**Beyond the Threads: Websockets in Rust for seamless communication**

[[Tapas Das](https://tdtapas.medium.com/?source=post_page-----e40d10e8a0e3--------------------------------)](https://tdtapas.medium.com/?source=post_page-----e40d10e8a0e3--------------------------------)

[[Dev Genius](https://blog.devgenius.io/?source=post_page-----e40d10e8a0e3--------------------------------)](https://blog.devgenius.io/?source=post_page-----e40d10e8a0e3--------------------------------)

[Tapas Das](https://tdtapas.medium.com/?source=post_page-----e40d10e8a0e3--------------------------------)

·

Follow

Published in

[Dev Genius](https://blog.devgenius.io/?source=post_page-----e40d10e8a0e3--------------------------------)

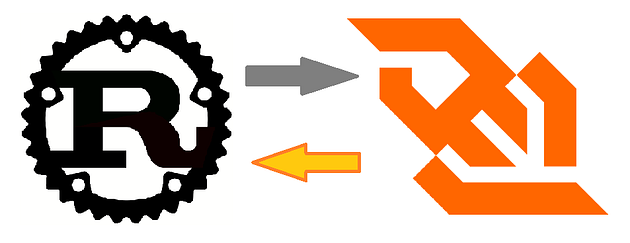
·

12 min read

·

Jan 3

25

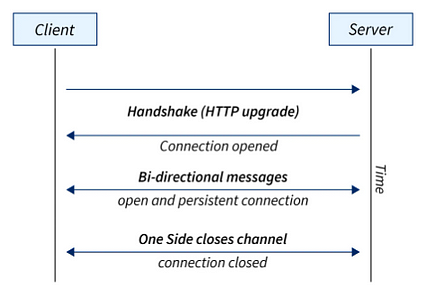


Imagine you have two toy walkie-talkies, and you and your friend want to talk to each other. Normally, one person talks, and the other person listens, like when you take turns talking on the walkie-talkie.

Now, think of websockets like magical walkie-talkies that let you and your friend talk at the same time, like having a conversation! With regular walkie-talkies (HTTP), one person sends a message, and the other person receives it. But with websockets, it’s like you can both talk and listen whenever you want without waiting for the other person to finish!

So, websockets are like special walkie-talkies that make it super easy for you and your friend to have a fun and fast conversation, just like magic! You can play games, tell jokes, and share stories instantly. It’s like having a direct line to your friend, and you can talk back and forth as much as you want, making it really cool and fast!

Websockets are a communication protocol that enables two-way communication between a client (like a web browser) and a server over a single, long-lived connection. Unlike the traditional request-response model of HTTP, where a client sends a request to the server, and the server responds, websockets allow real-time, bidirectional communication.



In this article, I’ll be discussing about setting up websockets with the awesomeness of Rust, in order to achieve efficient and low-latency real-time communication.

Refer below article for more information about how websockets work.

**[System Design Basics: WebSockets](https://medium.com/geekculture/system-design-basics-websockets-80aa2b5d5e52?source=post_page-----e40d10e8a0e3--------------------------------" \t "_blank)**

[What are WebSockets? How are they different from traditional network protocols?](https://medium.com/geekculture/system-design-basics-websockets-80aa2b5d5e52?source=post_page-----e40d10e8a0e3--------------------------------" \t "_blank)

[medium.com](https://medium.com/geekculture/system-design-basics-websockets-80aa2b5d5e52?source=post_page-----e40d10e8a0e3--------------------------------" \t "_blank)

Before we start, familiarity with **[tokio](https://tokio.rs/" \t "_blank)** and [**warp**](https://github.com/seanmonstar/warp) crates, in Rust, for building webservers will be extremely useful, since these crates will be doing all the heavy-lifting for the websocket communication.

**Setting up the project**

Let’s start by first setting up a brand new Cargo project for doing all our Rust coding for websockets.

cargo new rusty-websockets

**Defining the initial dependencies**

futures = { version = "0.3", default-features = false }  
serde = { version = "1.0", features = ["derive"] }  
serde\_json = "1.0"  
tokio = { version = "1.28", features = ["full"] }  
tokio-stream = "0.1"  
ulid = { version = "1.1", features = ["serde"] }  
warp = { version = "0.3", features = ["tls"] }



1. **futures** to help with asynchronous data streams of the WebSocket.
2. **serde** to serialize and deserialize Rust data structures efficiently and generically.
3. **serde\_json** to serialize and deserialize JSON.
4. **tokio** to build a runtime for writing reliable, asynchronous, and slim applications.
5. **tokio-stream** to get the utilities required for working with streamand tokio.
6. **ulid** to generate unique identifiers for each user.
7. **warp** to setup the web server framework for websocket.

**Setting the websocket client**

use std::{collections::HashMap, convert::Infallible, sync::Arc};  
use tokio::sync::{mpsc, Mutex};  
use warp::{ws::Message, Filter};  
  
#[derive(Debug, Clone)]  
/// Defines the structure for connected websocket client  
pub struct Client {  
 /// Unique identifier (ulid) for users who are using the client  
 pub user\_id: String,  
  
 /// Sender is used to send messages to the connected client (mpsc::UnboundedReceiver)  
 pub sender: Option<mpsc::UnboundedSender<std::result::Result<Message, warp::Error>>>,  
}  
  
/// Map of connection IDs for clients that can be safely passed across threads  
pub type Clients = Arc<Mutex<HashMap<String, Client>>>;  
  
/// Function to extract the "Clients" data and return a Filter matching any route.  
pub fn with\_clients(  
 clients: Clients,  
) -> impl Filter<Extract = (Clients,), Error = Infallible> + Clone {  
 warp::any().map(move || clients.clone())  
}

First, I have defined a struct “**Client”** for representing the connected client, consisting of below fields.

1. **user\_id**: Randomly generated ulid
2. **sender**: This is a **mpsc::UnboundedSender** type that allows to send messages to the **UnboundedReceiver**, which is the client

Next, I have defined a new alias type with definition type **Clients = Arc<Mutex<HashMap<String, Client>>>;**. This helps keeping track of connected clients in a HashMap wrapped in a Mutex from the tokio library, which is then wrapped in an Arc from the std (standard) library.

The Mutexallows for locking the HashMap resource to prevent deadlocks or race conditions if multiple threads try to access the HashMap.

The Arcis a thread-safe reference-counting pointer, this allows us to share the data with multiple threads, which is very useful for an asynchronous web server.

The filter function **with\_clients** extracts the *Clients* data, and returns a Warp Filter matching any route and composes the filter with a function receiving the extracted data, in this case the **clients**.

**Setting the websocket server**

use std::{collections::HashMap, convert::Infallible, net::Ipv4Addr, sync::Arc};  
use tokio::sync::{mpsc, Mutex};  
use warp::{  
 reject::Rejection,  
 reply::Reply,  
 ws::{Message, Ws},  
 Filter,  
};  
  
/// Function to run the websocket server on given IP address and Port number  
pub async fn start(ip\_addr: Ipv4Addr, port: u16) {  
 // Creating new instance of the "Clients" type  
 let clients: Clients = Arc::new(Mutex::new(HashMap::new()));  
  
 println!("Configuring websocket route");  
  
 // Creating the websocket route for the server  
 // 1) Define the path as "ws". So the full path for the client will become: <ip\_addr>:<port>/ws  
 // 2) Add a WebSocket filter that yields a "Ws" object that will be used to upgrade the connection to a WebSocket connection.  
 // 3) Add new instance of "Clients" type  
 // 4) Configure the handler function that is called to handle this route.  
 let ws\_route = warp::path("ws")  
 .and(warp::ws())  
 .and(with\_clients(clients.clone()))  
 .and\_then(ws\_handler);  
  
 // Adding a CORS filter that allows any origin  
 let routes = ws\_route.with(warp::cors().allow\_any\_origin());  
  
 println!("Starting server @ {}:{}", ip\_addr, port);  
  
 // Running the Warp server on given IP address and Port number  
 warp::serve(routes).run((ip\_addr, port)).await;  
}

First, I have defined a “**start**” function which takes *IP Address* and *Port Number* as input argument. This function is used for starting the websocket server.

Next, I’m creating a new instance of the “**Clients**” type using *Arc*, *Mutex* and *Hashmap*. The client instance is being passed to the “**ws\_handler**” function using the filter “**with\_clients**”.

I have defined the websocket routes (**ws**) using **warp::path**; so the full path for the client will become: **<ip\_addr>:<port>/ws**. I have also added a CORS filter to the routes configuration to allow any origin.

Finally, I am running the warp server at provided IP Address and Port, with the routes configuration.

**Websocket Handler**

/// Handler function to receive the HashMap of clients, and pass this to the client\_connection function in the ws module  
async fn ws\_handler(ws: Ws, clients: Clients) -> Result<impl Reply, Rejection> {  
 // Websocket protocol upgrade for handling incoming communications  
 Ok(ws.on\_upgrade(move |socket| ws\_conn::client\_connection(socket, clients)))  
}

Here, I am creating a very simple asynchronous function, that then calls another function and returns its result, that implements **Reply**.

A warp *Reply* is a type that can be converted into a HTTP response. The function calls **ws.on\_upgrade**, which finishes the websocket protocol upgrade and configures the function (**client\_connection**, which we will write after this section) to be used to handle websocket communication.

**Websocket Connection**

use crate::ws\_server::{Client, Clients};  
use futures::{FutureExt, StreamExt};  
use tokio::sync::mpsc;  
use tokio\_stream::wrappers::UnboundedReceiverStream;  
use ulid::Ulid;  
use warp::ws::{Message, WebSocket};  
  
/// Establishes a websocket connection with given client  
pub async fn client\_connection(ws: WebSocket, clients: Clients) {  
 println!("Establishing client connection... {:?}", ws);  
  
 // Splitting the WebSocket stream object into separate Sender and Receiver objects, for individual tasks ownership  
 let (client\_ws\_sender, mut client\_ws\_receiver) = ws.split();  
  
 // Creating unbounded channel and splitting into sender and receiver streams  
 let (client\_sender, client\_receiver) = mpsc::unbounded\_channel();  
  
 // Defining the receiver stream for receiving messages from channel  
 let client\_receiver = UnboundedReceiverStream::new(client\_receiver);  
  
 // Spawning separate thread for keeping the sender stream open until the client has disconnected  
 tokio::task::spawn(client\_receiver.forward(client\_ws\_sender).map(|result| {  
 if let Err(e) = result {  
 eprintln!("Failed to send message using websocket - {}", e.to\_string());  
 }  
 }));  
  
 // Generating unique identifier for the user  
 let ulid: String = Ulid::new().to\_string();  
  
 // Creating new "Client" struct instance for given user and sender stream  
 let new\_client: Client = Client {  
 user\_id: ulid.clone(),  
 sender: Some(client\_sender),  
 };  
  
 // Acquiring lock on the client list and inserting the "new\_client" object into the clients HashMap  
 clients.lock().await.insert(ulid.clone(), new\_client);  
  
 // Loop to handle the incoming messages from the client  
 // The loop will keep running until the client is disconnected.  
 while let Some(result) = client\_ws\_receiver.next().await {  
 let msg = match result {  
 Ok(msg) => msg,  
 Err(e) => {  
 eprintln!(  
 "Failed to receive message using websocket - {}",  
 e.to\_string()  
 );  
 break;  
 }  
 };  
 client\_msg(&ulid, msg, &clients).await;  
 }  
  
 clients.lock().await.remove(&ulid);  
 println!("Websocket disconnected.");  
}

I am starting off by splitting the WebSocket stream into separate Sender (**client\_ws\_sender**) and Receiver (**client\_ws\_receiver**) objects, for individual ownership. Next, I am creating an unbounded MPSC channel, and splitting that also into Sender (**client\_sender**) and Receiver (**client\_receiver**) streams.

I have defined the unbounded receiver stream for receiving messages. Using Tokio, I am spawning a new task that keeps the **client\_ws\_sender** stream open until the client has disconnected.

Next, I have defined the **Client** object (*new\_client*), and obtain a lock on the client list and insert the *new\_client* object into the clients HashMap using ulid as the key.

I have then defined an infinite loop that receives the next item in the stream. This loop keeps running until the client is disconnected. When the client sends a message the loop will be entered and the message will be extracted and then further processed by **client\_msg (**which we will write after this section).

When the client can no longer be reached due to having disconnected or for other reasons, the infinite loop waiting for items will stop, and code execution will move to last line, where we remove the client by id from the HashMap tracking the clients.

**Websocket Message**

/// Send and receive message to/from given client  
async fn client\_msg(user\_id: &str, msg: Message, clients: &Clients) {  
 println!("Received message from {}: {:?}", user\_id, msg);  
  
 let message: &str = match msg.to\_str() {  
 Ok(v) => v,  
 Err(\_) => return,  
 };  
  
 if message == "ping" || message == "ping\n" {  
 let locked = clients.lock().await;  
 match locked.get(user\_id) {  
 Some(v) => {  
 if let Some(sender) = &v.sender {  
 println!("Sending pong");  
 let \_ = sender.send(Ok(Message::text("pong")));  
 }  
 }  
 None => return,  
 }  
 return;  
 };  
}

The function takes **user\_id**as input argument, so that I can retrieve the latest information about the client from the clients map if we need it.

Next, I am converting the received message to a string. Then I am checking if the message says “**ping**”, if so, we go into the process of sending “**pong**” back to the client.

I am first getting the client by **client\_id** from the clients map, if the client is found, then the sender object from the client sends a **Message::text** containing “**pong**”. In any other case, I simply return from the function.

**Putting it all together**

*ws\_server.rs*

//! Module to implement websocket server to receive and process incoming requests from clients  
  
use crate::ws\_conn;  
use std::{collections::HashMap, convert::Infallible, net::Ipv4Addr, sync::Arc};  
use tokio::sync::{mpsc, Mutex};  
use warp::{  
 reject::Rejection,  
 reply::Reply,  
 ws::{Message, Ws},  
 Filter,  
};  
  
#[derive(Debug, Clone)]  
/// Defines the structure for connected websocket client  
pub struct Client {  
 /// Unique identifier (ulid) for users who are using the client  
 pub user\_id: String,  
  
 /// Sender is used to send messages to the connected client (mpsc::UnboundedReceiver)  
 pub sender: Option<mpsc::UnboundedSender<std::result::Result<Message, warp::Error>>>,  
}  
  
/// Map of connection IDs for clients that can be safely passed across threads  
pub type Clients = Arc<Mutex<HashMap<String, Client>>>;  
  
/// Function to extract the "Clients" data and return a Filter matching any route.  
pub fn with\_clients(  
 clients: Clients,  
) -> impl Filter<Extract = (Clients,), Error = Infallible> + Clone {  
 warp::any().map(move || clients.clone())  
}  
  
/// Function to run the websocket server on given IP address and Port number  
pub async fn start(ip\_addr: Ipv4Addr, port: u16) {  
 // Creating new instance of the "Clients" type  
 let clients: Clients = Arc::new(Mutex::new(HashMap::new()));  
  
 println!("Configuring websocket route");  
  
 // Creating the websocket route for the server  
 // 1) Define the path as "ws". So the full path for the client will become: <ip\_addr>:<port>/ws  
 // 2) Add a WebSocket filter that yields a "Ws" object that will be used to upgrade the connection to a WebSocket connection.  
 // 3) Add new instance of "Clients" type  
 // 4) Configure the handler function that is called to handle this route.  
 let ws\_route = warp::path("ws")  
 .and(warp::ws())  
 .and(with\_clients(clients.clone()))  
 .and\_then(ws\_handler);  
  
 // Adding a CORS filter that allows any origin  
 let routes = ws\_route.with(warp::cors().allow\_any\_origin());  
  
 println!("Starting server @ {}:{}", ip\_addr, port);  
  
 // Running the Warp server on given IP address and Port number  
 warp::serve(routes).run((ip\_addr, port)).await;  
}  
  
/// Handler function to receive the HashMap of clients, and pass this to the client\_connection function in the ws module  
async fn ws\_handler(ws: Ws, clients: Clients) -> Result<impl Reply, Rejection> {  
 // Websocket protocol upgrade for handling incoming communications  
 Ok(ws.on\_upgrade(move |socket| ws\_conn::client\_connection(socket, clients)))  
}

*ws\_conn.rs*

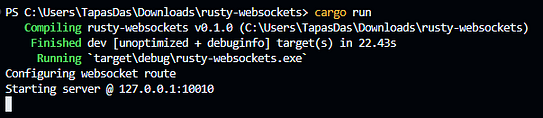
//! Module to handle the established websocket connection and receiving messages from the websocket client  
  
use crate::ws\_server::{Client, Clients};  
use futures::{FutureExt, StreamExt};  
use tokio::sync::mpsc;  
use tokio\_stream::wrappers::UnboundedReceiverStream;  
use ulid::Ulid;  
use warp::ws::{Message, WebSocket};  
  
/// Establishes a websocket connection with given client  
pub async fn client\_connection(ws: WebSocket, clients: Clients) {  
 println!("Establishing client connection... {:?}", ws);  
  
 // Splitting the WebSocket stream object into separate Sender and Receiver objects, for individual tasks ownership  
 let (client\_ws\_sender, mut client\_ws\_receiver) = ws.split();  
  
 // Creating unbounded channel and splitting into sender and receiver streams  
 let (client\_sender, client\_receiver) = mpsc::unbounded\_channel();  
  
 // Defining the receiver stream for receiving messages from channel  
 let client\_receiver = UnboundedReceiverStream::new(client\_receiver);  
  
 // Spawning separate thread for keeping the sender stream open until the client has disconnected  
 tokio::task::spawn(client\_receiver.forward(client\_ws\_sender).map(|result| {  
 if let Err(e) = result {  
 eprintln!("Failed to send message using websocket - {}", e.to\_string());  
 }  
 }));  
  
 // Generating unique identifier for the user  
 let ulid: String = Ulid::new().to\_string();  
  
 // Creating new "Client" struct instance for given user and sender stream  
 let new\_client: Client = Client {  
 user\_id: ulid.clone(),  
 sender: Some(client\_sender),  
 };  
  
 // Acquiring lock on the client list and inserting the "new\_client" object into the clients HashMap  
 clients.lock().await.insert(ulid.clone(), new\_client);  
  
 // Loop to handle the incoming messages from the client  
 // The loop will keep running until the client is disconnected.  
 while let Some(result) = client\_ws\_receiver.next().await {  
 let msg = match result {  
 Ok(msg) => msg,  
 Err(e) => {  
 eprintln!(  
 "Failed to receive message using websocket - {}",  
 e.to\_string()  
 );  
 break;  
 }  
 };  
 client\_msg(&ulid, msg, &clients).await;  
 }  
  
 clients.lock().await.remove(&ulid);  
 println!("Websocket disconnected.");  
}  
  
/// Send and receive message to/from given client  
async fn client\_msg(user\_id: &str, msg: Message, clients: &Clients) {  
 println!("Received message from {}: {:?}", user\_id, msg);  
  
 let message: &str = match msg.to\_str() {  
 Ok(v) => v,  
 Err(\_) => return,  
 };  
  
 if message == "ping" || message == "ping\n" {  
 let locked = clients.lock().await;  
 match locked.get(user\_id) {  
 Some(v) => {  
 if let Some(sender) = &v.sender {  
 println!("Sending pong");  
 let \_ = sender.send(Ok(Message::text("pong")));  
 }  
 }  
 None => return,  
 }  
 return;  
 };  
}

*main.rs*

mod ws\_conn;  
mod ws\_server;  
  
use std::net::Ipv4Addr;  
  
#[tokio::main]  
async fn main() {  
 // IP Address for websocket connection  
 let ip\_addr: Ipv4Addr = "127.0.0.1".parse().unwrap();  
  
 // Default port for websocket connection  
 let port: u16 = 10010;  
  
 ws\_server::start(ip\_addr, port).await;  
}

**Testing the websocket connection**

Let’s first start the websocket server using: cargo run



Then, we can test the websocket using Postman at below address.

ws://127.0.0.1:10010/ws



**Securing the websocket connection**

The websocket that we deployed is not secured for internet facing applications. So we need to secure the connection **TLS (Transport Layer Security)**so that messages can be delivered in encrypted and secured format to prevent attackers from listening in on conversations.

In order to achieve this, we need to generate a public-private key pair which will be used to by Warp to serve the content securely.

openssl req -newkey rsa:2048 -new -nodes -x509 -days 3650 -keyout ssl\_privkey.pem -out ssl\_cert.pem

This command will generate 2 files.

1. ssl\_cert.pem — SSL certificate (public key)
2. ssl\_privkey.pem — SSL private key

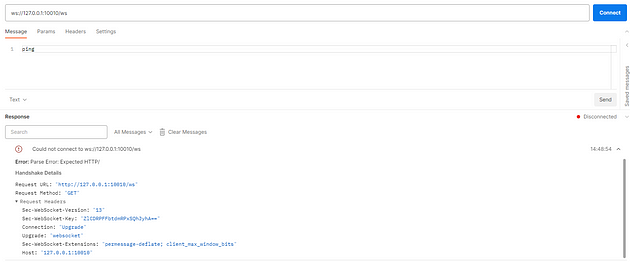
Once the files are generated, we can go ahead with configuring the same in **main.rs** first.

mod ws\_conn;  
mod ws\_server;  
  
use std::net::Ipv4Addr;  
  
#[tokio::main]  
async fn main() {  
 // IP Address for websocket connection  
 let ip\_addr: Ipv4Addr = "127.0.0.1".parse().unwrap();  
  
 // Default port for websocket connection  
 let port: u16 = 10010;  
  
 // Set SSL certificates path  
 let ssl\_cert: &str = "ssl\_cert.pem";  
 let ssl\_key: &str = "ssl\_privkey.pem";  
  
 ws\_server::start(ip\_addr, port, ssl\_cert, ssl\_key).await;  
}

Next, we will modify the warp server to use TLS connection and SSL certificates for security.

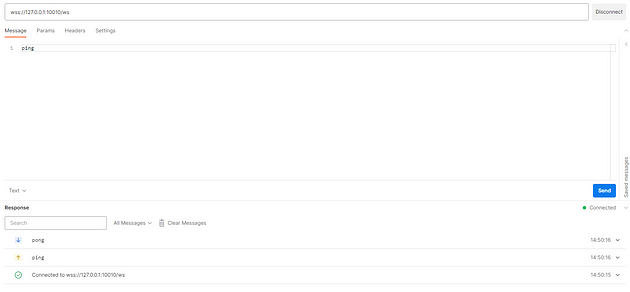
// Running the Warp server on given IP address and Port number  
warp::serve(routes)  
 .tls()  
 .cert\_path(ssl\_cert)  
 .key\_path(ssl\_key)  
 .run((ip\_addr, port)).await;

Let’s try to test the websocket connection now at ws://127.0.0.1:10010/ws.



We can see that the unsecured connection no longer works. We need to switch to secure websockets (**wss**) now to establish the connection.

wss://127.0.0.1:10010/ws



The secure connection is successfully established, and we can send and receive messages without worrying about attackers.

**Summary**

In this article, we discussed on how to setup websocket connection using Rust. We also looked at how to secure the websocket connection using TLS and SSL certificates.

We can improve on this by generating the SSL certificates programatically in Rust and trying it to user sessions to dynamically rotate the keys. We can also add logging and error handling mechanisms for efficiency and traceability.

Let me know your thoughts on this article.

[Rust](https://medium.com/tag/rust?source=post_page-----e40d10e8a0e3---------------rust-----------------)

[Websocket](https://medium.com/tag/websocket?source=post_page-----e40d10e8a0e3---------------websocket-----------------)

[Openssl](https://medium.com/tag/openssl?source=post_page-----e40d10e8a0e3---------------openssl-----------------)

[Tls](https://medium.com/tag/tls?source=post_page-----e40d10e8a0e3---------------tls-----------------)

[Realtime](https://medium.com/tag/realtime?source=post_page-----e40d10e8a0e3---------------realtime-----------------)