МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»**
Кафедра технологий программирования

ПОДДЕРЖАНИЕ ИЗБЫТОЧНОСТИ ДЛЯ НЕНАДЁЖНЫХ РАСПРЕДЕЛЁННЫХ СИСТЕМ

АВТОРЕФЕРАТ МАГИСТЕРСКОЙ РАБОТЫ

студента 2 курса 273 группы
направления 02.04.03 — Математическое обеспечение и администрирование
информационных систем
факультета КНиИТ
Соколкова Павла Вячеславовича

Научный руководитель
доцент, к.ф.-м.н. _____ Г. Г. Наркайтис

Заведующий кафедрой
к.ф.-м.н. _____ И. А. Батраева

Саратов 2021

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Много компаний сталкиваются с проблемой постоянного увеличения объёма данных. Разработчики стремятся повысить производительность файловых систем, которые оказались критически важными для общей производительности широкого класса приложений. Научные и инженерные сообщества продолжают добиваться успехов в увеличении производительности и масштабируемости распределенных систем хранения.

Чтобы справляться с постоянно увеличивающимися нагрузками, потоком пользователей и объёмом вычислений, важно развивать вычислительную систему или, другими словами, центр обработки данных. Необходимо построить и поддерживать такую сетевую и программную архитектуру, которая была бы способна к масштабированию, то есть, без перерыва в работе и незаметно для пользователя увеличить мощность. В настоящее время наиболее популярным для этого способом является «горизонтальная» масштабируемость. При данном подходе в систему добавляют новые узлы, они дополняют уже существующий комплекс, обеспечивая необходимую для предприятия мощность. Отличие «горизонтальной» масштабируемости от «вертикальной» состоит в том, что в первом случае добавляют не самые производительные сервера, их объединяют в кластер. За счет большого количества таких систем достигается необходимый уровень скорости работы или надежности всего комплекса. При втором, «вертикальном», подходе наращивание производительности происходит за счет использования более мощных, но дорогостоящих компонентов.

Для надежного хранения данных наиболее логичным является «горизонтальная» масштабируемость. Информация дублируется на разные узлы, которые могут быть разделены не только логически, но и физически, например, они могут располагаться в разных центрах обработки данных, помещениях, странах, иметь независимые источники питания. Существуют решения, в которых распределённые системы имеют единую файловую систему, и взаимодействие узлов скрыто от пользователя интерфейсом. Вопрос распределённых файловых систем рассматривается в [1].

В подобных распределённых системах возникает необходимость равномерного распределения данных между узлами, поддержание непротиворечивого состояния, а также доступность всей системы, то есть, способность в любой момент времени обрабатывать поступающие запросы пользователей.

При использовании распределённых хранилищ возникает необходимость не только обеспечивать быстрый обмен данными, но и гарантировать их сохранность и безопасность, так как сбои в больших системах не редки и скорее являются нормой, чем исключением. Распределенные системы хранения обеспечивают надежный доступ к данным благодаря избыточности, распределенной по индивидуально ненадежным узлам. Сценарии применения включают центры обработки данных, одноранговые системы хранения и хранилища в беспроводных сетях. Перечисленные варианты использования встречаются повсеместно, и всем им присуща названная проблема. Этот факт указывает на актуальность данной работы, в рамках которой разработано распределённое надёжное хранилище.

Целью данной работы является разработка распределённого хранилища, поддерживающего избыточность данных. Разрабатываемый продукт также должен самостоятельно, в автоматическом режиме справляться с отказом устройств, то есть быть способным восстанавливать и создавать необходимое количество копий сохранённой информации. Данная система также должна быть *прозрачной* для пользователя, это означает, что приложение выполняет обработку данных с логической точки зрения так, будто все операции осуществляются единственным компьютером. В контексте поставленной цели можно выделить следующие задачи:

1. Изучить существующие программные комплексы для создания распределённого хранилища;
2. Разработать архитектуру программного продукта;
3. Разработать эффективное распределённое хранилище с избыточными данными;
4. Провести анализ и тестирование разработанного продукта.

Методологические основы для построения надёжного распределённого хранилища и для поддержания избыточности представлены в работах С. Вейла [2–4], В.В. Воеводина [5], Э. Таненбаума [6], Т. Кормена [7].

Магистерская работа состоит из введения, 3 разделов, заключения, списка использованных источников и 6 приложений. Общий объем работы — 66 страниц, из них 50 страниц — основное содержание, включая 16 рисунков, цифровой носитель в качестве приложения, список использованных источников информации — 31 наименование.

1 Существующие программные комплексы

Проблема поддержания избыточности и надёжности в распределённой системе является достаточно популярной и актуальной. Многие компании сталкиваются с названной проблемой и вынуждены создавать собственное решение, подходящее для конкретных задач. Поэтому уже существует некоторое количество программных систем, направленных на поддержание избыточности. Рассмотрим некоторые из них.

В [3] приводится пример реализации распределённого хранилища. Авторы указывают, что разработанная система является крайне эффективной, надёжной и масштабируемой. Приложение использует алгоритм CRUSH [2], который также будет рассмотрен и использован в данной работе.

В [8] описывается другое распределённое хранилище, которое представляет собой логически единую файловую систему, физически распределённую на наборе рабочих станций. Доступность и надёжность хранимых данных достигается за счет рандомизированного хранилища. Недостатком данной системы являются недостаточно эффективные подсистемы, из-за которых становится невозможным быстрый доступ к малому набору данных десятком тысяч клиентов.

Некоторые другие распределённые хранилища, например, Vesta [9], Galley [10], PVFS [11], Swift [12], также имеют свои недостатки. В некоторых из них имеется довольно медленное обращение к метаданным или отсутствует достаточно надежная репликация и избыточность данных.

Работы в области частичной репликации и поддержания надежности копированием данных продолжается до сих пор, в [13] предлагается подход для преобразования реляционной базы данных для асинхронной репликации. В [14] рассматривается новая стратегия поддержания надёжности с усовершенствованием метода инкрементной репликации. В работе [15] предлагается собственный алгоритм репликации на основе использования транзакций. Автор работы [16] предлагает решение с использованием механизма хэш-функций [17].

В [18] предлагают следующие виды репликации:

1. «снимок»: периодическое копирование всей информации с сервера-«мастера» на «слейв»-серверы;
2. «журнальный»: копирование изменённых данных при отслеживании из-

менений в журнале транзакций;

3. «слияние»: данных анализируются и объединяются на «мастере» и «слейвах».

1.1 Алгоритм репликации данных

Объектно-ориентированное хранилище — это архитектура, которая обещает улучшение управляемости, масштабируемости и производительности. В отличие от обычных блочных жестких дисков, объектно-ориентированные устройства хранения (OSD) управляют внутренним распределением дисков, предоставляя интерфейс для чтения и записи объектов с переменным размером. Такие устройства объединяют центральный процессор, сетевой интерфейс, локальный кэш с нижележащим запоминающим устройством или RAID [19–21]. В такой системе данные каждого файла обычно распределены по относительно небольшому количеству именованных объектов по всему кластеру хранения. Объекты реплицируются на нескольких устройствах (или используют некоторые другие схемы резервирования) для защиты от потери данных при наличии сбоев. Объектные системы хранения упрощают компоновку данных, заменяя большие списки блоков маленькими объектами и уменьшая сложность выделения памяти для больших блоков. Хотя это значительно улучшает масштабируемость за счет сокращения метаданных и уменьшения сложности размещения файлов, задача распределения данных между тысячами устройств, как правило, с различной емкостью и производительностью остаётся.

Большинство систем просто записывают новые данные на недостаточно используемые устройства. Основная проблема с этим подходом заключается в том, что данные крайне редко перемещаются после того, как их один раз запишут. Даже идеальное распределение станет несбалансированным, когда хранилище будет увеличено, потому что новые диски либо будут пусты, либо будут содержать только новые данные. Старые или новые диски могут быть заняты в зависимости от загруженности системы, но только в самых редких ситуациях они будут использоваться в равной степени.

Надежное решение состоит в том, чтобы распределить все данные в системе случайным образом среди доступных устройств хранения. Это приводит к вероятностно-сбалансированному распределению и равномерно перемешивает старые и новые данные вместе. Когда новое хранилище добавлено,

случайный образец существующих данных переносится на новые устройства хранения для восстановления баланса. Главным преимуществом данного подхода является то, что в среднем все устройства будут загружены одинаково, что позволит системе хорошо работать при любой потенциальной рабочей нагрузке. Кроме того, в большом хранилище один большой файл будет случайным образом распределен по большому набору доступных устройств, обеспечивая высокий уровень параллелизма и совокупной пропускной способности. Тем не менее, простое распределение на основе хэша не справляется с изменениями в количестве устройств, влекущими за собой массовую перестановку данных. Существуют рандомизированные схемы распределения, которые осуществляют репликацию путем распределения реплик каждого диска по многим другим устройствам, но они страдают от высокой вероятности потери данных из-за одновременных сбоев устройств.

В данной работе будет рассматриваться CRUSH (контролируемая репликация при масштабируемом хэшировании) — алгоритм псевдослучайного распределения данных, который эффективно и надежно распределяет объект реплики в гетерогенном структурированном кластере. CRUSH реализован как псевдослучайная, детерминированная функция, которая отображает входное значение, обычно идентификатор объекта или группы объектов, в список устройств для хранения реплики объекта. Отличие от обычных подходов при таком размещении данных состоит в том, что для CRUSH требуется только компактное иерархическое описание устройств, составляющих хранилище и знание политики размещения реплик. Этот подход имеет два ключевых преимущества: во-первых, он полностью распределен так, что любая заинтересованная сторона в большой системе может самостоятельно рассчитать местоположение любого объекта; во-вторых, требуются небольшие метаданные, в основном статические, изменяющиеся, только когда устройства добавляются или удаляются.

CRUSH предназначен для оптимального распределения данных с использованием доступных ресурсов, эффективной реорганизации данных, когда устройства хранения добавляются или удаляются, и обеспечивает гибкость при размещении реплик объекта, которые максимизируют надёжность при наличии возможности отказа устройств.

CRUSH наиболее близок к семейству алгоритмов RUSH [4], на которых

он основан. RUSH остается единственным описанным в литературе набором алгоритмов, который использует функцию отображения вместо явного хранения метаданных и поддерживает эффективное добавление и удаление взвешенных устройств. Несмотря на эти основные свойства, ряд причин делают RUSH мало применимым на практике. CRUSH полностью обобщает полезные элементы RUSH$_P$ и RUSH$_T$ при разрешении ранее нерешенной надежности и проблемы репликации, предлагая большую производительность и гибкость.

1.2 Алгоритм CRUSH

Алгоритм CRUSH распределяет объекты данных по запоминающим устройствам в соответствии с весом каждого устройства, аппроксимируя равномерное распределение вероятностей. Распределение контролируется иерархической кластерной картой, представляющей доступные ресурсы хранения и состоящей из логических элементов, из которых она построена. Например, можно описать большую установку с точки зрения рядов серверных шкафов, шкафов, заполненных дисковыми полками, и полок, заполненных запоминающими устройствами. Политика распространения данных определяется в терминах правил размещения, которые определяют, сколько целевых объектов реплики выбрано из кластера и какие ограничения накладываются на размещение реплики. Например, можно указать, что три зеркальные реплики должны быть размещены на устройствах в разных физических шкафах, чтобы они не разделяли одну и ту же электрическую цепь.

Получая одно входное целочисленное значение x, CRUSH выведет упорядоченный список $\overline{R}$ из n различных целей хранения. CRUSH использует сильную хэш-функцию с несколькими входами, чьи входы включают в себя x, что делает отображение полностью детерминированным и независимо вычисляемым, используя только карту кластера, правила размещения и x. Распределение является псевдослучайным в том смысле, что нет очевидной корреляции между результирующим выходным сигналом от аналогичных входных данных или с элементами, хранящимися на любом устройстве хранения. Мы говорим, что CRUSH генерирует декластеризованное распределение реплик в том смысле, что набор устройств, совместно использующих реплики для одного элемента, также является независимым от всех других элементов.

2 Разработанное приложение

2.1 Архитектура приложения

Разрабатываемое приложение представляет собой несколько независимых программных компонентов, запускаемых на рабочих станциях отдельными процессами, которые могут выполняться в виде демонов [22]. Они могут быть запущены как на одном сервере, так и на распределённом кластере, в таком случае обмен сообщениями и взаимодействие компонентов происходит за счет высокоскоростной передачи данных в сети Интернет. Независимость процессов даёт ряд преимуществ, например, нет необходимости устанавливать весь программный комплекс целиком. В случае добавления нового устройства хранения будет достаточно запустить лишь только компонент, отвечающий непосредственно за обработку операций с данными и их хранение, затем обновить карту кластера, проинформировав мониторы о присоединение нового устройства.

На верхнем уровне архитектура представляет собой классическое клиент-серверное приложение с добавлением вспомогательного компонента в виде монитора. В комплексе работает не единственное устройство хранения, а набор таких устройств. Также присутствует дублирование монитора.

Таким образом, основными элементами разработанного приложения являются следующие компоненты:

— Клиентское приложение;
— Монитор;
— Устройство хранения.

Также важным элементом всей системы является карта кластера, но она представляет собой лишь объект в памяти, а не отдельный активный процесс, поэтому не была включена в список.

Взаимодействие между компонентами происходит за счет обмена сообщениями через сокеты. В запущенной системе всегда должен быть активен как минимум один монитор и одно устройство хранения, поскольку именно эти компоненты обеспечивают функциональную полноту и является критически важными частью приложения.

Стандартный сценарий функционирования с описанием выполняемых операций следующий:

— Клиенту необходимо выполнить некоторую операцию с данными, храня-

щимися в кластере, происходит конкретное определение такой операции и инициализация объекта для сохранения (или иной операции), его ключа и содержания;
- Клиент хранит текущую известную ему карту и версию карты кластера;
- Клиент запрашивает у монитора актуальную версию карты и при необходимости саму карту;
- С помощью алгоритма CRUSH осуществляется поиск первичного устройства хранения для объекта с заданным ключом;
- Клиент инициирует запрос к первичному устройству, отправляет данные, получает соответствующий ответ на свой запрос;
- Устройство хранения аналогично клиенту запрашивает версию карты и саму карту кластера у монитора;
- Устройство хранения получает запрос от клиента на выполнение некоторой операции, если это операция *GET_OBJECT*, то отправляет данные клиенту;
- Для иных операций устройство хранения по карте кластера с помощью CRUSH определяет вторичные устройства для заданного ключа и дублирует операции на обнаруженные экземпляры;
- Активный монитор постоянно опрашивает устройства хранения для поддержания актуального состояния карты;
- Устройство хранения в случае изменения карты инициирует восстановление данных или репликацию на новых вторичных устройствах.

Таким образом, взаимодействие между клиентом и устройством хранения является практически прямым, за тем исключением, что клиенту сначала необходимо определить текущее активное хранилище через информацию, агрегируемую монитором. Однако в случае отсутствия изменений в карте, клиенту не требуется выполнять лишние запросы к монитору, и он может напрямую взаимодействовать с подходящим хранилищем.

2.2 Карта кластера

Конфигурация всей системы хранения представляется в виде так называемой карты кластера. Это абстрактный объект, который хранится во всех компонентах системы (клиент, монитор, хранилище) и отражает структуру системы хранения, то есть в каком виде представлены реальные сервера в вычислительной сети.

Устройства хранения могут быть представлены в системе в виде группы с определёнными специфичными характеристиками. Объекты могут быть объединены в «корзины» [2] или сегменты разного типа, на данный момент поддерживаются унифицированные сегменты, где элементы хранятся в виде простого набора массива элементов. Сущности в такой корзине расположены последовательно, возможен доступ по индексу.

Для текущей разработки был выбран именно этот тип корзин за счет своей простоты и удобства. Добавление новых типов сегментов в текущей архитектуре не представляет сложностей. Для этого необходимо, во-первых, реализовать саму структуру данных для хранения, во-вторых, дополнить файл *bucket.c* функцией инициализации корзины нового типа и функцией *crush* для этого типа. Такой способ расширения используется как один из принципов чистой архитектуры [23].

Карта кластера необходима для использования алгоритма CRUSH и детальной настройки системы. С её помощью можно описать вычислительную сеть. Например, указать, что некоторые сервера находятся в одной стойке, набор стоек расположен в одной комнате в конкретном дата-центре. Такой подход позволяет управлять возможными рисками и лучше распределять нагрузку.

2.3 Клиентское приложение

Компонент представляет собой запускаемый на стороне клиента процесс, который производит основные операции с данными, то есть, так называемую, четверку CRUD операций [24]: создание, чтение, модификация, удаление. Клиент, как и другие компоненты системы, хранит карту кластера, которая была описана в предыдущей части. По данной карте и при помощи алгоритма CRUSH [2] он определяет, с каким устройством хранения ему нужно взаимодействовать. То есть, на какое хранилище можно инициировать запись или какое хранилище является ведущим для данного файлового объекта, чтобы выполнить одну из трёх других операций.

Клиент в любой момент времени взаимодействует не более чем с одним монитором, но это могут быть разные мониторы в зависимости от состояния их работоспособности.

2.4 Монитор

Данный компонент представляет собой вспомогательный процесс, который выступает промежуточным звеном между клиентом и хранилищем. Задача монитора состоит в отслеживании работоспособности кластера или, конкретнее, проверке состояния устройств хранения. В отдельном потоке монитор постоянно, с некоторой задержкой, опрашивает все известные ему устройства хранения, внося соответствующую информацию в карту кластера. Изменение состояния (одно из «работает», «не работает», «неизвестно») любого из устройств приводит к увеличению версии карты, что, в свою очередь, потребует обновления её локальной копии на клиентах и в хранилище.

2.5 Устройство хранения

Данный компонент является главным элементом всей системы. Устройства хранения (по аналогии с [3] в реализации названные OSD) выполняют основные функции всего кластера, именно они отвечают за сохранение данных и поддержание избыточности. Помимо обработки запросов от клиентов, они также взаимодействуют друг с другом. Это необходимо для репликации информации, хранимой в кластере.

Каждое устройство хранения представляет собой процесс, запущенный на отдельной вычислительной физической или виртуальной машине. Каждому такому устройству должен соответствовать отдельный физический жесткий диск, но это требование не обязательно.

Информация, получаемая сервером от клиента, хранится в оперативной памяти. Для хранения объектов используется красно-черное дерево. Это позволяет более эффективно работать с данными и обрабатывать запросы клиентов.

ЗАКЛЮЧЕНИЕ

Постоянно увеличиваются объёмы информации, которую необходимо обрабатывать и хранить. При этом становится недостаточно хранить все данные в одном месте, во-первых, это приводит к появлению единой точки отказа — если такое устройство выйдет из строя, информация либо будет недоступна в течение неопределённого времени, либо бесследно исчезнет. Во-вторых, при увеличении количества пользователей увеличивается и количество запросов для выполнения операций с хранимыми данными, увеличиваются задержки. Это также может стать критичным для приложений, в которых важно время отклика и выполнения запросов.

В рамках данной работы было разработано полностью функционирующее приложение для распределённого хранения данных, поддерживающее репликацию, способное в автоматическом режиме реагировать на отключение устройств и потерю с ними соединения. В разработанном комплексе отсутствуют единые точки отказа, так как нет централизованного хранения информации или метаданных, содержащих информацию о кластере. Помимо восстановления устройств хранения, поддерживается дублирование мониторов, способствующее увеличению надёжности всего программного комплекса.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 *Лесковец, Ю.* АНАЛИЗ БОЛЬШИХ НАБОРОВ ДАННЫХ / Ю. Лесковец, Д. Ульман, А. Раджараман. — Москва: ДМК Пресс, 2016. — 498 с.

2 *Weil, S.* Crush: Controlled, scalable, decentralized placement of replicated data / S. Weil, S. Brandt, E. Miller, C. Maltzahn // *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. — 2006. — no. 6. — P. 31.

3 *Weil, S.* Ceph: A scalable, high-performance distributed file system. / S. Weil, S. Brandt, E. Miller, D. Long, C. Maltzahn // *7th Symposium on Operating Systems Design and Implementation*. — 2006. — no. 7. — Pp. 307–320.

4 *Honicky, R. J.* Replication under scalable hashing: a family of algorithms for scalable decentralized data distribution / R. J. Honicky, E. L. Miller // *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings*. — 2004. — P. 96.

5 *Воеводин, В. В.* Параллельные вычисления / В. В. Воеводин, В. В. Воеводин. — СПб: БХВ-Петербург, 2002. — С. 608.

6 *Таненбаум, Э.* Распределенные системы. Принципы и парадигмы / Э. Таненбаум, М. ван Стеен. — Питер, 2003. — С. 877.

7 *Кормен, Т. Х.* Алгоритмы. Построение и анализ. Третье издание / Т. Х. Кормен, Ч. И. Лейзерсон, Р. Л. Ривест, К. Штайн. — Москва: Вильямс, 2015. — С. 1323.

8 Farsite: Federated, available, and reliable storage for an incompletely trusted environment / A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell et al. // *SIGOPS Oper. Syst. Rev.* — 2003. — Vol. 36, no. SI. — P. 1–14.

9 *Corbett, P. F.* The vesta parallel file system / P. F. Corbett, D. G. Feitelson // *ACM Trans. Comput. Syst.* — 1996. — Vol. 14, no. 3. — P. 225–264.

10 *Nieuwejaar, N.* The galley parallel file system / N. Nieuwejaar, D. Kotz // *Parallel Comput.* — 1997. — Vol. 23, no. 4–5. — P. 447–476.

11 *Latham, R.* A next-generation parallel file system for linux cluster. / R. Latham, N. Miller, R. Ross, P. Carns, C. Univ // *LinuxWorld Mag.* — 2004. — Vol. 2, no. 1.

12 *Cabrera, L.* Swift: Using distributed disk striping to provide high i/o data rates / L. Cabrera, D. D. Long // *Computing Systems*. — 1991. — no. 4(4). — Pp. 405–436.

13 *Сорокин, В. Е.* Репликация данных в иерархических информационных системах с непостоянной связностью узлов / В. Е. Сорокин // *Программные продукты и системы*. — 2015. — № 4 (112). — С. 203–209.

14 *Иванов, И. А.* Динамическая репликация данных для обеспечения надежности в облачных центрах данных / И. А. Иванов // *Научный потенциал молодежи и технический прогресс*. — 2018. — С. 34–35.

15 *Шаталова, Ю. Г.* Разработка системы репликации для распределенной базы данных предприятия / Ю. Г. Шаталова, Я. В. Жиглов // *Символ науки*. — 2017. — Т. 2, № 3. — С. 137–137.

16 *Шмелев, В. В.* Алгоритмы синхронизации ненормализованных баз данных с использованием хэш функций / В. В. Шмелев, А. В. Родионов // *Сборник статей по материалам XV международной научно-практической конференции. В 3 частях*. — 2018. — Т. 1, № 3. — С. 22–25.

17 *Шнайер, Б.* Прикладная криптография. Протоколы, алгоритмы, исходные тексты на языке Си / Б. Шнайер. — Москва: Издательство Триумф, 2003. — 816 с.

18 *Борсук, Н. А.* Анализ методов репликации баз данных при разработке онлайн-сервиса / Н. А. Борсук, О. О. Козеева // *Символ науки*. — 2016. — № 11-3. — С. 39–41.

19 *Ghemawat, S.* The google file system / S. Ghemawat, H. Gobioff, S.-T. Leung // *ACM SIGOPS Operating Systems Review*. — 2003. — Vol. 37. — Pp. 29–43.

20 *Welch, B.* Managing scalability in object storage systems for hpc linux clusters. / B. Welch, G. Gibson // *21st IEEE Conference on Mass Storage Systems and Technologies*. — 2004. — Pp. 433–445.

21 A cost-effective, high-bandwidth storage architecture / G. Gibson, D. Nagle, K. Amiri, J. Butler, F. Chang, H. Gobioff, C. Hardin et al. // *ACM SIGPLAN Notices*. — 1998. — Vol. 33. — Pp. 92–103.

22 *Немет, Э.* UNIX И LINUX РУКОВОДСТВО СИСТЕМНОГО АДМИНИСТРАТОРА, 4-у изд. / Э. Немет, Г. Снайдер, Т. Хейн, Б. Уэйли. — Москва: Вильямс, 2012. — 1312 с.

23 *Мартин, Р.* Чистая архитектура. Искусство разработки программного обеспечения / Р. Мартин. — СПб: Питер, 2019. — 352 с.

24 *Martin, J.* Managing the Data-base Environment / J. Martin. — Prentice-Hall, 1983. — 766 pp.