Introduction

In this paper I'll detail my design of channel based communication between generic backends. I'll also discuss an alternative approach: using an entity component system to drive parallel agent systems. Both systems have their strengths and weaknesses, but this paper is meant to provide context to both approaches. Protoyping of the channel based design, along with this text and its accompanying diagram can be found here.

Threaded Channel Based Communication

Messages, channels, and handlers

At its core, all we're doing is passing messages between agents, having them perform work based on those message, and sending messages out to other agents, so we'd better do it well.

The first of two approaches for structuring messages is simple raw messages bound by a type constraint that dictates how that message should be interpreted. Message variants are of the kind query, read, or NOP. A query variant acts as an input to an agent, additionally notifying the agent that it must output a response to its siblings. A Read variant also acts as an input to an agent, but will not result in creation of a new message. A NOP messages signals to an agent that no more work is to be done. We aren't writing assembly, so why do we have a NOP? Our agents listen for messages on green threads, which while light, take up cpu resources. We can reduce this load using condition variables or primatives built into Tokio that don't periodically poll, but we'd still like to join these threads when we no an agent is no longer required for a system to function. This is when we'd send a NOP instruction, telling an agent to shut down.

The second approach is structured data, but not data that we inherently structure. By providing a Message trait to consumers of the library, we allow for a more granular approach to system specific messaging. Data looks a lot different in a chat application than it does in a chess application. By

The secret third option is blending the previous options. In order to maintain a high level of precision and performance, we need internal mechanisms for agent lifetimes and action signals, but that doesn't mean our end users can't also have flexibility with their data. Imagine this implementation:

```
trait MessageBody {
    type output;
}

enum MessageKind {
    Query(Arc<dyn MessageBody>),
    Read(Arc<dyn MessageBody>),
    Nop,
}
```

Now that we have messages, we need a medium to distribute them. Channels are a great choice for safely passing data between agent threads. In the standard library we have an unbound multi-produces single-consumer channel that will likely work for our use cases, but incase they aren't flexible enough, the crossbeam crate provides a multi-producer multi-consumer channel, in addition to a good deal of other concurency tools like thread parking and shared Read Write Locks. When an agent is constructed, an entangled message sender and reciever are produced alongside of it. The Sender is a smart pointer to the inner buffer write mechanics and can be cheaply cloned, when being assigned to other agents. In brief, agents have a message reciever, an origin copy of their sender, and a collection of senders from other agents, to which they can write. This is how agents communicate: an agent waits until it receives a message on its receive channel, does some work, and then dispatches a message on one or more of the channels in its sibling sender collection.

Backends and Entities

Messages constitute the majority of the system's complexity in the system and now that we've explained them we should discuss the agents themselves. I've chosen to make agents generic, in that they don't necessarily have to be language models themselves. There are so many other potential components in a given system, like human users, databases, chrono apis, search engines, etc. Shoe-horning llm backends into their own category and writing a bunch of glue code to make everything homogenous seems like an easily quelled nightmare. The solution is to treat every component as it's own entity with the same communication capabilities accross the board. We achieve this by providing a Backend trait with methods acting on our message variants. Query takes a message body and an iterator of send channels, read takes only the message body, and nop acts as an inplace implementation of the drop trait. As we've discussed, an agent thread waits for a message to be recieved, but now we can see that upon recieving a message, the agent has the methods implemented to handle those variants.

In the case of any standard LLM, it's easy to conceptualize about how you'd implement these methods, but what about a user backend? This is where we might implement our functionallity for displaying messages in a chat application. Upon recieving a read message, we simply write the contents to stdout, query might read from stdin and dispatch the buffer to another agent, and nop can smoothly handle disconnecting the user client. It makes sense to standardize some of these n-category backends alongside wrappers around llm backends, but we should also re-export the Backend trait to allow users of the library to implement their own backends.

Entities represent instances of backends within our system. They are assigned a unique identifier upon initialization and hold atomic reference counters to some dynamic backend polymorph. Weak referencing is a more or less the standard way of holding handles to entities in systems with complex relationships like this. It also gives us the added advantage of swapping out an agent's entity at runtime without risking the deallocation of the initial entity. This is useful for scenarios where a system has a long and expensive lifetime that we want to run as optimally as it can. We can perform real time analytics on entity performance and interpolate between more performant nodes within our system, for a given relationship.

Agent Systems

Now that we have a solid understanding of how these primitive components work, let's discuss how these components come together in the context of agent systems. An agent system is essentially a collection of agents that collaborate to achieve a specific goal or perform a set of tasks. In our design, agents communicate through channels and process messages, allowing for the coordination of complex behaviors.

Agents can be configured to specialize in certain tasks or domains, making the system modular and extensible. For example, language models excel at processing natural language queries, while a database agent efficiently handles data storage and retrieval. By combining these specialized agents, we create an agent system capable of handling a wide range of scenarios.

In a sophisticated system, particularly when dealing with a multitude of interconnected components, managing multiple send channels efficiently becomes crucial. Consider a scenario where agents in a system end up in a dependency cycle. We have the primitive variants to avoid a death march, by simply dispatching a read.

Entity Component System

We've been talking a lot about entities, components, and systems, so maybe we should talk about ECS (entity component systems). An ECS is a pattern commonly used in game development, but it finds relevance in many places and potentially ours. In essence, an ECS breaks down a complex system into atomic instances or entities, components, data bound to entities, and systems that filter entities based on component queries, mutating and/or acting on those entities.

In the context of our project we can think of these parts as such:

- **Entities:** As we've discussed, entities represent instances of backends within our system. They encapsulate the unique identity and handle referencing between different components.
- **Components:** Components are the building blocks of functionality. Each aspect or capability of a backend can be considered a component. For example, a language model backend might have components for natural language processing, sentiment analysis, etc.
- **Systems:** Systems orchestrate the interactions between entities and components. They define how components should operate within the broader context of the system.

By adopting an ECS approach, we achieve extremely modularity and scalable architecture. Components can be added, removed, or replaced independently, allowing for easy adaptation to changing requirements.