Cage: an Actor library for Rust

Preston Sahabu
supervised by Dan Grossman and Eric Reed
University of Washington Computer Science & Engineering
June 11, 2014

Abstract

Physical limitations have forced computer hardware designers into providing multiple processors rather than faster ones, forcing software engineers to parallelize their programs for performance gains. Separately, systems languages such as C and C++ have long been criticized for their lack of safety, which is to say their programs are easily incorrect and difficult to reason about.

Rust, a new systems language from Mozilla, seeks to address both parallelism and safety with modern language concepts. To those ends, Rust's concurrency primitive is the task, which can be thought of as a lightweight thread that does not share memory with any other task. Message passing is used between these tasks in order to coordinate their work. While this primitive is sophisticated and suited to parallelism, the development community has expressed a desire for an Actor library.

The Actor model is a concurrent object-oriented programming model, where each Actor maintains state and takes actions based on the messages they receive from other Actors. In this investigation, the suitability of the Rust language for an Actor library is evaluated through an implementation. Various criteria will be evaluated in development, including library overhead and ease of use. It is expected that the implementation will be backed by the task primitive and its associated structures.

The success of the Actor model within the Rust language would continue its goal of bringing straightforward concurrency and safety into the systems language space.

Background

From objects to Actors

The Actor model of concurrent programming bears strong resemblance to object-oriented programming: both protocols bestow their data with behavior, protect internal state, and utilize on message passing. The key difference between them is that while object-oriented models are based on synchronous message passing, the Actor model is based on asynchronous message passing.

To conceptualize this idea, an Actor can be thought of as an object with a message queue, or an "inbox". When one Actor wants to communicate to another, it constructs a message and places it in the other's inbox. The other Actor processes its messages from its inbox one at a time, dispatching them to the appropriate procedures.

Once a message is in an inbox, it will be processed, but there are no temporal guarantees on when that processing will occur. After all, the other Actor may have received a deluge of messages prior to the new arrival, or it may have been waiting for a message. As such, the Actor model is a suitable model for concurrency when the order of an Actor's messages relative to those of others does not matter; the model is more concerned with data flow than synchronized minutia.

The Actor model can provide two major benefits to system performance. First, its principle of no shared memory heads off the need for lock-based approaches. Secondly, Actors run independently of one another, so they can be run in parallel. These benefits are especially valued on modern systems that increasingly rely on concurrency and parallelism for computational speedup.

Rust: because C++

The Rust programming language came about due to frustration with current systems languages. Programs written in those languages tend to be difficult to reason about due to their unsafe type systems and approaches to memory management. In light of modern language developments to address those issues, the only major advantage that systems languages still have is that they are extremely performant. Mozilla, the organizational backer of Rust, continues to endure the struggles of C++ in their Firefox browser, but they sought to create a better systems language.

In contrast to past reactions to C++, such as Java, Rust is keeping manual memory management to maintain performance, and ensures safety by surrounding it with a sophisticated type system and compiler. Rust also seeks to be more expressive than past systems languages, with one approach being the incorporation of functional programming concepts such as abstract data types and closures. Finally, as a systems language, Rust is strongly influenced by modern hardware challenges, which includes the move toward parallelism for performance gains. As such, Rust has developed a robust concurrency primitive with its tasks, which are a well-developed maturation of threads.

Rust has yet to reach a future-proof version 1.0, but its open-source development community hopes to finalize one by the end of 2014.

The motivation for this project

The development community for Rust expressed a desire for an Actor library because of the convenient abstraction it provides. However, the two background aspects of this project fit together very well outside of strict pragmatism: the Actor model addresses concurrency with an eye toward safety at the theoretical level, while Rust addresses safety with an eye toward concurrency at the language level. Should Rust succeed at implementing the Actor model, it will further its goal of spreading safety and concurrency in a pragmatic, performant way.

The Rust environment

After grasping the underpinnings of the Actor model, it made sense to examine what was available in the Rust language as well as its standard library before designing the Actor library.

Tasks and Channels

The first aspect that immediately jumps out to an observer of Rust's concurrency model is its task primitive. These are runtime scheduled closures running in user-level threads, and their well-advertised limitation is that they do not share memory with each other, at least not without great consternation. Rust's memory ownership rules also become very restrictive when interacting with tasks, as their backing closures capture ownership on all variables within their scope. (For ownership, one item is allowed to own a segment of memory at a time, be it on the stack or heap. Though loaning is possible without a change in ownership, task closures must have full ownership.)

To be clear, this does not mean that tasks cannot communicate with each other, but it does mean that different tasks cannot refer to the same address in memory. This restriction immediately rules out data races in most concurrent code. More importantly for this project, a task is almost exactly what is needed to back an Actor. It has the difficult systems aspects handled already (scheduling, stack allocation, etc.) while providing an independent thread of execution.

The only significant difference between a task and an Actor is that an Actor is associated with a structured, object-like set of data, whereas a task is essentially a free-running closure. These concepts can be stitched together, though maybe not elegantly. Another issue with tasks is their error handling scheme, which is to simply fail and hopefully be restarted again. In the context of the Actor model this is not desired, so providing users with a non-crash oriented way to propagate failure would be ideal.

For cross-task communication, the Rust library offers channels. These are bounded or unbounded queues containing a single data type with a `Sender` and a `Receiver` interface. Channels are a perfect implementation of the Actor model's inboxes, but they require a unifying message type that owns all of its memory.

Structs, Impls, and Traits

Structs are similar to how they are presented in C, but these structs also define concrete data types. Impls are methods in the same namespace as a struct, and can either be static, pure functions or functions called on an instance of that struct. Traits are similar to interfaces and mixins of other

languages, in that they are implemented on structs to give them the trait type, though there can also be default implementations.

Together these form Rust's model of data structuring, all centered around composition. There is no sense of inheritance, so it won't be possible to have a functioning base Actor that receives additional powers from the user. Rather, the user's Actor will have to be defined by a trait, and the Actor model will have to be filled in around it.

The Akka model

There have been several realizations of the Actor model since its conception, with Erlang's efforts being salient among them. However, the most recent and well-maintained implementation of the Actor model has been Akka, which is written in Scala and works in the Java Virtual Machine. Given the industrial popularity of Scala and Akka, it made sense to model a Rust Actor library after it. Below are the ideas of the Akka model that were seen as useful for a Rust implementation.

Overall design

The overall design of Akka is based on four entities: `Actor`, `ActorCell`, `ActorRef`, and `ActorPath`. Every `Actor` has access to an `ActorCell` context for creating children, referred to by an `ActorRef` and identified by an `ActorPath`. The `ActorRef` design choice was with remote Actors in mind, where the `ActorRef` lives on the local machine while the actual Actor could be across a network. In the final design of the Rust library, the `Actor` class was roughly translated into a trait, but the `ActorCell` is roughly equivalent to a `Context`, while the `ActorRef` is similar to an `Agent`.

Broadcasting

One prevailing theme of the Akka approach is the ability to send to arbitrary Actors across the system. This was achieved by Actors through `actorSelection()`, which allowed the lookup of Actors through a directory-like hierarchy. While this does provide some interesting abilities such as broadcasting to a group of Actors, it forces Actors to take part in a global state, and does not reinforce the idea of sending other Actor inboxes within messages. This aspect ended up being especially difficult to implement because Rust's channels are not one-to-many in any convenient abstraction.

Library message types

Akka utilizes singletons of universally understood messages to communicate at the library level, such as a `PoisonPill` to terminate Actor operation. This idea does not map well to Rust, which avoids shared memory at every opportunity, but it did inspire the use of library message types within an enum.

Futures and Any types

To get from Actor to non-Actor code, there would have to be some non-blocking value that an Actor could return as an intermediary for the actual value. In Akka, this translates to a `Future`, and Rust has a similar concept in its libraries. Another similarly named item between the two languages was an `Any` type on messages, but the use of that type is semantically very different, requiring extra effort in the Rust type system (this effort is expanded on in User messages).

Design iterations

The initial designs of the Rust Actor library were heavily influenced by the Akka design, in that Akka's ideas were either wholly accepted or generally rejected. Though the final construction ended up fairly close to the Akka design in general layout, below is an overview of the different approaches taken.

Typed Actors

Actors in Akka are untyped, meaning that they take Any argument and return nothing. While returning nothing makes sense because all messages are filtered through a single inbox and it would be impossible to distinguish a return from a call, it was odd that an Actor would be permitted to take messages that it couldn't necessarily handle.

One way to potentially handle this issue in Rust is to add a type parameter to the Actor trait, and on instantiation only allow that type of message through. This might be possible if message sending were restricted to an Actor's parent and its children. However, this becomes difficult to verify when sending messages to arbitrary Actors, as is the norm in Akka through `actorSelection()`. In addition, defining the message types that are permissible between two Actors would become extremely difficult. For example, there may be a message type that you want everyone to understand, whereas there's an additional message type for a special Actor. Defining what the special Actor can receive becomes difficult, and making sure it is received correctly is even harder.

While appealing given Rust's emphatic type system, bounding user messages that can be received is not feasible or even necessarily desirable. However, similar ideas were used at the library's message level. For example, on failures and undelivered messages, the sender would receive a special library message, which would be processed into a special call into the Actor outside of `receive()`, specifically `failure()` and `undelivered()`. This serves to clean up the `receive()` function, increase readability, and help with debugging.

Concurrent tree of all-powerful ActorRefs

Akka takes a particular interest in tracking the parent and children of Actors, and it achieves this mostly through `ActorRefs`. Though the idea of tracking immediate family made it into the final design, there were many ways to do it. The first of these were all powerful `ActorRefs` that maintained everything outside of the user Actor itself, including the tree of relations with other `ActorRefs` and the `Sender`.

The nightmare of such a concurrent tree across tasks with conflicting demands for sending messages and pointer fiddling was too daunting and objectively miserable to forge ahead with.

However, the notion of traversing a tree of nodes to find another Actor remained. `Contexts`, the reduced Rust successor to `ActorRefs`, track the senders to an Actor's parent and children. A special library message is sent along from Actor to Actor containing a list of path tokens, which is gradually traced using each `Context's` mini-directory, with appropriate off-ramps for failure. Though not as potentially "fast" on the lookup as a concurrent tree, the lack of locks is definitely cleaner and more idiomatic of safe Rust.

Hash table of Agents

Agents are basically inboxes for Rust Actors, but they can be Cloned and sent across task boundaries without any impact on the receiving end so they are extremely versatile. One idea was to make a hash table of these Agents for the arbitrary Actor lookup. While appealing, there would be no sense of a directory structure, and so broadcasting a message to an arbitrary group of Actors would be difficult. Also considered was a hash table of `Contexts`, but because those are dynamic when spawning children there would have been locking issues again.

While there isn't a direct corollary to this in the final implementation, a lock was eventually needed. In the `Stage` object, which maintains an Actor model from a root, it was possible for the main thread to start a new Actor from the root while the root was responding to a request about children of the root. Since only two threads were in contention, a simple mutex covered it.

User messages

Any way you want it

One deceptively easy concept in Scala became extraordinarily difficult in Rust. As mentioned before, all Akka actors receive `Any` messages. Since Rust had a similar `Any` trait, it figured to be a one-to-one port. However, the semantics of these `Any` types are different and they are used under very different circumstances.

Trait objects

At first, implementing user messages was straightforward: it was an empty `Message` trait that was bound by the `Send` kind so that it could be shipped along a channel in the Actor system. (Kinds are unique in that they function similarly to a trait, but are determined by the compiler based on the composition of a type rather than its behavior.) In order to send arbitrary user types across channels, `Message` is being used as a trait object, which is Rust's way of implementing polymorphism with a trait. This allows anything that implements `Message` and `Send` to be placed into a `Box<Message:Send>` (a `Box` is a heap allocated smart pointer) and sent across. On the receiving end, it seemed trivial for the Actor to cast these into `Any` types and perform runtime reflection.

Broadcasting

However, the first difficulty came when it was time to broadcast a message across a group of Agents. The nature of channels was designed to be one-to-one or many-to-one, in which case

non-cloneable, owned types were fine. However, with one-to-many sending, it was necessary to clone messages, and `Box<Message>` had no such implementation of `Clone` in its trait object.

Resolving this problem required a non-idiomatic maneuver of further bounding Message with a `Clone` trait and implementing a non-standard clone method on the trait object, `clone_me()`, so that `Box<Message>` could be cloned.[1] Ultimately this was positive for the users of the library, as they were now able to conveniently clone their `Box<Message>` items after coming across a channel, but it was also necessary to implement broadcasting.

Runtime reflection

In Scala, `Any` is the root of the object hierarchy, so all objects are `Any` types by default, and those methods can be called on any object through inheritance. In Rust, `Any` is a universal trait with implemented methods, but it can only be accessed if the item's type is listed as `Any`. Unfortunately, due to the use of trait objects when receiving `Box<Message:Send>` from a channel at the library level, it was not possible to cast it to a `Box<Any>` for the user because the underlying concrete type was concealed. Even though all types implement the `Any` trait, the compiler was unwilling to go through with what it saw as a potentially unsafe cast.

At this point it seemed sensible to backtrack and discard the Message trait, opting to send boxed `Any` types as messages, bounded by `Send` and `Clone` to maintain channel use and broadcasting. Unfortunately, it is not possible to bind the trait object `Any` by `Clone` because `Clone` is not a trait object bound; this is because `Clone` is behavior based, not a kind.

In order to bolt `Any` functionality onto the `Message` trait, an extremely non-idiomatic approach was taken: `Any` was added to the list of bounds on `Message`, and one of the `Any` extension traits (`std::any::AnyRefExt`) was manually implemented on `Message` to achieve runtime reflection.[2] Unfortunately, creators of Actors have to import that extension trait into their namespaces so that their code will compile. To conceal the difficulty of working with pseudo-`Any` types, a macro originally developed for `Any` types called match_any! can be imported to the Actor creator namespace.[3]

Taken altogether, the `Message` trait progressed from an empty marker to bounded by three types, one of which was internally implemented, and another implementation to clone `Box<Message>`. Though somewhat disorganized at the library level, the awkwardness is minimally exposed to the user.

The final design

trait Actor

The Actor trait is implemented by structs that would like to be put in the Actor system. It requires Actors to implement a function to receive `Messages` and take action on them using its `Context`.

---

[1] Thank you to Rust contributor huon for this approach.

[2] Thank you to Rust contributor Chris Morgan for this approach, based on the headers of his HTTP Rust project, Teepee. (http://chrismorgan.info/blog/teepee-design-header-representation.html)

[3] Thank you to classmate and Rust contributor Ty Overby for the match_any! macro.

It also requires a constructor to create a new instance that will be owned by the backing task. Other methods are offered if the Actor would like to implement them, such as `pre_start()` which runs before messages are received, or `undelivered()` when a message goes undelivered.

struct Agent and enum CageMessage

`Agents` function as inboxes for Actors in this library and are capable of being cloned and passed around to other Actors. `CageMessages` are sent along the channels that back an `Agent`, whose variants function as wrappers for the receiving task to use as dispatch information. For example, while a `UserMessage` wraps a message intended for `receive()`, `Failure` wraps a message intended for `failure()`, and `Kill` halts an Actor's receiving altogether.

struct Context and struct Stage

The `Context` struct is owned by the backing task and tracks the `Agents` of an Actor, its parent, and its children. It also allows an Actor to start a child Actor, stitching together the user's Actor struct with the task closure through an indefinite loop on the receiver and pattern matching on the incoming `CageMessages`. However, at the base of the Actor hierarchy there has to be a root that is not a normal Actor, and the `Stage` serves as that platform.

Example use

Examples can be found at: https://github.com/sahabp/RustActors/tree/master/Cage/examples

Conclusions and future extensions

Outside of the non-idiomatic use of trait objects for the Message type to behave as desired, the library code is actually quite straightforward. Though creators of Actors have to import a strange, seemingly irrelevant type in order for their code to compile, the resulting code is legible and maintainable. The Rust type system did not make translating Akka concepts easy, but that was more a result of moving from an inheritance-based system. As for the library's pragmatic application, that is to be left to the Rust development community for judgment.

Future iterations of a Rust Actor library could include Akka's especially powerful ability to communicate across machines. Akka actually implements such capabilities in TCP, which could become quickly complicated for the Cage system to deal with. As such, the incorporation of Rust's Unix socket libraries for inter-process communication should be the basis of any remote Actor extensions. Potential problems in this space include keeping a local reference while the Actor is on another machine; `Agents` could potentially fill this role, though they would need to be modified for sending across processes.

Though the typed Actor concept was discredited above, it was more that there was no convenient model to work from, and less that there was a solid, actually existing problem with the idea. Such a setup would increase the safety of Actors and reduce their complexity. Within the context of Cage, it is possible that `Agents` and `Actors` could gain type parameters (`Agent<T>, Actor<T>`) that define what set of `Messages` are valid for them to take, upon which other checks could be made.