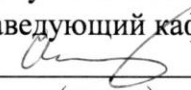


Министерство образования и науки Российской Федерации
федеральное государственное бюджетное образовательное учреждение
высшего образования
«Алтайский государственный технический университет им. И.И. Ползунова»

Факультет (институт) Информационных технологий
Кафедра Прикладная математика
Направление Программная инженерия

УДК 004.075

Допустить к защите в ГЭК
Заведующий кафедрой
 С. А. Кантор
(подпись) (инициалы, фамилия)
“ ” 2016 г.

БАКАЛАВРСКАЯ РАБОТА

БР 09.03.04.14.000 ПЗ

(обозначение документа)

**Фреймворк для распределения задач с использованием
сетей функциональных вычислений**

(тема бакалаврской работы)

Пояснительная записка

Студент группы ПИ-21 Кристалев Данил Андреевич
(фамилия, имя, отчество)

Руководитель

проекта (работы) к.ф.-м.н., доцент каф ПМ
(должность, ученая степень)

С.М.Старолетов
(инициалы, фамилия)

Барнаул 2016

Реферат

Работа посвящена проектированию и разработки фреймворка для распределения задач с использованием сетей функциональных вычислений.

Разработанное программное обеспечение позволяет создавать распределенные системы обработки данных, для написания которых могут быть использованы различные языки программирования.

Пояснительная записка содержит 126 страниц, 52 рисунка, 1 таблицу и 31 использованный источник.

Ключевые слова: актер, акторная система, поведение актора, передача сообщений, фреймворк, Akka, Scala, Erlang, JInterface.

The abstract





The work is dedicated to the design and development of framework for distribution of tasks with using network functional computing.

The developed software allows you to create distributed data processing system for writing which different programming languages can be use.

The work contains 126 pages, 52 pictures, 1 table and 31 sources used.

Keywords: actor, actor system, behavior actor, message passing, framework, Akka, Scala, Erlang, JInterface.

Работа допущена к опубликованию в полном объеме в соответствии с п. 38 Приказа Минобрнауки от 29.06.2015г. № 636.

						БР 09.03.04.14.000 ПЗ			
Изм.	Коп. уч.	Лист	№ док.	Подпись	Дата	Фреймворк для распределения задач с использованием сетей функциональных вычислений	Стадия	Лист	Листов
Разработал		Кристалев Д.А.			28.06.16		У	2	126
Рук. проекта		Старозетов С.М.			29.06.16		АлтГТУ ФИТ ПИ-21		
Н. контролер		Потупчик А.И.			28.06.16				
Зав. кафедр.		Кантор С. А.							

Содержание

Введение.....	8
1 Обзор существующих инструментов	10
1.1 Общие сведения.....	10
1.2 Функциональные языки программирования распределенных систем	13
1.2.1 Erlang	15
1.2.2 Scala	16
1.3 Возможные способы организации общения акторов	17
1.3.1 Apache Kafka	18
1.3.2 Сторонние фреймворки	22
1.3.3 Итоги	23
1.4 Абстрактная модель системы взаимодействия акторов.....	24
1.5 Постановка задачи.....	25
2 Проектирование фреймворка	26
2.1 Обоснование выбора языка программирования	26
2.2 Область применения проектируемого фреймворка	26
2.3 Проектирование классов фреймворка.....	27
2.4 Архитектурные решения	28
2.4.1 Объектно-ориентированные решения.....	29
2.4.2 Функциональные решения	29
2.5 Примеры решения задач.....	31
2.5.1 Пример “Файловая система на процессах”	31
2.5.1.1 Постановка задачи	31
2.5.1.2 Анализ задачи	31
2.5.1.3 Диаграмма классов для решения задачи.....	32

2.5.1.4	Диаграммы действий акторов	35
2.5.1.5	Диаграммы взаимодействия акторов	47
2.5.2	Пример “Поиск минимального значения в непрерывном потоке данных”	49
2.5.2.1	Постановка задачи	49
2.5.2.2	Анализ задачи	49
2.5.2.3	Диаграмма классов для решения задачи.....	50
2.5.2.4	Диаграммы действий акторов.....	51
2.5.2.5	Диаграммы взаимодействия акторов	56
3	Разработка фреймворка	57
3.1	Общие сведения о разработке фреймворка и систем на его основе	57
3.2	Спецификации на фреймворк	59
3.3	Релизация решенных задач	63
3.3.1	Задача “Файловая система на процессах”	63
3.3.1.1	Erlang реализация	63
3.3.1.2	Спецификации на Scala реализацию	66
3.3.2	Задача “Поиск минимального значения в непрерывном потоке данных”	72
3.3.2.1	Erlang реализация	72
3.3.2.2	Спецификации на Scala реализацию	73
4	Тестирование	75
4.1	Обзор системы тестирования.....	75
4.2	Задача на тестирование.....	75
4.3	Ожидаемое поведение	76
4.4	Общий подход к тестированию	77

4.5 Результаты тестирования	80
4.6 Итоги тестирования	85
Заключение	86
Список использованных источников	87
Приложение А Задание на бакалаврскую работу	Ошибка! Закладка не определена
Приложение Б Руководство пользователя.....	92
Приложение В Исходный код.....	98

Введение

Мир меняется, и требования к программам меняются вместе с ним. Еще несколько лет назад действительно большие приложения имели десятки серверов, секундные отклики, часы офлайн-обслуживания и гигабайты данных. Сегодня приложения разворачиваются на чем угодно, начиная от мобильных платформ, заканчивая облачными кластерами, которые работают на основе тысяч многоядерных процессоров. Пользователи ожидают миллисекундное время отклика и бесперебойное время работы. Данные измеряются в петабайтах. Сегодняшним требованиям просто не могут удовлетворять вчерашние архитектуры приложений. По нашему мнению, на сегодняшний день наиболее актуальные требования к системам следующие:

- Отзывчивость. Система реагирует своевременно, если это возможно.
- Отказоустойчивость. Система продолжает отвечать, даже при сбоях.
- Масштабируемость. Система должна легко переноситься (распространяться) на другое железо, не теряя при этом отзывчивость.

Сегодня стал набирать популярность функциональный подход[1] к программированию благодаря своим особенностям, к которым можно отнести: отсутствие побочных эффектов, краткость программ и их выразительность по сравнению с императивными программами, «ленивые» вычисления, возможность «горячей» замены кода и т.д. Благодаря отсутствию побочных эффектов при программировании в функциональном стиле, отсутствует множество проблем, связанных с распараллеливанием программ, например, нет проблемы «гонки данных»[2]. Это и есть главный фактор, популяризирующий сегодня функциональный подход. Но у такого подхода есть и недостатки, к которым можно отнести: обязательный высокоэффективный сборщик мусора, больший, по сравнению с императивными программами, объем памяти, относительно долгая скорость компиляции и т.д. Но эти проблемы в большей степени удалось решить и поэтому использование функциональных языков сегодня стало привлекательным. К функциональным языкам, которые представляют большой интерес в рамках нашего исследования, относят такие языки как Scala[3] и Erlang[4].

Цель данной работы – проектирование фреймворка для написания систем распределения задач и реализация систем, демонстрирующей работу фреймворка.

1 Обзор существующих инструментов

1.1 Общие сведения

Сегодня есть несколько решений, которые могут соответствовать потребностям отказоустойчивости, отзывчивости и масштабируемости: STM (транзакционная память) [5], Dataflow Concurrency(конкурентные потоки данных) [6] и Actors(акторная модель) [7]. Рассмотрим подход с акторной моделью. Приложения, написанные согласно такой модели, удовлетворяют перечисленным требованиям. Помимо этого, приложения, основанные на акторах, полагаются на асинхронные передачи сообщений и это - главная особенность указанного подхода. Благодаря механизму передачи сообщений можно установить границы между компонентами, которые обеспечивают слабосвязность, изоляцию, прозрачность местоположения, а также предоставляет средства для делегирования ошибки в виде сообщения.

Актор (Actor, от action-действие) – в агентно-ориентированном программировании [8] и модели акторов – программная сущность заданной структуры и механизмов взаимодействия. В акторной модели единственным механизмом взаимодействия является передача сообщений.

Первым языком, реализующим данную модель, стал в 1987 году функциональный язык Erlang. К плюсам языка можно отнести: легковесные процессы с собственным высокоэффективным планировщиком, OTP (фреймворк для построения отказоустойчивых приложений) [9], функциональная парадигма и т.д. Недостатками являются: необходимость непопулярной виртуальной машины BEAM [10], малочисленное и медленно развивающееся сообщество, из-за чего у языка почти нет открытых фреймворков, некоторые недостатки синтаксиса языка. Но сегодня стали появляться альтернативы Erlang с его акторной моделью, которых раньше не было. Например, язык Go [11], разрабатываемый Google, реализует акторную модель, правда со своими особенностями, такими как каналы.

Однако существует и система, которая очень напоминает Erlang, но при этом лишена недостатков этого языка. Это решение – язык Scala с использованием библиотеки Akka [12].

Scala – современный мультипарадигменный язык программирования. Scala развивается, у него активное сообщество, а главное, этот язык использует JVM с её JIT компилятором, ускоряющим работу приложений, умным сборщиком мусора и богатой инфраструктурой [10]. На этом языке была разработана библиотека Akka, которая реализует акторную модель во многом схожую с той, что используется в Erlang.

Для примера посмотрим приложения, написанные на языке Erlang и Scala, реализующие игру двух акторов в пинг-понг.

Пинг-понг:

- Один актор шлет другому Ping и ждет ответа.
- Другой получает Ping, и шлет Pong в ответ.
- Акторы запускаются параллельно, причем, они должны знать друг о друге, для того, чтобы отправлять сообщения.

Реализация пинг-понг на языке Erlang [13]:

```
-module('pingpong').
-export([main/0]).

%%%функция, инициализирующая пинг-понг. точка входа в программу.
main()->spawn(fun() ->ping(spawn(fun() ->pong() end) end) ).

%%% процесс(актор) ping, получает pid процесса pong для отправки ему сообщения
ping(PidPong)->
    io:format("I'm ping, pids are: ~w ~w ~n ", [PidPong,self()]),
    PidPong ! {ping, self()}, %%%отправка сообщения ping
    receive
pong -> io:format("PONG received~n") %%%получение ответа и вывод сообщения об
ЭТОМ
    end.

%%% процесс(актор) pong
pong() ->
    io:format("I'm pong, pids are: ~w ~n ", [self()]),
    receive
{ping, PidPing}-> io:format("PING received, sending pong~n "), %%%получение
сообщения ping и вывод сообщения об этом
    PidPing ! pong %%%отправка ответа
```

end.

Результат работы программы:

I'm ping, pids are: <0.30.0> <0.29.0>

I'm ping, pids are: <0.30.0>

PING received, sending pong

PONG received

Реализация пинг-понг на языке Scala:

```
class PingServer extends Actor{
  override def receive = {
    case "Ping" => { //получение сообщения ping и вывод сообщения об этом
      println(Thread.currentThread().getName() + ": Ping!")
      sender ! "Pong" //отправка ответа
    }
    case _ => println("Unknown message!")
  }
}

//актор, которому при создании указывается на какой сервер слать сообщения
class PongClient(server: ActorRef) extends Actor{
  override def receive = {
    case "Start" => server ! "Ping" //отправка сообщения на сервер
    case "Pong" => {
      println(Thread.currentThread().getName + ": Pong!") //получение ответа и
вывод сообщения об этом
      //остановка акторной системы
      context.stop(server)
      context.stop(self)
      context.system.terminate();
    }
    case _ => println("Unknown message!")
  }
}

object Main extends App {
  val system = ActorSystem("PingPong"); //новая акторная система
  val server = system.actorOf(Props[PingServer]) //создание и запуск актора
  PingServer
```

```
val client = system.actorOf(Props(new PongClient(server))) //создание и
запуск актора PongClient
client ! "Start" //отправка инициализирующего сообщения

}
```

Результат работы программы:

```
PingPong-akka.actor.default-dispatcher-3: Ping!
PingPong-akka.actor.default-dispatcher-4: Pong!
```

Как видно в обоих примерах, акторы действительно работают в разных потоках или процессах, а процесс получения и отправки сообщений во многом похож.

1.2 Функциональные языки программирования распределенных систем

Scala и Erlang – функциональные языки программирования. Рассмотрим подробнее функциональную парадигму программирования.

Lisp [14] – второй по возрасту после Фортрана, широко используемый язык программирования. При этом Lisp являлся первым функциональным языком программирования. В наше время до сих пор существуют и активно используются диалекты этого языка, такие как Clojure [15], Scheme и Common Lisp. Почему же в наше время, когда явное преимущество имеет императивный подход к программированию, в частности объектно-ориентированный, до сих пор не потерял актуальность и функциональный подход, который во многом отличается от первого, и кроме того начал активно набирать сторонников?

Рассмотрим особенности функциональной парадигмы программирования. Выделим следующие преимущества этого подхода:

- Повышение надежности кода. Одной из главных особенностей функциональных языков программирования, является то, что данные не изменяются. Другими словами, нельзя переприсваивать значения переменных. Это, в свою очередь, лишает разработчика необходимости

следить за «побочными эффектами» функций. С точки зрения функционального программирования, все выражения в программе, являются функциями. В том числе подпрограммы и методы в рамках этой концепции нужно рассматривать как функции.

- Краткость и простота программ. Программы на функциональных языках обычно короче и проще, чем те же самые программы на императивных языках. Это связано с тем, что программа, написанная в функциональном стиле, описывает спецификацию проблемы, но не содержит четкой последовательности шагов ее решения. Таким образом, код, написанный в таком стиле, содержит больше смысла, и, как правило, его объем меньше, чем аналогичный код в императивном стиле, а читать такой код проще.
- Строгая типизация. В большинстве функциональных языков программирования используется строгая типизация, с выводом типов, основанном на системе типов Хиндли-Милнера [26], которая обеспечивает высокий показатель повторного использования кода за счет параметрического полиморфизма. Строгая система типов обеспечивает повышенную надежность программ и дает разработчику возможность доказать корректность важных участков программы.
- Многопоточность. Одним из самых больших источников ошибок в многопоточных программах является изменение одних и тех же данных из нескольких потоков без надлежащей синхронизации. Если вы используете неизменяемые данные и функции без побочных эффектов, вы можете быть уверены, что такие неправильные изменения не произойдут, и вам не нужно беспокоиться о сложных схемах синхронизации.

Так же к плюсам функционального программирования можно отнести: функции высшего порядка, «ленивые» вычисления, возможность горячей замены кода и т.д.

Помимо преимуществ, у функционального подхода есть и недостатки: обязательный высокоэффективный сборщик мусора, большой, по сравнению с императивными программами, необходимый объем памяти, относительно долгая

скорость компиляции и т.д. Так же отсутствие «побочных эффектов», порождает ряд проблем, например, с вводом и выводом данных.

Однако технологии развиваются. Благодаря популярности таких языков, как Java и C#, большое развитие получили системы, на которых они исполняются: JVM и CLR. Как следствие эти исполняющие среды имеют одни из лучших (по показателям эффективности) сборщики мусора и JIT компиляторы, а память на сегодняшний день уже не является ресурсом, на котором экономят разработчики. Это объясняет появление новых функциональных языков программирования, основанных на JVM и CLR: Clojure, Scala, F#, Kotlin [16] и т.д. Так же теория категорий, смогла дать теоретические обоснования для такой концепции, как монада [17]. Именно при помощи монад в чистом функциональном языке Haskell реализован ввод и вывод данных, при этом, не нарушая принципов функциональной парадигмы.

Решение основных проблем в использовании функциональных языков и те преимущества, которые они дают при разработке программ, делают функциональную парадигму привлекательной при разработке современного программного обеспечения.

К функциональным языкам, позволяющим без проблем создавать распределенные системы можно отнести Scala, Erlang, Elixir, Clojure и т.д. Создание легко масштабируемых систем является одной из главных особенностей этих языков. Рассмотрим подробнее два языка: Scala и Erlang.

1.2.1 Erlang

Erlang (Ericcson Language) – функциональный язык, ориентированный на решение различных задач, как специфических, так и общего назначения в том числе по распределенной обработке данных. Разработан телекоммуникационной компанией Ericson для управления телефонными коммутаторами, далее развивается как продукт с открытым исходным кодом (лицензия Apache Licence 2.0).

Язык содержит в себе концепции, близкие языкам Lisp (рекурсивная работа со списками) и Prolog (вывод, сопоставление с образцом)[13].

Плюсы языка Erlang:

- Функциональная парадигма программирования
- Компиляция в байт-код для виртуальной машины BEAM (легковесные процессы, собственный планировщик процессов, сборщик мусора у каждого процесса и тд.) .
- Сопоставление с образцом (pattern matching) (удобно для реализации протоколов, разбора данных, экспертных систем).
- Ориентация на процесс как основную единицу программы.
- OTP (Open telecom platform) – модель рабочих и главных процессов, предназначена для написания отказоустойчивых высоконагруженных приложений.
- Документно-ориентированная (NoSQL) база данных (Mnesia).
- Средства горячей замены кода без остановки работающей системы.

Минусы языка:

- Низкая производительность, из-за чего на важных участках используют вызов кода на C.
- Отсутствие большого, по сравнению, например, с Java, количества библиотек и фреймворков.
- Отсутствие статической типизации.

1.2.2 Scala

Scala (Scalable Language) - мультипарадигменный (функциональный и объектно-ориентированный) язык программирования общего назначения. Язык был разработан и активно развивается под руководством Мартина Одерски, одного из главных разработчика компилятора Java (javac).

Плюсы языка Scala:

- Легкая интеграция Java кода (среда выполнения Scala - JVM)

- Строгая типизация с удобной системой типов. Компилятор сам умеет выводить типы за разработчика.
- Легкая реализация асинхронного параллелизма (actors, futures, promises).
- Трейты. Аналог интерфейсов в Java. Однако помимо методов, могут содержать поля.
- Сопоставление с образцом (pattern matching). Так же предоставлены механизмы для создания структур данных удобный для сопоставления с образцом: кейс-классы и кейс-объекты
- Функциональная парадигма.

Минусы языка:

- Долгая компиляция по сравнению, например, с Java.
- Отсутствие легковесных процессов, как у Erlang.
- Высокий порог вхождения. Язык предоставляет очень много инструментов и механизмов, с которыми непросто разобраться.

1.3 Возможные способы организации общения акторов

Акторы общаются между собой при помощи отправки и получения сообщений. В пределах одного языка, который позволяет реализовать систему акторов, уже существуют механизмы общения. Однако, так как понятие актор высоко абстрактное и к конкретному языку программирования не привязано, нет ограничений на однородность акторов, то есть возможность организовать общение акторов двух разных систем, в том числе написанных на разных языках программирования.

Рассмотрим два способа организации общения Erlang процессов и Scala акторов: при помощи брокера сообщений Apache Kafka [18] и при помощи библиотеки JInterface [19].

1.3.1 Apache Kafka

Одним из способов организации общения акторов является использование технологии Apache Kafka в качестве посредника для передачи сообщений.

Apache Kafka является распределенным, масштабируемым и реплицирующим брокером сообщений.

Основные принципы работы Kafka:

- Kafka поддерживает каналы сообщений, называемые темой;
- Процессы, которые отправляют сообщения по теме, называются производителями;
- Процессы, которые подписаны на тему и получают опубликованные сообщения, называются потребителями;
- Kafka стартует на кластере, состоящем из одного или нескольких серверов, каждый из которых называется брокер.

Таким образом, на высоком уровне производители посылают сообщения через сеть на Kafka кластер, который обслуживает потребителей, как показано на рисунке 1.1.

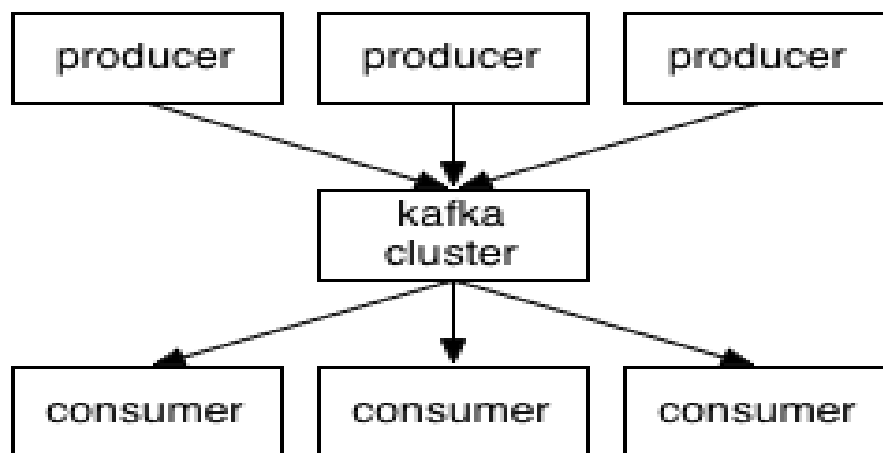


Рисунок 1.1 - Схема работы брокера сообщений Kafka

Общение между клиентами и сервером осуществляется за счет простого, высокопроизводительного языка, поддерживающего TCP протокол. Для Kafka существует множество клиентов, в том числе для Erlang и Scala.

У подхода, использующего Kafka, есть положительные и отрицательные стороны. К плюсам можно отнести:

- Клиенты на многих языках, легкое расширение для другого языка;
- Масштабируемое решение из коробки;
- Гарантированная доставка сообщения;
- Достаточно знать адрес Kafka кластера для общения, а не всех узлов.

Минусы:

- Необходимо настроить и поддерживать Kafka кластер;
- Актеры общаются не напрямую, а через посредника. В случае неисправности кластера, акторы не смогут общаться, а отправленные сообщения потеряются;

Для реализации этого подхода можно воспользоваться клиентами. Пример использования клиента для Scala:

Scala producer.

```
package com.colobu.kafka
import kafka.producer.ProducerConfig
import java.util.Properties
import kafka.producer.Producer
import scala.util.Random
import kafka.producer.Producer
import kafka.producer.Producer
import kafka.producer.Producer
import kafka.producer.KeyedMessage
import java.util.Date
object ScalaProducerExample extends App {
  val events = args(0).toInt
  val topic = args(1)
  val brokers = args(2)
  val rnd = new Random()
  val props = new Properties()
  props.put("metadata.broker.list", brokers)
  props.put("serializer.class", "kafka.serializer.StringEncoder")
  props.put("producer.type", "async")
  val config = new ProducerConfig(props)
  val producer = new Producer[String, String](config)
  for (nEvents <- Range(0, events)) {
    val runtime = new Date().getTime();
```

```

    val ip = "192.168.2." + rnd.nextInt(255);
    val msg = runtime + "," + nEvents + ",www.example.com," + ip;
    val data = new KeyedMessage[String, String](topic, ip, msg);
    producer.send(data);
  }
}

```

Scala consumer.

```

import java.util.Properties
import java.util.concurrent._
import scala.collection.JavaConversions._
import kafka.consumer.Consumer
import kafka.consumer.ConsumerConfig
import kafka.utils._
import kafka.utils.Logging
import kafka.consumer.KafkaStream
class ScalaConsumerExample(val zookeeper: String,
                           val groupId: String,
                           val topic: String,
                           val delay: Long) extends Logging {

  val config = createConsumerConfig(zookeeper, groupId)
  val consumer = Consumer.create(config)
  var executor: ExecutorService = null
  def shutdown() = {
    if (consumer != null)
      consumer.shutdown();
    if (executor != null)
      executor.shutdown();
  }

  def createConsumerConfig(zookeeper: String, groupId: String): ConsumerConfig
= {
    val props = new Properties()
    props.put("zookeeper.connect", zookeeper);
    props.put("group.id", groupId);
    props.put("auto.offset.reset", "largest");
    props.put("zookeeper.session.timeout.ms", "400");
    props.put("zookeeper.sync.time.ms", "200");
    props.put("auto.commit.interval.ms", "1000");
    val config = new ConsumerConfig(props)
    config
  }
}

```

```

def run(numThreads: Int) = {
    val topicCountMap = Map(topic -> numThreads)
    val consumerMap = consumer.createMessageStreams(topicCountMap);
    val streams = consumerMap.get(topic).get;

    executor = Executors.newFixedThreadPool(numThreads);
    var threadNumber = 0;
    for (stream <- streams) {
        executor.submit(new ScalaConsumerTest(stream, threadNumber, delay))
        threadNumber += 1
    }
}

object ScalaConsumerExample extends App {
    val example = new ScalaConsumerExample(args(0), args(1),
args(2), args(4).toLong)
    example.run(args(3).toInt)
}

class ScalaConsumerTest(val stream: KafkaStream[Array[Byte], Array[Byte]], val
threadNumber: Int, val delay: Long) extends Logging with Runnable {
    def run {
        val it = stream.iterator()
        while (it.hasNext()) {
            val msg = new String(it.next().message());
            System.out.println(System.currentTimeMillis() + ", Thread " + threadNumber
+ ": " + msg);
        }
        System.out.println("Shutting down Thread: " + threadNumber);
    }
}

```

Схема общения Scala и Erlang при таком подходе изображена на рисунке 1.2.

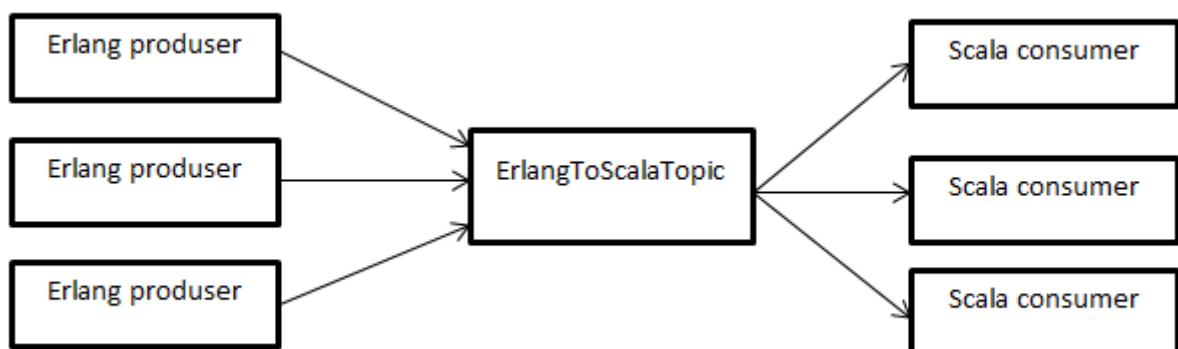


Рисунок 1.2 - Схема отправки сообщений от Erlang Process до Scala Actor

При таком подходе будут создаваться некоторые темы, в которые производители-процессы Erlang будут отправлять сообщения, а потребители-акторы Scala будут их получать, и на основе полученных сообщений принимать решения. Аналогичная схема работает и в противоположном направлении.

1.3.2 Сторонние фреймворки

Другим подходом организации общения между акторами является использование сторонних фреймворков. Примером такого фреймворка может служить JInterface от Ericsson.

JInterface – это средство коммуникации Java и Erlang. Благодаря тому, что Scala позволяет использовать Java код, то JInterface можно использовать как средство коммуникации Scala и Erlang.

Набор классов в JInterface можно поделить на две категории: обеспечивающие общение, и дающие представление для Java о типах данных Erlang. Все последние являются подклассами `OtpErlangObject`, и они обозначаются префиксом `OtpErlang`. Например: `OtpErlangAtom`, `OtpErlangInt`, `OtpErlangMap` и т.д.

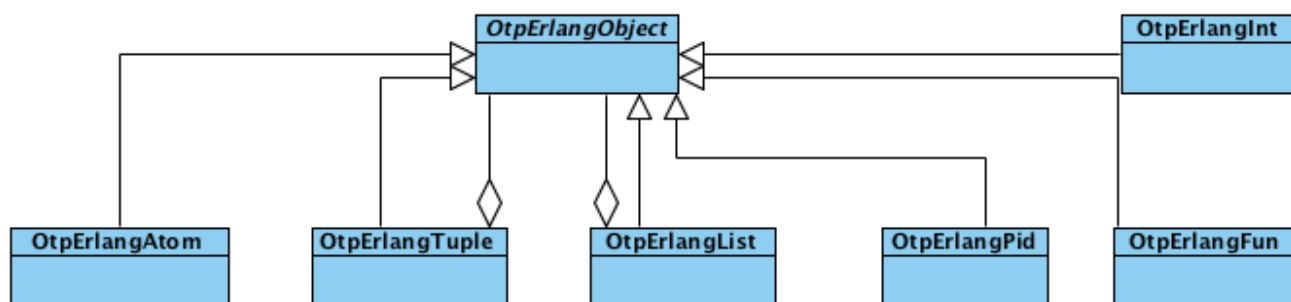


Рисунок 1.3 - Диаграмма типов данных Erlang JInterface[13]

Поскольку эта библиотека предоставляет механизм для общения с Erlang, получателями сообщений могут быть Erlang процессы или экземпляры `com.ericsson otp.erlang.OtpMbox`, у каждого из которых есть свой уникальный pid, и, возможно, зарегистрированное имя. Классы JInterface обеспечивают:

- использование типов данных Erlang;
- преобразование данных между типами Java и Erlang;
- кодирование и декодирование данных Erlang для хранения и передачи;

- общение между узлами Java и Erlang процессами.

Однако JInterface не дает возможности писать собственные акторы на языке Java, он всего лишь позволяет Java коду обращаться к Erlang системе. А так как Erlang система может ответить не сразу, то приходится ожидать ответа, что, как правило, будет связано с остановкой вычислительного потока. Однако Scala дает возможность создать акторы, а использование JInterface может послужить как непосредственный механизм общения Scala и Erlang акторов.

Плюсы использования JInterface:

- Прямое взаимодействие Scala и Erlang кода (обмен сообщениями без посредников, по сравнению с Kafka Apache);
- Возможность описания взаимодействия вплоть до RPC (удаленного вызова процедур), а не только обмен сообщениями;
- JInterface – от создателей Erlang, приемлемая производительность.

Минусы:

- Только Erlang и JVM;
- Различные типы данных Erlang и Scala, необходимо приводить много типов.

1.3.3 Итоги

Были рассмотрены основные способы организации общения акторов. Ни один из подходов не позволяет организовывать прямое общение «актор-актор». В случае подхода с Apache Kafka, главным минусом является наличие посредника между системами, в виде кластера Apache Kafka. В случае использования сторонних фреймворков, в частности рассмотренного JInterface, недостатком является то, что этот фреймворк не позволяет писать параллельную программу, согласно акторной модели, а позволяет только обращаться к Erlang системе.

1.4 Абстрактная модель системы взаимодействия акторов

Рассмотрим абстрактную модель взаимодействия акторов, на примере двух систем, написанных на языках Scala и Erlang. В качестве посредника будет служить так же акторная система. Схема такого взаимодействия представлена на рисунке 1.4.

На схеме представлены все элементы, входящие в такую систему. Scala System и Erlang System – это две существующие акторные системы. Написание системы, связывающей две данные, подразумевает наличие некоторого фреймворка для этого. Этот фреймворк будет предоставлять необходимые инструменты, для организации общения со Scala и Erlang системами. На схеме они обозначены как Scala Framework Interface и Erlang Framework Interface соответственно. Эти инструменты должны обеспечивать организацию общения данных систем, однако, при помощи написания собственных акторов, выполняющих функции транслятора из одной системы в другую. Так же фреймворк, должен будет позволять создавать клиентские акторы, которые смогут обращаться к такой системе.

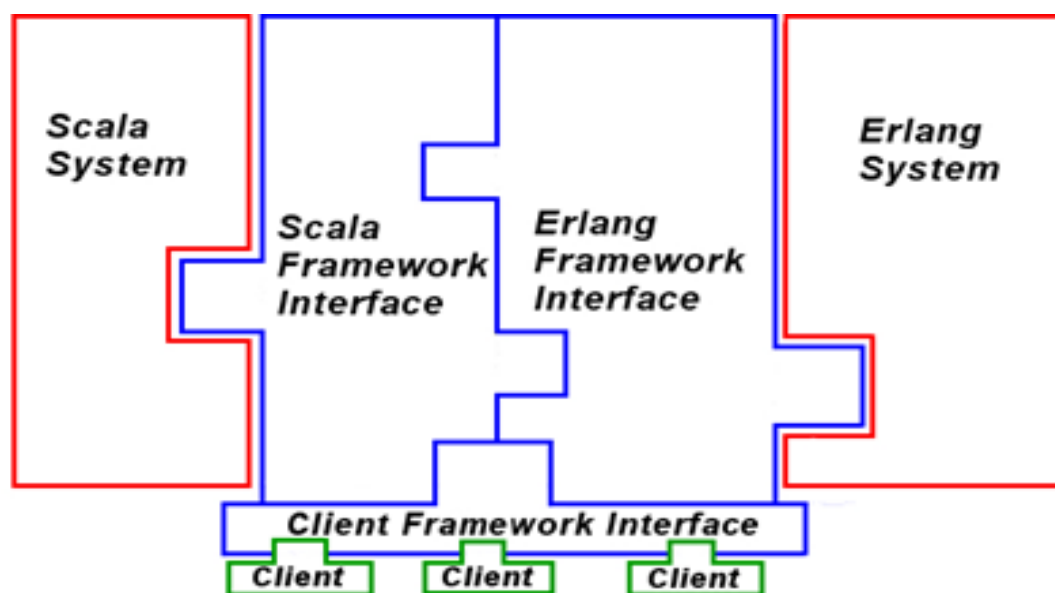


Рисунок 1.4 - Схема взаимодействия Scala и Erlang систем

Таким образом, наличие фреймворка, позволяющего организовывать общение между различными акторными системами при помощи все тех же акторов, позволит эффективно создавать распределенные системы на основе модели акторов с ее плюсами, вне зависимости от используемых языков.

1.5 Постановка задачи

Была поставлена задача реализовать фреймворк, который позволит использовать как Erlang, с рядом его уникальных решений, так и Scala с JVM и богатой инфраструктурой, для создания распределенных систем, использующих акторную систему. Именно благодаря высокому уровню абстракции акторов, должна быть возможна передача сообщений между системой, написанной на Erlang и системой, написанной на Scala, а также потенциально другими системами. Фреймворк должен позволять создавать связи между двумя подсистемами, а так же предоставлять возможность создания API для обращения к такой системе и получения от нее ответов.

Цель данной работы – проектирование и реализация фреймворка для написания систем распределения задач и реализация систем, демонстрирующей работу фреймворка.

2 Проектирование фреймворка

2.1 Обоснование выбора языка программирования

Для реализации фреймворка мы выбрали язык программирования Scala по нескольким причинам:

- Scala - кроссплатформенный язык программирования
- Наличие большого количества готовых библиотек и фреймворков, в том числе написанных на языке Java
- Язык реализует функциональную парадигму программирования со всеми преимуществами, описанными в разделе 1.2 данной работы
- Отсутствие проблемы, связанной с проектированием дополнительных классов для организации общения с акторами Scala

2.2 Область применения проектируемого фреймворка

Проектируемый фреймворк позволит создавать эффективные системы в областях, изображенных на рисунке 2.1.

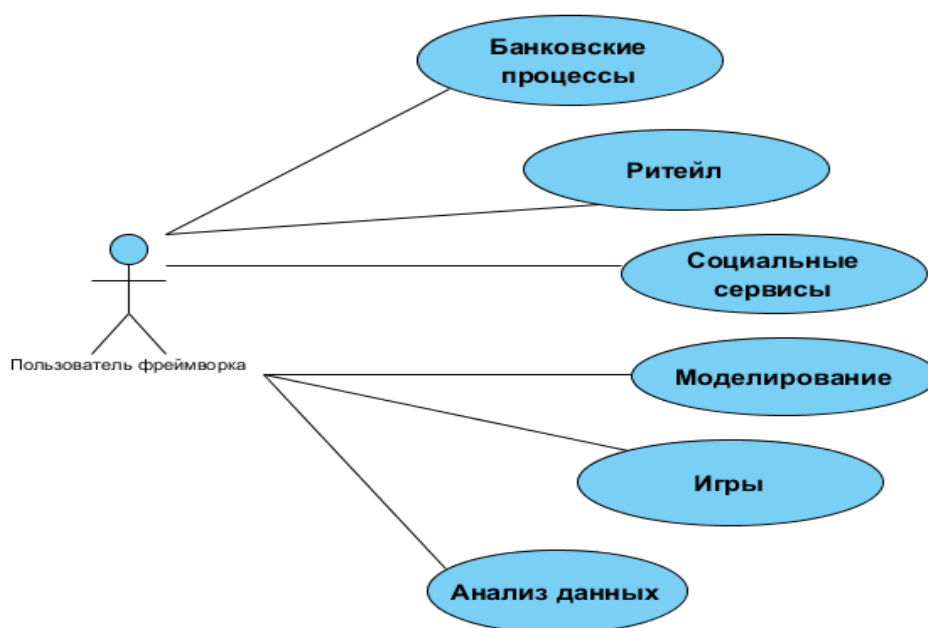


Рисунок 2.1- Области применения фреймворка

Это только часть областей, в которых может быть использован фреймворк. Любая система, с высокой загруженностью и малым временем отклика, хороший

кандидат для использования проектируемого фреймворка. Акторы позволят управлять ошибками во время эксплуатации системы, распределять нагрузку, а так же дают возможность горизонтальной и вертикальной масштабируемости[22].

2.3 Проектирование классов фреймворка

На рисунке 2.2 представляет UML диаграмму классов фреймворка. Синим цветом указаны спроектированные классы, фиолетовым - классы из библиотеки JInterface(OtpMbox, OtpNode, OtpErlangObject), розовым - классы из стандартной библиотеки Scala и библиотеки Akka(Actor, Any).

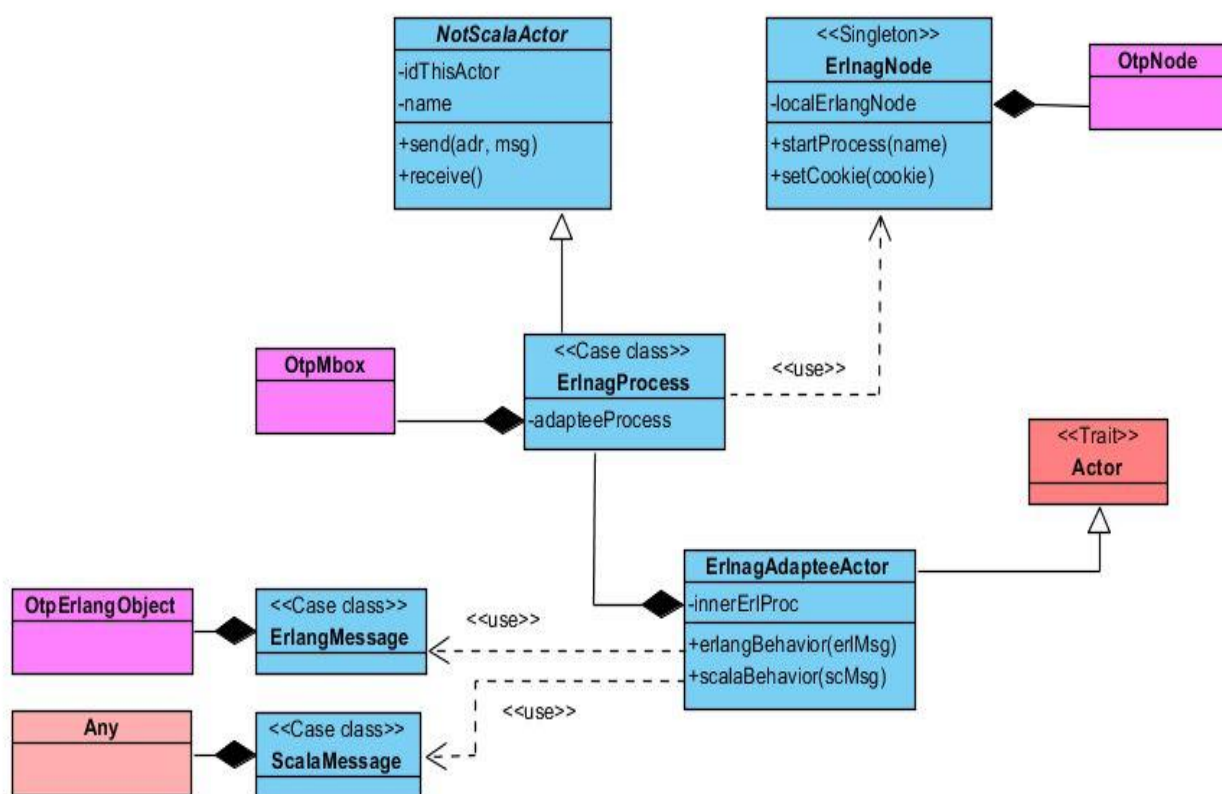


Рисунок 2.2 - UML диаграмма классов фреймворка

Главной идеей реализуемого фреймворка было создание такого класса, объекты которого были бы одновременно и Erlang процессами и Scala акторами. Таким классом стал **ErlangAdapteeActor**. Рассмотрим подробнее структуру спроектированных классов.

NotScalaActor - абстрактный класс для адаптируемых акторов. Каждый из акторов имеет свое имя (`name`) и идентификатор (`idThisActor`). Например, для Erlang

процессов идентификатором является pid. Так же каждый актор, согласно акторной модели, может принимать сообщения (receive) и отправлять сообщения другому актору (send).

ErlangProcess - кейс-класс[23], реализующий шаблон проектирования “Адаптер” [24] для класса OtpMbox из библиотеки JInterface. Адаптируемый класс OtpMbox приводится к интерфейсу NotScalaActor. Для создания объектов класса OtpMbox (поле adapteeProcess) внутри класса ErlangProcess, используется фабричный метод startProcess из класса ErlangNode, реализующего шаблон проектирования “Одиночка”[24].

ErlangNode - одиночный объект[25], представляющий собой обертку над классом OtpNode. Предоставляет фабричный метод startProcess для создания новых Erlang процессов, а так же метод setCookie, для настройки cookie[13] локального узла.

ErlangAdapteeActor - абстрактный класс, реализующий интерфейс Actor из библиотеки Akka, благодаря чему является Scala актором. Так же агрегирует объект класса ErlangProcess, при помощи которого он может общаться с другими Erlang процессами, в том числе написанными на языке Erlang. Класс предоставляет два метода для реализации: scalaBehavior и erlangBehavior. Акторы, создаваемые путем наследования от данного класса, используют в качестве принимаемых сообщений объекты кейс-классов ErlangMessage и ScalaMessage.

ScalaMessage - кейс-класс, внутри которого хранится сообщение типа Any, для обработки методом scalaBehavior из класса ErlangAdapteeActor.

ErlangMessage - кейс-класс, внутри которого хранится сообщение типа OtpErlangObject, для обработки методом erlangBehavior из класса ErlangAdapteeActor.

2.4 Архитектурные решения

Для проектирования данного фреймворка были использованы как шаблоны проектирования из объекто-ориентированного подхода к программированию, так и решения из функционального подхода. Рассмотрим эти решения.

2.4.1 Объектно-ориентированные решения

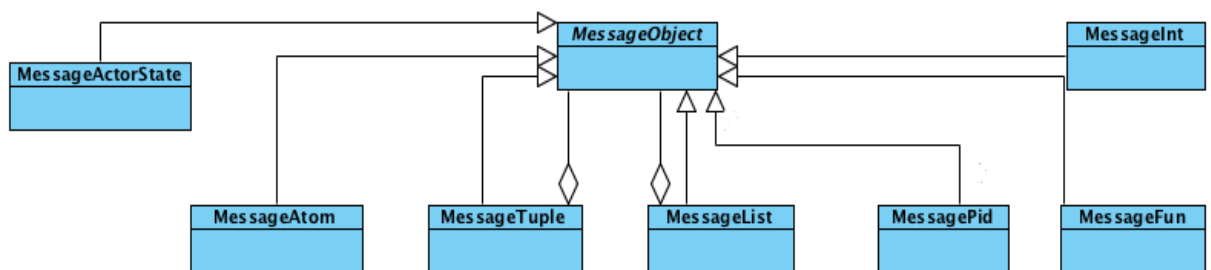
Шаблон проектирования “Адаптер”. Данный шаблон применяется в классе ErlangProcess. Целью использования данного паттерна было приведение класса OtpMbox к интерфейсу класса NotScalaActor.

Шаблон проектирования “Одиночка”. В Scala предусмотрен компактный механизм реализации данного паттерна при помощи одиночного объекта. Данный шаблон применяется для реализации ErlangNode, который хранит в себе экземпляр класса OtpNode. Разработчики JInterface рекомендуют создавать именно один локальный OtpNode[27] для общения с Java(Scala) кодом, поэтому было решено использовать этот паттерн.

Шаблон проектирования “Фабричный метод”. Данный шаблон реализуется для создания объектов класса OtpMbox при помощи объекта ErlangNode. Так же язык Scala неявно реализует этот паттерн при использовании кайс-классов.

2.4.2 Функциональные решения

Для реализации фреймворка, который основывается на обобщенном понятие актора, необходимо также реализовать и обобщенные типы сообщений для этих акторов. На рисунке 2.3 представлена диаграмма классов таких сообщений.



Рисунко 2.3 - UML диаграмма классов сообщений

Каждый класс в представленной диаграмме является оберткой над типами Erlang и Scala. Так, например, MessageObject будет оберткой над классами OtpErlangObject и Any. При расширении фреймворка на какой-либо другой язык также придется расширять и классы, представленные на диаграмме.

Однако есть другой способ обобщить используемые сообщения благодаря такому подходу, как типовые параметры. Благодаря типовым параметрам и ограничениям типов можно добиться того, что адаптируемые акторы всегда будут работать с сообщениями нужного типа. Другими словами, `ErlangProcess` всегда будет получать и отправлять сообщения типа `OtpErlangObject` или наследуемых от него. При этом, благодаря строгой типизации, компилятор выдаст ошибку, если мы попытаемся использовать недопустимый тип.

Рассмотрим код, реализующий данный подход.

```
abstract class NotScalaActor[TypeMessage](val name:String){
  type IdentifierOtherActor
  def send[T <: TypeMessage](adr: IdentifierOtherActor)(msg: T) : Unit
  def send[T <: TypeMessage](aname:String,node:String, msg: T) : Boolean
  def receive : Option[TypeMessage]
  def idThisActor : IdentifierOtherActor}
```

В приведенном коде `TypeMessage` является типовым параметром, который обозначает тип сообщений, с которым работает актор. Запись вида `def send[T <: TypeMessage](aname:String,node:String, msg: T)` означает, что тип параметра `msg` является классом или потомком класса, который был передан в типовой переменной `TypeMessage`. Так же в классе `NotScalaActor` поле `idThisActor` имеет тип `IdentifierOtherActor`. `IdentifierOtherActor` - это псевдоним для типа идентификатора актора адаптируемой системы.

Рассмотрим реализацию `ErlangProcess`.

```
case class ErlangProcess(override val name:String)
  extends NotScalaActor[OtpErlangObject](name){
  override type IdentifierOtherActor = OtpErlangPid
  def idThisActor : IdentifierOtherActor = adapteeProcess.self()
  . . .
}
```

Как видно из кода, `ErlangProcess` наследуется от `NotScalaActor`. В качестве типового параметра передается класс `OtpErlangObject`, гарантирующий нам, что `ErlangProcess` будет работать только с сообщениями этого типа и его подтипов. Под псевдонимом типа идентификатора `IdentifierOtherActor`, будем подразумевать тип `OtpErlangPid`.

Таким образом, использование типовых параметров позволило нам избавиться от побочной иерархии классов для сообщений, обеспечивая при этом должную надежность и гибкость кода.

2.5 Примеры решения задач

Как было указано в разделе 2.1 данной работы, проектируемый фреймворк позволит решать задачи разного рода и из разных предметных областей. Ниже будут приведены примеры решения задач из разных областей.

2.5.1 Пример “Файловая система на процессах”

Ниже будут представлены разделы, отражающие проектирование программы для решения задачи.

2.5.1.1 Постановка задачи

Задача относится к классу задач на взаимодействия нескольких систем, распределенных в одной сети.

Формулировка задачи: файлы на диске представлены процессами или акторами, которые получают сообщение и выполняют действие со своим файлом например, delete, copy и rename. Все процессы, отвечающие за обслуживание своего файла, работают асинхронно и параллельно.

Необходимо создать систему для работы с такой файловой системой. Запросы к такой системе должны выполняться асинхронно и параллельно.

2.5.1.2 Анализ задачи

Понятно, что файлов в файловой системе может быть очень много, и поэтому будет очень много процессов. Мы хотим, чтоб мощности одной средней машины хватало для запуска такой файловой системы, значит потоки должны быть легковесными, а значит реализовать их стоит на Erlang.

Такие файловые системы можно запустить на нескольких машинах одновременно, и возникает необходимость иметь к ним централизованный доступ для управления. Программа, реализующий такой доступ, должна иметь удобный интерфейс и запускаться с любой машины в сети. В Erlang интерфейсы реализовать довольно сложно, а также исполняющая среда специфична, поэтому управляющую систему лучше реализовать на Scala.

2.5.1.3 Диаграмма классов для решения задачи

На рисунке 2.4 представлена UML диаграмма классов для решения данной задачи.

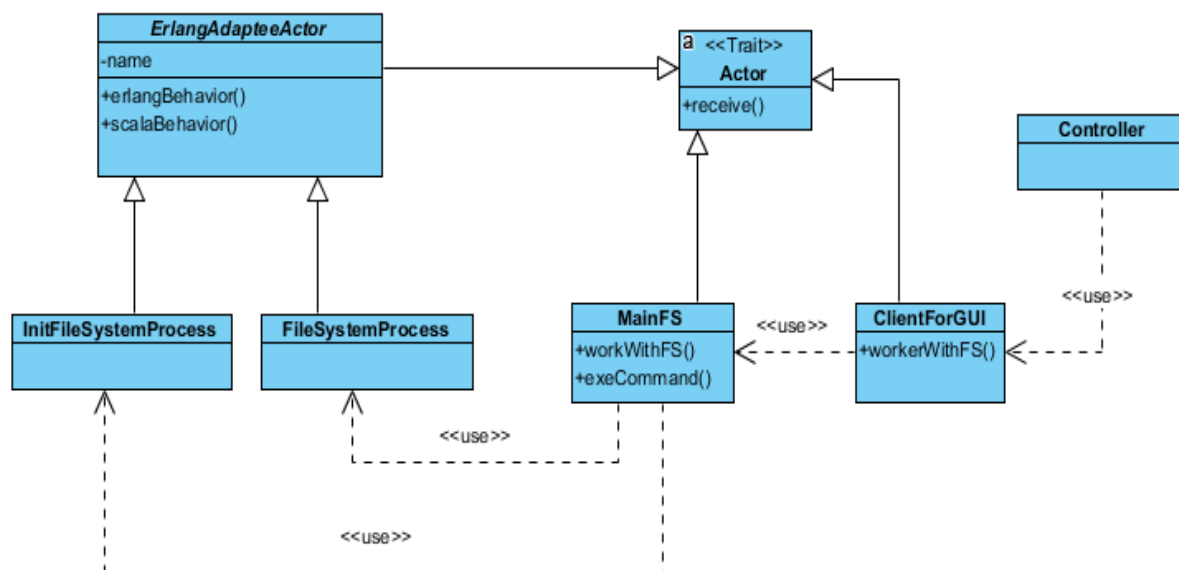


Рисунок 2.4 - Диаграмма классов данной задачи

Рассмотрим подробнее данную диаграмму.

ErlangAdapteeActor - абстрактный класс спроектированного фреймворка. Предоставляет два метода для реализации дочерними классами: `erlangBehavior` и `scalaBehavior`.

InitFileSystemProcess - класс, представляющий актор для инициализации файловой системы.

FileSystemProcess - класс, представляющий актор для выполнения команд файловой системы.

Actor - интерфейс акторов Scala из библиотеки Akka.

MainFS - класс, представляющий главный актор для работы с файловой системой. Этот актор имеет два состояния: состояние инициализации системы и состояние работы с системой. Также актор создает и использует экземпляры классов `InitFileSystemProcess` и `FileSystemProcess`.

ClientForGUI - класс, представляющий актор для связи пользовательского интерфейса с файловой системой. Этот актор имеет два состояния: состояние инициализации системы и состояние работы с системой. Также актор создает и использует экземпляр класса `MainFS`.

Controller - класс, предоставляющий контроллер для шаблона проектирования MVC. Этот класс инициализирует локальную систему акторов для работы с файловой системой, а также, благодаря специальным механизмам, общается с актором `ClientForGUI`.

Помимо классов, реализующих акторы системы, также необходимы классы, реализующие сообщения в этой системе. На рисунке 2.5 представлена UML диаграмма классов сообщений для инициализации системы.

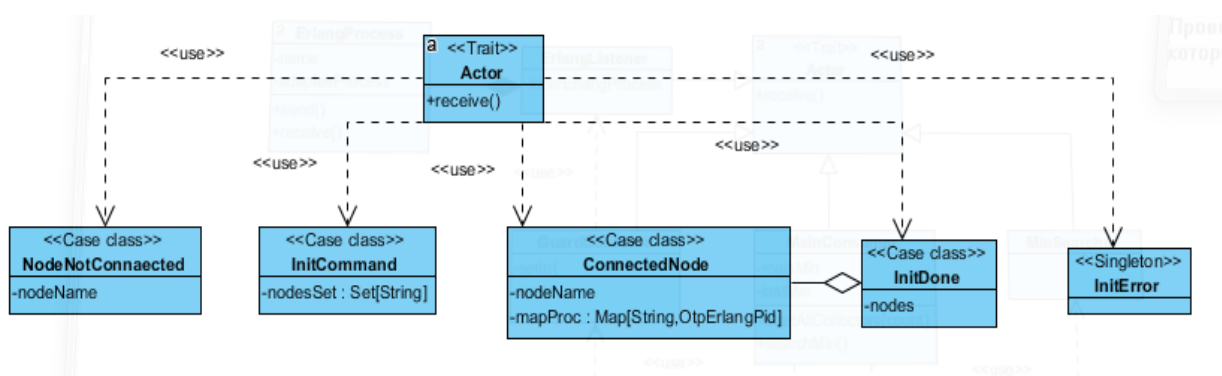


Рисунок 2.5 - Диаграмма классов сообщений для актора `MainFS`

NodeNotConnect - кейс-класс, представляющий сообщение об не подключенном узле системы. Поле `nodeName`- имя не подключенного узла.

InitCommand - кейс-класс, представляющий сообщение-команду об начале инициализации файловой системы. Поле `nodesList` - множество узлов, на которых должна быть запущена файловая система.

ConnectedNode - кейс-класс, представляющий сообщение об подключенном узле системы. Поле `nodeName`- содержит имя подключенного узла, а поле `mapProc` -

является ассоциативным массивом, ключами которого являются имена файлов на подключенном узле, а значения - это идентификаторы процессов (pid) обслуживающих соответствующий файл.

InitDone - кейс-класс, представляющий сообщение об завершенной инициализации. Поле nodes - множество подключенных узлов.

InitError - кейс-объект, представляющий сообщение об ошибке инициализации.

На рисунке 2.6 представлена диаграмма классов сообщений для работы с файловой системой.

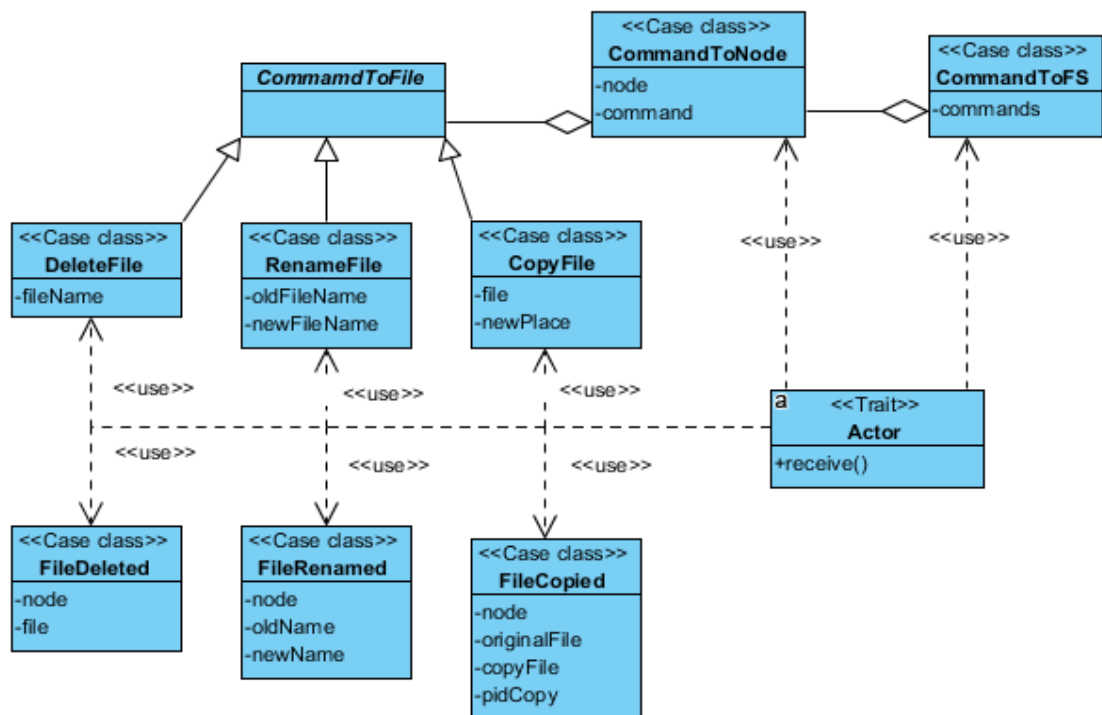


Рисунок 2.6 - Диаграмма классов сообщений для работы с системой

CommandToFile - абстрактный класс для команд к файловой системе.

DeleteFile - кейс-класс, представляющий команду для удаления файла. Поле fileName - имя удаляемого файла.

RenameFile - кейс-класс, представляющий команду для переименования файла. Поле oldFileName - оригинальное имя файла, поле newFileName - новое имя файла.

CopyFile - кейс-класс, представляющий команду для копирования файла. Поле `file` - имя копируемого файла, поле `newPlace` - имя скопированного файла.

CommandToNode - кейс-класс, представляющий команду для файла на определенном узле. Поле `node` - имя узла, поле `command` - экземпляр класса `CommandToFile`.

CommandToFS - кейс-класс, представляющий набор команд ко всей файловой системе. Поле `commands` - множество экземпляров класса `CommandToNode`.

FileDeleted - кейс-класс, представляющий сообщение об успешном удалении файла. Поле `node` - имя узла, на котором был удален файл, поле `file` - имя удаленного файла.

FileRenamed - кейс-класс, представляющий сообщение об успешном переименовании файла. Поле `node` - имя узла, на котором файл был изменен, поле `oldName` - старое имя файла, поле `newName` - новое имя файла.

FileCopied - кейс-класс, представляющий сообщение об успешном копировании файла. Поле `node` - имя узла, на котором файл был изменен, поле `originalFile` - оригинальное имя файла, поле `copyFile` - имя скопированного файла, поле `pidCopy` - `pid` процесса обслуживающего новый файл.

2.5.1.4 Диаграммы действий акторов

На рисунке 2.7 представлена use-case диаграмма, описывающая действия акторов системы во время инициализации.

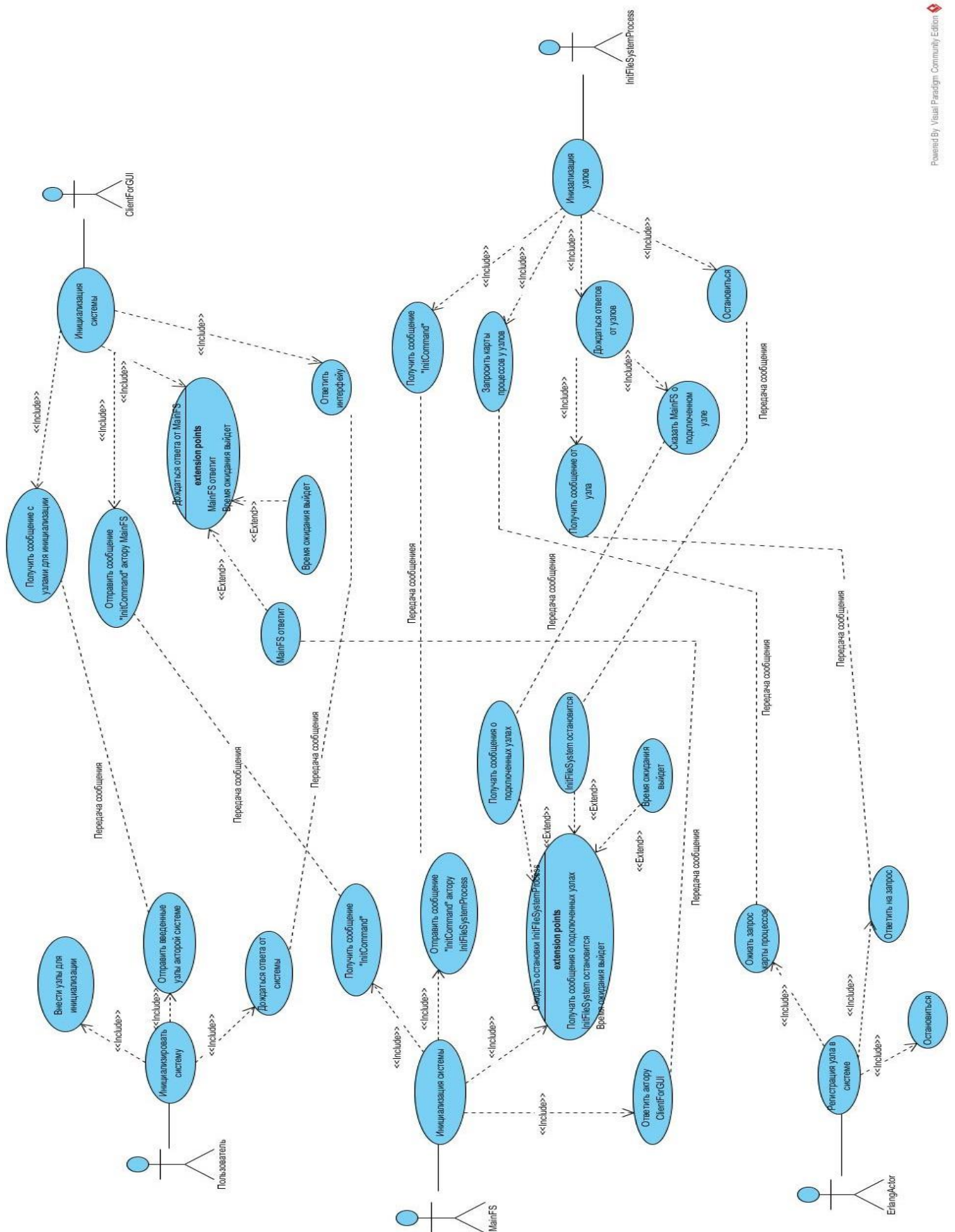


Рисунок 2.7 - Поведение акторов системы во время инициализации

Ниже рассмотрим подробнее поведение каждого актора.

На рисунке 2.8 представлена use-case диаграмма, описывающая действия пользователя во время инициализации системы.

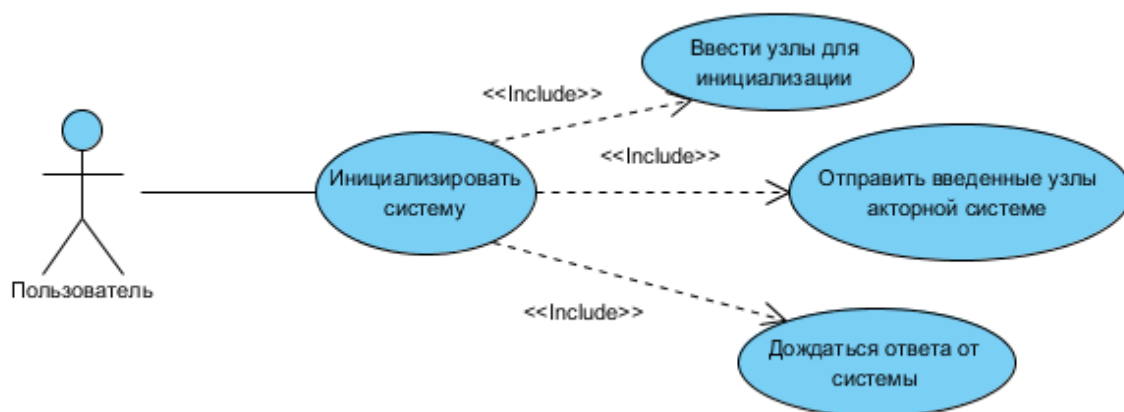


Рисунок 2.8 - Действия пользователя для инициализации системы

Пользователь во время инициализации выполняет следующие действия:

- Вводит имена узлов для инициализации.
- Отправляет введенные узлы акторной системе для выполнения инициализации.
- Ожидает ответ от системы, либо получает ошибку об повешении лимита времени ожидания.

На рисунке 2.9 представлена use-case диаграмма, описывающая поведение актора ClientForGUI во время инициализации.

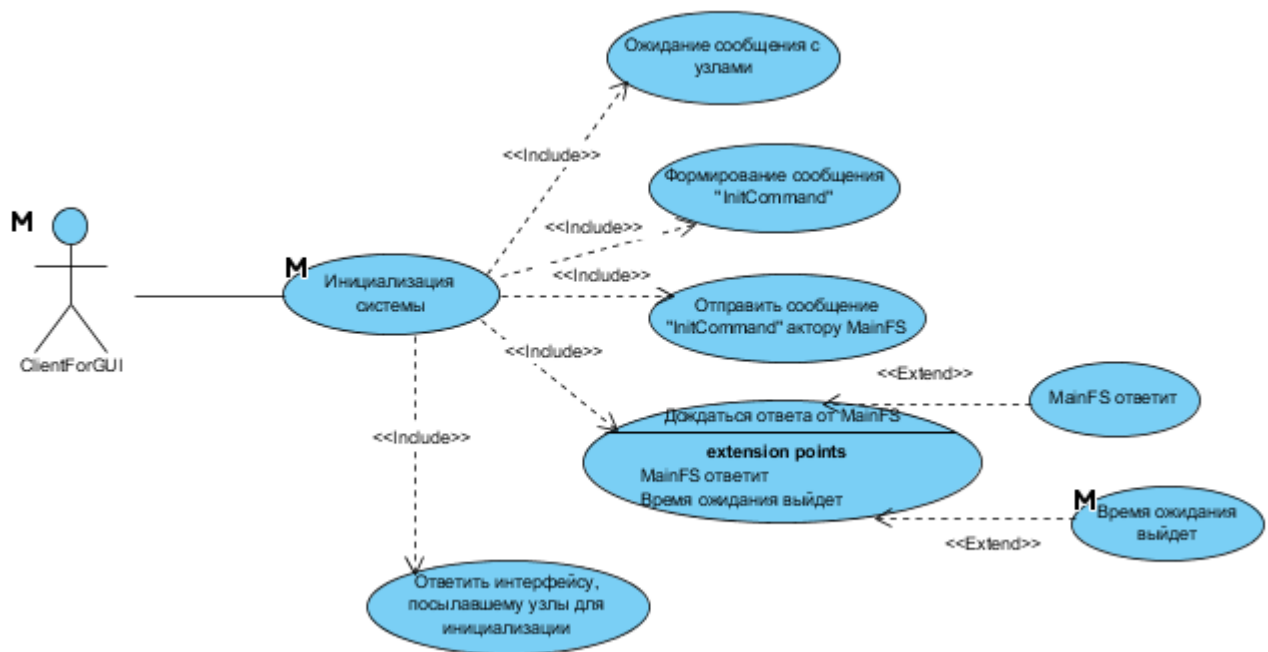


Рисунок 2.9 - Действия актора ClientForGUI во время инициализации

Актор ClientForGUI выполняет следующие действия в состоянии инициализации системы:

- Ожидает сообщение с узлами для инициализации от пользовательского интерфейса.
- На основе полученных узлов формирует сообщение “InitCommand”.
- Отправляет сформированное сообщение “InitCommand” актору MainFS.
- Дождается ответа от актора MainFS, при этом актор может успеть ответить, а также может закончиться время ожидания ответа.
- В зависимости от полученного ответа от MainFS, формируется сообщение для пользовательского интерфейса и отправляется ему.

На рисунке 2.10 представлена use-case диаграмма, описывающая поведение актора MainFS во время инициализации.

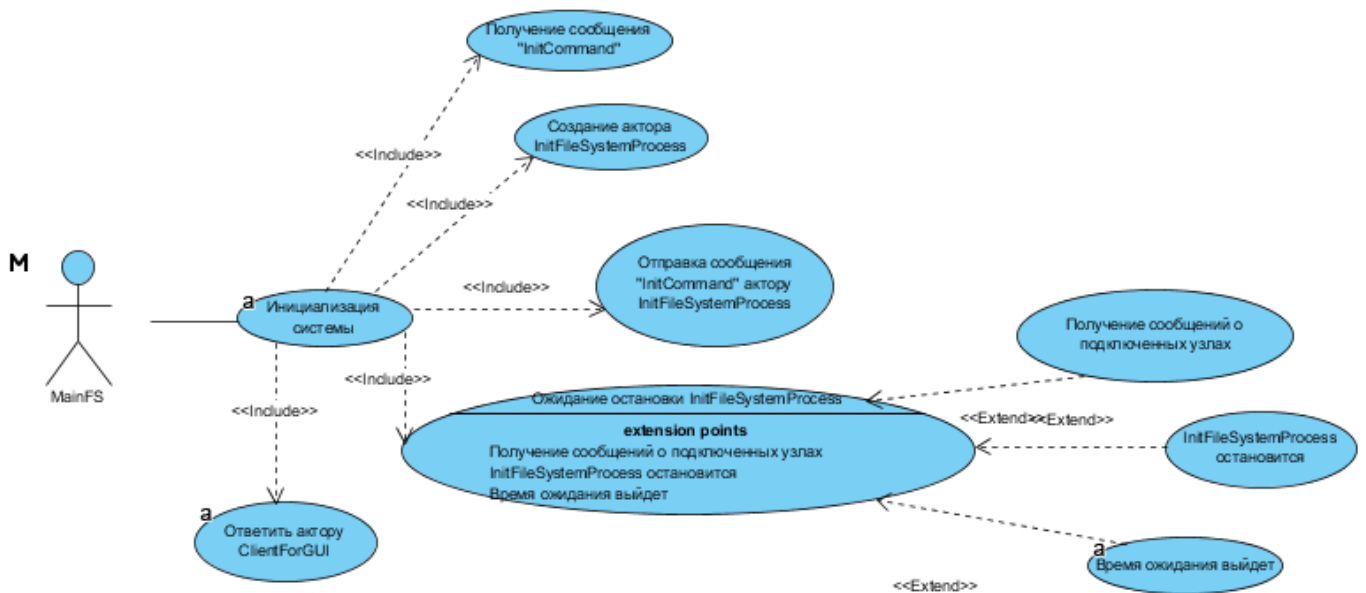


Рисунок 2.10 - Действия актора MainFS во время инициализации

Актор MainFS выполняет следующие действия в состоянии инициализации системы:

- Ожидает сообщение “InitCommand”.
- Создает актор InitFileSystemProcess для делегации ему обязанностей по инициализации.
- Отправляет сообщение “InitCommand” актору InitFileSystemProcess.
- Ожидает остановки актора InitFileSystemProcess, при этом получает сообщения о подключенных узлах. Актор InitFileSystemProcess может успеть остановиться, либо время ожидания выйдет.
- На основе сообщений о подключенных узлах, формирует сообщение “InitDone” или “InitError” и отправляет актору ответ ClientForGUI.

На рисунке 2.11 представлена use-case диаграмма, описывающая поведение актора InitFileSystemProcess.

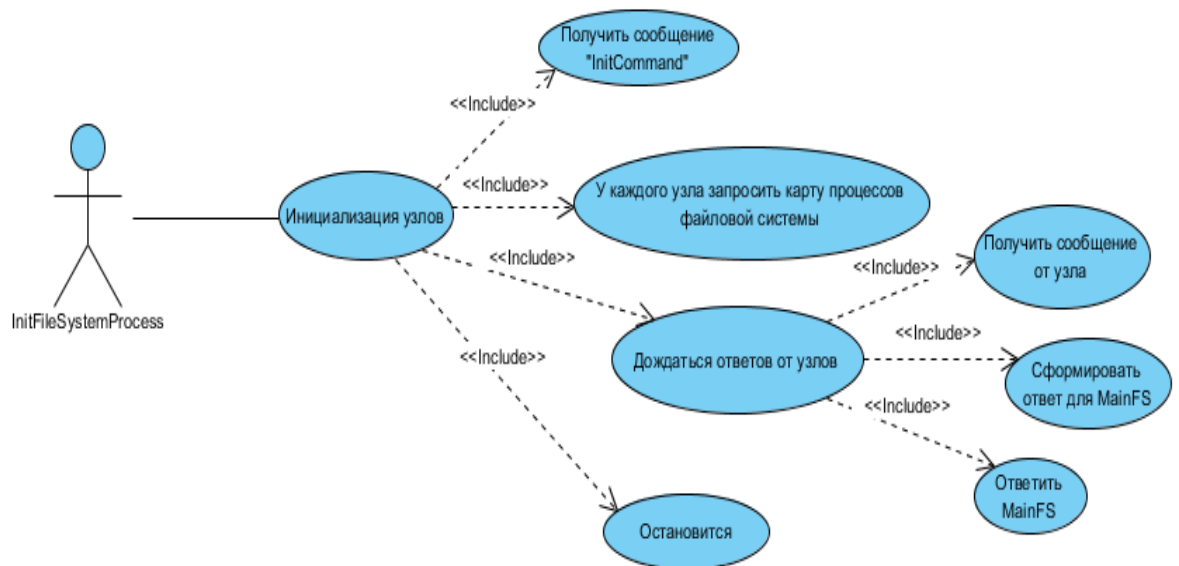


Рисунок 2.11 - Действия актора `InitFileSystemProcess`

Актор `InitFileSystemProcess` выполняет следующие действия при инициализации узлов:

- Получает сообщение "InitCommand".
- Каждому узлу из сообщения "InitCommand" отправляет запрос на получение карты процессов узла.
- Получает ответы от подключенных узлов, формирую сообщения "NodeConnected" для MainFS.
- После получения всех ответов, актор останавливается.

На рисунке 2.12 представлена use-case диаграмма, описывающая поведение актора, который инициализирует файловую систему на Erlang узле.

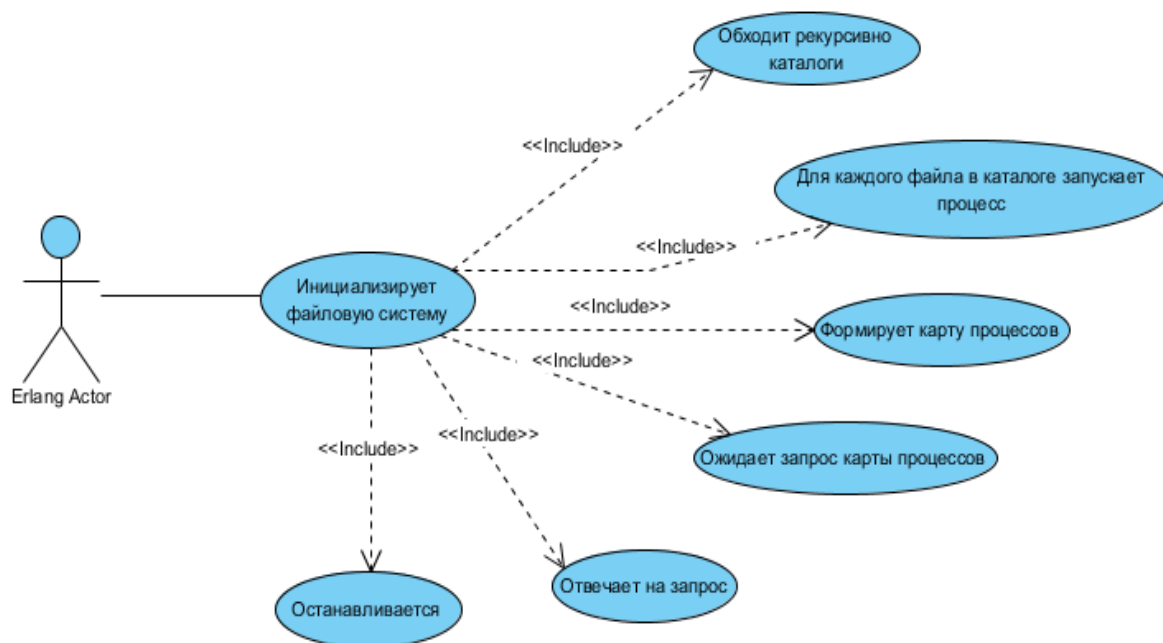


Рисунок 2.12 - Поведение Erlang актора

Erlang процесс выполняет следующие действия при регистрации узла в системе:

- Формирует карту процессов, обходя все каталоги и запуская для каждого файла в каталоге процесс.
- Ожидает сообщения подключения узла к системе.
- Отвечает за запрос карты процессов.
- Останавливается.

На рисунке 2.13 представлена use-case диаграмма, описывающая действия акторов во время работы с системой.

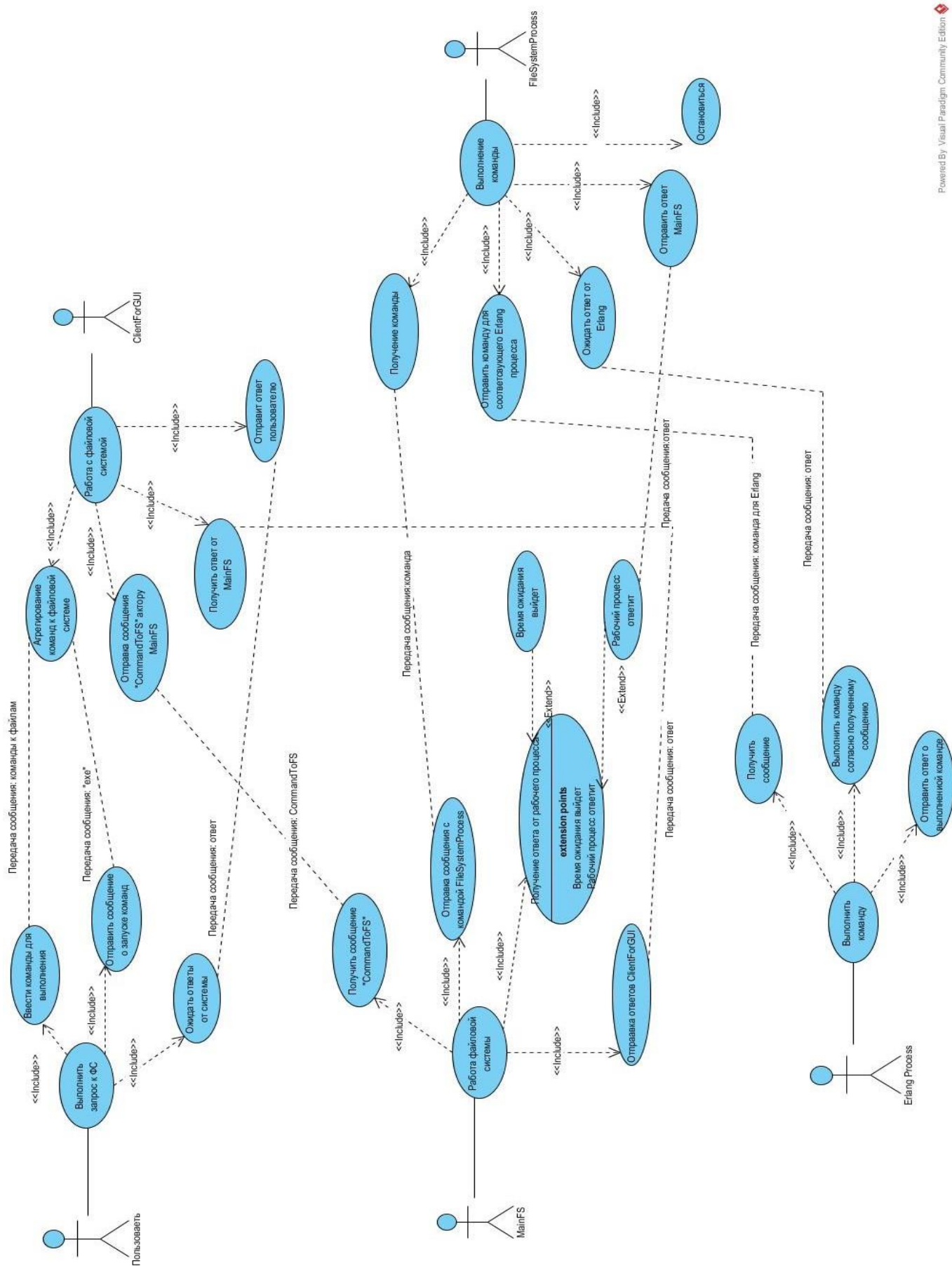


Рисунок 2.13 - Действия акторов во время работы с системой

Ниже рассмотрим подробнее поведение каждого актора.

На рисунке 2.14 представлена use-case диаграмма, описывающая действия пользователя во время инициализации системы.

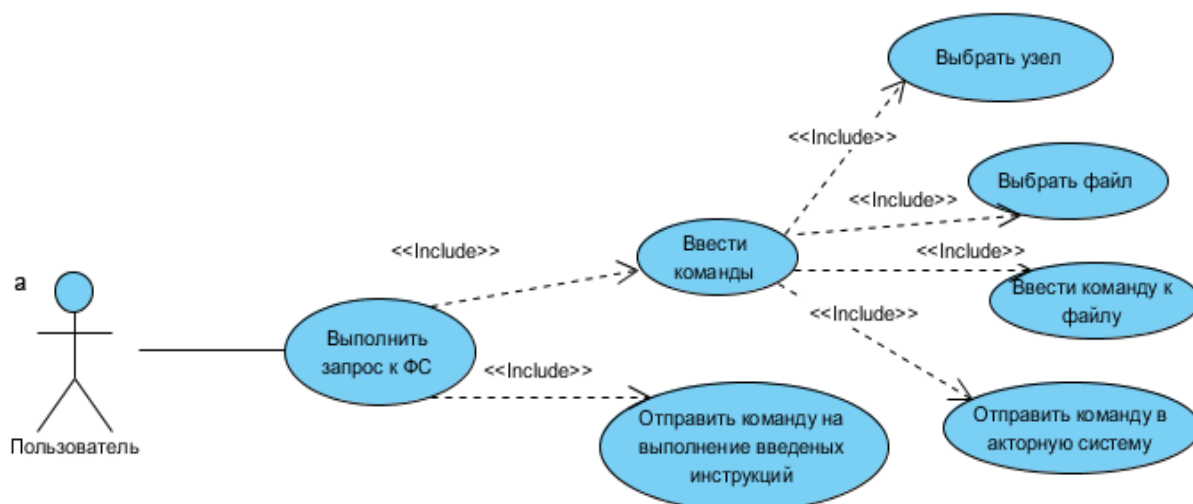


Рисунок 2.14 - Действия пользователя при работе с файловой системой

Пользователь во время работы с системой выполняет следующие действия:

- Вводит команды. Для этого пользователь выбирает узел, выбирает файл, к которому он хочет отправить команду, вводит команду и отправляет команду в систему.
- После ввода всех необходимых команд пользователь отправляет сообщение о выполнении введенных команд к системе.

На рисунке 2.15 представлена use-case диаграмма, описывающая действия актора ClientForGUI в состоянии работы с системой.

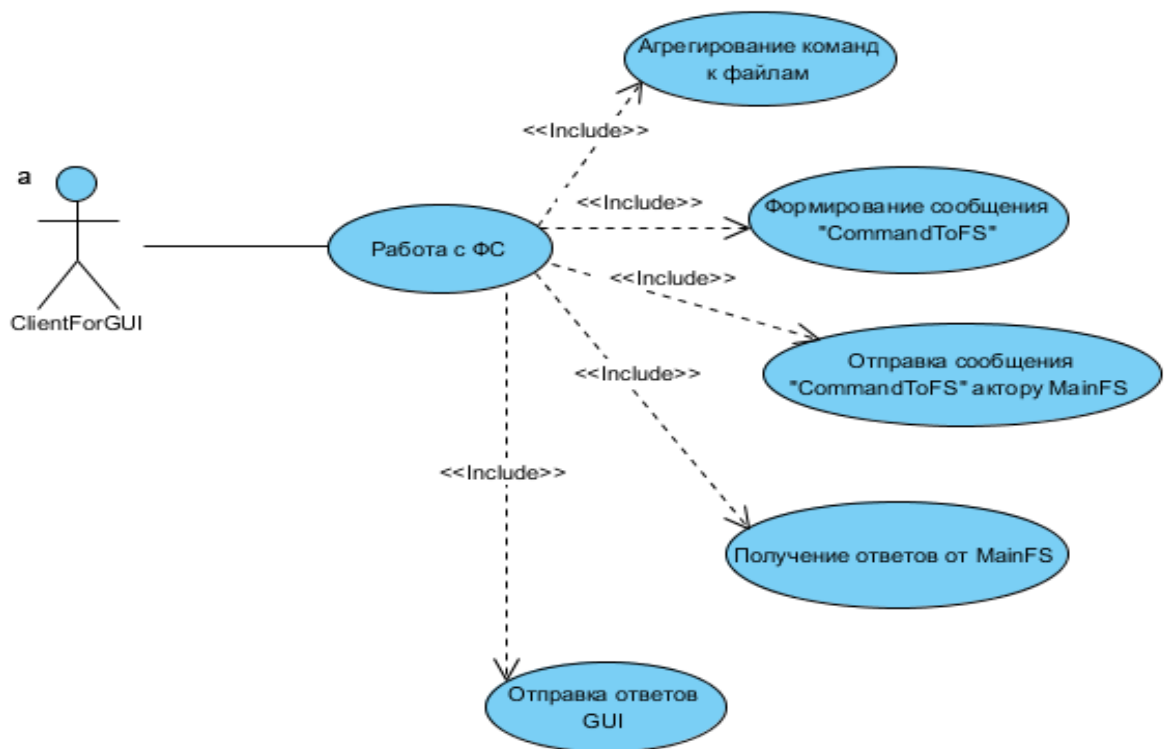


Рисунок 2.15 - Действия актора ClientForGUI в состоянии работы с системой
 Актор ClientForGUI выполняет следующие действия в состоянии работа с системой:

- Агреггирует команды от пользователя. При этом приводит каждую команду к одному из подклассов класса “CommandToFile”.
- После получения сообщения об выполнении всех агрегированных команд, формирует сообщение “CommandToFS”.
- Отправляет сообщение “CommandToFS” актору MainFS. Все введенные команды забываются актором.
- Получает новые команды от пользователя и ответы о выполненных операциях от MainFS.
- Отправляет ответы о выполненных операциях интерфейсу пользователя.

На рисунке 2.16 представлена use-case диаграмма, описывающая действия актора MainFS в состоянии работы с системой.

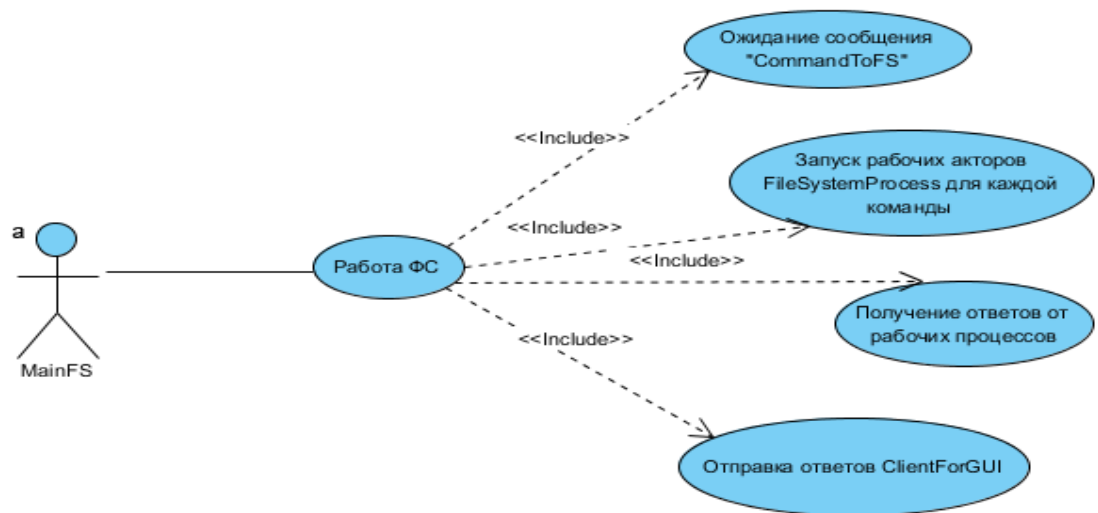


Рисунок 2.16 - Действия актора MainFS в состоянии работы с системой

Актор MainFS выполняет следующие действия в состоянии работы с системой:

- Ожидает сообщения “CommanfToFS”.
- Создает отдельный поток для запуска рабочих процессов FileSystemProcess для выполнения команд к файловой системе.
- Получает ответы от рабочих процессов.
- Отправляет ответы актору ClientForGUI.

На рисунке 2.17 представлена use-case диаграмма, описывающая действия актора FileSystemProcess.

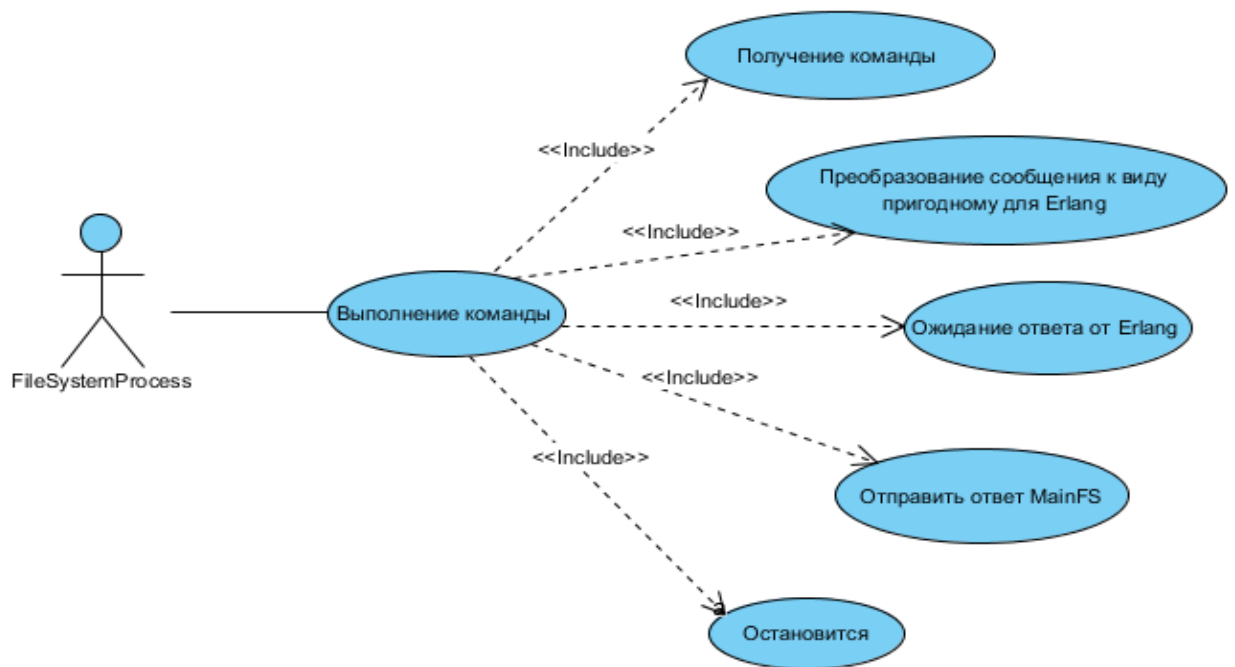


Рисунок 2.17 - Действия актора FileSystemProcess

Актор FileSystemProcess выполняет следующие действия во время выполнения команды:

- Получает команду.
- Преобразует команду к сообщению для Erlang процесса для выполнения команды.
- Отправляет сформированное сообщение.
- Ожидает ответ от Erlang процесса.
- Отправляет ответ MainFS.
- Останавливается.

На рисунке 2.18 представлена use-case диаграмма, описывающая действия Erlang актора, обслуживающего файл.

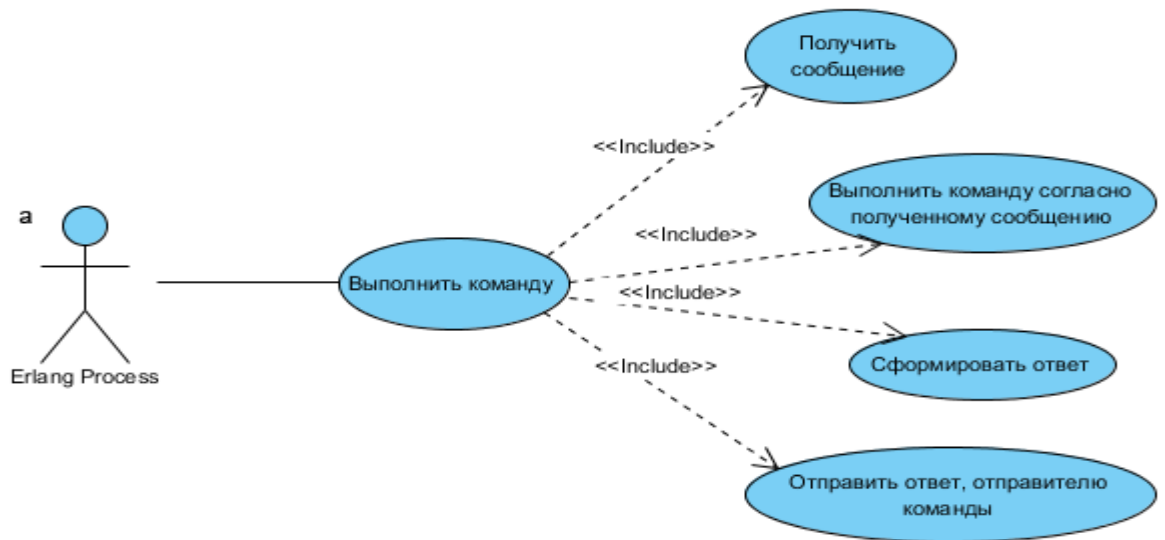


Рисунок 2.18 - Действия Erlang актора обслуживающего файл

Erlang процесс выполняет следующие действия во время выполнения команды:

- Получает сообщение.
- Ищет действия, которое необходимо выполнить согласно полученному сообщению.
- Выполняет действие.
- Формирует ответ о выполненном действии.
- Отправляет ответ на pid процесса, который был указан в полученном сообщении.

2.5.1.5 Диаграммы взаимодействия акторов

На рисунке 2.19 представлена диаграмма общения акторов во время инициализации файловой системы. На диаграмме указаны сообщения, которые передаются между акторами во время инициализации системы.

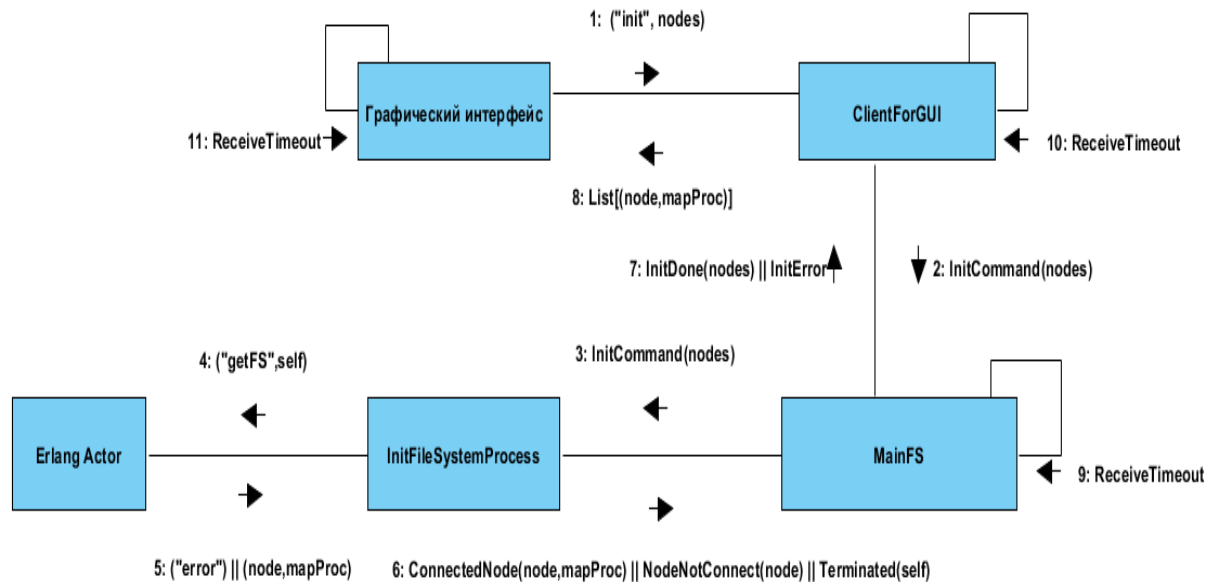


Рисунок 2.19 - Диаграмма общения акторов во время инициализации

На рисунке 2.20 представлена диаграмма общения акторов во время выполнения запросов к файловой системе. На диаграмме указаны сообщения, которые передаются между акторами во время работы системы.

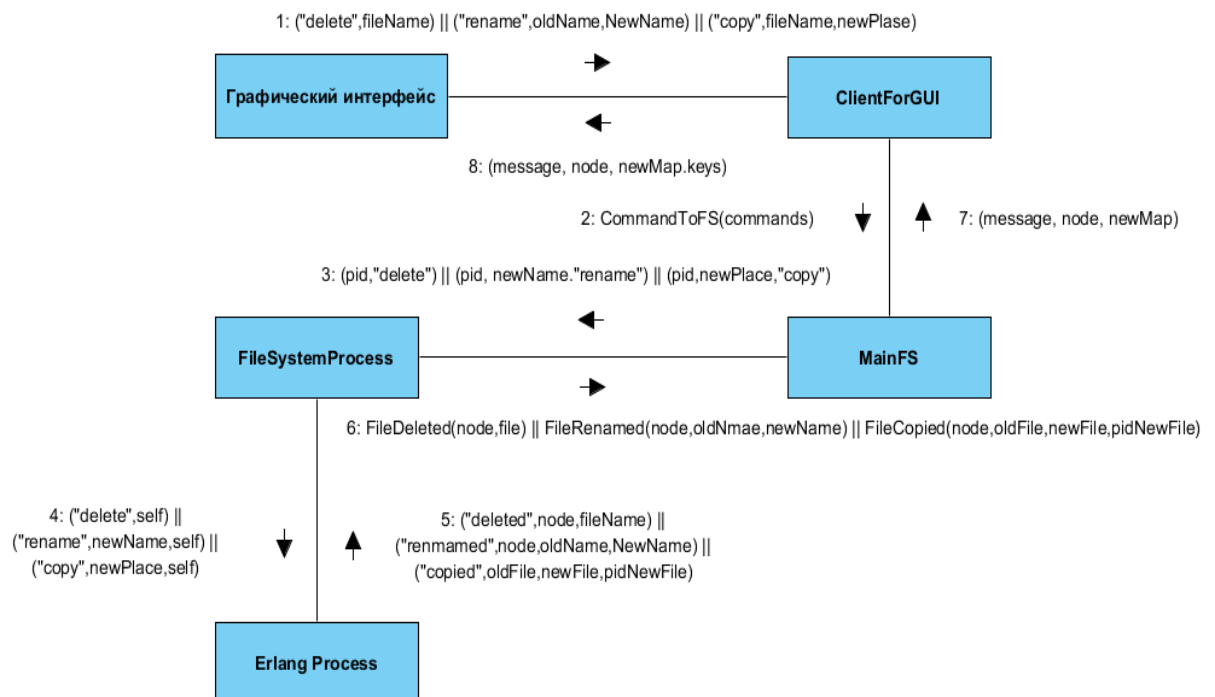


Рисунок 2.20 - Диаграмма общения акторов во время работы системы

2.5.2 Пример “Поиск минимального значения в непрерывном потоке данных”

Ниже будут представлены разделы, отражающие проектирование программы для решения задачи.

2.5.2.1 Постановка задачи

Задача относится к классу задач на обработку больших объемов данных с агрегацией результатов.

Формулировка задачи: имеется большое количество источников данных (свыше 15тыс). В произвольный промежуток времени они генерируют случайное число, однако интервал времени между двумя генерациями не превышает 10 секунд. Генерация каждого числа происходит независимо от других.

Необходимо создать систему, которая бы искала глобальный минимум из всех сгенерированных данных, а также позволяла узнать минимум в последней порции полученных данных.

2.5.2.2 Анализ задачи

В качестве источников данных лучше выбрать Erlang процессы, так как их достаточно много, а производительность генераторов не критична.

Обработка данных при этом должна быть достаточно быстрой и Scala по вычислительной производительности превосходит Erlang, поэтому систему обработки данных лучше написать на Scala. Количество сообщений в очереди к актору ограничено и, чтобы успокоить такой напор данных, будем использовать несколько ScalaCollector. Это отдельные акторы, их не так много, как производителей, и они служат для промежуточного сбора данных и обработки данных (как минимум хранят данные в множествах, чтоб избавиться от ненужных повторений).

2.5.2.3 Диаграмма классов для решения задачи

На рисунке 2.21 представлена UML диаграмма классов для решения данной задачи.

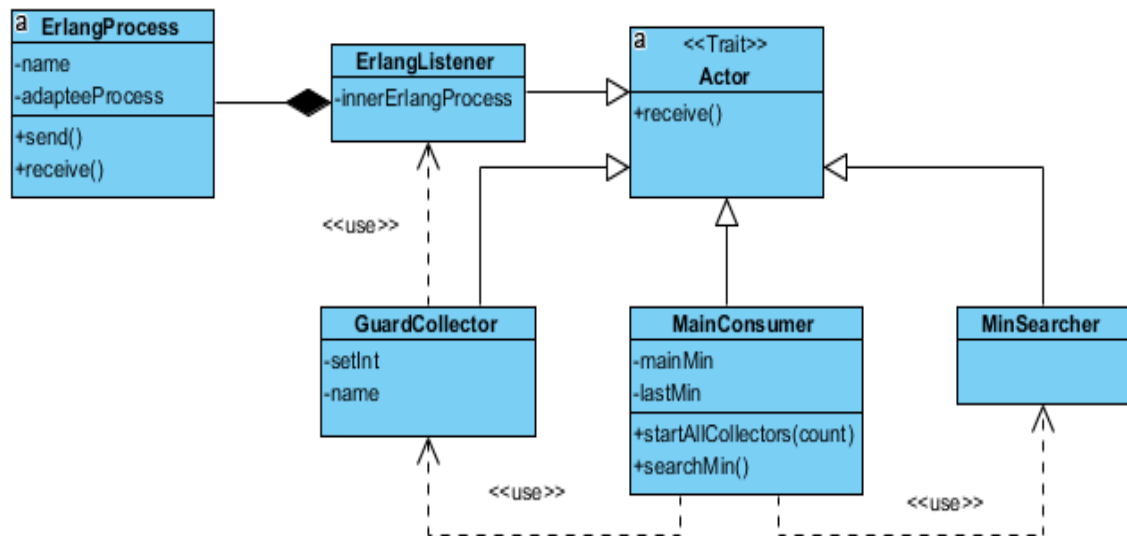


Рисунок 2.21 - Диаграмма классов данной задачи

Рассмотрим подробнее данную диаграмму.

ErlangProcess - класс из спроектированного фреймворка. Представляет собой обертку над классом `OtpMbox`.

ErlangListener - класс, представляющий собой актор, который слушает Erlang процессы и пересылает все полученные сообщения своему **GuardCollector**. Данная реализация не использует **ErlangAdapteeActor**, так как важна скорость приема сообщений, которая у реализации с использованием **ErlangAdapteeActor** будет ниже в связи с особенностью реализации.

Actor - интерфейс акторов Scala из библиотеки Akka.

GuardCollector - класс, представляющий актор для агрегирования данных и первичной обработки. Поле `setInt` - множество чисел, которые получает коллектор. При достижении определенного количества чисел, это множество отправляется актору **MainConsumer** и обнуляется. Поле `name` - имя **ErlangListener**, которого использует.

MainConsumer - класс, представляющий актора, который организует поиск минимального значения и хранит искомые величины в своем состоянии. Поле `mainMin` - глобальный минимум, `lastMin` - минимум с последней полученной порции данных. Метод `startAllCollector` - запускает коллекторы, количество которых было передано в сообщении к актору `MainConsumer`. Метод `searchMin` - делит полученные данные на порции и запускает акторы `MinSearcher` для поиска минимума в этих порциях.

MinSearcher - класс, представляющий актора для поиска минимум в небольшой порции данных.

Помимо классов, реализующих акторы системы, также необходимы классы, реализующие сообщения в этой системе. На рисунке 2.22 представлена UML диаграмма классов сообщений для инициализации системы.

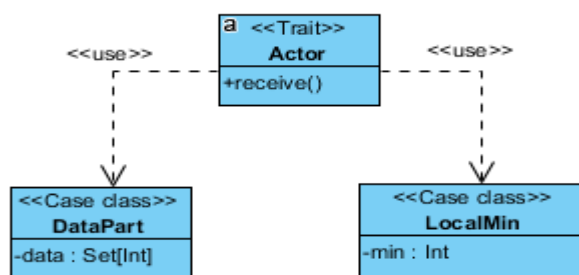


Рисунок 2.22 - Диаграмма классов сообщений

DataPart - кейс-класс, представляющий собой сообщение с порцией данных для обработки. Поле `data` - множество чисел, среди которых будет.

LocalMin - кейс-класс, представляющий собой сообщение с найденным минимумом. Поле `min` - минимальное число из порции данных.

2.5.2.4 Диаграммы действий акторов

На рисунке 2.23 представлена use-case диаграмма, описывающая действия акторов во время работы системы.

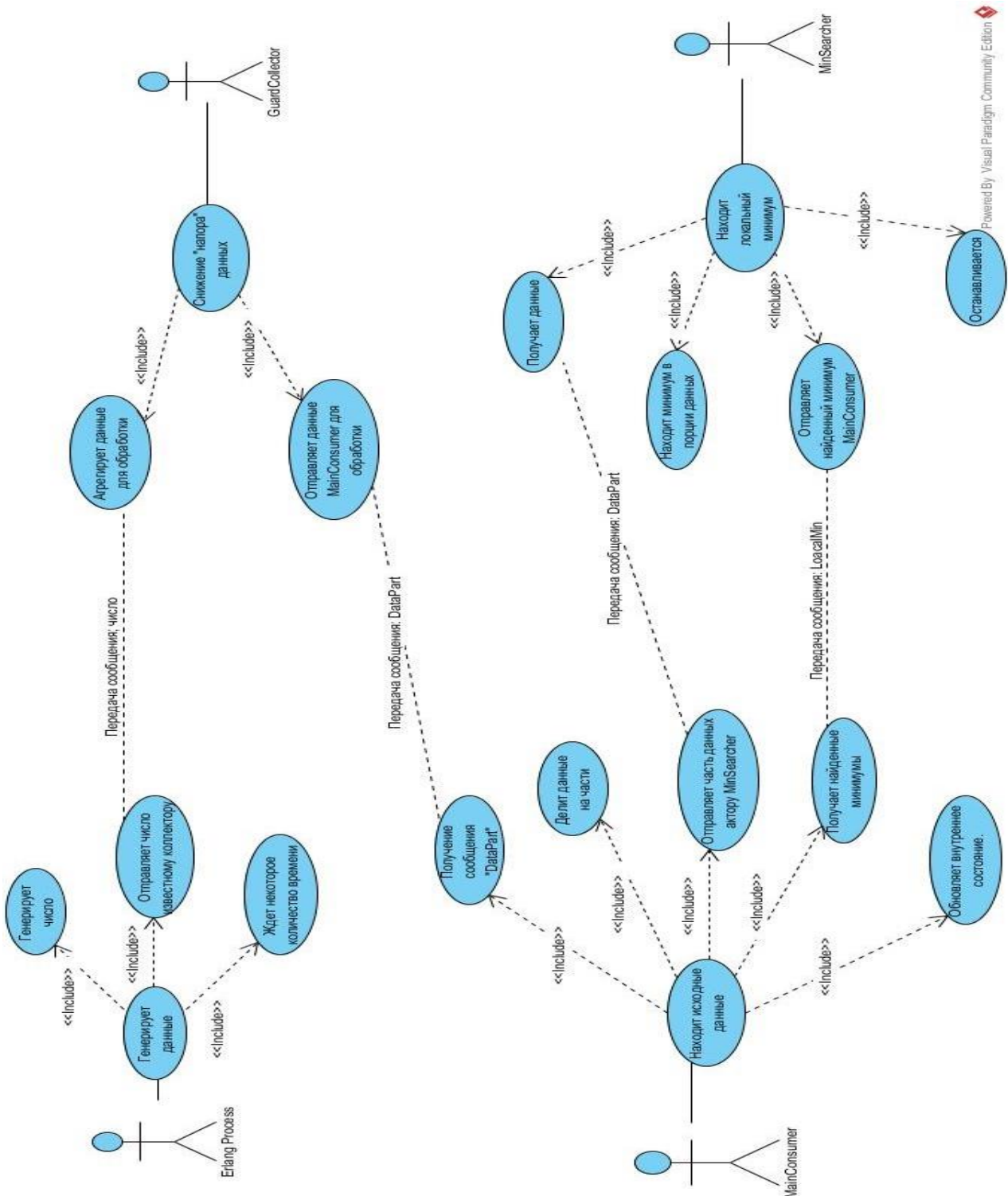


Рисунок 2.23 - Действия акторов во время работы системы

На рисунке 2.24 представлена use-case диаграмма, описывающая действия актора ErlangListener.

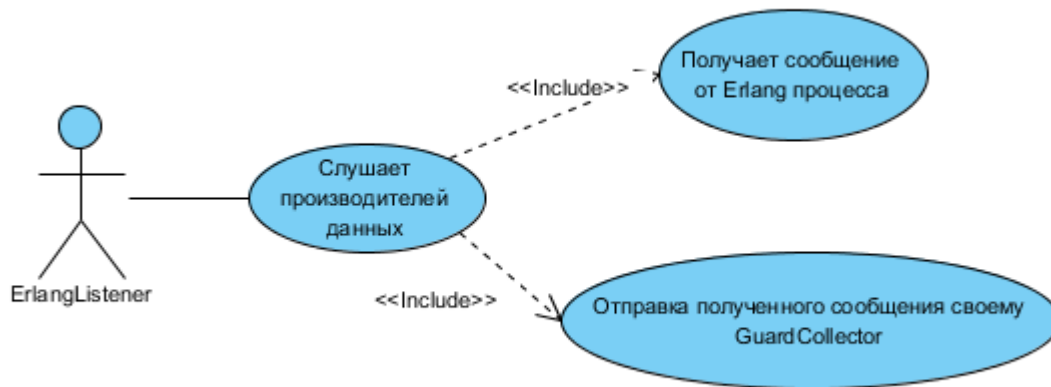


Рисунок 2.24 - Действия актора ErlangListener

ErlangListener во время работы выполняет следующие действия:

- Получает сообщение от Erlang процесса.
- Преобразование данных к Scala типу.
- Передает преобразованные данные своему GuardCollector.

На рисунке 2.25 представлена use-case диаграмма, описывающая действия актора GuardCollector.

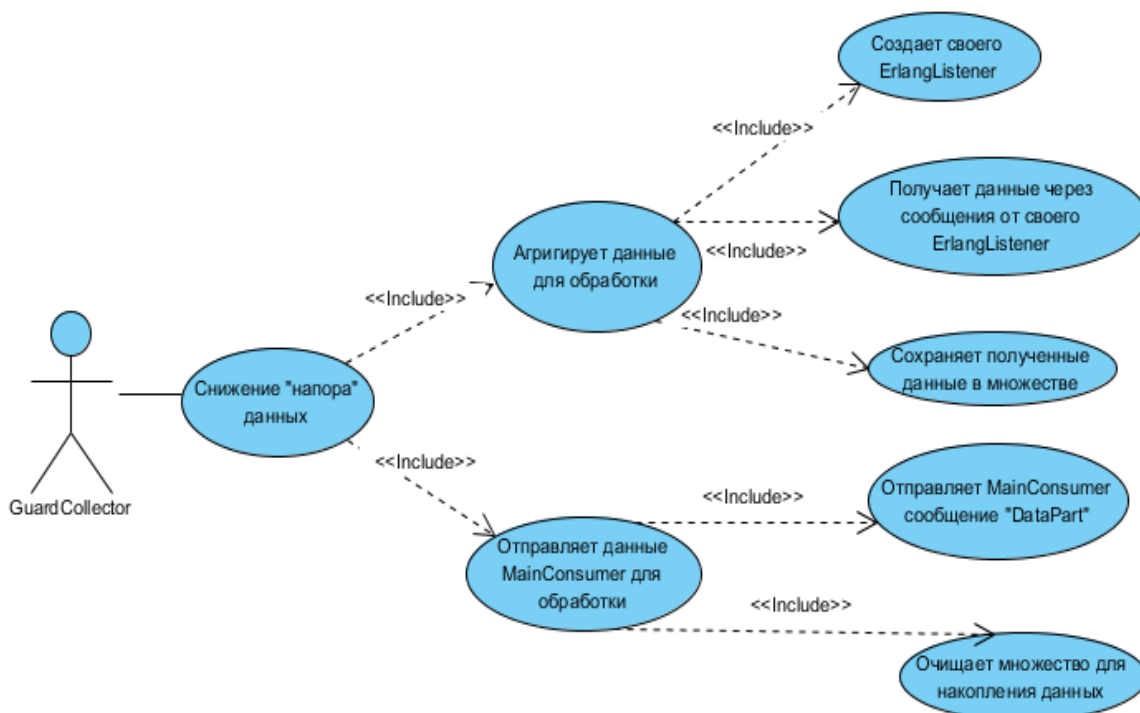


Рисунок 2.25 - Действия актора GuardCollector

Актор GuardCollector выполняет следующие действия во время своей работы:

- Создает помощника актора ErlangListener для получения сообщений от Erlang процессов.
- Получает данные и сохраняет их в множество.
- После накопления определенного количества данных отправляет их в сообщении “DataPart” актору MainConsumer.
- Отправленные данные удаляются из актора.

На рисунке 2.26 представлена use-case диаграмма, описывающая действия актора MainConsumer.

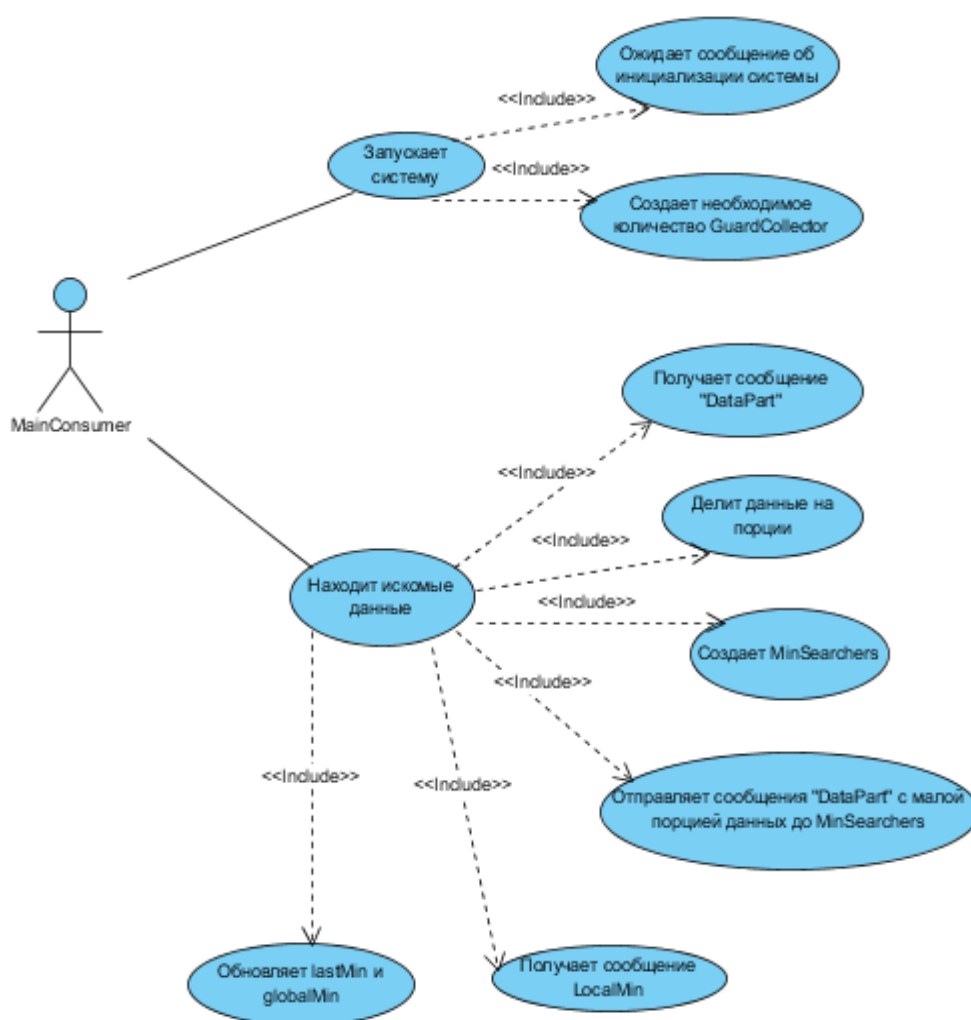


Рисунок 2.26 - Действия актора MainConsumer

Актор MainConsumer во время работы системы выполняет следующие действия:

- Получает данные в сообщении “DataPart”.

- Делит данные на порции, создавая для каждой порции MinSearcher.
- Отправляет каждому MinSearcher порцию данных.
- Получает ответы с LocalMin от рабочих процессов.
- Обновляет состояние на основе полученных данных

На рисунке 2.27 представлена use-case диаграмма, описывающая действия актора MinSearcher.

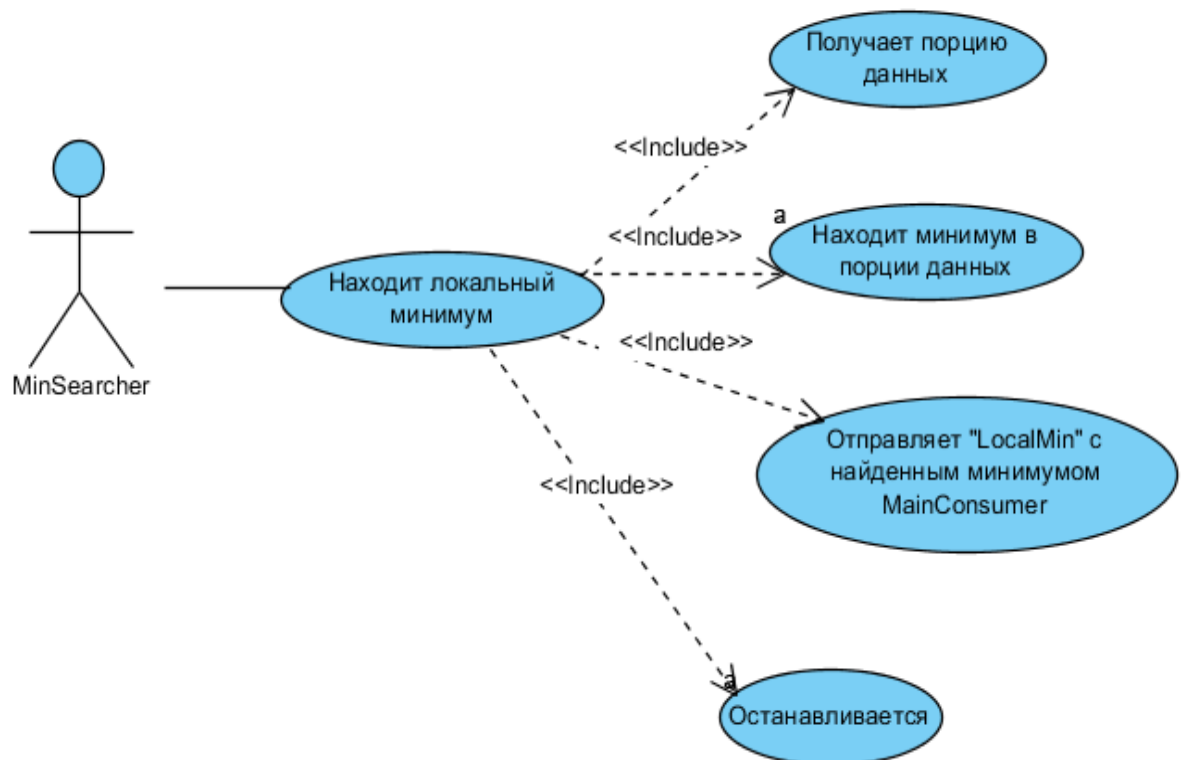


Рисунок 2.27 - Действия актора MinSearcher

Актор MinSearcher при поиске данных выполняет следующие действия:

- Получает порцию данных для обработки.
- Находит минимум в своей порции данных.
- Отправляет сообщение “LocalMin” актору MainConsumer с найденным минимумом.
- Останавливается.

На рисунке 2.28 представлена use-case диаграмма, описывающая действия Erlang процесса, который генерирует данные.

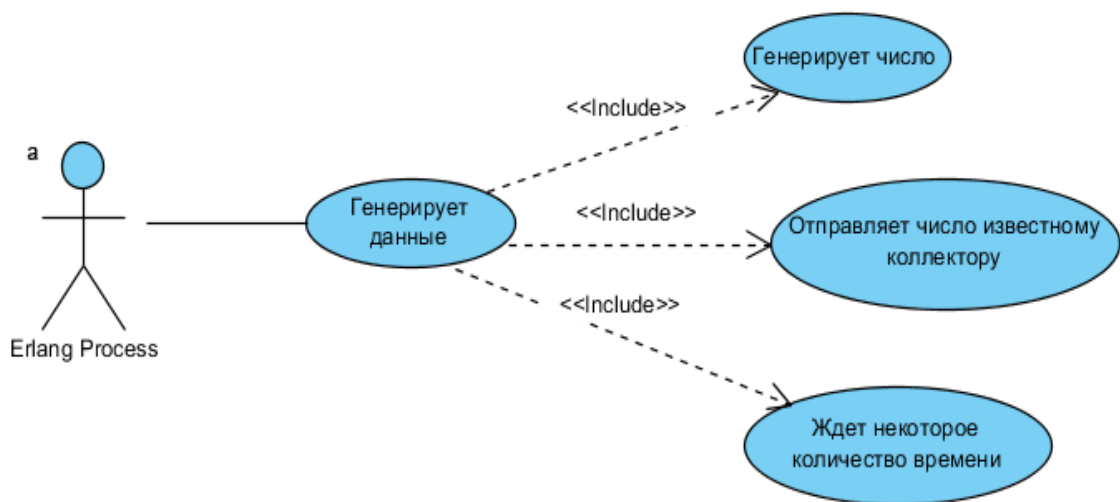


Рисунок 2.28 - Действия Erlang процесса

Erlang процесс во время генерации данных выполняет следующие действия:

- Генерирует случайное число.
- Отправляет число коллектору, который был передан в качестве аргумента.
- Ожидает некоторое количество времени, не превышающее десяти секунд.

2.5.2.5 Диаграммы взаимодействия акторов

На рисунке 2.29 представлена диаграмма общения акторов во время работы системы. На диаграмме указаны сообщения, которые передаются между акторами во время получения данных и поиска минимума.

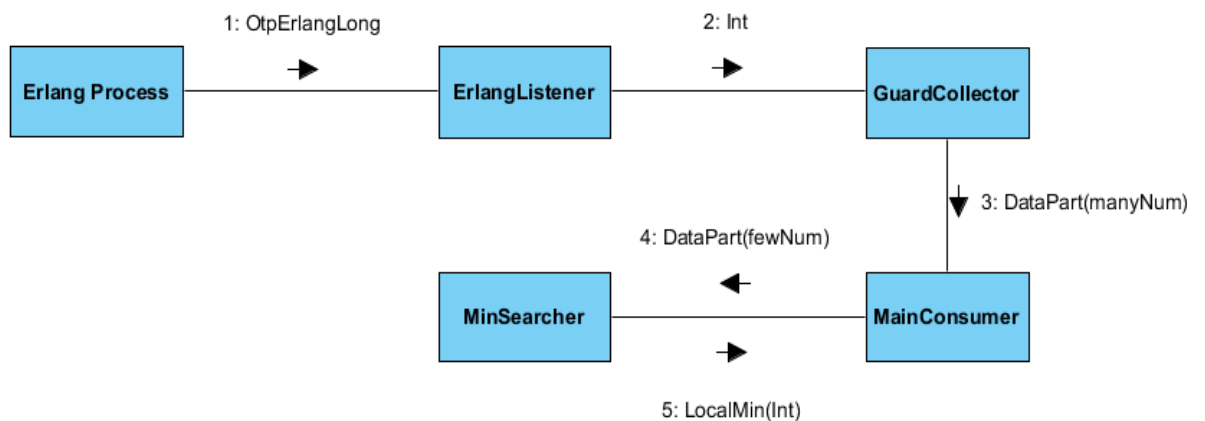


Рисунок 2.29 - Диаграмма общения акторов во время работы системы

3 Разработка фреймворка

3.1 Общие сведения о разработке фреймворка и систем на его основе

Для реализации поставленной задачи был спроектирован абстрактный класс `NotScalaActor`. Спецификация данного класса представлена в разделе 3.2 данной работы.

Для расширения фреймворка и добавления поддержки новой системы необходимо создавать потомков `NotScalaActor`. При создании нового потомка необходимо указывать явно родительский класс тех сообщений, с которыми работает добавляемая система. Также необходимо переопределять все абстрактные методы класса `NotScalaActor` с учетом особенностей добавляемой системы. Так, для организации общения с Erlang узлами, появилась необходимость создать класс `ErlangNode`, который содержит фабричный метод для создания локальных Erlang процессов, обеспечивающих общение с удаленными Erlang узлами.

После создания потомка класса `NotScalaActor` необходимо создать класс актора на базе интерфейса `Actor` из библиотеки Akka. Этот класс должен агрегировать объект созданного класса. При помощи этого объекта будет осуществляться общение с добавленной системой.

Организация общения Scala актора с другой системой строится в переопределенном методе интерфейса `Actor` - `receive()`. Данный метод имеет возвращаемый тип `Receive`, что в свою очередь всего лишь псевдоним для типа `PartialFunction[Any, Unit]`. Другими словами, метод `receive()` возвращает функцию, которая принимает параметр типа `Any`, то есть любой параметр, и возвращает результат типа `Unit`, то есть ничего не возвращает. При этом тело такой функции - это набор выборочных (`case`) утверждений[28]. Набор выборочных утверждений является основой такого механизма, как сопоставление с образцом (`pattern matching`)[29]. Для каждого принимаемого актором сообщения строится свое выборочное утверждение.

Рассмотрим подробнее механизм сопоставления с образцом (`pattern matching`) в языке Scala. Scala имеет встроенный механизм сопоставления с образцом. Этот

механизм позволяет сопоставлять любые данные по принципу первого соответствия. То есть если сопоставляемому параметру соответствует несколько выборочных утверждений, то для обработки этого параметра будет выбрано первое по порядку утверждение. Рассмотрим небольшой пример:

```
object MatchTest extends App {  
  def matchTest(x: Any): String = x match {  
    case 3 => "three"  
    case y: Int => "Int"  
    case SomeCaseClass(num) => num.toString  
    case z: SomeClass => "some not case class"  
    case _ => "other"  
  }  
  println(matchTest(3))  
}
```

В данном примере блок с выборочными(case) утверждениями определяет функцию, которая отображает некоторый параметр в String. Как видно из примера, сопоставляться могут любые типы. В итоге работы этой программы на консоль будет выведена "three", хотя передаваемому параметру соответствует несколько образцов: "case 3", "case y: Int" и "case _". Однако, по принципу первого соответствия будет выбрано "case 3". Выборочное утверждение вида "case _" соответствует любому сопоставляемому параметру.

Для создания класса, потомки которого будут акторами, способными общаться с несколькими системами, рекомендуется сделать следующее:

- Создать сообщение, которое будет инициализировать получение сообщения актором от внедряемой системы;
- Создать кейс-класс, который будет являться оберткой над сообщениями внедряемой системы. Как например, `case class ErlangMessage(msg: OtpErlangObject);`
- Создать абстрактные функции для обработки сообщений от Scala системы, то есть `scalaBehavior(msg: Any)`, и обработки сообщений от внедряемой системы, как например, `erlangBehavior(msg: OtpErlangObject);`

После создания всех необходимых инструментов необходимо организовать следующую обработку сообщений создаваемым абстрактным классом в методе `receive()`:

- Создать выборочное (case) утверждение, которое бы обрабатывало сообщение об инициализации получения сообщения от внедряемой системы;
- После получения сообщения от внедряемой системы, это сообщение обернуть в соответствующий кейс-класс и отправить самому себе;
- Создать выборочное (case) утверждение, которое бы обрабатывало созданный кейс-класс. При этом, сообщение внутри этого кейс-класса передавать в качестве параметра функции, которая будет его обрабатывать. Например, сообщение `msg` из кейс-класса `ErlangMessage` передавать, как параметр функции `erlangBehavior`. Аналогично и для `ScalaMessage`.

Создание такого абстрактного класса позволит создавать потомков, которые будут являться акторами, при этом способными общаться с несколькими системами. Необходимо лишь будет переопределять функции, обрабатывающие входящие сообщения от разных систем.

Для отправки сообщений акторам из внедряемой системой следует использовать одну из реализаций метода `send` абстрактного класса `NotScalaActor`. Для отправки сообщений Scala акторам следует использовать специальный оператор `“!”`. Пример использования этого оператора: `Actor ! message`, где `Actor` - это `ActorRef` (ссылка на актор) получателя, а `message` - сообщение любого типа. Для получения сообщений используется метод `receive()`. Обработку сообщений необходимо строить на основе сопоставления с образцом (pattern matching), как уже рассматривалось выше.

3.2 Спецификации на фреймворк

`abstract class NotScalaActor[TypeMessage] extends AnyRef` - Абстрактный класс для акторов, не являющихся потомками `akka.actor.Actor`.

TypeMessage - суперкласс типов сообщений, с которыми работает актор.

Конструктор

NotScalaActor(name: String)

name - имя актора.

Типовые поля

abstract type IdentifierOtherActor - Псевдоним для типов идентификаторов акторов

Абстрактные поля

abstract def idThisActor: IdentifierOtherActor

returns идентификатор актора

abstract def receive: Option[TypeMessage] - Метод получения сообщения

returns - scala.Some - если сообщение получено, scala.None - иначе

abstract def send[T <: TypeMessage](aname: String, node: String, msg: T):

Boolean - Метод отправки сообщений другим акторам

T - тип сообщения. Подкласс типа TypeMessage

aname - имя актора получателя

node - узел, на котором живет актор получатель

msg - сообщение

returns - true - если актор получатель доступен и сообщение отправлено, false - иначе

abstract def send[T <: TypeMessage](adr: IdentifierOtherActor)(msg: T): Unit

- Метод отправки сообщений другим акторам

T - тип сообщения. Подкласс типа TypeMessage

adr - идентификатор актора получателя

msg - сообщение

Поля

val name: String - имя актора

case class ErlangProcess(name: String) extends

NotScalaActor[OtpErlangObject] - Представление Erlang процессов в фреймворке

Конструктор

ErlangProcess(name: String)

name - имя актора

Типовые поля

type IdentifierOtherActor - OtpErlangPid В качестве идентификатора используется com.ericsson.otp.erlang.OtpErlangPid

Поля

val adapteeProcess: OtpMbox - Внутренний Erlang процесс

def idThisActor: IdentifierOtherActor

returns - идентификатор актора

val name: String - имя актора

def receive: Option[OtpErlangObject] - Получение сообщения

returns - scala.Some - если сообщение получено, scala.None - иначе

def send[T <: OtpErlangObject](aname: String, node: String, msg: T): Boolean

- Отправка сообщения другому Erlang актору

T - тип сообщения. Подкласс типа OtpErlangObject

aname - имя актора получателя

node - узел, на котором живет актор получатель

msg - сообщение

returns - true - если актор получатель доступен и сообщение отправлено, false - иначе

def send[T <: OtpErlangObject](adr: IdentifierOtherActor)(msg: T): Unit -

Отправка сообщения другому Erlang актору

T - тип сообщения. Подкласс типа OtpErlangObject

adr - идентификатор актора получателя

msg - сообщение

object ErlangNode - Локальный Erlang узел для общения с удаленными Erlang узлами

Поля

def apply(name: String): OtpMbox - Фабричный метод для создания OtpMbox(Erlang процесс). Используется ErlangProcess

name - имя создаваемого Erlang процесса

returns - объект класса OtpMbox

val localErlangNode: OtpNode - Внутренний Erlang узел

def setCookie(coo: String): String - Устанавливает Cookie для локального узла, для безопасного общения с удаленными узлами

coo - задаваемый Cookie

returns - установленный Cookie

abstract class ErlangAdapteeActor extends Actor - Абстрактный класс, потомки которого одновременно и Erlang процессы и Scala акторы.

Принимаемые сообщения

ErlangMessage - сообщение для обработки методом erlangBehavior();

ScalaMessage - сообщение для обработки методом scalaBehavior();

Конструктор

ErlangAdapteeActor(name: String)

name - имя актора

Абстрактные поля

abstract def erlangBehavior[U <: OtpErlangObject](msg: U): Unit - Поведение данного актора, как Erlang процесса

U - тип сообщения

msg - обрабатываемое сообщение

abstract def scalaBehavior[U](msg: U): Unit - Поведение данного актора, как Scala актора

U - тип сообщения

msg - обрабатываемое сообщение

case class ErlangMessage(msg: OtpErlangObject) - Сообщение для ErlangAdapteeActor.

msg - сообщение для обработки ErlangAdapteeActor.erlangBehavior()

case class ScalaMessage(msg: Any) - Сообщение для ErlangAdapteeActor.

msg - сообщение для обработки ErlangAdapteeActor.scalaBehavior()

3.3 Релизация решенных задач

3.3.1 Задача “Файловая система на процессах”

Релизация этой задачи имеет две части: Erlang и Scala. Ниже будет подробно рассмотрена реализация Erlang части, а также будут представлены спецификации к Scala коду.

3.3.1.1 Erlang реализация

Для решения задачи используется регистрация имен основных процессов: aggregationProcess(MapAllFiles) и controlProcess(MapAllFiles).

Функция рекурсивного обхода файлов[13]:

```
recurserM(Dir) ->
  {ok, Filenames}=file:list_dir(Dir),
  processOneM(Dir, Filenames).

processOneM(Parent, [First|Next]) ->
  FullName=Parent++"/"++First,
  IsDir=filelib:is_dir(FullName),
  if
    IsDir==true -> recurserM(FullName);
```

```

        IsDir==false -> io:format(FullName+"~n"),
        Master = whereis(superfilesystem),      %%superfilesystem - имя процесса
aggregationProcess
        Master!{FullName,erlang:spawn(fun() ->processBody(FullName) end)}
end,
        processOneM(Parent,Next);
processOneM(_Parent,[])->ok.

```

При обходе списка файлов в каталоге каждый элемент списка проверяется. Если элемент списка каталог, то для него запускается функция `recursorM(Dir)`, если же элемент - файл, то получаю `Pid` агрегирующего процесса и отправляю ему картеж в виде `{NameFile, PidForThisFile}`, где `PidForThisFile` - `pid` процесса, обрабатывающий файл с именем `NameFile`.

Процесс, обрабатывающий файл, выглядит следующим образом:

```

processBody(FullName) ->
    receive
        {delete,PidWorkerOnMainActor} ->
            Rez = file:delete(FullName),
            if
                Rez == ok ->
                    PidWorkerOnMainActor ! {deleted,node(),FullName},
exit(self(),normal);
                true ->
                    PidWorkerOnMainActor ! {error}
            end;
        {rename, NewName, PidWorkerOnMainActor} ->
            Rez = file:rename(FullName,NewName),
            if
                Rez == ok ->
                    PidWorkerOnMainActor ! {renamed,node(),FullName,NewName},
processBody(NewName);
                true ->
                    PidWorkerOnMainActor ! {error}
            end;
        {copy, NewPlace, PidWorkerOnMainActor} ->
            {Rez,_} = file:copy(FullName,NewPlace),
            if
                Rez == ok ->
                    PidWorkerOnMainActor !
{copied,node(),FullName,NewPlace,erlang:spawn(fun() ->processBody(NewPlace) end)};
            end;
    end

```

```

        true ->
            PidWorkerOnMainActor ! {error}
        end
    end,
    processBody(FullName) .

```

Данный пример реализует переименование, удаление и копирование файла.

Агрегирующий процесс - это процесс, который ожидает обхода всех файлов в запущенном каталоге и строит структуру данных `maps #{String=>Pid}`, где ключом будет являться имя файла, а значение `Pid` функции обрабатывающий этот файл.

Агрегирующий процесс:

```

aggregationProcess(MapAllFiles)->
    receive
        {Key,Value} ->
            AllFiles = maps:put(Key,Value,MapAllFiles),
            aggregationProcess(AllFiles);
        ok->
            ControlPr = spawn(fun() -> controlProcess(MapAllFiles) end),
            register(getFS,ControlPr),
            exit(self(),normal)
    end,
    aggregationProcess(MapAllFiles) .

```

Карта процессов строится, не нарушая принципов функционального программирования, так как каждый раз в качестве параметра функции передается новая карта либо старая, старая на новую явно в теле функции не изменяется. По завершению обхода, агрегирующий процесс создает процесс, регистрирующий эту файловую систему и передает ему созданную карту процессов. Сам же агрегирующий процесс нормально останавливается.

Регистрирующий процесс:

```

controlProcess(MapAllFiles)->
    receive
        {getFS,PidSender} ->
            io:format("FS connected ~n"),
            PidSender ! {node(),MapAllFiles},
            exit(self(),normal)
    end,

```

```
controlProcess(MapAllFiles).
```

Регистрирующий процесс располагается на узле по имени “getFS”. Поле запуска программы, этот процесс будет создан и начнет ожидать сообщения с запросом карты процессов. После передачи карты регистрирующий процесс нормально останавливается.

Функция, запускающая эту файловую систему:

```
startFS() ->
  AgrProcPid = spawn(fun() -> aggregationProcess(maps:new()) end),
  register(superfilesystem, AgrProcPid),
  AgrProcPid ! recursorM(".").
```

3.3.1.2 Спецификации на Scala реализацию

class ClientForGUI extends Actor - Клиентский актор. Служит прослойкой между интерфейсом и системой акторов. Данный актор имеет два состояния: инициализация системы и работы с системой

Принимаемые сообщения

Инициализация системы

scala.Tuple2("init", Set[String]) - сообщение для начала инициализации системы.

InitDone - сообщение об успешной инициализации системы

InitError - сообщение об неполадках во время инициализации

ReceiveTimeout - сообщение о завершении времени ожидания

Работа системы

FileDeleted - сообщение об удалении файла

FileRenamed - сообщение об переименовании файла

FileCopied - сообщение об копировании файла

CommandToFS - сообщение с командами для файловой системы

“exe” - сообщение о запуске команд к файловой системе

Отправляемые сообщения

Инициализация системы

InitCommand - сообщение для начала инициализации системы.

List[Tuple2[String,List[String]]] - сообщение со списком кортежей, у которых первый элемент имя подключенного узла, а второй - список файлов на узле.

“Error init” - сообщение для клиента об ошибке во время инициализации

“Receive Timeout in init” - сообщение о превышении ожидания инициализации

Работа системы

CommandToFS - сообщение с командами для файловой системы

String - сообщение о завершенной операции

Tuple2(String,List[String]) - сообщение с обновленной картой узла

Актор общается с: MainFS, интерфейсом пользователя

class MainFS extends Actor - Главный актор в файловой системе. Данный актор инициализирует систему и обрабатывает запросы к ней. Данный актор имеет два состояния: инициализация системы и работы с системой

Принимаемые сообщения

Инициализация системы

InitCommand - сообщение для начала инициализации системы.

ConnectedNode - сообщение о подключенном к системе узле

NodeNotConnect - сообщение о неподключенном к системе узле

Terminated - сообщение об остановке актора, за которым следит данный актор

ReceiveTimeout - сообщение о завершении времени ожидания

Работа системы

FileDeleted - сообщение об удалении файла

FileRenamed - сообщение о переименовании файла

FileCopied - сообщение о копировании файла

CommandToFS - сообщение с командами для файловой системы

Отправляемые сообщения

Инициализация системы

ScalaMessage - сообщение для гибридных акторов

InitDone - сообщение об успешной инициализации системы

InitError - сообщение об ошибке во время инициализации

Работа системы

ScalaMessage - сообщение для гибридных акторов

Tuple3(String, String, Map[String, _]) - сообщение с информацией о завершенной команде

Актор общается с: ClientForGUI, InitFileSystemProcess, FileSystemProcess.

class InitFileSystemProcess extends ErlangAdapteeActor - гибридный актор, инициализатор файловой системы

Принимаемые сообщения

Scala сообщения

Set[String] - узлы для инициализации

Erlang сообщения

OtpErlangTuple(node, mapProc) - ответ от инициализированного Erlang узла

Отправляемые сообщения

Scala сообщения

ConnectedNode - сообщение о подключенном к системе узле

NodeNotConnect - сообщение о неподключенном к системе узле

Erlang сообщения

OtpErlangTuple("getFS", self) - сообщение для получения карты процессов от удаленного узла

Актор общается с: MainFS, управляющими процессами Erlang.

class FileSystemProcess extends ErlangAdapteeActor - гибридный актор - обработчик команды файловой системы

Принимаемые сообщения

Scala сообщения

pidFile - идентификатор процесса, обрабатывающего файл

scala.Tuple2(pidFile: OtpErlangPid, "delete") - команда на удаление файла

scala.Tuple3(pidFile: OtpErlangPid, newName:String, "rename") - команда на переименование файла

scala.Tuple3(pidFile: OtpErlangPid, copyName:String, "copy") - команда на копирование файла

Erlang сообщения

OtpErlangTuple("deleted", node, file) - ответ об удалении файла

OtpErlangTuple("renamed", node, oldFileName, newFileName) - ответ о переименовании файла

OtpErlangTuple("copied", node, oldFileName, newFileName, pidNewFile) - ответ о копировании файла

Отправляемые сообщения

Scala сообщения

FileDeleted - сообщение об удалении файла

FileRenamed - сообщение о переименовании файла

FileCopied - сообщение о копировании файла

Erlang сообщения

OtpErlangTuple("delete", self) - команда об удалении для файла

OtpErlangTuple("rename", newName, self) - команда о переименовании файла

OtpErlangTuple("copy", copyName, self) - команда о копировании файла

Актор общается с: MainFS, обслуживающими файлы Erlang процессами

abstract class CommandToFile extends AnyRef - Абстрактный класс для команд к файлам

case class CopyFile(file: String, newPlace: String) extends CommandToFile - Команда для копирования файла

file - имя файла

newPlace - имя копии файла file

case class DeleteFile(file: String) extends CommandToFile - Команда для удаления файла

file - имя удаляемого файла

case class RenameFile(file: String, newFileName: String) extends CommandToFile - Команда для переименования файла

file - имя файла

newFileName - новое имя файла

case class CommandToNode(node: String, command: CommandToFile) - Сообщение с командой для конкретного узла

node - имя узла, к которому адресована команда

command - команда для файла на узле

case class CommandToFS(commands: Set[CommandToNode]) - Сообщение с командами для файловой системы

commands - команды для файловой системы

case class InitCommand(nodesList: Set[String]) - Сообщение для начала инициализации системы.

nodesList - узлы для инициализации

case class InitDone(nodes: Set[ConnectedNode]) - Сообщение об успешной инициализации системы

nodes - успешно подключенные узлы

object InitError - Сообщение об неполадках во время инициализации

case class NodeNotConnect(nodeName: String) - Сообщение о неподключенном узле

nodeName - имя узла

case class ConnectedNode(nodeName: String, mapProc: Map[String, OtpErlangPid]) - Сообщение об успешно подключенном узле системы

nodeName - имя узла

mapProc - карта процессов подключенного узла

case class FileCopied(node: String, originalFile: String, copyFile: String, pidCopy: OtpErlangPid) - Сообщение о копировании файла

node - имя узла, на котором был скопирован файл

originalFile - имя оригинала файла

copyFile - имя скопированного файла

pidCopy - идентификатор процесса, обслуживающего новый файл

case class FileDeleted(node: String, file: String) - Сообщение об удалении файла

node - имя узла, на котором был удален файл

file - имя удаленного файла

case class FileRenamed(node: String, oldFile: String, newFile: String) - Сообщение о переименовании файла

node - имя узла, на котором был переименован файл

oldFile - старое имя файла

newFile - новое имя файла

3.3.2 Задача “Поиск минимального значения в непрерывном потоке данных”

Реализация этой задачи имеет две части: Erlang и Scala. Ниже будет подробно рассмотрена реализация Erlang части, а также будут представлены спецификации к Scala коду.

3.3.2.1 Erlang реализация

В качестве поставщиков данных, были использованы Erlang процессы. Они получают в качестве аргумента имя коллектора, которому они должны слать данные. Эти процессы ожидают некоторое произвольное количество времени, не превышающее 10 секунд, генерируют случайное число, посылают своему коллектору и рекурсивно вызывают сами себя.

Реализация процесса поставщика данных:

```
data_producer(Collector) ->
    timer:sleep(rand:uniform(10000)),
    Ch = rand:uniform(10000),
    {Collector, 'ScalaErlangNode@danil-pc'} ! Ch,
    data_producer(Collector).
```

Инициализирующий процесс принимает в качестве параметров список коллекторов, на которые будут посылаться данные, и число уже запущенных процессов для первого в списке коллектора. Реализация инициализирующего процесса использует сопоставление с образцом (pattern matching) языка Erlang.

Реализация инициализирующего процесса:

```
start([],_) -> ok;
start([_|T],3500) -> start(T,0);
start(Collectors,Count) ->
    [H|_] = Collectors,
    P = spawn(fun() -> data_producer(H)end),
    io:format("~w producer start. Send to ~w ~n",[P,H]),
    start(Collectors,Count+1).
```

Данная реализация запускает по 3500 процессов поставщиков на один коллектор.

3.3.2.2 Спецификации на Scala реализацию

class MainConsumer extends Actor - Главный актор в данной системе. Для получения данных этот актор создает GuardCollector, а для обработки полученных чисел создает MinSearcher В своем состоянии хранит искомые величины.

Принимаемые сообщения

"Start system" - сообщение о запуске системы

DataPart - сообщение с данными для обработки

Terminated - сообщение об остановке одного из GuardCollector

LocalMin - сообщение с минимумом в некоторой порции данных

"Get min" - запрос состояния актора

Отправляемые сообщения

"get" - сообщение о начале получения данных

DataPart - сообщение с данными для обработки

Актор общается с: GuardCollector, MinSearcher, интерфейсом пользователя.

class GuardCollector extends Actor - Актор коллектор, уменьшающий поток данных к главному процессу. Для получения данных создает себе помощника ErlangListener

Принимаемые сообщения

Int - числа для поиска минимума

Отправляемые сообщения

DataPart - сообщение с набором данных для обработки

Актор общается с: ErlangListener, MainConsumer

class ErlangListener extends Actor - Актор, принимающий сообщения от Erlang процессов и отправляющий своему GuardCollector

Принимаемые сообщения

Scala сообщения

"get" - сообщение о начале получении всех сообщений от Erlang

Erlang сообщения

OtpErlangLong - число, сгенерированное Erlang процессом

Отправляемые сообщения

Scala сообщения

Int - полученное число для обработки

Актор общается с: GuardCollector, удаленными Erlang процессами.

case class DataPart(data: Set[Int]) - Сообщение с данными для обработки

data - данные для обработки

case class LocalMin(min: Int) - Сообщение с найденным минимумом

min - найденный минимум

4 Тестирование

4.1 Обзор системы тестирования

Для тестирования акторной системы в Akka предусмотрен специальный модуль, отвечающий за это. Этот модуль называется akka-testkit. Akka-testkit позволяет осуществлять синхронное модульное тестирование[30] и асинхронное интеграционное тестирование[30]. Akka-testkit основан на другом фреймворке для тестирования - ScalaTest. ScalaTest - это основной инструмент использующийся для тестирования Scala программ.

Для синхронного модульного тестирования akka-testkit предоставляет доступ к конкретному актору при помощи специального класса TestActorRef. Получив доступ к актору таким способом, можно протестировать все его методы при помощи стандартного модульного тестирования, например, при помощи фреймворка ScalaTest. Однако, основной механизм взаимодействия с актором - это передача сообщений. Поэтому тестировать акторы следует при помощи асинхронного интеграционного тестирования. Внутренние методы, если они сложные, лучше выносить и тестировать отдельно, прежде чем использовать их в поведении создаваемого актора. Для этого можно использовать REPL[31] или создавать тесты для отдельных функций, вынесенных, например, в некоторый объект.

После того, как логика акторов проверена или она достаточно проста, необходимо проверить поведение акторов в своем предполагаемом окружении. Окружение зависит от многих факторов. Начинать тестирование следует отправкой конкретного сообщения конкретному актору и ожидать конкретный ответ. Затем необходимо переходить к тестированию сетей акторов, используя тот же принцип. Для проверки ответов akka-testkit предоставляет удобные инструменты, наподобие тех, что используются в стандартном модульном тестировании.

4.2 Задача на тестирование

Тестирование в рамках данной работы приведено для демонстрации инструментов интеграционного тестирования акторных систем. Для тестирования

выбрана задача “Файловая система на процессах”, так как эта задача на взаимодействие акторов и она позволяет в полной мере продемонстрировать тестирующую систему.

Целью тестирования была проверка поведения системы во время инициализации и работы с ней.

4.3 Ожидаемое поведение

Таблица 4.1 - Ожидаемое поведение системы

Предусловия	Действия	Ожидаемый результат
Узел не существует в системе	Попытаться подключить узел, используя сообщение InitConnected	Получить в ответ сообщение InitError
Узел существует в системе, но файловой системы на нем нет	Попытаться подключить узел, используя сообщение InitConnected	Получить в ответ сообщение InitError
Узел существует в системе с активированной файловой системой	Попытаться подключить узел, используя сообщение InitConnected	Получить в ответ сообщение InitDone с подключенным узлом
Файл существует на узле	Попытаться удалить файл, используя команду DeleteFile	Получить в ответ сообщение FileDeleted с удаленным файлом
Файл существует на узле	Попытаться переименовать файл, используя команду RenameFile	Получить в ответ сообщение FileRenamed с переименованным файлом
Файл существует на узле	Попытаться скопировать файл, используя команду CopyFile	Получить в ответ сообщение FileCopied со скопированным файлом
Файлы существуют на узле	Попытаться к каждому из файлов параллельно запустить команду	Получить ответы от всех команд за приемлемое время
Файла нет на узле	Попытаться запустить команду к файлу	Долго не получать ответ

Продолжение таблицы 4.1

Предусловия	Действия	Ожидаемый результат
Узел не подключен	Попытаться запустить команду к узлу	Долго не получать ответ

4.4 Общий подход к тестированию

Во время тестирования актору MainFS, который отвечает за работу системы, отправлялись сообщения, которые посылает ему актор ClientForGUI. Затем получались ответы от MainFS и сравнивались с ожидаемыми. Использовался асинхронный интеграционный подход к тестированию.

Код реализованных тестов:

```
import akka.actor.ActorSystem
import akka.actor.Props
import akka.testkit.{ TestKit, ImplicitSender}
import filesystem._
import org.scalatest.{ WordSpecLike, Matchers, BeforeAndAfterAll}
import scala.concurrent.duration._

class TestFS() extends TestKit(ActorSystem("FileSystemOnActors")) with
ImplicitSender

with WordSpecLike with Matchers with BeforeAndAfterAll{

  override def afterAll {
    TestKit.shutdownActorSystem(system)
  }

  //Актор MainFS
  "An MainFS actor" must {
    //должен возвращать ответ InitError, когда узла не существует в системе
    "send back messages InitError object" in {
      val mainFS = system.actorOf(Props[MainFS])
      mainFS ! InitCommand(Set("node@node"))
      expectMsg(8 second, InitError)
    }

    //должен возвращать ответ InitError, когда узел не отвечает
    "send back messages InitError object" in{
      val mainFS = system.actorOf(Props[MainFS])
      mainFS ! InitCommand(Set("m1@danil-pc"))
    }
  }
}
```

```

        expectMsg(10 second, InitError)
    }
}
//когда система инициализирована
"When system is init" must {
    //тестируемый актер MainFS
    val mainFS = system.actorOf(Props[MainFS])

    //при получении сообщения InitCommand должен возвращать сообщение InitDone
    с подключенными узлами

    "send back messages InitDone" in {
        mainFS ! InitCommand(Set("m2@danil-pc"))
        expectMsgPF() {
            case InitDone(set: Set[ConnectedNode]) => set.head.nodeName should
            be("m2@danil-pc")
        }
    }

    //при получении сообщения DeleteFile должен возвращать сообщение
    FileDeleted с удаленным файлом

    "send back message FileDeleted" in {
        val commandToDelete = DeleteFile("./f1.txt")
        val commandToNode = CommandToNode("m2@danil-pc", commandToDelete)
        val commandToFS = CommandToFS(Set(commandToNode))
        mainFS ! commandToFS
        expectMsg(FileDeleted("m2@danil-pc", "./f1.txt"))
    }

    //при получении сообщения CopyFile должен возвращать сообщение FileCopied с
    скопированным файлом

    "send back message FileCopied" in {
        val commandToCopy = CopyFile("./f2.txt", "./copy_f2.txt")
        val commandToNode = CommandToNode("m2@danil-pc", commandToCopy)
        val commandToFS = CommandToFS(Set(commandToNode))
        mainFS ! commandToFS
        expectMsgPF() {
            case FileCopied("m2@danil-pc", "./f2.txt", "./copy_f2.txt", _) => "OK"
        }
    }
}

//при получении сообщения RenameFile должен возвращать сообщение
FileRenamed с переименованным файлом

"send back message FileRenamed" in {

```

```

    val commandToRename = RenameFile("./f3.txt", "./f3_renamed.txt")
    val commandToNode = CommandToNode("m2@danil-pc", commandToRename)
    val commandToFS = CommandToFS(Set(commandToNode))
    mainFS ! commandToFS
    expectMsg(FileRenamed("m2@danil-pc", "./f3.txt", "./f3_renamed.txt"))
}

```

*//при получении сообщения нескольких сообщений должны быть возвращены все
ответы с интервалом не больше 3 секунды*

```

"send back messagw about complite actions" in{
    val commandToDelete = DeleteFile("./m1.txt")
    val commandToCopy = CopyFile("./m2.txt", "./copy_m2.txt")
    val commandToRename = RenameFile("./m3.txt", "./m3_renamed.txt")
    val commandToNodeOne = CommandToNode("m2@danil-pc", commandToRename)
    val commandToNodeTwo = CommandToNode("m2@danil-pc", commandToDelete)
    val commandToNodeThree = CommandToNode("m2@danil-pc", commandToCopy)
    val
                                commandToFS
                                =
CommandToFS(Set(commandToNodeOne, commandToNodeTwo, commandToNodeThree))
    mainFS ! commandToFS
    var count = 0;
    receiveWhile(10 second, 3 second, 3) {
        case m3 @ FileRenamed("m2@danil-pc", "./m3.txt", "./m3_renamed.txt") =>
count += 100
        case m1 @ FileDeleted("m2@danil-pc", "./m1.txt") => count += 1
        case m2 @ FileCopied("m2@danil-pc", "./m2.txt", "./copy_m2.txt", _) =>
count += 10
        case _=>
    }
    count should be(111)
}

```

*//При попытке удалить несуществующий файл, система не отвичает как минимум
минуту*

```

"not send answer from file" in {
    val commandToDelete = DeleteFile("./aaa.txt")
    val commandToNode = CommandToNode("m2@danil-pc", commandToDelete)
    val commandToFS = CommandToFS(Set(commandToNode))
    mainFS ! commandToFS
    expectNoMsg(1 minute)
}

```

//При попытке удалить файл с неподключенного узла, система не отвечает как минимум минуту

```
"not send answer from node" in {  
  val commandToDelete = DeleteFile("./f1.txt")  
  val commandToNode = CommandToNode("m3@danil-pc", commandToDelete)  
  val commandToFS = CommandToFS(Set(commandToNode))  
  mainFS ! commandToFS  
  expectNoMsg(1 minute)  
}  
}  
}
```

4.5 Результаты тестирования

Тело теста на подключение отсутствующего узла:

//должен возвращать ответ InitError, когда узла не существует в системе

```
"send back messages InitError object" in {  
  val mainFS = system.actorOf(Props[MainFS])  
  mainFS ! InitCommand(Set("node@node"))  
  expectMsg(8 second, InitError)  
}
```

Результат тестирования представлен на рисунке 4.1.

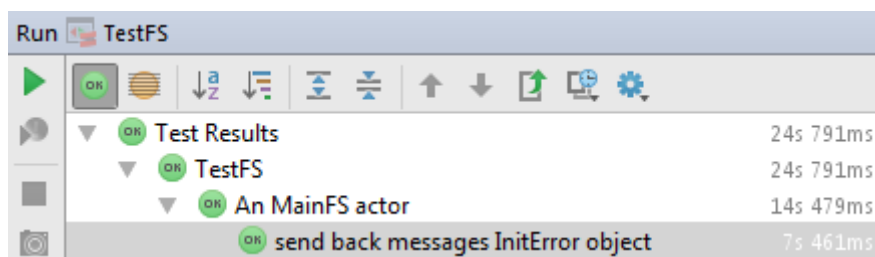


Рисунок 4.1 - Результат тестирования на подключение отсутствующего узла

Тело теста на подключение узла, без инициализированной файловой системы на нем:

//должен возвращать ответ InitError, когда узел не отвечает

```
"send back messages InitError object" in {  
  val mainFS = system.actorOf(Props[MainFS])  
  mainFS ! InitCommand(Set("m1@danil-pc"))  
  expectMsg(10 second, InitError)  
}
```

Результат тестирования представлен на рисунке 4.2.

Test Results	Duration
Test Results	24s 791ms
TestFS	24s 791ms
An MainFS actor	14s 479ms
send back messages InitError object	7s 461ms
send back messages InitError object	7s 18ms

Рисунок 4.2 - Результат тестирования на подключение узла, без инициализированной файловой системы на нем

Тело теста на подключение узла с инициализированной на нем файловой системой:

```
//при получении сообщения InitCommand должен возвращать сообщение InitDone с
подключенными узлами

"send back messages InitDone" in {
  mainFS ! InitCommand(Set("m2@danil-pc"))
  expectMsgPF() {
    case InitDone(set: Set[ConnectedNode]) => set.head.nodeName should
be ("m2@danil-pc")
  }
}
```

Результат тестирования представлен на рисунке 4.3.

Test Results	Duration	Log
Test Results	24s 791ms	
TestFS	24s 791ms	
An MainFS actor	14s 479ms	
send back messages InitError object	7s 461ms	
send back messages InitError object	7s 18ms	
When system is init	10s 312ms	
send back messages InitDone	117ms	m2@danil-pc is connected Initialization complete!

Рисунок 4.3 - Результат тестирования на подключение узла с инициализированной файловой системой

Тело теста на удаление существующего файла:

```
//при получении сообщения DeleteFile должен возвращать сообщение
FileDeleted с удаленным файлом

"send back message FileDeleted" in {
  val commandToDelete = DeleteFile("./f1.txt")
```

```

val commandToNode = CommandToNode("m2@danil-pc", commandToDelete)
val commandToFS = CommandToFS(Set(commandToNode))
mainFS ! commandToFS
expectMsg(FileDeleted("m2@danil-pc", "./f1.txt"))
}

```

Результат тестирования представлен на рисунке 4.4.

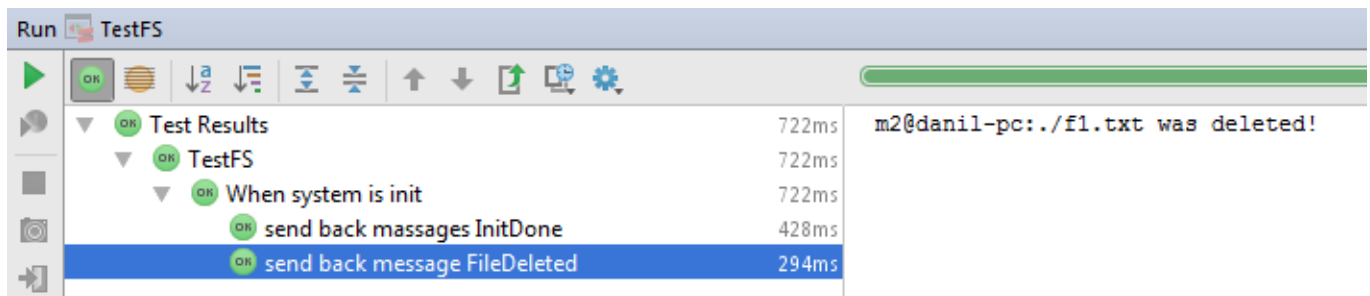


Рисунок 4.4 - Результат тестирования на удаление существующего файла

Тело теста на копирование существующего файла:

//при получении сообщения CopyFile должен возвращать сообщение FileCopied с скопированным файлом

```

"send back message FileCopied" in {
  val commandToCopy = CopyFile("./f2.txt", "./copy_f2.txt")
  val commandToNode = CommandToNode("m2@danil-pc", commandToCopy)
  val commandToFS = CommandToFS(Set(commandToNode))
  mainFS ! commandToFS
  expectMsgPF() {
    case FileCopied("m2@danil-pc", "./f2.txt", "./copy_f2.txt", _) => "OK"
  }
}

```

Результат тестирования представлен на рисунке 4.5.

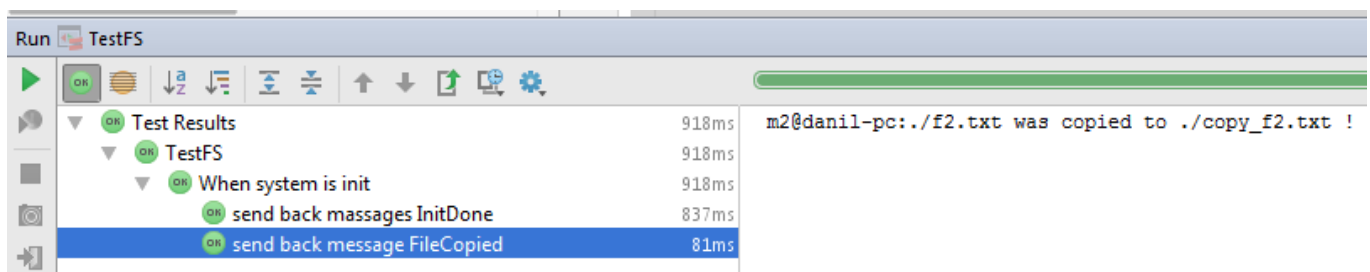


Рисунок 4.5 - Результат тестирования на копирование существующего файла

Тело теста на переименование существующего файла:

//при получении сообщения RenameFile должен возвращать сообщение FileRenamed с переименованным файлом


```

"send back message FileRenamed" in {
    val commandToRename = RenameFile("./f3.txt", "./f3_renamed.txt")
    val commandToNode = CommandToNode("m2@danil-pc", commandToRename)
    val commandToFS = CommandToFS(Set(commandToNode))
    mainFS ! commandToFS
    expectMsg(FileRenamed("m2@danil-pc", "./f3.txt", "./f3_renamed.txt"))
}

```

Результат тестирования представлен на рисунке 4.6.

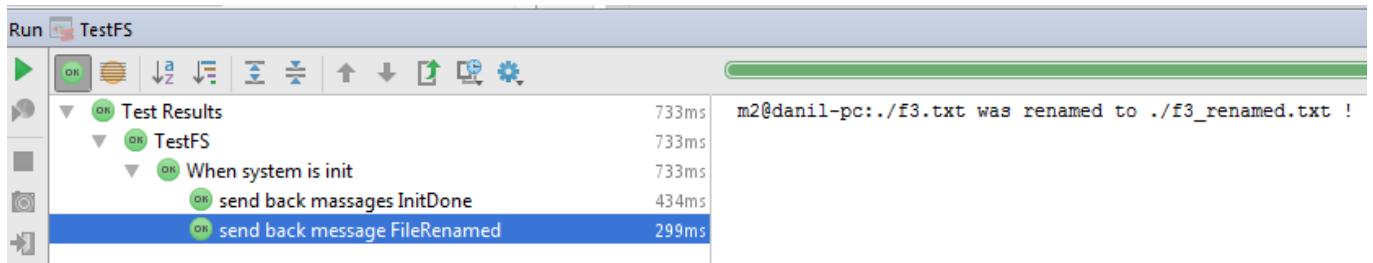


Рисунок 4.6 - Результат тестирования на переименование существующего файла

Тело теста на применение нескольких команд к существующим файлам:

//при получении сообщения нескольких сообщений должны быть возвращены все ответы с интервалом не больше 3 секунды

```

"send back messagw about complite actions" in{
    val commandToDelete = DeleteFile("./m1.txt")
    val commandToCopy = CopyFile("./m2.txt", "./copy_m2.txt")
    val commandToRename = RenameFile("./m3.txt", "./m3_renamed.txt")
    val commandToNodeOne = CommandToNode("m2@danil-pc", commandToRename)
    val commandToNodeTwo = CommandToNode("m2@danil-pc", commandToDelete)
    val commandToNodeThree = CommandToNode("m2@danil-pc", commandToCopy)
    val
        commandToFS
        =
    CommandToFS(Set(commandToNodeOne, commandToNodeTwo, commandToNodeThree))
    mainFS ! commandToFS
    var count = 0;
    receiveWhile(10 second, 3 second, 3) {
        case m3 @ FileRenamed("m2@danil-pc", "./m3.txt", "./m3_renamed.txt") => count
+= 100
        case m1 @ FileDeleted("m2@danil-pc", "./m1.txt") => count += 1
        case m2 @ FileCopied("m2@danil-pc", "./m2.txt", "./copy_m2.txt", _) => count
+= 10
        case _ =>
    }
    count should be(111)
}

```

```
}
```

Результат тестирования представлен на рисунке 4.7.

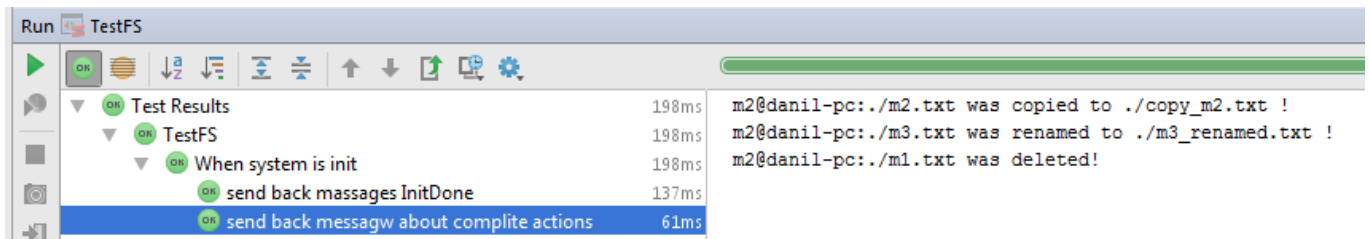


Рисунок 4.7 - Результат тестирования на применение нескольких команд к существующим файлам

Тело теста на применение команды к не существующему файлу:

//При попытке удалить несуществующий файл, система не отвечает как минимум
минуту

```
"not send answer from file" in {  
  val commandToDelete = DeleteFile("./aaa.txt")  
  val commandToNode = CommandToNode("m2@danil-pc", commandToDelete)  
  val commandToFS = CommandToFS(Set(commandToNode))  
  mainFS ! commandToFS  
  expectNoMsg(1 minute)  
}
```

Результат тестирования представлен на рисунке 4.8.

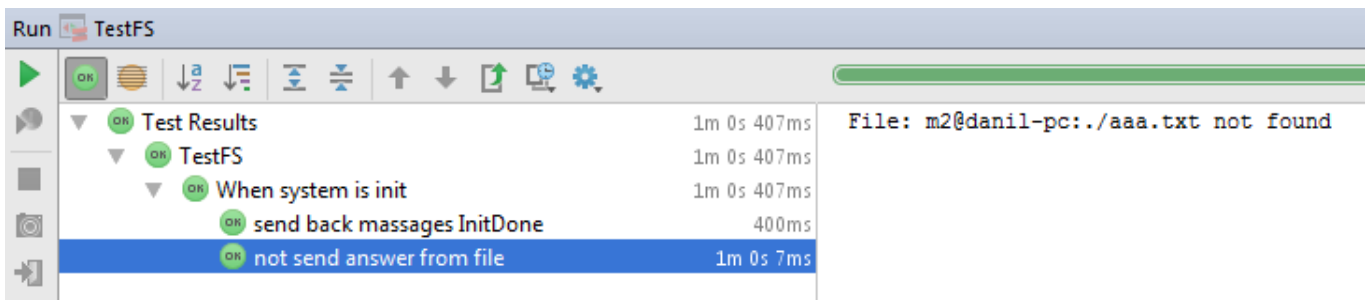


Рисунок 4.8 - Результат тестирования на применение команды к не существующему файлу

Тело теста на применение команды к не подключенному узлу:

//При попытке удалить файл с неподключенного узла, система не отвечает как
минимум минуту

```
"not send answer from node" in {  
  val commandToDelete = DeleteFile("./f1.txt")  
  val commandToNode = CommandToNode("m3@danil-pc", commandToDelete)  
  val commandToFS = CommandToFS(Set(commandToNode))
```

```

mainFS ! commandToFS
expectNoMsg(1 minute)
}

```

Результат тестирования представлен на рисунке 4.9.

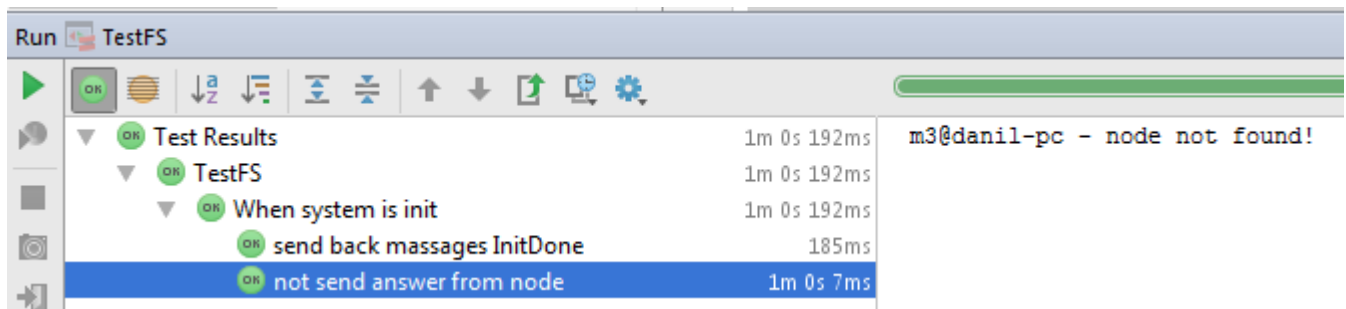


Рисунок 4.9 - Результат тестирования на применение команды к не подключенному узлу

4.6 Итоги тестирования

В результате проведенного тестирования, были написаны асинхронные интеграционные тесты, показывающие поведение тестируемой системы при инициализации и работы с ней. Так же тесты показывают скорость системы основанной на акторах. Из проведенных тестов видно, что система работает достаточно быстро.

OK send back messages InitDone	374ms
OK send back message FileDeleted	57ms
OK send back message FileCopied	11ms
OK send back message FileRenamed	8ms
OK send back messagw about complite actions	32ms

Рисунок 4.10 – Общие результаты тестирования рабочей системы

Заключение

Таким образом, поставленные задачи были решены. Спроектирован и реализован фреймворк для распределения задач с использованием сетей функциональных вычислений. На основе реализованного фреймворка были решены задачи, для демонстрации основных возможностей фреймверка. Так же была протестирована одна из написанных систем с использованием интеграционных тестов. Данная работа была представлена на научной конференции «Наука и Молодежь – 2016» [1].

Список использованных источников

1. Кристалеv Д.А., Старолетов С.М. Фреймворк для распределения задач с использованием сетей функциональных вычислений // Сборник трудов 13-й Всеросс. научн.-техн. конфер. студентов, аспирантов и молод. Ученых «Наука и Молодежь – 2016». Секция «Информационные технологии». Подсекция «Программная инженерия», 22 апреля 2016, Барнаул: АлтГТУ, 2016, - 4с.
2. Параллельные вычисления и многопоточное программирование [текст] / В.А. Биллиг - М.: НОУ "Интуит", 2016 .- С. 134 - 179.
3. The Scala Programming Language [Электронный ресурс] / Режим доступа - <http://www.scala-lang.org/>
4. Erlang Programming Language [Электронный ресурс] / Режим доступа - <http://www.erlang.org/>
5. Транзакционная память - первые шаги / Леонид Черняк // «Открытые системы» , № 04, 2007 [Электронный ресурс] / Режим доступа - <http://www.osp.ru/os/2007/04/4219769/>
6. Dataflow Programming - Concept, Languages and Applications / Tiago Boldt Sousa. [Электронный ресурс] / Режим доступа - http://paginas.fe.up.pt/~prodei/dsie12/papers/paper_17.pdf
7. Actors: A Model of Concurrent Computation in Distributed Systems / Gul Agha // MIT Press, 1985. [Электронный ресурс] / Режим доступа - <https://www.cypherpunks.to/erights/history/actors/AITR-844.pdf>
8. Agent-oriented programming / Yoav Shoham .- Technical Report STAN-CS-90-1335.- Computer Science Department, Stanford University, 1990. [Электронный ресурс] / Режим доступа - <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.123.5119&rep=rep1&type=pdf>
9. Erlang / OTP Documentation [Электронный ресурс] / Режим доступа - <http://erlang.org/erldoc>

10. Comparison of Erlang Runtime System and Java Virtual Machine / Tõnis Pool
[Электронный ресурс] / Режим доступа - <http://ds.cs.ut.ee/courses/course-files/To303nis%20Pool%20.pdf>
11. The Go Programming Language [Электронный ресурс] / Режим доступа - <https://golang.org/>
12. Akka Documentation [Электронный ресурс] / Режим доступа - <http://akka.io/docs/>
13. Старолетов С. М. Функциональные языки распределённых систем: Учебно-методическое пособие. / С. М. Старолетов— Барнаул: АлтГТУ, 2015. – 81с.
14. Lisp (programming language) [Электронный ресурс] / Режим доступа - [https://en.wikipedia.org/wiki/Lisp_\(programming_language\)](https://en.wikipedia.org/wiki/Lisp_(programming_language))
15. The Clojure Programming Language [Электронный ресурс] / Режим доступа <http://clojure.org/>
16. The Kotlin Programming Language [Электронный ресурс] / Режим доступа <https://kotlinlang.org/>
17. Monads for functional programming / Philip Wadler, 1992. [Электронный ресурс] / Режим доступа <http://homepages.inf.ed.ac.uk/wadler/papers/marktoberdorf/baastad.pdf>
18. Apache Kafka a high-throughput distributed messaging system [Электронный ресурс] / Режим доступа <http://kafka.apache.org/>
19. The Jinterface Package [Электронный ресурс] / Режим доступа http://erlang.org/doc/apps/jinterface/jinterface_users_guide.html
20. Why functional programming matters / John Hughes .- Research Topics in Functional Programming, 1990 .- 17–42 с. [Электронный ресурс] / Режим доступа - <https://www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf>
21. Kotlin in Action [текст]/ Dmitry Jemerov, Svetlana Isakova .- Manning Publications, 2016. - 325 с.

22. Масштабируемость [Электронный ресурс] / Режим доступа <https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%81%D1%88%D1%82%D0%B0%D0%B1%D0%B8%D1%80%D1%83%D0%B5%D0%BC%D0%BE%D1%81%D1%82%D1%8C>
23. Case Classes [Электронный ресурс] / Режим доступа <http://docs.scala-lang.org/tutorials/tour/case-classes.html>
24. Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. Приемы объектно-ориентированного проектирования. Паттерны проектирования. — СПб: «Питер», 2007. — С. 366.
25. Singleton Objects [Электронный ресурс] / Режим доступа <http://docs.scala-lang.org/tutorials/tour/singleton-objects.html>
26. Polymorphic Type Inference / Michael I. Schwartzbach [Электронный ресурс] / Режим доступа <http://cs.au.dk/~mis/typeinf.pdf>
27. Package com.ericsson.otp.erlang Description [Электронный ресурс] / Режим доступа <http://erlang.org/doc/apps/jinterface/java/com/ericsson/otp/erlang/package-summary.html>
28. Понимание PartialFunction [Электронный ресурс] / Режим доступа https://twitter.github.io/scala_school/ru/pattern-matching-and-functional-composition.html
29. Pattern Matching [Электронный ресурс] / Режим доступа <http://docs.scala-lang.org/tutorials/tour/pattern-matching.html>
30. Виды тестирования и подходы к их применению [Электронный ресурс] / Режим доступа <https://habrahabr.ru/post/81226/>
31. REPL [Электронный ресурс] / Режим доступа <https://ru.wikipedia.org/wiki/REPL>

Приложение А

Задание на бакалаврскую работу

Министерство образования и науки Российской Федерации
федеральное государственное бюджетное образовательное
учреждение высшего образования
“АЛТАЙСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ им.
И.И.ПОЛЗУНОВА”

УТВЕРЖДАЮ

Зав. кафедрой С. А. Кантор

" " _____ 2016 г.

ЗАДАНИЕ № 14 НА БАКАЛАВРСКУЮ РАБОТУ

По направлению подготовки _____ Программная инженерия
По профилю _____ Разработка программно-информационных систем
студенту группы _____ ПИ-21

Кристалеvu Данилу Андреевичу

фамилия, имя, отчество

Тема: **Фреймворк для распределения задач с использованием сетей
функциональных вычислений**

Утверждено приказом ректора А-893 от 29.03.2016

Срок исполнения работы 28.06.2016

Задание принял к исполнению Кристалеv Д.А.

ЧК
подпись

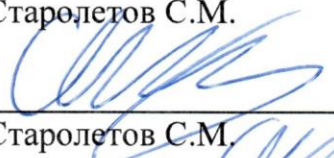
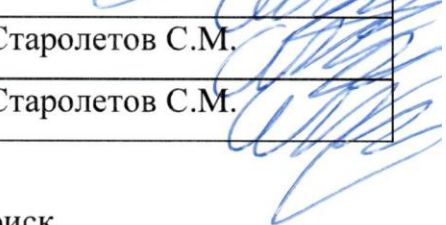
фамилия, имя, отчество

БАРНАУЛ 2016

1 Исходные данные

Задание на выполнение, техническая документация на библиотеки Akka и JInterface, документация на языки программирования Scala и Erlang, ресурсы internet, ГОСТы

2 Содержание разделов проекта

Наименование и содержание разделов проекта	Трудоемкость, % от всего объема проекта	Срок выполнения	Консультант (Ф.И.О., подпись)
1 Изучение предметной области	20	17.05.2016	Старолетов С.М. 
2 Проектирование ПО	30	24.05.2016	Старолетов С.М. 
3 Реализация ПО	30	05.06.2016	Старолетов С.М.
4 Написание отчета	20	28.06.2016	Старолетов С.М.

3 Научно-библиографический поиск

3.1 По научно-технической литературе просмотреть РЖ «Кибернетика», «Программное обеспечение»

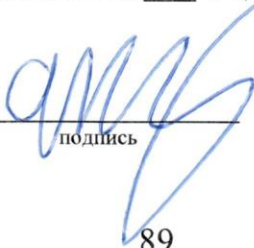
за последние 10 лет и научно-технические журналы «Программирование», «Программная инженерия», «Информационные технологии»

_____ за последние 10 лет.

3.2 По нормативной литературе просмотреть указатели государственных и отраслевых стандартов за последний год.

3.3 Патентный поиск провести за 4 года по странам Россия

Руководитель проекта


подпись

С.М. Старолетов
и., о., фамилия

Приложение Б

Руководство пользователя

Для того, что бы воспользоваться фреймворком, необходимо добавить пакет “myframework” в созданный проект. Пакет находится на диске, прилагаемом к этой работе. Так же в создаваемом проекте необходимо добавить зависимость к библиотеке Akka и JInterface. Рекомендации по использованию фреймворка представлены в разделе 3.1 данной работы.

Для запуска примера «Файловая система на процессах» необходимо запустить Erlang систему и Scala систему. Для запуска Erlang системы необходимо в корень каталога, в котором лежат файлы для файловой системы поместить файл filesystem.beam. После этого необходимо запустить узел Erlang командой `werl` из командной строки с текущим каталогом, в котором лежит файл `filesystem.beam`. Затем необходимо инициализировать файловую систему на узле командой `filesystem:startFS()`, как показано на рисунке Б.1.

```
(m1@danil-pc)1> filesystem:startFS().  
./erl_crash.dump  
./filesystem.beam  
./pg.beam  
./renamecopy2.txt  
ok  
(m1@danil-pc)2> █
```

Рисунок Б.1 – Пример запуска файловой системы на узле

После подготовки всех Erlang узлов к работе, необходимо запустить Scala систему для управления файловой системой. Для этого необходимо запустить файл `filesystem.jar`. Главная форма приложения представлена на рисунке ниже.

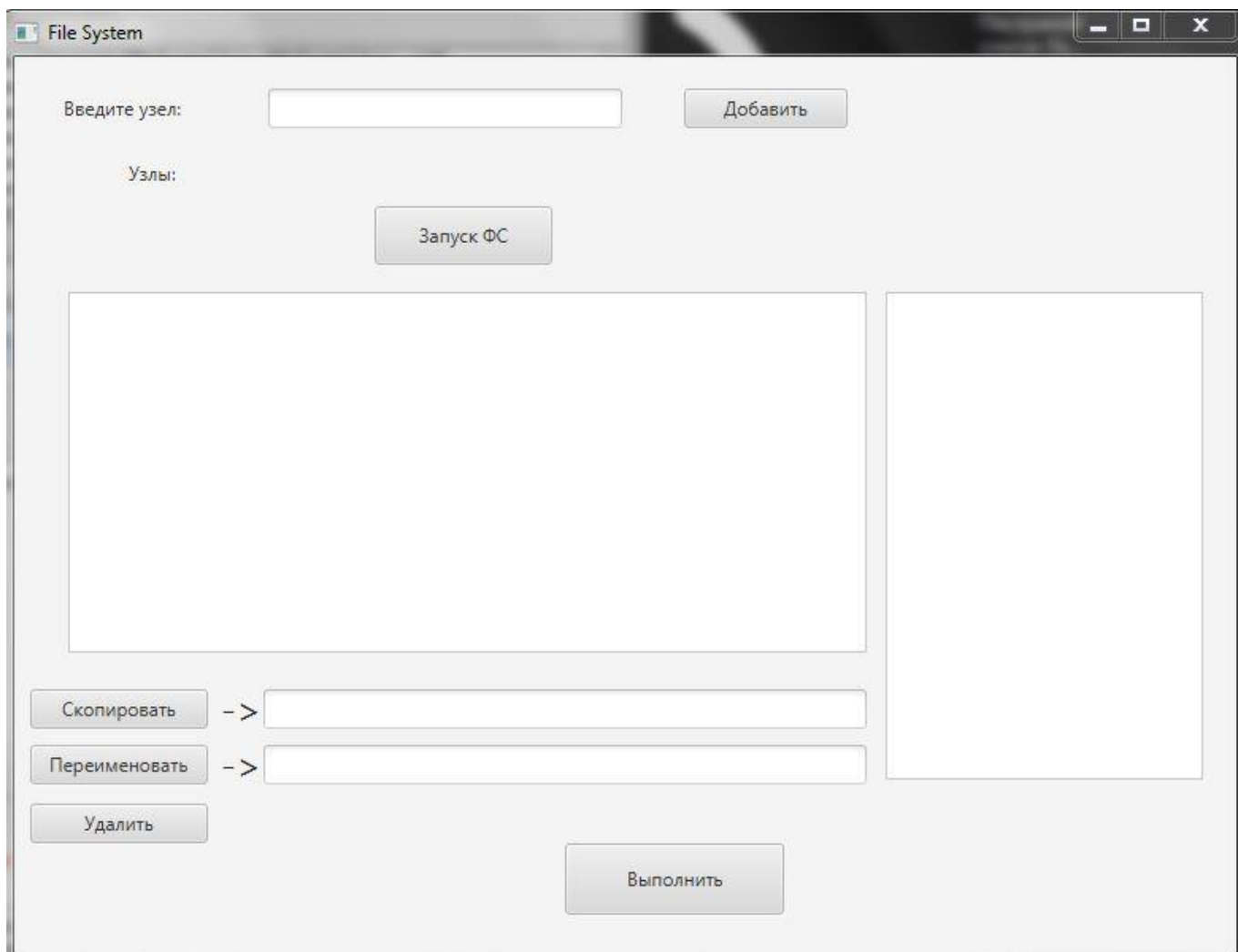


Рисунок Б.2 – Главная форма управляющего приложения

После запуска приложения, необходимо ввести имена Erlang узлов, на которых запущена файловая система, как на рисунке Б.3.

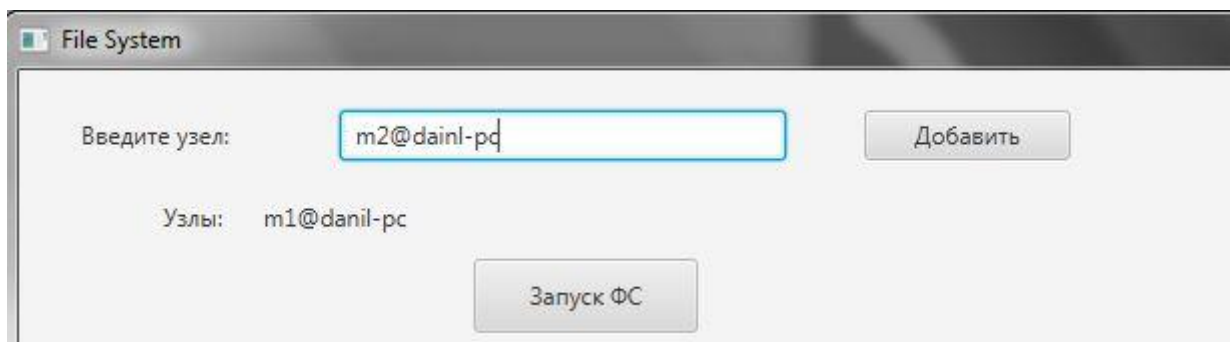


Рисунок Б.3 – Ввод узлов файловой системы

После ввода всех узлов необходимо нажать кнопку «Запуск ФС». Система начнет инициализацию. После завершения инициализации все подключенные узлы будут отображены на форме, как показано на рисунке Б.4.

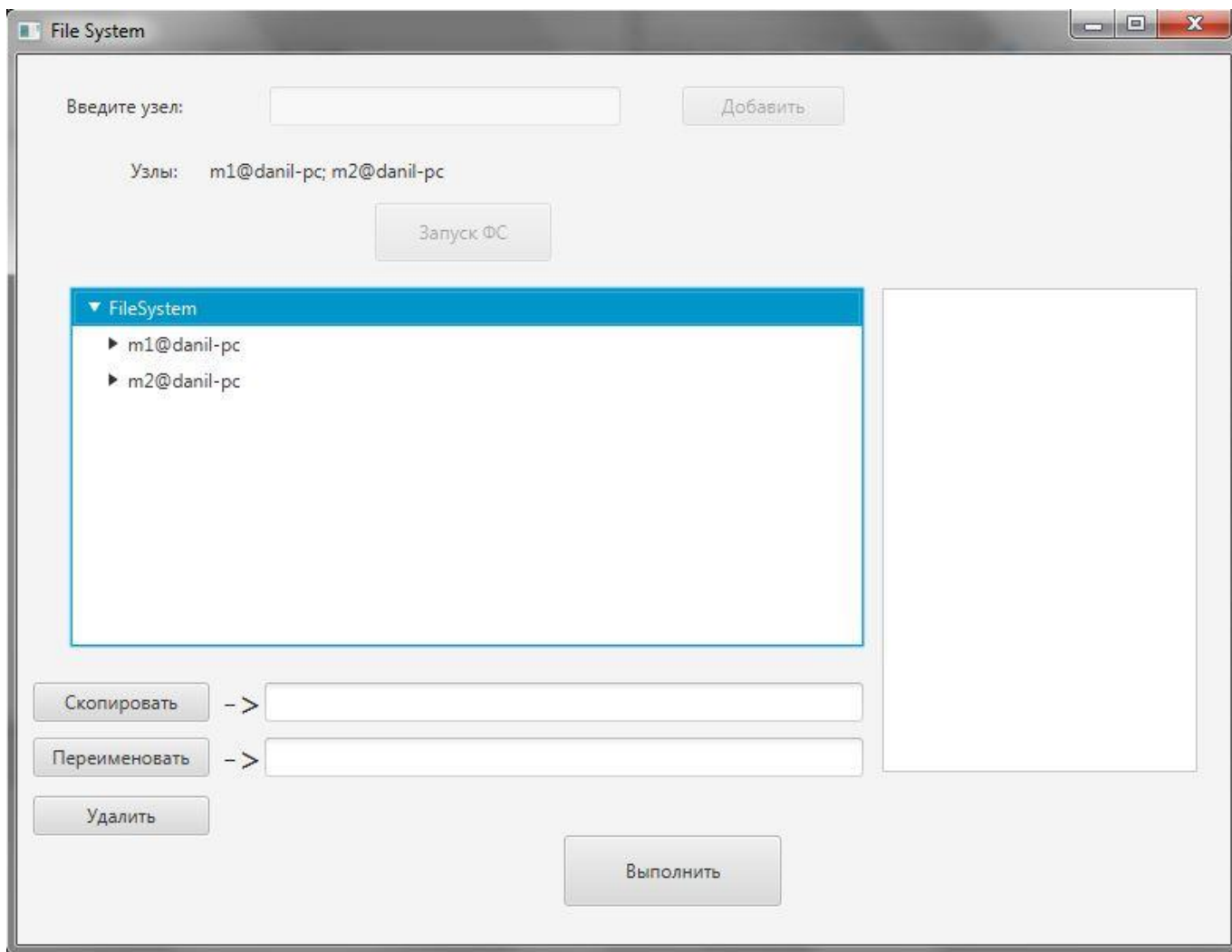


Рисунок Б.4 – Отображение подключенных узлов

После подключения Erlang узла в его консоли появляется надпись «FS connected», как на рисунке Б.5.

```
(m1@danil-pc)1> filesystem:startFS().  
./erl_crash.dump  
./filesystem.beam  
./pg.beam  
./renamecopy2.txt  
ok  
FS connected  
(m1@danil-pc)2>
```

Рисунок Б.5 – Подключенный Erlang узел

После инициализации системы и подключения узлов, можно начать работать с файлами системы. Для этого необходимо вводить команды к файлам. Чтоб ввести команду к файлу, необходимо выбрать узел, затем выбрать файл на узле, после чего снизу формы выбрать одну из команд. Ввод команды на копирования файла представлен на рисунке Б.6.

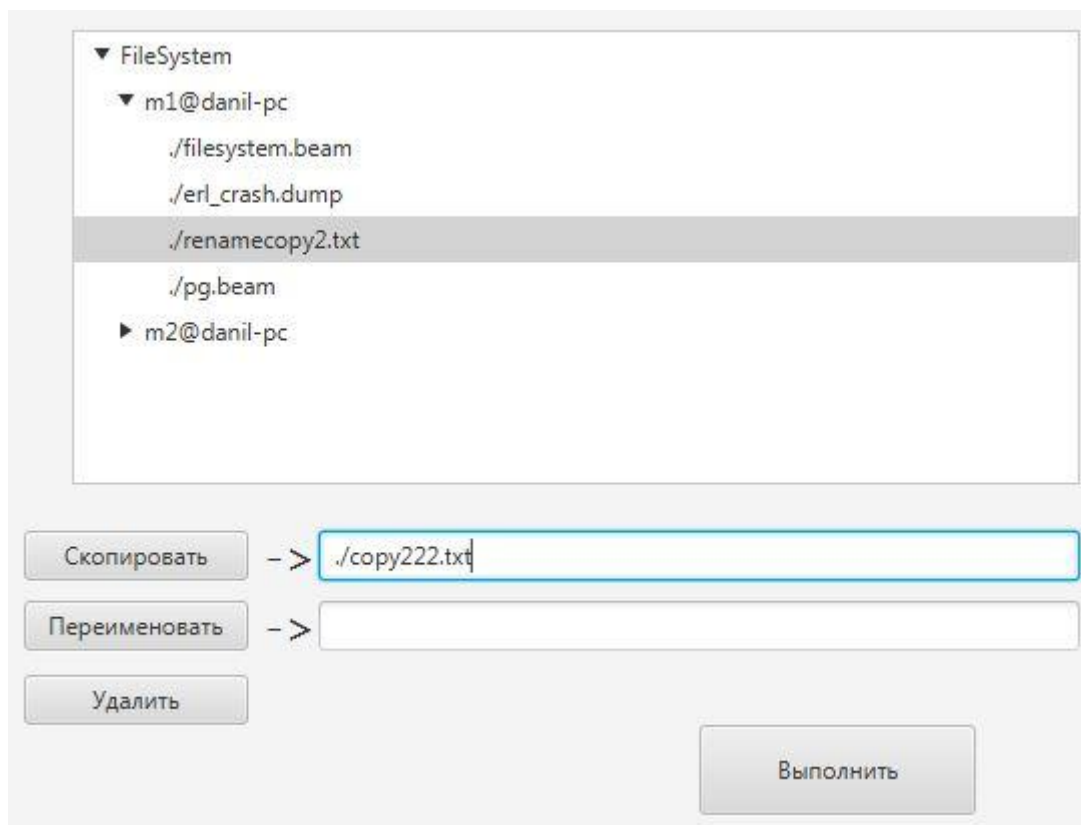


Рисунок Б.6 – Ввод команды на копирование файла

После ввода имени копии файла необходимо нажать кнопку «Скопировать» и команда будет добавлена в буфер на выполнение. Для переименования файла необходимо сделать аналогичные действия. Для удаления файла достаточно выбрать файл на узле и нажать кнопку «Удалить».

После ввода всех команд, нужно нажать кнопку «Выполнить» для выполнения всех введенных команд. Пример выполнения команды копирования представлен на рисунке Б.7.

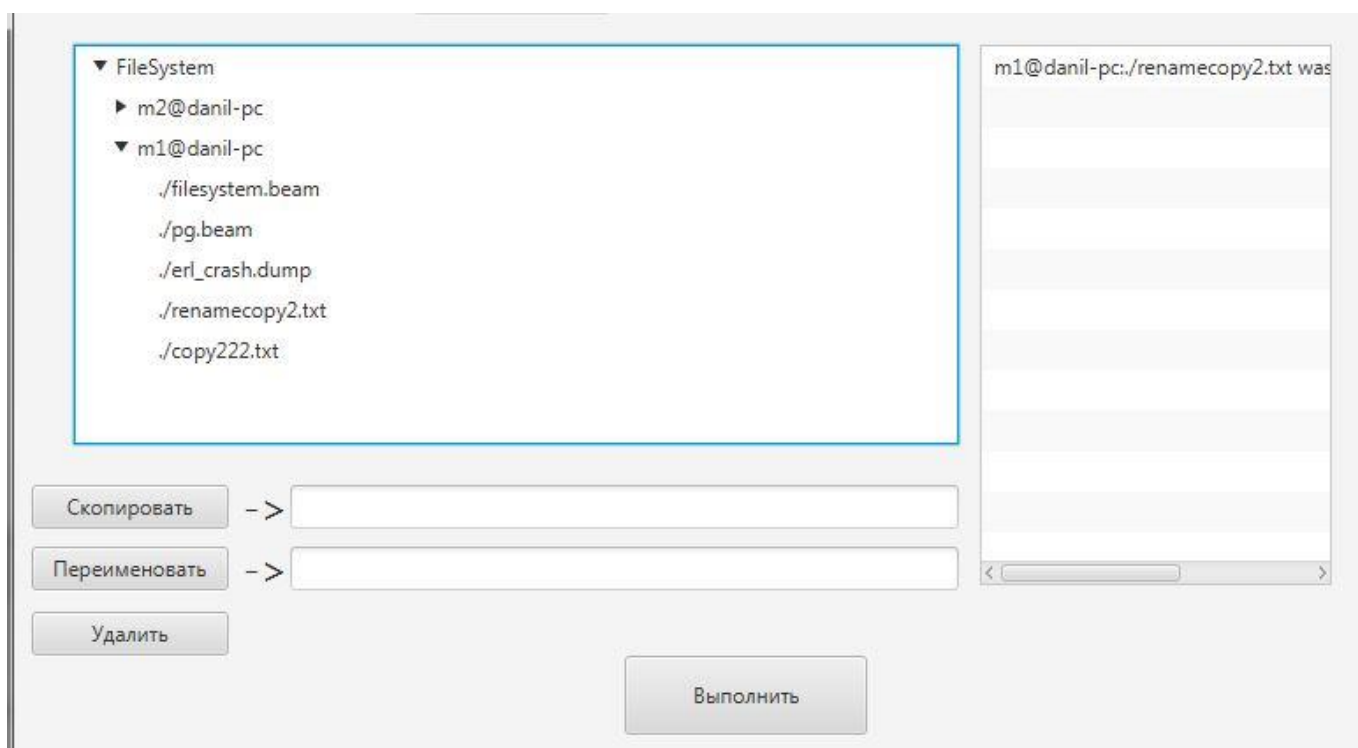


Рисунок Б.7 – Выполнение команды копирования

Для запуска примера «Поиск минимального значения в непрерывном потоке данных» необходимо запустить для начала Scala систему. Для этого необходимо запустить ScalaConsumer.jar, и на появившейся форме нажать кнопку «Start». Пример запущенной Scala системы на рисунке Б.8.

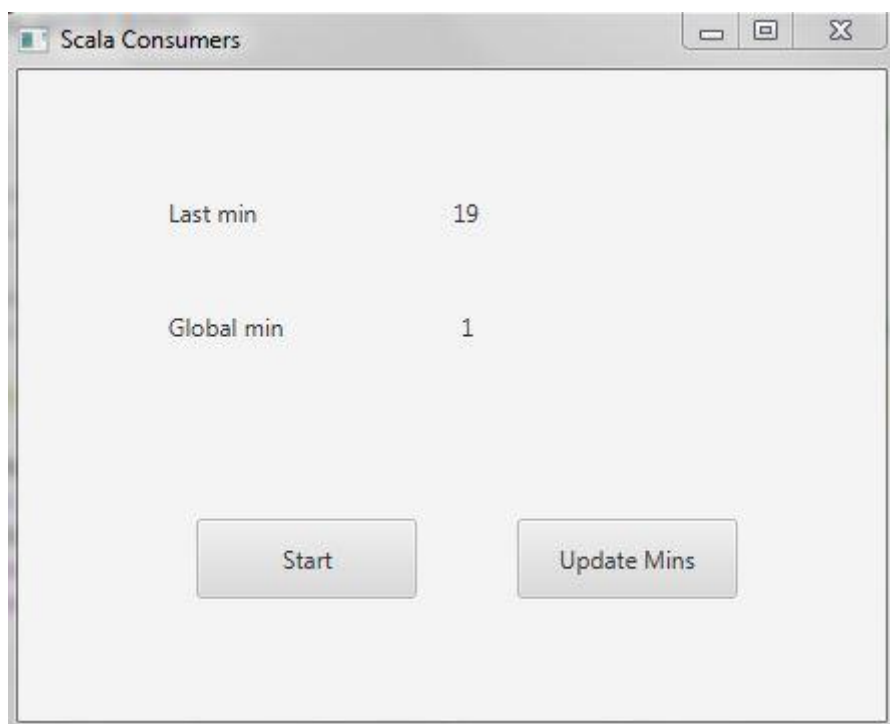


Рисунок Б.8- Работающая Scala система

После запуска Scala системы необходимо запустить Erlang узел с процессами, производящими данные. Для этого необходимо запустить Erlang узел командой `werl` из каталога в котором лежит файл `data_prod.beam`. После запуска узла необходимо запустить процессы, генерирующие данные. Команда для запуска процессов генерирующих данные представлена на рисунке Б.9.

```
(m1@danil-pc)1> data_prod:start(['collector1','collector2','collector3','collector4','collector5'],0).
```

Рисунок Б.9 – Команда для запуска процессов

Приложение В

Исходный код

«Фреймворк»

ActorBridge.scala

```
package myframework

/**
 * Абстрактный класс для акторов, не являющихся потомками [[akka.actor.Actor]]
 * @see
 * [[http://doc.akka.io/api/akka/2.4.7/?_ga=1.40424290.1913219468.1458323745#akka.a
 * ctor.Actor]]
 *
 * @param name имя актора
 * @tparam TypeMessage суперкласс типов сообщений, с которыми работает актор
 */
abstract class NotScalaActor[TypeMessage](val name:String){

  /** Псевдоним для типов идентификаторов акторов*/
  type IdentifierOtherActor

  /**
   * Метод отправки сообщений другим акторам
   *
   * @param adr идентифекатор актора получателя
   * @param msg сообщение
   * @tparam T тип сообщения. Подкласс типа [[TypeMessage]]
   */
  def send[T <: TypeMessage](adr: IdentifierOtherActor)(msg: T) : Unit

  /**
   * Метод отправки сообщений другим акторам
   *
   * @param aname имя актора получателя
   * @param node узел, на котором живет актор получатель
   * @param msg сообщение
   * @tparam T тип сообщения. Подкласс типа [[TypeMessage]]
   * @return true - если актор получатель доступен и сообщение отправлено,
   false - иначе
   */
  def send[T <: TypeMessage](aname:String,node:String, msg: T) : Boolean

  /**
   * Метод получения сообщения
   *
   * @return [[scala.Some]] - если сообщение получено, [[scala.None]] - иначе
   */
  def receive : Option[TypeMessage]

  /**
   *
   * @return идентификатор актора
   */
  def idThisActor : IdentifierOtherActor}
```


ScalaErlangBridge.scala

```
package myframework
import com.ericsson.otp.erlang.{OtpMbox, OtpErlangPid, OtpErlangObject, OtpNode}

/**
 * Представление Erlang процессов в фреймворке
 *
 * @param name имя актора
 */
case class ErlangProcess(override val name:String)
  extends NotScalaActor[OtpErlangObject](name){
  /** В качестве идентификатора используется
  [[com.ericsson.otp.erlang.OtpErlangPid]]
  * @see
  [[http://erlang.org/doc/apps/jinterface/java/com/ericsson/otp/erlang/OtpErlangPid.html]]
  */
  override type IdentifierOtherActor = OtpErlangPid

  /**
   * Внутренний Erlang процесс
   */
  val adapteeProcess:OtpMbox = ErlangNode(name)

  /** Отправка сообщения другому Erlang актору
   *
   * @param adr идентификатор актора получателя
   * @param msg сообщение
   * @tparam T тип сообщения. Подкласс типа [[OtpErlangObject]]
   */
  override def send[T <: OtpErlangObject](adr:IdentifierOtherActor)(msg: T):
Unit = adapteeProcess.send(adr,msg)

  /** Отправка сообщения другому Erlang актору
   *
   * @param aname имя актора получателя
   * @param node узел, на котором живет актор получатель
   * @param msg сообщение
   * @tparam T тип сообщения. Подкласс типа [[OtpErlangObject]]
   * @return true - если актор получатель доступен и сообщение отправлено,
false - иначе
   */
  override def send[T <: OtpErlangObject](aname:String,node:String,msg: T) :
Boolean =
    if(adapteeProcess.ping(node,1000)) {
      adapteeProcess.send(aname,node,msg)
      true
    } else
      false

  /**Получение сообщения
   *
   * @return [[scala.Some]] - если сообщение получено, [[scala.None]] - иначе
   */
  override def receive: Option[OtpErlangObject] = adapteeProcess.receive(1)
  match {
    case msg:OtpErlangObject =>
      Some(msg)
    case _=>
      None
  }
}

/**
```

```

    *
    * @return идентификатор актора
    */
    def idThisActor : IdentifierOtherActor = adapteeProcess.self()
}

/**
 * Локальный Erlang узел для общения с удаленными Erlang узлами
 * @see
 * [[http://erlang.org/doc/apps/jinterface/java/com/ericsson/otp/erlang/OtpNode.htm
 * l]]
 */
object ErlangNode{
    /**
     * Внутренний Erlang узел
     */
    val localErlangNode = new OtpNode("ScalaErlangNode")

    /**
     * Фабричный метод для создания [[OtpMbox]] (Erlang процесс). Используется
     * [[ErlangProcess]]
     * @see
     * [[http://erlang.org/doc/apps/jinterface/java/com/ericsson/otp/erlang/OtpMbox.htm
     * l]]
     *
     * @param name имя создаваемого Erlang процесса
     * @return объект класса [[OtpMbox]]
     */
    def apply(name:String) = localErlangNode.createMbox(name)

    /**
     * Устанавливает Cookie для локального узла, для безопасного общения с
     * удаленными узлами
     * @param coo задаваемый Cookie
     * @return установленный Cookie
     */
    def setCookie(coo:String) = localErlangNode.setCookie(coo)
}

```

ErlangAdapteeActor.scala

```

package myframework
import akka.actor.Actor
import com.ericsson.otp.erlang._

/**Абстрактный класс, потомки которого одновременно и Erlang процессы и Scala
акторы.

*==Принимаемые сообщения==
* - [[ErlangMessage]] - сообщение для обработки методом [[erlangBehavior()]];<br>
* - [[ScalaMessage]] - сообщение для обработки методом [[scalaBehavior()]];
*
* @see
* [[http://doc.akka.io/api/akka/2.4.7/?_ga=1.40424290.1913219468.1458323745#akka.actor
* .Actor]]
* @param name имя актора

```

```

*/
abstract class ErlangAdapteeActor(val name:String) extends Actor{
  case object ContinueGetMessage

  /**Внутренний Erlang процесс для общения с другими Erlang процессами*/
  protected val innerErlProc = ErlangProcess(name)

  /**
   *
   * @usecase получение сообщений, для обработки
   */
  override def receive: Receive = {
    case ContinueGetMessage =>
      val o = innerErlProc.receive
      val mess = o.getOrElse(new OtpErlangAtom("nothing"))
      mess match {
        case at:OtpErlangAtom if at.toString == "nothing" =>
          self ! ContinueGetMessage
        case _=>
          self ! ErlangMessage(mess)
      }
    case ErlangMessage(msg) =>
      erlangBehavior(msg)
    case ScalaMessage(msg) =>
      scalaBehavior(msg)
  }

  /**
   * Поведение данного актора, как Erlang процесса
   * @param msg обрабатываемое сообщение
   * @tparam U тип сообщения
   */
  def erlangBehavior[U<:OtpErlangObject](msg:U) : Unit

  /**Поведение данного актора, как Scala актора
   *
   * @param msg обрабатываемое сообщение
   * @tparam U тип сообщения
   */
  def scalaBehavior[U](msg:U) : Unit

  protected def russianChars(fileName:OtpErlangObject):String = fileName match {

```

```

    case list:OtpErlangList =>
        list.elements().map(_ asInstanceOf[OtpErlangLong]).map(_ charValue()).mkString
    case str:OtpErlangString =>
        str.toString.tail.dropRight(1)
  }
}

```

TypeMessage.scala

```
package myframework
```

```
import com.ericsson.otp.erlang.OtpErlangObject
```

```

/**
 * Сообщение для [[ErlangAdapteeActor]].
 * @param msg сообщение для обработки [[ErlangAdapteeActor.erlangBehavior()]]
 */
case class ErlangMessage(msg:OtpErlangObject)

/**
 * Сообщение для [[ErlangAdapteeActor]].
 * @param msg сообщение для обработки [[ErlangAdapteeActor.scalaBehavior()]]
 */
case class ScalaMessage(msg:Any)

```

«Файловая система на процессах»

ClientForGUL.scala

```
package filesystem
```

```
import akka.actor.{Actor, ActorRef, Props, ReceiveTimeout}
```

```
import scala.concurrent.duration._
```

```

/**
 * Клиентский актор. Служит прослойкой между интерфейсом и системой акторов
 *
 * Данный актор имеет два состояния: инициализация системы и работы с системой
 *
 * ==Принимаемые сообщения==
 */

```

```

* ===Инициализация системы===
* - scala.Tuple2("init", Set[String]) - сообщение для начала инициализации
системы.<br>
* - [[InitDone]] - сообщение об успешной инициализации системы<br>
* - [[InitError]] - сообщение об неполадках во время инициализации <br>
* - [[ReceiveTimeout]] - сообщение о завершении времени ожидания<br>
*
* ===Работа системы ===
* - [[FileDeleted]] - сообщение об удалении файла<br>
* - [[FileRenamed]] - сообщение об переименовании файла<br>
* - [[FileCopied]] - сообщение об копировании файла<br>
* - [[CommandToFS]] - сообщение с командами для файловой системы<br>
*/

class ClientForGUI extends Actor{
  val mainFS = context.actorOf(Props[MainFS],"mainFS")
  context.watch(mainFS)
  var lastSender:ActorRef = null
  override def receive: Receive ={
    case ("init",setNodes:scala.collection.mutable.Set[String]) =>
      context.setReceiveTimeout(10 second)
      mainFS ! InitCommand(setNodes.toSet)
      lastSender = sender()
    case InitDone(setNode)=>
      context.setReceiveTimeout(Duration.Undefined)
      val listFileOnNode = setNode
        .map(_._1.nodeName)
        .zip(setNode
          .map(_._2.mapProc)
          .map(_._3.keys))
        .toList
      lastSender ! listFileOnNode
      context.become(workerWithFS)
    case InitError =>
      lastSender ! "Error init"
    case ReceiveTimeout =>
      lastSender ! "Receive Timeout in init"
  }

  private var commands: Set[CommandToNode] = Set.empty
  def workerWithFS:Receive = {
    case ("delete", node:String, file:String) =>

```

```

    val delFile = DeleteFile(file)
    val comandToNode = CommandToNode(node, delFile)
    commands += comandToNode
case ("rename",node:String,oldFileName:String,newFileName:String) =>
    val renameFile = RenameFile(oldFileName, newFileName)
    val comandToNode = CommandToNode(node, renameFile)
    commands += comandToNode
case ("copy",node:String,oldFilePlace:String,newFilePlace:String) =>
    val copyFile = CopyFile(oldFilePlace, newFilePlace)
    val comandToNode = CommandToNode(node, copyFile)
    commands += comandToNode
case "exe" =>
    lastSender = sender()
    mainFS ! CommandToFS(commands)
    commands = Set.empty
case (str:String, node:String, map:Map[String,_])=>
    lastSender ! str
    lastSender ! (node,map.keys.toList)
}
}

```

MainFS.scala

```
package filesystem
```

```

import akka.actor.{ReceiveTimeout, Terminated, Actor}
import com.ericsson.otp.erlang.OtpErlangPid
import myframework.{ScalaMessage, ErlangNode}

```

```

import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.Future
import scala.concurrent.duration._

```

```
/**
```

```

 * Главный актор в файловой системе. Данный актор инициализирует систему и
обрабатывает запросы к ней.

```

```

 *

```

```

 * Данный актор имеет два состояния: инициализация системы и работы с системой

```

```

 *

```

```

 * ==Принимаемые сообщения==

```

```

 *

```

```

 * ===Инициализация системы===

```

```

 * - [[InitCommand]] - сообщение для начала инициализации системы.<br>

```

```

* - [[ConnectedNode]] - сообщение о подключенном к системе узле<br>
* - [[NodeNotConnect]] - сообщение о неподключенном к системе узле <br>
* - [[Terminated]] - сообщение об остановке актора, за которым следит данный
актор<br>
* - [[ReceiveTimeout]] - сообщение о завершении времени ожидания<br>
*
* ===Работа системы===
* - [[FileDeleted]] - сообщение об удалении файла<br>
* - [[FileRenamed]] - сообщение о переименовании файла<br>
* - [[FileCopied]] - сообщение о копировании файла<br>
* - [[CommandToFS]] - сообщение с командами для файловой системы<br>
*/
class MainFS extends Actor {

  /**Карта процессов файловой системы, где key- имя файла, value - pid процесса
  обслуживающего файл*/
  private var filesystem = Map.empty[String, Map[String,
  OtpErlangPid]].withDefaultValue(Map.empty)

  /**Ссылка на инициализирующий актор [[InitFileSystemProcess]]*/
  private val initProc = context.actorOf(InitFileSystemProcess.props("InitProcess"),
  "InitProcess")
  context.watch(initProc)
  /**Количество текущих работающих команд*/
  private var countWorkers = 0
  /**Ссылка на актора - клиента системы*/
  private var lastSender = Actor.noSender

  /**Множество подключенных узлов*/
  private var setConnectedNode: Set[ConnectedNode] = Set.empty

  /**
   * Состояние инициализации системы
   *
   */
  override def receive: Receive = {
    case InitCommand(nodeList) =>
      context.setReceiveTimeout(7 second)
      initProc ! ScalaMessage(nodeList)
      lastSender = sender()
    case conNode @ ConnectedNode(nameNode, mapProc) =>
      //по хорошему проверка, что таких нодов еще нет в нашей карте

```

```

    filesystem += nameNode -> mapProc
    println(nameNode + " is connected")
    setConnectedNode += conNode
case NodeNotConnect(node) =>
    println(node + " isn't connected")
case Terminated(inpr) if inpr equals initProc =>
    context.setReceiveTimeout(Duration.Undefined)
    println("Initialization complite!")
    lastSender ! InitDone(setConnectedNode)
    context.become(workWithFS)
case ReceiveTimeout =>
    context.setReceiveTimeout(Duration.Undefined)
    if (setConnectedNode.nonEmpty) {
        lastSender ! InitDone(setConnectedNode)
        context.become(workWithFS)
    }
    else{
        lastSender ! InitError
        context.stop(self)
    }
}

/**
 *
 * Состояние работы с системой
 */
private def workWithFS:Receive = {
    case FileDeleted(node, nameFile) =>
        if (filesystem(node).nonEmpty) {
            val mapFile = filesystem(node)
            if (mapFile.contains(nameFile)) {
                val newMap = mapFile - nameFile
                filesystem -= node
                filesystem += node -> newMap
                val str = node + ":" + nameFile + " was deleted!"
                println(str)
                lastSender ! (str, node, newMap)
                countWorkers = countWorkers - 1
            }
        }
    case FileRenamed(node, oldFile, newFile) =>
        if (filesystem(node).nonEmpty) {
            val mapFile = filesystem(node)

```



```

    if (mapFile.contains(oldFile)) {
        val pidOldFile = mapFile(oldFile)
        val helpMap = mapFile - oldFile
        val newMap = helpMap + (newFile->pidOldFile)
        filesystem -= node
        filesystem += node -> newMap
        val str = node + ":" + oldFile + " was renamed to " + newFile + " !"
        println(str)
        lastSender ! (str, node, newMap)
        countWorkers = countWorkers - 1
    }
}

case FileCopied(node, originalFile, newFile, pidNewFile) =>
    if (filesystem(node).nonEmpty) {
        val mapFile = filesystem(node)
        if (!mapFile.contains(newFile)) {
            val newMap = mapFile + (newFile->pidNewFile)
            filesystem -= node
            filesystem += node -> newMap
            val str = node + ":" + originalFile + " was copied to " + newFile + " !"
            println(str)
            lastSender ! (str, node, newMap)
            countWorkers = countWorkers - 1
        }
    }

case CommandToFS(commands) =>
    lastSender = sender()
    commands.foreach (com => Future(exeCommand(com)) )
}

/**
 * Запускает акторы, обрабатывающие команды к файловой системе
 *
 * @param command список команд к файловой системе
 */
private def exeCommand(command:CommandToNode) = {
    val CommandToNode(node, commandToFile) = command
    if (!filesystem(node).isEmpty) {
        if (ErlangNode.localErlangNode.ping(node, 1000)) {
            val mapPR = filesystem(node)
            commandToFile match {
                case DeleteFile(nameFile) =>
                    val pid = mapPR.get nameFile

```

```

        pid match {
            case Some(x) =>
                val worker = context.actorOf(FileSystemProcess.props("delete:" +
countWorkers), "delete:" + countWorkers)
                worker ! ScalaMessage((x, "delete"))
                countWorkers = countWorkers + 1
            case None =>
                println("File: " + node + ":" + nameFile + " not found")
        }
        case RenameFile(oldFile,newFile) =>
            val pid = mapPR get oldFile
            pid match {
                case Some(x) =>
                    val worker = context.actorOf(FileSystemProcess.props("rename:" +
countWorkers), "rename:" + countWorkers)
                    worker ! ScalaMessage((x,newFile, "rename"))
                    countWorkers = countWorkers + 1
                case None =>
                    println("File: " + node + ":" + oldFile + " not found")
            }
        case CopyFile(nameFile,copyName) =>
            val pid = mapPR get nameFile
            pid match {
                case Some(x) =>
                    val worker = context.actorOf(FileSystemProcess.props("copy:"
+countWorkers), "copy:" + countWorkers)
                    worker ! ScalaMessage((x,copyName, "copy"))
                    countWorkers = countWorkers + 1
                case None =>
                    println("File: " + node + ":" + nameFile + " not found")
            }
        }
    } else println(node + " not connected")
} else println(node + " - node not found!")
}
}

```

InitFileSystemProcess.scala

```

package filesystem

import akka.actor.Props
import com.ericsson.otp.erlang._

```

```

import myframework.ErlangAdapteeActor

/**Актор - инициализатор файловой системы
 * ==Принимаемые сообщения==
 *
 * ===Scala сообщения===
 * - Set[String]- узлы для инициализации<br>
 *
 * ===Erlang сообщения===
 * - OtpErlangTuple(node,mapProc) - ответ от инициализированного Erlang узла<br>
 *
 *
 * @param name имя актора
 */
class InitFileSystemProcess(name: String) extends ErlangAdapteeActor(name) {
  /**Множество ожидаемых подключения узлов*/
  private var setNodes: Set[String] = Set.empty

  /**@inheritdoc
   *
   * @param msg обрабатываемое сообщение
   * @tparam U тип сообщения
   */
  override def erlangBehavior[U <: OtpErlangObject](msg: U): Unit = msg match
  {
    case m: OtpErlangTuple =>
      m.elementAt(1) match {
        case mapFiles: OtpErlangMap =>
          val f = russianChars _
          val scalaMapFiles: Map[String, OtpErlangPid] = mapFiles
            .keys() //берем все названия файлов
            .map(f)
            .zip(mapFiles.values()) //все в кортеж
            .groupBy(_._1)
            .map { case (k, v) => (k,
v.map(_._2).apply(0).asInstanceOf[OtpErlangPid]) }
          context.parent !
          ConnectedNode(m.elementAt(0).toString.tail.dropRight(1), scalaMapFiles)
          setNodes -= m.elementAt(0).toString.tail.dropRight(1)
          if (setNodes.isEmpty)
            context.stop(self)
          else

```

```

        self ! ContinueGetMessage
    }
}

/**@inheritdoc
 *
 * @param msg обрабатываемое сообщение
 * @tparam U тип сообщения
 */
override def scalaBehavior[U](msg: U): Unit = msg match {
    case nodeList: Set[String] =>
        nodeList foreach (node => startFSonNode(node))
        self ! ContinueGetMessage
}

/**Подключение узла к файловой системе
 *
 * @param node подключаемый узел
 */
private def startFSonNode(node: String) = {
    val msg = new Array[OtpErlangObject](2)
    msg(0) = new OtpErlangAtom("getFS")
    msg(1) = innerErlProc.idThisActor
    if (!innerErlProc.send("getFS", node, new OtpErlangTuple(msg)))
        context.parent ! NodeNotConnect(node)
    else setNodes += node
}
}

/**
 * Объект-путник, служащий поставщиком фабричного метода для создания
конфигурация для актора [[InitFileSystemProcess]]
 */
object InitFileSystemProcess {
    def props(name: String): Props = Props(new InitFileSystemProcess(name))
}

```

WorkerFileSystem.scala

```
package filesystem
```

```

import akka.actor.Props
import com.ericsson.otp.erlang._
import myframework.ErlangAdapteeActor

```

```

/**Объект-путник, служащий поставщиком фабричного метода для создания конфигурация
для актора [[FileSystemProcess]]*/
object FileSystemProcess {
  def props(name: String) = Props(new FileSystemProcess(name))
}

/**Актор - обработчик команды файловой системы
  * ==Принимаемые сообщения==
  *
  * ===Scala сообщения===
  * 'pidFile - идентификатор процесса, обрабатывающего файл'<br>
  * - scala.Tuple2(pidFile: OtpErlangPid, "delete") - команда на удаление файла<br>
  * - scala.Tuple3(pidFile: OtpErlangPid, newName:String, "rename") - команда на
переименование файла<br>
  * - scala.Tuple3(pidFile: OtpErlangPid, copyName:String, "copy") - команда на
копирование файла<br>
  *
  * ===Erlang сообщения===
  * - OtpErlangTuple("deleted", node, file) - ответ об удалении файла<br>
  * - OtpErlangTuple("renamed", node, oldFileName, newFileName) - ответ о
переименовании файла<br>
  * - OtpErlangTuple("copied", node, oldFileName, newFileName, pidNewFile) - ответ о
копировании файла<br>
  *
  *
  * @param name имя актора
  */
class FileSystemProcess(name: String) extends ErlangAdapteeActor(name) {

  /**@inheritdoc
    *
    * @param msg обрабатываемое сообщение
    * @tparam U тип сообщения
    */
  override def erlangBehavior[U <: OtpErlangObject](msg: U): Unit = msg match
  {
    case tuple: OtpErlangTuple =>
      tuple.elementAt(0).toString match {
        case "deleted" =>
          val file = russianChars(tuple.elementAt(2))

```

```

        val mes =
FileDeleted(tuple.elementAt(1).toString.tail.dropRight(1), file)
        context.parent ! mes
        context.stop(self)
    case "renamed" =>
        val oldfile = russianChars(tuple.elementAt(2))
        val newfile = russianChars(tuple.elementAt(3))
        val mes = FileRenamed(tuple.elementAt(1).toString.tail.dropRight(1),
            oldfile, newfile)
        context.parent ! mes
        context.stop(self)
    case "copied" =>
        val oldfile = russianChars(tuple.elementAt(2))
        val newfile = russianChars(tuple.elementAt(3))
        val mes = FileCopied(tuple.elementAt(1).toString.tail.dropRight(1),
            oldfile, newfile,
            tuple.elementAt(4).asInstanceOf[OtpErlangPid])
        context.parent ! mes
        context.stop(self)
    }
}

/**@inheritdoc
 *
 * @param msg обрабатываемое сообщение
 * @tparam U тип сообщения
 */
override def scalaBehavior[U](msg: U): Unit = msg match {
    case (pidFile: OtpErlangPid, "delete") =>
        val msg = new Array[OtpErlangObject](2)
        msg(0) = new OtpErlangAtom("delete")
        msg(1) = innerErlProc.idThisActor
        innerErlProc.send(pidFile) (new OtpErlangTuple(msg))
        self ! ContinueGetMessage
    case (pidFile: OtpErlangPid, newName:String, "rename") =>
        val msg = new Array[OtpErlangObject](3)
        msg(0) = new OtpErlangAtom("rename")
        msg(1) = new OtpErlangString(newName)
        msg(2) = innerErlProc.idThisActor
        innerErlProc.send(pidFile) (new OtpErlangTuple(msg))
        self ! ContinueGetMessage
    case (pidFile: OtpErlangPid, copyName:String, "copy") =>
        val msg = new Array[OtpErlangObject](3)

```

```

        msg(0) = new OtpErlangAtom("copy")
        msg(1) = new OtpErlangString(copyName)
        msg(2) = innerErlProc.idThisActor
        innerErlProc.send(pidFile) (new OtpErlangTuple(msg))
        self ! ContinueGetMessage
    }
}

```

AllUsingMessage.scala

```
package filesystem
```

```
import com.ericsson.otp.erlang.{OtpErlangObject, OtpErlangPid}
```

```
/**Сообщение об успешно подключенном узле системы
```

```
 *
```

```
 * @param nodeName имя узла
```

```
 * @param mapProc карта процессов подключенного узла
```

```
 */
```

```
case class ConnectedNode(nodeName:String,mapProc:Map[String,OtpErlangPid])
```

```
/**Сообщение о неподключенном узле
```

```
 *
```

```
 * @param nodeName имя узла
```

```
 */
```

```
case class NodeNotConnect(nodeName:String)
```

```
/**
```

```
 * Сообщение для начала инициализации системы.
```

```
 * @param nodesList узлы для инициализации
```

```
 */
```

```
case class InitCommand(nodesList:Set[String])
```

```
/**
```

```
 * Сообщение об успешной инициализации системы
```

```
 * @param nodes успешно подключенные узлы
```

```
 */
```

```
case class InitDone(nodes:Set[ConnectedNode])
```

```
/**
```

```
 * Сообщение об неполадках во время инициализации
```

```
 */
```

```
case object InitError
```

```

/**
 * Абстрактный класс для команд к файлам
 */
abstract class CommandToFile

/**
 * Команда для удаления файла
 * @param file имя удаляемого файла
 */
case class DeleteFile(file: String) extends CommandToFile

/**
 * Команда для переименования файла
 * @param file имя файла
 * @param newFileName новое имя файла
 */
case class RenameFile(file: String, newFileName: String) extends CommandToFile

/**
 * Команда для копирования файла
 *
 * @param file имя файла
 * @param newPlace имя копии файла file
 */
case class CopyFile(file: String, newPlace: String) extends CommandToFile

/**
 * Сообщение с командой для конкретного узла
 * @param node имя узла, к которому адресована команда
 * @param command команда для файла на узле
 */
case class CommandToNode(node: String, command: CommandToFile)

/**
 * Сообщение с командами для файловой системы
 * @param commands команды для файловой системы
 */
case class CommandToFS(commands: Set[CommandToNode])

/**
 * Сообщение об удалении файла
 * @param node имя узла, на котором был удален файл
 * @param file имя удаленного файла

```



```

    */
    case class FileDeleted(node: String, file: String)

    /**
     * Сообщение о переименовании файла
     * @param node имя узла, на котором был переименован файл
     * @param oldFile старое имя файла
     * @param newFile новое имя файла
     */
    case class FileRenamed(node:String, oldFile:String, newFile:String)

    /**
     * Сообщение о копировании файла
     * @param node имя узла, на котором был скопирован файл
     * @param originalFile имя оригинала файла
     * @param copyFile имя скопированного файла
     * @param pidCopy идентификатор процесса обслуживающего новый файл
     */
    case class FileCopied(node:String, originalFile:String, copyFile:String,
pidCopy:OtpErlangPid)

```

Controller.java

```

import akka.actor.ActorRef;
import akka.actor.ActorSystem;
import akka.actor.Inbox;
import akka.actor.Props;
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.control.*;
import scala.Tuple2;
import scala.Tuple3;
import scala.Tuple4;
import scala.collection.immutable.List;
import scala.collection.immutable.Set;
import scala.concurrent.duration.Duration;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;
import filesystem.ClientForGUI;

```

```

public class Controller {

    private scala.collection.mutable.HashSet<String> setNodes = new
scala.collection.mutable.HashSet<>();
    final ActorSystem system = ActorSystem.create("FileSystemOnActor");
    final ActorRef clientActor = system.actorOf(Props.create(ClientForGUI.class),
"clientActor");

    private final Inbox inbox = Inbox.create(system);
    @FXML
    private Label labelNodes;

    @FXML
    private ListView<String> listViewCompliteCommand;

    @FXML
    private TreeView<String> treeViewNodesFiles;

    @FXML
    private TextField textFieldNode;

    @FXML
    private Button buttonAddNode;

    @FXML
    private Button buttonStartFS;

    @FXML
    private TextField textFieldCopy;

    @FXML
    private TextField textFieldRename;

    @FXML
    void clickOnButtonAddNode(ActionEvent event) {
        if(!textFieldNode.getText().equals("")){
            setNodes.add(textFieldNode.getText());
            labelNodes.setText(setNodes.reversed().mkString("; "));
            textFieldNode.clear();
        }
    }

    @FXML
    void clickOnButtonStartFS(ActionEvent event) {

```

```

        textFieldNode.setDisable(true);
        buttonAddNode.setDisable(true);
        buttonStartFS.setDisable(true);
        final scala.Tuple2<String,scala.collection.mutable.Set> tuple =
scala.Tuple2.apply("init",setNodes);
        inbox.send(clientActor, tuple);
        try {
            Object answ = inbox.receive(Duration.create(12, TimeUnit.SECONDS));
            if (answ instanceof List){
                List listTuples = (List) answ;
                TreeItem<String> rootTree = new TreeItem<>("FileSystem");
                for(int i = 0 ; i < listTuples.length() ; i++){
                    if (listTuples.apply(i) instanceof Tuple2){
                        Tuple2 t = (Tuple2)listTuples.apply(i);
                        String node = (String)t._1();
                        Set<String> setFile = (Set<String>)t._2();
                        List<String> listFile = setFile.toList();
                        TreeItem<String> nodeTree = new TreeItem<>(node);
                        for(int j = 0 ; j < listFile.length() ; j++){
                            TreeItem<String> item = new
TreeItem<>(listFile.apply(j));
                                nodeTree.getChildren().add(item);
                            }
                            rootTree.getChildren().add(nodeTree);
                        }
                    }
                }
                treeViewNodesFiles.setRoot(rootTree);
            }else{
                if (answ instanceof String) {
                    String str = (String) answ;
                    textFieldNode.setText(str);
                }
            }
        } catch (java.util.concurrent.TimeoutException e) {
            // timeout
            textFieldNode.setText("Что-то пошло не так?!");
        }
    }
}

```

@FXML

```

void clickOnCopyButton(ActionEvent event) {
    String oldFile =
treeViewNodesFiles.getSelectionModel().getSelectedItem().getValue();
}

```

```

        String node =
treeViewNodesFiles.getSelectionModel().getSelectedItem().getParent().getValue();
        String newFile = textFieldCopy.getText();
        final Tuple4<String,String,String,String> mes =
Tuple4.apply("copy",node,oldFile,newFile);
        inbox.send(clientActor, mes);
        textFieldCopy.clear();
    }

```

```

@FXML
void clickOnRenameButton(ActionEvent event) {
    String oldFile =
treeViewNodesFiles.getSelectionModel().getSelectedItem().getValue();
    String node =
treeViewNodesFiles.getSelectionModel().getSelectedItem().getParent().getValue();
    String newFile = textFieldRename.getText();
    final Tuple4<String,String,String,String> mes =
Tuple4.apply("rename",node,oldFile,newFile);
    inbox.send(clientActor, mes);
    textFieldRename.clear();
}

```

```

@FXML
void clickOnDeleteButton(ActionEvent event) {
    String file =
treeViewNodesFiles.getSelectionModel().getSelectedItem().getValue();
    String node =
treeViewNodesFiles.getSelectionModel().getSelectedItem().getParent().getValue();
    final Tuple3<String,String,String> mes = Tuple3.apply("delete",node,file);
    inbox.send(clientActor, mes);
}

```

```

private Thread myThread = new Thread(()-> {
    while(true){
        try {
            Object answ = inbox.receive(Duration.create(5,
TimeUnit.MILLISECONDS));
            if (answ instanceof String){
                String s = (String) answ;
                listViewCompliteCommand.getItems().add(s);
            }else{
                if (answ instanceof Tuple2) {
                    Tuple2 t = (Tuple2) answ;

```

```

        TreeItem<String> rootTree = treeViewNodesFiles.getRoot();
        String node = (String) t._1();
        List<String> listFile = (List<String>) t._2();
        TreeItem<String> nodeTree = new TreeItem<>(node);
        rootTree.getChildren().remove(
            rootTree.getChildren().stream()
                .filter(x ->
x.getValue().equals(node)).findFirst().orElse(null));
        for (int j = 0; j < listFile.length(); j++) {
            TreeItem<String> item = new
TreeItem<>(listFile.apply(j));
            nodeTree.getChildren().add(item);
        }
        rootTree.getChildren().add(nodeTree);
    }
}
} catch (TimeoutException e) {continue;}
}
});

@FXML
void clickOnRunButton(ActionEvent event) {
    inbox.send(clientActor, "exe");
    if (!myThread.isAlive())
        myThread.start();
}

}

```

filesystem.erl

```

-module(filesystem).
-author("Данил").

%% API
-export([startFS/0,
recurserM/1]).

recurserM(Dir)->
    {ok, Filenames}=file:list_dir(Dir),
    processOneM(Dir,Filenames).

processOneM(Parent, [First|Next])->

```

```

FullName=Parent++"/"++First,
IsDir=filelib:is_dir(FullName),
if
    IsDir==true -> recursorM(FullName);
    IsDir==false -> io:format(FullName++"~n"),
        Master = whereis(superfilesystem),   %%superfilesystem - имя процесса
aggregationProcess
        Master!{FullName,erlang:spawn(fun() ->processBody(FullName) end)}
end,
processOneM(Parent,Next);
processOneM(_Parent,[])->ok.

processBody(FullName) ->
    receive
        {delete,PidWorkerOnMainActor} ->
            Rez = file:delete(FullName),
            if
                Rez == ok -> PidWorkerOnMainActor ! {deleted,node(),FullName},
exit(self(),normal);
                true -> PidWorkerOnMainActor ! {error}
            end;
        {rename, NewName, PidWorkerOnMainActor} ->
            Rez = file:rename(FullName,NewName),
            if
                Rez == ok -> PidWorkerOnMainActor ! {renamed,node(),FullName,NewName},
processBody(NewName);
                true -> PidWorkerOnMainActor ! {error}
            end;
        {copy, NewPlace, PidWorkerOnMainActor} ->
            {Rez,_} = file:copy(FullName,NewPlace),
            if
                Rez == ok ->
                    PidWorkerOnMainActor ! {copied,node(),FullName,NewPlace,erlang:spawn(fun()
->processBody(NewPlace) end)};
                    true -> PidWorkerOnMainActor ! {error}
            end

end,

processBody(FullName).

aggregationProcess(MapAllFiles)->
    receive

```

```

{Key,Value} ->
    AllFiles = maps:put(Key,Value,MapAllFiles),
    aggregationProcess(AllFiles);
ok->
    ControlPr = spawn(fun() -> controlProcess(MapAllFiles) end),
    register(getFS,ControlPr),
    exit(self(),normal)
end,
aggregationProcess(MapAllFiles).

controlProcess(MapAllFiles)->
    receive
        {getFS,PidSender} ->
            io:format("FS connected ~n"),
            PidSender ! {node(),MapAllFiles},
            exit(self(),normal)
    end,
    controlProcess(MapAllFiles).

startFS()->
    AgrProcPid = spawn(fun()-> aggregationProcess(maps:new()) end),
    register(superfilesystem,AgrProcPid),
    AgrProcPid ! recursorM(".").

```

«Поиск минимального значения в непрерывном потоке данных»

ErlangListener.scala

```

package bigdata

import akka.actor.{Actor, Props}
import com.ericsson.otp.erlang.{OtpErlangAtom, OtpErlangLong}
import myframework.ErlangProcess

import scala.annotation.tailrec

/**
 * Актор, принимающий сообщения от Erlang процессов и отправляющий своему
 * [[GuardCollector]]
 *
 * ==Принимаемые сообщения==
 *
 * ===Scala сообщения===
 *
 * - "get" - сообщение о начале получении всех сообщений от Erlang<br>
 *
 */

```

```

* ===Erlang сообщения===
* - OtpErlangLong - число, сгенерированное Erlang процессом<br>
*
* ==Отправляемые сообщения==
*
* ===Scala сообщения===
* - Int - полученное число для обработки<br>
*
* '''Общается с:''' [[GuardCollector]], удаленными Erlang процессами.
* @param name имя актора
*/
class ErlangListener(name:String) extends Actor{
  /**Внутренний Erlang процесс для общения с другими Erlang процессами*/
  private val innerErlProc = ErlangProcess(name)
  override def receive: Receive ={
    case "get" =>
      listenErlang()
  }

  /**
   * Получение сообщений от erlang процессов и отправка их своему
   [[GuardCollector]]
   */
  @tailrec
  private def listenErlang(): Unit ={
    val o = innerErlProc.receive
    val mess = o.getOrElse(new OtpErlangAtom("nothing"))
    mess match {
      case num:OtpErlangLong =>
        context.parent ! num.intValue()
      case _ =>
    }
    listenErlang()
  }
}

/**Объект-путник, служащий поставщиком фабричного метода для создания конфигурация
для актора [[ErlangListener]]*/
object ErlangListener{
  def props(name:String) = Props(new ErlangListener(name))
}

```



```

package bigdata

import akka.actor.{Actor, Props}

/**
 * Актор коллектор, уменьшающий поток данных к главному процессу.
 * Для получения данных создает себе помощника [[ErlangListener]]
 * ==Принимаемые сообщения==
 * - Int - числа для поиска минимума<br>
 *
 * ==Отправляемые сообщения==
 * - [[DataPart]] - сообщение с набором данных для обработки <br>
 *
 * '''Общается с:''' [[ErlangListener]], [[MainConsumer]]
 * @param name имя актора
 */
class GuardCollector(name:String) extends Actor{
  /**Множество чисел для обработки*/
  private var setInt: Set[Int] = Set.empty

  /**Создание акотра слушателя [[ErlangListener]]*/
  private val listener = context.actorOf(ErlangListener.props(name))
  listener ! "get"
  override def receive: Receive = {
    case n:Int =>
      setInt+=n
      if(setInt.size == 5000){
        context.parent ! DataPart(setInt)
        setInt = Set.empty
      }
  }
}

/**Объект-путник, служащий поставщиком фабричного метода для создания конфигурация
для актора [[GuardCollector]]*/
object GuardCollector{
  def props(name:String) = Props(new GuardCollector(name))
}

MainConsumer.scala

package bigdata

import akka.actor.{Actor, Props, Terminated}

```

```

import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.Future

/**
 * Главный актор в данной системе.
 * Для получения данных этот актор создает [[GuardCollector]], а для обработки
полученных чисел
 * создает [[MinSearcher]]
 * В своем состоянии хранит искомые величины.
 *
 * ==Принимаемые сообщения==
 * - "Start system" - сообщение о запуске системы<br>
 * - [[DataPart]] - сообщение с данными для обработки<br>
 * - [[Terminated]] - сообщение об остановке одного из [[GuardCollector]]<br>
 * - [[LocalMin]] - сообщение с минимумом в некоторой порции данных<br>
 * - "Get min" - запрос состояния актора<br>
 *
 * ==Отправляемые сообщения==
 * - "get" - сообщение о начале получения данных<br>
 * - [[DataPart]] - сообщение с данными для обработки<br>
 *
 * '''Общается с:''' [[GuardCollector]], [[MinSearcher]], интерфейсом
пользователя.
 */
class MainConsumer extends Actor{
  /**Глобальный минимум*/
  private var mainMin = 99999999
  /**Последний полученный минимум*/
  private var localMin = mainMin
  override def receive: Receive = {
    case "Start system" =>
      startAllCollectors(5)
      println("System start!")
    case Terminated(worker) =>
      println(worker.toString() + " : Terminated")
    case DataPart(setInt) =>
      Future{searchMin(setInt)}
      println("Get bigdata.DataPart from: " + sender().path)
    case LocalMin(min) =>
      localMin = min
      if(min < mainMin)
        mainMin = min
      println("Get new mins: " + localMin)
  }
}

```

```

    case "Get min" =>
        sender() ! (mainMin, localMin)
}

/**
 *Метод для запуска [[GuardCollector]]
 * @param count количество запускаемых [[GuardCollector]]
 */
def startAllCollectors(count: Int): Unit = {
    if (count > 0) {
        val collector = context.actorOf(GuardCollector.props("collector" +
count), "collector" + count)
        context.watch(collector)
        collector ! "get"
        startAllCollectors(count-1)
    }
}

/**
 * Делит данные на небольшие порции для обработки, создает [[MinSearcher]] и
отправляет им эти порции
 * @param setInt набор данных для обработки
 */
def searchMin(setInt: Set[Int]) = {
    def helpFun(tailSet: List[Int]): Unit = tailSet match {
        case Nil => Unit
        case x: List[Int] =>
            val k = x.splitAt(1000)
            val searcher = context.actorOf(Props[MinSearcher])
            searcher ! DataPart(k._1.toSet)
            helpFun(k._2)
    }
    helpFun(setInt.toList)
}
}

```

MinSearcher.scala

```

package bigdata

import akka.actor.Actor

/**

```

```

* Актор, который ищет в небольшой порции данных min и отправляет его
[[MainConsumer]]
*
* ==Принимаемые сообщения==
* - [[DataPart]] - сообщение с данными для обработки<br>
*
* ==Отправляемые сообщения==
* - [[LocalMin]] - сообщение с минимумом в полученной порции данных<br>
*
*   '''Общается с:''' [[MainConsumer]]
*/
class MinSearcher extends Actor{
  override def receive: Receive = {
    case DataPart(setInt) =>
      context.parent ! LocalMin(setInt.min)
      context.stop(self)
  }
}

```

DataPart.scala

```

package bigdata

/**
 * Сообщение с данными для обработки
 * @param data данные для обработки
 */
case class DataPart(data:Set[Int])

```

LocalMin.scala

```

package bigdata

/**
 * Сообщение с найденным минимумом
 * @param min найденный минимум
 */
case class LocalMin(min:Int)

```

Controller.java

```

import akka.actor.ActorRef;
import akka.actor.ActorSystem;
import akka.actor.Inbox;
import akka.actor.Props;
import bigdata.MainConsumer;

```

```

import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.control.*;
import scala.Tuple2;
import scala.concurrent.duration.Duration;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

public class Controller {

    private scala.collection.mutable.HashSet<String> setNodes = new
scala.collection.mutable.HashSet<>();
    final ActorSystem system = ActorSystem.create("ScalaConsumer");
    final ActorRef clientActor = system.actorOf(Props.create(MainConsumer.class),
"bigdata.MainConsumer");
    private final Inbox inbox = Inbox.create(system);

    @FXML
    private Label lableGlobalMin;

    @FXML
    private Label lableLocalMin;

    @FXML
    void clickStartButton(ActionEvent event) {
        String mes = "Start system";
        inbox.send(clientActor, mes);
    }

    @FXML
    void clickUpdateButton(ActionEvent event) {
        inbox.send(clientActor, "Get min");
        try {
            Object answ = inbox.receive(Duration.create(10, TimeUnit.SECONDS));
            if (answ instanceof Tuple2) {
                Tuple2 t = (Tuple2) answ;
                Integer globalMin = (Integer) t._1();
                Integer localMin = (Integer) t._2();
                lableGlobalMin.setText(globalMin.toString());
                lableLocalMin.setText(localMin.toString());
            }
        }
    }
}

```

```

    } catch (TimeoutException e) {
        lableGlobalMin.setText("Very long answer");
        lableLocalMin.setText("Very long answer");
    }
}
}

```

data_prod.erl

```

-module('data_prod').
-author("Данил").

```

```

%% API
-export([start/2]).

```

```

data_producer(Collector) ->
    timer:sleep(rand:uniform(10000)),
    Ch = rand:uniform(10000),
    %%io:format("~p~n", [Ch]),
    {Collector, 'ScalaErlangNode@danil-pc'} ! Ch,
    data_producer(Collector).

start([],_) -> ok;
start([_|T],3500) -> start(T,0);
start(Collectors,Count) ->
    [H|_] = Collectors,
    P = spawn(fun() -> data_producer(H)end),
    io:format("~w producer start. Send to ~w ~n", [P,H]),
    start(Collectors,Count+1).

```