Advanced Rust - Lab 7: Async Programming

Lukáš Hozda

2026

Exercise 1: Task Spawning and Channels

Objective

Build a parallel work distribution system using async tasks and channels.

Instructions

Implement a system that:

1. Spawns multiple worker tasks
2. Distributes work items to workers via a channel
3. Collects results from workers via another channel
4. Processes all results

Requirements

```rust
use tokio::sync::mpsc;

/// Represents a work item to process
#[derive(Debug)]
struct WorkItem {
    id: u32,
    data: String,
}

/// Represents the result of processing a work item
#[derive(Debug)]
struct WorkResult {
    id: u32,
    processed: String,
}

/// Simulates processing a work item
/// Keep like this or do something funny that takes time
async fn process_item(item: WorkItem) -> WorkResult {
    // simulate some async work
    tokio::time::sleep(tokio::time::Duration::from_millis(100)).await;
    WorkResult {
        id: item.id,
        processed: item.data.to_uppercase(),
    }
}
```

```rust
/// Spawns worker tasks that receive work items and send back results
async fn spawn_workers(
    num_workers: usize,
    work_rx: mpsc::Receiver<WorkItem>,
    result_tx: mpsc::Sender<WorkResult>,
) {
    // - convert work_rx into a shared receiver (hint: Arc<Mutex< ... >>)
    // - spawn num_workers tasks
    // - each worker should loop, receiving items and sending results
    // - workers should exit when the channel closes
}

/// Distributes work items to workers
async fn distribute_work(work_tx: mpsc::Sender<WorkItem>, items: Vec<WorkItem>) {
    // - send each item through the channel
}

/// Collects all results from workers
async fn collect_results(mut result_rx: mpsc::Receiver<WorkResult>) -> Vec<WorkResult> {
    // - receive all results until the channel closes
    // - return them as a vector
}

#[tokio::main]
async fn main() {
    let items: Vec<WorkItem> = (0..10)
        .map(|i| WorkItem {
            id: i,
            data: format!("item_{}", i),
        })
        .collect();

    let num_workers = 3;
    let (work_tx, work_rx) = mpsc::channel(32);
    let (result_tx, result_rx) = mpsc::channel(32);

    // spawn the worker pool
    let workers = tokio::spawn(spawn_workers(num_workers, work_rx, result_tx));

    // distribute work
    distribute_work(work_tx, items).await;

    // collect results
    let results = collect_results(result_rx).await;

    // wait for workers to finish
    workers.await.unwrap();

    println!("Processed {} items:", results.len());
    for result in &results {
        println!("  {} -> {}", result.id, result.processed);
    }
```

```
    assert_eq!(results.len(), 10);
}
```

Exercise 2: Select and Timeout Patterns

Objective

Implement a request handler with timeout and cancellation support using select!.

Instructions

Implement a system that:

1. Processes requests with a configurable timeout
2. Supports graceful shutdown via a cancellation signal
3. Handles multiple concurrent operations with select!

Requirements

```rust
use std::time::Duration;
use tokio::sync::{mpsc, oneshot};
use tokio::time::timeout;

/// A request to be processed
#[derive(Debug)]
struct Request {
    id: u32,
    payload: String,
    /// channel to send the response back
    response_tx: oneshot::Sender<Response>,
}

/// Response to a request
#[derive(Debug)]
struct Response {
    id: u32,
    result: Result<String, RequestError>,
}

#[derive(Debug)]
enum RequestError {
    Timeout,
    Cancelled,
    ProcessingFailed(String),
}

/// Simulates processing a request (don't modify)
async fn do_processing(payload: &str) -> Result<String, String> {
    // simulate variable processing time
    let delay = (payload.len() * 50) as u64;
    tokio::time::sleep(Duration::from_millis(delay)).await;

    if payload.contains("fail") {
        Err("processing failed".to_string())
```

```
    } else {
        Ok(format!("processed: {}", payload))
    }
}

/// Processes a single request with timeout
async fn process_with_timeout(
    request: Request,
    timeout_duration: Duration,
) {
    // - use tokio::time::timeout to wrap do_processing
    // - send appropriate Response back through request.response_tx
    // - handle timeout and processing errors
}

/// Runs the request processing loop with shutdown support
async fn run_processor(
    mut request_rx: mpsc::Receiver<Request>,
    mut shutdown_rx: oneshot::Receiver<()>,
    timeout_duration: Duration,
) {
    // - process incoming requests
    // - stop when shutdown signal received
    // - handle channel closure gracefully
}

/// Sends a request and waits for response
async fn send_request(
    request_tx: &mpsc::Sender<Request>,
    id: u32,
    payload: String,
) -> Response {
    // - create a oneshot channel for the response
    // - send the request
    // - wait for and return the response
}

#[tokio::main]
async fn main() {
    let (request_tx, request_rx) = mpsc::channel(32);
    let (shutdown_tx, shutdown_rx) = oneshot::channel();

    let timeout_duration = Duration::from_millis(600);

    // spawn the processor
    let processor = tokio::spawn(run_processor(request_rx, shutdown_rx, timeout_duration));

    // send some requests
    let payloads = vec!["hi", "hello world", "fail please", "aaaaaaaaaaaaaaaaaaaaa"];

    let mut responses = Vec::new();
    for (i, payload) in payloads.iter().enumerate() {
        let response = send_request(&request_tx, i as u32, payload.to_string()).await;
        println!("Request {}: {:?}", i, response.result);
```

```
        responses.push(response);
    }

    // shutdown
    drop(request_tx);
    let _ = shutdown_tx.send(());
    processor.await.unwrap();

    // verify results
    assert!(responses[0].result.is_ok()); // "hi" - fast, succeeds
    assert!(responses[1].result.is_ok()); // "hello world" - medium, succeeds
    assert!(matches!(
        responses[2].result,
        Err(RequestError::ProcessingFailed(_))
    )); // contains "fail"
    assert!(matches!(responses[3].result, Err(RequestError::Timeout))); // too long
}
```

Exercise 3: Async TCP Chat Server

Objective

Build a simple TCP chat server that handles multiple clients concurrently.

Instructions

Implement a chat server that:

1. Accepts multiple TCP connections
2. Broadcasts messages from any client to all connected clients
3. Handles client disconnections gracefully
4. Uses async I/O for all operations

Requirements

```
use std::collections::HashMap;
use std::sync::Arc;
use tokio::io::{AsyncBufReadExt, AsyncWriteExt, BufReader};
use tokio::net::{TcpListener, TcpStream};
use tokio::sync::{broadcast, Mutex};

type ClientId = u32;

/// Shared state for the chat server
struct ChatServer {
    /// broadcast channel for messages
    broadcast_tx: broadcast::Sender<ChatMessage>,
    /// connected clients (id -> username)
    clients: Mutex<HashMap<ClientId, String>>,
    /// counter for generating client ids
    next_id: Mutex<ClientId>,
}

#[derive(Clone, Debug)]
```

```rust
struct ChatMessage {
    from: String,
    content: String,
}

impl ChatServer {
    fn new() -> Arc<Self> {
        let (broadcast_tx, _) = broadcast::channel(100);
        Arc::new(Self {
            broadcast_tx,
            clients: Mutex::new(HashMap::new()),
            next_id: Mutex::new(0),
        })
    }

    /// Registers a new client and returns their id
    async fn register_client(&self, username: String) -> ClientId {
        // TODO: implement this function
        // - generate a new client id
        // - store the username
        // - return the id
    }

    /// Removes a client from the server
    async fn remove_client(&self, id: ClientId) {
        // TODO: implement this function
    }

    /// Broadcasts a message to all clients
    fn broadcast(&self, message: ChatMessage) {
        // TODO: implement this function
        // - send message through broadcast channel
        // - ignore errors (no receivers is fine)
    }

    /// Returns a receiver for broadcast messages
    fn subscribe(&self) -> broadcast::Receiver<ChatMessage> {
        self.broadcast_tx.subscribe()
    }
}

/// Handles a single client connection
async fn handle_client(server: Arc<ChatServer>, stream: TcpStream) {
    // - split the stream into reader and writer
    // - read the first line as the username
    // - register the client
    // - announce that they joined
    // - spawn a task to forward broadcast messages to this client
    // - read lines from the client and broadcast them
    // - handle disconnection (remove client, announce departure)
    //
    // hints:
    // - use BufReader for line-based reading
    // - use stream.into_split() to get separate read/write halves
```

```
    // - use tokio::select! to handle reading and shutdown
}

/// Runs the chat server
async fn run_server(addr: &str) -> std::io::Result<()> {
    // - bind to the address
    // - create the ChatServer
    // - accept connections in a loop
    // - spawn a task for each connection
}

#[tokio::main]
async fn main() -> std::io::Result<()> {
    println!("Starting chat server on 127.0.0.1:8080");
    println!("Connect with: nc 127.0.0.1 8080");
    println!("First line you send will be your username");

    run_server("127.0.0.1:8080").await
}
```

Client

Implement a chat client that connects to the server. Place this in `src/bin/client.rs`.

```
use tokio::io::{AsyncBufReadExt, AsyncWriteExt, BufReader};
use tokio::net::TcpStream;

/// Reads lines from stdin and sends them to the server
async fn send_messages(mut writer: tokio::net::tcp::OwnedWriteHalf) {
    // - read lines from stdin using BufReader
    // - send each line to the server
    // - exit when stdin closes
    //
    // hint: use tokio::io::stdin() for async stdin
}

/// Receives messages from the server and prints them
async fn receive_messages(reader: tokio::net::tcp::OwnedReadHalf) {
    // - read lines from the server
    // - print each line to stdout
    // - exit when connection closes
}

#[tokio::main]
async fn main() -> std::io::Result<()> {
    let addr = std::env::args()
        .nth(1)
        .unwrap_or_else(|| "127.0.0.1:8080".to_string());

    let username = std::env::args()
        .nth(2)
        .unwrap_or_else(|| "anonymous".to_string());

    // - connect to the server
    // - send the username as the first line
```

```
    // - split the stream into reader and writer
    // - spawn tasks for sending and receiving
    // - wait for either task to finish (use select!)

    Ok(())
}
```

To test, run the server and client in separate terminals:

```
# terminal 1: run the server
cargo run

# terminal 2: run a client
cargo run --bin client -- 127.0.0.1:8080 alice

# terminal 3: run another client
cargo run --bin client -- 127.0.0.1:8080 bob
```