Оглавление

[Введение 4](#_Toc389108491)

[1. Обзор предметной области 5](#_Toc389108492)

[1.1. Системы хранения данных 5](#_Toc389108493)

[1.2. Мультиагентные технологии 5](#_Toc389108494)

[1.3. Платформа для моделирования 6](#_Toc389108495)

[2. Постановка задачи 8](#_Toc389108496)

[3. Математические основы и алгоритм 9](#_Toc389108497)

[4. Реализация 12](#_Toc389108498)

[4.1. Моделирование времени 12](#_Toc389108499)

[4.2. Централизованная работа с хранимыми данными 14](#_Toc389108500)

[4.3. Динамическое задание топологии графа мультиагентной системы, задержек и помех 15](#_Toc389108501)

[4.4. Проверка целостности данных 18](#_Toc389108502)

[5. Имитационные эксперименты по выбору 20](#_Toc389108503)

[6. Апробация 22](#_Toc389108504)

[Заключение 24](#_Toc389108505)

[Список литературы 25](#_Toc389108506)

# 

# 

# Введение

Мультиагентные технологии находят все более широкое применение в различных сферах деятельности. Одним из возможных применений могут быть системы хранения данных (СХД). В распределенных СХД появляется проблема передачи данных между устройствами, если они расположены далеко друг от друга или соединены медленными и не надежными каналами связи. В работе рассматриваются системы хранения данных, построенные по технологии RAID.

RAID (*redundant array of independent disks*) – это массив из нескольких устройств хранения данных, связанных между собой скоростными каналами передачи информации и воспринимаемых внешней системой как единое целое. Такой массив должен сохранять целостность данных при выходе из строя одного или нескольких устройств, а также обеспечивать высокую скорость чтения или записи данных. Каждое устройство хранения данных можно воспринимать как полностью независимый объект, находящийся сколь угодно далеко от других устройств. В этом контексте RAID-массив можно представить как мультиагентную систему, где в качестве одного агента выступает одно устройство хранения данных. Причем некоторые агенты либо вообще не могут общаться друг с другом, либо общаются по медленным и не надежным каналам передачи данных, а также только в определенные моменты времени.

Примером распределенной RAID-подобной СХД с ненадежными связями между устройствами хранения данных может служить группа из беспилотных летательных аппаратов (БПЛА). Каждый БПЛА выступает в качестве одного устройства хранения данных. Передача данных между двумя БПЛА возможна только тогда, когда они подлетают близко друг к другу. При выходе из строя нескольких БПЛА в дальнейшем возможно восстановление данных, которые были на них.

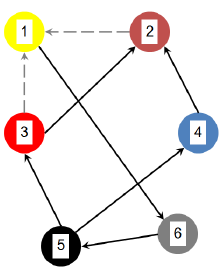


Рис 1. Топология связей шести агентов с переменными ребрами 1-3 и 2-1.

Используя алгоритм локального голосования в мультиагентных системах можно посчитать контрольные суммы на устройствах хранения данных с произвольной топологией связей между устройствами, в условиях передачи данных с помехами и задержками [1-3]. На рис. 1 показан возможный пример топологии связей с переменными ребрами 1-3 и 2-1.

# 

# 1. Обзор предметной области

## 1.1. Системы хранения данных

*Системой хранения данных* (СХД) называется совокупность оборудования и программного обеспечения, предназначенная для хранения больших объемов информации и бесперебойного доступа к ней. Одновременное использование нескольких жестких дисков в СХД позволяет увеличить объем хранимой информации и распараллелить чтение и запись данных, ускоряя тем самым процесс работы с ними [11, 12].

При работе с СХД необходимо иметь возможность восстановить утраченные данные в случае выхода из строя одного или более дисков. Достичь этого можно за счет систем резервного копирования или хранения избыточных данных на дисках. Высокая устойчивость к потере информации достигается полным резервным копированием («зеркало»), но такое решение проблемы влечет за собой понижение скорости работы системы.

Альтернативным решением проблемы может быть технология наподобие RAID 6. RAID 6 дополнительно хранит два диска с контрольными суммами и может восстанавливать до двух дисков с данными.

Одной из СХД, которая обеспечивает высокую целостность и быстрый доступ к данным, является СХД RAIDIX. В качестве RAID-контроллера там выступает компьютер с установленной операционной системой Linux. В Linux добавлен набор модулей, который и осуществляет поддержку конкретного уровня RAID. В частности, там есть отдельные модули, отвечающие за такие вычисления в рейде, как поиск сбойных данных и их восстановление.

Несмотря на то, что в СХД RAIDIX все устройства хранения данных соединены высокоскоростными и надежными каналами связи, принципы хранения контрольных сумм и восстановления данных на нескольких дисках после их поломки могут использоваться в любых СХД. Будем называть такие СХД – RAID - подобными.

## 1.2. Мультиагентные технологии

Суть мультиагентных технологий заключается в принципиально новом методе решения задач [13]. В отличие от классического способа, когда проводится поиск некоторого четко определенного (детерминированного) алгоритма, позволяющего найти наилучшее решение проблемы, в мультиагентных технологиях решение получается автоматически в результате взаимодействия множества самостоятельных целенаправленных программных модулей – так называемых агентов.

Агенты общаются между собой по каналам связи, которые можно изобразить в виде графа. Пример такого графа был показан на рисунке 1. Причем переменное ребро обозначает, что агенты могут общаться только в определенные моменты времени.

Примером применения мультиагентных технологий является использование алгоритма локального голосования, который позволяет достигать консенсуса в стохастической динамической сети с неполной информацией и задержками в измерениях [1].

## 1.3. Платформа для моделирования

Для моделирования использовалась платформа JADE [10]. Java Agent Development Framework (JADE) — программная среда разработки мультиагентных систем и приложений. Она включает в себя:

* Среду выполнения агентов. Агенты регистрируются и работают под управлением среды;
* Библиотеку классов, которые используются для разработки агентных систем;
* Набор графических утилит для администрирования и наблюдения за жизнедеятельностью активных агентов.

JADE полностью написана на языке программирования Java с использованием таких продвинутых возможностей как Java RMI, Java CORBA IDL, Java Serialization и Java Reflection API. Она упрощает разработку мультиагентных систем благодаря использованию FIPA-спецификаций и с помощью ряда инструментов, которые поддерживают фазы исправления ошибок и развертывания системы. Эта агентная платформа может распространяться среди компьютеров с разными операционными системами, и ее можно конфигурировать через удаленный GUI-интерфейс. Процесс конфигурирования этой платформы достаточно гибкий: ее можно изменить даже во время исполнения программ, для этого необходимо просто переместить агентов с одной машины на другую. Единственным требованием этой системы является установка на машине Java Run Time 1.2. Коммуникационная архитектура предлагает гибкий и эффективный процесс обмена сообщениями, где JADE создает очередь и управляет потоком ACL-сообщений (Agent Communication Language), которые являются приватными для каждого агента. Агенты способны обращаться к очереди с помощью комбинации нескольких режимов своей работы: блокирование, голосование, перерыв в работе и сопоставление с эталоном (что касается методов поиска). На данный момент в системе используется Java RMI, event-notification, и IIOP, но легко можно добавить и другие протоколы. Также предусмотрена возможность интеграции SMTP, HTTP и WAP. Большинство коммуникационных протоколов, которые уже определены международным сообществом разработчиков агентных сред, доступны и могут иллюстрироваться на конкретных примерах после определения поведения системы и ее основных состояний. SL и онтология управления агентами также имплементированы вместе с поддержкой определенных пользователем контентных языков, а также онтологий, которые могут быть имплементированы и зарегистрированы агентами и использованы системой. С целью существенного расширения работоспособности JADE предусмотрена возможность интеграции с JESS и Java-оболочкой CLIPS. JADE используется рядом компаний и академических групп. Среди них можно выделить такие, как: BT, CNET, NHK, Imperial College, IRST, KPN, University of Helsinky, INRIA, ATOS.

# 

# 2. Постановка задачи

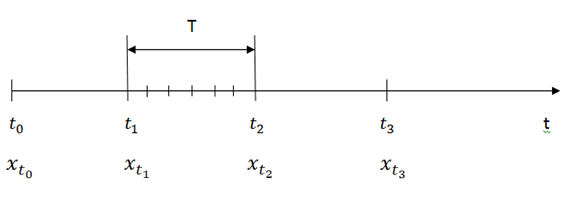
Целью данной работы является исследование возможности применения мультиагентных технологий для создания распределенных RAID-подобных систем хранения данных. Для достижения цели были выделены следующие задачи:

* На основе алгоритма локального голосования разработать алгоритм для подсчета контрольных сумм в RAID-массивах;
* На основе полученного алгоритма реализовать модель RAID-подобной распределенной СХД;
* На реализованной модели СХД произвести моделирование хранения данных в группе БПЛА и апробировать сценарий по восстановлению файлов на нескольких БПЛА.

# 3. Математические основы и алгоритм

Рассмотрим мультиагентную систему из n агентов. Пусть i, i = 1, … ,n – номер агента, t – время, N = {1, … ,n} – множество агентов, , - числа на агенте под номером i в момент времени . Будем вычислять и хранить m видов контрольных сумм от данных на агентах. В таком случае минимальное кол-во агентов, при котором сохраняется целостность данных, будет равно (n – m).

Для простоты будем считать, что данные передаются без помех, без задержек, и каждый агент хранит только одно число. Расчет контрольных сумм проходит в заданные интервалы времени. В случае успешного завершения всех вычислений на текущем интервале, в дальнейшем возможно восстановление тех данных, которые были записаны до его начала. В начале интервала происходит вычисление целочисленной функции от числа на агенте, и полученное значение сохраняется в заранее зарезервированной области памяти:



Далее по алгоритму локального голосования пересчитываются значения :

;

Cогласно [1] при определенных условиях:

По определению предела получаем:

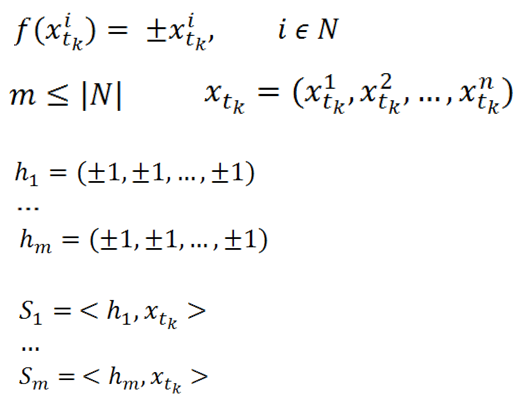
;

И если взять:

то так как все целые, то при округлении к ближайшему целому получим точное значение .

Таким образом, научились считать сумму чисел , где j = 1, .. , |N|.

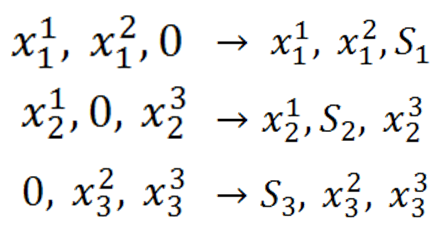
При изменении функции меняются числа , где j = 1, .. , |N|. Это позволяет по полученному алгоритму считать контрольные суммы разного вида от чисел , где i = 1, .. , |N|:



Если вектора , … , линейно независимые, то можно по контрольным суммам восстановить до m чисел на различных агентах.

После вычисления контрольной суммы на каждом агенте только один агент сохраняет её у себя, а остальные удаляют, чтобы не занимать лишнюю память. Для простоты рассмотрим случай контрольной суммы только одного вида.

В общем случае каждый агент хранит массив чисел. Таким образом, все числа можно представить в виде матрицы, у которой столбцы - это агенты, а строки - это ряды чисел по которым считаются контрольные суммы. На диагонали этой матрицы можно изначально хранить нули, а в дальнейшем хранить там контрольные суммы. Тогда возможно на каждом агенте хранить как данные, так и контрольные суммы:



Здесь верхний индекс у числа обозначает номер агента, на котором оно находится, а нижний индекс – это номер элемента в массиве у данного числа.

Для восстановления потерянных данных достаточно пересчитать все контрольные суммы, положив неизвестные числа равными нулю и решить систему линейных уравнений относительно неизвестных чисел.

# 4. Реализация

Для апробации предлагаемого алгоритма был реализован фреймворк, предоставляющий возможность гибкой настройки и задания параметров мультиагентной системы из N агентов на платформе JADE. Этот фреймворк применяется для запуска алгоритмов хранения и восстановления данных, в том числе в случае, когда система не статична и подвержена изменениям во времени. В частности, такая модель отражает поведение группы взаимодействующих БПЛА в режиме реального времени.

Составными частями данного фреймворка являются следующие компоненты:

* моделирование времени;
* централизованная работа с хранимыми данными;
* динамическое задание топологии графа системы, задержек и помех;
* проверка целостности данных.

## 4.1. Моделирование времени

Каждый агент в системе существует в непрерывном времени, и его состояние может быть описано в любой момент. Однако выбранный алгоритм подразумевает общение между агентами в дискретные моменты времени. Таким образом, нужно было поддержать возможность генерации временных тактов. Для этого можно было пойти двумя путями. Либо завести некоторый внешний механизм, который будет посылать такты всей системе. либо можно создать некоторого псевдоагента (будем называть его *тикер*), функцией которого будет посылка сигналов всем остальным агентам. При этом нужно, чтобы такты были согласованы между собой, то есть чтобы все агенты работали внутри одного такта, а не в разных.

Для этого для каждого агента в системе был заведен отдельный поток, который работает независимо от остальных. Потоки обмениваются информацией (числами, хранимыми в памяти агентов) и некоторой вспомогательной информацией, определяющей тип сообщения. Когда тикер генерирует очередной такт, всем агентам в системе посылается специальное сообщение, о том, что такт случился. После этого происходит обмен данными и метаинформацией, после чего агенты синхронизируются, и происходит проверка, что нет задержек на следующий такт. Это было реализовано при помощи добавления метода finished\_tick() в абстрактный класс агента. Этот метод реализуется для каждого наследника этого класса, и вызывается, когда агент завершил работу в текущем такте. При этом вызов метода finished\_tick() может зависеть от состояний других агентов (обработана ли вся информация). Поэтому в реализации этого метода были использованы мониторы Java для синхронизации потоков объектов [4]. Общая схема представлена на рис. 2.

Например, вот так может выглядеть код на Java, описывающий синхронизацию агентов, в том случае, когда для завершения текущего агента, требуется завершение его соседей:

private void barrier(Agent agent) {

synchronized (agent.sync) {

try {

while (agent.finished\_tick() == false) {

agent.sync.wait();

}

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

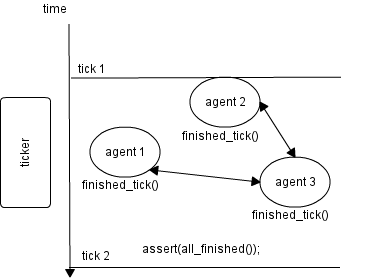


Рис 2. Синхронизация агентов внутри одного такта

Проверки таких инвариантов, как не превышение времени исполнения внутри такта, сделаны с помощью встроенных средств Java (начиная с Java версии 1.4) – assert [5]. Чтобы активировать эти проверки, нужно запускать Java c опцией -ea.

Примером может послужить проверка на то, что все агенты завершились вовремя:

assert (finished\_all == true) && (current\_time < next\_tick)

## 4.2. Централизованная работа с хранимыми данными

Как говорилось выше, в памяти каждого агента хранятся некоторые данные.  В модели их можно представлять массивом действительных чисел. Данные имеют в начальный момент времени (до того как система активизируется) некоторые начальные значения, которые в ходе работы агентов как-то изменяются. Для моделирования этого поведения память каждого агента представлена отдельным файлом. Во-первых, это наиболее близко соответствует реальной модели, в которой каждый агент имеет собственную память, а во-вторых, избавляет от потенциальных проблем с синхронизацией, когда различным агентам нужно модифицировать общий файл одновременно.

Файлы с данными хранятся в специальной директории, имя которой указывается параметром при запуске алгоритма. Каждый файл имеет следующую структуру. В самом верху файла указывается id агента, а начиная со следующей строки перечисляется массив данных. При старте системы происходит открытие файлов и сохранение дескрипторов в глобальный массив. Обращаться к файлам можно через интерфейс работы с дисками (памятью). Далее описаны функции из интерфейса:

* DiskLoad(id, start, array) - загружает в массив <array> указанное число элементов массива из файла агента с идентификационным номером <id>, начиная с элемента под номером <start>.
* DiskStore(id, start, array) - сохраняет в массив <array> элементы из файла агента с идентификационным номером <id>, начиная с элемента под номером <start>.
* DiskCreateZero(id, n) - создает для агента с идентификационным номером <id> диск размера <n>, заполненный нулями.
* DiskCreateRandom(id, n) - создает для агента с идентификационным номером <id> диск размера <n>, заполненный произвольными числами.
* DiskCorrupt(id) - выводит из строя диск для агента с идентификационным номером <id>.
* DiskRecovery(id) - восстанавливает данные на диске для агента с идентификационным номером <id>.

Все операции проводимые с дисками журналируются в специальный файл disk\_journal.output, в котором также указывается время начала и выполнения операции в тиках.

Так как файлы могут иметь довольно большой размер, постоянное обращение к файлам сильно замедляет работу системы. Поэтому для тех агентов, у которых больше всего связей, я использовал класс MappedByteBuffer [6]. Он позволяет мэпировать в оперативную память указанный участок файла. Соответственно, чем больше связей у агента, тем больше запросов он обрабатывает. Поэтому целесообразно сохранять эти данные в оперативную память, а не напрямую в файл. Это позволяет получить некоторый выигрыш в производительности системы в целом. В частности, это дает возможность уменьшить время между соседними тиками, за счет того, что высоконагруженные агенты стали меньше обращаться к диску.

Для замеров производительности также был реализован специальный класс AgentTimeMeasurement, который собирает статистику времени работы каждого агента внутри одного такта [8]. Также при помощи этого класса можно получить среднее и наибольшее время работы агента внутри такта.

Ниже приведен псевдокод, описывающий работу с файлами агентов.

for (int i = 0; i < agentsCount; ++i) {

DiskStore(i, startI, arrayI);

}

runAlgorithm();

DiskCorrupt(1);

DiskCorrupt(7);

DiskRecover(1);

DiskRecover(7);

assert `diff file\_result file\_template`

## 4.3. Динамическое задание топологии графа мультиагентной системы, задержек и помех

Как уже было сказано, система изменяется с течением времени. При этом изменяются не только связи между агентами, но и качество канала связи, что влечет за собой появление задержек и помех. Реализована возможность описания этих событий.

Описания динамического изменения топологии графа системы реализовано при помощи сервисов, которые представляют собой классы, реализующие изменение топологии в зависимости от времени. Эти классы реализуют метод transformTopology(), которые принимает на вход текущую топологию графа, а также номер текущего такта.

Можно сделать множество классов, предоставляющих различные реализации изменения топологии. Выбор того, какую именно реализацию использовать, реализовано при помощи механизма внедрения зависимости (dependency injection) [7]. Внедрение зависимости - это процесс предоставления внешней зависимости программному компоненту. Используя внедрение зависимости, объект просто предоставляет свойство, которое в состоянии хранить ссылку на нужный тип сервиса; и когда объект создается, ссылка на реализацию нужного типа сервиса автоматически вставляется в это свойство (поле), используя средства среды. В случае с топологией, ссылка на нужную реализацию сохраняется в объект тикера при его создании. Перед генерацией очередного такта, тикер вызывает метод соответствующего класса для изменения топологии графа.

Моделирование задержек и помех также производится тикером. Для этого он вызывает метод transformLink(), который вводит задержку или помеху для данного ребра в графе. Как и трансформация топологии, это реализовано при помощи внедрения зависимости. Задержки реализованы при помощи увеличения специального поля, соответствующего связи между агентами и означающего время от момента инициирования запроса на посылку данных и до момента реальной посылки. Помехи реализуется при помощи указания поля со значением помехи для каждой связи. По сути, помеха представляет собой некоторое число, которое будет прибавлено к посылаемому значению.

Так как связь может оборваться в тот момент, когда один агент отправил данные, а агент-получатель еще не принял, то использование стандартного метода JADE msgSend() не подходит, так как сообщение будет доставлено в любом случае. Поэтому для моделирования ситуации с обрывом связи в процессе передачи сообщения, все сообщения сохраняются в общую глобальную очередь для всех агентов. Таким образом, в момент, когда метод transformTopology() убирает связи в графе, автоматически модифицируется и очередь сообщений, удаляя те сообщения, которые не смогут быть доставлены.

Все  отброшенные сообщения, потерянные связи и помехи логируются в файл topology.output в той же директории, где находятся файлы дисков, в формате: <тип ошибки> : <связь> : <значение>, где

* <тип ошибки> - отброшенное ребро, сообщение, добавленная задержка или помеха
* <связь> - указание двух агентов, соединенных связью, на которой произошла ошибка
* <значение> - 0 или 1, если связь удалена/добавлена, значение величины задержки ли помехи.

На рис. 3 приведена схема взаимодействия процессов модификации топологии, задержек и помех.

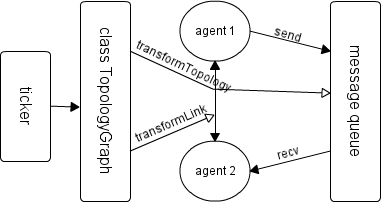


Рис 3. Модель воздействия внешних факторов на мультиагентную систему.

Ниже приведен фрагмент реализованного метода runAlgorithm(), используемого в разделе выше, с использованием изменений в топологии, задержек и помех.

transformCallback () {

tiker.transformTopology(currentTick, currentGraph);

tiker.transformLink(currentTick, currentGraph);

}

runAlgorithm() {

ticker.start(1000);

for (int i = 0; i < agentsCount; ++i) {

i.start();

}

callOnEachTick(transformCallback);

}

## 4.4. Проверка целостности данных

Проверка целостности данных происходит путем искусственного вывода из строя одного из дисков (файлов) с последующим восстановлением информации и сравнением с файлом, содержащим корректные данные. Это реализовано при помощи методов DiskCorrupt() и DiskRecovery(), описанными выше. Сравнение на идентичность осуществляется при помощи команды diff для файла с восстановленными данными и файла с эталонными данными. Момент вызова функции DiskCorrupt() может быть указан как время в тактах от начала исполнения при запуске алгоритма.

Реализован тестовый фреймворк, который запускает алгоритм с различными таймаутами для вызова метода DiskCorrupt, для различных агентов и для различного числа дисков, вышедших из строя.

За основу для написания тестового фреймворка был выбран Autotestnet [9]. Этот фреймворк используется для тестирования различных систем через удаленное соединение по сети, например по протоколу telnet. И каждое устройство представляется в этом фреймворке отдельным модулем (выглядит как обычная директория), к которому в дальнейшем добавляются тесты. Схема представления устройства отдельным модулем хорошо вписывается в концепцию мультиагентных систем, поэтому эту идея была позаимствована из Autotestnet.

При запуске этому фреймворку передается конфигурационный файл, в котором перечислены тесты в формате: <номер тика> : <номера дисков вышедших из строя>. Все тесты выполняются последовательно. В случае ошибки одного из тестов, выполнение прекращается. Также реализован pretty printer для вывода информации о выполнении тестов. Ниже приведен пример файла и пример вывода.

Файл конфигурации:

10 : 1

100 : 3

1000 : 4 5

Результат исполнения:

test case “10 : 1” passed (3 ticks link disabled)

test case “100 : 3” passed (0 times link disabled)

test case “1000 : 4 5” failed (0 times link disabled)

See statistic in the file recovery.output

Также отслеживаются обрывы всех связей определенного агента, то есть те такты, в которые агент изолирован от остальных агентов, а значит, не может обмениваться информацией. Если такой агент выходит из строя, то восстановить его данные не представляется возможным. Поэтому такие ситуации фиксируются в файле errors.output, а также число тактов, проведенных в изоляции, сообщается при выводе результатов теста. В вышеприведенном примере это строка “3 ticks link disabled”, что означает, что агент 1 был изолирован в течение трех тактов в промежутке с 0 по 10-ый такт.

В целом структура реализованного фреймворка представлена на рисунке 4,



Рис 4. Архитектура фреймворка

где CONFIG - это самый верхний уровень, который отвечает за конфигурацию мультиагентной системы. CORE - это основополагающая часть, в которой сосредоточена логика работы алгоритма. TRANSFORMATION - это часть, отвечающая за динамику изменения системы. И наконец, TEST - это одна из важнейших составляющих, позволяющая тестировать алгоритм.

# 

# 5. Имитационные эксперименты по выбору

Ниже представлены графики зависимости времени (в тактах) подсчета контрольных сумм от количества агентов при разных топологиях связей без помех и без задержек. Параметр размера шага в алгоритме локального голосования был выбран равным 0.1. Числа на агентах выбирались случайно в диапазоне от 1 до 16000.

На рис. 5 приведены графики при фиксированных топологиях типа «звезда» и «кольцо».

Рис. 5. График зависимости времени подсчета контрольных сумм при фиксированных топологиях типа «звезда» и «кольцо»

В случае подсчета контрольных сумм без использования алгоритма локального голосования нужно было бы передавать данные от каждого к каждому. В общем случае при постоянных топологиях это потребовало бы тактов, где n – кол-во агентов.

Из графика видно, что время подсчета контрольных сумм при использовании мультиагентных технологий гораздо меньше чем и зависимость близка к линейной.

При выполнении групповых заданий БПЛА, как правило, могут общаться между собой только попарно и в определенные моменты времени.

На рисунках шесть и семь показаны возможные варианты переменных топологий связей у 4 БПЛА при выполнении групповых заданий. Назовем их топологии 1 и 2 соответственно.

Рис. 6. Периодическое изменение во времени топологии 1

Рис. 7. Периодическое изменение во времени топологии 2

На Рис. 8 приведены графики при переменных топологиях 1 и 2.

Рис. 8. График зависимости времени подсчета контрольных сумм при при переменных топологиях 1 и 2

Из графика видно, что использование топологии 1 для общения БПЛА предпочтительнее, чем использование топологии 2. А также можно заметить, что зависимость опять близка к линейной.

# 6. Апробация

Тестирование на реальных аппаратных платформах всегда сложнее, чем на модели, симуляторе. Причина в том, что симуляторы и модели учитывают лишь ограниченное число свойств реальной системы. Однако тестировать ПО сразу на реальной целевой платформе зачастую не рационально, так как это требует денежных затрат на покупку оборудования, а также вызывает сложности с настройкой необходимой конфигурации. Предложенный фреймворк позволяет удобно и гибко писать тесты для модельной системы, и таким образом находить ошибки на наиболее ранних стадиях тестирования, что позволяет снизить риски и затраты с реальным оборудованием. Данный фреймворк можно использовать полигона для тестирования алгоритмов хранения данных в мультиагентной системе с реальными БПЛА.

Каждый отдельный БПЛА представляется агентом в данной системе. Из всех его частей для моделирования требуется лишь память, которая представляется файлом с массивом данных, как уже было рассказано выше. Остальные свойства БПЛА значения не имеют. Помимо самих БПЛА, существует также среда, в которой они взаимодействуют. Эта среда вносит некоторые сложности во взаимодействие БПЛА.

Во-первых, это взаимодействие БПЛА. То есть БПЛА имеют ограниченный радиус доступности по сети, ввиду чего связи между БПЛА иногда могут пропадать. Этот процесс обусловлен различными факторами и поэтому невозможно предоставить какую-то общую функцию для всех топологий. Поэтому у пользователя есть возможность описывать собственные трансформации.

Во-вторых, из-за неполадок с оборудованием, либо из-за программных задержек, сообщения между БПЛА могут передаваться не сразу. Такое поведение эмулируется при помощи модельных задержек, добавляемых к связям в графе агентов. Кроме того, то, что сообщение было передано с одного БПЛА, еще не гарантирует, что это сообщение будет доставлено до получателя. Эта ситуация также моделируется при помощи очереди “висячих” сообщений, в которую агенты добавляют сообщения. После этого эти сообщения могут быть отброшены при помощи пользовательского метода.

В-третьих, при передаче данные могут быть искажены. Это моделируется при помощи добавления некоторого числа к передаваемым данным.

Разработанный фреймворк использовался для тестирования на встречающихся на практике топологиях и на разном числе агентов.

Предложенный алгоритм был апробирован на разных мультиагентных системах с различными топологиями и различным числом агентов. Тесты проводились при помощи предложенного тестового фреймворка, описанного в разделе “Проверка целостности данных”.

Для тестов были созданы 10 агентов, и соответственно 10 файлов с данными, которые размещаются в директории config/. Далее была реализована функция изменения топологии, которая убирает одну произвольную связь на 10 тактов каждые 100 тактов. После этого был создан файл с 50 тестами tests.in, в котором разместил тесты в формате, описанном выше. Эти тесты проверяли корректность восстановления данных в случае выхода из строя одного, двух и трех агентов.

Проведенные тесты позволили выявить и исправить несколько ошибок в алгоритме.

Система была апробирована при хранении на ней и восстановлении после потери фотографий местности полученных на борту беспилотных летательных аппаратов (БПЛА).

# Заключение

# В рамках дипломной работы на основе алгоритма локального голосования был разработан алгоритм подсчета контрольных сумм в RAID-подобных массивах с ненадежными и медленными каналами связи между устройствами хранения данных. Для тестирования системы был реализован фреймворк, который моделирует работу распределенной RAID-подобной СХД и позволяет гибко задавать переменную топологию связей, помехи и задержки при передаче данных. С помощью разработанного фреймворка были проведены замеры времени подсчета контрольных сумм для разных топологий связи, в том числе тех, что могут быть у БПЛА при выполнении групповых заданий. Система была апробирована при хранении на ней и восстановлении после потери фотографий местности полученных на борту беспилотных летательных аппаратов (БПЛА).

# 

# 

# Список литературы

1. *Амелина Н.О., Фрадков А.Л.* [Приближенный консенсус в стохастической динамической сети с неполной информацией и задержками в измерениях](http://elibrary.ru/item.asp?id=18236867) // [Автоматика и телемеханика](http://elibrary.ru/contents.asp?issueid=1075912). 2012. [№ 11](http://elibrary.ru/contents.asp?issueid=1075912&selid=18236867). С. 6-29.
2. *Амелина Н.О.* [Применение протокола локального голосования для децентрализованной балансировки загрузки сети с переменной топологией и помехами в измерениях](http://elibrary.ru/item.asp?id=20268200) *//* [Вестник Санкт-Петербургского университета. Серия 1: Математика. Механика. Астрономия](http://elibrary.ru/contents.asp?issueid=1141327). 2013.[№ 3](http://elibrary.ru/contents.asp?issueid=1141327&selid=20268200). С. 12-20.
3. *Amelina N., Granichin O., Kornivetc A.* [Local voting protocol in decentralized load balancing problem with switched topology, noise, and delays](file:///C:\Distrib\Учеба\Диплом\Отчет\papers\1495CDC12.pdf)// In: Proc. of the 52nd IEEE Conference on Decision and Control, December 10-13, 2013, Firenze, Italy. P. 4613-4618.
4. Synchronization in Java. URL : <http://docs.oracle.com/javase/tutorial/essential/concurrency/sync.html>
5. Programming with assertions. URL : <http://docs.oracle.com/javase/7/docs/technotes/guides/language/assert.html>
6. Memory-Mapped Files. URL : <http://www.yaldex.com/java_tutorial/0600410861.htm>
7. Dependency injection. URL : <http://en.wikipedia.org/wiki/Dependency_injection>
8. JETM. URL : <http://jetm.void.fm/>
9. Autotestnet. URL : <http://autotestnet.sourceforge.net/doc_user_guide_v2_0.html>
10. Документация JADE (Java Agent Development Environment) URL : <http://jade.cselt.it>
11. *Somasundaram G., Shrivastava A.* Information Storage and Menagement. Published by Wiley Publishing, Inc., Indianapolis, Indiana, 2009. 478 с.
12. *Troppens U., Muller-Friedt W., Wolafka R., Haustein N., Erkens R.* Storage Networks Explained. Second Edition. Published by John Wiley & Sons Ltd, 2009. 602 с.
13. *Амелина Н.О. ,* Мультиагентные технологии, адаптация, самоорганизация, достижение консенсуса // Стохастическая оптимизация в информатике 2011. Т. 7. [№ 1](http://elibrary.ru/contents.asp?issueid=1075912&selid=18236867). С. 9-13.