

Internet Programming

Programming Assignment 2: Distributed Programming with Sockets

Deadline: Tue 2 October 2018, 23:59

1 A Key-Value Store

In this part of the assignment, you will have to implement a simple TCP-based key-value store. This will consist of a server implementing the key-value store, and a client making use of it. The server must accept clients' connections and serve their *put* and *get* requests for *key-value pairs*. All key-value pairs should be stored by the server only in memory. Keys and values are alphanumeric (i.e., strings).

Protocol definition The communication between the client and the server should strictly implement the following protocol. Upon connection the client sends one or more requests to the server. Each request starts with a byte specifying the type of operation, where the value 103 means *get*, and the value 112 means *put*. Any other number should cause an error and the immediate disconnection of the server. Following the byte specifying the operation, the client should send a key (in case of *get*), or a key and a value (in case of *put*). Both keys and values are to be sent as null-terminated strings. This completes the submission of a request, and nothing more should be sent by the client, except if it wants to submit another request in the same connection, in which case after the `'\0'` byte terminating the first (in case of *get*) or second (in case of *put*) string, it should carry on directly with the following request's operation type and string(s). Any number of requests may be sent in a single connection, without limit.

Upon reception of a request, the server behaves as follows. In case of a *get* request, the server should look its memory store up, and send a reply to the client. The first byte of the reply tells whether the requested key was found (number 102) or not found (number 110). If the key was found, its corresponding null-terminated value should be sent directly after the number 102. If it was not found, sending the number 110 suffices, i.e., you should not send anything additional after that byte.

Both the server and the client should close the connection instantly if they notice that the other party is not abiding by the aforementioned protocol. Other than that, the server never decides to close the connection. The client is free to close the connection any time. Typically, the client will close the connection after it has sent a number of requests and has received the corresponding responses, but in case it abruptly terminates at any arbitrary moment it should not cause any problem to the server.

Client implementation The client should be implemented in `client.c`, and must accept a variable number of command-line arguments. The first argument is the server hostname and the second is its port. After that it should include one or more requests, where a request can be either `get <key>` or `put <key> <value>`. For *get* requests, the client should print the returned value followed by a new line, or just a new line if the key was not in the store. For *put* requests it should not output anything at all.

A typical series of client invocations should look like this:

```
$ ./client pcoms001.vu.nl 5555 put city Amsterdam
$ ./client pcoms001.vu.nl 5555 get city
Amsterdam
$ ./client pcoms001.vu.nl 5555 put country NL get country
NL
$ ./client pcoms001.vu.nl 5555 get city put postcode 1081HV get country
Amsterdam
NL
$ ./client pcoms001.vu.nl 5555 get city get inexistentkey get postcode
Amsterdam

1081HV
$
```

Server implementation The server should be listening on all interfaces of its machine, at a TCP port provided by the user as the one-and-only command-line argument. You should implement four versions of the server, called **serv1**, **serv2**, **serv3**, and **serv4**. Each version must be implemented using a different structure, as follows:

1. **Iterative server:** **serv1** must be implemented using an iterative structure. I.e., it processes client connections one at a time. The listen backlog should be set to the value 5 (which is also the default value).
2. **On-demand forking server:** **serv2** must support concurrent processing of multiple client connections. It must use the **one-process-per-client structure**.
3. **Preforking server:** **serv3** must be implemented using a **pre-forking structure**. The number of worker processes should be defined as a second command-line argument.
4. **Multi-threaded server:** Finally, **serv4** must be implemented using multiple threads. You are free to create threads on demand, or prepare a pool of them in advance, whatever you find most convenient.

With the exception of **serv3** taking two arguments, all other server programs take one argument, the port number. The servers could be invoked as follows (of course, each maintaining its own separate key-value store):

```
$ ./serv1 5555
$ ./serv2 5556
$ ./serv3 5557 10
$ ./serv4 5558
$
```

Inside the server you should implement the key-value store by using and implementing the interface defined in **keyvalue.h** (Figure 1).

Synchronization Your servers (except for the iterative one) should allow multiple clients to concurrently get any values, as reading values does not incur race conditions. However, for putting values you should take explicit steps to guarantee correctness, but you should do it in the most lightweight way possible. E.g., updating the value of the same key should be protected by mutual exclusion, while updating different keys could be done in parallel. But even when updating different keys you should make sure that your internal data structures (e.g., of your hashmap) won't get corrupted. Of course you should also have mutual exclusion between a process that puts a key and any processes getting that same key.

```

#ifndef __KEYVALUE_H
#define __KEYVALUE_H

/*
 * Returns the value associated with the key, or NULL
 * if the key is not in the store.
 */
char *get(char *key);

/*
 * Puts the <key,value> pair into the store.
 * If the key already existed, its previous value
 * is discarded and replaced by the new one.
 */
void put(char *key, char *value);

#endif

```



Figure 1: `keyvalue.h`

2 A Talk Program

In the second part of this assignment, you will have to implement a talk program that allows two users to chat over the network. Everything the first user types is sent to the second user to be displayed on his/her screen, and vice-versa.

For establishing a connection, one of the programs must be launched in *server mode* and the other in *client mode*. Both modes must be implemented in a single program called `talk.c`. After the initial connection establishment, the two programs behave identically. During the chat session data must be sent as they become available, **without waiting for the next carriage return**. Remember that when using `write()/send()` on a network file descriptor you are not guaranteed that the whole specified buffer will be transferred. Consult the lecture notes for handling this problem.

If one command-line argument is supplied it should be interpreted as a port number, and `talk` must start in server mode listening for incoming TCP connections at that port. If two command-line arguments are supplied, the program must start in client mode and interpret the arguments as the server hostname and port it should connect to.

The end of the session is signaled by a  +  keystroke. After the end of the session, both programs must exit.