NOTE: This proposal has been superseded

Please see Revision #3 of this proposal.
-Chris (Jan 17, 2021)

Protocol-based Actor Isolation: Draft #2

Author: Chris Lattner
November 18, 2020

Major Contributors:
Dave Abrahams, Doug Gregor, Paul Cantrell, Matthew Johnson

Introduction

The Swift Concurrency Roadmap was recently announced, and a key part of that roadmap is a proposal for an actor system. Actors are a powerful well known framework for building concurrent systems out of computations that communicate asynchronously.

One of the major goals of the proposed actor system is to "provide a mechanism for isolating state in concurrent programs to eliminate data races." Such a mechanism will be a major progression - most widely used languages and systems that provide concurrent programming abstractions do so in a way that exposes programmers to a wide range of bugs, including race conditions, deadlocks and other problems. That said, the proposal for actors does *not* actually provide a mechanism for isolating mutable state or eliminating data races.

This proposal describes an approach to address one of the challenging problems in this space - how to safely transfer values between actors (e.g. for arguments and result values of actor method calls). This approach is almost entirely library defined - the only language/compiler extension (for auto synthesizing a protocol) is already well precedented in Swift.

Global variables (and static members of classes) are another problem that is not covered in the core proposal - they are an orthogonal issue that is explored in the "Actors vs Global State" paper.

Motivation

Each actor instance in a program represents an "island of single threaded-ness", which makes them a natural synchronization point that holds a bag of mutable state. Each actor performs computation in parallel with other actors, but we want the vast majority of code in such a system to be synchronization free -- building on the logical independence of the actor, and using its queue as a synchronization point for its data.

However, this model is only safe when actors do not share unprotected mutable state: if two actors have a pointer to the same class instance and mutate state contained within it, they will race and have other synchronization problems. As such, a key question is: “when and how do we allow data to be transferred between actors?” Such transfers occur in arguments and results of actor method calls, in the proposed cross-actor ‘let’ references, and with global/static variables.

The Swift Language aspires to provide a safe and powerful programming model: in the case of an actor model, we want to achieve three things:

1) We want Swift programmers to get a static compiler error when they try to pass a value between actors that could introduce unprotected shared mutable state.
2) We want advanced programmers to be able to implement libraries with sophisticated techniques (e.g. a concurrent hash table) that can be used in a safe way by others.
3) We need to embrace the existing world, which contains a lot of code that wasn’t designed with the actor model in mind. We need a smooth and incremental migration story.

Before we jump into the proposed solution, let’s take a look at some common cases that we would like to be able to model along with the opportunities and challenges of each. This will help us reason about the design space we need to cover.

💖 Swift + Value Semantics

The first kind of type we need to support are simple values like integers. These can be trivially transferred between actors because they do not contain pointers.

Going beyond this, Swift has a strong emphasis on types with value semantics, which are safe to transfer across concurrent boundaries. Except for classes, Swift’s mechanisms for type composition provide value semantics when their elements do. This includes generic structs, as well as its core collections: for example, `Dictionary<Int, String>` can be directly shared across actor boundaries. Swift’s Copy on Write approach means that collections can be transferred without proactive data copying of their representations -- an extremely powerful fact that I believe will make the Swift actor model more efficient than other systems in practice.

However, everything isn’t simple here: the core collections can **not** be safely transferred across actor boundaries when they contain general class references, unsafe pointers, and other non-value types. We need a way to differentiate between the cases that are safe to transfer and those that are not.

Value Semantic Composition

Structs, enums and tuples are the primary mode for composition of values in Swift. These are all safe to transfer across actors -- if the data they contain is safe to transfer.

Immutable Classes

One common and efficient design pattern in concurrent programming is to build immutable data structures - it is perfectly safe to transfer a reference to a class between actors if the state within it never mutates. This design pattern is extremely efficient (because no synchronization - beyond ARC for the refcount) is required, can be used to build advanced data structures, and is widely explored by the pure-functional language community.

Internally Synchronized Reference Types

A common design pattern in concurrent systems is for a class to provide a “thread safe” API: they protect their state with explicit synchronization (mutexes, atomics, etc). Because the public API to the class is safe to use from multiple actors, the reference to the class can be directly transferred between actors safely.

Actor references themselves are an example of this: they are safe to pass between actors by passing a pointer, since the mutable state within an actor is implicitly protected by the actor queue.

“Transferring” Objects Between Actors

A fairly common pattern in concurrent systems is for one actor/thread to build up a data structure containing unsynchronized mutable state, then “hand it off” to a different actor to use by transferring the raw pointer. This is correct without synchronization if (and only if) the sending actor stops using the data that it built up - the result is that only one actor dynamically accesses the mutable state at a time.

There are both safe and unsafe ways to achieve this, e.g. see the discussion about “exotic” type systems in the “Alternatives Considered” section at the end.

Deep Copying Classes

One safe way to transfer reference types is to make a deep copy of the data structures, ensuring that the source and destination actor each have their own copy of mutable state. This can be expensive for large structures, but is/was commonly used in some Objective-C frameworks. General consensus is that this should be *explicit*, not something implicit in the definition of a type.

Motivation Conclusion

This is just a sampling of patterns, but as we can see, there are a wide range of different concurrent design patterns in widespread use. The design center of Swift around value types and encouraging use of structs is a very powerful and useful starting point, but we need to be able to reason about the complex cases as well - both for communities that want to be able express high performance APIs for a given domain but also because we need to work with legacy code that won't get rewritten overnight.

As such, it is important to consider approaches that allow library authors to express the intent of their types, it is important for app programmers to be able to work with uncooperative libraries retroactively, and it is important that we provide safety as well as unsafe escape hatches so we can all just "get stuff done" in the face of an imperfect world that is in a process of transition.

Finally, our goal is for Swift (in general and in this specific case) is to be a highly principled system that is sound and easy to use. In 20 years, many new libraries will be built for Swift and its ultimate concurrency model. These libraries will be built around value semantic types, but should also allow expert programmers to deploy state of the art techniques like lock-free algorithms, use immutable types, or whatever other design pattern makes sense for their domain. We want users of these APIs to not have to care how they are implemented internally.

Proposed Solution + Detailed Design

The high level design of this proposal revolves around an `ActorSendable` and `ValueSemantic` protocol, autosynthesized conformance to the protocol for many value types, adoption of the `ValueSemantic` by standard library types.

Beyond the basic proposal, in the future we would like to add support for closures, a set of simple adapter types to handle legacy compatibility cases, and first class support for Objective-C frameworks. These are described in the following section.

ActorSendable Protocol

The core of this proposal is the `ActorSendable` marker protocol defined in the Swift standard library:

```
protocol ActorSendable {}
```

The compiler rejects any attempts to pass data between actors when the argument or result does not conform to the `ActorSendable` protocol:

```
actor class SomeActor {
```

```
  // async functions are usable *within* the actor, so this
  // is ok.
  func doThing(string: NSMutableString) async {...}
}

// ... but they cannot be called by other actors or other code:
func f(a: SomeActor, myString: NSMutableString) async {
  // error: 'NSMutableString' may not be passed across actors;
  //        it does not conform to 'ActorSendable'
  await a.doThing(string: myString)
}
```

The `ActorSendable` protocol models types that are allowed to be safely passed across actor boundaries by copying the value. This includes value semantic types, references to immutable reference types, internally synchronized reference types, and potentially other future type system extensions for unique ownership etc.

Note that incorrect conformance to this protocol can introduce bugs in your program (just as an incorrect implementation of `Hashable` can break invariants). For example, it would be incorrect (and very unwise!) to add to your codebase, because it results in a shared reference to mutable state:

```
extension NSMutableString : ActorSendable {}
```

While this is a possible bug, Swift doesn't define away all classes of bugs, and this is a relatively obscure thing to do. Allowing reference types to conform to this protocol is essential to allow advanced types to work nicely with the actor system, e.g. those that are internally synchronized or immutable by definition.

ValueSemantic Protocol

While we need the ability for experts to define types that are `ActorSendable`, the norm in Swift is for many types to have proper value semantics. Such behavior is inherent to aggregates of value semantic types, and there are generic algorithms that want to be constrained to value semantic types. This is a commonly discussed and requested feature in general for Swift.

As such, we define a `ValueSemantic` protocol that refines `ActorSendable`, since all value semantic types can be safely passed across actor boundaries. The `ValueSemantic` protocol is another marker protocol defined as:

```
protocol ValueSemantic : ActorSendable { }
```

The tricky part of the `ValueSemantic` protocol is defining exactly what it means and what types are allowed to conform to it. This definition is a key subtask of the general concurrency feature proposal, but the final definition of that is left as a separable design task. This is hard and expert level attention is necessary here.

Tuple conformance to ActorSendable and ValueSemantic

Swift has hard coded conformances for tuples to specific protocols, and this should be extended to `ActorSendable` and `ValueSemantic`, when the tuples elements are all `ActorSendable` or `ValueSemantic` (respectively).

Auto-synthesized Struct/Enum ValueSemantic Conformances

Value semantic types are extremely common in Swift and aggregates of them are also correctly value semantic. As such, the Swift compiler should implicitly auto-synthesize conformances for non-public structs and enums that are compositions of other `ValueSemantic` types. It does so by adding the marker protocol to the struct/enum when appropriate:

```
struct MyPerson { var name: String, age: Int }
struct MyNSPerson { var name: NSMutableString, age: Int }

actor class SomeActor {
  // Structs and tuples are ok to send and receive!
  public func doThing(x: MyPerson, y: (Int, Float)) async {..}

  // error: MyNSPerson doesn't conform to ActorSendable due to
unsendable 'NSMutableString' member.
  public func doThing(x: MyNSPerson) async {..}
}
```

Because of this behavior, user-defined struct/enums will "just work" by default with actor methods in a correct and easy to use way in the majority case. This also means that structs/tuples containing `NSMutableStrings` and other non-`ValueSemantic` types will not be `ActorSendable` by default which is good for memory safety.

This proposal chooses to limit this to non-public types because Swift generally wants extra attention paid to the public APIs of types, but we could make autosynthesis happen independent of access control, or we could make it only happen when explicitly opt'd into. (Hat tip to Matthew Johnson for suggesting this).

This approach follows the precedent of SE-0185, SE-0266, and SE-0283, but chooses to make the conformance to `ValueSemantic` implicit. An alternative design would follow those proposals more closely and require an explicit "`: ValueSemantic`" on the definition of `MyPerson`. Please see "Alternatives Considered" at the end of this proposal for more discussion about this.

Adoption of ValueSemantic by Standard Library Types

While normal user-defined value types will not have to interact with `ActorSendable` or `ValueSemantic` directly, the standard library is the bottom of the pile of turtles and is defined in terms of more complex things like builtin LLVM types. As such, standard library types like `Int` and `String` need to conform to `ValueSemantic` explicitly.

These value semantic types need to conform by adopting the marker protocol:

```
extension Int : ValueSemantic {}
extension String : ValueSemantic {}
// … etc.
```

Similarly, conditional conformances for various collection and optional types can be defined naturally:

```
extension Array : ValueSemantic where Element : ValueSemantic {}
```

Beyond actors, such conformances are important for generic algorithms that want to be constrained to value semantic types.

All actors references are themselves `ActorSendable`, so they can implicitly conform to `ActorSendable` or perhaps it is best for the proposed `Actor` protocol to conform.

Note that `UnsafeMutablePointer` and `UnsafeBufferPointer` are highly debatable - the value semantic proposal should define whether it is correct for them to be marked as `ValueSemantic` or not.

A nice aspect of this proposal is that it directly composes with the existing Swift generics system and builds on the existing modeling power of the Swift type system. No new fancy type system machinery is required.

Future Work / Follow-on Projects

In addition to the base proposal, there are several follow-on things that should be explored once this proposal converges, to really fill out the programming model here.

Introduce an @actorSendable attribute for closures

There are many useful reasons why you'd want to send bits of computation between actors in the form of a closure, and some cases which are obviously safe to do so (e.g. closures that don't capture anything). Going further, the `ActorSendable` protocol and its formalism allows us to allow captures to be transferred between actors as well -- so long as the captured values are `ActorSendable`. This could possibly be extended to `ValueSemantic` and `@valueSemantic` if there was a need.

What we would really like to say is that a specific class of closures are `ActorSendable` - and that kind of closure needs to be demarcated as part of its type. This would spelled as an attribute, allowing something like this to work:

```
actor class MyContactList {
  func filteredElements(_ fn: @actorSendable (ContactElement) -> Bool)
async -> [ContactElement] { … }
}
```

Which could then be used like so:

```
// Closures with no captures are ok!
list = await contactList.filteredElements { $0.firstName != “Max” }
// Capturing a ‘searchName’ string is ok, because it is ActorSendable.
list = await contactList.filteredElements { $0.firstName==searchName }
// @actorSendable is part of the type, so passing a compatible
// function works as well.
list = await contactList.filteredElements(dynamicPredicate)
```

Unfortunately, Swift doesn't currently allow non-nominal types like functions to conform to protocols, but recently added some hard coded conformances for tuples. Such a thing could be added for functions and `ActorSendable` as well.

One question to decide is how to handle "var" captures - normally closures capture them by-reference, which isn't what we want here. We could either reject the second example above and require:

```
list = await contactList.filteredElements {
   [searchName] in $0.firstName == searchName
}
```

Or we could make by-copy capture be implicit for `@actorSendable` closures. Hat tip to cukr for pointing this out.

Adaptor Types for Legacy Codebases

NOTE: This section is not considered part of the proposal - it is included just to illustrate aspects of the design.

The proposal above provides good support for simple composition and Swift types that are updated to support Actors. Further, Swift's support for retroactive conformance of protocols makes it possible for users to work with codebases that haven't been updated yet.

However, there is an additional important aspect of compatibility with existing frameworks that is important to confront: frameworks are sometimes designed around dense graphs of mutable objects with ad hoc structures. While it would be nice to "rewrite the world" eventually, practical Swift programmers will need support to "get things done" in the meantime. By analogy, when Swift first came out, most Objective-C frameworks were not audited for nullability. We introduced "`ImplicitlyUnwrappedOptional`" to handle the transition period, which gracefully faded from use over the years.

To illustrate how we can do this with actors, consider a pattern that is common in Objective-C frameworks: passing an object graph across threads by "transferring" the reference across threads - this is useful but not safe! Programmers will want to be able to express these things as part of their actor APIs within their apps.

This can be achieved by the introduction of a generic helper struct:

```
@propertyWrapper
struct UnsafeTransfer<T: AnyObject> : ActorSendable {
  var wrappedValue: T
  init(wrappedValue: Wrapped) {
    self.wrappedValue = wrappedValue
  }
}
```

For example, let's assume that `NSMutableDictionary` isn't updated to know about `ActorSendable` (just to have a concrete example of an existing type that we want to work

with). The struct above would allow you (as an app programmer) to write actor APIs in your application like this:

```
actor class MyAppActor {
  // The caller *promises* that it won't use the transferred object.
  public func doStuff(dict: UnsafeTransfer<NSMutableDictionary>) async
}
```

While this isn't particularly pretty, it is effective at getting things done on the caller side when you need to work with unaudited and unsafe code. This can also be sugared into a parameter attribute using the recently proposed extension to property wrappers for arguments, allowing a prettier declaration and caller-side syntax:

```
actor class MyAppActor {
  // The caller *promises* that it won't use the transferred object.
  public func doStuff(@UnsafeTransfer dict: NSMutableDictionary) async
}
```

Objective-C Framework Support

Objective-C has established patterns that would make sense to pull into this framework en-masse, e.g. the `NSCopying` protocol is one important and widely adopted protocol that should be onboarded into this framework.

General consensus is that it is important to make copies explicit in the model, so we can implement an `NSCopied` helper like so:

```
@propertyWrapper
struct NSCopied<Wrapped: NSCopying>: ActorSendable {
  let wrappedValue: Wrapped

  init(wrappedValue: Wrapped) {
    self.wrappedValue = wrappedValue.copy() as! Wrapped
  }
}
```

This would allow individual arguments and results of actor methods to opt-into a copy like this:

```
actor class MyAppActor {
  // The string is implicitly copied each time you invoke this.
```

```
  public func lookup(@NSCopied name: NSString) -> Int async
}
```

One random note: the Objective-C static type system is not very helpful to us with immutability here: statically typed `NSString`'s may actually be dynamically `NSMutableString`'s due to their subclass relationships. Because of this, it isn't safe to assume that values of `NSString` type are dynamically immutable -- they should be implemented to invoke the `copy()` method.

Various sugar

There are many possible ways to add sugar to parts of this proposal. This section captures some of them, but they can be implemented locally in downstream packages, are debatable, and such debate would detract from the core proposal. I think it is really important to split them out to follow-on proposals if others are interested in pursuing them.

Matthew Johnson points out that most value semantic types already conform to `Equatable` and usually `Hashable`. Explicit conformance could be made more convenient if we introduced a couple typealiases:

```
typealias EquatableValue = Equatable & ValueSemantic
typealias HashableValue = Hashable & ValueSemantic
```

Source Compatibility

This is fully source compatible with existing code bases - it is a purely additive proposal. Furthermore, by including this in "Actors 1.0," it eliminates a major source break in "Actors 2.0" that would be required to lock down on what is passed across actor boundaries.

Effect on API resilience

The *implicit* conformance of structs and enums to `ValueSemantic` has a subtle impact on resilience: adding a non-`ValueSemantic` member to a struct will break the struct's conformance to `ValueSemantic`, and the implicit nature of this conformance may make it difficult to detect. For this reason, we may choose to make `ValueSemantic` a required annotation on a struct (see alternatives considered below).

Alternatives Considered

There are several alternatives that make sense to discuss w.r.t. this proposal. Here we capture some of the bigger ones.

Make ValueSemantic and ActorSendable explicitly unsafe

One disadvantage of this proposal is that manually conforming a reference type to `ValueSemantic` is an unchecked unsafe operation:

```
// This conformance is incorrect!
class Box : ValueSemantic {
  var x : Int
}
```

Swift typically handles this by requiring the word “Unsafe” to be uttered when performing unsafe operations. We could handle this in a couple of ways, e.g. by putting the word Unsafe into the protocol name:

```
// I know what I’m doing!
class Thing : UnsafeValueSemantic {...}
```

The problem with this approach is that the `ValueSemantic` protocol is also extremely useful as a generic constraint, and its use there is perfectly safe -- it is really just the conformance that is unsafe.

If this is an important problem to solve, we could require a new `@unsafe` attribute in the conformance list:

```
// error: conforming a class to ValueSemantic is an unsafe operation
class Thing1 : ValueSemantic {...}

// Ok
class Thing2 : @unsafe ValueSemantic {...}
```

There are two ways we could achieve this: the quick and dirty approach would be to special case the `ValueSemantic` and `ActorSendable` protocols in the conformance checker.

A more general way to go would be to introduce a new `@unsafeConformance` attribute that could be applied to any protocol to enable this behavior:

```
@unsafeConformance
protocol ActorSendable {}
```

This would make “unsafe conformance” a more general part of the language. I’m not sure if unsafe conformances come up in other domains, but if so, a general solution like this would be nice.

Explicit struct/enum Conformance to ValueSemantic

The proposal suggests that structs and enums should conform to ValueSemantic any time all of their members conform. Another approach is to follow the prior art in autosynthesized Swift protocols more closely and require an explicit conformance for the struct:

```
struct MyPerson2 : ValueSemantic {
  var name: String, age: Int
}
actor class SomeActor {
  // error: MyPerson (above) doesn't conform to ActorSendable!
  public func doThing(x: MyPerson) async {..}

  // Ok, explicitly conforms.
  public func doThing(x: MyPerson2) async {..}
}
```

Both models work fine, here are some tradeoffs I see:

- The "`: ValueSemantic`" marker is mostly boilerplate.
- The conformance does induce some minor code bloat implicitly.
- Explicit conformances would give you a compiler error eagerly if you define a struct with non-sendable things and try to make it implicitly sendable. With implicit conformances you only get the error when trying to send it in an actor method.
- Implicit conformances have a minor resilience issue mentioned above.
- Explicit conformances are more consistent with `Hashable` and `Equatable`, and consistency is good.
- It is easier to start out with explicit conformance and later switch to implicit conformance (without breaking source compatibility) than the other way around.
- Not all struct/enum compositions of value semantic types are themselves value semantic (examples), so we might need a way to disable autosynthesis, making the proposal more complicated.
- While the "only implicit for non-public types" rules is intended to be a pragmatic concession, I would expect it to lead to questions like "Why does making my type public cause my build to break?" on stack overflow?

It isn't obvious how common "user defined types passed across actor boundaries" will be. Overall, if the boilerplate issue is acceptable then it seems better to make this explicit.

Codable conforming to ValueSemantic

Not proposed, but it would be perfectly possible to make all `Codable` types conform to `ValueSemantic`. This would implement a deep copy by coding and then decoding the value. While this would be semantically correct, this would be a significant and surprising performance impact for some types, it seems better to have library authors implement native support and produce a compiler error in the meantime.

Exotic Type System Features

The Swift Concurrency Roadmap mentions that a future iteration of the feature set could introduce new type system features like "`mutableIfUnique`" classes, and it is easy to imagine that move semantics and unique ownership could get introduced into Swift someday.

While it is hard to evaluate future proposals without all the details, such features should compose very naturally with this design: we can either make those types conform to `ActorSendable`, or we could change the check in the compiler itself to allow both `ActorSendable` and <novel type system feature> types as arguments and results.

Unique Pointers

Some languages (C++, Rust, etc) have unique pointer types, and it might seem appealing to try to allow transferring unique pointers directly across actor boundaries. However, this is only safe if the system prevents formation of interior pointers.

For example, a hypothetical system that allowed unique references could permit something like this:

```
unique class C1 { let subReference: C2 }
class C2 { var mutableState : Int }
```

It is not safe in general to allow pointer transfers of "C1" references across actors - even though the reference to C1 is unique, the sending actor could have other references to the underlying C2 object. As such, sound type systems that attempt to achieve this end up needing to tightly constrain the design space around unique references.

Unique references also cannot directly model DAGs and other common structures.

Synthesize conformances for ActorSendable

This proposal provides great support for types with value semantics (including immutable reference types), while allowing more advanced types like concurrent hash tables to be passed

as actor arguments and results seamlessly by conforming to `ActorSendable`. However, this proposal does not make compositions of `ActorSendable` types as elegant to use as compositions of `ValueSemantic` types.

One alternative is to extend autosynthesis and tuple conformances to aggregates of "merely `ActorSendable`" types. If we went with this direction, we should also add more conditional conformances for standard library collections as well, e.g.:

```
extension Array : ActorSendable where Element : ActorSendable {}
```

This is a decision made with the goal of reducing complexity of the proposal, guided by the idea that internally synchronized types are important to support, but probably not important to sugar. This sugar isn't likely to be used, and if it is more important than currently expected, it can always be added incrementally in the future.

Support an implicit copy hook

The first revision of this proposal allowed types to define custom behavior when they are sent across actor boundaries, through the implementation of an `unsafeSendToActor` protocol requirement. This increased the complexity of the proposal, admitted undesired functionality (implicit copy behavior) and made the recursive aggregate case more expensive and would result in larger code size.

Do Not Enforce Transfers in "Actors 1.0"

The current proposal for the actor system suggests that we launch "Actors 1.0" without any enforcement of cross-actor value transfers, but then introduce an "Actors 2.0" system sometime later that locks down on this (presumably with new exotic type system features available). It suggests introducing an "`actorlocal`" concept and attributes to opt types out of actor isolation. One could imagine that "Actors 2.0" would ship a year or more after "Actors 1.0" and, therefore, that a significant amount of user and library code will be written against "Actors 1.0".

That approach has several downsides compared to this proposal:

1) None of the "Actor 1.0" code will be memory safe - which is the primary stated goal of the actor model. We will have introduced a new Swift concurrency model that is extremely unsafe. People will form first impressions about it, and many bugs will be had.
2) The proposed type system extensions are not very expressive, certainly not to all of the cases covered by this proposal, so some code may not be able to migrate to Actors 2.0.
3) "Actors 2.0" will be extremely incompatible with "Actors 1.0" and will put the Swift community through a very difficult and unnecessary migration.

Furthermore, the proposed `actorlocal` feature is effectively the same thing as `ActorSendable`, but with a different sense and implemented as a language feature instead of a library feature. It has a profound impact on the entire type system - because it isn't a protocol, the generics system needs to be significantly extended to propagate and support the bit. We would need the equivalent of conditional and retroactive conformance for the bit, etc. The additional attributes also complicate the Swift language, both its implementation and its user experience.

Conclusion

This proposal defines a very simple approach for making data transfers safe across actors. It requires minimal compiler/language support (just the `ValueSemantic` synthesis behavior), is extensible by users, works with legacy code bases, and provides a simple model that we can feel good about even 20 years from now.

Because the feature is almost entirely a library feature that builds on existing language support, it is easy to define wrapper types that extend it for domain specific concerns (along the lines of the `NSCopied` example above), and retroactive conformance makes it easy for users to work with older libraries that haven't been updated to know about actors yet.

I would highly recommend that we include this proposal (or something like it) in the "Actors 1.0" definition and launch.