**Asynchronous Rust : Building and Testing a Real-Time Message Broker (Concept Implementation)**

[[Shobhit chaturvedi](https://medium.com/@learnwithshobhit?source=post_page-----02f524fb5ff7--------------------------------)](https://medium.com/@learnwithshobhit?source=post_page-----02f524fb5ff7--------------------------------)

[Shobhit chaturvedi](https://medium.com/@learnwithshobhit?source=post_page-----02f524fb5ff7--------------------------------)

·

Follow

4 min read

·

22 hours ago

4

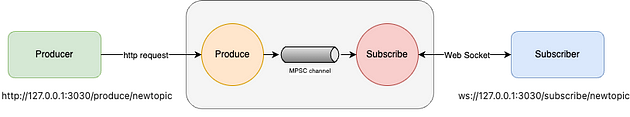
**Introduction**

In this article, we’ll delve into building a real-time message broker using Rust, showcasing its powerful concurrency features. We’ll use Warp for our web server and Tokio for managing asynchronous tasks. Additionally, we’ll create a WebSocket client to test the broker’s functionality.

**Prerequisites**

* Basic understanding of Rust and asynchronous programming.
* Familiarity with WebSockets and network communication concepts.

**Design**



Simple Message Broker (Rust)

**Part 1: Building the Message Broker**

**Overview**

**Overview of the Broker**

* The message broker will allow clients to produce events to topics and subscribe to them.
* It uses Warp for the HTTP and WebSocket server and Tokio for asynchronous handling.

**Setting Up the Broker**

Dependencies

First, let’s look at the dependencies:

use futures\_util::{SinkExt, StreamExt};  
use std::collections::{HashMap, VecDeque};  
use std::sync::{Arc, RwLock};  
use tokio::sync::mpsc;  
use tokio::sync::mpsc::UnboundedSender;  
use warp::filters::ws::Message;  
use warp::{self, Filter};

* futures\_util for working with futures and streams.
* Standard collections for managing data structures.
* Arc and RwLock for thread-safe shared state.
* tokio::sync::mpsc for message passing between tasks.
* warp for web server and WebSocket functionalities.

The Broker Structure

Here’s our broker struct:

type Topic = String;  
type Event = String;  
type WsSender = UnboundedSender<warp::ws::Message>;  
  
struct Broker {  
 events: Arc<RwLock<HashMap<Topic, VecDeque<Event>>>>,  
 subscribers: Arc<RwLock<HashMap<Topic, Vec<WsSender>>>>,  
}

* events: Stores events for each topic.
* subscribers: Keeps track of subscribers to each topic.

Broker Functionalitynew Function

Creates a new instance of the broker:

impl Broker {  
 fn new() -> Self {  
 Broker {  
 events: Arc::new(RwLock::new(HashMap::new())),  
 subscribers: Arc::new(RwLock::new(HashMap::new())),  
 }  
 }  
 // ... other functions will go here ...  
}

produce Function

Handles the publishing of events:

async fn produce(&self, topic: Topic, event: Event) {  
 let mut events = self.events.write().unwrap();  
 events  
 .entry(topic.clone())  
 .or\_default()  
 .push\_back(event.clone());  
  
 // Notify all subscribers asynchronously.  
 let subscribers\_list;  
 {  
 let subscribers = self.subscribers.read().unwrap();  
 subscribers\_list = subscribers.get(&topic).cloned().unwrap\_or\_default();  
 }  
  
 for ws\_sender in subscribers\_list {  
 // Attempt to send the event to the WebSocket client.  
 let \_ = ws\_sender.send(warp::ws::Message::text(event.clone()));  
 }  
 }

* Adds an event to the respective topic.
* Notifies all subscribers about the new event.

subscribe Function

Manages new subscriptions:

pub fn subscribe(&self, topic: Topic, socket: warp::ws::WebSocket) {  
 let (ws\_sender, mut ws\_receiver) = socket.split();  
  
 let (tx, mut rx) = mpsc::unbounded\_channel::<Message>();  
  
 {  
 let mut subs = self.subscribers.write().unwrap();  
 subs.entry(topic).or\_default().push(tx);  
 }  
  
 tokio::task::spawn(async move {  
 while let Some(result) = ws\_receiver.next().await {  
 match result {  
 Ok(message) => {  
 // Process the valid message here  
 if message.is\_text() {  
 println!(  
 "Received message from client: {}",  
 message.to\_str().unwrap()  
 );  
 // Additional processing...  
 }  
 }  
 Err(e) => {  
 // Handle the error (e.g., log it)  
 eprintln!("WebSocket error: {:?}", e);  
 break; // Optional: break the loop on error  
 }  
 }  
 }  
 println!("WebSocket connection closed");  
 });  
  
 tokio::task::spawn(async move {  
 let mut sender = ws\_sender;  
  
 while let Some(msg) = rx.recv().await {  
 let \_ = sender.send(msg).await;  
 }  
 });  
 }

* Splits the WebSocket into a sender and receiver.
* Adds the sender to the subscribers list.
* Handles incoming WebSocket messages.

**Server Setup and Route Handlers**

Setting up Warp server and routes:

#[tokio::main]  
async fn main() {  
 let broker = Arc::new(Broker::new());  
 let broker\_clone1 = Arc::clone(&broker);  
 let broker\_clone2 = Arc::clone(&broker);  
  
 let produce = warp::path!("produce" / String)  
 .and(warp::post())  
 .and(warp::body::json())  
 .and(warp::any().map(move || Arc::clone(&broker\_clone1)))  
 .and\_then(  
 move |topic: String, event: Event, broker\_clone2: Arc<Broker>| async move {  
 broker\_clone2.produce(topic, event).await;  
 Ok::<\_, warp::Rejection>(warp::reply())  
 },  
 );  
  
 let subscribe = warp::path!("subscribe" / String).and(warp::ws()).map(  
 move |topic: String, ws: warp::ws::Ws| {  
 let broker\_clone3 = Arc::clone(&broker\_clone2);  
 ws.on\_upgrade(move |socket| async move {  
 broker\_clone3.subscribe(topic.clone(), socket);  
 })  
 },  
 );  
  
 let routes = produce.or(subscribe);  
  
 println!("Broker server running at http://127.0.0.1:3030");  
 warp::serve(routes).run(([127, 0, 0, 1], 3030)).await;  
}

* produce route for event publishing.
* subscribe route for handling subscriptions.

**Part 2: Implementing the WebSocket Client**

**Client Overview**

The WebSocket client will simulate a real user subscribing to topics and receiving messages.

**Client Code**

Dependencies and Setup

Import necessary libraries:

use std::sync::{Arc, RwLock};  
use futures\_util::{sink::SinkExt, stream::StreamExt};  
use tokio::time::{sleep, Duration};  
use tokio\_tungstenite::{connect\_async, tungstenite::protocol::Message};  
use url::Url;

websocket\_client Function

Establishes a WebSocket connection and manages messages:

async fn websocket\_client(topic\_url: &str) {  
 // The URL of the WebSocket server to connect to  
 let url = Url::parse(topic\_url).expect("Invalid URL");  
  
 // Connect to the WebSocket server  
 let (ws\_stream, \_) = connect\_async(url).await.expect("Failed to connect");  
  
 println!("WebSocket client connected");  
  
 let (mut write, mut read) = ws\_stream.split();  
 let message = Arc::new(RwLock::new(String::new()));  
 let message\_1 = message.clone();  
 // Spawn a task to handle incoming messages  
 tokio::spawn(async move {  
 let msg\_lock = message\_1.clone();  
 while let Some(message) = read.next().await {  
 match message {  
 Ok(msg) => {  
 let mut ms = msg\_lock.write().unwrap();  
 \*ms = msg.to\_text().unwrap().to\_string();  
 println!("Received message: {}", msg.to\_text().unwrap());  
 }  
 Err(e) => {  
 eprintln!("Error receiving message: {:?}", e);  
 break;  
 }  
 }  
 }  
 });  
  
 // Continuously send messages  
 loop {  
 let msg\_lock = message.clone();  
 let ms = msg\_lock.read().unwrap().clone();  
 if let Err(e) = write.send(Message::Text(ms)).await {  
 eprintln!("Error sending message: {:?}", e);  
 break;  
 }  
 sleep(Duration::from\_secs(5)).await;  
 }  
}

* Connects to the message broker’s WebSocket server.
* Reads and sends messages over the WebSocket.

**Running the Client**

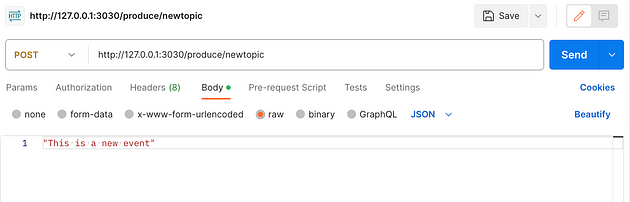
Demonstrate how to run the client:

#[tokio::main]  
async fn main() {  
 websocket\_client("ws://127.0.0.1:3030/subscribe/newtopic").await;  
}

**Steps for Demo :**

1. run the broker server first using cargo run.
2. Now run the web client it will try to subscribe newtopic to the broker.
3. now we can produce the newtopic from the broker using post web url e.g.

http://127.0.0.1:3030/produce/newtopic   
//with message body under raw json  
"This is a new event of new topic"



Produce Topic

**Conclusion**

We’ve explored creating a simple message broker in Rust and testing it with a WebSocket client. This example highlights Rust’s capabilities in building efficient, concurrent network applications.

**Further Exploration**

Encourage readers to expand upon this foundation, experimenting with different topics, messages, and extending the broker’s functionality.

This article inspired from —

<https://blog.devgenius.io/building-an-event-broker-in-rust-0748dbfb9346>

<https://kafka.apache.org/>