Instituto Superior Técnico

Programação de Sistemas

MEEC

Projecto - *Key-value store*

Autores:
Miguel Henrique Cardoso (78269)
Francisco Maria Girbal Eiras (79034)

25 de Maio de 2016

1 Introdução

> O armazenamento de dados é feito em serviços que poderão ser acedidos de qualquer lugar do mundo, a qualquer hora, não havendo necessidade de instalação de programas ou de armazenar dados. O acesso a programas, serviços e arquivos é remoto, através da Internet - daí a alusão à nuvem. O uso desse modelo (ambiente) é mais viável do que o uso de unidades físicas.
>
> - Computação em nuvem, Wikipedia (...)

Neste âmbito, o objectivo deste projecto consiste em desenvolver um sistema *key-value store* para armazenar informação de um tipo indefinido (`byte stream`) num servidor. Este deve ser constituido por um servidor e uma API para comunicação entre os clientes e o servidor.

O servidor deve ter algumas características específicas, nomeadamente possuir tolerância à falha (*fault tolerance*) e uma divisão explícita entre *front server* e *data server* de forma a que isto seja possível. O *front server* serve meramente como a fachada do serviço, redirecionando posteriormente os clientes para o *data server* onde poderão completar os pedidos.

2 Funcionamento do projeto

De forma a compilar o projeto na sua totalidade, executa-se o comando no terminal `make all` ou simplesmente `make`. É possível compilar apenas o *data server* ou o *front server* recorrendo aos comandos `make data-server` ou `make front-server` respetivamente. Após a compilação, é possível remover os objetos (ficheiros .o criados) através do comando `make clean`.

2.1 Inicialização dos servidores (*front* e *data*)

É necessário inicializar ambos os servidores aquando o início da utilização. A ordem pela qual são inicializados é indiferente. O mecanismo de inicialização de ambos os servidores é idêntico à inicialização após a verificação de falta (recuperação de tolerância à falta[1]).

A arquitetura do *front server* é bastante simples de representar, possui apenas duas *threads*, uma para lidar com o *data server* (tolerância à falta) e outra para reencaminhar os pedidos dos clientes para o *data server*.

[1] Ver a secção 2.3 (*Fault tolerance*)

No entanto, a arquitetura, estrutura e mecanismos do *data server* são um pouco mais complicados, encontrando-se explicados na secção seguinte.

2.2 Arquitetura do *data server*

O *data server* possui uma arquitetura do tipo *multi-threaded* de forma a receber pedidos de vários clientes simultaneamente. Para além disso, possui uma *thread* específica para comunicação com o *front server* (de forma a permitir *fault tolerance*), e ainda uma *thread* que permite guardar a estrutura automaticamente ao fim de um número fixo de segundos (de forma a que a recuperação no inicio do *data server* seja mais rápida[2]). De uma forma esquemática, é possível representar o *data server* com o modelo apresentado na Figura 1.

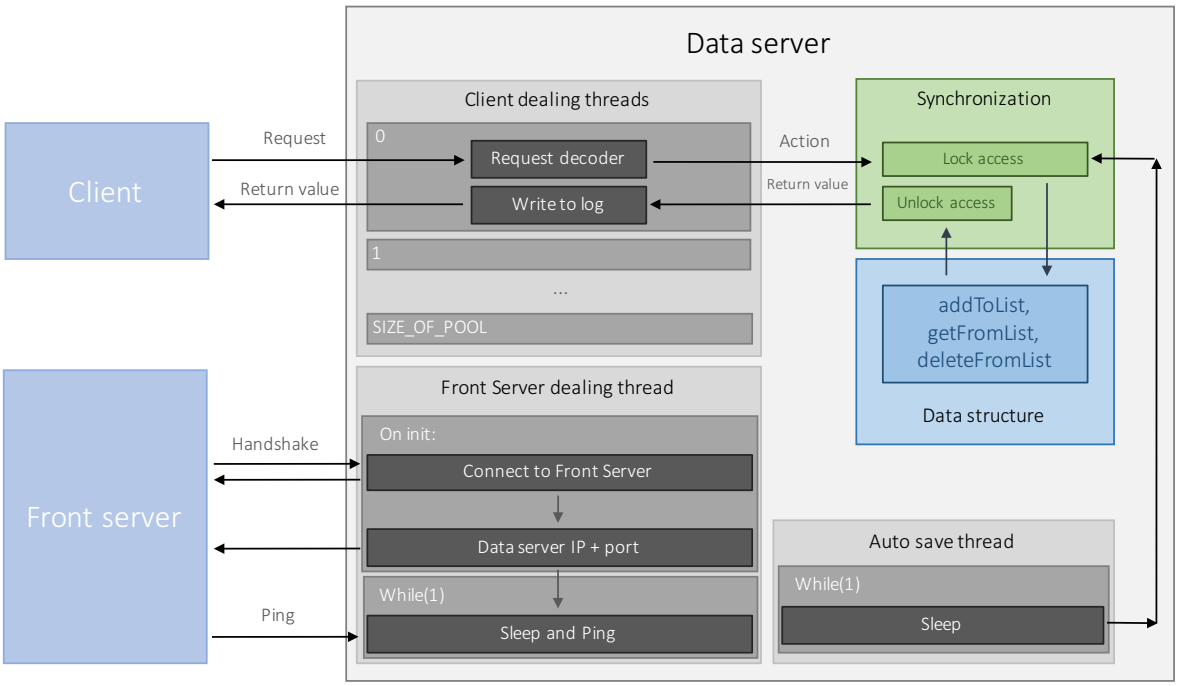

Figura 1: Representação esquemática do *data server*

Nas seguintes subsecções estão descritos os mecanismos internos de cada um dos componentes do *data server*.

2.2.1 Estrutura de dados

A escolha da estrutura de dados foi crucial no desenvolvimento do projeto. Por um lado teria que ser uma estrutura que permitisse leituras e escritas rápidas (para evitar

[2]Ver a secção 2.2.4 (*backup* de dados)

colocar clientes em espera), mas por outro teria que ter um comprimento ilimitado e desconhecido. Desta forma, optou-se por uma ***hashtable***.

Por forma a implementá-la, utilizou-se uma implementação de uma lista abstrata, em que cada elemento da lista é definido através da estrutura `node` apresentada abaixo.

```
typedef struct node
{
  uint32_t key;
  char * value;
  int value_length;
  struct node * next;
} node;
```

Apesar do elemento `value` da lista ser considerado um `char *`, na realidade este representa um `byte stream`, como claramente identificado pelo facto da estrutura também guardar o tamanho do elemento.

Nesse sentido, implementaram-se várias funções que permitem criar, adicionar, eliminar, obter e imprimir elementos da lista. Os cabeçalhos destas funções estão indicados abaixo:

```
//This function creates a new list.
node * createList();

//This function adds data to a certain key, taking into account overwrite value.
int addToList(node * first, uint32_t key, char * value, int value_length, int overwrite);

//This function searches for key, in the list, puts saved data into value.
int getFromList(node * first, uint32_t key, char ** value);

//This function finds the key in the list, and deletes it.
int deleteFromList(node * first, uint32_t key);

//This function travels through the list, freeing it from memory.
void deleteList(node * first);

//This function travels through the list, printing all information in each node, for debug purposes.
void printList(node * first);

//This function travels through the list, writing down all information in each node to the file described by fd.
void snapList(node * first, int fd);
```

Recorrendo, no programa principal do *data server*, ao seguinte código, implementou-se a *hashtable* utilizada para guardar os pares *key-value*. Após a inicialização do *data*

server, é alocada memória para os vários compartimentos da *hashtable*, de forma a que esta esteja pronta a ser utilizada na comunicação cliente - *data server*.

```
#define NUMBER_BUCKETS 199

node ** hashtable;

int hash_function(uint32_t key) {
  return key*(key + 3)%NUMBER_BUCKETS;
}
```

A *hashfunction* utilizada para selecionar a posição de cada chave na hashtable baseia-se no método *Knuth Variant on Division*[3]. De forma a maximizar os resultados deste método para números inteiros, não negativos e, aparentemente, escolhidos ao acaso, utilizaram-se 199 partições (número primo, próximo de 200).

2.2.2 Sincronização no acesso a estruturas de dados

Devido à estrutura *multi-thread* do sistema, é necessário garantir que *threads* concorrentes não acedem à estrutura de dados ao mesmo tempo, de forma a não danificar a mesma.

Para esta sincronização utilizam-se mecanismos do tipo ***mutex*** que garantem que duas operações não acedem à estrutura de dados (secção crítica) na mesma janela temporal.

A utilização deste mecanismo ao invés de *read-write locks* deve-se à eficiência temporal dos primeiros[4]. Nesse sentido, escolheu-se o primeiro mecanismo.

De forma a maximizar a eficiência do programa, implementou-se um *mutex* por cada *bucket* da *hashtable*, já que estes *buckets* são independentes entre si, pelo que um processo pode escrever, ler ou eliminar elementos de listas relacionadas com outros *bucket*s. Desta forma, obtém-se o seguinte código para os *mutex* para cada um dos *buckets*:

```
pthread_mutex_t lock[NUMBER_BUCKETS];
```

Cada acesso à estrutura de dados (só ocorre na função `thread_dealing` do *data server*) requer previamente um *lock* do *mutex* específico para o *bucket* da *hashtable*. Após a escrita, leitura ou eliminação de uma posição da lista correspondente ao *bucket* da posição correta da *hashtable*, ocorre um *unlock* de forma a que outro cliente possa aceder à informação presente nesta lista. Por exemplo, ao receber um `READ`, o *data server* executa o seguinte código:

[3]Retirado de: https://www.cs.hmc.edu/ geoff/classes/hmc.cs070.200101/homework10/hashfuncs.html
[4]Retirado de https://www.mpi-sws.org/ bbb/papers/pdf/ecrts09-long.pdf, página 7

```
pthread_mutex_lock(&lock[hashkey]);

mensagem.value_length = getFromList(hashtable[hashkey], mensagem.key, &
    newdados);

pthread_mutex_unlock(&lock[hashkey]);
```

Desta forma, garante-se sincronização e acesso sequencial à estrutura de dados por parte dos vários clientes (na realidade o *data server* é que trata do acesso à estrutura por parte dos clientes, através da API[5]).

2.2.3 Criação e gestão de *threads* para pedidos de clientes

A criação e gestão das *threads* para pedidos de clientes é realizada através de um sistema de ***pool of threads*** que degenera num segundo sistema *on demand*. Inicialmente são criadas `SIZE_OF_POOL` *threads*, tal como numa *pool*, tendo estas a possibilidade de fazer `accept` a qualquer cliente, já que o *socket* do *data-server* se apresenta como uma variável global.

Após esta criação inicial, é da responsabilidade das respetivas *threads* criar e gerir as mesmas, de forma a que estejam disponíveis, a todo o momento, `SIZE_OF_POOL` *threads* vazias e prontas a aceitar clientes. Este número só varia quando um cliente ocupa ou vaga uma das *threads*. No caso de o cliente ocupar uma *thread* vazia é necessário que esta verifique se o número de *threads* vazias é suficiente ou não. Em termos de código, isto traduz-se em:

```
if (n_empty < SIZE_OF_POOL)
{
  // Ha poucas threads activas, criar uma nova e prosseguir
  pthread_t temp_thread;
  int iret = pthread_create( &temp_thread, NULL, thread_dealing, NULL);
  n_empty++;
  ...
}
```

Após a terminação do cliente (este realiza um `kv_close` na API), a *thread* verifica se existem mais *threads* vazias do que é necessário. Nesse caso, termina ordeiramente a *thread* em questão. Isto reflete-se no código nas linhas:

[5] Ver a secção 2.4 (API)

```
if (n_empty > SIZE_OF_POOL)
{
  // Ha muitas threads vazias, desactivar esta
  n_empty--;
  break;
}
```

2.2.4 *Backup* de dados

Existem dois mecanismos distintos de *backup* de dados implementados, através de ***log*** das operações realizadas sobre a estrutura ou através de um ***snapshot*** da mesma. A diferença entre os dois mecanismos baseia-se na frequência com que estes ocorrem, e a eficiência com que a recuperação é realizada através de cada um deles.

O *log* das operações realizadas ocorre após cada operação (de escrita ou de eliminação de elementos da *hashtable*, já que leituras não afetam a estrutura), sendo uma operação rápida e eficiente em termos de escrita no ficheiro (recorrendo à função `write_to_log`), já que apenas requer a escrita da estrutura `message`[6] e do possível valor da estrutura (no caso de uma escrita).

Devido ao facto dos valores armazenados na estrutura serem do tipo `byte stream`, optou-se por armazenar os valores no ficheiro de *logs* através de uma representação binária.

A recuperação da estrutura original a partir dos *logs* é, no entanto, uma operação lenta, já que tem que passar pelo processo de decisão associado a uma `message` (semelhante a um cliente enviar dados através do *socket* - apresentado na função `log_recovery`). Desta forma, surgiu o seguinte mecanismo que coexiste no *backup* dos dados, o ***snapshot*** da estrutura.

O *snapshot* da estrutura consiste em guardar sequencialmente no ficheiro, em formato binário também, os elementos da *hashtable*. Para cada elemento é guardada toda a informação armazenada na estrutura, `key, value_length, value`. Este armazenamento no ficheiro é lento, já que requer o *lock* de todos os *mutex* previamente e a utilização da função implementada no ficheiro `list.c`, `snapList(hahstable[i], fd)` para cada um dos *buckets* da *hashtable*. Posteriormente, é necessário ainda fazer *unlock* de todos os *mutex*.

A recuperação da estrutura original a partir do *snapshot* é bastante rápida, já que requer apenas passar a `key` lida pela *hashfunction* e adicionar o elemento lido à lista diretamente.

Neste sentido, devido à rapidez dos *logs* a escrever e a rapidez do *snapshot* na recuperação, o ideal é um compromisso entre estes dois tipos de *backups* de dados, tal

[6] Ver a secção 2.5 (Comunicação cliente - servidor e vice-versa)

como utilizado no projeto.

Coexistem, assim, dois mecanismos de *backup*. Os *logs* são utilizados sempre que um cliente realiza uma operação sobre a lista (escrita ou eliminação de elementos). O *snapshot* da estrutura ocorre automaticamente ao fim de `SAVE_TIME` através de uma *thread* adicional para este efeito, e quando o servidor termina ordeiramente (através da terminação ordeira do *front server*[7]). Após este *snapshot*, o ficheiro de *logs* é limpo.

Aquando o reiniciar do servidor (por exemplo, de recuperação de *fault tolerance*), este recupera a estrutura gravada no ficheiro de *snapshot*, aplicando posteriormente as modificações apresentadas no ficheiro de *logs*, resultando na recuperação completa da estrutura prévia.

2.3 *Fault tolerance*

O mecanismo de tolerância à falta implementado é baseado numa ligação permanente que se estabelece entre o *data server* e o *front server*. Nesta ligação obtida através da utilização de um *socket* TCP, o *front server* envia um ***ping*** para o *data server* onde transfere a informação necessária para o *data server* reagir correspondentemente à sua morte. O funcionamento do mecanismo básico da *fault tolerance* está representado esquematicamente na Figura 2.

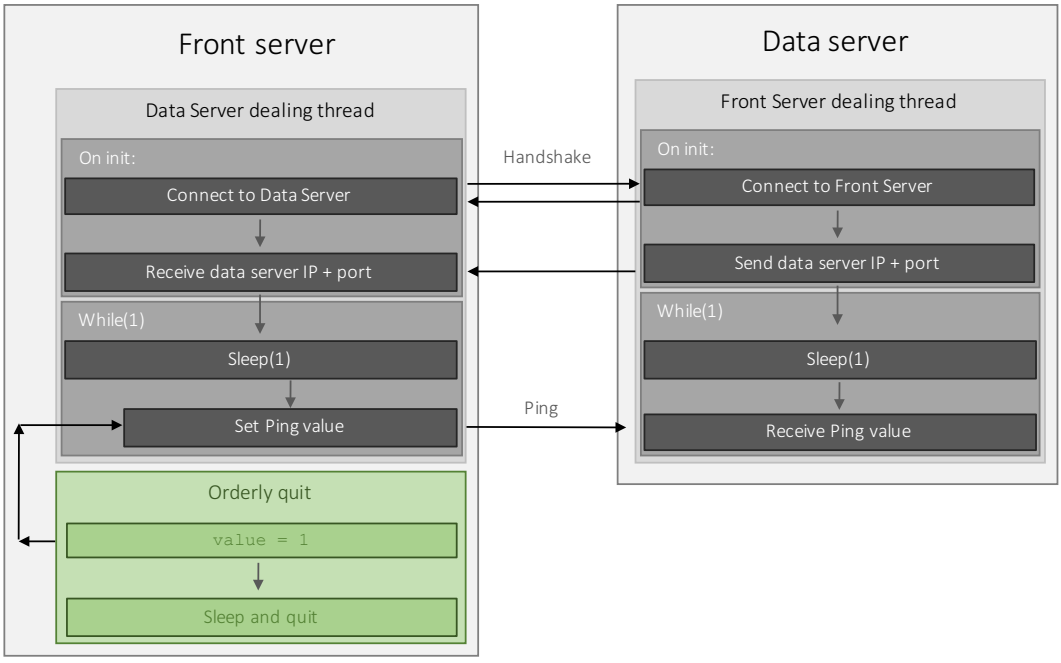

Figura 2: Representação esquemática do mecanismo de *fault tolerance* entre o *data server* e o *front server*

Inicialmente, a conexão permanente entre o *front server* e o *data server* é esta-

[7]Ver a secção 2.3 (*Fault tolerance*)

belecida através de um *handshake*, de forma a que ambas as partes reconheçam e que o *front server* receba o `ds_ip` e o `ds_port`. Isto ocorre através do código (no *data server* e correspondente no *front-server*):

```
handshake = DATA_SERVER;
send(use_sock_fd, &handshake, sizeof(handshake), 0);

nbytes = recv(use_sock_fd, &handshake, sizeof(handshake), 0);

if(handshake == ERROR)
  exit(0);

send(use_sock_fd, ds_ip, 50, 0);
send(use_sock_fd, &ds_port, sizeof(ds_port), 0);
```

A partir daqui, estas *threads* ficam em *loop* a enviar e receber (respetivamente) um sinal de *ping*. Desta forma, cada uma sabe da existência da outra. Caso o *front server* seja terminado ordeiramente, este envia um *ping* com o valor de `1`, correspondendo a um sinal de terminação ordeira do *data server* também. Caso contrário, activa-se a *fault tolerance*, através da função `execve` e código adicional de *setup*.

Após a primeira terminação inesperada de algum dos servidores, o *data server* assume sempre o papel de pai (na cadeira do *bash*), de forma a garantir que existe uma relação bem definida entre ambos. Desta forma é possível que, caso qualquer um deles termine inesperadamente, este seja reinicializado pelo outro. Isto ocorre através da função de inicialização do *front server* na *thread* `front_server_communication`, através da função `relaunch_front_server`.

Caso seja o *data server* a sair inesperadamente, o *front server* recorre ao inicio do *data server* em modo de recuperação (passando-lhe um segundo argumento igual a 1), como demonstrado no código:

```
char * buffer[] = {"./data-server", "1", 0};
execve(buffer[0], buffer, NULL);
```

Desta forma, o *data server* reinicia e inicializa uma nova instância do *front server* à qual se liga imediatamente através da função referida anteriormente.

É de notar que, de acordo com o código escrito e com o enunciado, o *data server* se encontra sempre entre os portos `9988` e `9998`, o *front server* entre os portos `9999` e `10009` e a conexão inicial entre o *front server* e o *data server* para garantir a *fault tolerance* entre os portos `9990` e `9980`.

2.3.1 Tratamento de erros

Qualquer erro de comunicação entre o *data server* e o *front server* leva à morte imediata do servidor que o detectou, através da função `exit` e um código de erro específico. Caso ambos os servidores o tenham detetado, não é possível recuperar, já que qualquer recuperação de ambos resultaria no mesmo erro.

No entanto, no caso de apenas um servidor encontrar um erro na comunicação, o outro automaticamente o reinicia. Neste caso, a *fault tolerance* resulta na proteção contra os erros. Uma excepção a este caso, é se os servidores não estiverem na fase de comunicação, já que nenhum sabe da existência do outro, pelo que não é possível verificar a existência de erros.

2.4 API

O outro objectivo do projecto, para além da criação de uma *key-value store*, era a implementação de uma API que permitisse a qualquer cliente interagir com os servidores, sem se preocupar com a implementação dessas funções e garantindo a segurança e funcionamento da comunicação

Para esse efeito foram escritos dois ficheiros, `psiskv.h` e `psiskv_lib.c`. No primeiro ficheiro encontram-se os protótipos das funções, no segundo a sua implementação, escondida do cliente.

2.4.1 Funções de comunicação

Tal como pedido no enunciado do projeto, foram implementadas as funções essenciais, representadas pelos seus protótipos apresentados a seguir. A função `kv_connect` recebe o ip e o porto do *front server*, sendo posteriormente redirecionada para o *data server* pelo primeiro, estabelecendo e devolvendo o *file descriptor* do *data server*, onde todos os restantes pedidos são completados (`kv_write`, `kv_read`, `kv_delete` e `kv_close`). Esta dinâmica está explicada em mais detalhe na secção 2.5 (Comunicação cliente - servidor e vice-versa).

```c
#define ERROR -1
#define CLIENT 1
#define OK 2
#define MIN(a,b) (((a) < (b))? (a):(b))

//This function connect with a front server located in kv_server_ip with port kv_server_port.
int kv_connect(char * kv_server_ip, int kv_server_port);

//This function closes the connection between the client and the data server.
void kv_close(int kv_descriptor);

//This function writes in key, the data stored in value, depending on overwrite.
int kv_write(int kv_descriptor, uint32_t key, char* value, int value_length, int overwrite);

//This function writes in key, the data stored in value, depending on overwrite.
int kv_read(int kv_descriptor, uint32_t key, char* value, int value_length);

//This function writes in key, the data stored in value, depending on overwrite.
int kv_read_optimized(int kv_descriptor, uint32_t key, char** value);

//This function deletes the data stored in key.
int kv_delete(int kv_descriptor, uint32_t key);
```

Para além das funções básicas requeridas pelo enunciado, desenvolveu-se ainda uma função extra, o `kv_read_optimized` que permite ao utilizador da API receber diretamente a totalidade dos dados encontrados na posição do dada por `key` sem ter que proceder a uma nova leitura após a primeira. Isto deve-se à alocação interna do parâmetro `value` (pelo que não é preciso saber *a priori* o tamanho do campo a ler, nem é preciso voltar a ler caso seja superior ao tamanho alocado).

2.4.2 Protocolo de comunicação

As mensagens trocadas entre a API e o servidor seguem um tipo de dados especificamente desenvolvido para esta comunicação, o `message`, como definido no código abaixo.

```c
#define READ 0
#define WRITE 1
#define OVERWRITE 2
#define DELETE 3

typedef struct message {
  int operation;
  uint32_t key;
  int value_length;
} message;
```

Em que o parâmetro `operation` corresponde sempre a uma das operações definidas acima.

A API estabelece a comunicação inicial, enviando uma mensagem para o servidor, que posteriormente é descodificada (na função `thread_dealing` do *data server*). Dependendo do tipo de mensagem e do objetivo final, pode ser ou não devolvido um valor intermédio (neste caso apenas a operação `READ` devolve o valor do elemento em `key`) e um valor final (no caso do `READ`, `WRITE` ou `OVERWRITE`) de forma a indicar se a operação foi bem sucedida ou o **código de erro** em caso deste.

2.5 Comunicação cliente - servidor e vice-versa

A comunicação entre o cliente e o servidor (na realidade *front server* e *data server* ocorre sempre através da biblioteca fornecida nos ficheiros `psiskv_lib.c` e `psiskv.h`. Desta forma, e tendo em conta os resultados apresentados nas secções anteriores, obtém-se, na Figura 3, o esquema total representativo do sistema.

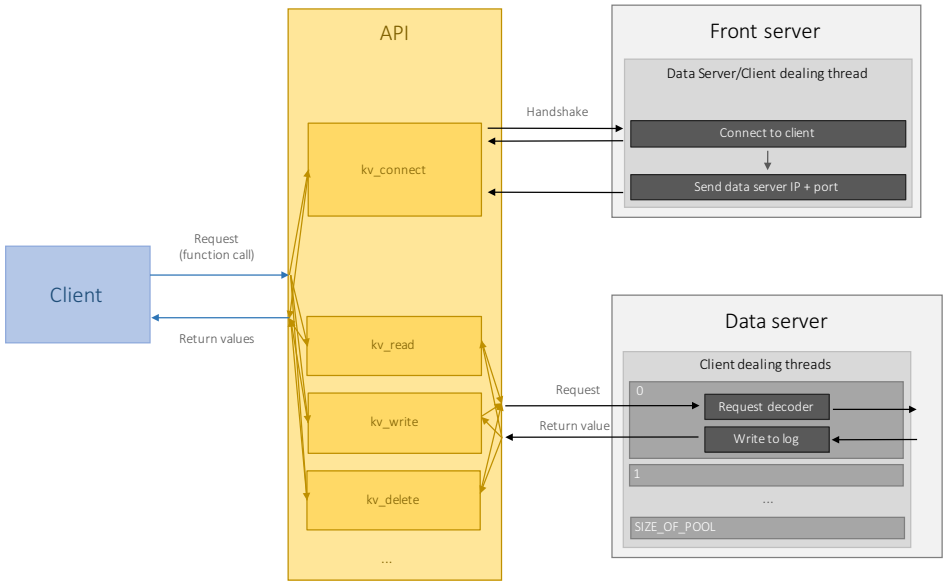

Figura 3: Representação esquemática do sistema total e comunicação com clientes

3 Conclusão

No geral, o projeto foi bem sucedido. O sistema completo não demonstra falhas em termos técnicos, apesar de algumas ineficiências.

Em primeiro lugar a estrutura de dados. Muitas linguagens de programação (como por exemplo *Python* e *Java*) suportam dicionários, que correspondem a *key-value store* optimizado. Desta forma, provavelmente seria possível melhorar o desempenho total do sistema adoptando uma destas estruturas.

Em segundo lugar, o mecanismo de gestão de *threads* também não é o mais eficiente possível, já que este coloca algum *overhead* nos clientes aquando da verificação e criação de *threads* quando não existem suficientes. Um sistema mais eficiente seria um controlo externo das *threads* (através, por exemplo, de uma *thread* auxiliar) que verificaria e guardava as *threads* ativas, sendo possível desta forma criar novas *threads* antes dos clientes chegarem, caso não existissem suficientes.

Por último, o mecanismo de sincronização ao acesso a estruturas de dados também não é o mais eficiente, já que se tem que fechar um *bucket* da *hashtable* para modificar apenas uma posição. Uma solução mais eficiente poderia passar por vários *mutex* associados a cada posição da lista, que poderiam ser acedidos através de uma função especificamente de lista para executar uma outra função sobre esse elemento específico da lista. Assim, seria possível percorrer a lista e elementos no mesmo *bucket* não seriam bloqueados por alteração de apenas um.

Apesar destas ineficiências, concluímos que o projeto foi bem sucedido, tendo todos os objetivos estabelecidos sido cumpridos. Este projeto foi útil na motivação para a disciplina e para o que nos reserva o futuro na área da Arquitetura de Sistemas.