

СОДЕРЖАНИЕ

[ОПРЕДЕЛЕНИЯ 5](#_Toc513629588)

[ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ 6](#_Toc513629589)

[ВВЕДЕНИЕ 7](#_Toc513629590)

[1. Обзор предметной области 8](#_Toc513629591)

[1.1. Описание интегрируемых продуктов 11](#_Toc513629592)

[1.2. Выполнение рассматриваемого процесса вручную 12](#_Toc513629593)

[1.3. Предлагаемый вариант автоматизации 14](#_Toc513629594)

[2. Проектирование 15](#_Toc513629595)

[2.1. Функциональная архитектура 15](#_Toc513629596)

[2.2. Проблемы, исследуемые в рамках ВКР 18](#_Toc513629597)

[2.3. Системная архитектура 25](#_Toc513629598)

[2.4. Программная архитектура 32](#_Toc513629599)

[2.5. Архитектура данных 41](#_Toc513629600)

[3. Оценка отказоустойчивости системы 46](#_Toc513629601)

[3.1. Методы оценки 46](#_Toc513629602)

[3.2. Результаты оценки 48](#_Toc513629603)

[ЗАКЛЮЧЕНИЕ 51](#_Toc513629604)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 52](#_Toc513629605)

[ПРИЛОЖЕНИЕ А 54](#_Toc513629606)

[ПРИЛОЖЕНИЕ Б 58](#_Toc513629607)

[ПРИЛОЖЕНИЕ В 59](#_Toc513629608)

ОПРЕДЕЛЕНИЯ

* *Система Хранения Данных* – комплексное программно-аппаратное решение по организации надёжного хранения информационных ресурсов и предоставления гарантированного доступа к ним [1].
* *LUN* – номер, используемый для идентификации логического устройства хранения данных, адресуемого протоколами iSCSI и FCP [3].
* *Storage Pool* – единое хранилище гомогенных или гетерогенных физических дисков, на основе которого могут быть созданы *LUN* [2]*.*
* *Redundant Array of Independent Disks* – технология виртуализации данных, которая объединяет несколько дисков в логический элемент для избыточности и повышения производительности [4].
* *Virtual Pool* – абстракция над нижележащими *Storage Pools* с определёнными характеристиками производительности и защиты данных [5].

ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

* *СХД* – Система Хранения Данных.
* *SP –* Storage Pool.
* *RAID –* Redundant Array of Independent Disks.
* *VP –* Virtual Pool.

ВВЕДЕНИЕ

Выделение ресурсов блокового доступа *СХД* c предъявлением требований приложениями к изменяемым во времени показателям производительности является одной из ключевых особенностей разрабатываемых компанией ЕМС *СХД*. Данная функциональность реализована только в старших поколениях *СХД* и не поддерживается более бюджетными конфигурациями *СХД*, такими, как EMC VNX. Реализовать данную концепцию для *СХД* рассматриваемой серии можно с помощью решения по интеграции 3 различных продуктов компании EMC, которое позволит автоматизировать процесс консолидации ресурсов *СХД* в *VP*, который будет удовлетворять требованиям заданного приложения к времени отклика пространства блокового доступа, с последующим выделением на нем *LUN*. Управление ресурсами *СХД* является глобальной задачей, выходящей за рамки одного центра обработки данных на уровень взаимодействия между различными *СХД*, расположенных в различных частях света. Построение отказоустойчивого распределенного решения позволяет обеспечить надежность процесса и консистентность потоков данных при управлении ресурсами *СХД* даже при выходе каналов связи, соединяющих один или несколько центров обработки данных с другими.

1. Обзор предметной области

На данный момент с помощью *СХД* ИТ-компании осуществляют эффективное использование и управление промышленными приложениями, такими как сервера баз данных (PostgreSQL, Oracle Database), почтовые сервера, разнообразные файловые хранилища и виртуальные рабочие станции. Различные приложения в зависимости от характера использования ресурсов блокового доступа имеют к ним разные требования:

* Динамические показатели производительности (время отклика, пропускная способность с учетом используемого канала связи и др.)
* Надёжность (настройки резервного копирования данных, организация бэкапов и репликации)
* Доступ (защищенный доступ и протоколы доступа)

В *СХД* серии EMC VNX вводится концепция *Storage Pool (SP)*, которая подразумевает под собой объединение однотипных или разнотипных дисков в логическую структуру, которая конфигурирует их в набор однотипных *RAID*-массивов с целью последующего выделения на них *LUN* в абстракции от конкретно используемого набора дисков (SSD, NL-SAS, SATA, SAS) или конфигурации *RAID*-массива (*RAID10*, *RAID5*, *RAID6* и др.). Осуществление конфигурирования добавленных в *SP* дисков в *RAID*-массивы осуществляется автоматически средствами *СХД*. *SP,* основанные на дисках одного типа называются гомогенными, а на дисках разного типа – гетерогенными. Конкретная конфигурация целевых *RAID*-массивов указывается при создании *SP*. Пример различных конфигураций гомогенных *SP* приведен на рис. 1.

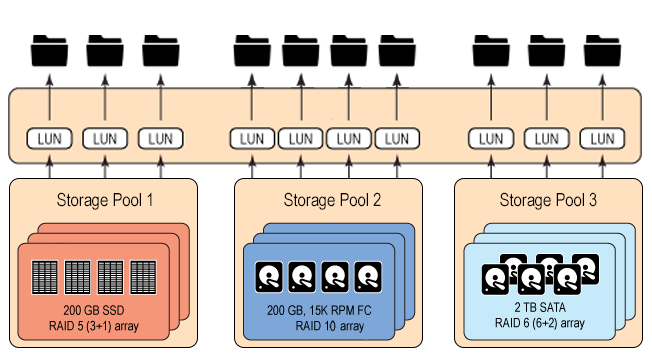


Рисунок 1 – Гомогенные *SP*

Первичным ресурсом *СХД*, относящемуся к ресурсам блокового доступа, являются выделенные на *SP* *LUN*, которые могут иметь разные параметры производительности в зависимости от статических и динамических характеристик *SP*, на котором они были выделены.

К статическим характеристикам относят, например,

* Количество дисков
* Тип дисков
* Объем дисков

К динамическим характеристикам относят:

* Число запросов в единицу времени
* Тип запроса (чтение / запись, последовательно / случайно)
* Размер обрабатываемого блока данных
* Утилизация

Так как один и тот же *SP* может быть использован одновременно для выделения *LUN* более, чем одному приложению, то его производительность зависит от нагрузки, оказываемой всеми приложениями, использующими *LUN* данного отдельно взятого *SP*. Ввиду большой волатильности динамических метрик производительности, их принято рассматривать в агрегации с использованием фиксированного временного промежутка, который варьируется в зависимости от специфики конкретно поставленной задачи.

Концепция *Virtual Pool (VP)* служит для получения более высокого уровня абстракции над *SP*. Она позволяет осуществить группировку однотипных *SP*, где параметром группировки может являться:

* Статические характеристики *SP* (Число оборотов в минуту, тип дисков, форм-фактор)
* Тип доступных протоколов доступа (*iSCSI*, *FC*)
* Характеристики надежности хранения (уровень репликации, настройка резервного копирования)
* Тип используемой *СДХ*

*VP* может объединять *SP,* входящие в состав различных *СХД*, расположенных в рамках одного центра обработки данных. Выделение *LUN* на *VP* осуществляется путем выбора первого доступного на данный момент *SP*, удовлетворяющего заданным требованиям.

Примеры различных конфигураций *VP* приведены на рис. 2.

Как и в случае со *SP*, *VP* поддерживает автоматическую агрегацию доступных *SP* в текущем центре обработки данных в логические группы с параметрами, указанными администратором. Ручное перераспределение и ассоциирование *SP* также является доступным для администратора.

Недостатком этой концепции на данный момент является то, что она не поддерживает агрегирование *SP* в *VP* с предъявлением требований к динамическим параметрам производительности. Например, создать *VP*, который гарантирует время отклика не более 3мс. с использованием только лишь готовых программно-аппаратных решений не представляется возможным. Стоит также принять во внимание, что само по себе значение времени отклика в изоляции от нагрузки, планируемой к подаче на *LUN*, не имеет особого смысла, поэтому такой вид консолидации должен также включать описание характеристик типового приложения, планируемого к развертыванию *LUN*, выделенном на таком *VP*. Задача вычисления характеристик отдельно взятого *SP* с уже имеющимися показателями производительности под дополнительной нагрузкой выходит за рамки концепции *VP* и должна решаться внешними компонентами.

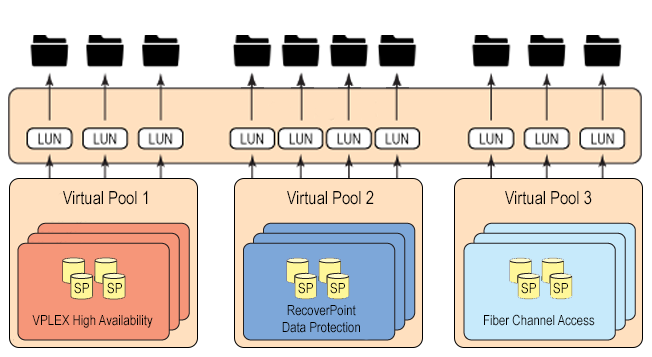


Рисунок 2 – Различные конфигурации *VP*

Целью данной работы является улучшение прототипа интегрированного решения по консолидации ресурсов *СХД* серии EMC VNX на основе требований приложений к времени отклика используемого пространства блокового доступа. Данный прототип был создан в рамках ранее выполненной бакалаврской ВКР и обладал ограниченной функциональностью.

Решение задачи по такой консолидации состоит из следующих ключевых сценариев работы:

1. Определение подходящих *SP* и их объединение в *VP*.
2. Выделение *LUN* на *VP* с проверкой соответствия хотя бы одного *SP* требованиям к производительности.
3. Осуществление активного мониторинга использования *LUN* c целью выявления ненормативного (как чрезмерной утилизации, так и длительного простоя ресурсов в отличии от заданного характера нагрузки).

Разработанный на этапе выполнения бакалаврской ВКР прототип реализовывает только первый из рассмотренных пунктов. В данной ВКР рассматривается улучшение имеющегося прототипа с целью реализации пункта «б» и модифицирование имеющегося программного решения в отказоустойчивую распределённую систему в соответствии с функциональными требованиями к системе.

* 1. Описание интегрируемых продуктов
     1. ViPR Controller

Первый из интегрируемых продуктов – EMC ViPR Controller – предоставляет возможность управлять ресурсами *СХД* компании EMC и сторонних производителей посредством одного унифицированного интерфейса [6].

Именно данный программный продукт реализует концепцию *VP*. Все *СХД* подключаются к данному продукту посредством специального компонента – SMI-S провайдера. Для каждой серии *СХД* этот компонент является уникальным.

С помощью данного продукта можно осуществлять следующие операции над *СХД*:

* Извлекать информацию о *СХД*, входящих в нее *SP* и их статических характеристиках.
* Создавать, удалять и управлять *VP*.
* Создавать, удалять и ассоциировать *LUN* c подключенными сетевыми устройствами.

Взаимодействие с данным продуктом возможно посредством пользовательского веб-интерфейса, а также REST API [7], для которого имеются реализации клиентских программных библиотек [8].

* + 1. ViPR Storage Resource Manager

Вторым интегрируемым продуктом является решение по активному мониторингу и анализу характеристик производительности *СХД* в реальном времени - EMC ViPR SRM [9]. В отличие от ViPR Controller, данному продукту не требуется компонент-посредников для взаимодействия с *СХД* – связь с ними осуществляется напрямую, по бинарному протоколу. Такой подход позволяет получать информацию о динамических показателях производительности, не доступную ViPR Controller.

Взаимодействие с данным продуктом возможно посредством пользовательского веб-интерфейса, а также SOAP & REST API [9], для которого имеются реализации клиентских программных библиотек [8].

* + 1. VNX Sizer

Третьим интегрируемым продуктом, позволяющим вычислять характеристики отдельно взятого *SP* с уже имеющимися показателями производительности под дополнительной, является EMC VNX Sizer.

Изначально данный продукт поставлялся как консольная утилита с пользовательским интерфейсом в виде настольного приложения.

В рамках бакалаврской ВКР для этого компонента была спроектирована и разработана оболочка-программный компонент, которая позволяет использовать данную утилиту посредством REST API. В данной ВКР именно она используется для взаимодействия с другими компонентами системы.

* 1. Выполнение рассматриваемого процесса вручную

Задача по группировке в *VP* подходящих по времени отклика конкретным типам приложений *SP* была автоматизирована с помощью прототипа, созданного в рамках бакалаврского ВКР. Последующее выделение *LUN* на созданном *VP* с проверкой соответствия требуемому времени отклика решается вручную.

На рис. 3 и 4 соответственно изображены диаграммы последовательности, наглядно иллюстрирующие процесс ручного выполнения администратором действий по созданию *VP* и последующему выделению на нем *LUN* заданного размера. Соответствующие данным процессам варианты автоматизации, реализованные в бакалаврской ВКР и предлагаемые к автоматизации в данной ВКР представлены на рис. 5 и 6 соответственно.

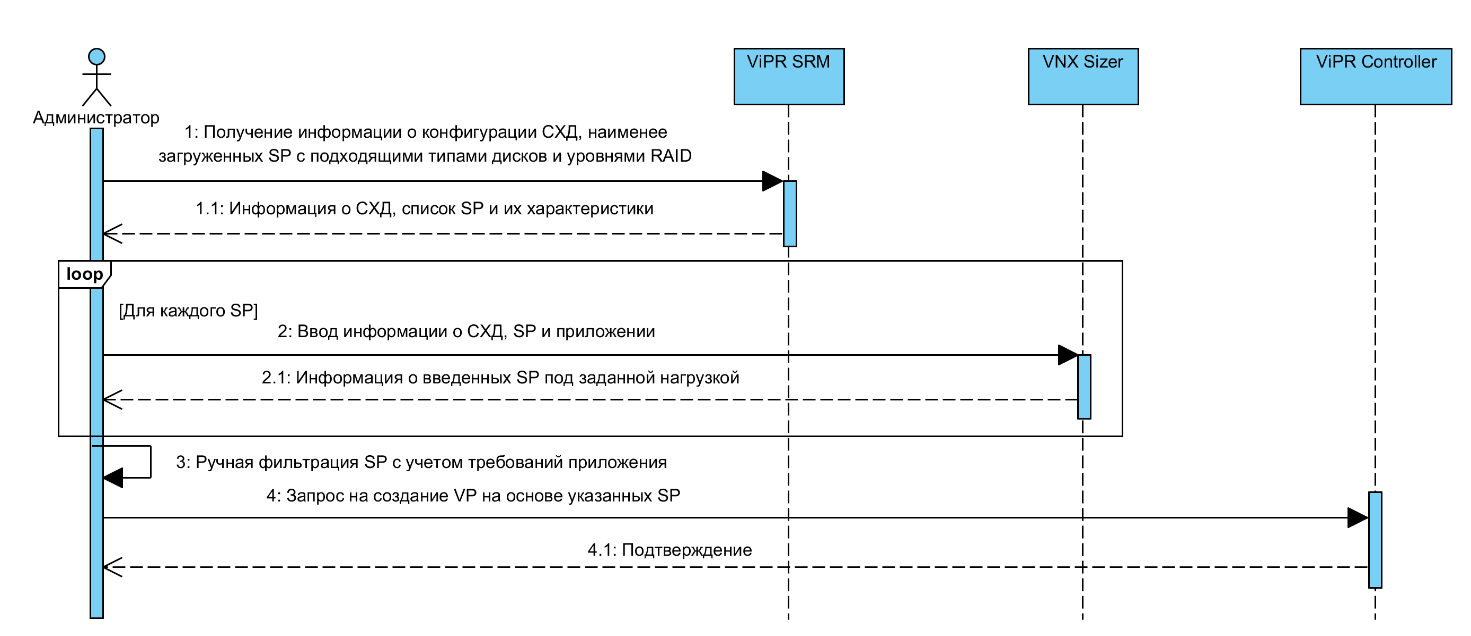


Рисунок 3 – Выполнение процесса по созданию VP вручную в рамках бакалаврской ВКР

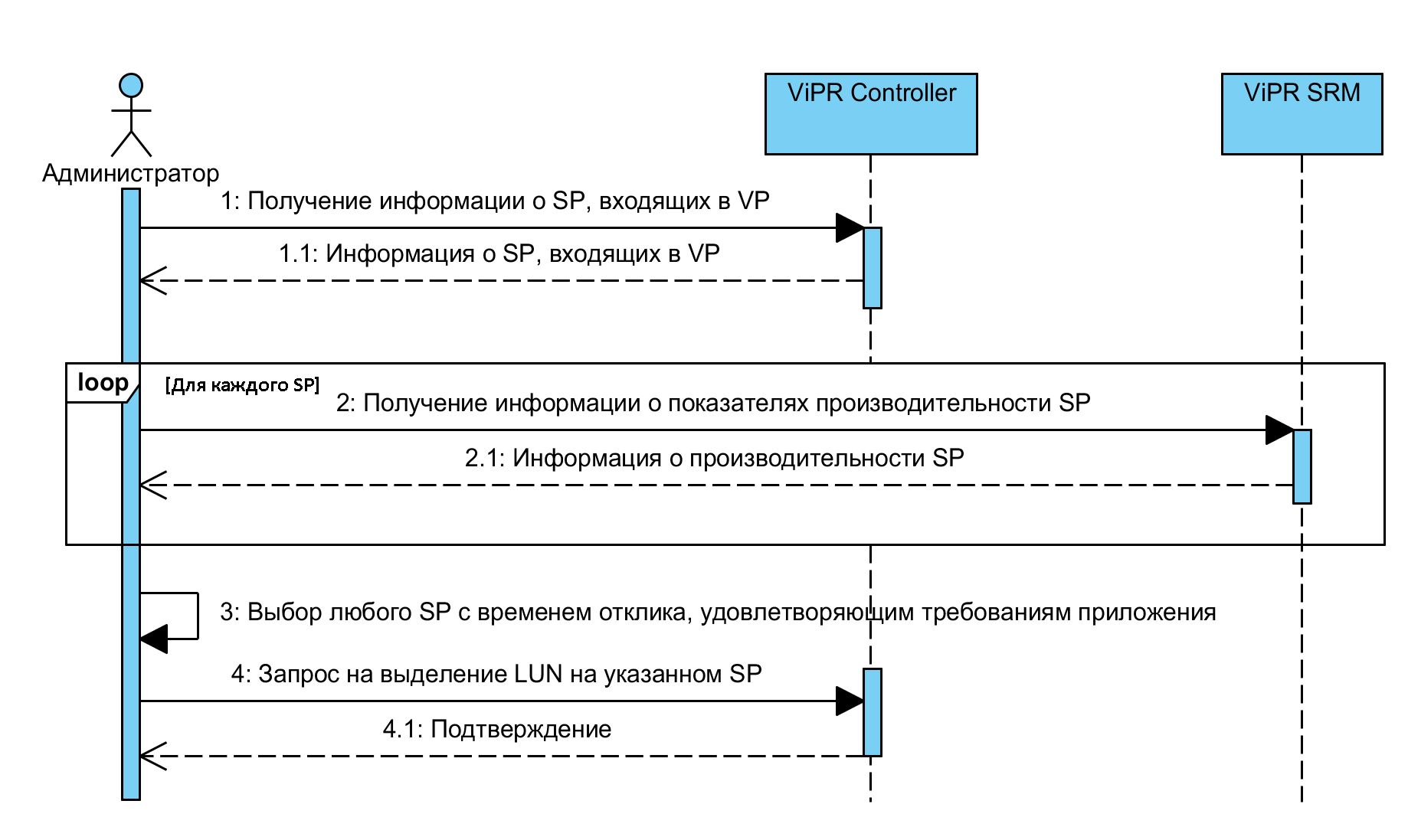


Рисунок 4 – Выполнение процесса по выделению LUN на VP вручную в рамках данной ВКР

* 1. Предлагаемый вариант автоматизации

На рисунках 5 и 6 соответственно продемонстрированы предлагаемые вариант автоматизации процессов, выполненные в рамках бакалаврской ВКР и в текущей работе. Как можно заметить, администратору больше не требуется осуществление каких-либо действий вручную, за исключением ввода исходных данных и выбора соответствующего *VP*, на котором планируется выделение *LUN*.

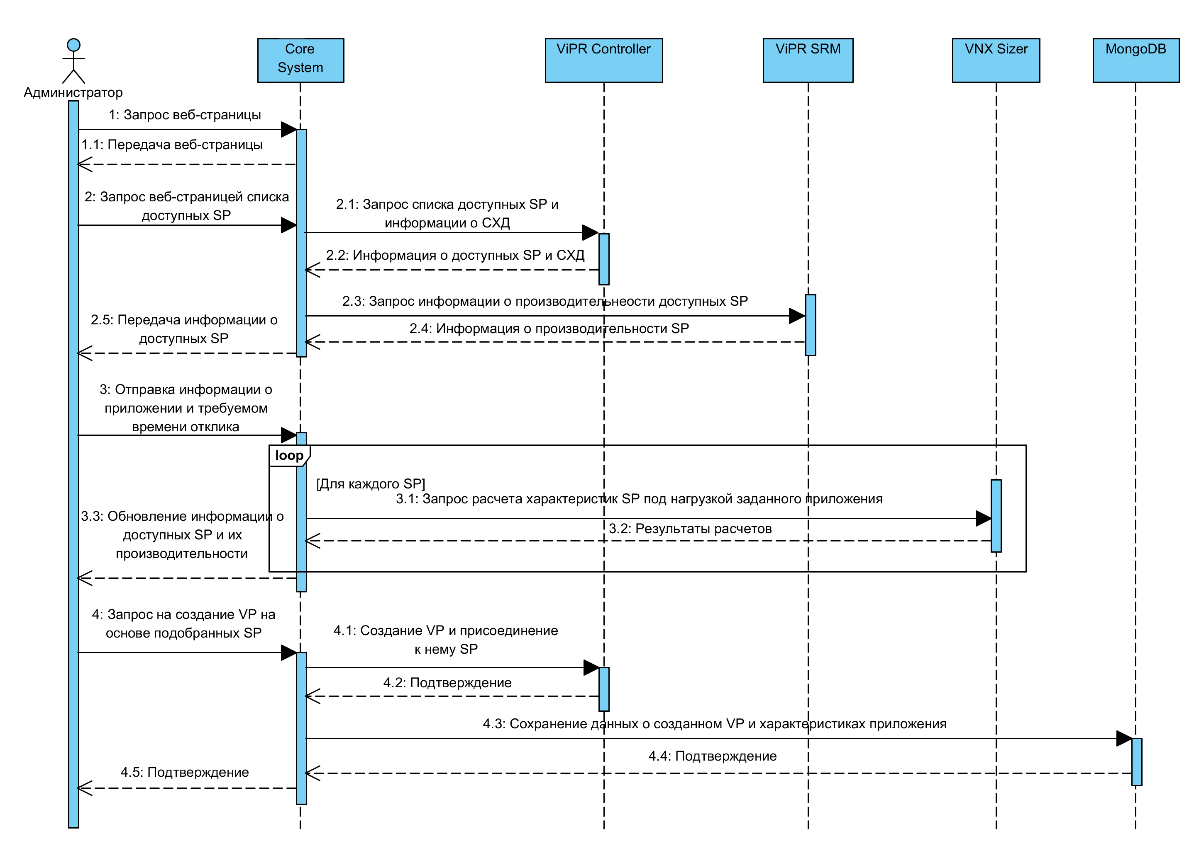


Рисунок 5 – Автоматизированное выполнение процесса по созданию VP

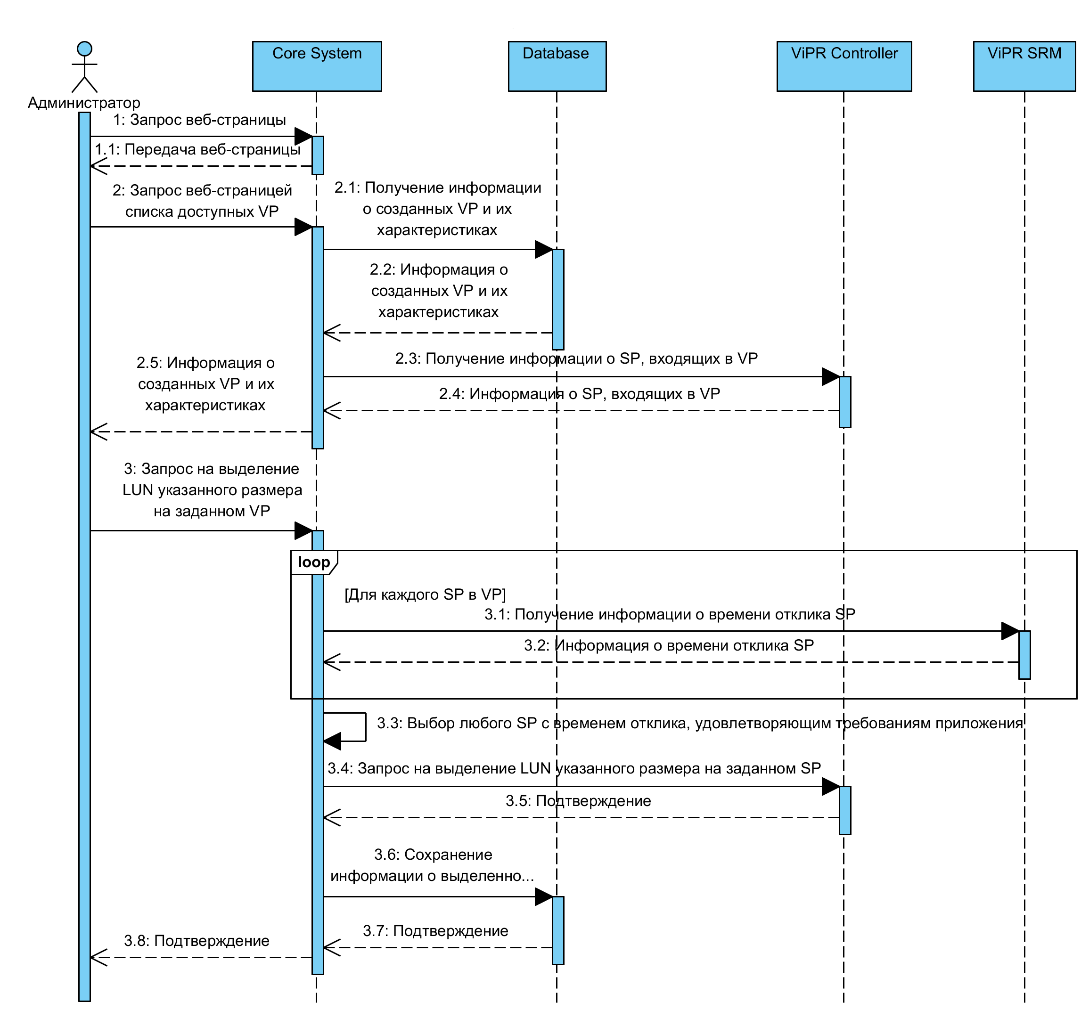


Рисунок 6 – Автоматизированное выполнение процесса по выделению LUN на созданном VP

1. Проектирование
   1. Функциональная архитектура

В рамках бакалаврской ВКР в качестве функционального требования выделялся только один вариант использования, подразумевающий непосредственную консолидацию подходящих под динамические параметры производительности приложения *SP* в *VP* (рис. 7).

В контексте данной ВКР функциональные требования расширяются одним новым вариантом использования (рис. 8) и рядом функциональных требований:

* Поддержка многопользовательского режима
* Поддержка распределенного режима работы

Под многопользовательским режимом подразумевается добавление идентификации и аутентификации в разрабатываемый прототип с возможностью одновременного выполнения различными пользователями операций, предусмотренных основными вариантами использования (рис. 7, 8). В рамках разрабатываемого прототипа авторизацию для всех пользователей можно считать безусловной.

Работа в распределенном режиме подразумевает возможность запуска как одного, так и нескольких копий приложения с помощью изменения конфигурации. При этом копии приложения должны корректно учитывать изменения, вносимые как различными пользователями внутри одной копии, так и между различными копиями. Система должна быть спроектирована с учетом высокой надежности взаимодействующих узлов и низкой надежности соединяющих их каналов связи. Интегрируемые компоненты ViPR Controller и ViPR SRM поддерживают распределенный режим работы. Компонент VNX Sizer является легковесным REST-сервисом без сохранения состояния, что позволяет осуществить его развертывание в каждом центре обработки данных независимо. Подразумевается, что в каждом центре обработки данных, в котором разворачивается копия системы, данные внешние системы присутствуют. Во время «изоляции» центра обработки данных при выходе из строя каналов связи копия системы должна поддерживать управление ресурсами *СХД* в локальном центре обработки данных. При восстановлении связи между центрами обработки данных копии приложения должны автоматически обменяться информацией об независимых изменениях, внесенных администраторами во время изоляции центра обработки данных.

Основные варианты использования предусматривают одновременную работу до 10 пользователей (администраторов центров обработки данных). Требование к работе в режиме кластера обусловлено распределенной архитектурой интегрируемых компонентов ViPR Controller и ViPR SRM. Данные продукты позволяют объединять в единую информационную систему различные центры обработки данных в разных частях света. В критических ситуациях отдельный центр обработки данных может оказаться изолирован от остальных, но при этом управление ресурсами *СХД* внутри него все еще является актуальной задачей.

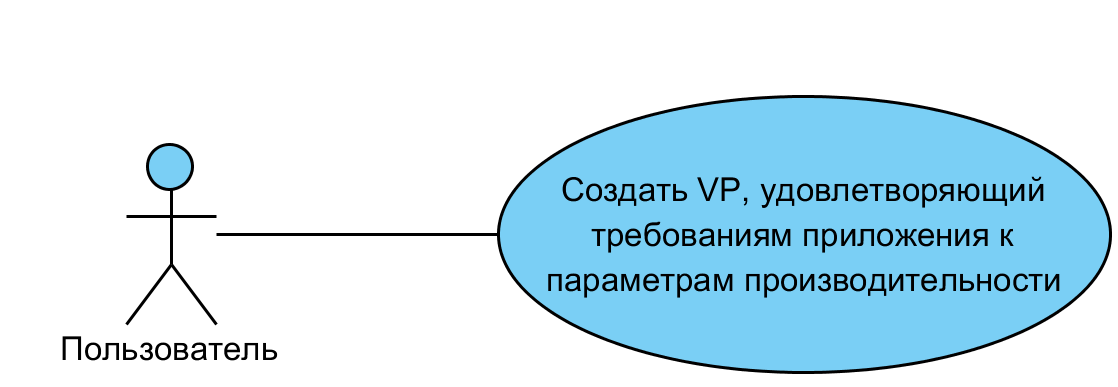


Рисунок 7 – Вариант использования главного компонента системы в бакалаврской ВКР

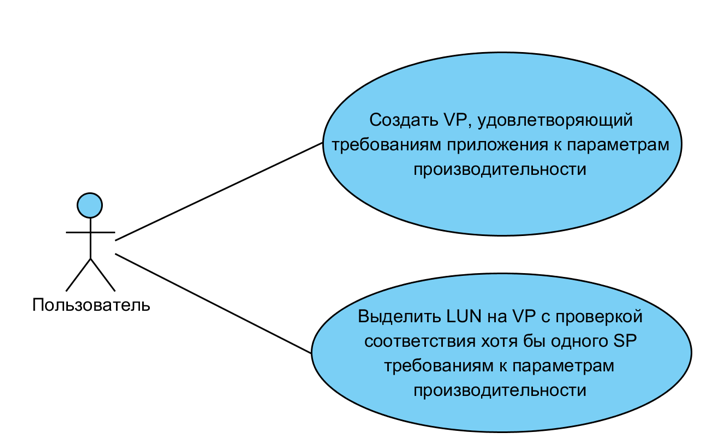


Рисунок 8 – Вариант использования главного компонента системы в данной ВКР

Сценарии, соответствующие вариантам использования по созданию *VP* и последующего выделения на нем *LUN* отображены на рис. 9 и 10 соответственно.

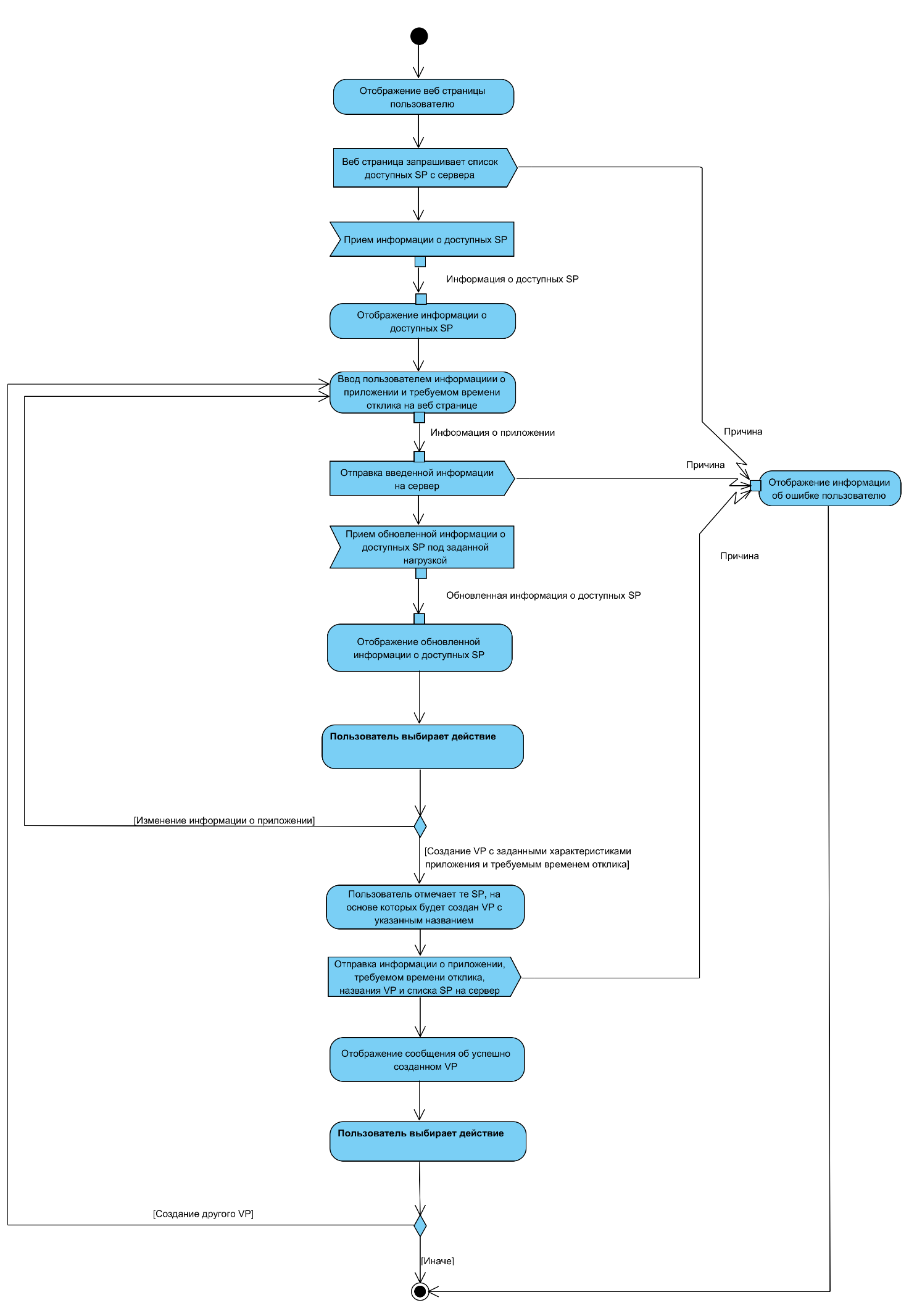


Рисунок 9 – Диаграмма деятельности по созданию *VP*

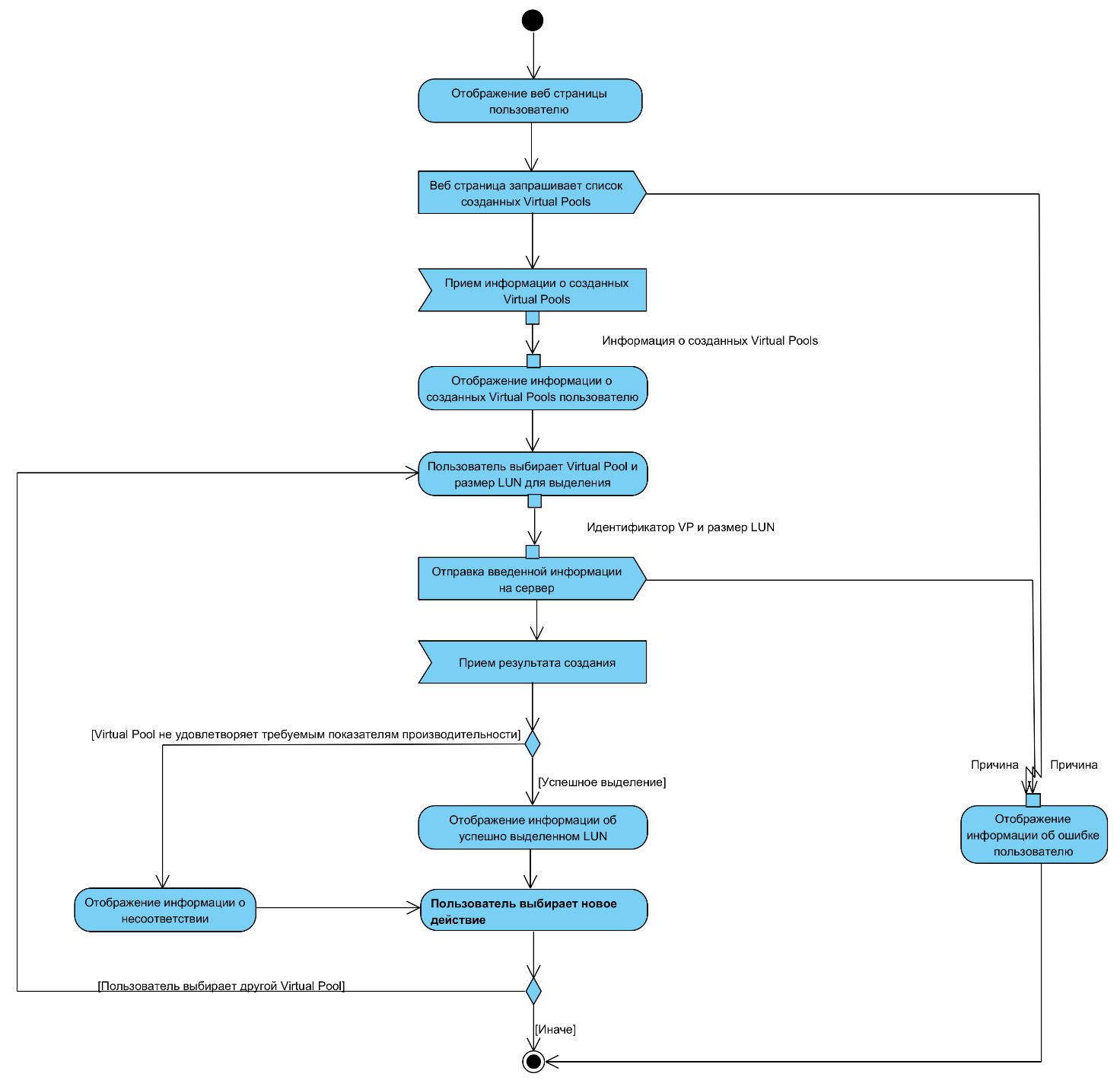


Рисунок 10 – Диаграмма деятельности по выделению *LUN* на ранее создавнном *VP*

* 1. Проблемы, исследуемые в рамках ВКР

Преобразование однопользовательского прототипа в многопользовательскую распределенную систему подразумевает не только добавление новой функциональности к имеющемуся прототипу, но и дополнительный анализ и переработку имеющейся архитектуры как на системном, так и на программном уровне, а также уровне данных. Предметом исследования в данной ВКР является выявление наилучшего способа организации многопользовательской работы с логическими ресурсами в распределенной системе с учетом используемого стека технологий и системной архитектуры.

В источнике [13] большое внимание уделяется разнице между доступностью и надежностью системы. Под доступностью подразумевается «способность системы к выполнению операций по требованию», а под надежностью «способность системы безошибочно выполнять операции, для которых она предназначена» [13, с.5]. Разрабатываемая система должна соответствовать как требованию к высокой доступности, так и надежности.

Данный источник [13, с.12] выделяет базовые принципы, которые необходимо соблюдать для увеличения доступности системы:

* Мониторинг копий приложений: периодическая проверка состояния и производительности копий приложения с целью предположения скорого выхода из строя одной или нескольких из них.
* Отслеживание изменений в конфигурации: своевременное обнаружение и принятие соответствующих мер при выходе копии приложения из строя или восстановления работоспособности ранее вышедшей из строя копии.

Источник [14, с. 78] также рекомендует исключить единую точку отказа при организации коммуникаций между копиями системы, за счет настройки избыточности сетевой конфигурации [14, с. 214-225]. Различные конфигурации такой избыточности приведены на рис. 11 и 12 [14, c. 219, 223]. Вариант с подключением нескольких сетевых интерфейсов сервера к одной сети является предпочтительным для высоконагруженных систем с большим потоком обмена данными, а подключение к нескольким различным сетям для систем с требованиями к высокой отказоустойчивости [14, с. 227]. Так как разрабатываемая система относится к классу отказоустойчивых систем, то использование второго варианта будет более предпочтительным.

На рис. 13 изображена схема, иллюстрирующая простую архитектуру организации взаимодействия двух серверов с клиентами [14, с. 370]. Она подразумевает исключение единой точки отказа за счет дублирования каждого компонента системы – сервера, используемых ими хранилищ данных и сетевых интерфейсов.

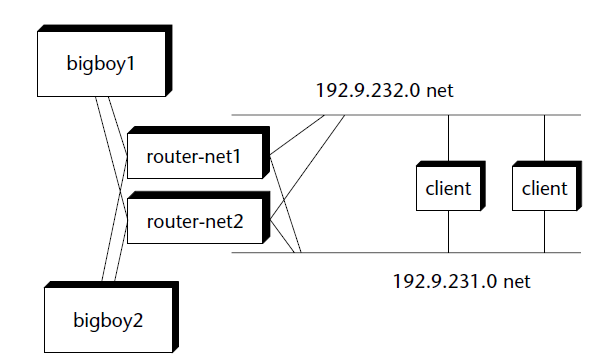


Рисунок 11 – Избыточность сети за счет подключения каждого сервера к нескольким сетям

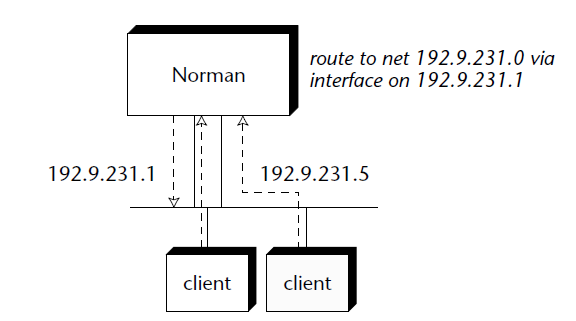


Рисунок 12 – Избыточность сети за счет подключения сервера к сети посредством нескольких интерфейсов сразу

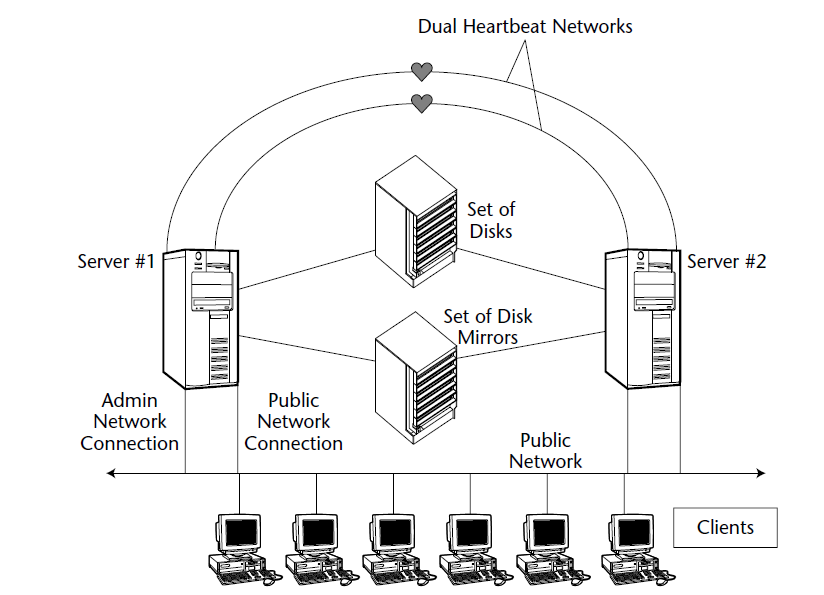


Рисунок 13 – Простая архитектура отказоустойчивого кластера из двух серверов

В классических трудах по проектированию распределенных информационных систем [12] описываются различные подходы, которые как по-отдельности, так и в совокупности могут помочь в решении поставленной задачи.

Используемая в имеющейся системе база данных MongoDB поддерживает работу в распределенном режиме за счет репликации (рис. 14, 15) [16, с.80-81] и шардирования (рис. 16) [17, c. 34]. Минусом данного подхода является использование архитектуры «мастер-реплика» (рис. 14) отсутствие поддержки архитектуры «мастер-мастер», что затрудняет использование данной СУБД без дополнительных средств обеспечения целостности данных при потере соединения между центрами обработки данных. Также это усложняет сетевое взаимодействие при большом числе активных копий системы, так как все коммуникации будут производиться через один узел.

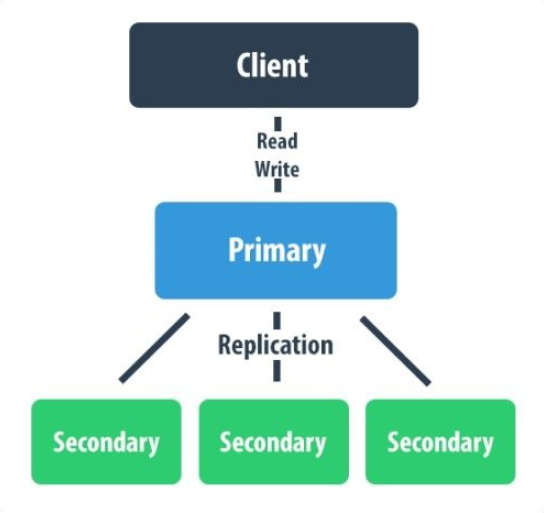


Рисунок 14 – Схема работы MongoDB при использовании репликации

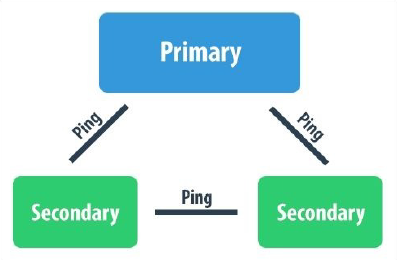


Рисунок 15 – Схема работы мониторинга состояния узлов MongoDB при использовании репликации

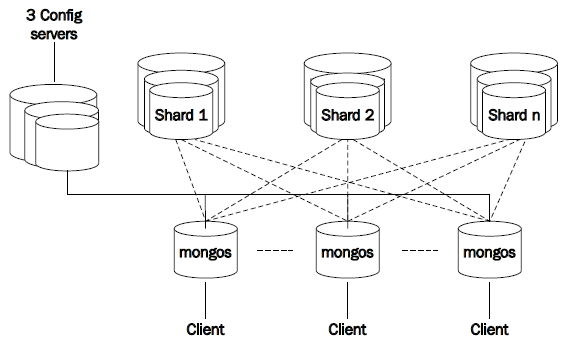


Рисунок 16 – Схема работы MongoDB при использовании шардирования

В трудах, посвященных лучшим практикам настройки и развертывания данной СУБД как в одиночном режиме, так и в центрах обработки данных (рис. 17) [18], подробно описываются недостатки использования данной СУБД при необходимости организации транзакционной логики приложения ввиду отсутствия встроенной поддержки транзакций и сложностью синхронизации изменений при частом обрыве соединения и переизбранием нового мастер-узла [18, 323-329; 17, с. 17-37].

Однако, использование данной СУБД имеет большие преимущества в случае независимого развертывания копии данной СУБД в каждом центре обработки данных без организации репликации / шардирования между ними при наличии внешней синхронизации и поддержки консистентности данных. Данный способ также позволит избежать проектирования ER модели и перепроектирования логики работы с БД для использования SQL СУБД с поддержкой вышеописанной функциональности на уровне БД.

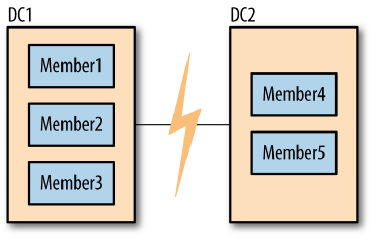


Рисунок 17 – Иллюстрация кластера узлов MongoDB в разных центрах обработки данных

СУБД Oracle предоставляет очень гибкий подход к архитектуре взаимодействия узлов СУБД. Она также поддерживает репликацию и шардирование, имеет поддержку репликации «мастер-мастер» [15, с. 9-1 – 9-3; 9-16 – 9-17], «мастер-реплика» [15, с. 9-4 – 9-15], а также настраиваемый механизм синхронизации данных после восстановления связи между узлами [15, 13-1 – 13-11]. Данная СУБД предоставляет богатые возможности по обеспечению консистентности данных, а также поддерживает транзакции. Однако, настройка каждого узла будет являться очень трудоемкой задачей, требующей высокой квалификации в области данной СУБД. Также, конфигурация данной СУБД будет сильно зависеть от размера кластера, что усложняет масштабирование.

Для решения проблемы синхронизации данных между копиями приложения можно воспользоваться модификацией подходов, описанных в источнике [12].

Современные системы кэширования / хранимые в оперативной памяти БД имеют богатый набор функциональности по работе в распределенном режиме, эффективной синхронизации данных при репликации и автоматической балансировке потока данных при использовании сетевых топологий, изображенных на рис. 11 и 12. Так как узлы проектируемой системы обладают высокой надежностью, данный подход имеет преимущество над подходами, использующими «классические», хранящие данные на файловой системе, СУБД, за счет легкости масштабирования и скорости работы. Сложностью при применении данного подхода является синхронизация содержимого кэша с используемой СУБД, т.к. несмотря на высокую надежность самих узлов, вероятность их выхода как самой копии приложения, так и используемой СУБД все равно присутствует.

На рис. 18 [12, с. 13] изображена классическая схема применения глобального кэша для увеличения доступа к наиболее часто используемым данным.

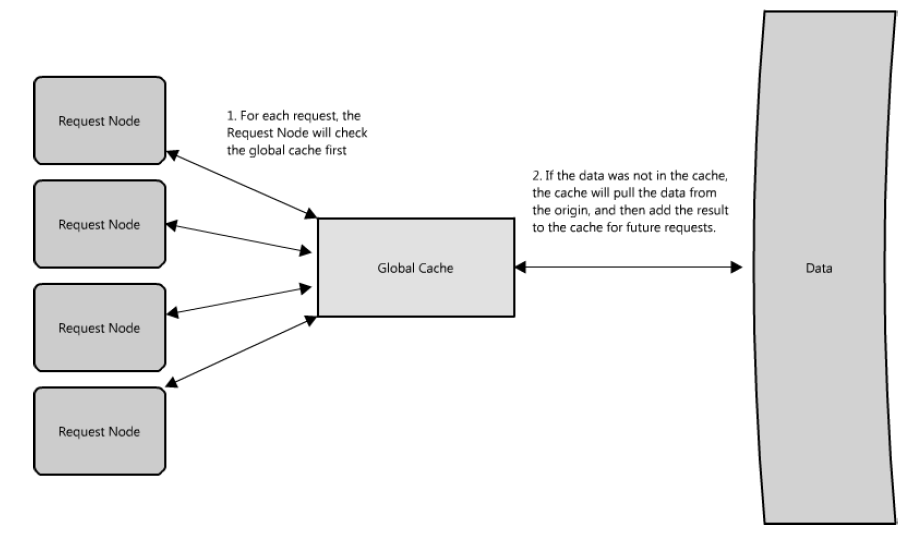


Рисунок 18 – Использование глобального кеша как хранимой в опреативной памяти базы данных

Если использовать хранимую в оперативной памяти базу данных подобно данному кэшу, но не только для чтения данных, но и для их сохранения, мы получаем унифицированный интерфейс работы с данными, сохраняющий все преимущества классических СУБД и нивелирующий их недостатки за счет поддержки большинством современных хранимых в оперативной памяти баз данных исключительных и распределенных блокировок, а также поддержкой некоторых из них автоматического сохранения данных в подключенную СУБД.

Использование выделенного глобального узла для такой БД является нарушением точки единого отказа, поэтому целесообразно использовать распределенные узлы БД, по узлу для каждой копии приложения в центре обработки данных, что соответствует подходу, изображенному на рис. 19 [12, с. 14]. Так как для работы каждого узла системы в каждый момент времени не требуется информация со всех узлов сразу, то данную БД можно использовать в роли кэша – на постоянной основе хранить только данные, непосредственно связанные с локальным узлом, а данные с других узлов, необходимые для расчета, динамически загружать с других узлов и кэшировать. При изоляции одного или нескольких узлов от других необходимость в данных, хранящихся на них, до момента восстановления соединения, отсутствует. Данный фактор позволяет исключить проблему полного дублирования информации в кластере. Такой подход позволит соблюсти принципы отказоустойчивых систем, изложенные в источнике [14].

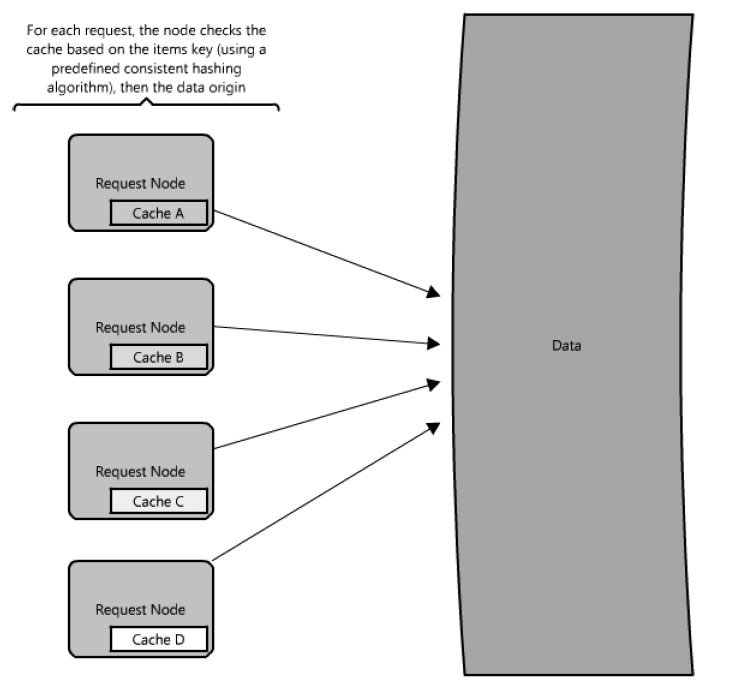


Рисунок 19 – Использование распределенного кеша как хранимой в оперативной памяти базы данных

* 1. Системная архитектура

Рис. 20 и 21 демонстрируют системную архитектуру прототипа в рамках выполнения бакалаврской ВКР и данной ВКР соответственно. Салатовым цветом на них выделены те компоненты, которые были разработаны во время соответствующего этапа работы по разработке данной системы. Как можно заметить, утилита VXN Sizer и ее сетевая оболочка, разработанные в рамках бакалаврской ВКР, в рамках данной ВКР уже становятся внешними системами по отношению к модифицируемым и разрабатываемым компонентам. На рис. 22 представлена архитектура отдельно взятого узла системы.

Для удобочитаемости диаграмм были применены следующие наименования структурных элементов:

* EMC VNX Storage System –*СХД* серии EMC VNX.
* Binary Protocol / Binary VNX Protocol – бинарный протокол, специфичный для *СХД*, позволяющий осуществлять работу как с конфигурацией *СХД*, так и с расположенными на ней данными.
* SMI-S Provider for VNX – специфичный для *СХД* рассматриваемой серии программный модуль, абстрагирующий выполнение высокоуровневых операций по низкоуровневому протоколу.
* SMI-S API – программный интерфейс модуля SMI-S Provider for VNX.
* Storage Processor Unit – один и аппаратных модулей *СХД* со специфичным для каждой серии *СДХ* ПО, предоставляющий бинарный протокол взаимодействия с *СДХ*.
* ViPR SRM SOAP API / SOAP API – программный интерфейс взаимодействия продукта ViPR SRM.
* ViPR REST API / REST API – программный интерфейс взаимодействия продукта ViPR Controller.
* VNX Sizer Engine – копия продукта VNX Sizer.
* VNX Sizer WS API Wrapper – компонент, реализующий сетевой программный интерфейс взаимодействия для утилиты VNX Sizer.
* VNX Sizer WS API / VNX Sizer REST API – сетевой программный интерфейс взаимодействия для утилиты VNX Sizer.
* Core System / Core System Node – копия главного компонент системы.
* Core System WS API / Core System REST API – сетевой программный интерфейс взаимодействия для главного компонента системы.

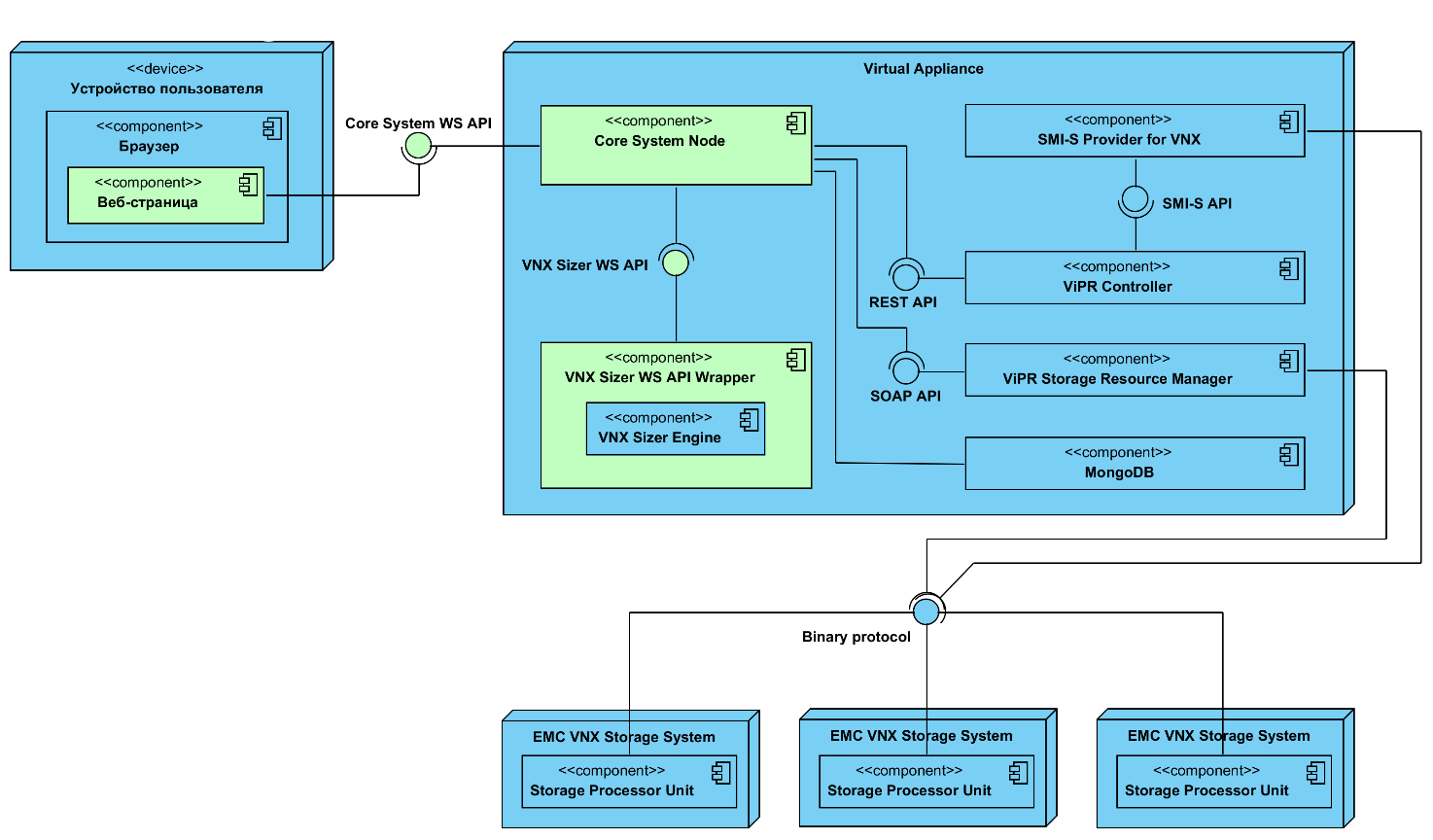


Рисунок 20 – Диаграмма развертывания системы в бакалаврской ВКР

В соответствии с требованиями к высокой доступности разрабатываемого приложения в условиях высокой отказоустойчивости узлов и низкой отказоустойчивости каналов связи, с помощью агрегации имеющихся подходов, рассмотренных в главе 2.2, было принято решение по переходу от монолитной архитектуры реализованного в рамках ВКР решения к распределенной архитектуре с горизонтальным масштабированием и избыточному подходу к организации каналов связи между отдельными узлами приложения.

Наличие функциональных требований по организации автоматической синхронизации обновлений, произведенных различными узлами приложения после восстановления каналов связи, а также организации блокировок совместно используемых администраторами ресурсов между различными узлами приложения, было принято решение использовать технологию, которая позволяет внедрить данную функциональность без применения серьезных и сложных подходов по ее реализации вручную. Для этого был выбран высокопроизводительная база данных с типом хранения ключ-значение с открытым исходным кодом Hazelcast [19]. Её использование оправдывается максимальной степенью интеграции с уже используемым стеком технологий – она написана на языке высокого уровня Java и предоставляет такие высокоуровневые примитивы синхронизации, как распределенный семафор, распределенные блокировки, распределенные коллекции с конкурентным и блокирующим доступом. Также, ее базовая функциональность позволяет организовывать работу в больших и сложных сетевых инфраструктурах, осуществлять прослушивание и синхронизацию с другими узлами с использованием нескольких сетевых интерфейсов как одной, так и разных сетей. В дополнение, он поддерживает гибкий механизм организации как полных реплик, так и традиционное партиционирование и шардирование. Дополнительной отличительной особенностью, которая также хорошо покрывает имеющиеся функциональные требования, является тонкая настройка политик автоматизированной синхронизации данных между узлами после восстановления связи в результате сбоев сети.

Рисунок 21 демонстрирует структуру компонентов системы. Для организации отказоустойчивого сетевого доступа используется подход с типом подключения между узлами системы «полный граф». С учетом отсутствия требования по большой масштабируемости разрабатываемого решения и ограниченном формальном списке функциональных требований данный подход является приемлемым решением.

Для организации наиболее прозрачного горизонтального масштабирования и простоты развертывания системы в той же виртуальной инфраструктуре принято решение по изоляции каждой копии узла приложения в отдельную виртуальную машину. Подход по использованию больших виртуальных контейнеров, содержащих до нескольких десятков виртуальных машин, объединенных одной или несколькими сетями, является очень распространенным в компании EMC – он упрощает процесс развертывания, быстрого масштабирования и замены вышедших из строя узлов системы на новые. При этом все накладные расходы по организации высокой доступности самих узлом и их коммуникации между различными центрами обработки данных, как и функционированию виртуальных машин в условиях изоляции центра обработки данных при аварии, берет на себя непосредственно комплекс программно-аппаратных средств, предоставляющий сервис данной виртуальной инфраструктуры.

Так как разрабатываемый продукт предполагается также к демонстрации ключевых особенностей заказчикам, для удобства развертывания и демонстрации была сконфигурирована схема развертывания системы из 3 узлов в инфраструктуре Docker (см. Приложение А).

Так как каждый узел приложения должен предоставлять полную функциональность даже в отсутствие других активных узлов, то вполне логичным является то, что внутри каждой виртуальной машины должны быть развернуты все базовые компоненты системы –VNX Sizer Engine, главный компонент системы, ViPR Controller, ViPR SRM. Также каждый узел должен обладать независимой СУБД для долговременного сохранения всех внесенных изменений, а также копию СУБД Hazelcast для прозрачной интеграции и взаимодействия с остальными копиями приложения.

Для упрощения работы с компонентом ViPR Controller в распределенном режиме было принято решение не объединять все его копии в единый логический кластер средствами продукта ViPR Controller, так как это может привести к проблемам когерентности информации, видимой различными узлами приложения после сбоев в сети, с информацией, доступной в СУБД Hazelcast – в конечной счете, по истечению достаточно большого периода времени после сбоя оба информационных пространства, как ViPR Controller, так и Hazelcast, придут в состояние событийной консистентности, но организация механизмов отслеживания подобных сбоев и корректная обработка ошибок в данной ситуации не является целесообразным, ведь основным вариантом использования как продута ViPR Controller, так и продукта ViPR SRM в рамках локальной виртуальной машины является управление теми *СХД*, которые подключены именно к локальным копиям рассматриваемых продуктов. Получение информации и управление *СХД*, подключенными к другим узлам системы можно осуществлять посредством прямых обращений к копиям соответствующих продуктов, развернутых на удаленных узлах приложения. Такой подход позволяет не только снизить накладные расходы сети на излишнюю синхронизацию между копиями данных продуктов при работе в распределенном режиме, но и упростить работу с совместно используемым ресурсами, ведь в таком случае появляется дополнительный контракт, в соответствии с которым каждый совместно используемый ресурс принадлежит одному и только одному узлу, а работу по организации конкурентного доступа к нему возьмет на себя СУБД Hazelcast. По этой же причине нет смысла организовывать сложную распределенную архитектуру СУБД, отвечающий за постоянное хранение данных в системе, т.к. каждый узел, с учетом всех функциональных требований, должен хранить только информацию об изменениях, выполненных над совместно используемыми ресурсами, находящимися в его непосредственном управлении.

Предлагаемая архитектура относится к классу архитектур «узел-узел», в которой каждая копия является равноправной, без выделенного узла, осуществляющего координацию обычных узлов между собой.

Таким образом, каждый узел предоставляет наружу следующий ряд интерфейсов:

* Core System REST API – для выполнения высокоуровневых операций через веб-интерфейс, доступен для пользователя снаружи виртуального контейнера.
* Hazelcast Binary API – низкоуровневый интерфейс меж узлового взаимодействия Hazelcast, недоступен снаружи виртуального контейнера

Остальные компоненты системы не предоставляют свои программные интерфейсы для других узлов системы, так как взаимодействие с ними осуществляется только главным разрабатываемым компонентом на соответствующем узле системы.

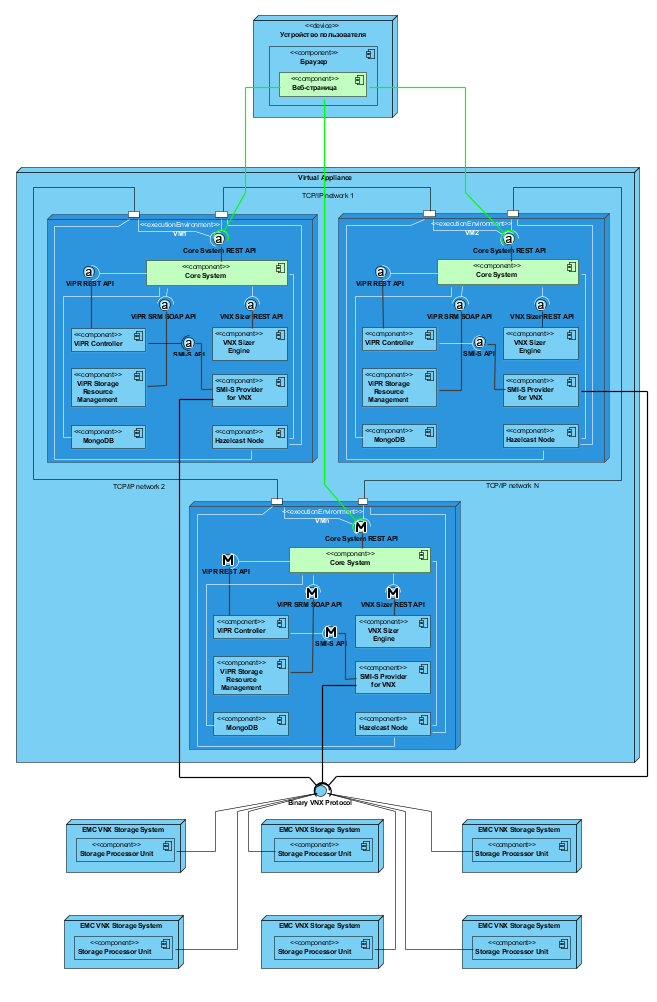


Рисунок 21 – Диаграмма развертывания системы в данной ВКР

Таким образом, каждый узел системы сможет выступать в роли точки входа кластер для выполнения основных операций в рассмотренных в главе 2.1 рамках вариантов использования. Получение информации о совместно используемых ресурсах и их показателях производительности будет осуществляться посредством вызовов API главного компонента системы на удаленных узлах.

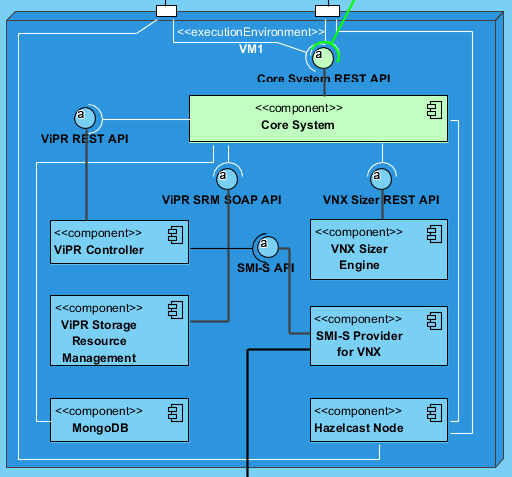


Рисунок 22 – Диаграмма развертывания отдельно взятого узла системы в данной ВКР

* 1. Программная архитектура

Диаграммы 23 и 24 соответственно отображают последовательность операций последовательности успешного сценария открытия страницы пользователем.

Ввиду однопользовательского режима работы системы в одном узле в бакалаврской ВКР было допустимо сохранения информации о *SP* на уровне пользовательской HTTP сессии. В данной ВКР для получения консистентных данных о доступных *SP* к управлению необходимо получение информации о доступных узлах системы и извлечение данной информации заново при каждой операции и при каждом обновлении страницы или списка *SP* пользователем. Для каждого удаленного узла системы осуществляется удаленный вызов по получению списка *SP* посредством соответствующего HTTP вызова. Полученные результаты объединяются с результатами информации, полученной от локальных компонент системы, и передаются пользователю.

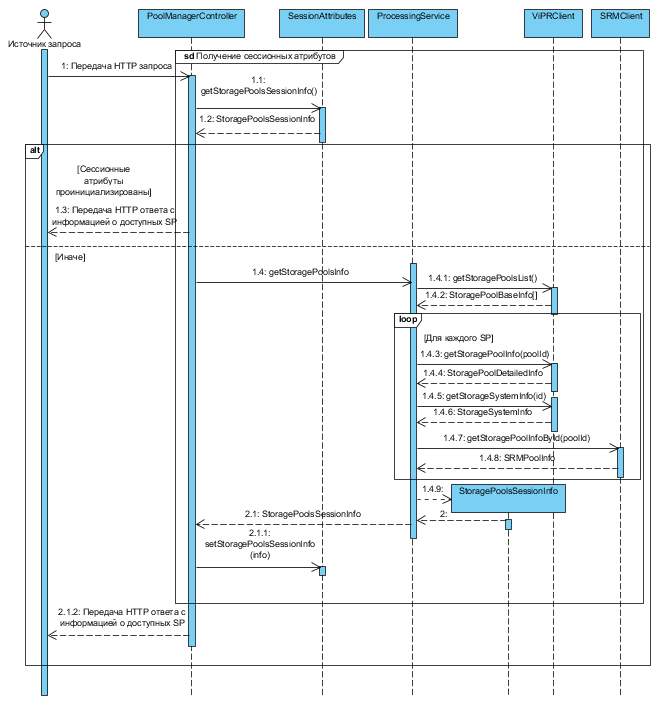


Рисунок 23 – Диаграмма последовательности успешного сценария открытия страницы пользователем в бакалаврской ВКР

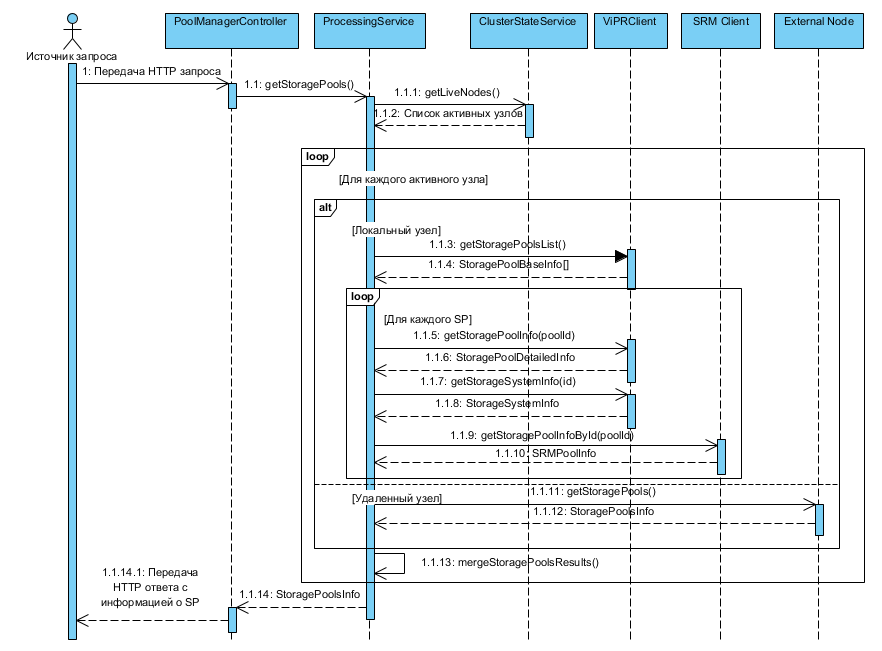


Рисунок 24 – Диаграмма последовательности успешного сценария открытия страницы пользователем в данной ВКР

Изменения в последовательности исполняемых операций, необходимых для осуществления расчетов нагрузки выбранных *SP* под заданной нагрузкой, проиллюстрированы на диаграммах 25 и 26. Так как компонент VNX Sizer является контекстно-независимым и позволяет осуществлять расчеты характеристик любых *SP* независимо от ЦОД, то расчеты характеристик *SP* как локального, так и удаленных узлов эффективнее всего осуществлять с использованием локального компонента VNX Sizer, а не осуществлять вызовы компонент удаленных узлов.

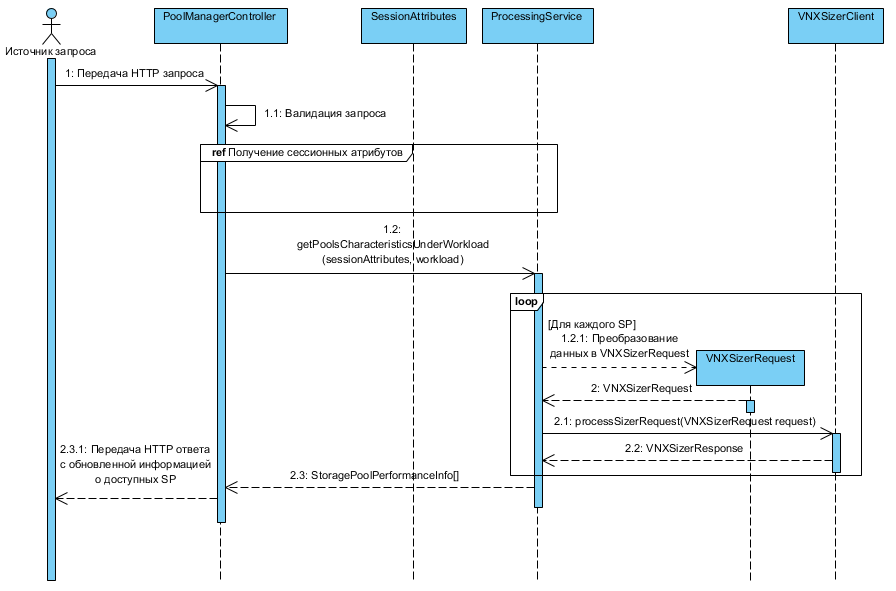


Рисунок 25 – Диаграмма последовательности успешного сценария по обработке запроса на расчет характеристик *SP* под заданной нагрузкой в бакалаврской ВКР

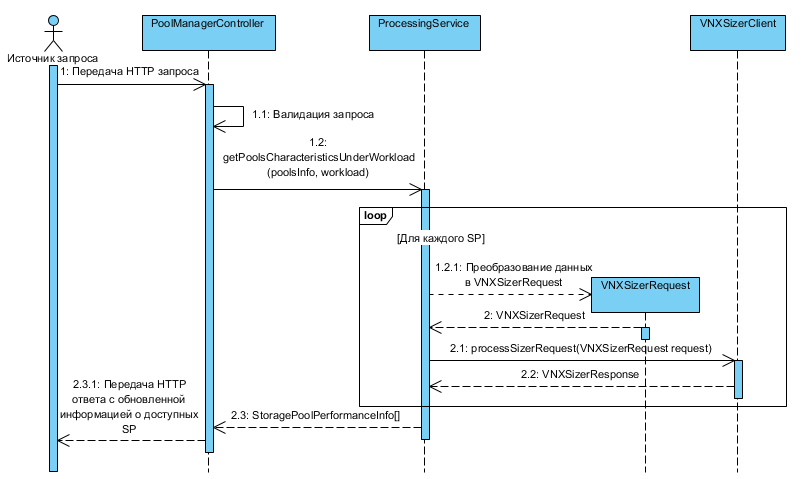


Рисунок 26 – Диаграмма последовательности успешного сценария по обработке запроса на расчет характеристик *SP* под заданной нагрузкой в данной ВКР

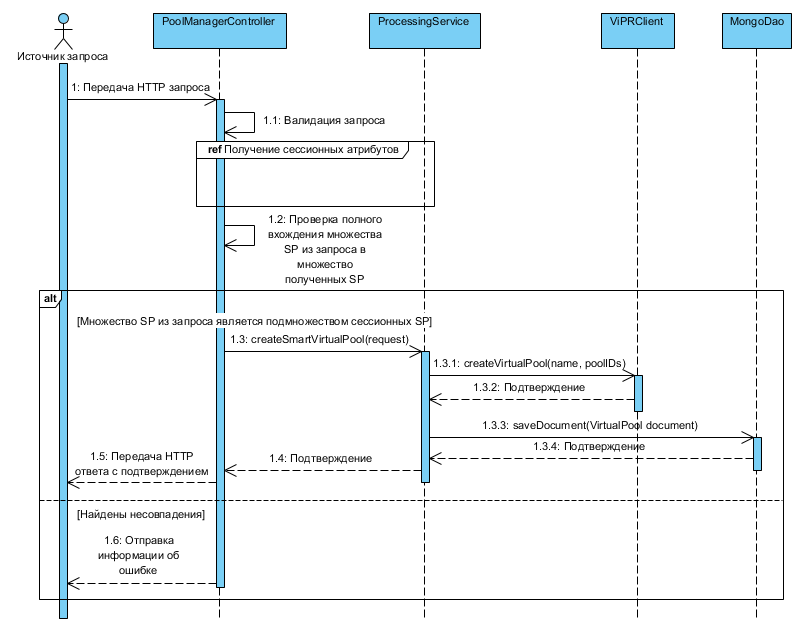


Рисунок 27 – Диаграмма последовательности успешного сценария по созданию *VP* в бакалаврской ВКР

На диаграммах 28 и 29 наглядно отражен процесс создания *VP* в бакалаврской и данной ВКР, соответственно. Учитывая ограничениеViPR Controller, которое позволяет создавать *VP* только из *SP*, находящихся в одном ЦОД (доступных одновременно только одному узлу), то задачу по созданию такого *VP* делегировать именно тому узлу, к которому относятся все выбранные пользователем *SP*. Также это требует дополнительной валидации запроса пользователя, проверяя, что все выбранные *SP* относятся к одному узлу системы.

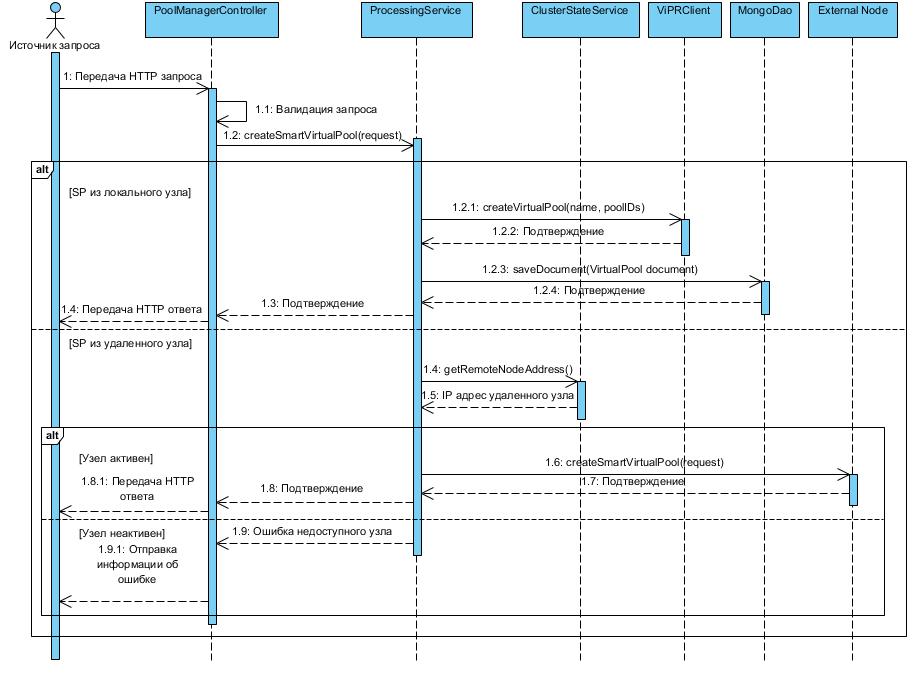


Рисунок 28 – Диаграмма последовательности успешного сценария по созданию *VP* в данной ВКР

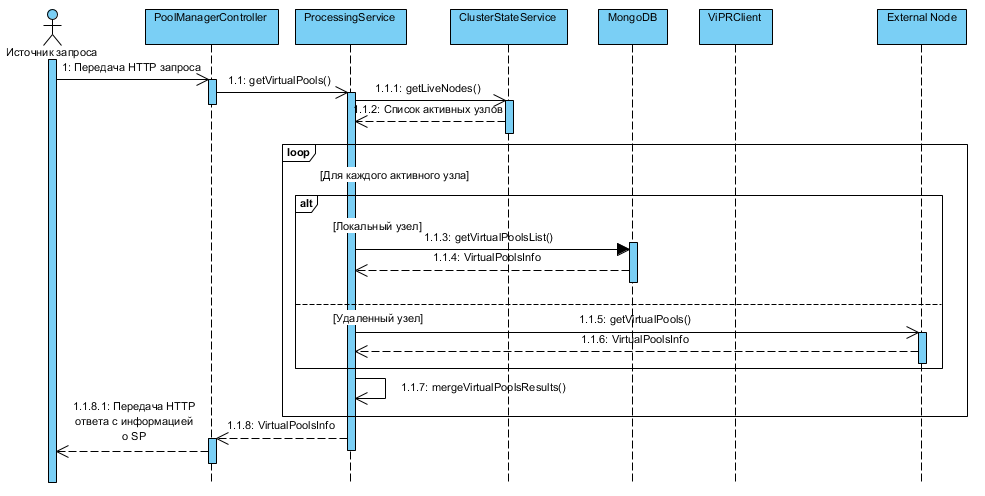


Рисунок 29 – Диаграмма последовательности успешного сценария по получению списка *VP*

Новый вариант использования, реализованный в рамках данной ВКР, изображен на диаграмме 30. Выделение *LUN* так же, как и в случае с предыдущим рассмотренным процессом, является операцией, связанной с отдельно взятым узлом системы. Вследствие этого осуществляется проверка доступности соответствующего узла системы, получение его адреса и выполнение вызова удаленного компонента.

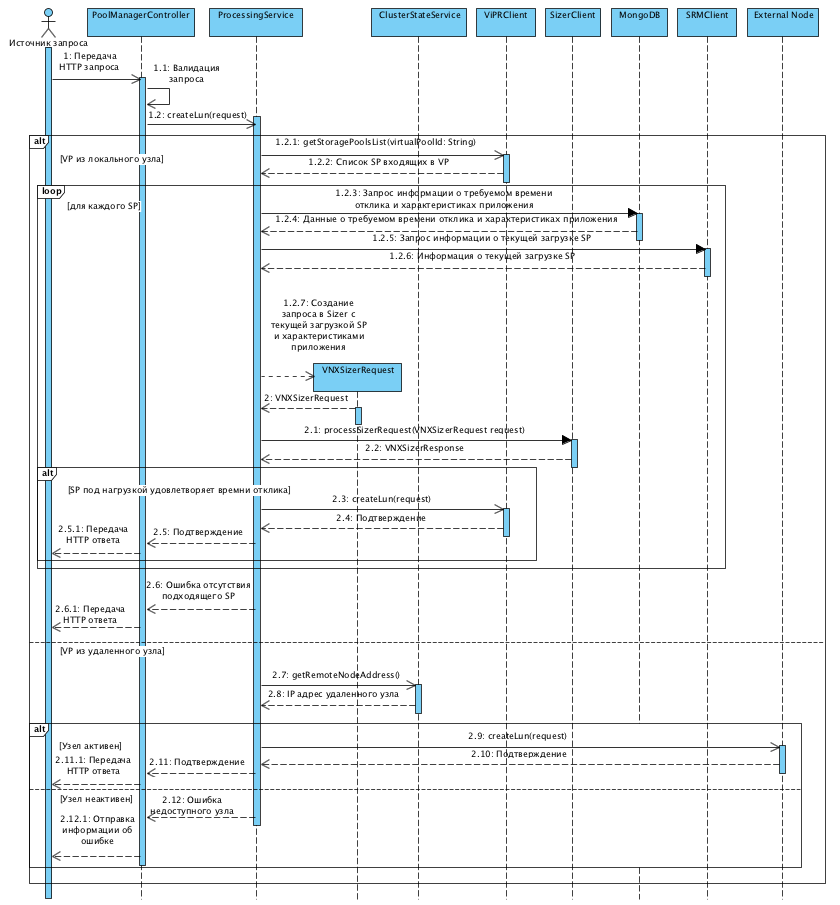


Рисунок 30 – Диаграмма последовательности успешного сценария по выделению *LUN* на *VP*

В качестве языка программирования для разработки системы был использован язык Java. Данное решение, наряду с выбором основного фреймворка Spring, обусловлено тем, что данный базовый стек технологий используется в качестве основного в продукте ViPR Controller, легкость интеграции в который является одним из важных требований к разрабатываемой системе. Также на этот выбор повлияло то, что разработанная в рамках бакалаврской ВКР система основана на том же стеке технологий, что в значительной степени снижает трудозатраты и увеличивает эффективность модификации данной системы в рамках текущей ВКР.

Использование принципа слабой связности, на котором построен фреймворк Spring, позволяет программировать гибкие решения, которые легко тестируются и позволяют заменять реализации компонент при использовании выделенных интерфейсов их взаимодействия друг с другом. Данный аспект позволил эффективно реализовать систему каскадного отказа интегрированных компонент системы – при выходе интегрируемого компонента из строя, использующие его компоненты автоматически переключаются на другой компонент, имеющий такой же программный компонент взаимодействия, но возможно, не предоставляющий весь функционал в полном объеме, что позволяет осуществлять эффективную демонстрацию работоспособности системы заказчикам даже при наличии некорректной конфигурации основных интегрируемых компонент системы.

Неоспоримым плюсом фреймворка Spring, упрощающим разработку и позволяющим избегать типовых ошибок кодирования, является декларативный стиль конфигурирования системы посредством использования файлов конфигурации и языковых аннотаций [11].

Данный фреймворк имеет большое разнообразие модулей, которые автоматизируют и упрощают следующие операции, необходимые в рамках данной ВКР:

* Запуск и настройка встроенного сервера приложений
* Сериализация и десериализация запросов
* Слабое связывание компонентов
* Работа с СУБД (MongoDB, Hazelcast)
* Предоставление HTTP клиента для вызовов внешних компонентов
* Ведение лога системы
* Конфигурирование API системы

Рисунок 31 отлично отображает все аспекты использования Spring Framework в приложении к программной архитектуре главного компонента системы.

Салатовым цветом на данном рисунке обозначены контроллеры, осуществляющие обработку запросов приложений. Для удобства чтения диаграмм они помечены стереотипом Controller Bean, что соответствует одноименному типу компонент во фреймворке Spring.

Контроллеры разделены на 2 части – основной (PoolManagerController) контроллер обработки локальных запросов (DistributedPoolManagerController). Данное разделение обусловлено тем, что в пользовательском веб-интерфейсе для пользователя нет разницы между ресурсами, ассоциированными с различными узлами системы, и типовые запросы по управлению такими ресурсами не привязаны к конкретному вычислительному узлу системы.

Основной контроллер берет на себя ответственность по определению целевого узла (узлов) пришедшего запроса и осуществлению агрегации ответов от них посредством делегирования соответствующих вызовов сервисному компоненту, реализующему интерфейс RemoteNodeExecutor.

Этот сервисный компонент наряду с другими сервисными компонентами помечены желтым цветом и имеют стереотип Service Bean. Работа с СУБД MongoDB осуществляется посредством вызовов компонента-репозитория, реализующего интерфейс MongoDao (выделен бирюзовым цветом морской волны и имеет стереотип Repository Bean.

Использование данных компонент является наиболее часто используемым вариантом использования фреймворка Spring и позволяет избегать написания и проектирования излишнего объёма программного кода для выполнения типовых задач, связанных с проектированием клиент-серверных приложений с трехзвенной архитектурой [10].

Данный фреймворк также берет на себя работу по осуществлению сериализации и десериализации тем, параметров и заголовков HTTP запросов и ответов в соответствующие им программные единицы языка Java.

Предоставление и продвижение данным фреймворком принципа слабой связности и легкой заменимости компонент, реализующих один и тот же интерфейс [10], значительно упростило работу над реализацией устойчивого к отказам программных частей главного компонента системы, который по умолчанию использует основные клиенты к интегрируемым продуктам, но при выходе их из строя автоматически переключится на резервные компоненты, эмулирующие поведение данных продуктов в объемах, позволяющих продемонстрировать основные идеи и концепции, привносимые данным комплексным интегрированным решением. Для этого был написан унифицированный абстрактный класс, предоставляющий подобную логику для всех его наследников (см. Приложение Б).

Использование взаимозаменяемых значительно снизило накладные расходы на написание модульных и интеграционных тестов, покрывающих большую часть разработанной системы.

Концепция каскадного разделения ответственности программных компонент внутри главного компонента системы при обработке типового запроса в целом была сохранена, промежуточными звеньями в имеющейся архитектуре стали сервисы ClusterStateService (см. Приложение В) и RemoteNodeExecutor, которые в совокупном использовании определяют, необходимо ли задействование локальных ресурсов узла для выполнения запроса, либо же этот запрос требует обработки другим узлом, после чего осуществляют делегирование обработки такого запроса другому узлу (рис. 31).

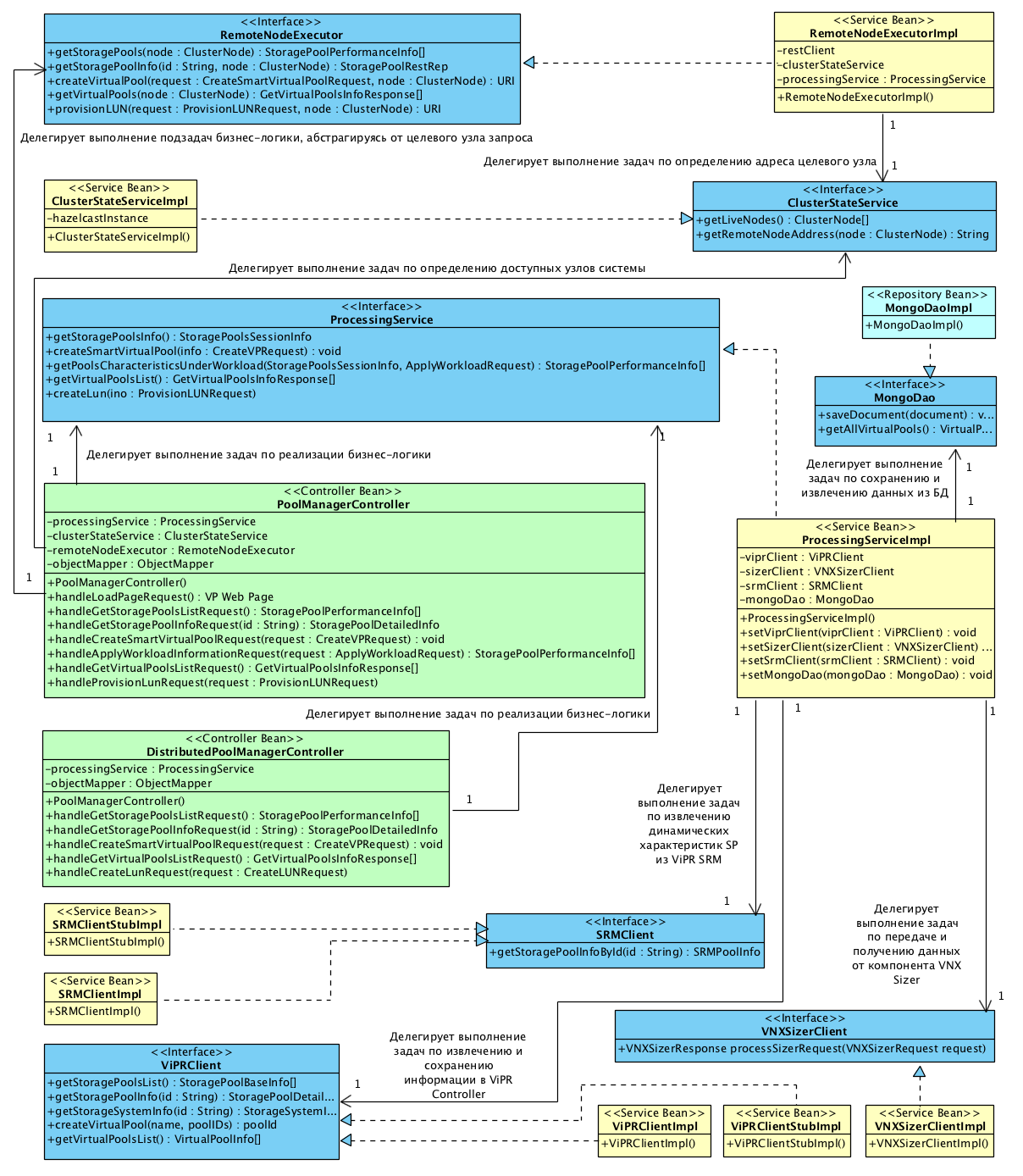


Рисунок 31 – Диаграмма классов главного компонента системы

* 1. Архитектура данных
     1. Архитектура постоянно хранимых данных

На рисунке 31 изображена диаграмма хранимых данных. Так как для выделения *LUN* единственной необходимой хранимой информацией является список созданных с помощью разработанной системы *VP* со списком приложений, планируемых к развертыванию и целевым значением времени отклика дискового пространства, то в СУБД MongoDB в коллекции осуществляется хранение документов именно такой структуры. Информация о том, какие *SP* входят в отдельно взятый *VP*, не сохраняется, так как эта информация доступна посредством REST API компонента ViPR Controller, а также ввиду того, что конфигурация *VP* может быть изменена администратором вручную посредством компонента ViPR Controller.

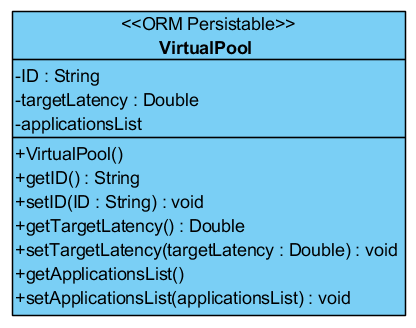


Рисунок 32 – Диаграмма постоянно хранимых данных в данной ВКР

* + 1. Архитектура временно хранимых данных

На рисунке 32 отражена диаграмма временно хранимых данных. Данная информация хранится в СУБД Hazelcast и используется для предоставления каждому узлу системы информации о других рабочих узлах в системе. Информация хранится в виде отображения идентификатора узла, определяемым автоматически при запуске узла, на список IP-адресов, по которым главный компонент системы может обрабатывать входящие запросы. При подключении узла системы или восстановлении связи между ним и другими узлами системы в данное отображение вносится информация о текущем восстановленном узле. СУБД Hazelcast сконфигурирована таким образом, что при выходе одного или нескольких каналов связи из строя соответствующие узлы, прямое соединение между которыми было потеряно, не получают информацию друг о друге из данного отображения. После получения информации о доступных узлах в системе, узел, обрабатывающий запрос, на основании информации о локальных сетевых интерфейсах и списке адресов удаленного узла автоматически определяет адрес, по которому удаленный узел будет доступен. Данная информация кэшируется главным компонентом для более эффективной обработки запросов.

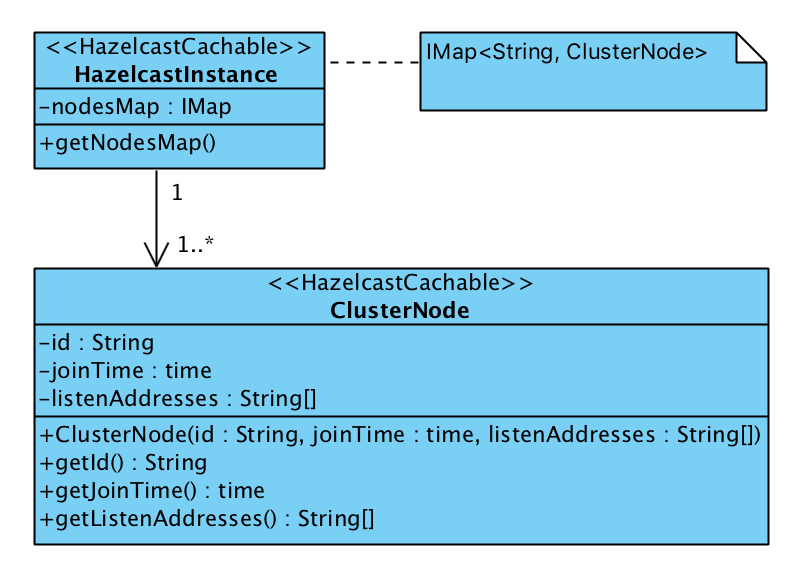


Рисунок 33 – Диаграмма временно хранимых данных в данной ВКР

* + 1. Архитектура API

Архитектура программного слоя взаимодействия главного компонента системы в значительной степени повторяет архитектуру системы, разработанной в рамках бакалаврской ВКР, расширяя ее появлением нового контроллера, осуществляющего обработку запросов, эквивалентных запросам в основной контроллер главного компонента системы, но выполняющий их исключительно над ресурсами, локализованными в рамках узла, на который пришел такой запрос. Такой контроллер обрабатывает следующие операции:

* Получение списка всех доступных *SP*
* Получение детальной информации о *SP* по его идентификатору
* Создание *VP*
* Получение списка созданных *VP*
* Выделение *LUN* на *VP*

Более детально программные интерфейсы основного контроллера и контроллера обработки локальных запросов изображены на рисунках 34 и 35 соответственно.

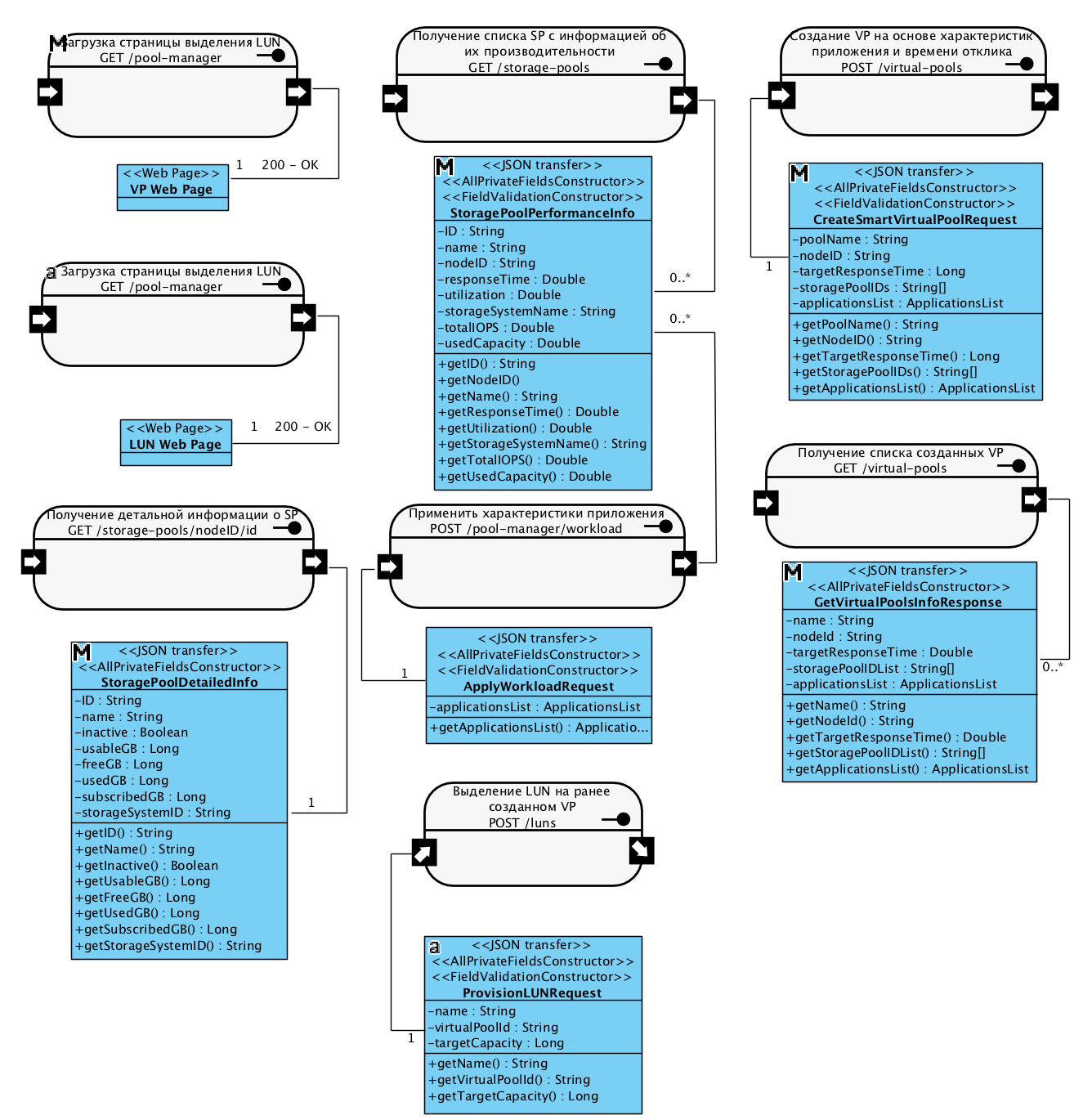


Рисунок 34 – Диаграмма программного интерфейса основного контроллера главного компонента системы

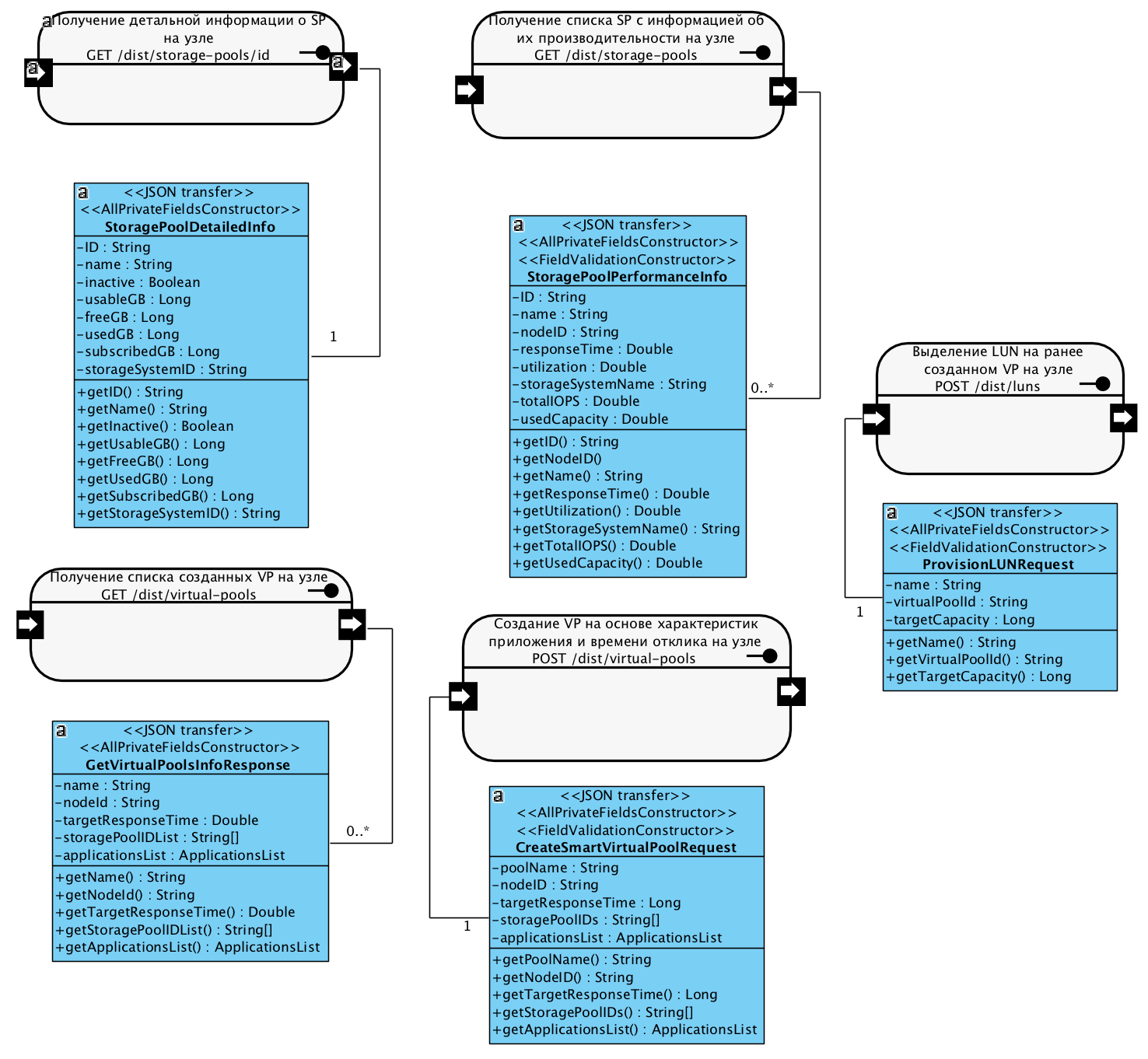


Рисунок 35 – Диаграмма программного интерфейса контроллера обработки локальных запросов главного компонента системы

Входные и выходные параметры методов контроллера обработки локальных запросов идентичные параметрам соответствующих методов основного контроллера.

1. Оценка отказоустойчивости системы
   1. Методы оценки

В различных статях, посвященных исследованиям различных метрик распределенных и высоконагруженных систем, выделяются следующие метрики гарантоспособности систем [20, 21]:

* Безотказность – свойство системы непрерывно сохранять работоспособное состояние в течение определённого времени работы
* Готовность – способность системы выполнять необходимые функции при определённых условиях эксплуатации и технического обслуживания в фиксированный интервал времени при условии обеспечения необходимыми внешними ресурсами
* Живучесть – свойство системы сохранять или восстанавливать способность выполнять основные функции в определённом объеме и на протяжении заданной наработки при изменении условий эксплуатации
* Целостность – свойство системы быть неизменной при функционировании в условиях случайных или преднамеренных искажений, или разрушающих воздействий извне системы
* Конфиденциальность – свойство системы обеспечивать защиту от несанкционированного использования информации или технического средства, подмены информации или повреждения информации изнутри системы
* Обслуживаемость – способность системы подлежать техническому обслуживанию, модификации и ремонту
* Функциональная безопасность – способность системы при наличии отказа не причинять опасных воздействий на человека или окружающую среду

Так как проектируемая система является распределённой системой с ненадежными каналами связи, для нее можно применять любую из вышеописанных метрик. Целью данной работы является выделение такой архитектуры системы, которая в данных терминах является:

* Безотказной
* Готовой
* Живучей
* Целостной

Иные метрики отказоустойчивости системы выходят за рамки данной работы.

Безотказность и готовность рассматриваемой системы подразумеваются по умолчанию – в отсутствие сетевых сбоев любого из каналов связи между узлами приложения система должна функционировать без сбоев, осуществляя взаимодействие с любым из активных узлов системы.

Живучесть системы является одной из ключевых метрик, рассматриваемых в данной работе. В случае изменения внешней среды (выхода из строя одного или нескольких каналов связи между узлами системы) пользователям должен предоставляться доступ ко всем узлам системы, каналы связи между которыми не были нарушены. Также важным критерием является скорость восстановления доступа к узлу, один или несколько каналов связи к которому были нарушены.

Целостность в данном случае рассматривается как согласованность и целостность данных, хранимых на узле, при выходе одного или нескольких узлов связи, соединяющих данный узел с другими узлами приложения. Все изменения, внесенные в копию приложения, запущенную на таком узле, до выхода из строя канала связи, обеспечивающего внесение данных изменений, должны быть сохранены на узле и быть доступны узлу-инициатору изменений после восстановления соответствующего канала связи из строя. Данные изменения должны быть доступны всем узлам, каналы связи между которыми до узла-исполнителя изменений не были повреждены, сразу после сохранения изменений на узле-исполнителе.

Таким образом, систему можно отнести к классу событийно-консистентных систем хранения и обработки данных с нулевым уровнем репликации данных. Как было выведено в разделе функциональной архитектуры, для проектируемой системы нет необходимости осуществлять репликацию данных отдельного узла приложения на другие узлы, так как данная информация в соответствии с выделенными функциональными требованиями является полезной только при наличии неповрежденного канала связи между рассматриваемым узлом и узлом, запрашивающим с него информацию.

В источниках также приводятся примеры статистических метрик по данным высокоуровневым метрикам, которые позволяют рассчитать определенные статистические коэффициенты, отражающие ту или иную сторону гарант способности системы [20]. Из них выбраны те, которые максимально показательно описывают сценарии, исследование которых является предметом данной НИР.

Коэффициент живучести

Где – число работоспособных состояний в случае обобщенного отказа кратности , – общее количество состояний системы, – количество функциональных единиц живучести.

Коэффициент готовности компонента

Где – средняя наработка компонента на отказ, – среднее время восстановления компонента. В контексте данной работы можно рассматривать время работы на отказ как характеристику канала связи, а время восстановления как характеристику узла приложения.

С помощью данных формулы и примерной оценки характеристик отказоустойчивости каналов связи можно произвести расчет данных метрик для предсказания метрик работоспособности системы в реальных условиях или условиях симуляции.

Для разработанной системы проведены тесты с использованием симуляции сбоев работы сетевых каналов связи в среде тестирования. Для каждого из сценариев тестирования произведены расчеты с использованием данных формул и выполнены замеры времени возвращения в работу компонентов, изолированных нерабочими каналами связи, после чего осуществлено сопоставление полученных результатов с результатами, предсказанными по данным формулам.

* 1. Результаты оценки

Для тестирования использовалась конфигурация, представленная на рисунке 21. Замеры осуществлялись путем конфигурирования соответствующей инфраструктуры с конфигурацией утилиты *netem* для достижения выбранного уровня потери пакетов и задержки сети.

Так как время определения потери и восстановления канала связи между узлами напрямую связано с интервалом обмена служебными пакетами между узлами Hazelcast, то для наглядности в сводной таблице отражено время обновлении информации об узле после восстановления канала за вычетом данного интервала.

При тестировании объем передаваемой информации за 1 операцию обновления состояния между узлами не превышал 1КБ, при получении обновленной информации после восстановления связи – 2КБ. Параметр средней наработки на отказ отдельно взятого канала связи был принят за 30 календарных дней в соответствии с типовыми политиками проведения сетевой профилактики ввиду отсутствия статистики о выходе из строя каналов связи.

В качестве метрики усреднения использовался 95% перцентиль после проведения 1000 замеров. Таблицы 1, 2 и 3 иллюстрируют результаты эксперимента при 0%, 1% и 2% потерь пакетов в рассматриваемом канале связи, соответственно

Таблица 1 – Результаты замеров характеристик системы при потере пакетов 0%.

|  |  |  |
| --- | --- | --- |
| Время задержки сети, мс | Время восстановления информации об узле, мс |  |
| 0 | 112 | 0.9999999568 |
| 30 | 396 | 0.9999998472 |
| 60 | 584 | 0.9999997747 |
| 90 | 621 | 0.9999997604 |

Таблица 2 – Результаты замеров характеристик системы при потере пакетов 1%.

|  |  |  |
| --- | --- | --- |
| Время задержки сети, мс | Время восстановления информации об узле, мс |  |
| 0 | 3328 | 0.9999987161 |
| 30 | 3675 | 0.9999985822 |
| 60 | 4404 | 0.9999983009 |
| 90 | 4872 | 0.9999981204 |

Таблица 3 – Результаты замеров характеристик системы при потере пакетов 2%.

|  |  |  |
| --- | --- | --- |
| Время задержки сети, мс | Время восстановления информации об узле, мс |  |
| 0 | 4678 | 0.9999981952 |
| 30 | 5225 | 0.9999979842 |
| 60 | 5604 | 0.999997838 |
| 90 | 5895 | 0.9999977257 |

Как можно видеть из результатов замеров, при отсутствии потери пакетов в сети и нулевой задержки сети Hazelcast обеспечивает обновление информации с задержкой, не отличимой для конечного пользователя. В случае возникновения потерь в сети в размере и увеличения времени отклика сети скорость обновления информации замедляется до нескольких секунд. Это объясняется тем, что Hazelcast использует одновременно TCP и UDP для взаимодействия между узлами данной СУБД – UDP для обнаружения восстановления связи и TCP для установления постоянного соединения и передачи данных после восстановления соединения. Полученные результаты являются допустимыми и соответствуют ожиданиям от системы, решающей рассматривающую задачу при заданных условиях.

Данные о значении показателя живучести системы из 3 узлов при различных отказах представлены в таблице 4.

Таблица 4 – Показатель живучести для системы из 3 узлов.

|  |  |  |  |
| --- | --- | --- | --- |
| Кратность отказа, | Общее число состояний системы, | Число работоспособных состояний, |  |
| 0 | 8 | 8 | 1 |
| 1 | 7 | 0.875 |
| 2 | 4 | 0.5 |
| 3 | 1 | 0.125 |

В качестве состояний системы принимались состояния, когда все узлы доступны для работы друг с другом (1), когда связь между любыми 2 узлами нарушена (3), когда 2 связи между узлами нарушено (3) и когда все связи между узлами нарушены (1).

ЗАКЛЮЧЕНИЕ

В рамках данной ВКР была определена и подтверждена актуальность и практическая значимость выбранной темы магистерской диссертации, сформирован терминологический базис и обзор прикладной предметной области с описанием используемых компонентов и систем, подлежащих интеграции. В соответствии с целями и задачами, решаемыми в данной ВКР, были подобраны различные практики и подходы к решению задач построения высоко надежных распределенных систем подобного класса, проведен глубокий анализ различных подходов, используемых для организации высоко надежной системы при заданных условиях, а также осуществлено детальное рассмотрение возможностей, предоставляемыми уже используемыми программными продуктами, по организации высоко надежной системы. В рассмотрение также были взяты и другие современные технологии, и программные продукты, которые могут решить поставленную задачу, проведен их сравнительный анализ, обоснован выбор и определены методы оценки отказоустойчивости разработанной системы.

С учетом функциональных требований и требований к высокой доступности приложения была сформирована функциональная, системная, программная архитектуры, а также архитектура хранимых данных и проектируемых интерфейсов, предложена методика оценки качества работы спроектированного системы, произведены испытания с целью оценки качества по предложенной методике.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Отдел информатизации образования, «Система Хранения Данных (СХД)», [Электронный ресурс]. URL: https://hotuser.ru/shd (дата обращения: 23.04.2018).
2. EMC Corporation, “Introduction to the EMC VNX2 Series”, [Электронный ресурс]. URL: http://www.emc.com/collateral/white-papers/h12145-intro-new-vnx-series-wp.pdf — 6-7с. (дата обращения: 23.04.2018).
3. Techopedia, “Logical Unit Number (LUN)”, [Электронный ресурс]. URL: http://www.techopedia.com/definition/321/logical-unit-number-lun (дата обращения: 23.04.2018).
4. Справочный центр RU-TLD, “RAID”, [Электронный ресурс]. URL: http://ru-tld.ru/h/help\_system:servera:raid:raid (дата обращения 23.04.2018).
5. EMC Corporation, “What is a ViPR Virtual Pool?”, [Электронный ресурс]. URL: http://www.emc.com/techpubs/vipr/what\_is\_virtual\_pool-1.htm (дата обращения: 23.04.2018).
6. EMC Corporation, “ViPR Controller”, [Электронный ресурс]. URL: http://www.emc.com/products/storage/software-defined-storage/vipr-controller.htm (дата обращения: 23.04.2018).
7. EMC Corporation, “EMC ViPR REST API”, [Электронный ресурс]. URL: http://build.coprhd.org/jenkins/job/CH-coprhd-controller-master/ws/CH-coprhd-controller-master/build/gradle/tools/apidocs/apidocs/index.html (дата обращения: 23.04.2018).
8. EMC Corporation, “ViPR Controller Java Client”, [Электронный ресурс]. URL: http://community.emc.com/docs/DOC-33848 (дата обращения: 23.04.2018).
9. EMC Corporation, “ViPR SRM”, [Электронный ресурс]. URL: http://www.emc.com/data-center-management/vipr-srm.htm#!resources (дата обращения: 23.04.2018).
10. Pivotal Software, Inc., “Spring Framework Reference Documentation”, [Электронный ресурс]. URL: https://docs.spring.io/spring/docs/current/spring-framework-reference/pdf/spring-framework-reference.pdf (дата обращения: 23.04.2018).
11. Pivotal Software, Inc., “Spring Boot Reference Guide”, [Электронный ресурс]. URL: https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/pdf/spring-boot-reference.pdf (дата обращения: 23.04.2018).
12. Brown A. The Architecture of Open Source Applications, Volume II / Brown A., Wilson G. – Raleigh, North Carolina, United States: Lulu, 2008.
13. Atchison L. Architecting for Scale: High Availability for Your Growing Applications / Atchison L. – Sebastopol, CA, United States: O’Reilly Media, Inc, 2016.
14. Marcus E. Blueprints for High Availability / Marcus E., Stern H. – Indianapolis, Indiana: Wiley Publishing, 2003.
15. Oracle, Inc., “Oracle Database: High Availability Best Practices 11g” [Электронный ресурс]. URL: https://docs.oracle.com/cd/E11882\_01/server.112/e10803.pdf (дата обращения: 23.04.2018).
16. Mehrabani, A. MongoDB High Availability / Mehrabani, A. – Birmingham, UK: Packt Publishing, 2014.
17. Dasadia, C. MongoDB Cookbook / Dasadia, C., Nayak, A. – Birmingham, UK: Packt Publishing, 2016.
18. Chodorow, K. MongoDB: The Definitive Guide / Chodorow, K. – Sebastopol, CA, United States: O’Reilly Media, Inc, 2013.
19. Hazelcast, Inc. “Hazelcast Documentation, version 3.8.2”, [Электронный ресурс]. URL: http://docs.hazelcast.org/docs/3.8.2/manual/pdf/hazelcast-documentation-3.8.2.pdf (дата обращения 23.04.2018)
20. Федухин А.В., Сеспедес Гарсия Н.В., “Атрибуты и метрики гарантоспособных компьютерных систем” [Электронный ресурс]. URL: https://cyberleninka.ru/article/n/atributy-i-metriki-garantosposobnyh-kompyuternyh-sistem (дата обращения: 23.04.2018)
21. Карпов В.С., Ивутин А.Н., Суслин А.А., “Подход к реализации методики оценки надежности ПО на основе комплексных метрик” [Электронный ресурс]. URL: https://cyberleninka.ru/article/v/podhod-k-realizatsii-metodiki-otsenki-nadezhnosti-po-na-osnove-kompleksnyh-metrik (дата обращения: 23.04.2018)

ПРИЛОЖЕНИЕ А

**version: '3.3'  
services:  
 node-1:  
 image:** com/emc/coprhd/sp:${VERSION}  
 **container\_name:** node-1  
 **build:  
 context:** .  
 **dockerfile:** ./Dockerfile  
 **args:** - VERSION=${VERSION}  
 **environment:** - NODE=1  
 **ports:** - 80:8080  
 - 8080:8080  
 - 5005:5005  
 **networks:  
 internal-1:  
 ipv4\_address:** 172.48.8.2  
 **dc-12:  
 ipv4\_address:** 172.48.11.4  
 **dc-13:  
 ipv4\_address:** 172.48.13.4  
 **mongo-1:  
 image:** mongo:3.6  
 **container\_name:** mongo-1  
 **ports:** - 27017:27017  
 **networks:  
 internal-1:  
 ipv4\_address:** 172.48.8.3  
 **hz-1:  
 image:** com/emc/coprhd/sp/hz:3.9.3  
 **container\_name:** hz-1  
 **ports:** - 5701:5701  
 **build:  
 context:** hz-config  
 **dockerfile:** Dockerfile  
 **args:** - NODE=1  
 **networks:  
 internal-1:  
 ipv4\_address:** 172.48.8.4  
 **dc-12:  
 ipv4\_address:** 172.48.11.2  
 **dc-13:  
 ipv4\_address:** 172.48.13.2  
 **sizer-1:  
 image:** com/emc/vnxsizer-api:latest  
 **container\_name:** sizer-1  
 **ports:** - 8081:8080  
 **networks:  
 internal-1:  
 ipv4\_address:** 172.48.8.5  
 **srm-1:  
 image:** com/emc/coprhd/srm:${VERSION}  
 **container\_name:** srm-1  
 **ports:** - 8082:8080  
 **networks:  
 internal-1:  
 ipv4\_address:** 172.48.8.6  
 **vipr-1:  
 image:** com/emc/coprhd/vipr:${VERSION}  
 **container\_name:** vipr-1  
 **ports:** - 8083:4443  
 **networks:  
 internal-1:  
 ipv4\_address:** 172.48.8.7  
 **node-2:  
 image:** com/emc/coprhd/sp:${VERSION}  
 **container\_name:** node-2  
 **build:  
 context:** .  
 **dockerfile:** ./Dockerfile  
 **args:** - VERSION=${VERSION}  
 **environment:** - NODE=2  
 **ports:** - 1080:8080  
 - 18080:8080  
 - 15005:5005  
 **networks:  
 internal-2:  
 ipv4\_address:** 172.48.9.2  
 **dc-12:  
 ipv4\_address:** 172.48.11.5  
 **dc-23:  
 ipv4\_address:** 172.48.12.4  
 **mongo-2:  
 image:** mongo:3.6  
 **container\_name:** mongo-2  
 **ports:** - 37017:27017  
 **networks:  
 internal-2:  
 ipv4\_address:** 172.48.9.3  
 **hz-2:  
 image:** com/emc/coprhd/sp/hz:3.9.3  
 **container\_name:** hz-2  
 **ports:** - 15701:5701  
 **build:  
 context:** hz-config  
 **dockerfile:** Dockerfile  
 **args:** - NODE=2  
 **networks:  
 internal-2:  
 ipv4\_address:** 172.48.9.4  
 **dc-12:  
 ipv4\_address:** 172.48.11.3  
 **dc-23:  
 ipv4\_address:** 172.48.12.2  
 **sizer-2:  
 image:** com/emc/vnxsizer-api:latest  
 **container\_name:** sizer-2  
 **ports:** - 18081:8080  
 **networks:  
 internal-2:  
 ipv4\_address:** 172.48.9.5  
 **srm-2:  
 image:** com/emc/coprhd/srm:${VERSION}  
 **container\_name:** srm-2  
 **ports:** - 18082:8080  
 **networks:  
 internal-2:  
 ipv4\_address:** 172.48.9.6  
 **vipr-2:  
 image:** com/emc/coprhd/vipr:${VERSION}  
 **container\_name:** vipr-2  
 **ports:** - 18083:4443  
 **networks:  
 internal-2:  
 ipv4\_address:** 172.48.9.7  
 **node-3:  
 image:** com/emc/coprhd/sp:${VERSION}  
 **container\_name:** node-3  
 **build:  
 context:** .  
 **dockerfile:** ./Dockerfile  
 **args:** - VERSION=${VERSION}  
 **environment:** - NODE=3  
 **ports:** - 2080:8080  
 - 28080:8080  
 - 25005:5005  
 **networks:  
 internal-3:  
 ipv4\_address:** 172.48.10.2  
 **dc-23:  
 ipv4\_address:** 172.48.12.5  
 **dc-13:  
 ipv4\_address:** 172.48.13.5  
 **mongo-3:  
 image:** mongo:3.6  
 **container\_name:** mongo-3  
 **ports:** - 47017:27017  
 **networks:  
 internal-3:  
 ipv4\_address:** 172.48.10.3  
 **hz-3:  
 image:** com/emc/coprhd/sp/hz:3.9.3  
 **container\_name:** hz-3  
 **ports:** - 25701:5701  
 **build:  
 context:** hz-config  
 **dockerfile:** Dockerfile  
 **args:** - NODE=3  
 **networks:  
 internal-3:  
 ipv4\_address:** 172.48.10.4  
 **dc-23:  
 ipv4\_address:** 172.48.12.3  
 **dc-13:  
 ipv4\_address:** 172.48.13.3  
 **sizer-3:  
 image:** com/emc/vnxsizer-api:latest  
 **container\_name:** sizer-3  
 **ports:** - 28081:8080  
 **networks:  
 internal-3:  
 ipv4\_address:** 172.48.10.5  
 **srm-3:  
 image:** com/emc/coprhd/srm:${VERSION}  
 **container\_name:** srm-3  
 **ports:** - 28082:8080  
 **networks:  
 internal-3:  
 ipv4\_address:** 172.48.10.6  
 **vipr-3:  
 image:** com/emc/coprhd/vipr:${VERSION}  
 **container\_name:** vipr-3  
 **ports:** - 28083:4443  
 **networks:  
 internal-3:  
 ipv4\_address:** 172.48.10.7  
  
**networks:  
 internal-1:  
 driver:** bridge  
 **attachable:** true  
 **ipam:  
 driver:** default  
 **config:** - **subnet:** 172.48.8.0/24  
 **internal-2:  
 driver:** bridge  
 **attachable:** true  
 **ipam:  
 driver:** default  
 **config:** - **subnet:** 172.48.9.0/24  
 **internal-3:  
 driver:** bridge  
 **attachable:** true  
 **ipam:  
 driver:** default  
 **config:** - **subnet:** 172.48.10.0/24  
 **dc-12:  
 driver:** bridge  
 **attachable:** true  
 **ipam:  
 driver:** default  
 **config:** - **subnet:** 172.48.11.0/24  
 **dc-23:  
 driver:** bridge  
 **attachable:** true  
 **ipam:  
 driver:** default  
 **con]fig:** - **subnet:** 172.48.12.0/24  
 **dc-13:  
 driver:** bridge  
 **attachable:** true  
 **ipam:  
 driver:** default  
 **config:** - **subnet:** 172.48.13.0/24

ПРИЛОЖЕНИЕ Б

*/\*  
 \* Copyright 1994-2018 EMC Corporation. All rights reserved.  
 \*/*  
  
**public interface** FallBackable **extends** BeanNameAware {  
 String getName();  
  
 **void** initialize();  
}

*/\*  
 \* Copyright 1994-2018 EMC Corporation. All rights reserved.  
 \*/***package** com.emc.coprhd.sp.service.common;  
  
**public abstract class** AbstractFallBackable<T **extends** FallBackable>

**implements** FallBackable {  
 **protected** List<T> **clients**;  
 **protected** T **activeClient**;  
 **protected** String **name**;  
  
 @Override  
 **public** String getName() {  
 **return name**;  
 }  
  
 @Override  
 @SuppressWarnings(**"SuspiciousGetterSetter"**)  
 **public void** setBeanName(**final** String name) {  
 **this**.**name** = name;  
 }  
  
 **protected** <R> R performCascadeOperation(

**final** Function<T, R> operation,

**final** Logger log) {  
 **final** Iterator<T> it = **clients**.subList(**clients**.indexOf(**activeClient**),

**clients**.size()).iterator();  
 *//noinspection BooleanVariableAlwaysNegated* **do** {  
 **try** {  
 **return** operation.apply(**activeClient**);  
 } **catch** (Exception e) {  
 log.error(**"Error during client {} initialization"**,

**activeClient**.getName(), e);  
 **if** (it.hasNext()) {  
 **final** T next = it.next();  
 log.info(**"Switching client to {}"**, next.getName());  
 **activeClient** = next;  
 } **else** {  
 **throw new** StartupException(**"No more clients available!"**, e);  
 }  
 }  
 } **while** (**true**);  
 }  
}

ПРИЛОЖЕНИЕ В

*/\*  
 \* Copyright 1994-2018 EMC Corporation. All rights reserved.  
 \*/*  
  
@Service  
**public class** ClusterStateServiceImpl **implements** ClusterStateService {  
 **private static final** Logger ***LOGGER*** = LoggerFactory.*getLogger*(ClusterStateServiceImpl.**class**);  
  
 **private final** String **nodeId**;  
 **private final** HazelcastInstance **hzInstance**;  
 **private final** IMap<String, ClusterNode> **nodes**;  
 **private final** Set<AddressInfo> **ignoredNetworks**;  
 **private final** ClusterMembershipListener **listener**;  
  
 @Autowired  
 **public** ClusterStateServiceImpl(  
 **final** IMap<String, ClusterNode> nodes,  
 **final** HazelcastInstance hzInstance,  
 **final** ClusterMembershipListener listener,  
 @Value(**"${com.emc.coprhd.sp.node-id}"**) **final** String nodeId,  
 @Value(**"${com.emc.coprhd.sp.ignored-networks}"**)

**final** String ignoredNetworks) {  
 **this**.**nodes** = nodes;  
 **this**.**nodeId** = nodeId;  
 **this**.**listener** = listener;  
 **this**.**hzInstance** = hzInstance;  
 **this**.**ignoredNetworks** = Stream.*of*(ignoredNetworks.split(**","**))  
 .map(i -> **new** AddressInfo(**""**, i))  
 .collect(Collectors.*toSet*());  
 }

@Override  
 @SuppressWarnings(**"ReturnOfNull"**)  
 **public** String getNodeAddress(**final** ClusterNode node) {  
 **try** {  
 ***LOGGER***.debug(**"{} node: {}"**, RuntimeUtils.*enterMethodMessage*(), node);  
 **final** Set<AddressInfo> localAddresses = NetworkUtils.*getHostAddresses*();  
 localAddresses.removeAll(**ignoredNetworks**);  
 **final** Set<AddressInfo> nodeAddresses = node.getListenAddresses();  
 **final** Set<AddressInfo> common = Sets.*intersection*(nodeAddresses,

localAddresses);  
 ***LOGGER***.debug(**"Common networks for node {} is {}"**, node, common);  
 **final** String result = common.stream()

.map(AddressInfo::getAddress).findFirst()  
 .orElseThrow(() -> **new** NoSuchElementException(

**"No listen address found for node "** + node));  
 ***LOGGER***.debug(**"{} node: {} address: {}"**,

RuntimeUtils.*exitMethodMessage*(), node, result);  
 **return** result;  
 } **catch** (SocketException e) {  
 ***LOGGER***.error(**"Can't get node address {}!"**, node, e);  
 **return null**;  
 }  
 }

@Override  
 **public** List<ClusterNode> getAvailableNodes() {  
 ***LOGGER***.debug(RuntimeUtils.*enterMethodMessage*());  
 **final** List<ClusterNode> result = **new** ArrayList<>(**nodes**.values());  
 ***LOGGER***.debug(**"{} nodes: {}"**, RuntimeUtils.*exitMethodMessage*(), result);  
 **return** result;  
 }  
  
 @EventListener(ContextRefreshedEvent.**class**)  
 **void** contextRefreshedEvent() {  
 ClusterUtils.*addNodeInfoToCluster*(**ignoredNetworks**, **nodeId**, **hzInstance**, **nodes**);  
 **hzInstance**.getCluster().addMembershipListener(**listener**);  
 }  
}