Моя жизнь с ~~акторами~~ ~~актрисами~~ актерами
Опыт внедрения модели актеров на F#

ВАГИФ АБИЛОВ, MILES

Коротко о себе

Работаю в норвежской компании Miles

Электронная почта: vagif.abilov@gmail.com

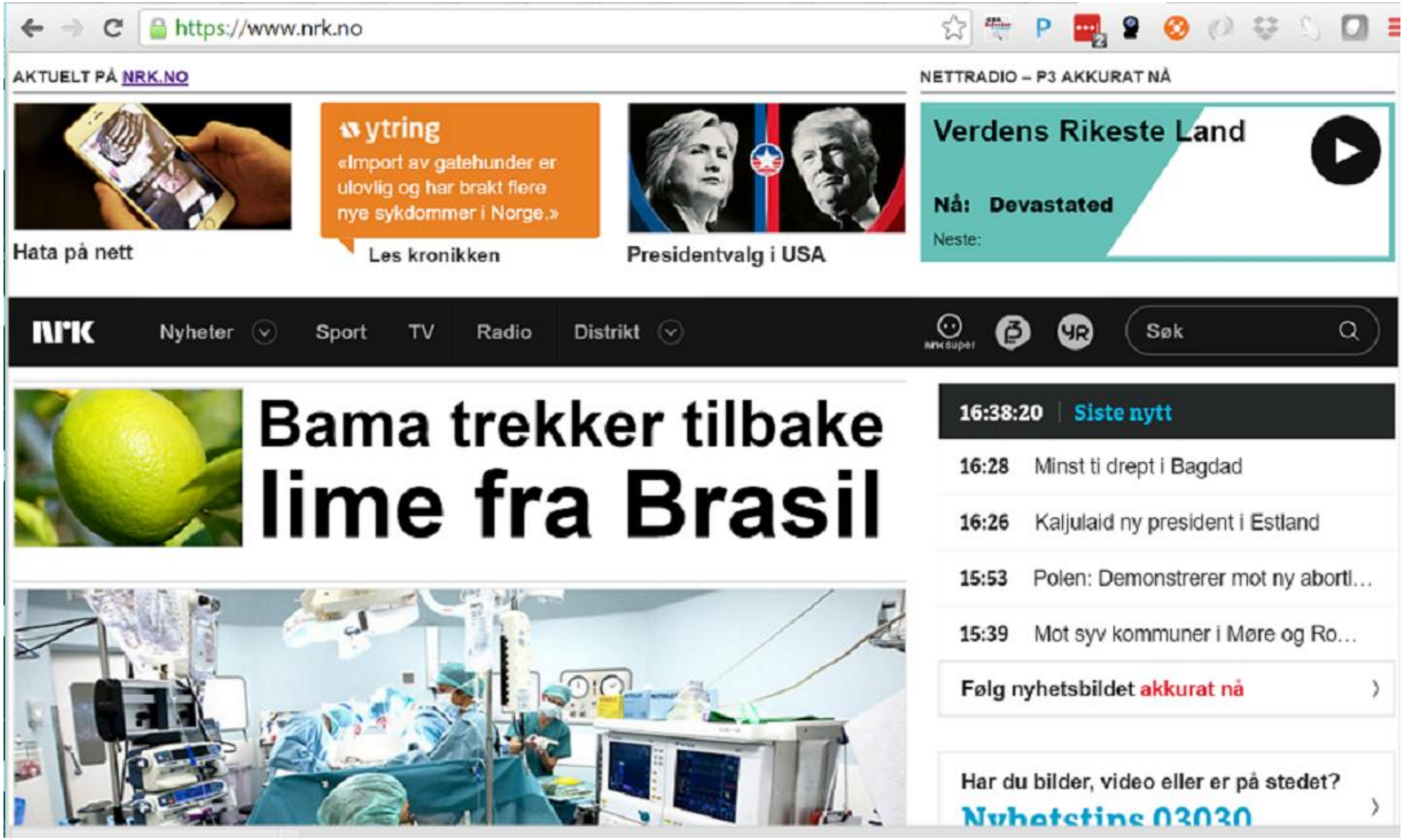
Twitter: @ooobject

GitHub: object

Блог: http://vagifabilov.wordpress.com/

Статьи: http://www.codeproject.com

Поддерживаю библиотеку Simple.OData.Client, участвую в других опенсорсных проектах


https://www.nrk.no
AKTUELT PÅ NRK.NO
NETTRADIO – P3 AKKURAT NÅ
Hata på nett
ytring
«Import av gatehunder er ulovlig og har brakt flere nye sykdommer i Norge.»
Les kronikken
Presidentvalg i USA
Verdens Rikeste Land
Nå: Devastated
Neste:
NRK
Nyheter
Sport
TV
Radio
Distrikt
Søk
Bama trekker tilbake
lime fra Brasil
16:38:20
Siste nytt
16:28 Minst ti drept i Bagdad
16:26 Kaljulaid ny president i Estland
15:53 Polen: Demonstrerer mot ny abortl…
15:39 Mot syv kommuner i Møre og Ro…
Følg nyhetsbildet akkurat nå
Har du bilder, video eller er på stedet?

NRK

Norsk Rikskringkasting (Норвежская вещательная компания)

Основана в 1933-м

Первая радиопередача – 1925 год

Первая телепередача – 1960 год

Более миллиона теле- и радиопрограмм в цифровом архиве

Дистрибуция медиафайлов

Медиафайлы поступают из различных источников:

- Записи прямых трансляций
- Видео, снятые журналистами на камеры и мобильные телефоны
- Полученные по контракту от правообладателей

Медиафайлы транскодируются с различным качеством для оптимального воспроизведения на различных устройствах

Доступ к медиаконтенту

Доступ к медиаконтенту производится на основе времени действия прав на трансляцию

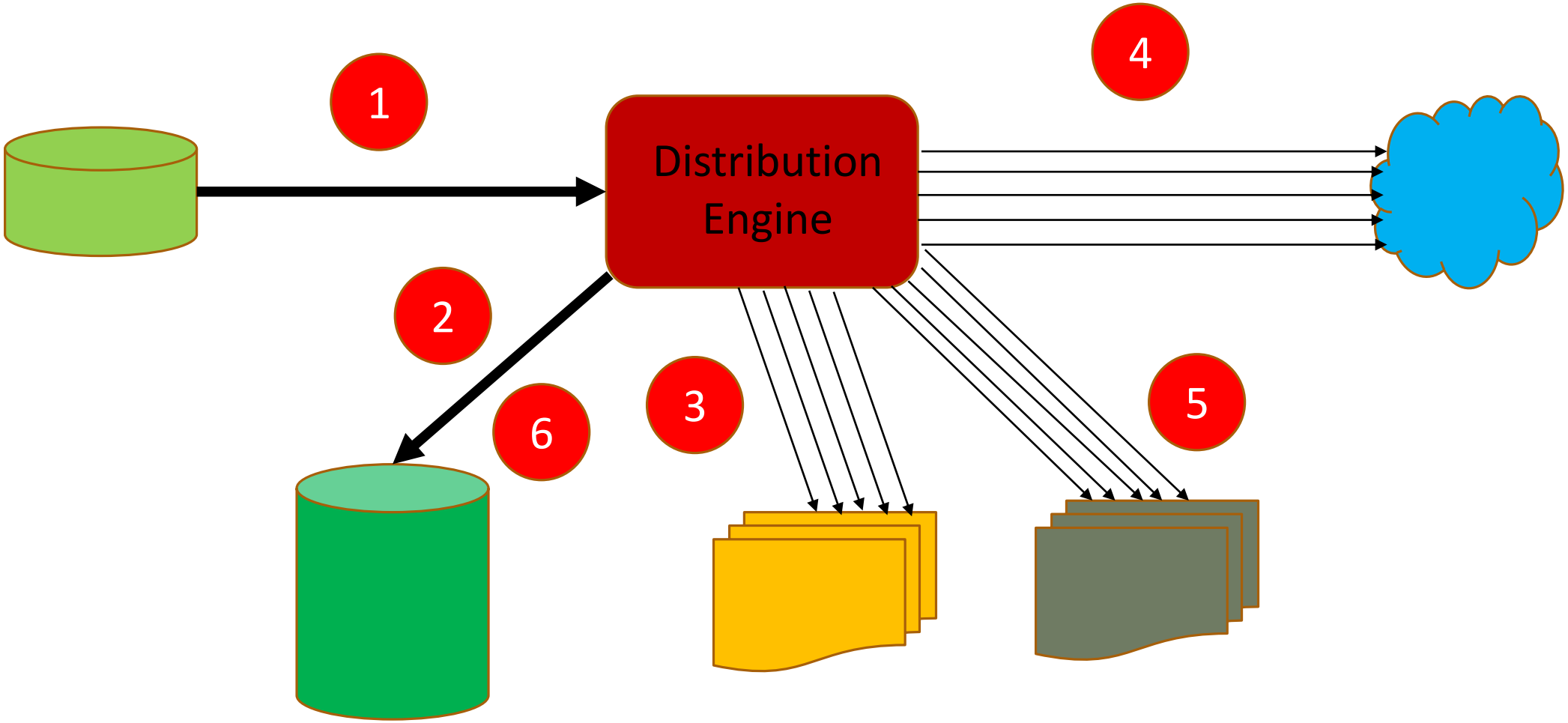
Медиафайлы закачиваются в облачные хранилища с информацией о региональных ограничениях воспроизведения:

- Корпоративное (для внутреннего пользования)
- Региональное (только на территории Норвегии)
- Неограниченное

Наш проект

Медиадистрибутор

Media Distribution Engine


1
Distribution Engine
4
2
6
3
5

Как файлы обрабатывались раньше?

Система первого поколения разработана в 2010-2012 годах

Написана на C# в духе принципов SOLID

- .NET 4.0
- Oracle Entity Framework provider (Devart dotConnect)
- RabbitMQ
- Rebex FTP
- StructureMap

Нужно было ее переписывать?

План выступления - мотивация

Почему нужно было переписывать?

Почему модель актеров и Akka.NET?

Почему F#?

План выступления - выученные уроки

Akka.NET F# API и конечные автоматы

«Говори, не спрашивай» означает «Говори и слушай»

Как мы научились не бояться исключительных состояний и полюбили супервайзоры

Ни SQL, ни NoSQL: на помощь приходят персистентные актеры

Масштабирование актеров

План выступления - reF#actoring

Пример переписывания актера с C# на F#

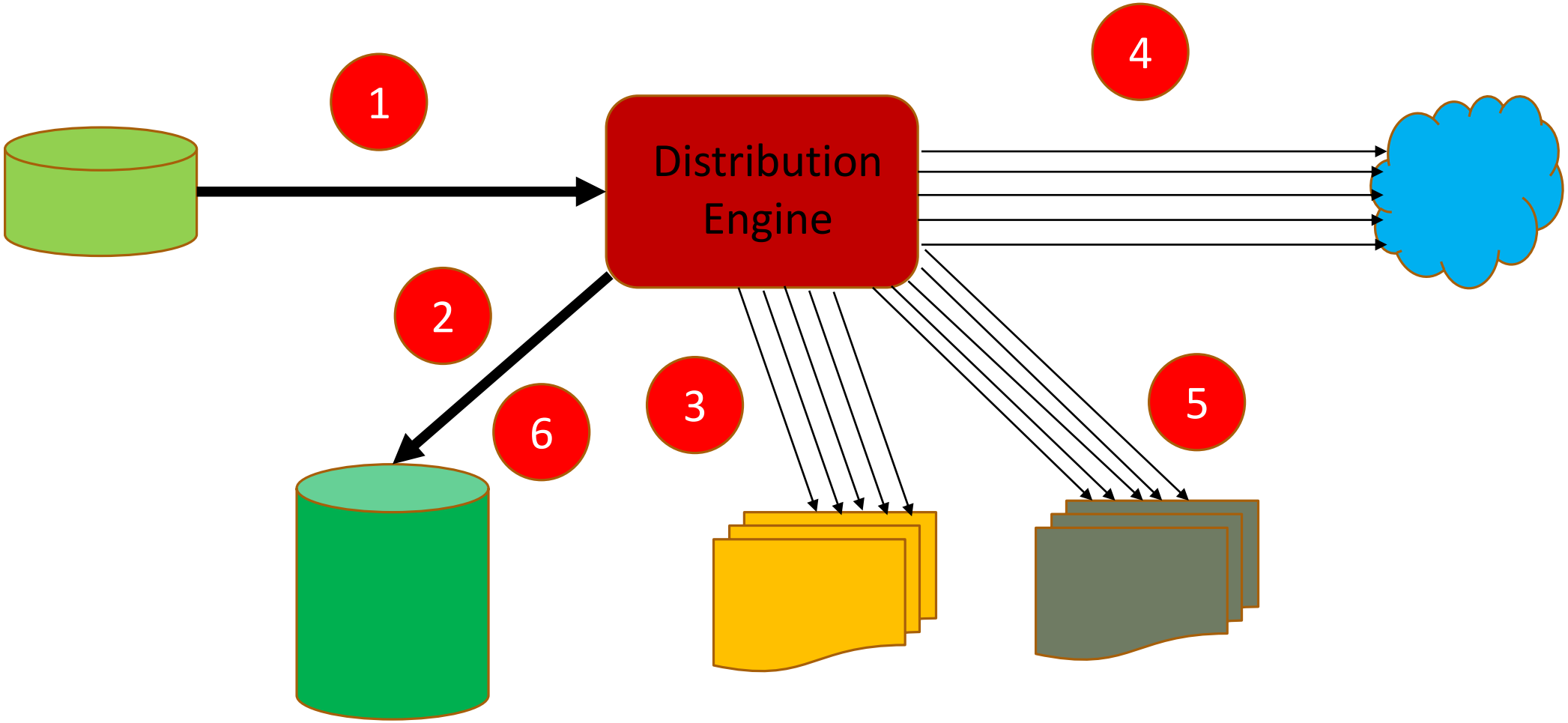
План выступления - заключение

Актеры –не «серебряные пули»

Почему нужно было переписывать?

~~Because F*ck You, That’s Why!~~

Нет, наш выбор не был движим ~~исключительно~~ технологическим интересом


1
Distribution Engine
4
2
6
3
5

Функциональные изменения

Старая система была написана под конкретного облачного провайдера, сегодня NRK работает с несколькими провайдерами облачных хранилищ данных

Изменение протокола файлообмена (переход с FTP на HTTP)

Поллинг баз данных заменен на очереди сообщений

Изменение принципов дизайна системы

Отказ от ручного управления потоками (threads)

Отказ от изменяемости (mutability) внутренних данных

Строгая изоляция компонент

Замена моделирования по принципам ООП на модели, основанные на обработке сообщений (message driven design)

Использование функциональных трансформаций

Модель актеров за три минуты

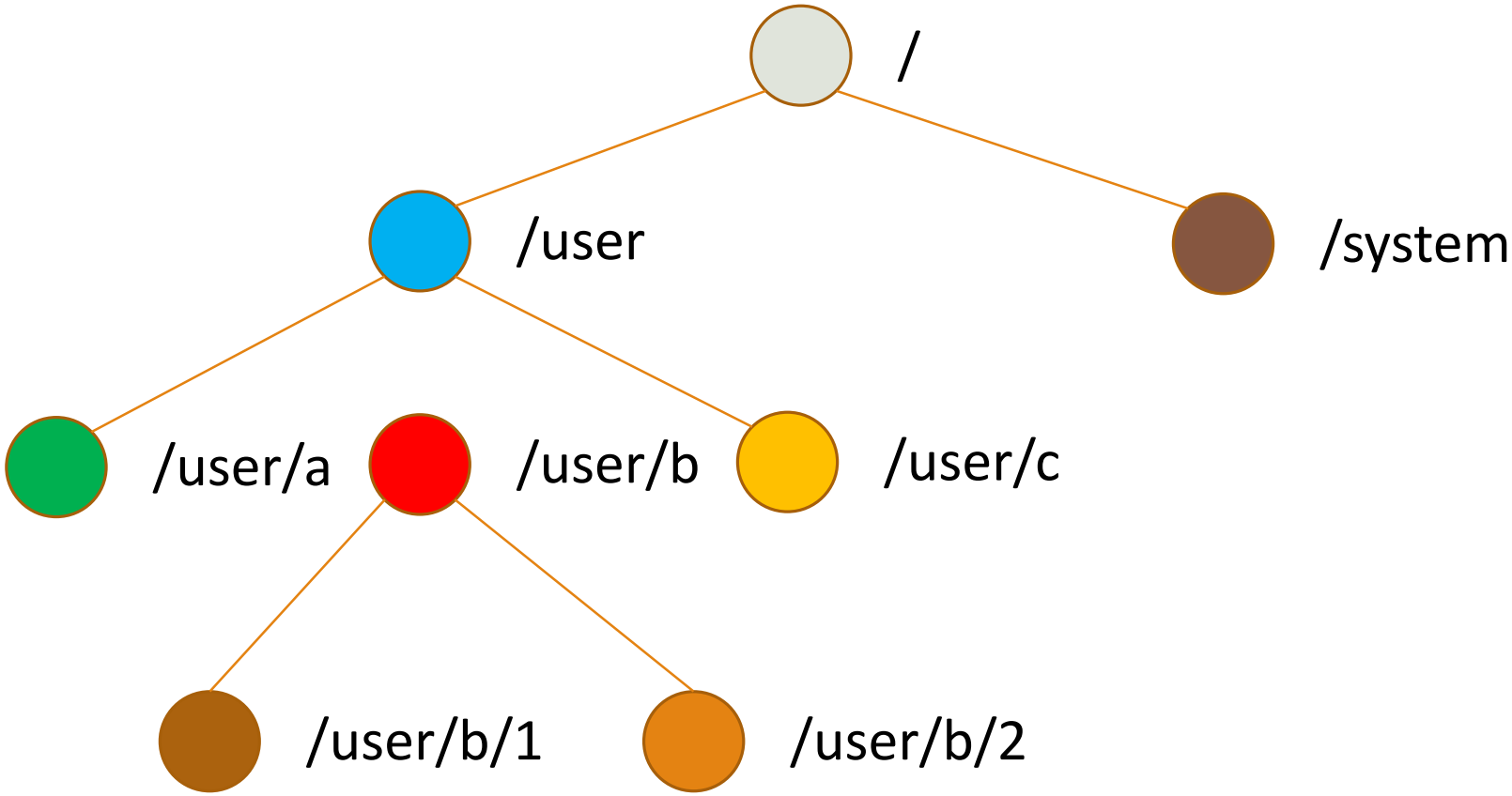
Актер может:

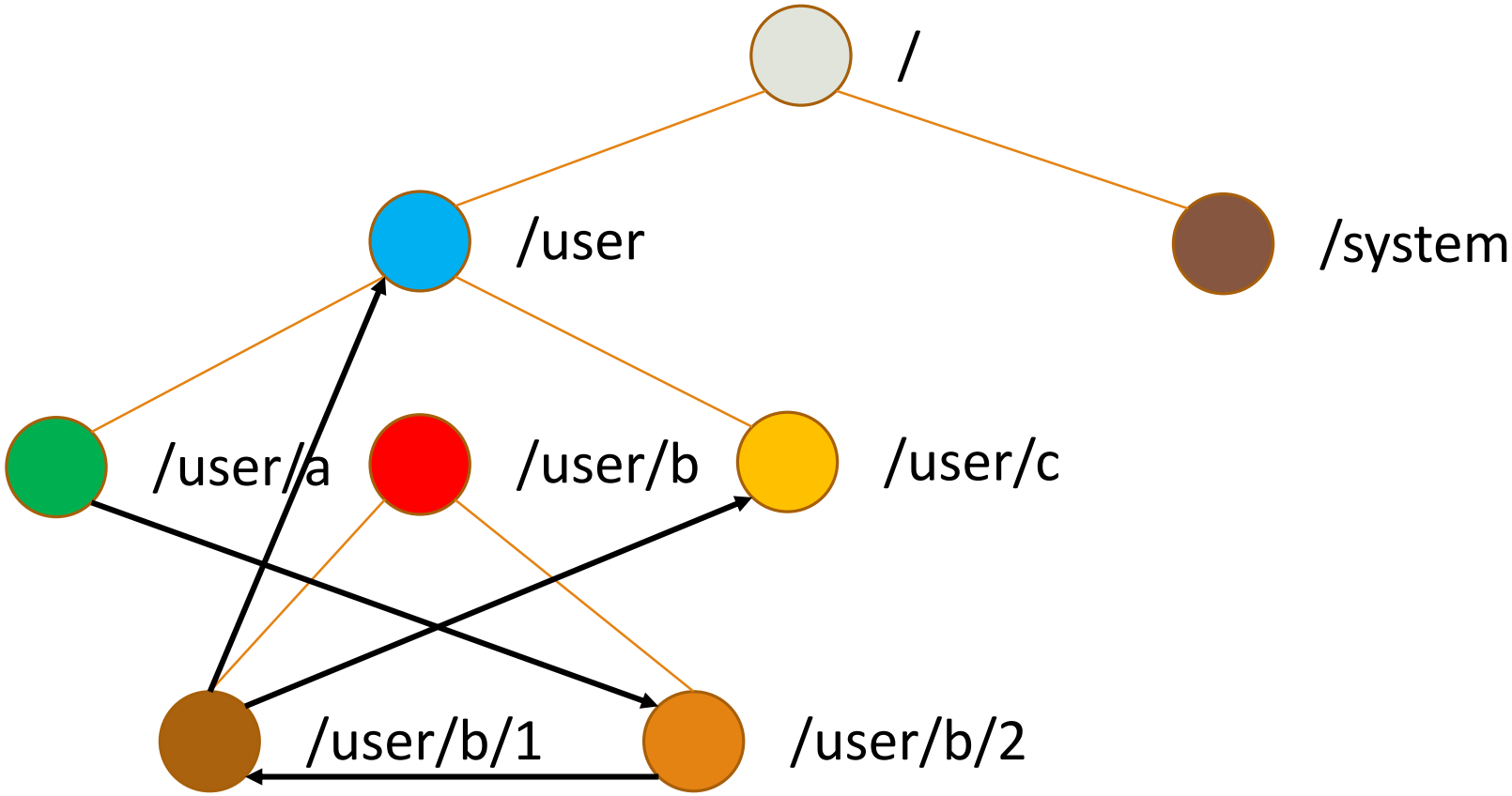
Посылать сообщения другим актерам

Последовательно обрабатывать получаемые сообщения

Изменять свое состояние

Создавать и наблюдать (supervise) за другими актерами


/
/user
/system
/user/a
/user/b
/user/c
/user/b/1
/user/b/2


/
/user
/system
/user/a
/user/b
/user/c
/user/b/1
/user/b/2

Актеры в сравнении с традиционными объектами ООП

Вы можете дать вашим объектам человеческие имена, но это не вдохнет в них жизнь

Объекты себе не принадлежат - ими управляют другие объекты (это ООП, не СОП)

Владеющий потоком владеет управлением

Актеры в сравнении с традиционными объектами ООП

Актер может провести большую часть жизни в глубоком сне, но в периоды пробуждения он и только он несет исключительную ответственность за управление своим состоянием

Воспитанные актеры не пристают к другим актерам с вопросами –они рассылают им сообщения, на которые *может* последовать ответ

Почему модель актеров и Akka.NET?

Можно сформулировать короткой фразой, которую узнают знакомые с российской историей: «Караул устал»

Нет – ручному управлению потоками

Нет – ручному контролю над состояниями общего доступа

Декларативное изменение состояний: актеры как конечные автоматы

Почему F#?

Нет – изменяемым переменным!

Идиоматический F# API, актеры как функции без локальных переменных

Актер в F# может определяться как конечный автомат, каждому состоянию которого соответствует функция

F# создан для функциональных трансформаций!

Меньше церемоний, больше полезного кода (и меньше кода)

Простейший актер, написанный на F#

```
let myActor (mailbox: Actor<_>) =

    let rec loop () = actor {
        let! message = mailbox.Receive()
        printfn "%A" message
        return! loop ()
    }

    loop ()
```

Akka.NET F# API и конечные автоматы

```
let myActor (mailbox: Actor<_>) =

    let rec disconnected () = actor {
        let! message = mailbox.Receive()
        match message with
        | Connect ->
            let connection = CreateConnection ()
            return! connected (connection)
        | _ ->
            printfn "%A is invalid in disconnected state" message
            return! disconnected ()
    }

    // Продолжение на следующем слайде
```

Akka.NET F# API и конечные автоматы

```
// … продолжение

and connected (connection) = actor {
    let! message = mailbox.Receive()
    match message with
    | Disconnect ->
        ReleaseConnection (connection)
        return! disonnected ()
    | _ ->
        printfn "%A is invalid in connected state" message
        return! connected (connection)
}

disconnected ()
```

“Говори, не спрашивай” означает “говори и слушай”

Единственный способ обмена информацией между актерами – посылка сообщений

Стандартный метод посылки сообщения в Akka – Tell, но имеется и метод Ask, ожидающий ответа

F# API определяет операторы <! и <? дляTell и Ask

Приостановка работы актера – сам по себе плохой признак с точки зрения реактивного программирования

Но что если отклик вообще не придет?

THIS IS NOT FINE!!

Но этого мало – есть еще кое-что

быстродействие

ASK

просто

УЖАСНО

Быстродействие различных операций

Тестовая машина: HP Spectre x360 i5 2.20 GHz, Windows 10
Время в секундах на 1 миллион операций

Исполнение метода без возврата результата	0.003 (~300M/c)
Исполнение метода с возвратом результата	0.004 (~250M/c)

Быстродействие различных операций

Тестовая машина : HP Spectre x360 i5 2.20 GHz, Windows 10
Время в секундах на 1 миллион операций

Исполнение метода без возврата результата	0.003 (~300M/c)
Исполнение метода с возвратом результата	0.004 (~250M/c)
Метод с возвратом Async Task с await	1.864 (~500K/c)
Метод с возвратом Async Task<int> с await	1.790 (~500K/c)

Быстродействие различных операций

Тестовая машина : HP Spectre x360 i5 2.20 GHz, Windows 10
Время в секундах на 1 миллион операций

Исполнение метода без возврата результата	0.003 (~300M/c)
Исполнение метода с возвратом результата	0.004 (~250M/c)
Метод с возвратом Async Task с await	1.864 (~500K/c)
Метод с возвратом Async Task<int> с await	1.790 (~500K/c)
Сообщение Tell актеру	0.233 (~4M/c)

Быстродействие различных операций

Тестовая машина : HP Spectre x360 i5 2.20 GHz, Windows 10
Время в секундах на 1 миллион операций

Исполнение метода без возврата результата	0.003 (~300M/c)
Исполнение метода с возвратом результата	0.004 (~250M/c)
Метод с возвратом Async Task с await	1.864 (~500K/c)
Метод с возвратом Async Task<int> с await	1.790 (~500K/c)
Сообщение Tell актеру	0.233 (~4M/c)
Сообщение Ask актеру с ожиданием ответа	6.395 (~150K/c)

Почему Ask работает так медленно?

Подробности: http://bartoszsypytkowski.com/dont-ask-tell-2/

Посольку актеры обмениваются сообщениями через почтовые ящики, как актер получит ответ на Ask?

1. Искать отклик на Ask в основном почтовом ящике, извлекая найденное сообщение вне очереди –нарушает последовательность обработки сообщений, усложняет реализацию почтового ящика
2. Создать временный актер для получения ответного сообщения

Akka использует 2-й метод, что существенно влияет на быстродействие исполнения Ask

Но что если нужно дождаться ответа?

Ожидая отклик, актер переходит в новое состояние (используя метод Become в C# или новую функцию в F#)

Находясь в состоянии *ожидания*, актер откладывает (Stash) сообщения, требующие получения того отклика

Когда ожидаемый отклик приходит, актер вызывает UnstashAll, чтобы востребовать отложенные сообщения и восстанавливает свое оригинальное состояние

Пример 1: замена Ask - до

```
let rec loop () =
    actor {
        let! message = mailbox.Receive ()
        match message with
        | Process item ->
            let (details : ItemDetails) =
                actorA <? GetItemDetails item |> Async.RunSynchronously
            actorB <! UseItemDetails item details
            return! loop ()
    }
```

Пример 1: замена Ask - после

```
let rec idle () =
    actor {
        let! message = mailbox.Receive ()
        match message with
        | Process item ->
            actorA <! GetItemDetails item
            return! awaiting_details (item)

        return! idle ()
    }

    // Продолжение на следующем слайде
```

Пример 1: замена Ask - после

```
// … продолжение

and awaiting_details (item) =
    actor {
        let! message = mailbox.Receive ()
            match message with
            | ItemDetails details ->
                actorB <! UseItemDetails item details
                mailbox.UnstashAll()
                return! idle ()
            | _ ->
                mailbox.Stash()
                return! awaiting_details ()
        }
```

Пример 2: сбор 1000000 откликов

```
// … остальной код опущен для ясности

and collecting (responses : Response list) =
   actor {
      let! message = mailbox.Receive ()
      match message with
      | Response response ->
         return! collecting (response :: responses)
      | Enough ->
         mailbox.UnstashAll()
         return! idle ()
      | _ ->
            mailbox.Stash()
         return! collecting (responses)
   }
```

Дополненные тесты быстродействия

Тестовая машина : HP Spectre x360 i5 2.20 GHz, Windows 10
Время в секундах на 1 миллион операций

Исполнение метода без возврата результата	0.003 (~300M/c)
Исполнение метода с возвратом результата	0.004 (~250M/c)
Метод с возвратом Async Task с await	1.864 (~500K/c)
Метод с возвратом Async Task<int> с await	1.790 (~500K/c)
Сообщение Tell актеру	0.233 (~4M/c)
Сообщение Ask актеру с ожиданием ответа	6.395 (~150K/c)
Сообщение Tell актеру с агрегацией откликов	0.884 (~1M/c)

Когда же можно пользоваться Ask?

“Пишите код так, как если бы программист, который будет его сопровождать, был опасным психопатом, знающим, где вы живете” Джон Вудс об использовании оператора «запятая» (C++ форум, 1991)

Схожим образом, считайте, что актер, к которому обращен метод Ask, страдает маниакальной депрессией

Нет, это не эмоциональный совет, а чисто практический

И все же когда можно пользоваться Ask?

Запрос ссылки на актера IActorRef

Системный запрос Identify

Запрос состояние персистентного актера

Запросы к актерам с тривиальной реализацией и без обращения к внешним ресурсам (на задайтесь вопросом: зачем нужны такие актеры?)

Как мы научились не бояться исключительных состояний и полюбили супервайзеры

Устойчивость(resilience)

1. способность объекта восстанавливать состояние, эластичность
2. способность восстанавливаться после сбоев

Наблюдение (supervision)

Инвентарь (prop)

(театр.) объект, используемый актером на сцене или при съемках фильма

Исключительные состояния в C#

```
connection.ExecuteCommand(cmd);
```

Исключительные состояния в C#

```
try
{
    connection.ExecuteCommand(cmd);
}
catch (SqlException ex)
{
    throw new ProductUpdateException("Unable to update product", ex);
}
```

Исключительные состояния в C#

```
try
{
    connection.ExecuteCommand(cmd);
}
catch (SqlException ex)
{
    // Retry?
    // Recover database connection?
    // What to do with other places that use the same connection?
}
catch (Exception ex)
{
    // Or should we catch other exception types?
}
```

Исключительные состояния в C#

```
queue.Subscribe(x => HandleMessage(x));

void HandleMessage(QueueMessage msg)
{
    try
    {
        Process(msg);
        msg.Ack();
    }
    catch (QueueException ex)
    {
        // Reconnect? Subscribe? BTW, we are in a different thread
    }
}
```

«Exception handling considered harmful»

Статья Джейсона Паттерсона

http://www.lighterra.com/papers/exceptionsharmful

”Exception handling introduces a hidden, "out-of-band" control-flow possibility at essentially every line of code.”

”Exception handling does not fit well with most of the highly parallel programming models currently in use or being explored (fork/join, thread pools and task queues, the CSP/actor model etc), because exception handling essentially advocates a kind of single-threaded "rollback" approach to error handling, where the path of execution – implicitly a single path – is traversed in reverse by unwinding the call stack to find the appropriate error handling code.”

Обратимся к модели актеров

Начальная реализация актера обработки очереди сообщений (RabbitMQ)

```
type QueueCommand =
      | Connect of QueueDetails
      | Disconnect
      | Subscribe of IActorRef
      | Unsubscribe
      | Publish of QueueMessage
      | Receive of AckId * Payload
      | Ack of AckId
      | Nack of AckId
```

Актер обработки очереди сообщений (первая версия)

```
let queueActor (mailbox: Actor<_>) =

   let rec disconnected () =
      // … код реализации
   and connected (connection : IConnection) =
      // … код реализации
   and subscribed (connection : IConnection, subscriber : IActorRef) =
      // … код реализации

   disconnected ()
```

А что с обработкой ошибок?

Если исключительное сообщение не обработано в коде актера, решение о нем принимает супервайзер актера согласно выбранной стратегии наблюдения

Стратегия наблюдения (supervision strategy) может относиться лишь к данному конкретному актеру (one-for-one) или всем дочерним актерам супервайзера (all-for-one)

Стандартные стратегии наблюдения

При возникновении исключительного состояния супервайзер может принять одно из нижеследующих решений:

1. Перестартовать актера (по умолчанию)
2. Продолжить работу актера
3. Остановить работу актера
4. Эскалация принятия решения

Пример стратегии наблюдения

```
let strategy () =
    Strategy.OneForOne((fun ex ->
        match ex with
        | :? ArgumentNullException ->
            Stop
        | :? ArgumentOutOfRangeException ->
            Restart
        | :? ArgumentException ->
            Resume
        | _ ->
            Escalate), 3, TimeSpan.FromSeconds(10.))
```

Применение стратегии наблюдения

1. В коде актера возникает исключительное состояние
2. Актер приостанавливает работу
3. Супервайзер информируется об ошибке и актере, в коде которого она произошла
4. Супервайзер завершает работу актера, но сохраняет его почтовый ящик
5. Супервайзер создает новую версию удаленного актера, использую тот же *инвентарь* (*props*) и назначает ему сохраненный почтовый ящик

Пересмотренные команды актера

```
type QueueCommand =
      | ~~Connect~~                          // пересылается вместе с props
      | Disconnect
      | Subscribe of IActorRef
      | Unsubscribe
      | Publish of QueueMessage
      | Receive of AckId * Payload
      | Ack of AckId
      | Nack of AckId
```

Пересмотренные команды актера

```
type QueueCommand =
      | ~~Connect~~                          // пересылается вместе с props
      | ~~Disconnect~~                       // больше не требуется
      | Subscribe of IActorRef
      | Unsubscribe
      | Publish of QueueMessage
      | Receive of AckId * Payload
      | Ack of AckId
      | Nack of AckId
```

Пересмотренные команды актера

```
type QueueCommand =
      | ~~Connect~~                          // пересылается вместе с props
      | ~~Disconnect~~                       // больше не требуется
      | ~~Subscribe of IActorRef~~           // пересылается вместе с props
      | ~~Unsubscribe~~                      // больше не требуется
      | Publish of QueueMessage
      | Receive of AckId * Payload
      | Ack of AckId
      | Nack of AckId
```

Пересмотренный код актера

```
let queueActor (queueDetails: QueueDetails)
               (subsriber: IActorRef)
               (mailbox: Actor<_>) =

   let connection = factory.CreateConnection(queueDetails)
   // … подписать подписчиков (subscribers)

   let rec loop () =
      // … остальной код опущен для ясности

   loop ()
```

Запуск актера обработки очередей сообщений

```
let queue = spawn system "queues.file_upload" queueActor
                    queueDetails subscriber
```

```
queueDetails    - PROP
subscriber      - PROP
```

Ни SQL, ни NoSQL: на помощь приходят персистентные актеры

Функции сохраняемых состояний в Akka реализуются по принципам источников событий (event sourcing)

Новые проекты получают возможность реализации всех своих потребностей в сохранямых состояниях через персистентные актеры

Это первый случай в моей практике, когда в проекте мы обошлись без написания кода доступа к базам данных

Пример персистентного состояния

```
type FileDistribution = {
    StorageProvider : FileStorageProvider
    Locator : RemoteLocation
    CdnPath : AbsoluteUrl option
    GeoRestriction : GeoRestriction option
    AccessLevel : AccessLevel option
    State : DistributionState
    Length : uint64
    Timestamp : DateTimeOffset }

type DistributionLocatorCommand =
      | AssignLocator of FileDistribution
      | RemoveLocators
      | QueryLocators
      | TakeSnapshot
```

Персистентные адаптеры Akka.NET

In-memory + локальные файлы
Microsoft SQL Server
Azure tables
PostgreSql
Cassandra
MySql
MongoDB
Redis
Sqlite
Oracle

Журнал событий Akka.NET

```sql
CREATE TABLE EventJournal (
  Ordering BIGINT IDENTITY(1,1) PRIMARY KEY NOT NULL,
  PersistenceID NVARCHAR(255) NOT NULL,
  SequenceNr BIGINT NOT NULL,
  Timestamp BIGINT NOT NULL,
  IsDeleted BIT NOT NULL,
  Manifest NVARCHAR(500) NOT NULL,
  Payload VARBINARY(MAX) NOT NULL,
  Tags NVARCHAR(100) NULL
  CONSTRAINT QU_EventJournal UNIQUE (PersistenceID, SequenceNr)
)
```

Масштабирование актеров

RoundRobin

Broadcast

Random

ConsistentHashing

TailChopping

ScatterGatherFirstCompleted

SmallestMailbox

Пример 1: создание одиночного актера

```
let uploader =
      spawn system "akamai.uploader"
            (akamaiUploadActor baseUrl uploadReporter)
```

Пример 2: создание пула актеров

```
let uploader =
  spawnOpt system "akamai.uploader"
    (akamaiUploadActor baseUrl uploadReporter)
      [SpawnOption.Router(Routing.ConsistentHashingPool(
        Limits.AkamaUploadActorPoolSize)
      .WithHashMapping(
        Routing.ConsistentHashMapping akamaiUploaderGetHashBase)
      .WithSupervisorStrategy(
        getOneForOneSupervisorStrategy system))]
```

Музыкальная пауза

Goodbye $b

Песня о жизни актера

В песне использованы следующие термины Akka и модели актеров

1. Начальные данные, используемые при создании актера, называются props
2. Ссылки на актеров даются с помощью ActorRef
3. Akka использует имена $a, $b, $c для актеров пула
4. Если при ошибке supervision strategy требует перестартовать актера, новая копия получает то же самое имя
5. Если стратегия исправления ошибки задана как AllForOne + Restart, все актеры, наблюдаемые данным супервайзором, будут перестартованы

Goodbye $b

Песня о жизни актера

Goodbye, $b
Though I never knew at all
They’d chosen to restart you
And new instance got your name
But when you were alive
You played your role according to your props
And every single message knew
That for you she’s only one

And it seems to me
You lived your life
Like a good reactive code
Only using CPU cycles
When the message came
And I would've liked to have known you
But Supervisor’s fast
Your instance burned out long ago
Your ActorRef will last

Loneliness was tough
You had to sleep alone most of the time
And when you were awakened
They’d let you tell but never ask
Even when you died
It was another sibling's fail
But supervision strategy
Was clear and brutal: AllForOne

And it seems to me
You lived your life
Like a good reactive code
Only using CPU cycles
When the message came
And I would've liked to have known you
But Supervisor’s fast
Your instance burned out long ago
Your ActorRef will last

reF#actoring: переписывание актера с C# на F#

Akka содержит планировщик, использующий лишь оперативную память

Quartz.NET (изначально на Яве) – опенсорсная библиотека с поддержкой оперативной памяти и баз данных ADO.NET

Akka.Quartz.Actor – актер-планировщик, написанный на C# и использующий Quartz.NET (https://github.com/akkadotnet/Akka.Quartz.Actor)

Попробуем переписать его на F# (тем более на то были причины!)

Файлы и строки Akka.Quartz.Actor

QuartzActor.cs	97	93 (без комментариев)
QuartzJob.cs	35	32
Commands/IJobCommand.cs	9	6
Commands/CreateJob.cs	33	21
Commands/RemoveJob.cs	26	17
Events/IJobEvent.cs	13	10
Events/JobEvent.cs	23	17
Events/JobCreated.cs	36	30
Events/CreateJobFail.cs	26	20
Events/RemoveJobFail.cs	23	20

10 файлов, 2 интерфейса, 8 классов, 321 строк кода (267 без комментариев)

Переходим на F#

Всего один файл Schedule.fs
2 синонима (aliases) типов
2 discriminated unions
1 приватных типа
1 функция актера
85 строк кода (без комментариев)

Помните C#? 10 типов (2 + 8), 267 строк полезного кода

Актер-планировщик на F#

```
open System
open Akka.Actor

type JobMessage = obj
type JobId = obj

type JobCommand =
| CreateJob of IActorRef * JobMessage * DateTimeOffset
| RemoveJob of JobId

type JobCommandResult =
| Success of JobId
| Error of JobId * Exception
```

В проекте C# этот код занял 8 файлов (141 строку)

Актер-планировщик на F#

```
let scheduleActor props (mailbox: Actor<_>) =

    let t = TriggerBuilder.Create().StartNow().Build()
    let scheduler =
        match props with
        | Some props -> StdSchedulerFactory(props).GetScheduler()
        | None -> StdSchedulerFactory().GetScheduler()

    scheduler.Start()
    mailbox.Defer (fun _ -> scheduler.Shutdown())

    let rec loop () =
        actor {
        let! message = mailbox.Receive ()
            // … Код вынесен на следующий слайд
        }

    loop ()
```

Актер-планировщик на F#

```
match message with
| CreateJob (actor, message, scheduledTime) ->
    match (actor, scheduledTime) with
    | (null,_) -> mailbox.Sender()
                  <! Error (null, new ArgumentException("CreateJob actor is null"))
    | _ ->
        let trigger = TriggerBuilder.Create().StartAt(scheduledTime).Build()
        try
            let job = QuartzJob.CreateBuilderWithData(actor, message)
                          .WithIdentity(trigger.JobKey)
                          .Build()
            scheduler.ScheduleJob(job, trigger) |> ignore
            mailbox.Sender() <! Success trigger.JobKey
        with ex ->
            mailbox.Sender() <! Error (trigger.JobKey, ex)

| RemoveJob (jobKey) ->
    // … код для RemoveJob опущен
```

Итоги reF#actoring

Код F# более компактен, чем код C#, равно как и функции актеров на F#, в нашем случае они занимают на 60-70% меньше строк, чем соответствующий код на C#

F# discriminated unions работают отлично для определения сообщений актеров (комманд и событий)

Меньше проверок нулей, нули появляются лишь извне

Определения сообщений и код использующих их актера обычно можно разместить в одном файле

Статистика reF#actoring (весь проект)

Media Distribution Engine	C#	F#
Файлы	195	56
Всего строк	12089	6042
Пустые строки	1325	913
Строки комментариев	242	32
Строки только со скобками	3398	57
Строки namespace	1233	525
Строки полезного кода	5891 (55%)	4515 (88%)

Актеры – не «серебряные пули»

Помните наши тесты? Если все, что вам требуется, доступно прямым вызовом метода, так и поступайте!

Маленький стэк (stack trace) затрудняет отладку и лишает возможности выполнения транзакций, охватывающих группу операций

Актеры – не «серебряные пули»

Актеры предназначаются для внутреннего пользования – внешние системы не обязаны знать, что программа создана на основе модели актеров

Актеры нелегко компоновать (Akka streams призвана это улучшить)

Когда же стоит создавать актеров?

Параллельное исполнение операция

Декларативный контроль за изменением состояний

Конфигурация масштабирования

Достаточность односторонних сообщений

Спасибо!

Работаю в норвежской компании Miles

Электронная почта: vagif.abilov@gmail.com

Twitter: @ooobject

GitHub: object

Блог: http://vagifabilov.wordpress.com/

Статьи: http://www.codeproject.com

Поддерживаю библиотеку Simple.OData.Client, участвую в других опенсорсных проектах