Roll NO: IOT048332

**Assignment 2**

**ASYNCHRONOUS**

Rust Async is the new hot thing Rust’s land. It has hailed as a big milestone for Rust; especially for people developing highly performant-networking applications. The long time for development, the different incompatible versions, and the various libraries; however, might made it not very straightforward to grasp. There is a lot going and it is not obvious from where to start.

A telecommunication, asynchronous communication is transmission of data, generally without the use of an external clock signal, where data can be transmitted intermittently rather than in a steady stream. Encode any timing required to recover data from the communication symbols is within the symbols.

Any function can create a thread .A function that uses threads is usually just as easy as calling any normal function.

However, asynchronous functions require special support from the language or libraries

In Rust, async fun creates an asynchronous function, which returns a Future. The increased complexity of the asynchronous programming model is not always worth it. Asynchronous Rust ecosystem has undergone a lot of evolution over time. So, it can be hard to know what tools to use, what libraries to invest in, or what documentation to read. However, the Future trait inside the standard library and the async/await Lang feature has recently been stabilized.

Now, however, the ecosystem is still undergoing rapid development and the asynchronous Rust experience is unpolished. The async/await language feature is still new

Important extensions like async fun syntax in trait methods are still unimplemented, and the current compiler error messages can be difficult to parse. Async/.await is Rust’s built-in tools for writing asynchronous function that look like synchronous code.

The value returned by async fun is a Future. Future needs to be run on an executor, so that a task may be done. We can also use .await instead of block on inside async fun .await does not block the whole thread but wait for the specific Future. Asynchronous code allows us to run multiple tasks concurrently on the same OS thread. In a typical threaded application, if you wanted to download two different webpages at the same time, you would spread the work across two different threads

Hyper is an asynchronous HTTP library. We will use it to power our HTTP server and to make HTTP request. Enter async/await. Fast forward a year or two, and more and more people are realizing that while combinatory-based Futures are extremely powerful, they are not always that fun to write. They often result in deeply nested callbacks, which is not the ordinary way people write code in Rust. It also is not possible to hold a reference across callbacks, which leads to very awkward workarounds. Async/await, which had been considered from the beginning, was now a high priority.

Our example from earlier could be written like this using async/await:

* let id = id\_rpc(&my\_server).await;
* let row = get\_row(id).await;
* let encoded = json::encode(row);
* write\_string(my\_socket, encoded).await;

Even in this simple example, the code is easier to read! Many people who do not mind using unstable features on the nightly compiler decided to start using async/await directly, because of all the benefits that came with i

**DIAGRAM**



