Design Document for

Client Server Ruby Application

Designed by: Saifullah Mahmud Sumon | Last Updated: 20-Feb-2017

Overview

This is a simple client server application that:

- Uses TCP protocol to communicate between client and server.
- The client gathers, formats and sends data.
- The server receives data and logs them in a log file.

The server component listens to a specific port and the client must use the same port to establish communication.

High Level Requirement

There are two standalone applications that need to be designed:

The Client

The client needs to get 1024 bytes of random data from /dev/urandom of the host unix/linux environment. Then the data needs to be converted to UTF-8 encoding while discarding all non compliant characters. Then the UTF-8 text needs to be formatted with asterix (*) for space and newlines. This formatted string needs to be sent over TCP protocol to the server application. The client can terminate once the data is sent.

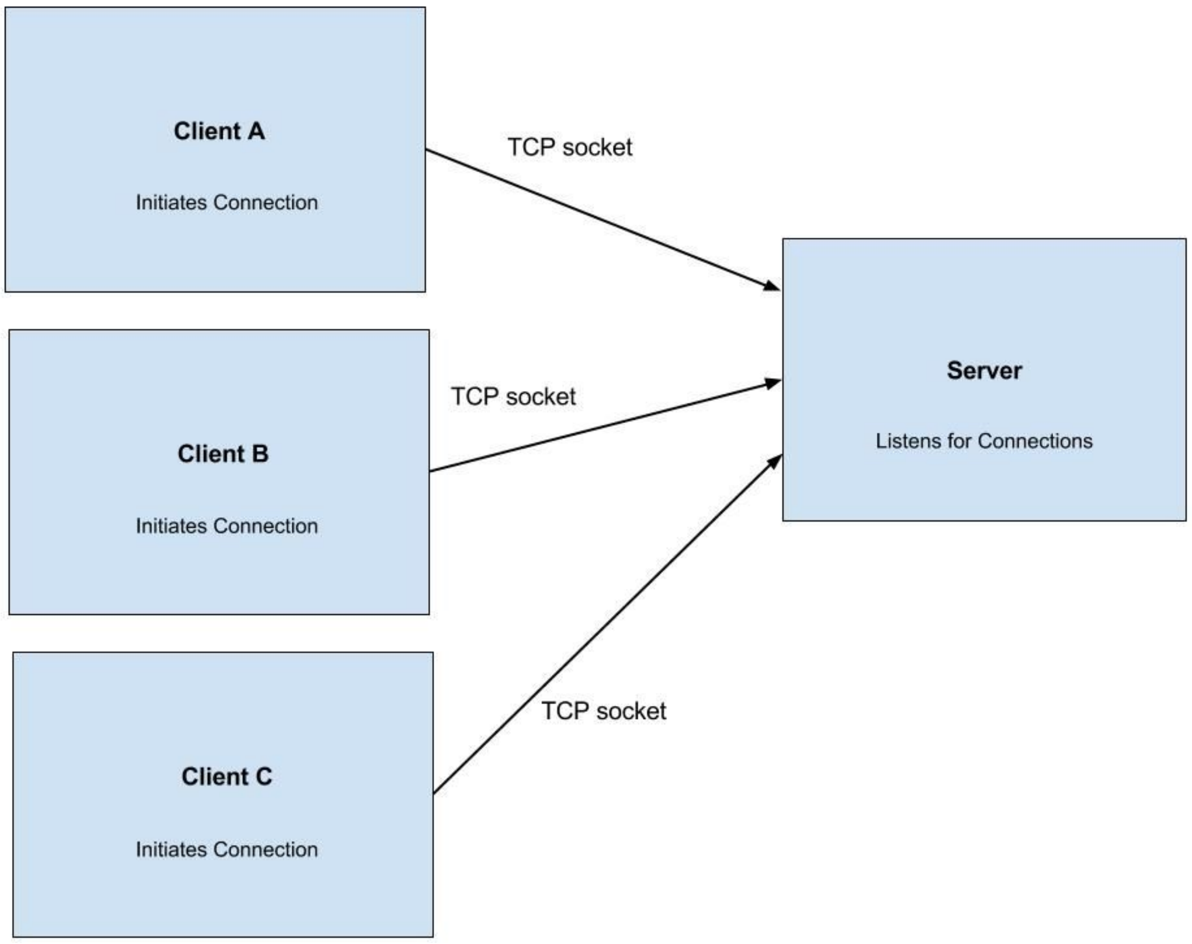
The Server

The server needs to listen for incoming TCP connections on a specific port. In case a client connects, it must read the data and disconnect the client. The server must keep listening for more such connections. The server needs to write the received text in a log file. The server must give the user a way to shut it down gracefully.

Project Scope

This application does not need to go beyond the given scenario of getting 1024 bytes of data at client side and logging that at server side. This concept can be improved upon exponentially, with possibility of sending large volume of data in chunks and using buffers or message queuing to help both sides keep up with each other. Handling more complex data formats and file types. In addition, server side does not need to validate or check the nature of incoming data. Because given the constraints, it knows client already did that.
.

High Level Interaction between client and server application:



Technical Design

Client

Client application needs to read random data from the /dev/urandom system device. It needs to manipulate said data, validate the data before sending and finally it needs to send the valid data over TCP socket.

Modularize the logical components and make sure each module solves one task. Also ensure easy reusability of each component.

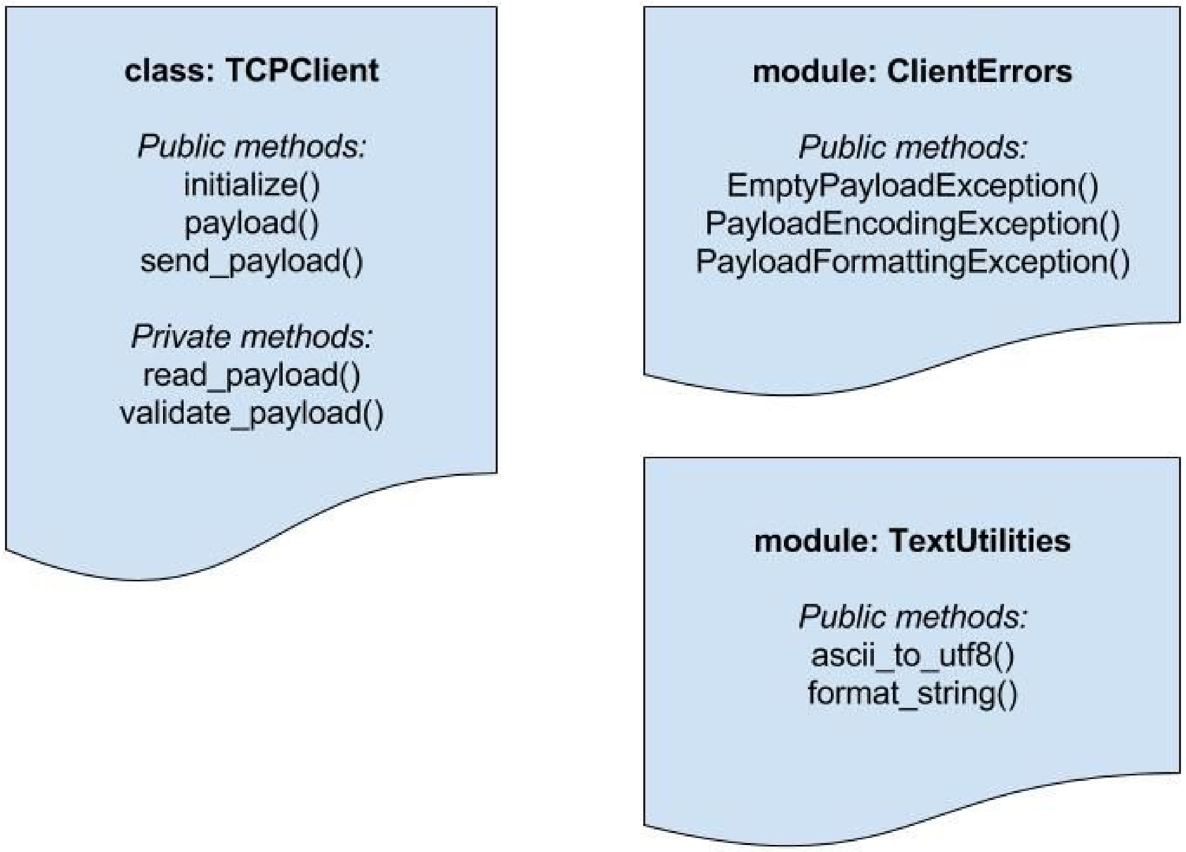
The application need to implement a TCP client. Core component class should be called **TCPClient**. It should implement these methods:

- initialize / constructor: Initialize the basic parameters for connection and also the data size to read. Should return the object.
- read_payload: Read the data from the source and handle the encoding and formatting tasks. Should return data in final format and encoding.
- validate_payload: Should check the payload and raise errors in case of mismatch. Should return nothing.
- payload: Should expose the payload text. For checking the current state of the text
- send_payload: Main logic flow. Returns false if anything goes wrong. Otherwise, returns true. Should call read_payload. Then validate it using validate_payload. If validation passes, open a TCP socket and attempt to send the payload. Should handle all possible error conditions and end execution on success or failure with proper return value.

Implement some custom errors for validation in the module **ClientErrors**. The custom errors should extend StandardError and provide basic error description. This gives us a way to standardise the error message based of the problem faced.

Implement the text conversion and formatting routines in the module called **TextUtilities**. Each methods here should done one set of modification and have defined expectation and return value. Should handle nil/empty input strings.

Diagram of components in Client application:



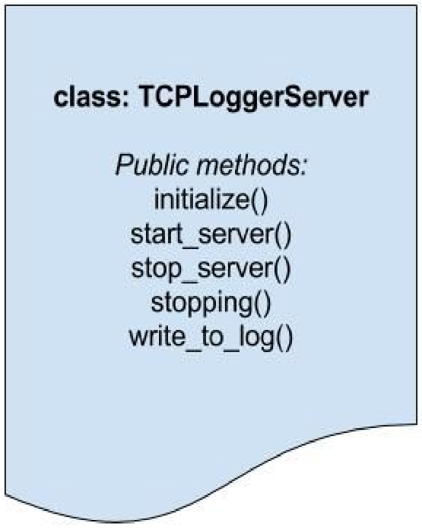
Server

Server application needs to implement a threaded server that continuously listens to a port. In case of incoming connections, it needs to start a new thread and handle the connection. It needs to read data from the socket and close the socket. In case of any errors, it needs to report the problem. If data was read successfully, server needs to append the data in the logfile with timestamp. Since this is a threaded application, the file write needs to be thread safe. Finally, the server needs provide to trap INT signal so that the user can shut it down using CTRL+C.

Since the server mostly does one thing, it can implement its functionalities in one class. The class should be named **TCPLoggerServer**. It should have the methods:

- initialize / constructor: Initialize the basic parameters for connection and also the name of the logfile. Should return the object.
- server_start: Main logic flow. Starts TCPServer instance. Starts a loop that handles incoming socket connections in a new thread. Handles incoming data. Provides error handling.
- server_stop: Kill the server process on demand.
- stopping: Print shutdown message and exit process indicating success.
- write_to_log: Append text to logfile. Lock file before and unlock after write, to be thread safe.

Diagram of components in Client application:



Testing And Coverage Reports

Test both the applications using **Rspec**. Put the specs in logical hierarchy inside spec/ directory. Make sure all the components are tested separately as well. Test for success and failure conditions. Add a rake task for running the whole test suite using:

```
rake rspec
```

Use **Simplecov** for generating test coverage reports. Test coverages should be put inside coverage/ directory. And the report should be in html format viewable in any web browser.

Note: Simplecov generates the reports automatically when the test suite is run using rspec.