Server Documentation

Overview

The

server

folder contains a Rust-based server application. The server handles HTTP POST requests to start downloads and uses WebSockets to communicate with clients. This documentation provides an overview of the server's structure and highlights key aspects related to multithreading, mutexes, and other OS-level considerations.

Folder Structure

```
server/
    .gitignore
    Cargo.lock
    Cargo.toml
    src/
        main.rs
        websocket.rs
    target/
```

Key Files

`Cargo.toml`

Defines the dependencies and configuration for the Rust project.

```
[package]
name = "server"
version = "0.1.0"
edition = "2021"

[dependencies]
serde = { version = "1.0.215", features = ["derive"] }
serde_json = "1.0.133"
tiny_http = "0.12.0"
urlencoding = "2.1.3"
ws = "0.9.2"
```

`src/main.rs`

The main entry point for the server application. It sets up the WebSocket server, handles incoming HTTP POST requests, and spawns the `yt-dlp` command to download videos.

Key Functions

- `main()` : Initializes the WebSocket server, finds the local IP address, and starts the HTTP server.
- `get_local_ip()` : Retrieves the local IP address of the machine.

Multithreading and Mutexes

The server uses multithreading and mutexes to handle concurrent client connections and manage shared resources.

```
fn main() {
    let clients = Arc::new(Mutex::new(Vec::new()));
    let clients_clone = clients.clone();

    // Start WebSocket server
    thread::spawn(move || {
        ws::listen("0.0.0.0:8001", |out| {
            WebSocketHandler {
                out,
                clients: clients_clone.clone(),
            }
```

```
        }
    })
    .unwrap()
});

// Find the local IP address
let local_ip = get_local_ip().unwrap_or_else(|| "Unknown IP".to_string());
// Clear the screen
print!("\x1B[2J\x1B[1;1H");
println!("\nLocal IP address of server: {}\n", local_ip);

// Start the server
let server = Server::http("0.0.0.0:8000").unwrap();
println!("Server started on http://0.0.0.0:8000");

for mut request in server.incoming_requests() {
    if request.method() == &Method::Post {
        let mut content = String::new();
        request.as_reader().read_to_string(&mut content).unwrap();

        // Parse the URL from the POST request body
        let url_encoded = content.split('=').nth(1).unwrap_or("").trim();
        let url = decode(url_encoded).unwrap_or_else(|_| "".into());

        if !url.is_empty() {
            // Spawn yt-dlp command
            let mut child = Command::new("yt-dlp")
                .arg(url.to_string())
                .stdout(Stdio::piped())
                .spawn()
                .expect("Failed to execute yt-dlp");

            // Read the output line by line
            if let Some(stdout) = child.stdout.take() {
                let reader = BufReader::new(stdout);
                for line in reader.lines() {
                    match line {
                        Ok(line) => {
                            println!("{}", line);
                            broadcast_message(clients.clone(), &line);
                        }
                        Err(e) => {
                            eprintln!("Error reading line: {}", e);
                            broadcast_message(clients.clone(), &format!("Error: {}", e));
                        }
                    }
                }
            }

            // Wait for the command to finish
            let _ = child.wait();

            // Send a response to the client
            let response = Response::from_string(
                "Download started. Check the server console for progress.",
            );
            request.respond(response).unwrap();
        } else {
            let response = Response::from_string("Invalid URL").with_status_code(400);
            request.respond(response).unwrap();
        }
    } else {
        let response =
            Response::from_string("Only POST method is supported").with_status_code(405);
        request.respond(response).unwrap();
    }
```

```
    }
}
```

`src/websocket.rs`

Defines the WebSocket handler and functions for broadcasting messages to connected clients.

Key Structures and Functions

- `ConsoleMessage`: A struct representing a message to be sent to the console.
- `WebSocketHandler`: A struct implementing the `ws::Handler` trait to handle WebSocket events.
- `broadcast_message()`: A function to broadcast a message to all connected WebSocket clients.

Multithreading and Mutexes

The WebSocket handler uses a `Mutex` to manage access to the list of connected clients.

```
 use serde::{Deserialize, Serialize};
use std::sync::{Arc, Mutex};
use ws::{Handler, Handshake, Message, Result, Sender};

#[derive(Serialize, Deserialize)]
pub struct ConsoleMessage {
    message_type: String,
    content: String,
}

pub struct WebSocketHandler {
    pub out: Sender,
    pub clients: Arc<Mutex<Vec<Sender>>>,
}

impl Handler for WebSocketHandler {
    fn on_open(&mut self, _: Handshake) -> Result<()> {
        let mut clients = self.clients.lock().unwrap();
        clients.push(self.out.clone());
        Ok(())
    }

    fn on_message(&mut self, msg: Message) -> Result<()> {
        // Echo message back for testing
        self.out.send(msg)
    }

    fn on_close(&mut self, _code: ws::CloseCode, _reason: &str) {
        let mut clients = self.clients.lock().unwrap();
        clients.retain(|client| client != &self.out);
    }
}

pub fn broadcast_message(clients: Arc<Mutex<Vec<Sender>>>, message: &str) {
    let console_msg = ConsoleMessage {
        message_type: "console".to_string(),
        content: message.to_string(),
    };

    let msg = serde_json::to_string(&console_msg).unwrap();

    let clients = clients.lock().unwrap();
    for client in clients.iter() {
        let _ = client.send(msg.clone());
    }
}
```

OS-Level Concepts

Multithreading

Multithreading is used in the Rust server to handle multiple client connections concurrently. Each client connection is handled in a separate thread, allowing the server to manage multiple connections simultaneously without blocking.

```
thread::spawn(move || {
    ws::listen("0.0.0.0:8001", |out| {
        WebSocketHandler {
            out,
            clients: clients_clone.clone(),
        }
    })
    .unwrap()
});
```

Mutexes

Mutexes are used to ensure safe access to shared resources across multiple threads. In this project, a `Mutex` is used to manage the list of connected clients, ensuring that only one thread can modify the list at a time.

```
let clients = Arc::new(Mutex::new(Vec::new()));
let clients_clone = clients.clone();
```

Inter-Process Communication (IPC)

The server communicates with the `yt-dlp` process using standard input/output. The server spawns the `yt-dlp` command and reads its output line by line, broadcasting each line to connected WebSocket clients.

```
let mut child = Command::new("yt-dlp")
    .arg(url.to_string())
    .stdout(Stdio::piped())
    .spawn()
    .expect("Failed to execute yt-dlp");

if let Some(stdout) = child.stdout.take() {
    let reader = BufReader::new(stdout);
    for line in reader.lines() {
        match line {
            Ok(line) => {
                println!("{}", line);
                broadcast_message(clients.clone(), &line);
            }
            Err(e) => {
                eprintln!("Error reading line: {}", e);
                broadcast_message(clients.clone(), &format!("Error: {}", e));
            }
        }
    }
}
```

Conclusion

The Rust-based server application uses multithreading and mutexes to handle concurrent client connections and manage shared resources efficiently. The server handles HTTP POST requests to start downloads and uses WebSockets to communicate with clients in real-time. The project demonstrates effective use of OS-level concepts such as multithreading, mutexes, and inter-process communication.