

Distributed Actor System in Rust

Zimon Dai

About me

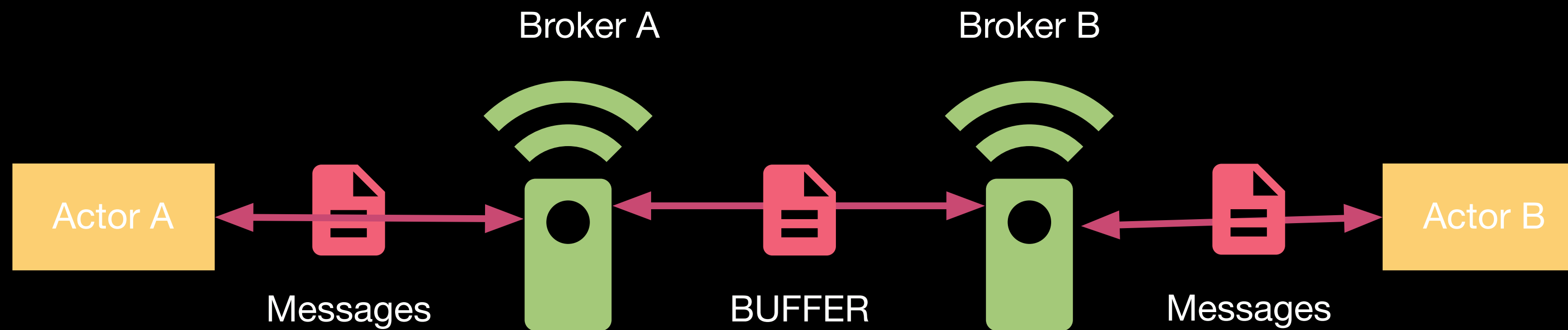
- Zimon Dai
- Senior Engineer @ Alibaba Inc
- Full-time Rust dev since 2016
- daizhuoxian@outlook.com

This talk is *not* about

- Details of full featured actor system
- Comparison with other popular actor systems (*Actix*, etc.)
- Feature introduction of any crate

This talk is about

- **How to solve common problems** when building an actor system in Rust
 - Compilation-stable type id
 - Proc macros
 - Specialization
 - *Tick*-Based actor system



The Type-id Problem

The Type-id Problem

- Messages need to be encoded into buffers (`Vec<u8>`) so they could be transferred in a network
- How could a receiver actor recover the type information of a message?

Give each message payload type a *Type Id*

```
pub struct Message {  
    /// Indicating the type of payload  
    pub type_uid: TypeId,  
    /// The receiver's address info  
    pub recv: RawId,  
    /// Payload is encoded into a buffer (type erased)  
    pub payload: Vec<u8>  
}
```

Trait `std::any::Any`

1.0.0 [-][src]


[+] Show declaration

[-] A type to emulate dynamic typing.

Most types implement `Any`. However, any type which contains a non-`'static` reference does not. See the [module-level documentation](#) for more details.

Required methods

[-] `fn get_type_id(&self) -> TypeId`

└  This is a nightly-only experimental API. (`get_type_id` [#27745](#))

Gets the `TypeId` of `self`.

Type id

- Type ID needs to be stable across the network, or it could lead to decoding error
- We could **not** use `std::any::Typeid`
 - It generates different type ids with each compilation
 - Network could be running software from different compilations

Proc Macro to the Rescue

- Get the ident of target struct payload
- Save the ident + id combo to a local file
- Read the file on next compilation to recover the type id

```
pub trait UniqueTypeId {  
    fn type_uid() -> TypeId;  
}
```

```
pub struct TypeId {  
    pub t: u64  
}
```

```
#[derive(UniqueTypeId)]
pub struct StartTaskRequest {
    pub task_id: u64,
    pub sender: Option<RawId>
}
```

The payload struct we need to assign a unique type id to

```
quote::quote! {
    impl #impl_generics UniqueTypeId for #ident #ty_generics #where_clause {
        fn type_uid() -> TypeId {
            let mut t = #id; Load id from local file
            TypeId { t }
        }
    }
}
```

types.toml x

```
17 ResetTaskRequest=16
18 StartTaskRequest=18
19 PluginDeployRequest=19
20 ClusterPluginsInfo=20
21 ClusterTaskStatus=21
22 ResourceConfigUpdate=22
23 TaskStatus=23
```

Type Id

- Once we get a stable type id, we could use it to erase / recover type information for networking
- `T` — — — (serialization) — — — `Vec<u8>`
- `&[u8]` — — — (Type Id matching deserialization) — — — `T`

This is more or less similar to *Reflection* in Java


```
1  if message.type_id == PayloadA::type_id() {  
2      let payload_a = PayloadA::deserialize(&message.payload);  
3      // Handle this message ...  
4  } else if message.type_id == PayloadB::type_id() {  
5      let payload_b = PayloadB::deserialize(&message.payload);  
6      // Handle this message...  
7  } else {  
8      // More possibilities  
9  }
```



You still need to match against all types !

Solve it with proc
macros (again...)

```
pub trait Actor: ActorLifecycle + Any {  
    /// Return current actor's id  
    fn id(&self) -> &RawId;  
    /// Return a mutable ref of current actor's id  
    fn id_mut(&mut self) -> &mut RawId;  
    /// All-in-one handler for messages  
    fn handle_message(&mut self, message: &Message);  
}
```

```
#[derive(Actor)]  
struct SampleActor {  
    id: RawId  
}
```

```
impl #impl_generics Actor for #name #ty_generics #where_clause {
    fn id(&self) -> &RawId {
        &self.id
    }
    fn id_mut(&mut self) -> &mut RawId {
        &mut self.id
    }
    fn handle_message(&mut self, m: &Message) {
        handle_message!(self, m #(&, #ident_msg_types )*)
    }
}
```

Need declarations of message types

```
#[macro_export]
macro_rules! handle_message {
    ($type: expr, $raw_msg: expr $(, $msg_type: ty)*) => {
        match &$raw_msg.type_uid {
            $(
                $(
                    x if *x == <$msg_type>::type_uid() => {
                        let payload = $raw_msg.payload::<$msg_type>().unwrap();
                        if let Err(e) = Handler::<$msg_type>::handle($type, payload) {
                            log::error!("Failed to handle message: {}", e);
                        }
                    }
                )*
                _ => {}
            )
        }
    };
}
```



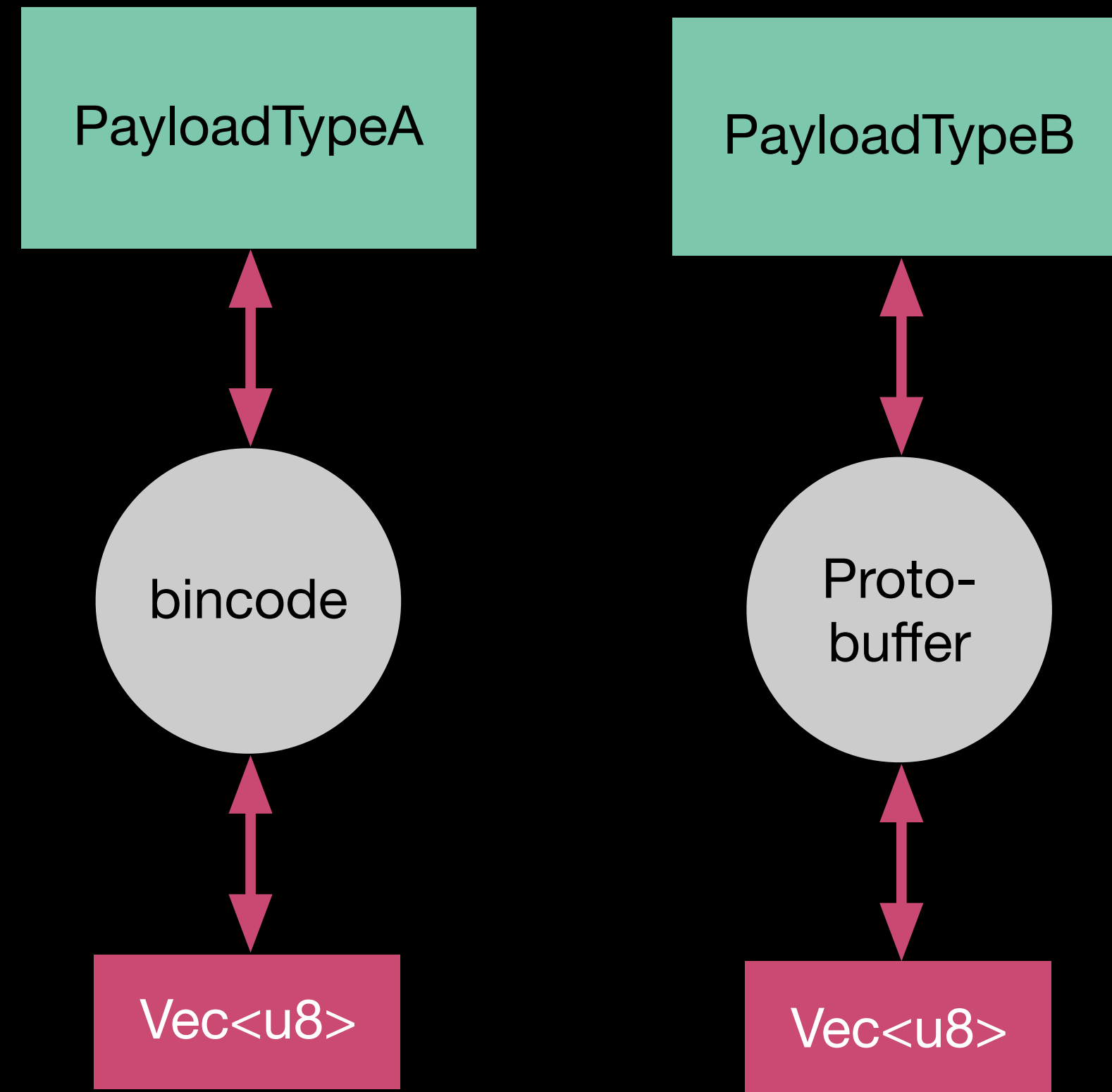
```
#[derive(Actor)]
#[Messages(PayloadA, PayloadB)]
struct SampleActor {
    id: RawId
}

impl Handler<PayloadA> for SampleActor {
    fn handle(&mut self, message: &PayloadA) {}
}

impl Handler<PayloadB> for SampleActor {
    fn handle(&mut self, message: &PayloadB) {}
}
```

Using proc macros we get:

- A super clean, self-explaining actor design
- Separating actor declaration / private logic with message handling logic
- Hiding dangerous type casting behind the curtain
- Minimal runtime cost (only an integer comparison)



The Codec Problem

The Codec Problem

- Messages could use different codecs
- We are adopting a fast se/de crate: *abomonation* by Frank McSherry
 - Super fast, but quite unsafe
 - Do not support HashMaps
- We could use different codecs for different messages
 - Important ones with hash maps: Bincode
 - Small, not-so-important messages: Abomonation

Specialization (RFC #1210)

- Allows trait impls to overlap with each other
- Allows a *default* impl of a trait

Specialization

Serde se/de traits

```
/// Data that could be serialized / deserialized to add/erase type information,  
/// wrapped in a [Message] and send around the cluster.  
pub trait Payload: Clone + Serialize + DeserializeOwned + UniqueTypeId + 'static {  
    /// Serialize payload into a buffer vector  
    fn serialize(&self) -> IoResult<Vec<u8>>;  
    /// Deserialize from a buffer slice, get a [Cow](std::borrow::Cow) value  
    /// back  
    fn deserialize(data: &[u8]) -> Result<Cow<'_, Self>, Error>;  
    /// Return the serialize buffer size of this payload, this method do NOT  
    /// do any serialization  
    fn size(&self) -> IoResult<u64>;  
}
```

Specialization

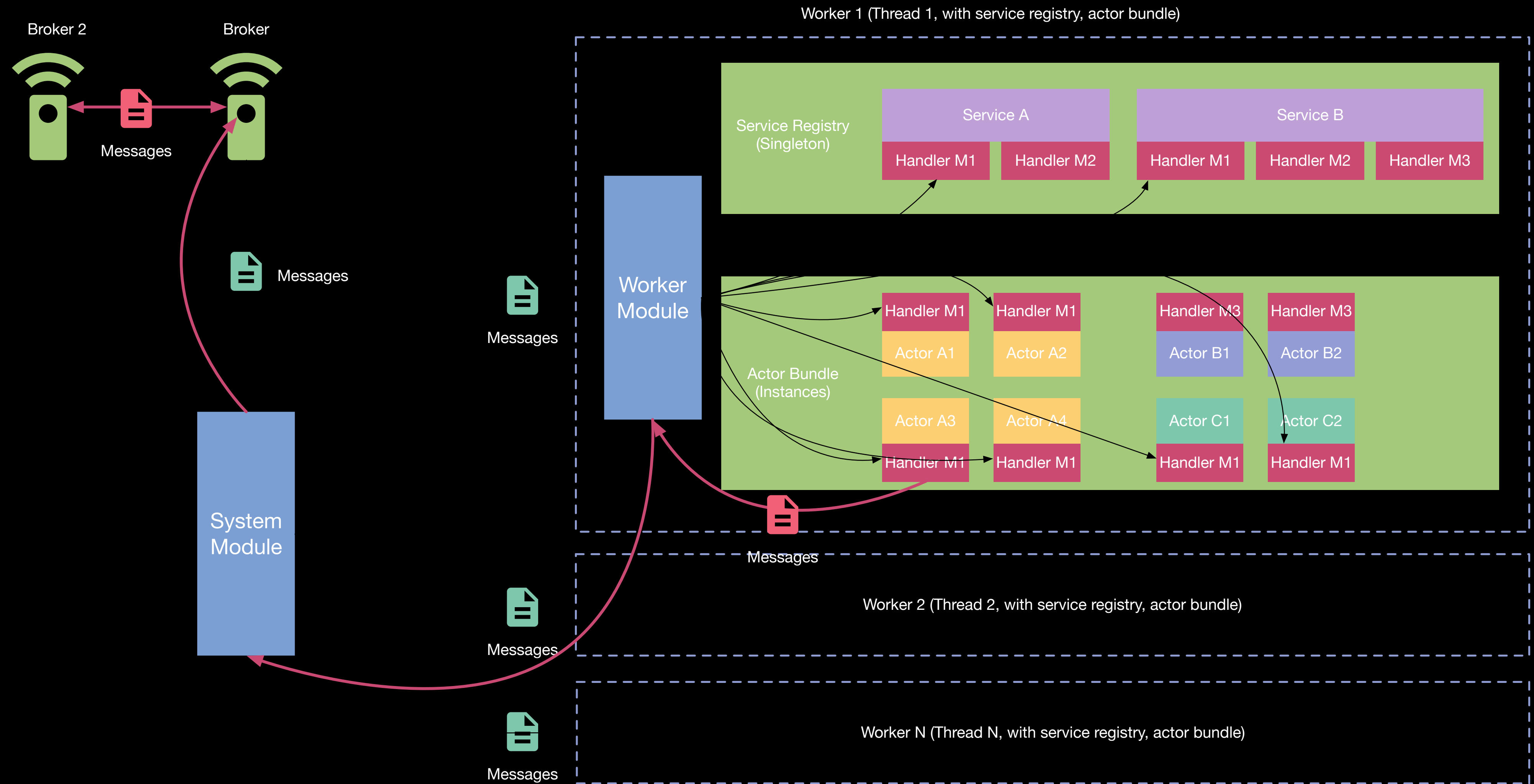
Default to serde/bincode

```
impl<T: Clone + Serialize + DeserializeOwned + UniqueTypeId + 'static> Payload for T {  
    default fn serialize(&self) -> IoResult<Vec<u8>> {  
        use std::error::Error;  
        bincode::serialize(&self)  
            .map_err(|err| std::io::Error::new(std::io::ErrorKind::InvalidInput, err.description()))  
    }  
  
    // Other impls  
}
```

```
impl<T: 'static + Abomination + DeserializeOwned + Serialize + UniqueTypeId + Clone> Payload for T {  
    fn serialize(&self) -> IoResult<Vec<u8>> {  
        let mut result = vec![];  
        unsafe { abomination::encode(self, &mut result)? };  
        Ok(result)  
    }  
  
    // other impls  
}
```

Specialization

- Available on nightly
- `#![feature(specialization)]`



Tick-based actor system

Tick - Why?

- Tick is useful for many use cases
 - Game design (logics are executed per frame)
 - Dataflow / Stream computation
 - Easier logic / waiting / event hook

Future with ticks

- Block tick for specific message

```
pub struct WaitForOnce {  
    deadline: u64,  
    message_signature: MessageSignature  
}
```

- Create a *Stream*, with each output, step tick forward by 1
 - Maintain a map of each tick's waits
 - If all waits are resolved, return Poll::Ready(messages)

Feature with ticks

- Wait for response
 - By setting deadline = 1
- User-defined pre-fetching
 - By setting dynamic deadline based on current traffic

Distributed Actor System

- Tick based message system
- Support multiple codecs with specialization
- Use compilation-stable type ids for arbitrary message type reflection
- We are working on open sourcing this actor framework in 2019

- Alibaba ❤️ Rust
- We are building quite some frameworks with Rust
- Looking forward to be a better participant of the community in 2019

Thanks for your time