DORA-RS

On Async Runtime

github.com/dora-rs/dora

Xavier Tao

TOKIO RUNTIME

Very powerful async engine that is able to run green threads, running highly-conccurrent tasks!

```rust
#[tokio::main]
async fn main() -> eyre::Result<()> {
```

ASYNC DESIGN: EVENT STREAM

```
enum Event {
    Input(InputEvent),
    InputsStopped,
    Operator {
        id: OperatorId,
        event: OperatorEvent,
    },
}
```

ASYNC DESIGN: EVENT STREAM

```rust
enum Event {
    Input(InputEvent),
    InputsStopped,
    Operator {
        id: OperatorId,
        event: OperatorEvent,
    },
}
```

```rust
let inputs = subscribe(&topics, communication).await?;
let input_events = inputs.map(Event::Input)
    .chain(stream::once(async { Event::InputsStopped }));
```

ASYNC DESIGN: EVENT STREAM

```rust
enum Event {
    Input(InputEvent),
    InputsStopped,
    Operator {
        id: OperatorId,
        event: OperatorEvent,
    },
}
```

```rust
let inputs = subscribe(&topics, communication).await?;
let input_events = inputs.map(Event::Input)
    .chain(stream::once(async { Event::InputsStopped }));
```

```rust
let mut operator_events = tokio_stream::StreamMap::new();
for operator in operators {
    operator_events.insert(operator.id, spawn_operator().await);
}
let op_events = operator_events.map(|(id, event)| Event::Operator { id, event });
```

ASYNC DESIGN: EVENT STREAM

```
enum Event {
    Input(InputEvent),
    InputsStopped,
    Operator {
        id: OperatorId,
        event: OperatorEvent,
    },
}
```

```
let inputs = subscribe(&topics, communication).await?;
let input_events = inputs.map(Event::Input)
    .chain(stream::once(async { Event::InputsStopped }));
```

```
let mut operator_events = tokio_stream::StreamMap::new();
for operator in operators {
    operator_events.insert(operator.id, spawn_operator().await);
}
let op_events = operator_events.map(|(id, event)| Event::Operator { id, event });
```

```
use futures_concurrency::Merge;
let mut events = (input_events, op_events).merge();

// now we have a normal while loop with a match instead of needing `select`
while let Some(event) = events.next().await {
    match event { ... }
}
```

CHALLENGES WITH ASYNC RUST

- CPU-bound operations in async functions
- Cancellation of spawned tasks
- Error/panic propagation

CPU-BOUND OPERATIONS IN ASYNC FUNCTIONS

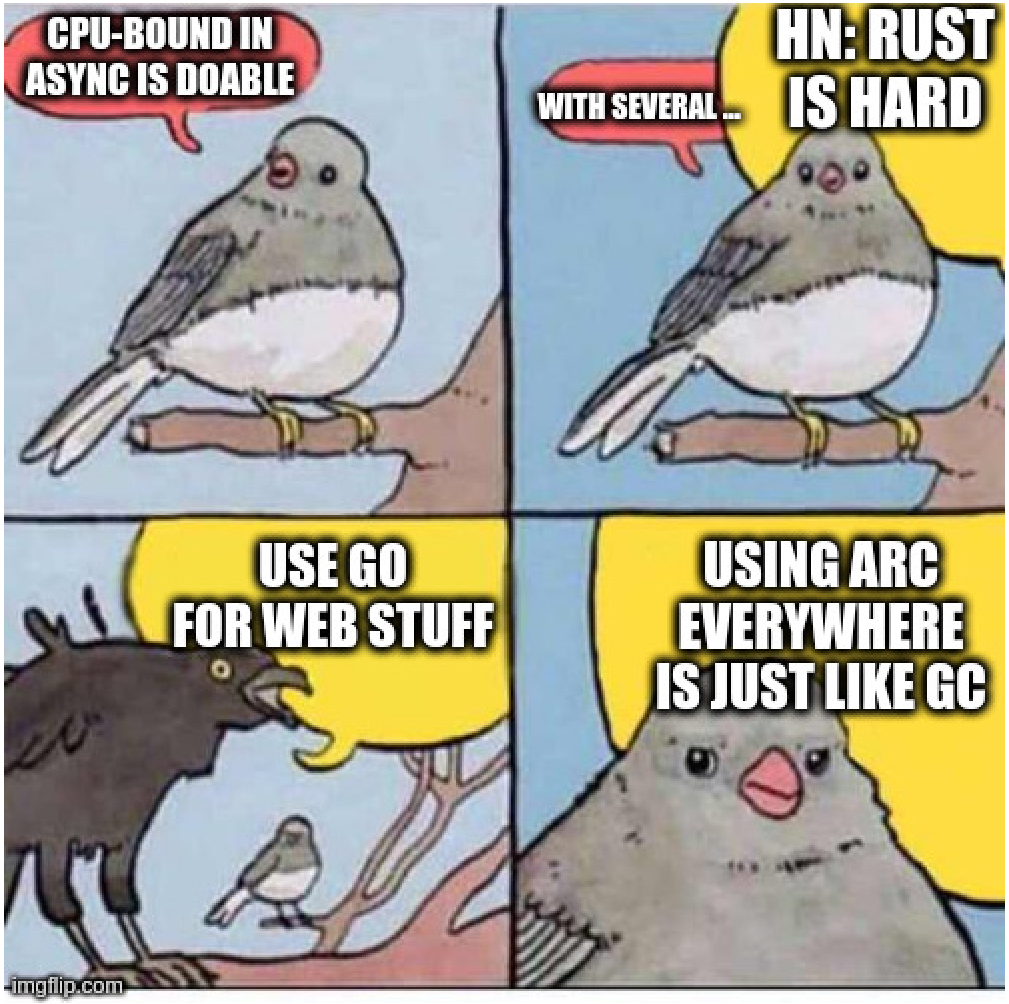
> *To give a sense of scale of how much time is too much, a good rule of thumb is* ***no more than 10 to 100 microseconds*** *between each `.await`.*

	CPU-bound computation	**Synchronous IO**	**Running forever**
`spawn_blocking`	Suboptimal	OK	No
`rayon`	OK	No	No
Dedicated thread	OK	OK	OK

— Alice Ryhl, in *“Async: What is blocking?”*

SPAWNING CPU-BOUND TASKS

- **tokio::spawn_blocking**: Optimal for reducing tail latency
    - Pros
        - Directly available threadpool
        - Work well enough on most cases
    - Cons
        - Need adequate parametrisation (workers, threads, timeout…)
- **rayon**: Optimal for maximizing cpu usage
    - Pros
        - Do not need Arc for ∞ read
        - Stack allocation of tasks instead of heap
        - Recursive parallelisation
    - Cons
        - Add a layer of technology


CPU-BOUND IN ASYNC IS DOABLE
WITH SEVERAL ...
HN: RUST IS HARD
USE GO FOR WEB STUFF
USING ARC EVERYWHERE IS JUST LIKE GC
imgflip.com

CANCELLATION OF SPAWNED TASKS

- Dropping a `JoinHandle` detaches the task
  → keeps running in the background

```
let task = tokio::spawn(async {
    let mut items = Vec::new();
    for _ in 0..n {
        items.push(read_large_file().await);
    }
    let _ = result_tx.send(items).await;
});
do_something_else()?; // ϟ `task` keeps running on error
let items = task.await;
```

CANCELLATION OF SPAWNED TASKS

- Dropping a `JoinHandle` detaches the task
  → keeps running in the background

```rust
let task = tokio::spawn(async {
    let mut items = Vec::new();
    for _ in 0..n {
        items.push(read_large_file().await);
    }
    let _ = result_tx.send(items).await;
});
do_something_else()?; // ϟ `task` keeps running on error
let items = task.await;
```

- Workarounds
  - manual checks (e.g. is `result_tx` closed?)
    → error-prone and less efficient
  - use `smol` instead of `tokio` or `async_std`
  - use `FutureExt:remote_handle` everywhere

```rust
let (task, handle) = task.remote_handle();
tokio::spawn(task);
do_something_else()?; // dropping `handle` cancels task
handle.await
```

ERROR/PANIC PROPAGATION

- Easy to accidentally discard a panic/error

```
tokio::spawn(async { panic!("foo") });
tokio::spawn(async { Result::<(), u32>::Err(1) });
```

  - neither a compiler warning nor a runtime error occurs (just some `stderr` output for the panic)
  - same with `async_std`
  - `must_use` warning with `smol` (tasks are canceled on drop)

ASYNC ANNOYANCES

- Compiler cannot infer error type in `async` blocks

```
let task = async {
    value.context("I/O error")?;
    Ok(())
 // ^^ cannot infer type for type parameter `E`
 // workaround: `Result::<_, anyhow::Error>::Ok(())`
};
```

ASYNC ANNOYANCES

- Compiler cannot infer error type in `async` blocks

```rust
let task = async {
    value.context("I/O error")?;
    Ok(())
 // ^^ cannot infer type for type parameter `E`
 // workaround: `Result::<_, anyhow::Error>::Ok(())`
};
```

- Clone boilerplate with `async move {}`

```rust
let id_clone = id.clone();
let tx_clone = tx.clone();
// etc ...
let task = async move {
    tx_clone.send(id_clone).await
};
// (we still need `id`, `tx`, etc. here)
```

(also applies to `move` closures)

DON’T USE `select` MACRO

- requires lots of pinning and fusing
- its custom syntax breaks tools such as `rust-analyzer` or `rustfmt`
- `FusedFuture` bound does not prevent poll after `Poll::Ready`

```
loop {
    let mut fut = (&mut fut).fuse(); // ⚡ (should be outside of loop)
    select! {
        _ = fut => println!("foo"),
    };
}
```

DON’T USE `select` MACRO

- requires lots of pinning and fusing
- its custom syntax breaks tools such as `rust-analyzer` or `rustfmt`
- `FusedFuture` bound does not prevent poll after `Poll::Ready`

```
loop {
    let mut fut = (&mut fut).fuse(); // ϟ (should be outside of loop)
    select! {
        _ = fut => println!("foo"),
    };
}
```

- Use an event stream instead
  - as proposed by Yoshua Wuyts in *“Futures Concurrency III”*

Q&A