RFP-195 Actor Runtime

Draft

15 Pages

Abstract

~~These are the requirements on~~The Actor model is an architectural pattern designed to support high-scale concurrency without the need for locking constructs and with simple memory safety rules. This RFP looks at how to add support for the Actor concurrency model to the OSGi environment. The general direction is to retain the composition of OSGi services as the basic model for creating applications, while at the same time allowing application developers to schedule concurrent execution with an actor runtime, rather than to use threads and locks. This requires that a balanced way is found to mix the blocking parts of the computation as expressed by calling services with the non-blocking parts of the computation as expressed by sending messages between actors. Each type of computation is structured around a modularity construct with different properties – bundles for the blocking and actors for the non-blocking. ~~Perhaps the middle ground lies with a type of "actor service" which communicates through messages rather than method calls.~~

0 Document Information

0.1 License

DISTRIBUTION AND FEEDBACK LICENSE, Version 2.0

The OSGi Alliance hereby grants you a limited copyright license to copy and display this document (the “Distribution”) in any medium without fee or royalty. This Distribution license is exclusively for the purpose of reviewing and providing feedback to the OSGi Alliance. You agree not to modify the Distribution in any way and further agree to not participate in any way in the making of derivative works thereof, other than as a necessary result of reviewing and providing feedback to the Distribution. You also agree to cause this notice, along with the accompanying consent, to be included on all copies (or portions thereof) of the Distribution. The OSGi Alliance also grants you a perpetual, non-exclusive, worldwide, fully paid-up, royalty free, limited license (without the right to sublicense) under any applicable copyrights, to create and/or distribute an implementation of the Distribution that: (i) fully implements the Distribution including all its required interfaces and functionality; (ii) does not modify, subset, superset or otherwise extend the OSGi Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the OSGi Name Space other than those required and authorized by the Distribution. An implementation that does not satisfy limitations (i)-(ii) is not considered an implementation of the Distribution, does not receive the benefits of this license, and must not be described as an implementation of the Distribution. "OSGi Name Space" shall mean the public class or interface declarations whose names begin with "org.osgi" or any recognized successors or replacements thereof. The OSGi Alliance expressly reserves all rights not granted pursuant to these limited copyright licenses including termination of the license at will at any time.

EXCEPT FOR THE LIMITED COPYRIGHT LICENSES GRANTED ABOVE, THE OSGi ALLIANCE DOES NOT GRANT, EITHER EXPRESSLY OR IMPLIEDLY, A LICENSE TO ANY INTELLECTUAL PROPERTY IT, OR ANY THIRD PARTIES, OWN OR CONTROL. Title to the copyright in the Distribution will at all times remain with the OSGi Alliance. The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted therein are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

THE DISTRIBUTION IS PROVIDED "AS IS," AND THE OSGi ALLIANCE (INCLUDING ANY THIRD PARTIES THAT HAVE CONTRIBUTED TO THE DISTRIBUTION) MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR TITLE; THAT THE CONTENTS OF THE DISTRIBUTION ARE SUITABLE FOR ANY PURPOSE; NOR THAT THE IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.
NEITHER THE OSGi ALLIANCE NOR ANY THIRD PARTY WILL BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF OR RELATING TO ANY USE OR DISTRIBUTION OF THE DISTRIBUTION.

Implementation of certain elements of this Distribution may be subject to third party intellectual property rights, including without limitation, patent rights (such a third party may or may not be a member of the OSGi Alliance). The OSGi Alliance is not responsible and shall not be held responsible in any manner for identifying or failing to identify any or all such third party intellectual property rights.

The Distribution is a draft. As a result, the final product may change substantially by the time of final publication, and you are cautioned against relying on the content of this Distribution. You are encouraged to update any implementation of the Distribution if and when such Distribution becomes a final specification.

The OSGi Alliance is willing to receive input, suggestions and other feedback (“Feedback”) on the Distribution. By providing such Feedback to the OSGi Alliance, you grant to the OSGi Alliance and all its Members a non-exclusive, non-transferable,

worldwide, perpetual, irrevocable, royalty-free copyright license to copy, publish, license, modify, sublicense or otherwise distribute and exploit your Feedback for any purpose. Likewise, if incorporation of your Feedback would cause an implementation of the Distribution, including as it may be modified, amended, or published at any point in the future ("Future Specification"), to necessarily infringe a patent or patent application that you own or control, you hereby commit to grant to all implementers of such Distribution or Future Specification an irrevocable, worldwide, sublicenseable, royalty free license under such patent or patent application to make, have made, use, sell, offer for sale, import and export products or services that implement such Distribution or Future Specification. You warrant that (a) to the best of your knowledge you have the right to provide this Feedback, and if you are providing Feedback on behalf of a company, you have the rights to provide Feedback on behalf of your company; (b) the Feedback is not confidential to you and does not violate the copyright or trade secret interests of another; and (c) to the best of your knowledge, use of the Feedback would not cause an implementation of the Distribution or a Future Specification to necessarily infringe any third-party patent or patent application known to you. You also acknowledge that the OSGi Alliance is not required to incorporate your Feedback into any version of the Distribution or a Future Specification.

I HEREBY ACKNOWLEDGE AND AGREE TO THE TERMS AND CONDITIONS DELINEATED ABOVE.

0.2 Trademarks

OSGi™ is a trademark, registered trademark, or service mark of the OSGi Alliance in the US and other countries. Java is a trademark, registered trademark, or service mark of Oracle Corporation in the US and other countries. All other trademarks, registered trademarks, or service marks used in this document are the property of their respective owners and are hereby recognized.

0.3 Feedback

This document can be downloaded from the OSGi Alliance design repository at https://github.com/osgi/design The public can provide feedback about this document by opening a bug at https://www.osgi.org/bugzilla/.

0.4 Table of Contents

0.5 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in 5.1.

```
Source code is shown in this typeface.
```

0.6 Revision History

The last named individual in this history is currently responsible for this document.

Revision	Date	Comments
Initial	Apr 16 2019	Todor Boev: Initial draft based largely on research in integrating Akka with OSGi services.
0.1	May 9 2019	Todor Boev: Numerous clarifications and updates. Added requirement numbers.
0.2	May 21 2019	Todor Boev: Substantially improved the persistence section, clarified the "ask" pattern
0.3	May 21 2019	Tim Ward: Updates to fill out the RFP ahead of discussions at the Face to Face meeting
0.4	May 24 2019	Todor Boev: Added minor clarifications resulting from the Face to Face meeting. Added an "Asynchronous microservice" use case.

1 Introduction

Software built using OSGi technology is primarily based around interactions between services published by OSGi bundles. These services are Java Objects, and therefore typically use a synchronous, blocking invocation model (as is common throughout Java). Service Objects may be called concurrently from many threads in many bundles, and therefore it is critical that services are thread-safe.

Writing thread-safe services is simple if the services are stateless, however if the service does have internal state then access to it must be protected correctly. This can quickly become difficult to manage, and result in a service object which is difficult to maintain. In some cases these stateful services could be implemented much more simply using the Actor Model. This RFP began as an investigation into the possibility of using the Actor model inside OSGi as a simpler way to handle concurrency for some types of OSGi services, especially those representing external devices.

Initial research was carried out using the Akka framework, however the intent of this RFP is to explore actors generally, and not a specific implementation.

2 Application Domain

Today's software needs to be highly concurrent for several reasons. On one hand concurrent programs can make efficient use of all processor cores available to them. In this way concurrent programs can scale through parallelism. This is critical since for many years the only way to increase computational power is to add more cores or more machines rather than to make faster cores. On the other hand a lot of the software today needs to operate in real world conditions where it faces unpredictable concurrent events coming from the environment much like humans do. These two forces combine to create the need for software that can use efficiently all computational resources available to execute as quickly as possible reactions to concurrently occurring events.

One proven way to build software that can handle such conditions is the Actor model of computation. On the JVM the most prominent implementation of this model is the Akka framework based on the Scala programming language. Outside the JVM the most prominent implementation of actors is the Erlang programming language and libraries and it's underlying runtime.

2.1 Terminology + Abbreviations

- **Message:** a data structure (DTO) usually ~~which is~~ considered immutable.
- **Behavior:** a functional object, which receives a message and returns a new behavior. Useful work is performed as side effects during the function call. State is passed from one behavior by enclosing it in the next behavior.

- **Actor:** an abstract container that manages the sequential atomic processing of messages by an unfolding chain of behaviors. The actor uses each behavior to processes one message and then replaces it with the next behavior.
- **Actor address**: an immutable opaque identifier of an actor. Each actor has at least one address. The only way to send a message to an actor is to sent it to it's address. Actors can send messages to their own address.
- **Asynchronous message delivery**: sending a message does not return a value and does not block behavior execution (fire and forget).
- **At-most-once message delivery**: once sent a message can be delivered to the recipient zero or one time.
- **Supervisor**: an actor that decides how to handle failures in one or more other actors. Each actor has zero or one supervisor.
- **Supervisor hierarchy**: the tree formed by traversing the supervisor relationships between all actors.
- **Supervisor strategy:** a function specified by the supervisor that is called to make a decision on what to do with a failed child. E.g. the function can return a signal "restart" to cause the failed child to be reset to it's initial behavior and continue processing.
- **Actor runtime**: the engine which drives the concurrent execution of a large number of actors

3 Problem Description

As all other applications OSGi programs increasingly need to be deployed in real world conditions where the primary model of interaction is through large numbers of concurrent events coming over network interfaces or other asynchronous and unreliable channels.

Up until now we have addressed these conditions with concurrency primitives like Promises and PushStreams. These models however are functional in nature and do not model well concurrently changing state. At present we model dynamic state with services and service events. While this model has proven to be an effective way to structure an application the communication between the stateful services is still done through regular method calls done on multiple threads and protected by locks. These concurrency mechanisms have proven extremely hard to get right on one machine let alone in a distributed environment. There is a need for a more robust communication mechanism between services. One alternative to blocking method calls is asynchronous message passing. Introducing this communication mechanism should complement Promises and PushStreams to give OSGi an all-round ability to handle highly concurrent loads.

4 Use Cases

4.1 Digital twins

The IoT “Digital Twin” model is implemented by providing a virtual representation of a real physical device. Users interact with the virtual representation of the device, with the state of the digital twin being synchronized with the physical device. By necessity a digital twin must contain a significant amount of internal state, and it is vital that the internal state remain consistent, even when a large number of users are competing for access to the digital twin. Implementing the digital twin using the actor model can greatly simplify the state machine representing the allowed actions and behaviours of the digital twin. This pattern is being used to help build automated factories and modular manufacturing pipelines.[3].

As OSGi is often used in industrial embedded control systems it would therefore be beneficial for an OSGi developer to implement a digital twin using the Actor Model, restricting access to the Digital Twin so that it can be implemented simply, while still remaining able to access OSGi services provided by other bundles from inside the Actor.

4.2 Asynchronous microservices

In microservice architectures an application is distributed between multiple highly-cohesive loosely coupled services which communicate over the network. Since each service handles a specific sub-domain of the overall this architecture requires high degree of cross-talk between multiple services to implement every use case. One issue often encountered is the implicit tight coupling between services caused by the use of blocking remote calls. The issue is that one service begins a blocking call which causes the next service in turn to do a blocking call and so on. All services have to wait before the final link in the chain completes the request and the “call stack” is unwound. It is increasingly clear that this issue can be effectively addressed by designing services to communicate through asynchronous messages. To support such services there is a need of programming models that allow developers to reason about the state of computation under asynchronous conditions. One such model is the actors model of computation.

It will be beneficial for OSGi to support actors in an enterprise setting. One benefit OSGi can bring to Actor computing is the ability to encapsulate Actor subsystems into bundles exposing and versioning only the public message protocol of the user facing Actors much like it is done with the interfaces of blocking services.

Requirements

The requirements describe the minimal viable actor system. Even though a lot of requirements go beyond the minimal actor execution model they are nevertheless needed to implement practical actor applications.

Still it is highly preferable to use existing OSGi specifications to support as many of the requirements as possible. E.g. Promises are a great candidate to support any interactions of actors with non-actor services. The goal is to have an OSGi system extended with the actor programming model, rather than other way around.

4.3 Execution model

~~This is simply a generic actor execution model not related to OSGi~~

The theoretical actor model of computation [5] is built on four fundamental rules:

- Actors process Messages in sequential steps
- In a step an Actor can send a Message to an Address
- In a step an Actor can create another Actor and receive back it's Address
- In a step an Actor can designate a Behavior to be used to process the next Message

The following requirements describe in greater detail how to support this execution model and do not necessarily include OSGi specifics. Violating any of the MUST requirements renders the actor runtime invalid.

Exec 1: The actor runtime MUST~~must~~ provide at lest one address for each actor

- **Exec 2**: The actor runtime MUST~~must~~ store behaviors as (functional) objects/closures
  - So that they can capture state by closing over effectively-final variables visible in their lexical scope.
- **Exec 3**: The actor runtime MUST~~must~~ guarantee that at all times an actor is associated with exactly one behavior
  - This also implies an actor is atomically initialized with a behavior upon creation
- **Exec 4**: An actor MUST~~must~~ have a way to access it's own address (self) from it's behavior
- **Exec 5**: An actor behavior MUST~~must~~ have a way to send messages to an actor address it has access to
  - Behaviors close over effectively-final actor addresses like any other piece of state.
- **Exec 6**: The actor runtime MUST~~must~~ guarantee asynchronous message delivery.
  - Behavior code MUST~~must~~ not block waiting for the message to be processed by the recipient.
- **Exec 7**: The actor runtime MUST~~must~~ guarantee "at-most-once" message delivery.
  - While it is allowed for messages to not be delivered it is not allowed for the same message to be delivered twice.
- **Exec 8**: It MUST~~must~~ be possible for messages to contain addresses of actors.
- **Exec 9**: An actor MUST~~must~~ process messages only in sequential atomic steps. At each step the actor first calls it's current behavior with the current message and gets back a new behavior. Then the actor replaces it's current behavior with that new behavior.
  - Therefore an actor maintains a continuation passing style of execution driven by messages.
  - It is permissible to execute each step on a different thread (e.g. from a pool), but it is not under any conditions permissible for more than one ~~two or more~~ thread~~s~~ to execute steps for a given actor concurrently. ~~*Violating this principle renders the actor system unusable.*~~
  - This still allows users to have a mutable behavior object, which always returns itself, thus enabling an "object oriented" style of actor.

- **Exec 10**: The actor runtime MUST~~must~~ not deliver messages to a behavior which is no longer current for an actor

- **Exec 11**: It MUST~~must~~ be possible for a behavior to return a value which signals no more messages will~~should~~ be processed. After this happens the actor runtime must stop delivering messages to the actor and perform any post-death actions (see supervision and “death watch”) below.

- **Exec 12**: An actor behavior MUST~~must~~ have a way to create a new actor by specifying an initial behavior for that actor. The creation call must return **synchronously** the address of the new actor.

- **Exec 13**: Messages exchanged by Actors MUST support type-safe data access. It is not acceptable for the programming model to provide only Map as an exchange type.

4.4 Supervision hierarchy and e~~E~~rror handling

Violating any of the MUST requirements renders the actor runtime invalid.

- **Err 1**: Each actor MUST~~must~~ have exactly one supervisor. The supervisor is always the actor who created the supervised (child) actor.

- **Err 2**: The supervisor MUST be~~is~~ established once during actor creation and can not be changed subsequently.

- **Err 3**: There MUST~~must~~ be exactly one root actor which has no supervisor.

- **Err 4**: There MUST~~must~~ be a way for a behavior to specify a supervisor strategy function when it creates a child actor.

  - The function is similar to a catch block that handles failures of the new actor

- **Err 5**: An actor MUST~~must~~ process the errors of it’s children sequentially as special messages within the regular message stream. The difference is that the error message is delivered to the parent’s supervisor strategy function rather than to the parent’s behavior.

  - The definition of an actor is thus expanded to include not just the current behavior function, but also the set of supervisor strategy functions it has attached to it’s children. Each member function of the actor processes the next message (error or regular respectively) as appropriate. No message is processed by more than one member function.

- **Err 6**: The following return values from the supervisor strategy function MUST~~must~~ be supported:

  - **resume**: the failed actor retains it’s behavior. It then proceeds to process the next message.

  - **reset**: the failed actor is reset to the behavior with which it was created. It will then proceed to process the next message.

  - **stop**: the failed actor is stopped. It can no longer process messages and is garbage collected.

  - **escalate**: the supervising actor is stopped. The error is propagated up the hierarchy to it’s supervisor.

- **Err 7**: When a supervisor is stopped all of it’s children MUST be~~are~~ stopped first.

4.5 Core capabilities

Not part of the core execution model, hard or impossible to implemented on top of the core execution model, so must be supported by the actor runtime instead.

- **Core 1**: Message ordering. For any pair of a sender actor and a receiver actor the order in which the messages are sent MUST be the same as the order in which messages are received. ~~the same message orderobserveSender and receiver~~
- **Core 2**: Death watch: an actor MUST be able to~~can~~ register an address in order to receive a message signifying that the actor referenced by this address has died.
  - This allows actors to clean invalid addresses they have
  - This allows actors to stop when the other actor was critical to their function, even though they are not it's supervisor
- **Core 3**: Message buffering. The actor runtime MUST~~**must**~~ be able to buffer (rather than drop) messages sent to an address while the actor targeted by the message executes it's current behavior.
- **Core 4**: Scheduled message delivery. It MUST~~**must**~~ be possible for an actor to schedule one-shot or periodic delivery of a message to an arbitrary address (including "self") with a given time interval.
- **Core 5**: Pub/Sub message delivery. The actor runtime SHOULD~~**should**~~ provide a global pub/sub subsystem where actors can subscribe for messages of a given type and respectively post messages.
  - Simplifies the support of callback patterns, by removing the need for initial registration
- **Core 6** (?): Dead letters: subscribe (using **Core 5**) to messages sent to addresses of dead actors
- **Core 7**: Logging: the actor runtime MUST~~**must**~~ provide behaviors with a logger that does not block execution ~~until the log is fully processed (e.g. committed to a file or printed on the console)~~

4.6 Core patterns

Can be implemented on top of the core execution model and capabilities as utilities, base classes, etc. or if preferable can be built into the actor runtime.

- **Pat 1**: Message deconstruction: there MUST be a way to examine a message object that reduces the need of "instance of" and "if" statements to a minimum.~~pattern matchingsomething to help with~~
  - Non-Java actor runtimes do this with "pattern matching" [4].
- **Pat 2**: "Ask" (Request/Response) message exchange: MUST be supported
  - Actor sends a request and starts a timer to send a failure message to "self".
  - If response arrives before the failure message the failure message is ignored.
  - If response arrives after failure message the response is ignored.
  - Similar to a Promise: the failure message and the response message race to resolve the promise. Once resolved no further updates can take effect.

- **Pat 3**: “At-least-once” message delivery: SHOULD be supported:
  - Rrequires message equality, receiver acknowledges messages, sender tracks outgoing messages (to resend unacknowledged)
  - The tracking of acknowledgments may be supported by persistence (see the Persistence section)
- **Pat 4**: “Exactly-once” message delivery: SHOULD be supported
  - R: requires message equality, receiver acknowledges messages, sender tracks outgoing messages (to resend unacknowledged), receiver tracks incoming messages (to ignore repeats)
  - The tracking of acknowledgments and duplicates may be supported by persistence (see the Persistence section)

4.7 OSGi integration

- **Int 1:** As a general rule actors SHOULDshould be treated more as a concurrency model, rather than as the default programming model. The structure of the program SHOULDshould be expressed first as standard OSGi service composition as much as possible (e.g. DS components). Then some services may have message oriented APIs and run actors to support them.
- **Int 2**: As a general rule all communication from the external world to the actor system MUST**must** be asynchronous.
  - In practice this means external parties should be able to send messages to an address provided by the actor system, just like regular actors do.
- **Int 3**: As a general rule all communication from the actor system to the external world SHOULD**should** be asynchronous.
  - In practice this means actors should wrap calls to the outside world in Promises, which once resolved send messages back to “self” as a regular external party (**Int 2)**. The Promises should be scheduled on threads that do not drive the actor message delivery (bulk-heading).
- **Int 4**: It MUST**must** be possible to initialize behaviors with arbitrary OSGi services prior to the first message being processed.
- **Int 5**: It MUST**must** be possible to initialize behaviors with configuration provided by the Configuration Admin prior to the first message being processed
- **Int 6**: It MUST**must** be possible to stop any actor when it’s configuration or a service dependency goes away
  - This MUST**must** happen asynchronously via messages or other native actor mechanisms (death watch).
  - Still the actor MUST**must** be prevented from calling stale services even though it will observe the service unregistration after the fact due to asynchrony. Since the actor should call the service through Promises as per **Int 2** one way is to fail those Promises immediately and let the actor run into the failure later within it’s regular message flow.

- **Int 7**: It SHOULD~~should~~ be possible for supervisors to specify an “escalate” strategy when a child starts to fail because of an unbound service / configuration.
    - In practice this means there must be a dedicated exception or error message to which the supervisor strategy function can react.
- **Int 8**: It MUST~~must~~ be possible for any bundle to call the actor system to discover actor addresses
- **Int 9**: It MUST~~must~~ be possible for any bundle to send a message to an actor address asynchronously
- **Int 10**: It MUST~~must~~ be possible for any bundle to do an asynchronous request/response exchange with an actor
    - In practice this means the external party uses the “ask” pattern (**Pat 2**).
- **Int 11**: It MUST~~must~~ be possible for any bundle to subscribe to messages or to publish messages (**Core 5**)
    - Relates to the pub/sub optional core capability
- **Int 12**: It MUST~~must~~ be possible for a bundle to create an actor and receive back it’s address

4.8 Persistence

Although actors can implement persistence by calling a standard OSGi service, it will be hard for developers to implement a behavior that guarantees consistency on one hand, but does not block on the other (e.g. for transactions).

This is why it is preferable that a safe behavior is supported directly by the actor system or as a core pattern. ~~In practice the pattern is likely to be implemented by instantiating a small system of actors~~.

- **Pers 1:** An actor MUST~~must~~ be able to receive, examine and update a persistent state object during message processing
- **Pers 2:** It MUST~~must~~ be possible to identify an actor’s persistent state across system reboots
- **Pers 3:** It MUST~~must~~ be possible to initialize an actor not only with a behavior, but with initial state.
- **Pers 4:** It MUST~~must~~ be possible for a behavior to request commit to storage before it executes any side effects (creating actors, sending messages, calling services)
    - One would expect applying persistence to cause messages to be handled as follows:
        - *(behavior) examine message and state →*
        - *(behavior) execute side effects →*
        - *(behavior) build new state →*
        - *(system) commit new state to storage*

- In this case it is possible for the actor to first affect the runtime state of the system, then for the system to crash before the state is committed. The system can't be recovered to a state that reflects what the actor has done.

- Instead applying persistence MUST~~must~~ cause messages to be handled as follows:
  - *(behavior) examine message and state* →
  - *(behavior) build new state* →
  - *(system) commit new state* to storage →
  - *(behavior) execute side effects*
  - In this case if the system crashes during commit one message will be lost, but the last committed state will remain valid since nothing else was done. On the other hand if the system crashes right after commit, but before the side effects are completed, once recovered the actor can re-run the side effects based on the last stored state.
- The pattern implies the behavior function should be split into two sub-functions:
  - "command handler" with signature**:** *(message, state)* → *state*
  - "effects handler" with signature**:** *state* → *void*
- A message is then processed as follows:
  - Call "command handler" with *(message, current-state)* and get *next-state*
  - Commit *next-state* to storage
  - Call the "effects handler" with *next-state* to execute any runtime effects
  - Return a next Behavior which encloses *next-state, command handler, effects handler*
- "Commit *next-state*", MUST~~must~~ be executed asynchronously
- Any messages that arrive while the actor is in the "Commit *next-state*" MUST~~must~~ be buffered (**Core 3**) until the entire sequence is completed

- **Pers 5**: It MUST~~must~~ be possible to re-run the behavior side effects when an actor is loaded from storage
  - I.e. run the "*effects handler*" from **Pers 4** on actor recovery.

4.9 Clustering

This is needed to implement classic actor "distributed applications", where the same tightly coupled code base is split across many VMs, specialized protocols to move messages between VMs are used, dedicated cluster discovery is used.

This is not needed to implement "distributed systems", consisting of multiple (not one) loosely coupled applications talking through generic protocols (e.g. REST, gRPC, MQTT).

This is why this section should be considered optional. Still within the scope of the section the usual meaning of MUST, SHOULD, MAY apply.

- **Clust 1:** An Actor MUST be able to send a message to an address representing a remote actor~~ssages to a remote actor addressProvide a way to send me~~
- **Clust 2:** An Actor MUST have a way to resolve a search query to an address of remote or local actor~~Provide a way to discover a remote or local actor address~~
  - I.e. call the system with some search query and get back an actor address.
  - This may be needed to support Persistence, since the stored state may have to include persistent IDs of actors that need to be resolved to addresses on reboot
- **Clust 3**: It MUST be possible to design messages so that they can be marshaled across the network.
- ~~**Clust 3(?):** Use the RSA to distribute actor services~~

5 Document Support

5.1 References

[1]. Bradner, S., Key words for use in RFCs to Indicate Requirement Levels, RFC2119, March 1997.

[2]. Software Requirements & Specifications. Michael Jackson. ISBN 0-201-87712-0

[3]. Digital Twin: Manufacturing Excellence through Virtual Factory Replication. Dr Michael Grieves. https://research.fit.edu/media/site-specific/researchfitedu/camid/documents/1411.0_Digital_Twin_White_Paper_Dr_Grieves.pdf

[4]. Pattern Matching: https://en.wikipedia.org/wiki/Pattern_matching

[5]. Actor Model: https://en.wikipedia.org/wiki/Actor_model

Add references simply by adding new items. You can then cross-refer to them by chosing <Insert><Cross Reference><Numbered Item> and then selecting the paragraph. ***STATIC REFERENCES (I.E. BODGED) ARE NOT ACCEPTABLE, SOMEONE WILL HAVE TO UPDATE THEM LATER, SO DO IT PROPERLY NOW.***

5.2 Author's Address

Name	Todor Boev
Company	Software AG
Address	
Voice	
e-mail	todor.boev@softwareag.com

Name	Tim Ward
Company	Paremus Ltd
Address	
Voice	
e-mail	tim.ward@paremus.com

5.3 End of Document