Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Systems@ETH Zürich

Autumn/Fall Term 2022

Computer Systems Bonus Assignment

Assigned on: **8th November, 2022**
Due by: **23rd December, 2022**

1 Introduction

In this graded bonus exercise, you will implement an in-memory key-value store: a server which stores and retrieves data items referred to by keys. Your server will listen on a TCP network socket, and respond to client requests to 'get' and 'set' values coming from programs across the network. These requests, and the responses your server must generate, are specified by a simple protocol we supply below.

Your task is to implement the server, but we do not require a specific model of implementation (thread based, event based, forking, ...). However, it has to be implemented in C, run on Linux, and must not require any external libraries apart from `libpthread` (for thread support). In fact, to simplify things further, your project should be handed in as a single C source file, which we will compile as detailed below.

We provide a (basic) test client that will interact with your server, and should be enough to tell you if you have implemented the basic protocol correctly. However, the full test we will run against your code will include plenty of other cases (including how to handle buggy, malicious, or overly-enthusiastic client programs).

1.1 Grading

This bonus exercise will grant you 0.25 grade points if it passes our automated test suite. The test suite consists of up to 1000 clients. Clients following the protocol should be handled correctly, misbehaving clients should be correctly terminated. A misbehaving client should not negatively impact any conforming client and of course not crash the server.

This is an individual task and you must write your own code, but it is fine (indeed, encouraged) to discuss design issues with your colleagues and use online resources. You might also consider sharing test code, but the code implementing the server and you hand in should be completely written by yourself.

2 Server requirements

The server should keep an internal database, correlating keys with their data values. Clients can set a key, by issuing the **SET** command. If the key does not exist it should be created, if it already

exists the data value should be replaced. All clients see the same database, a key set by one should be visible to all other connected clients immediately.

The server should listen on TCP port 5555 and keep accepting new connections.

Requests should be fully atomic, if a **GET** and a **SET** requests are launched at the same time (from different clients) the **GET** has to return either the data value before or after the **SET**.

No client should be able to block another client. For example also a client that reads very slowly should not block a faster client.

A misbehaving client should not be able to crash the server or cause a memory leak. This includes sending invalid requests or terminating the connection before finishing a request. If the server detects a misbehaving client, it should terminate the connection to that client, but continue functioning for other clients.

3 Protocol

The server has to implement two commands, **GET** and **SET**, and should process an infinite sequence of these commands.

When we say strings, we mean arbitrary length binary safe strings. They may contain null characters and might be up to 32Mb in length. To enable binary safe encoding, we prefix the length of the string in bytes. For example `$6$hallo\n` represents a 6-byte string.

All commands and responses are terminated with a newline character.

GET is followed by a key (encoded as string) the server should reply with a **VALUE** response followed by the keys value. If the key does not exist, the server should send the **ERR** response.

SET is followed by a key, followed by a value (both encoded as string). The server should reply with a **OK** response. If the key can't be stored (out of memory or similar error conditions) the server should send the **ERR** response.

The precise syntax in BNF is:

⟨*command*⟩ ::= '`GET`' ⟨*string*⟩ '`\n`'
  | '`SET`' ⟨*string*⟩ ⟨*string*⟩ '`\n`'

⟨*string*⟩ ::= '`$`' ⟨*int*⟩ '`$`' ⟨*char*⟩*

⟨*int*⟩ ::= [1-9] ([0-9])*

⟨*char*⟩ ::= any byte, 0-255

⟨*response*⟩ ::= '`OK\n`'
  | '`ERR\n`'
  | '`VALUE`' ⟨*string*⟩ '`\n`'

Note that there is **no** whitespace between keywords and/or string arguments.

3.1 Example client

We provide a simple example client. The client can be used as follows `./runtest.sh`. The script uses `netcat`, which we also recommend for interactive debugging. `netcat` is commonly found as `nc`.

3.2 Test environment

We will use the following command on a Ubuntu 20.04 LTS to test your solution:

`gcc -Wall -pthread server.c -o server && ./server`.

Then clients will start connecting from localhost, but ideally you bind to any interface.

3.3 Hints and tips

An excellent explanation of socket programming in C as well as the different possible server models can be found in the textbook "Computer Systems: A Programmer's Perspective, 3rd edition", in particular the chapters 11.4 - "The Sockets interface" and 12.1 (until 12.4) "Concurrent programming".

A good reference for sockets is the GNU libc software manual[1], it also contains an example of a `select` based server[2]. A good tutorial of a `thread` based server can be found here[3].

This is an excellent exercise to try out the latest Linux kernel API for polling file descriptors or implementing the shiniest and fastest hashing technique, but we encourage to do so only after you have a working solid and simple solution. Performance is not a factor in grading, but correctness is.

In our modest opinion: The easiest implementation is one that uses a one thread per client model and uses coarse grained locking to access the shared datastructure.

3.4 Handin

The single solution file has to be submitted in the courses Moodle page. See the section about the bonus task.

[1] `https://www.gnu.org/software/libc/manual/html_node/Sockets.html`
[2] `https://www.gnu.org/software/libc/manual/html_node/Server-Example.html`
[3] `http://www.mario-konrad.ch/blog/programming/multithread/tutorial-04.html`