System Programming
8th Laboratory (22th and 27th April 2016)

1 Lab 8

In this laboratory the students will continue the development of the last laboratory
The student should continue the implementation of the multithreaded server, allowing the concurrent handling of multiple clients.
Implementation of synchronization to the shared data-structure
The server should:

- handle all the read/writes between each accept and close of the socket in a single thread.
- share a data-structure: list of (key, value) pairs) between the various threads
- Guarantee of consistent access to the shared list.

1.1 Critical regions

Since the data-structure that store the (key, value) pairs is shared between various threads and concurrently this data styructure can be changed it is fundamental to guarantee that concurrent editing (insertion and remove) of the data is consistent.

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

1.2 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.

NOTE: please verify minor changes on the following pages (marked in yellow).

2 Key-value store

The objective of this work 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.

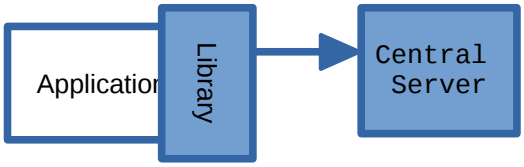
2.1 Definition

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.

2.2 Architecture

The overall architecture of the system is presented in the next figure (left centralized version, right distributed version):



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 implemeneted by the students and will allow the access of data stored the server.

2.3 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 **vaue_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 delet the value corresponding to key. 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.

2.4 Key-value implementation

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

The server will listen to the port 9999 or first available port and will handle a single client a a time:

> The server receives an accept (from the **kv_connect**) will read all the requests from the client (**kv_write** and **kv_read**). Only after the **kv_close** on the client the server should do a new accept.

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

- From client to the server
  - **kv_write** - key, value, length of the value
  - **kv_read** - key-value
- From the server to the client
  - **kv_write** - notification of success
  - **kv_read** - value, length of value

The pairs (key, value) should be stored in a suitable data-structure. The simplest being a linked list.

The communication should be done using SOCK_STREAM AF_INET sockets.

2.5 Thread creation

There are two main policies for the creation of threads:

- on demand
- pool of threads

2.5.1 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.

2.5.2 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.

3 Project

The code of the Library (on the clients), the data-structure (on the server), and the message structures will remain the same on the final project.