Generic Application-Server Crates

Built With Async Rust

Brendan Nolan

August 22, 2023

Goal

- Provide an *application-server* crate and a *client* crate, allowing users to request jobs which are too heavy for their local machines.
- Example use case: scientists in a wetlab, who must regularly run heavy models from their bench laptops.

Goal

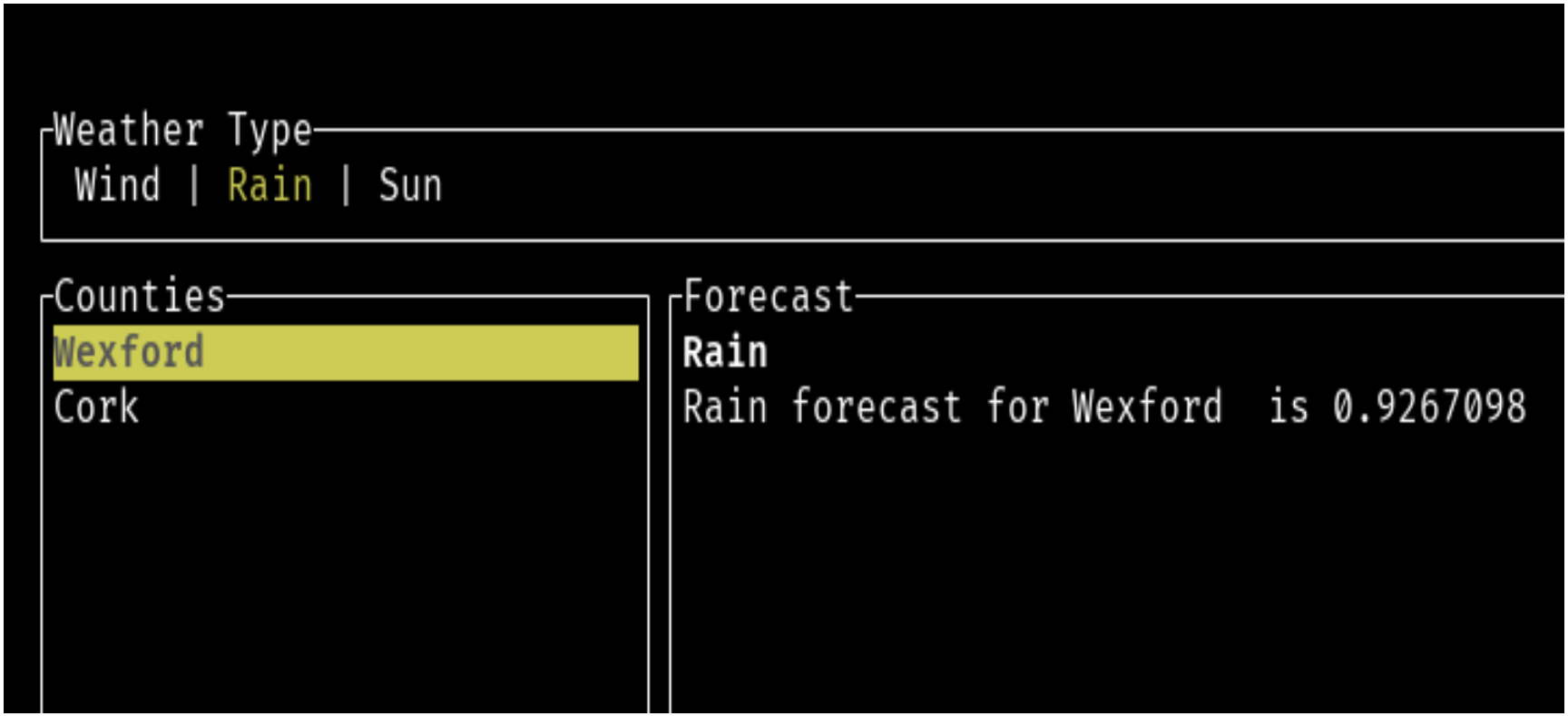
- Provide an *application-server* crate and a *client* crate, allowing users to request jobs which are too heavy for their local machines.
- Example use case: scientists in a wetlab, who must regularly run heavy models from their bench laptops.

How A User Creates A Custom Server

A user who wants a server that provides weather status must only do the following:

```
let (tx_shutdown, rx_shutdown) = watch::channel(());
let server_task = tokio::spawn(server_runner::run_server(
    "<IP Address>:<Port>",
    weather_provider,  // User's type - must implement RequestProcessor
    ShutdownListener::new(rx_shutdown),
));
```

Example Application



Must-Have Features

For crate users

- Easily customise server data processing, e.g.:
  - train ML algorithms
  - run planetary models
  - run structural analysis of timberframe houses
- Allow client to choose between
  - call-and-response model
  - sending and receiving in any order
- Remain responsive as number of clients connecting to a single server grows

Must-Have Features

For crate users

- Easily customise server data processing, e.g.:
  - train ML algorithms
  - run planetary models
  - run structural analysis of timberframe houses
- Allow client to choose between
  - call-and-response model
  - sending and receiving in any order
- Remain responsive as number of clients connecting to a single server grows

Must-Have Features

For crate users

- Easily customise server data processing, e.g.:
  - train ML algorithms
  - run planetary models
  - run structural analysis of timberframe houses
- Allow client to choose between
  - call-and-response model
  - sending and receiving in any order
- Remain responsive as number of clients connecting to a single server grows

Must-Have Features

For crate developers

- Do not waste resources by blocking threads while waiting for I/O
- (More insidious than the above) do not make asynchronous work wait for other asynchronous work unnecessarily

Must-Have Features

For crate developers

- Do not waste resources by blocking threads while waiting for I/O
- (More insidious than the above) do not make asynchronous work wait for other asynchronous work unnecessarily

Technology Choices

Criteria for chosen technology

- Has well documented library(s) for async I/O
- Eliminates as many bugs as possible as early as possible (strong bias towards compiled languages)

Technology Choices

Criteria for chosen technology

- Has well documented library(s) for async I/O
- Eliminates as many bugs as possible as early as possible (strong bias towards compiled languages)

C++: *boost.asio*

- Pros
  - *boost.asio* is a mature library
- Cons
  - awkward call-back mechanism
  - many object-lifetime footguns (use-after-free bugs)
  - static polymoprphism extremely awkward to use

C++: *boost.asio*

- Pros
  - *boost.asio* is a mature library
- Cons
  - awkward call-back mechanism
  - many object-lifetime footguns (use-after-free bugs)
  - static polymoprphism extremely awkward to use

Rust: *tokio*

- Cons
  - async I/O ecosystem in *Rust* is still young
- Pros
  - *async/await* syntax is far more ergonomic than passing callbacks around
  - typesystem eliminates use-after-free bugs
  - static polymorphism is user friendly

Rust: *tokio*

- Cons
  - async I/O ecosystem in *Rust* is still young
- Pros
  - *async*/*await* syntax is far more ergonomic than passing callbacks around
  - typesystem eliminates use-after-free bugs
  - static polymorphism is user friendly

Winner: *Rust* with *tokio*

Design Decisions

- Customization point should allow state (e.g. in order to cache results of previous model runs)

```
pub trait RequestProcessor<Req, Resp> {
    fn process(&self, request: &Req) -> Resp;
}
```

Design Decisions

- Support serialization by *serde* only
- Make three crates (for client, server, and I/O utilities) rather than combining all into one crate

Design Decisions

- Support serialization by *serde* only
- Make three crates (for client, server, and I/O utilities) rather than combining all into one crate

Design Decisions

- Follow the Golang philosophy: "Do not communicate by sharing memory - share memory by communicating"
- Employ the *actor* pattern, recommended by Alice Ryhl (tokio maintainer)
- Make it as simple as possible to set up a custom server

Design Decisions

- Follow the Golang philosophy: "Do not communicate by sharing memory - share memory by communicating"
- Employ the *actor* pattern, recommended by Alice Ryhl (tokio maintainer)
- Make it as simple as possible to set up a custom server

Design Decisions

- Follow the Golang philosophy: "Do not communicate by sharing memory - share memory by communicating"
- Employ the *actor* pattern, recommended by Alice Ryhl (tokio maintainer)
- Make it as simple as possible to set up a custom server

My Implementation Of The Actor Pattern

(Some generic parameters and trait bounds removed for readability)

```
pub struct Command<Req, Resp> {
    pub data: Req,
    pub responder: oneshot::Sender<Resp>,
}

// The "Handle"
pub struct JobDispatcher<Req, Resp> {
  // Receiver owned by "Actor" (jobs task)
  tx: mpsc::Sender<Command<Req, Resp>>,
}

impl Clone for JobDispatcher<Req, Resp> { ... }

impl JobDispatcher<Req, Resp> {
    pub async fn dispatch_job(&self, data: Req) -> oneshot::Receiver<Resp> {
      // Create command from data
      // Send command to jobs task
      // Return receiver from which job result can be collected
    }
}
```

To-Do List

server crate:

- Implement unified and ergonomic way to handle shutdown signals
- Add organised logging (with support for various levels of verbosity)
- Consider compatability of serialized data across architectures, versions of this crate, versions of dependencies (look into protobuf). (May be difficult because input/output data is generic by design.)
- Think carefully about backpressure
- Ensure there are no partial-read vulnerabilities
- Investigate automatically figuring out channel bounds, rather than asking the user to supply them
- Automated testing

To-Do List

server crate:

- Implement unified and ergonomic way to handle shutdown signals
- Add organised logging (with support for various levels of verbosity)
- Consider compatability of serialized data across architectures, versions of this crate, versions of dependencies (look into protobuf). (May be difficult because input/output data is generic by design.)
- Think carefully about backpressure
- Ensure there are no partial-read vulnerabilities
- Investigate automatically figuring out channel bounds, rather than asking the user to supply them
- Automated testing

To-Do List

server crate:

- Implement unified and ergonomic way to handle shutdown signals
- Add organised logging (with support for various levels of verbosity)
- Consider compatability of serialized data across architectures, versions of this crate, versions of dependencies (look into protobuf). (May be difficult because input/output data is generic by design.)
- Think carefully about backpressure
- Ensure there are no partial-read vulnerabilities
- Investigate automatically figuring out channel bounds, rather than asking the user to supply them
- Automated testing

To-Do List

server crate:

- Implement unified and ergonomic way to handle shutdown signals
- Add organised logging (with support for various levels of verbosity)
- Consider compatability of serialized data across architectures, versions of this crate, versions of dependencies (look into protobuf). (May be difficult because input/output data is generic by design.)
- Think carefully about backpressure
- Ensure there are no partial-read vulnerabilities
- Investigate automatically figuring out channel bounds, rather than asking the user to supply them
- Automated testing

To-Do List

server crate:

- Implement unified and ergonomic way to handle shutdown signals
- Add organised logging (with support for various levels of verbosity)
- Consider compatability of serialized data across architectures, versions of this crate, versions of dependencies (look into protobuf). (May be difficult because input/output data is generic by design.)
- Think carefully about backpressure
- Ensure there are no partial-read vulnerabilities
- Investigate automatically figuring out channel bounds, rather than asking the user to supply them
- Automated testing

To-Do List

server crate:

- Implement unified and ergonomic way to handle shutdown signals
- Add organised logging (with support for various levels of verbosity)
- Consider compatability of serialized data across architectures, versions of this crate, versions of dependencies (look into protobuf). (May be difficult because input/output data is generic by design.)
- Think carefully about backpressure
- Ensure there are no partial-read vulnerabilities
- Investigate automatically figuring out channel bounds, rather than asking the user to supply them
- Automated testing

To-Do List

server crate:

- Implement unified and ergonomic way to handle shutdown signals
- Add organised logging (with support for various levels of verbosity)
- Consider compatability of serialized data across architectures, versions of this crate, versions of dependencies (look into protobuf). (May be difficult because input/output data is generic by design.)
- Think carefully about backpressure
- Ensure there are no partial-read vulnerabilities
- Investigate automatically figuring out channel bounds, rather than asking the user to supply them
- Automated testing

To-Do List

client crate:

- Not I/O bound - does it need async? Would it be enough to have a read thread and a write thread?