System Programming - MEEC IST
2015/2016
Project

1 Objectives

The objective of this project is to develop a simple key-value store system.
The system will be composes of a programming API, a library executed in the client applications and a single server.
The server will be multithreaded and will offer simple fault tolerance

2 Key-value store

A key value store is a class of databases with a simple access API (similar to a hash table)

From Wikipedia:

> A key-value store, or key-value database, is a data storage paradigm designed for storing, retrieving, and managing associative arrays, a data structure more commonly known today as a dictionary or hash. Dictionaries contain a collection of objects, or records, which in turn have many different fields within them, each containing data. These records are stored and retrieved using a key that uniquely identifies the record, and is used to quickly find the data within the database.

2.1 API

The system should provide a programming API that allows the development of programs that use the key-value store to store values.

The required functions to implement are:

int kv_connect(char * kv_server_ip, int kv_server_port);

- This function establishes connection with a Key-value store. The pair (**kv_server_ip** and **kv_server_port)** corresponds to the address of the Controller
- This function return **-1** in case of error and a positive integer representing the contacted key-value store in case of success. The returned integer (key-value store descriptor) should be used in the following calls.

void kv_close(int kv_descriptor)

- This function receives a previously opened key-value store (**kv_descriptor**)and closes its connection.

int kv_write(int kv_descriptor, uint32_t key, char * value, uint32_t value_length, int kv_overwrite)

- This function contacts the key-value store represented by **kv_descriptor** and stores the pair (**key**, **value**). The **value** is an array of bytes with length of value_**length**.

- If **kv_overwrite** is 1 and the key already exist in the server the old value will be overwrite, and the function will return 0. If **kv_overwrite** is 0 and the key already exist in the server the function will fail and return -2. The system returns 0 in case of success and -1 in case of any other error.

int kv_read(int kv_descriptor, uint32_t key, char * value, uint32_t value_length)

- This function contacts the key-value store and retrieves the value corresponding to key. The retrieved value has maximum length of **value_length**. And is stored in the array pointed by **value**.
- If the values does not exist the function will return -2. If the key exist in the server this function will return the length of the value store (if the returned value is larger that length not all data was given to the program). In case of any other error the function will return -1.

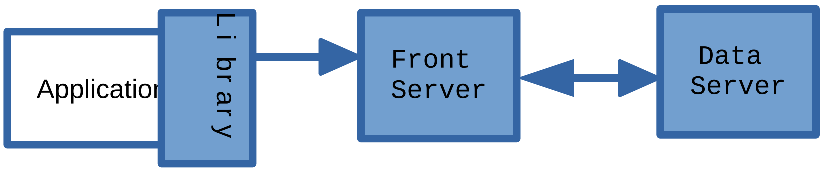
int kv_delete(int kv_descriptor, uint32_t key)

- This function contacts the key-value store to delete the value corresponding to key. If the pair (key. Value was successful deleted this function return 0, otherwise return -1. From this moment on any kv_read to the supplied key will return error.

These functions should be implemented as a library (psiskv_lib.c and psiskv.h files) that other programmers can add to their own code.

3 Components and communication

The overall architecture of the system is presented in the next figure:



Only the components in blue are developed in the context of this project. A testing application will be provided.

The library is composed of a set of data-types and functions that external programmers can use to interact with the database.

The interface (API) of this library is described in this document. The functions of this library will be implemented by the students and will allow the access of data stored the server.

The students should develop the library (psiskv_lib.c and psiskv.h files) and two servers.

The **Front Server** will listen to the port 9999 or first available port and will handle the **kv_connects** from the clients. All **kv_reads** and **kv_writes** will be received and handled by the **Data Server**.

Students should define the structure of the messages exchanged between the client and the servers:

- From client application to the **Front Server**

    - kv_connect
- From client application to the **Data Server**
    - **kv_write** – key, value, length of the value
    - **kv_read** – key-value
    - **kv_delete** – key-value
- From the **Data Server** to the client application
    - **kv_write** – notification of success
    - **kv_read** – value, length of value
    - **kv_read** – notification of success
- From the **Front Server** to the **Data Server**
    - initialization
    - termination
    - fault tolerance

The pairs (key, value) should be stored in a suitable data-structure in the **Data Server**.

The communication should be done using SOCK_STREAM AF_INET sockets.

3.1 Client connection

The **kv_connect** in the client should receive the address (ip+port) of a **Front Server**. After the connect/accept instructions the established connection should be redirected to the **Data Server**. From this point on, all the **kv_*instructions*** should be received and handled by the **Data Server**.

4 Multithreading

All servers should be multithreaded, to handle multiple clients concurrently.

4.1 Thread Creation policies

There are two main policies for the creation of threads:

- on demand
- pool of threads

4.1.1 On demand thread creation

With on demand thread creation all threads are created after the **accept** on the main. In the beginning of the server only one thread is executing. The main loop of this thread has the accept function and after it a thread is created. This thread receives as parameter the newly created socket.

4.1.2 Pool of threads

With this policy a set of threads is created before any connection is accepted. Each thread will be responsible for doing the accept and handling the following messages. After the close of the socket each thread should block again in the accept.

4.2 Critical regions

Since the data-structure that stores the (key, value) pairs is shared between various threads that concurrently change it, it is fundamental to guarantee that the concurrent editing (insertion and remove) does not affect data consistent.

Students should identify the various critical regions that occur when accessing the shared data (shared variables and list of (key, value) pairs).

4.3 Synchronization

Students should start implementing the necessary synchronization to guarantee that the identified critical region are guarded and that no two thread execute those critical regions simultaneously.

5 Server operation

Only the **Front Server** will interact with the operator by means of the keyboard and console.

5.1 Data backup

At all time, there should exist a persistent copy of the data stored in a file (the **Backup file**). This backup should be used when initialing the server or when restarting it from fault tolerance.

5.2 Server initiation

After the proper servers initialization there should be two processes running (**Front Server** and **Data Server**).
These servers can be created using the **fork** system call or created separately. In either cases the **Front Server** should be informed of the socket port of the **Data Server** where clients will communicate to (kv_read, kv_write and kv_delete).
If the **backup file** is not empty the **Data Server** should initialize the internal data-structure with that data in the file.

5.3 Server termination

On the **Front Server** the operator can issue a **quit** command or press Ctr-C. In these cases both the **Front** and **Data Servers** will be orderly terminated.

6 Fault Tolerance

If any of the servers is killed or abruptly terminated it should be relaunched automatically.
The **Data Server** will check for the aliveness of the **Font server** and vice versa.
During the recovery period it is not necessary to guarantee fault tolerance.

When the **Data Server** is recovered, the data should be re-loaded from the **Backup** File and all the client connections should be restored.

7 Report

Along with the project code, students should provide a simple writen report presenting the decisions and description of algorithms related to the following issues:

- Communication flow between clients and server
- Format of exchanged messages
- Data-structure to store the key. Value pairs
- Thread creation
- Synchronization on the access to shared data structures
- Data backup
- Servers initialization
- Fault tolerance