The System

For this assignment, your task is to implement a small compression service. This service will consume data in ASCII format over a TCP socket and return a compressed version of that data.

Messaging Format

All messages that flow over the socket share a common header that consists of three fixed width integer fields in the following order:

- A 32 bit wide magic value which is always equal to `0x53545259`.
- A 16 bit payload length
- A 16 bit request code / status code

The header may or may not be followed by a payload depending on the message type.

Lastly, **all** fields are in network byte order.

Requests

The compression service should support the following request types (request code noted in parenthesis):

- "Ping" (RC: 1)
  - Serves as a means to check that the service is operating normally.
- "Get Stats" (RC: 2)
  - Retrieves various internal statistics of the service. Note that these statistics **do not** have to be preserved across service instances.
- "Reset Stats" (RC: 3)
  - Resets the internal statistics to their appropriate default values.
- "Compress" (RC: 4)
  - Requests that some data be compressed using a particular compression scheme.

All other request codes should be considered invalid.

Request Formats

Ping / Get Stats / Reset Stats Requests

All three of these requests consist of only a header with the payload length set to zero and the request code set appropriately (e.g. 3 in the case of a "Reset Stats" request).

Compress Request

The "Compress" request will consist of a header followed by the ASCII payload to be compressed. For simplicity, you may reject all strings that do not consist of solely lowercase alphabetic characters (e.g. a-z). Note that your server should have an implementation defined maximum payload size (of at least 4KiB but less than 32KiB). Any request that is larger than this should result in an appropriate error.

Compression Algorithm

The compression algorithm that you should implement is a simplified prefix encoding compression scheme. Essentially you should replace all consecutively repeated characters in the given string by a prefix denoting the number of characters replaced followed by the character itself. Some examples:

```
a => a
aa => aa
aaa => 3a
aaaaabbb => 5a3b
aaaaabbbbbbaaabb => 5a6b3abb
abcdefg => abcdefg
aaaccddddhhhhi => 3acc4d4hi
123 => <invalid: contains numbers>
abCD => <invalid: contains uppercase characters>
```

Responses

Each of the above requests has a matching response as defined below. In all cases the status field of the header should be filled in appropriately from the following table:

Status Code	Meaning
0	OK
1	Unknown Error
2	Message Too Large
3	Unsupported Request Type
4 - 32	Reserved Status Code Range
33 - 128	Implementer Defined

Note the "Implementer Defined" bucket. You may add your own custom error codes where necessary. Please be sure to document these custom codes in the README.

Ping Response

A ping response consists of just a header with the payload length set to zero and the status code set to OK (0) if the service is operating normally or one of the error status codes if some error condition has occurred **that is not request specific** (e.g. service initialization failed, low memory condition, etc).

Get Stats Response

The "Get Stats" response should consist of two 32 bit unsigned integers followed by one unsigned byte:

- Total Bytes Received (A count of all bytes received by the service, including headers).
- Total Bytes Sent (A count of all bytes sent by the service, including headers).
- Compression Ratio (A number from 0 - 100 representing the performance of the service. e.g. if the service was asked to compress 298398921 bytes of data and was able to compress those bytes down to 129372810 bytes, the service's compression ratio would be 43).

Reset Stats Response

A "Reset Stats" response should consist of just a header with a payload length of zero and an appropriate status code.

Compress Response

A "Compress" response will consist of a header with payload size set appropriately followed by compressed ASCII data. If an error occurs, the response should be just a header with payload size set to zero and an appropriately set status code.

Submission Requirements

Your solution to this problem should be kept private and only shared with a Starry employee.

The submission itself should consist of a tar archive compressed with LZMA (`.tar.xz`) that contains, at minimum, the following files:

- A `README.md` containing:
  - A description of your target platform (e.g. Windows 10, Fedora 26, Ubuntu 17.04) and language, if applicable (e.g. C++14 compiled under GCC 7.2).
  - A brief description of the inner workings of your code
  - A list of third party libraries or other projects which you used along with a very short description of why you used that particular library

    - A description of any assumptions that you made while implementing your solution
    - Optionally, a description of any improvements that you would make to your code if given more time or resources
- A shell script named `build.sh` that does all the necessary work to setup and/or build your project
- A shell script named `run.sh` which starts your executable listening on port 4000

You are free to use any programming language that you prefer, though, we recommend something like C, C++, Rust, Go, or Python.

You may use any third party libraries that you'd like as long as you meet one or more of the following requirements:

- You package the dependency and provide it as part of your submission. An example of this would be packaging the source code for a C library that you've integrated into your `build.sh` build flow.
- The package is available for installation via package manager in the standard set of repositories for your target platform. An example of this would be the Boost C++ Libraries, which are available for installation via `dnf` on Fedora.
- The package is available via your target language's build system. For instance, the `log` crate from `crates.io`, which can be downloaded via `cargo`.

How We'll Evaluate Your Submission

We have a testing framework which will build and run your system using the scripts mentioned above, then make a series of automated requests to your system to test its functionality. *However*, of highest importance to us is clean, well-organized code, not 100% test result perfection.

During your on-site interview we'll discuss your submission, and focus especially on your thoughts on the following questions:

1. Do you notice anything deficient about the API as defined above?
2. Are there any changes you would make to the API to improve it?
3. If this server is running on a system with constrained memory, what measures would you have to take to ensure it does not crash due to a lack of available memory?
4. How could you improve the performance of your implementation?