orleans

This document outlines the purpose and the application of the Distribute Actor Model in Emailage's Email Risk Score API using Microsoft Orleans.

distributed systems

A distributed system is a system whose components are located on different networked computers, which communicate and coordinate their actions by passing messages to one another from any system.[1]

A good example of a distributed system is a group of containerised applications:

- web APIs:
  - placed behind a load balancer
  - serving customer traffic
- supporting applications

where applications communicate with a group of data sources and with each other via message queues or direct messages, perform business decisions, report on their health, etc.

traditional 3-tier microservices and their challenges

Microservices are often characterised by:

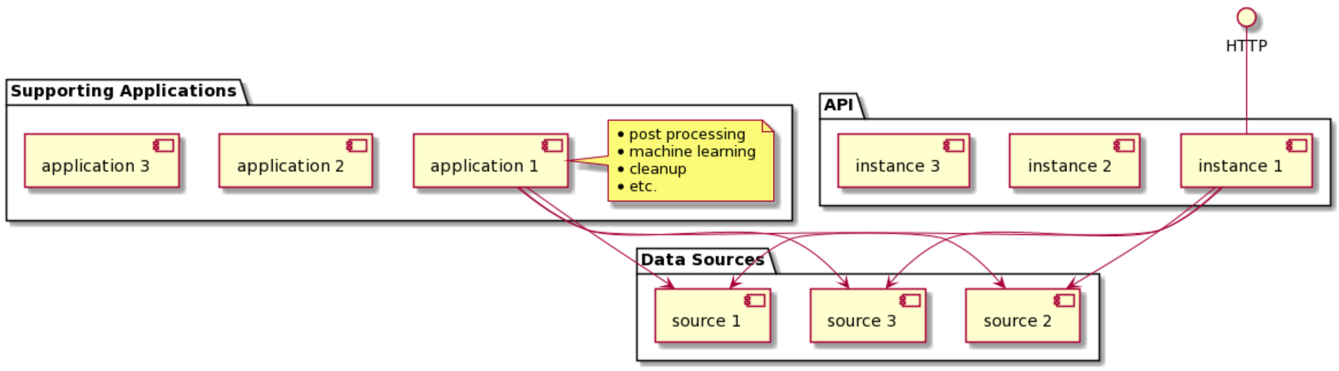
- their lack of internal state
- application instances being expendable - relatively simple to spin up and to tear down

which makes it fair to categorise microservices as distributed stateless applications.

While microservices allow engineers to easily scale distributed systems (both horizontally and vertically), they also present a number of challenges which become especially apparent when mission-critical speed is essential, namely:

- scaling - the ability to correctly respond to sudden changes in traffic volumes (it is not uncommon to see 10x traffic spikes in Emailage APIs). Application instances should be quick to wake up, warm up and start working with load balancers in order to serve traffic.
- request processing speed - the API should ideally be able to perform its tasks in predictable, relatively constant time.
- performance bottlenecks - if the application has a performance bottleneck (e.g. one task requiring excesive amounts of resources), scaling the application horizontally will just replicate those performance bottlenecks across the fleet of application instances.
- implementation speed - deployment, maintenance, inter-service communication, testing complexity [3]
- storage is often a bottleneck, efficient data shipping becomes a problem[4].

In simplest possible terms, 3-tier microservices can be illustrated with the following diagram:



Fore the sake of clarity, only one component from each group (instance 1, application 1) have visible connections to other groups.

actors

The Actor Model is a mathematical theory of computation that treats “Actors” as the universal primitives of concurrent digital computation [2]. The model has been used both as a framework for a theoretical understanding of concurrency, and as the theoretical basis for several practical implementations of concurrent systems. Recently, the Actor Model gained new popularity with the advent of cloud-based computing where:

- massive, distributed parallelism
- ability to quickly analyse and understand distributed systems by humans

are of critical importance.

In the simplest terms, Actors are characterised by two features:

- they don't share state - each actor is an independent entity with its own memory
- they only communicate indirectly via messages allowing for greater decoupling

Using the Actor Model in distributed systems allows for breaking down complex, resource-intensive problems into more manageable, focused fragments and sharing them between the components of a distributed system.

Popular languages adopting the Actor Models include:

- Erlang
- Elixir
- Scala
- P#, later Coyote
- Dart

virtual actors

With Project Orleans, the term "virtual actor" is introduced[6]:

Virtual Actor - adaptation of the Actor Model for challenges of the Cloud.[5]

This high level description translates into an improved version of the Actor Model characterised by four features[7]:

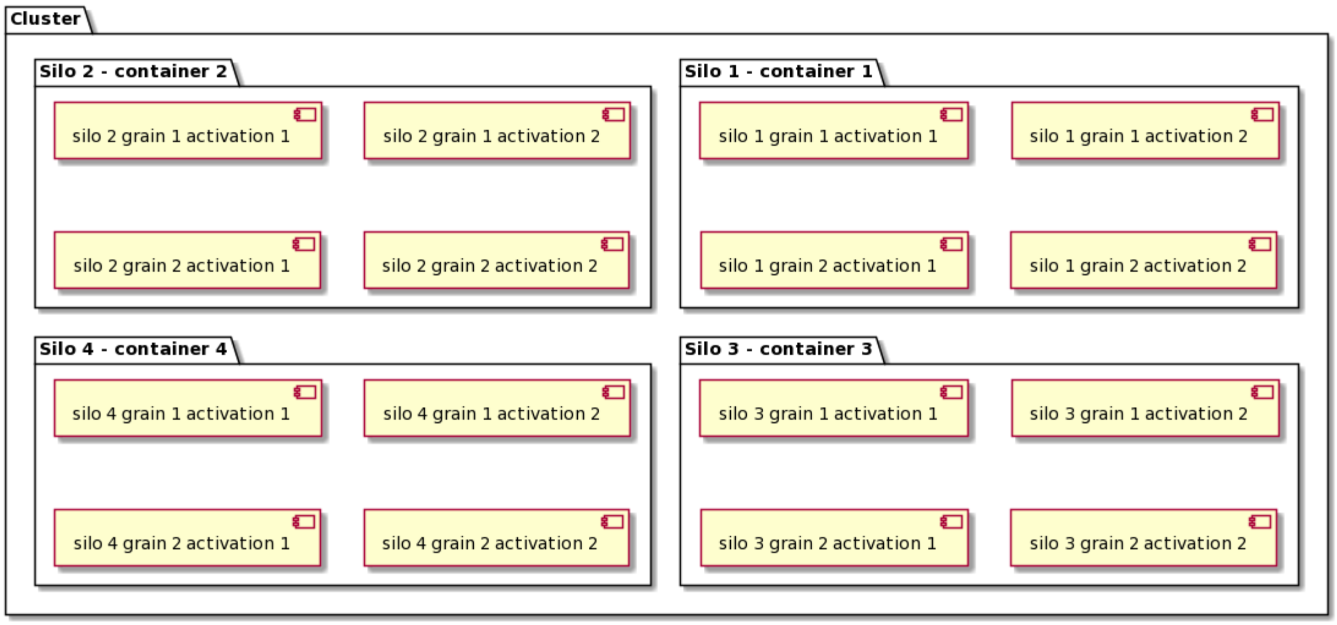
1. **Perpetual existence** - actors are purely logical entities that always exist, virtually. An actor cannot be explicitly created or destroyed and its virtual existence is unaffected by the failure of a server that executes it. Since actors always exist, they are always addressable.
2. **Automatic instantiation**: Orleans' runtime automatically creates in-memory instances of an actor called activations.
3. **Location transparency**: an actor may be instantiated in different locations at different times, and sometimes might not have a physical location at all.
4. **Automatic scale out**: currently, Orleans supports two activation modes for actor types: single activation mode (default), in which only one simultaneous activation of an actor is allowed, and stateless worker mode, in which many independent activations of an actorare created automatically by Orleans on-demand (up to a limit) to increase throughput.

In short: *Project "Orleans" is a programming model and runtime for building cloud native services.*[5]

Orleans-specific terminology:

- **grain** - virtual actor (implemented as a .NET class)
- **activation** - instace of a grain (in the OOP terms, if grains are classes, activations are their objects)
- **silo** - actor host (implemented as a .NET process, optionally hosted in a container)
- **cluster** - group of silos working together and sharing grain activations (implemented as a group of e.g. containers). Orleans cluster is only a logical grouping (cluster membership is tracked either in silos' memory or in persistent storage like DynamoDB).

Orleans cluster can be illustrated as a following group of components working together:


Cluster
Silo 2 - container 2
silo 2 grain 1 activation 1
silo 2 grain 1 activation 2
silo 2 grain 2 activation 1
silo 2 grain 2 activation 2
Silo 1 - container 1
silo 1 grain 1 activation 1
silo 1 grain 1 activation 2
silo 1 grain 2 activation 1
silo 1 grain 2 activation 2
Silo 4 - container 4
silo 4 grain 1 activation 1
silo 4 grain 1 activation 2
silo 4 grain 2 activation 1
silo 4 grain 2 activation 2
Silo 3 - container 3
silo 3 grain 1 activation 1
silo 3 grain 1 activation 2
silo 3 grain 2 activation 1
silo 3 grain 2 activation 2

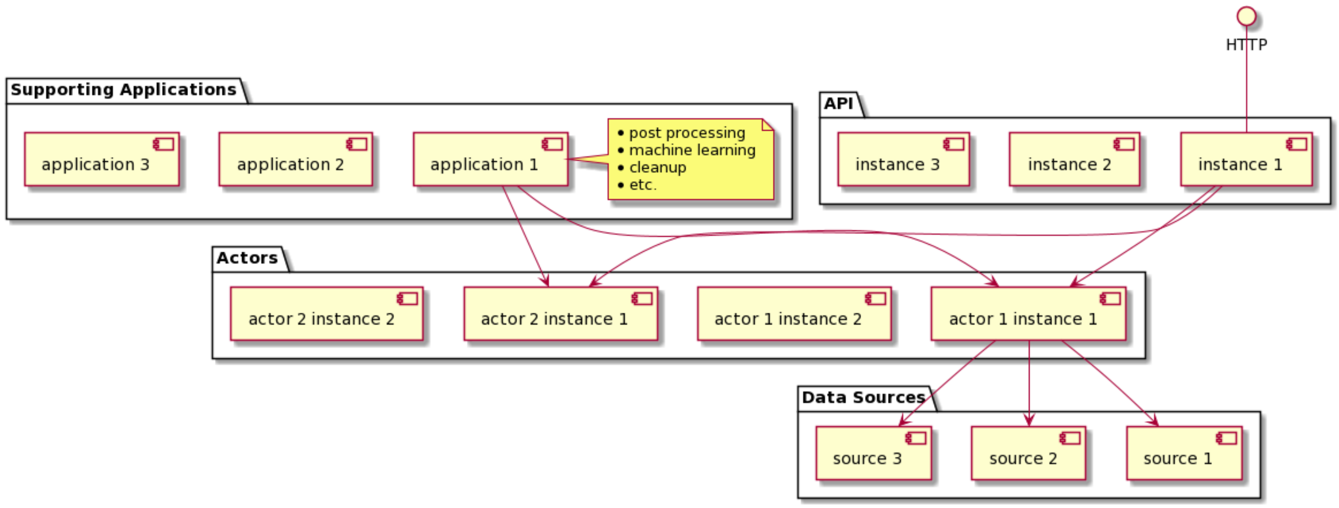
However, in reality, we can have hundreds of silos participating in each cluster and hundreds of thousands of grain activations participating in every silo.

actors in distributed stateless and stateful applications

The goal is to cleanly divide two aspects of every application:

- I/O - any database, service calls introducing uncertainty (latency, failure, etc.)
- state and pure (no side effects) state manipulations: mapping, filtering, business decisions, etc.
- API tasks are focused on what they should be focused - on handling customer requests
- stateful - potential for caching
- more granular horizontal scalability
- homogenous or heterogenous clusters
- all systems become much easier to reason about
- code becomes very reusable
- significant number of API tasks saved and can be dedicated to processing more customer requests
- distributed runtimes
- if the work is partitioned cleverly, Actors can give a tremendous performance boost to the API
- supporting applications can also use actors - better code reusability
- fast deployments (heterogenous silos), fast scaling (no warmup, internal load balancing and infrastructure state management)
- even if actors are stateless, they still provide value by the way they allow the caller to delegate the workload to distributed actors

In simplest possible terms, microservices working with actors can be illustrated with the following diagram:



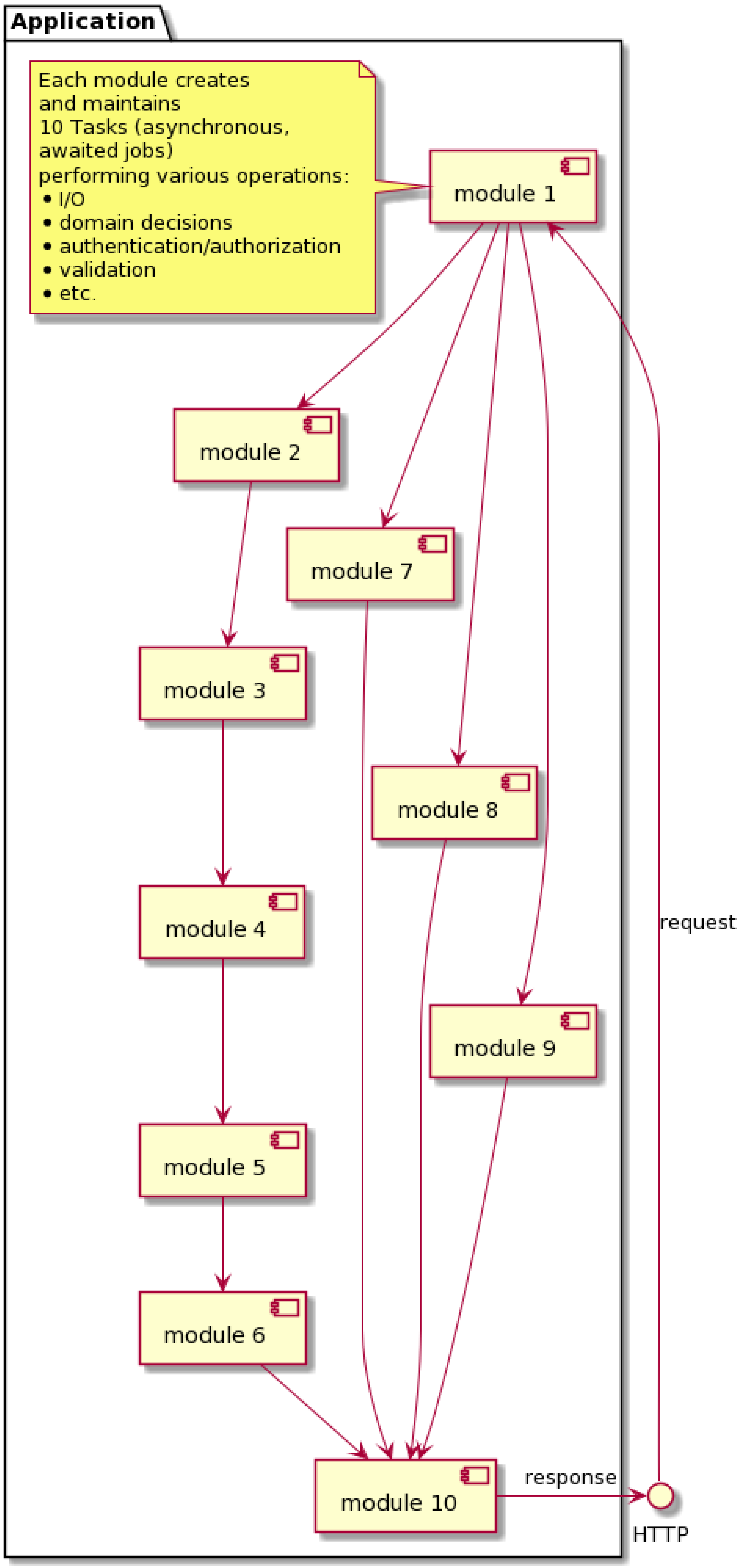
Fore the sake of clarity, only one component from each group (instance 1, application 1, actor 1 instance 1) have visible connections to other groups.

examples

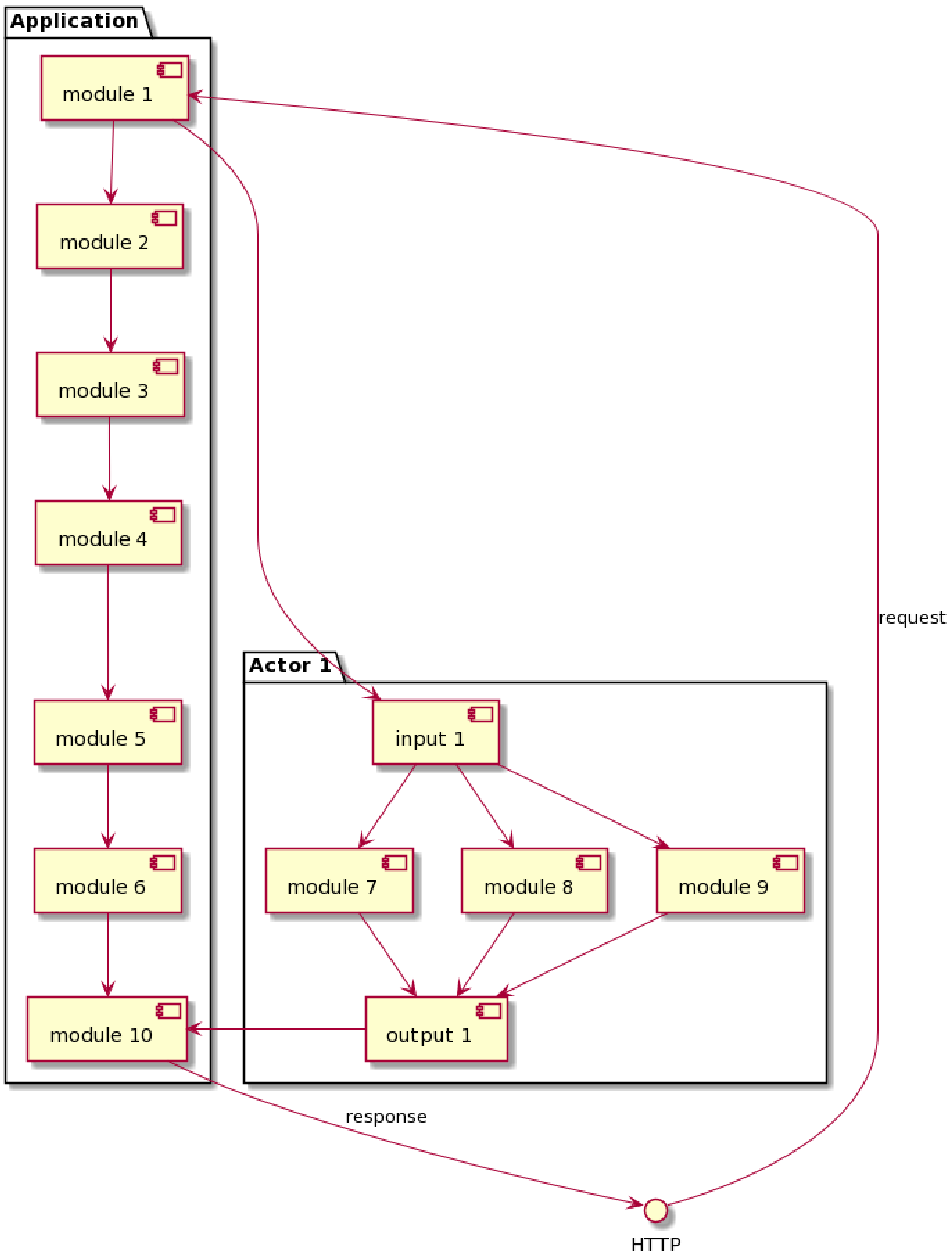
theoretical example 1

scenario

current state


Application
Each module creates
and maintains
10 Tasks (asynchronous,
awaited jobs)
performing various operations:
• I/O
• domain decisions
• authentication/authorization
• validation
• etc.
module 1
module 2
module 3
module 4
module 5
module 6
module 7
module 8
module 9
module 10
request
response
HTTP

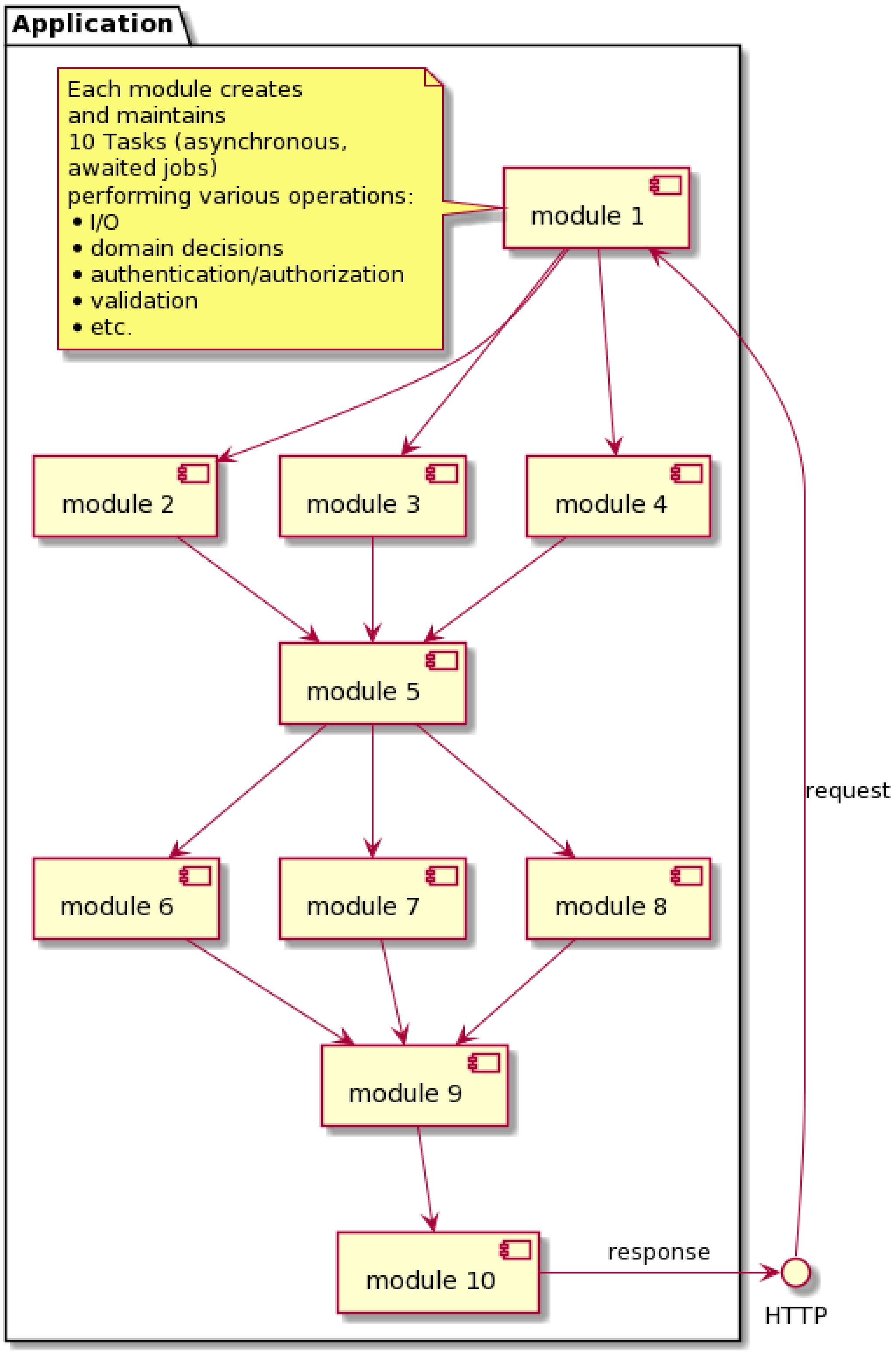
proposed changes



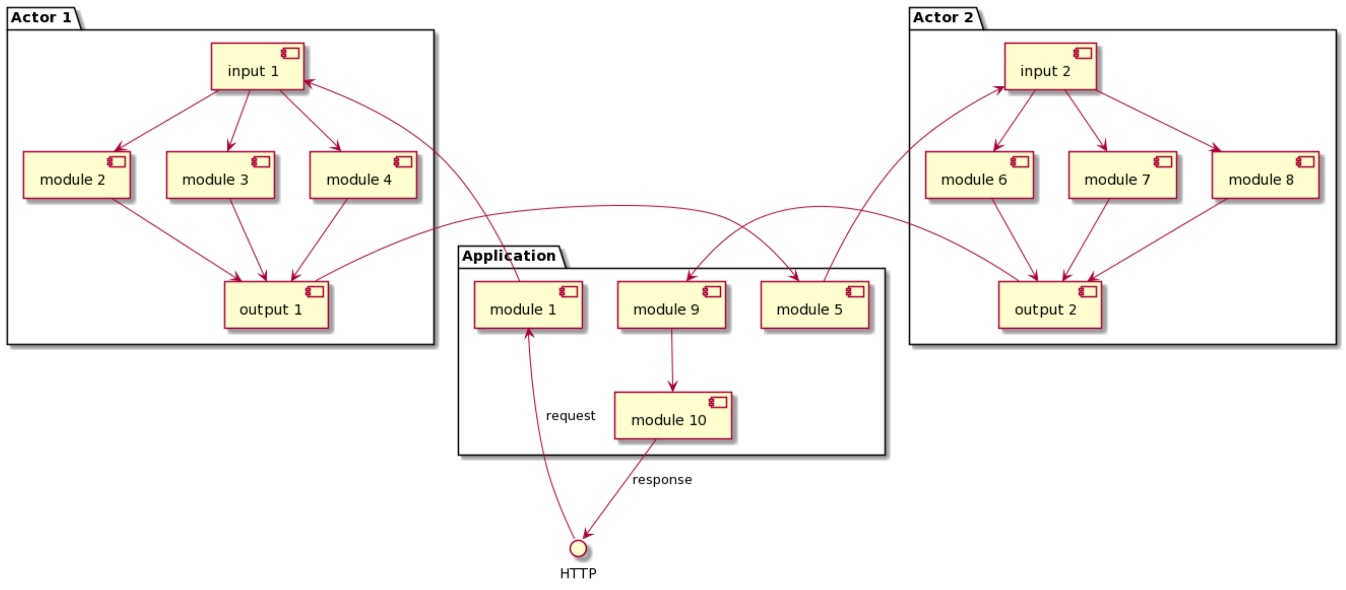
theoretical example 2

scenario

current state



proposed changes



practical example 1

practical example 2

lessons learned

generous memory allocation autoscaling criteria pure code grain testing is not easy, they should be just very thin wrappers

references

1 - Distributed systems: principles and paradigms.

Tanenbaum, Andrew S.; Steen, Maarten van (2002). Distributed systems: principles and paradigms. Upper Saddle River, NJ: Pearson Prentice Hall. ISBN 0-13-088893-1.

2 - A Universal Modular Actor Formalism for Artificial Intelligence

Hewitt, Carl; Bishop, Peter; Steiger, Richard (1973). "A Universal Modular Actor Formalism for Artificial Intelligence". IJCAI.

3 - Pattern: Microservice Architecture

https://microservices.io/patterns/microservices.html

4 - Actors for High-Scale Services

http://www.hpts.ws/papers/2013/Bykov.pdf

5 - Project Orleans: Distributed Virtual Actors for Programmability and Scalability

https://www.microsoft.com/en-us/research/uploads/prod/2016/02/philbe-disckeyotephilbefinal.pdf

6 - Orleans – Virtual Actors

https://www.microsoft.com/en-us/research/project/orleans-virtual-actors/?from=http%3A%2F%2Fresearch.microsoft.com%2Fprojects%2Forleans%2F

7 - Orleans: Distributed Virtual Actors for Programmability and Scalability

https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/Orleans-MSR-TR-2014-41.pdf

additional resources

- Orleans source code: https://github.com/dotnet/orleans
- Orleans documentation: https://dotnet.github.io/orleans/
- Coyote source code: https://github.com/microsoft/coyote/
- Actor Model of Computation: Scalable Robust Information Systems: https://arxiv.org/ftp/arxiv/papers/1008/1008.1459.pdf
- Orleans best practices: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/Orleans20Best20Practices.pdf
- Orleans Gitter: https://gitter.im/dotnet/orleans?at=5deaf4829319bb5190f24ffe
- Road to Orleans - a series of practical Orleans examples: https://github.com/PiotrJustyna/road-to-orleans