Estas consistem na criação de *sockets* e respetivo *bind* utilizando o endereço definido em `Basic.h` para o socket que irá escutar ligações com os clientes e o IP e porta recebidos como argumentos da função `main` do servidor local. Além disso, funções que adicionam um *timeout* a ações de aceitar conexões, ler e receber mensagens foram estabelecidas. Por fim, a função de comunicação com o servidor de autenticação é definida. Esta recebe o pedido a ser feito ao servidor de autenticação (criação ou eliminação de grupo, autenticação...), os argumentos necessários

e devolve ou sucesso, ou a resposta do servidor ou ainda erro de *timeout* (resposta demorou muito tempo).

3.2.3 Funções de *threads*

Finalmente, já abordando o ficheiro `KVS-LocalServer.c`, existem duas funções dedicadas a *threads*: `acceptConnections` e `handleConnection`. A forma como este tipo de *threads* são criados, a sua função e como é feita a sua gestão é o tópico da secção 5.1.2.

3.2.4 *Callback*

Quando uma *thread* `handleConnection` recebe um pedido para registar um *callback* de uma determinada chave, esta chama o método `hashWaitChange_key_value` passando como argumento essa mesma chave. Este, por sua vez, se encontrar a chave passada como argumento, irá esperar que a variável condicional (`cond`) do primeiro elemento da lista ligada receba o sinal para desbloquear (bloqueando assim também o cliente que ficará à espera da resposta (*read locked*)) que irá ocorrer quando esta for eliminada ou subscrita. Sendo assim, quando a chave for subscrita ou apagada por outro cliente, um sinal será enviado para a variável `cond` que a irá desbloquear e notificar o cliente de que o valor associado à chave solicitada foi alterado. Desta forma, é possível esperar pela alteração da chave sem ocupar ciclos do CPU.

3.2.5 `main` do servidor local

A função `main` do servidor local é responsável por inicializar a tabela de grupos-tabelas *key-value* e a variável de estado dos clientes além de todas as outras variáveis globais. De seguida entra no ciclo da *UI* em que aceita e executa pedidos do utilizador. Por fim, quando o utilizador termina o servidor e todos os clientes terminam a sua operação atual ou são expulsos por ultrapassarem o *timeout*, o servidor liberta toda a memória alocada e termina a sua execução.

3.3 KVS-AuthServer

De forma semelhante ao que foi feito na secção anterior, começa-se por definir as estruturas de dados a utilizar. Sabendo que o servidor deverá guardar os pares grupos-segredo, efetuou-se uma *hash table* para estas variáveis, da mesma maneira que no servidor local. Com a existência de vários servidores e, devido ao facto de ser necessário duas mensagens para a autenticação, grupo e segredo, foi necessário criar uma lista de mensagens. Estas mensagens incluem o tipo de operação a efetuar, o grupo, o segredo e ainda o estrutura do endereço do remetente para onde será enviada a resposta.

O programa começa por inicializar a socket e criar um *thread* responsável por esperar um *input* do utilizador. A única função desta *thread* é dar o sinal de encerramento, `server_status`, ao ciclo de respostas, que irá encerrar o programa.

Após a inicialização a *thread* principal entra num ciclo onde esta irá sistematicamente a ler da *socket* um pedido, analisar o pedido pela função `recoverClientMessage`() para obter a mensagem e verificar o pedido, procedendo à operação explicita na mensagem. Para o caso da autenticação que requer duas mensagens, a primeira instância irá enviar sucesso em caso de ter recebido a primeira mensagem e irá colocar esta mensagem numa lista à espera da segunda mensagem deste servidor.

A função `recoverClientMessage`() vai percorrer a lista de mensagens à procura de uma mensagem com o mesmo remetente. Caso não encontre, isto significa que estamos perante a primeira mensagem deste pedido do remetente, depois de analisado o pedido do tipo "*flag*:*group*"coloca-se numa mensagem. Apenas se o pedido corresponder a uma autenticação, adiciona-se a mensagem à lista de mensagens de modo a prepará-la para a próxima mensagem. Esta função devolve a estrutura da mensagem com a operação a ser feita na *hash table*. Após a conclusão do processamento, procede-se à sua eliminação, incluindo da lista de mensagens se necessário.

As funções de manipulação da *hash table* são equivalentes às da secção anterior para a *hash table key_value*, com a particularidade de que a autenticação pode ser feita com um *get secret* para comparar com o *secret* proposto.

Finalmente, a terminação do servidor, causada por um erro ou a pedido do utilizador, é feita, saíndo do ciclo de receção. Daqui elimina todos os dados da mensagem e da *hash table* e encerra a socket. Note-se que para limitar o tempo de espera no encerramento a receção tem um *timeout* associado.

4 Comunicação

4.1 Descrição das Sockets

Como proposto no enunciado, foram implementadas *UNIX stream sockets* e *INET datagram sockets* para, respectivamente, a comunicação entre o servidor local e a KVS-lib e a comunicação entre o servidor local e o servidor de autenticação. Para as *stream sockets*, a conecção é feita numa socket com o endereço "sockets/localserver". A conecção com o servidor de autenticação é feito com a escolha de um porto para a conecção e um IP, correspondente ao IP do servidor de autenticação. Estes serão argumentos do programa para o servidor local e apenas o porto será necessário como argumento para o servidor de autenticação. Como os argumentos são strings é necessário fazer a conversão necessária com `htons(atoi())` do porto (`atoi()` conversão de string para inteiro e `htons()` de inteiro para `sin_port`) e `inet_addr`() do endereço IP (de string para `s_addr`).

4.2 *Sequence Diagrams* das funções do KVS-lib

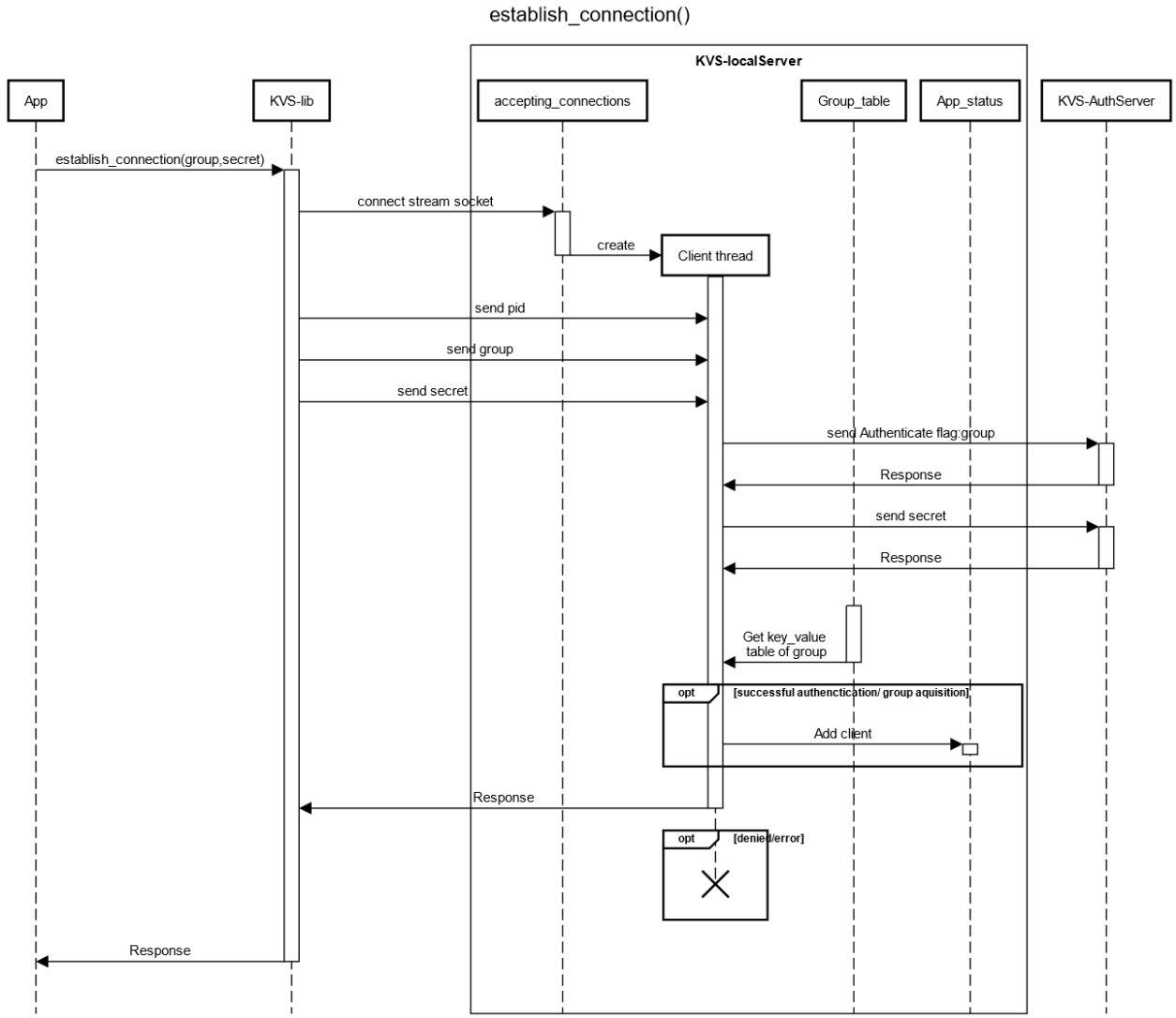

Figura 3: *Sequence Diagrams* para o establish_connection().

Note-se que *Group_table* e *App_status* não são módulos mas decidiu-se representar as operações em variáveis globais, pelo que decidiu-se incluir estes no diagrama apresentado. O mesmo foi feito para a *key_value_table* nos diagramas seguintes.

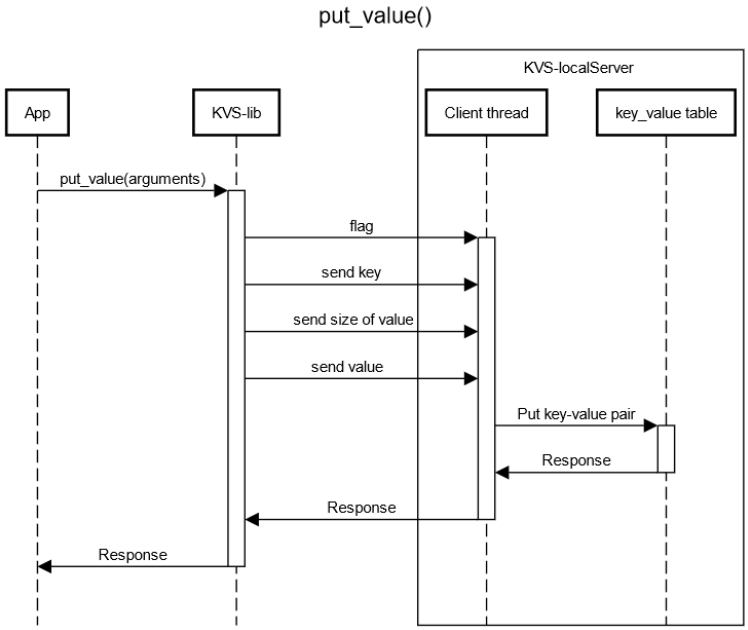

Figura 4: *Sequence Diagrams* para `put_value()`.

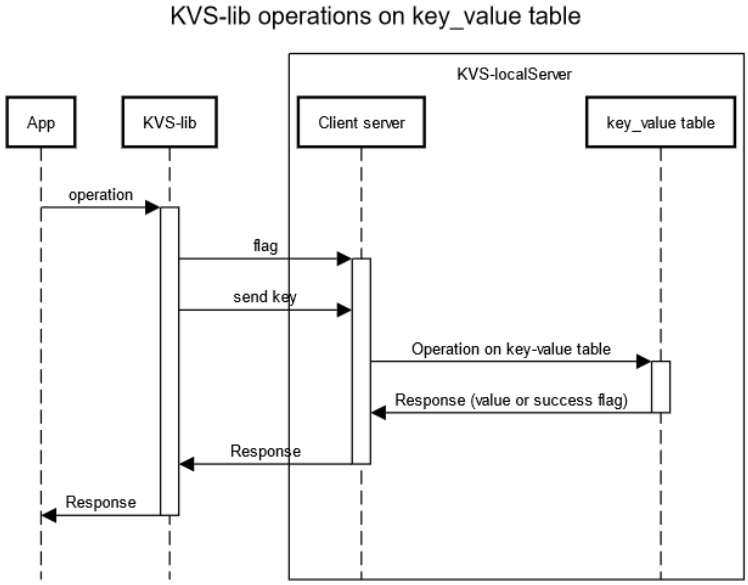

Figura 5: *Sequence Diagrams* para as operações `get_value()` e `delete_value()`.

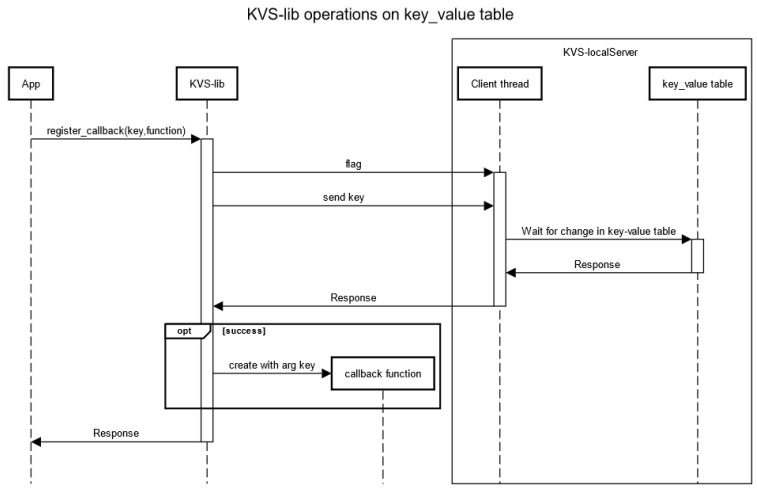

Figura 6: *Sequence Diagrams* para a função `register_callback()`.

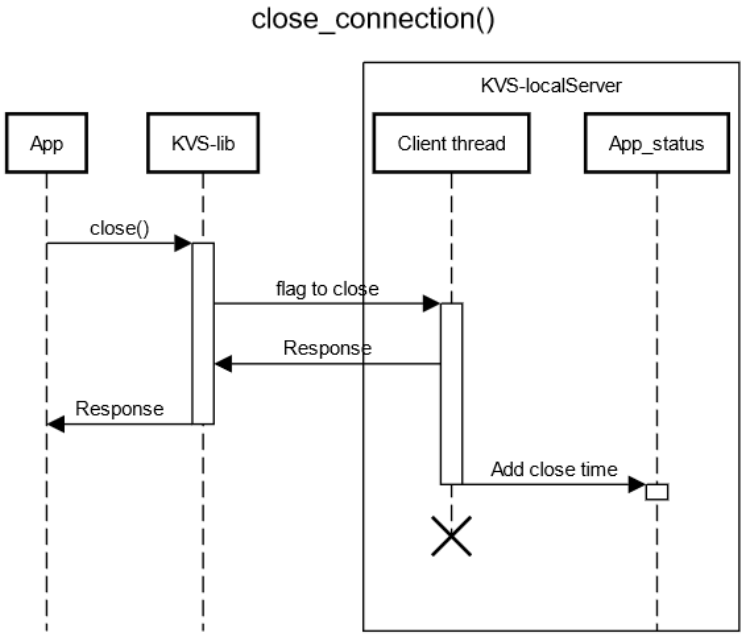

Figura 7: *Sequence Diagrams* para o `close_connection()`.

4.3 Protocolo de rede

Como são usadas *sockets* do tipo datagrama para a comunicação do servidor local com o servidor de autenticação, o protocolo usado para esta comunicação será UDP, figura 9. Este protocolo, faz com que o servidor de autenticação esteja constantemente em espera de mensagens de clientes, quando recebe uma mensagem fará o seu processamento e depois reenviará. Como foi escolhido um tamanho para a string *group* mais pequeno que o tamanho de um *ethernet frame*, pode-se assumir que, caso este *packet* chegue, este estará completo. Contudo, no caso de termos de enviar duas mensagens, par segredo e grupo, é necessário verificar de onde a mensagem vem, daí ter-se criado uma estrutura para salvaguardar várias mensagens. Note-se que envia-se o pedido juntamente com o grupo("*request*:*group*"), desta forma, apenas é necessário uma mensagem.

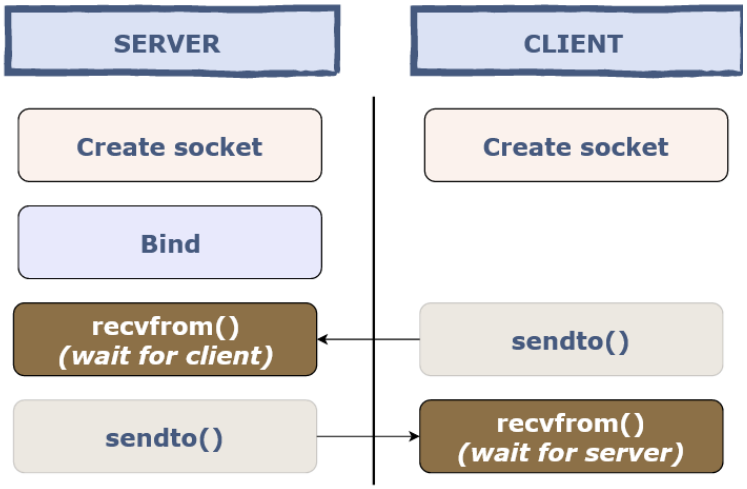

Figura 8: Protocolo UDP.

Este protocolo não garante a entrega das mensagens pelo que a não entrega irá ser verificada através de um *timeout* no servidor local. Se este *timeout* é alcançado, uma mensagem de erro é retornada pelo que deverá-se chamar a função outra vez.

A comunicação entre o servidor local e a KVS-lib é assegurada por sockets *stream*, pelo que, um erro no envio significaria que a conexão estaria comprometida. Por isso, a comunicação é feita dependendo da necessidade de cada função. Ou seja, em cada mensagem envia-se uma variável (*group,secret,key,value,...*), sendo a primeira a *flag* correspondente ao pedido.

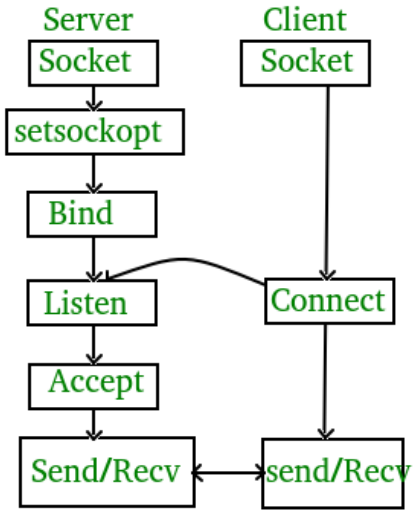

Figura 9: Protocolo para comunicação servidor local e KVS-lib.

5 Paralelismo

5.1 Gestão de *threads*

5.1.1 KVS-lib

A *thread* principal da biblioteca é responsável pela comunicação com o servidor local, ou seja, é responsável pela implementação das funções descritas no enunciado.

A única *thread* gerada pelo KVS-lib é a gerada pela *register_callback*, cuja funcionalidade é definida pelo programador da aplicação. A sua criação na *register_callback* é feita após uma alteração

na *key* monotorizada (resposta do servidor local).

5.1.2 *KVS-localServer*

O KVS-localServer tem essencialmente três tipos de *threads* diferentes. A primeira é a main que é responsável pela inicialização das variáveis globais, incluindo a socket para a conecção com o KVS-AuthServer, irá gerar a *thread* `accepting_connections` e, finalmente, irá ficar responsável pelo UI do KVS-localServer. Esta *thread* é criada na execução do programa.

A segunda, já explicada a criação na *thread main*, é a `accepting_connections`. Esta é responsável por gerar a socket de comunicação que ficará à escuta para a receção de conexões de aplicações.

Após estabelecida uma conexão, é criada uma *thread* para esta comunicação, `handle_connection`, que recebe como argumento o *file descriptor* do *socket* criado aquando do estabelecimento da conexão. Podem existir vários instâncias deste tipo de *thread*, uma para cada conexão, e esta é responsável por lidar com a comunicação com cada aplicação.

A `handle_connection` pode ser terminada quando a conexão com a App é terminada.

No encerramento do servidor, todas estas *threads* `handle_connection` são depois terminadas e juntadas com `pthread_join` no terminar da `accept_connection`, que por sua vez, irá terminar e juntar-se à *main* para o encerramento.

5.1.3 KVS-AuthServer

O servidor de autenticação apenas tem duas *threads* ao longo do seu funcionamento. A *thread* principal é responsável pela receção de mensagens e pelo seu processamento, consoante o explicado na secção da implementação. A segunda *thread* é responsável pelo UI (carregando *enter*, encerre o servidor). Esta é criada pela *thread* principal antes do ciclo da receção de mensagens.

5.2 Sincronização

O facto de termos variáveis globais no servidor local é a razão pela qual é necessário haver sincronização entre as *threads* que irão manipular estas variáveis.

Como é esperado que o número de *key-value pairs* seja menor ou não muito maior que o número de entradas da *hash-table*, então será expectável que o número de colisões na tabela deverá ser muito baixo, pelo que decidiu-se utilizar o *mutex* definido no primeiro elemento da coluna da tabela para criar a zona crítica das funções de leitura e escrita da tabela. O *mutex* engloba grande parte de cada função de escrita/leitura pois, após a chegada à tabela, esta ainda terá de percorrer os elementos com *hash key* igual, que deverão ser poucos, pelo que fez sentido esta implementação.

Outra situação em que foi necessária sincronização é no acesso à tabela dos grupos. Esta tabela deverá ser acedida apenas pela *main thread* na escrita e a única situação que requer a maior atenção é durante a eliminação de um grupo. Por isso, a sincronização é feita de forma semelhante à anterior. Além do mais, como o número de acessos a esta tabela deverá ser limitado, decidiu-se que o acesso à tabela seria restrito de modo a não ocupar espaço de memória. Isto foi feito com o *mutex access_group*.

O acesso ao status também tem de ser sincronizado com as outras *threads* responsáveis por adicionar um novo estado. Contudo, isto não pode ser feito com um *mutex* diferente do anterior pois, após todas as conexões relacionadas com um grupo a ser eliminado já estarem fechadas, não podem ser adicionadas mais conexões relacionadas a um grupo eliminado pelo que, usa-se o mesmo *mutex* nesta sincronização.

Finalmente, devido à natureza da comunicação do servidor de autenticação é necessário apenas 1 comunicação ao servidor de cada vez. Para tal, entre o primeiro envio e a última resposta foi criado um *mutex acess_auth*. Isto porque, o servidor estará à espera que respostas seguidas do mesmo servidor local sejam referentes ao mesmo pedido. Além do mais, como o servidor não garante a chegada, a resposta do servidor de autenticação tem de ser garantidamente referente ao mesmo pedido, pelo que este *mutex* inclui as respostas do servidor.

Referências

[1] João Nuno De Oliveira e Silva. *System Programming (MEEC/MEAer) - Project Assignment 2020/2021*. URL: `https://fenix.tecnico.ulisboa.pt/downloadFile/1407993358906840/projecto-v2.pdf`.