ID 2201 Report

Rudy : A Small Web Server

Yilai Chen

Project Overview

This project involves building a simple HTTP server using Erlang, comprising four key modules: `http.erl` for request parsing and response generation, `rudy.erl` as the core server logic, `control.erl` for managing server start/stop functions, and `test.erl` for benchmarking performance. The server handles basic HTTP requests and returns appropriate responses. A small benchmark program generates multiple requests and measures the server's response time.

Concept Explanation

1. Socket API Procedures

To create a server that communicates using TCP/IP sockets, the following steps were followed:

- **Opening a socket**: A listening socket is created using `gen_tcp:listen/2` with options such as `{active, false}` for passive mode and `{reuseaddr, true}` to allow reuse of the port.
- **Accepting connections**: Once a socket is opened, the server calls `gen_tcp:accept/1`, which waits for an incoming connection.
- **Receiving data**: After accepting a connection, data from the client is received using `gen_tcp:recv/2`.
- **Sending data**: The server responds using `gen_tcp:send/2` to transmit data back to the client.
- **Closing connections**: After completing the communication, the server closes the socket using `gen_tcp:close/1` to release resources.

2. Structure of the Server Process

The server is structured in a loop that continuously listens for and accepts connections:

- **Initialization**: The server is initialized using the `init/1` function, which opens the listening socket on a specified port.
- **Connection Handling**: The `handler/1` function accepts incoming connections in a recursive loop, ensuring the server is always ready for new clients.
- **Request Processing**: Once a connection is established, the server processes the client's request by parsing the HTTP request (using `http:parse_request/1`) and then sends an appropriate response using `reply/1`.
- **Concurrency**: Whenever a new connection comes in, the server starts a new Erlang process using `spawn/1` to handle the connection. This means that each connection has its own separate Erlang process to handle the request.

3. HTTP Protocol

The HTTP protocol is a request-response protocol used by web servers and clients (e.g., browsers) to exchange information:

- **Request Line**: A typical HTTP request consists of a method (such as `GET`), a URI (such as `/index.html`), and the HTTP version (`HTTP/1.1`).
- **Headers**: Following the request line, the client may send headers (such as `Host` or `User-Agent`) that provide additional information about the request.
- **Response**: The server responds with a status code (e.g., `200 OK` for a successful request), followed by headers and an optional body containing the requested resource.
- **Statelessness**: HTTP is stateless, meaning each request is independent and carries all the information necessary for the server to understand and process it.

Core program implementation

1. **init()**

```
init(Port) -> % 初始化服务器，监听端口
  Opt = [list, {active, false}, {reuseaddr, true}], % 套接字，被动模式，重用地址
  case gen_tcp:listen(Port, Opt) of
    {ok, Listen} ->
      handler(Listen), % 处理任何进来的连接
      gen_tcp:close(Listen), % 关闭套接字
      ok;
    {error, Error} ->
      io:format("Listen error: ~p~n", [Error]), % 打印错误信息，返回error
      error
  end.
```

2. **handler()**

```
handler(Listen) -> % 处理传入的客户端连接
  case gen_tcp:accept(Listen) of
    {ok, Client} ->
      request(Client), % 处理客户端请求
      handler(Listen);  % 循环监听新的连接
    {error, Error} ->
      io:format("Accept error: ~p~n", [Error]),
      error
  end.
```

3. **request()**

```
request(Client) ->  % 接收和处理客户端的 HTTP 请求
  Recv = gen_tcp:recv(Client, 0),
  case Recv of
    {ok, Str} ->
      case string:tokens(Str, " ") of
        ["GET", URI, "HTTP/1.1\r\n"] ->
          Request = {get, URI, v11}; % 构建 Request 元组
        _ ->
          Request = {error, "Invalid Request"} % 如果请求不匹配，返回一个错误
```

```
      end,
      Response = reply(Request),
      gen_tcp:send(Client, Response);
    {error, Error} ->
      io:format("rudy: error: ~w~n", [Error])
  end,
  gen_tcp:close(Client).
```

4. **reply()**

```
reply({{{get, URI, _}, _, _}}) -> % 生成并发送 HTTP 响应
  timer:sleep(40),
  http:ok("<html><body><h1>OK</h1></body></html>");
reply({error, "Invalid Request"}) ->
  timer:sleep(40),
  "HTTP/1.1 400 Bad Request\r\nContent-Type: text/html\r\n\r\n<html><body>
<h1>400 Bad Request</h1></body></html>".
```

Benchmark and Performance Evaluation

A small benchmark program was developed to measure server performance by generating requests and recording the time taken to receive responses. The benchmark helps answer the following questions:

1. **Requests per Second**: By running multiple tests, we can calculate how many requests per second the server can handle. The number of requests per second is influenced by factors such as network speed, server processing power, and the efficiency of the socket API.

   A total of ten experiments were conducted to take the average:

						Avg.(μs)
Original(μs)	1	2	3	4	5	
	70758	74752	76903	74035	95437	
	6	7	8	9	10	
	78541	73216	70759	74137	78029	**76657**
Artificial Delayed(μs)	1	2	3	4	5	
	5307801	5112423	5050470	5165568	4910899	
	6	7	8	9	10	
	4809728	4791193	4961689	4834919	4748903	**4969359**

   According to the experimental data and delay, 100 requests were completed in 4.969359 seconds, and one request was completed in about 0.0497 seconds, which means the speed is about 20.12 requests per second.

2. **Impact of Artificial Delay**:

   The server introduces an artificial delay (using `timer:sleep/40`) in responding to requests. And the above results show that the delay has a significant impact on the results

3. **Concurrent Benchmarks on Multiple Machines**: Running the benchmark from several machines simultaneously simulates real-world scenarios with multiple clients. As the number of machines increases, the server's performance should initially scale, but after a certain point, we may see a gradual increase in response times as process management overhead starts to become significant.

Increasing throughput

Create a new process for each request:

```
handler(Listen) -> % 处理传入的客户端连接
  case gen_tcp:accept(Listen) of
    {ok, Client} ->
      spawn(fun() -> request(Client) end), % 为每个客户端请求创建新进程处理
      handler(Listen);  % 循环监听新的连接
    {error, Error} ->
      io:format("Accept error: ~p~n", [Error]),
      error
  end.
```

- In `handler/1`, every time a client connection is received, a new process is created by calling `spawn(fun() -> request(Client) end)` to handle the request.
- This approach allows multiple requests to be processed concurrently, improving server throughput.