МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ, МИНИСТЕРСТВО НАУКИ, ВЫСШЕГО ОБРАЗОВАНИЯ И ИННОВАЦИЙ КЫРГЫЗСКОЙ РЕСПУБЛИКИ

МОО ВО Кыргызско-Российский Славянский университет
имени первого Президента Российской Федерации Б.Н. Ельцина

УТВЕРЖДАЮ
Зам. декана факультета ИТ
_____ Н. М. Комаров
22.09.2025

Проектирование распределенных информационных систем

рабочая программа дисциплины (модуля)

Закреплена за кафедрой **Информационных и вычислительных технологий**

Учебный план g090404_25_1пи_рпис.plx
Направление подготовки 09.04.04 - РФ, 710400 - КР Программная инженерия
Магистерская программа "Разработка программно-информационных систем"

Квалификация **магистр**

Форма обучения **очная**

Общая трудоемкость **4 ЗЕТ**

Часов по учебному плану 144
в том числе:
аудиторные занятия 38
самостоятельная работа 103,8

Виды контроля в семестрах:
зачет с оценкой 2
курсовая работа 2

Распределение часов дисциплины по семестрам

Семестр (<Курс>.<Семестр на курсе>)	**2 (1.2)**		Итого	
Недель	18			
Вид занятий	УП	РП	УП	РП
Лекции	16	16	16	16
Практические	22	22	22	22
Контактная работа в период теоретического обучения	2,2	2,2	2,2	2,2
В том числе инт.	8	8	8	8
В том числе в форме практ.подготовки	22	22	22	22
Итого ауд.	38	38	38	38
Контактная работа	40,2	40,2	40,2	40,2
Сам. работа	103,8	103,8	103,8	103,8
Итого	144	144	144	144

Программу составил(и):
ст. преп. , Беляев А.А.;к.т.н. , доцент, Хмелева И.В. _____

Рецензент(ы):
д.т.н., профессор, Лыченко Н.М. _____

Рабочая программа дисциплины
Проектирование распределенных информационных систем

разработана в соответствии с ФГОС 3++:

Федеральный государственный образовательный стандарт высшего образования - магистратура по направлению подготовки 09.04.04 Программная инженерия (приказ Минобрнауки России от 19.09.2017 г. № 932)

составлена на основании учебного плана:

Направление подготовки 09.04.04 - РФ, 710400 - КР Программная инженерия
Магистерская программа "Разработка программно-информационных систем"
утвержденного учёным советом вуза от 30.06.2025 протокол № 13.

Рабочая программа одобрена на заседании кафедры
Информационных и вычислительных технологий

Протокол от 03.09.2025 г. № 1
Срок действия программы: 2025-2027 уч.г.
Зав. кафедрой д.т.н. проф. Лыченко Н.М.

ЛИСТ РЕГИСТРАЦИИ ИЗМЕНЕНИЙ
к рабочей программе дисциплины (модуля)

Проектирование распределенных информационных систем

наименование дисциплины

№ п/п	Основание внесения изменения	Раздел/ пункт	Содержание изменения
1.	№357-П от 24 июня 2026- О преобразовании Естественно-технического факультета и Факультета архитектуры дизайна и строительства в Политехнический институт КРСУ	Титульный лист. Пункт Разработано	Заменить слова «Разработано Кафедра Информационных и вычислительных технологий» словами «Разработано: Высшая школа Цифровых технологий»
2.			
3.			
4.			
5.			

1. ЦЕЛИ ОСВОЕНИЯ ДИСЦИПЛИНЫ	
1.1	Целью изучения дисциплины «Проектирование распределенных информационных систем» является подготовка специалистов, обладающих фундаментальными знаниями и практическими навыками в области построения распределенных информационных систем и сетей, программной инженерии, теории и практики руководства проектами по созданию распределенных информационных систем.

2. МЕСТО ДИСЦИПЛИНЫ В СТРУКТУРЕ ООП	
Цикл (раздел) ООП:	Б1.В
2.1	**Требования к предварительной подготовке обучающегося:**
2.1.1	Дисциплины ООП бакалавриата по направлению 09.03.04 ПРограммная инженерия"
2.2	**Дисциплины и практики, для которых освоение данной дисциплины (модуля) необходимо как предшествующее:**
2.2.1	Тестирование и обеспечение качества программных систем, Методы сопровождения ПО, Проектирование высоконагруженных систем, Преддипломная практика

3. КОМПЕТЕНЦИИ ОБУЧАЮЩЕГОСЯ, ФОРМИРУЕМЫЕ В РЕЗУЛЬТАТЕ ОСВОЕНИЯ ДИСЦИПЛИНЫ (МОДУЛЯ)	
ПК-3: Владение методами программной реализации распределенных информационных систем	
Знать:	
Уровень 1	основные технологии построения распределенных систем: процедурные подходы, объектные и компонентные модели; базовые шаблоны проектирования распределенных систем.
Уметь:	
Уровень 1	применить изученные методы и алгоритмы на практике в процессе разработки реальных распределенных программных систем
Владеть:	
Уровень 1	практическими навыками работы с современными системами проектирования и разработки распределенных систем

В результате освоения дисциплины обучающийся должен

3.1	**Знать:**
3.1.1	- базовые принципы построения распределенных систем: синхронизация времени и логические часы, распределенные транзакции, взаимодействие по сети и Удаленный Вызов Процедур.
3.2	**Уметь:**
3.2.1	- применять механизмы и средства современных операционных систем, используемых для построения распределенных приложений,
3.2.2	- разрабатывать приложения в Map Reduce,
3.2.3	- создавать сетевые файловые системы.
3.3	**Владеть:**
3.3.1	навыками применения инструментальных программных средств, применяемые для построения распределенных систем.

4. СТРУКТУРА И СОДЕРЖАНИЕ ДИСЦИПЛИНЫ (МОДУЛЯ)

Код занятия	Наименование разделов и тем /вид занятия/	Семестр / Курс	Часов	Компетен-ции	Литература	Инте ракт.	Пр. подг.	Примечание
	Раздел 1. Базовые знания по распредленным системам							
1.1	Распределенные систмы. Map Reduce. Сетевые файловые системы. Взаимодействие по сети и Удаленный Вызов Процедур. Синхронизация времени и логические часы. /Лек/	2	2			2		
1.2	Удаленный вызов процедур и потоки /Пр/	2	2				2	
1.3	Echo Server /Ср/	2	18					
	Раздел 2. Отказоустойчивость							

2.1	Первичный бэкап. Двух-фазовый комит, введение в безопасность и отказоустойчивость /Лек/	2	2					
2.2	Согласованность данных : FLP невозможность, Paxos, Реплицируемые машины состояний, Raft. Отказоусточивость Byzantine /Лек/	2	2					
2.3	Первичный бэкап. Двух-фазовый комит. Реплицируемые машины состояний, Raft /Пр/	2	6			2	6	
2.4	Map Reduce /Ср/	2	16					
	Раздел 3. Масштабируемость, согласованность данных и транзакции							
3.1	Пиринговые системы и распредленные хэш таблицы. Возможнось согласованности данных. Масштабрируемые сервисы: Key-Value хранилища. /Лек/	2	2					
3.2	Масштабрируемые сервисы: Key-Value хранилища. /Пр/	2	6				6	
3.3	Сильная согласованность данных и теорема CAP. Причинная согласованность. Контроль конкурентности, блокировки, и восстановление, (OCC, MVCC) и распределенные транзакции. Разрешение конфликтов, Криптография, Недоверенные облачные сервисы /Лек/	2	2					
3.4	Raft /Ср/	2	24					
3.5	Контроль конкурентности, блокировки, и восстановление, (OCC, MVCC) и распределенные транзакции. Разрешение конфликтов /Пр/	2	4				4	
3.6	Отказоустойчивый Key/Value сервис /Ср/	2	24					
3.7	Blockchains. Сети дотавки контента. Distributed Mesh Wireless Networks. /Лек/	2	2			2		
	Раздел 4. Обработка больших данных							
4.1	Graph processing. Обработка потоков. Управление расписанием кластеров /Лек/	2	4					
4.2	Обработка потоков. Управление расписанием кластеров /Пр/	2	4			2	4	
4.3	Шардированый Key/Value сервис /Ср/	2	21,8					
4.4	Обработка потоков /КрТО/	2	2,2					

5. ФОНД ОЦЕНОЧНЫХ СРЕДСТВ

5.1. Контрольные вопросы и задания

Контрольные вопросы для проверки уровня обученности Знать:
1. Цели распредленных систем
2. Технология Map-Reduce
3. Назначение распредленных файловых систем NFS, AFS
4. Синхронизация данных в AFS, NFS и согласованность
5. Архитектура NFS
6. NFS с состоянием и без
7. Кэширование в NFS
8. Различные уровни взаимодействия в сети
9. RPC - архитектура
10. Схемы согласовнности (At-Least-Once, At-Most-Once, Exactly-once)
11. Синхронизация времени в распределенных системахю Wall Clock Time
12. Синхронизация времени в распределенных системахю Lamport
13. Синхронизация времени в распределенных системахю Vector
14. Первичный бэкап. Цели, Машина состояний, Атомарные операции
15. Отказоустойчивость. Jосновные вехи
16. Безопасность и живучесть распредленных систем
17. Распределенные транзакции. Двухвазовый комит
18. Консенсус в распределенных системахю Модели распредленных систем
19. Консенсус FLP
20. Paxos
21. Srawman
22. Raft
23. Системы точка-точка. Рапспредленные Хэш таблицы. Сервисы поиска
24. Отказоустойчивость систем точка-точка
25. Возможная согласованность. Bayou
26. Масштабирование систем. Партиционирование. Хэши. Храннилища Ключ/Значение
27. Сильная согласованность данных и CAP теорема
28. Причинная согласованность
29. Контроль конкуренции, блокировки и восстановление
30. Биткоин и блокчейн
31. Сети распространения данных
32. Обработка графов больших данных
33. Обработка рапредленных потоковых данных

Контрольные вопросы для проверки уровня обученности Уметь-Владеть:
1. Программные интерфесы map-reduce
2. Оптимизация map-reduce
3. Отказоустойчивость map-reduce
4. Реализация основных компонентов map-reduce
5. Натройка NFS в сети
6. Безопасность в NFS
7. Согласованность кэша NFS
8. Сокеты как средство коммуникации
9. Реализация RPC и схемы согласованности
10. Синзронизация времени. Протоколы
11. Хранение времени в распредленных системах
12. Реалищация первичного бэкапа в гипервизорах
13. Двухфазовый комит. Реализация.
14. Реализация Paxus
15. Реализация Raft
16. Реализация BitTorrent системы. Распредленное хранение и хэши

17. Партиционирование хранилищ ключ-значение
18. Рапределенные транзакции - (OCC, MVCC). Сериализуемость. Множество версий данных
19. Разрешение кофнликтов в распредленных системах. Критография
20. Реализация CDN для веб приложений
21. Планирование кластеров

5.2. Темы курсовых работ (проектов)

•Проекты с многопользовательским компонентом в режиме реального времени, которые подходят для реализации протокола Paxos. Можно использовать отдельных пользователей в кворуме Paxos, либо использовать Paxos для реплицированного состояния, в котором хранятся обновления состояний системы.
•Совместное редактирование документов в стиле документов Google. Система должна поддерживать редактирование и просмотр в режиме реального времени несколькими участниками. Для отказоустойчивости должны поддерживаться множественные реплики. Кэширование и/или миграция копий могут быть полезны для минимизации времени отклика приложения.
•Игры-симуляции, в стиле The Sims или Farmville. Состояние системы должно быть разделено пространственно, с репликацией для отказоустойчивости.
•Многопользовательская игра в режиме реального времени.
•Система бронирования авиабилетов. Каждая авиакомпания будет поддерживать собственную коллекцию серверов с достаточной репликацией состояния, чтобы обеспечить автоматическое переключение. Должна быть возможность забронировать путешествие, в котором участвуют несколько авиакомпаний.
•Система регистрации с малым временем задержки. Например, просмотр множества RSS-каналов и отправка всем подписчикам почтовых уведомлений при обновлении. Интерфейс взаимодействия как обычными RSS-каналами, так и с сервисом обновления новостей Google. Требуется реализация репликации и разбиения состояний системы мониторинга так, чтобы она могла масштабироваться и выдерживать сбои узлов.
•Проекты с участием множества агентов или пользователей, которые координируют свои обновления, и которые могут отображаться на карте (например, Карты Google). Можно интегрировать с Google Maps, Yandex Map, 2GIS и т.д.. API этих систем помогают упростить работу с графическим интерфейсом.
•Большие проекты, связанные с созданием приложений и игр для Android и iOS.
•Внедрение распределенной файловой системы, которая делает что-то интересное. Возможно, хранение MP3-файлов или фильмов. Или, возможно, для чего-то совершенно другого.

5.3. Фонд оценочных средств

Задания к практическим работам представлены в Приложении 1
Контрольные вопросы представлены в п.5.1

5.4. Перечень видов оценочных средств

Практические задания;
Контрольные вопросы.
Виды шкал оценивания представлены в Приложении 2.

6. УЧЕБНО-МЕТОДИЧЕСКОЕ И ИНФОРМАЦИОННОЕ ОБЕСПЕЧЕНИЕ ДИСЦИПЛИНЫ (МОДУЛЯ)

6.1. Рекомендуемая литература

6.1.1. Основная литература

	Авторы, составители	Заглавие	Издательство, год
Л1.1	В.В. Коваленко	Проектирование информационных систем: Учебное пособие: Основная	М.: Форум: НИЦ ИНФРА 2014
Л1.2	Олифер В., Олифер Н	Компьютерные сети. Принципы, технологии, протоколы.: Основная	Питер 2014
Л1.3	Олифер В. Г., Олифер Н. А.	Основы сетей передачи данных: Учебное пособие	Москва: Интернет-Университет Информационных Технологий (ИНТУИТ), Ай Пи Ар Медиа 2021

6.1.2. Дополнительная литература

	Авторы, составители	Заглавие	Издательство, год
Л2.1	Kenneth P Birman	Guide to Reliable Distributed Systems: Building High-Assurance Applications and Cloud-Hosted Services (Texts in Computer Science): Textbook	Springer; 2012 edition (January 16, 2012) 2012
Л2.2	Вендров	Проектирование информационных систем: Учебник	М., Финансы и статистика 2013
Л2.3	Фаулер М.	Шаблоны корпоративных приложений: Пер. с англ.	М.: Издательский дом "Вильямс" 2012

	Авторы, составители	Заглавие	Издательство, год
Л2.4	Кариев Ч. А.	Разработка Windows-приложений на основе Visual C#: Учебное пособие	Москва, Саратов: Интернет-Университет Информационных Технологий (ИНТУИТ), Вузовское образование 2017

6.2. Перечень ресурсов информационно-телекоммуникационной сети "Интернет"

Э1	MapReduce: Simplifed Data Processing on Large Clusters	https://www.cs.princeton.edu/courses/archive/fall16/cos41
Э2	The Design of a Practical System for Fault-Tolerant Virtual Machines	https://www.cs.princeton.edu/courses/archive/fall16/cos41
Э3	In Search of an Understandable Consensus Algorithm	https://www.cs.princeton.edu/courses/archive/fall16/cos41
Э4	Practical Byzantine Fault Tolerance	https://www.cs.princeton.edu/courses/archive/fall16/cos41

6.3. Перечень информационных и образовательных технологий

6.3.1 Компетентностно-ориентированные образовательные технологии

6.3.1.1	Изучение дисциплины студентами осуществляется в форме лекций, практических занятий в аудиторных условиях (компьютерные классы) и в процессе самостоятельной работы, контроля знаний.
6.3.1.2	Теоретическая информация представляется в виде компьютерных презентаций с использованием мультимедийных средств.
6.3.1.3	Практические занятия проводятся в компьютерных классах, оснащенных персональными компьютерами с необходимыми параметрами и с установленным необходимым программным обеспечением. Используется Интернет для получения дополнительной информации. Используется дискуссионный метод проведения занятий, где студенты могут высказать свое мнение по обсуждаемой проблеме.
6.3.1.4	Защита практических работ проводится в виде собеседования с преподавателем по теории и программной реализации работы.

6.3.2 Перечень информационных справочных систем и программного обеспечения

6.3.2.1	Программное обеспечение: Microsoft Visual Studio 2015/2017 Comunity Edition (C#).
6.3.2.2	Если по выбору учащегося используется другая среда разработки приложений на языке другом языке (например, C++, Go, Java, Python), то понадобятся среды под эти языки программирования.

7. МАТЕРИАЛЬНО-ТЕХНИЧЕСКОЕ ОБЕСПЕЧЕНИЕ ДИСЦИПЛИНЫ (МОДУЛЯ)

7.1	Компьютерный класс, аудитория с презентационным оборудованием.
7.2	Серверы для создания виртуальных машин под ОС Linux для развертываемых распредленных систем: 3x(32 Gb RAM, 6xHDD 2TB, i7(Xeon))

8. МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ОБУЧАЮЩИХСЯ ПО ОСВОЕНИЮ ДИСЦИПЛИНЫ (МОДУЛЯ)

1. Технологическая карта дисциплины представлена в Приложении 4.

2. Текущий контроль осуществляется в течение семестра в виде защиты практических работ.
Методические указания по выполнению практических работ представлены в электронной папке преподавателя (локальная сеть кафедры Информационных и вычислительных технологий КРСУ).
Подготовка к практическим работам предполагает самостоятельную работу студента по овладению навыками практического применения лекционного материала к решению задач моделирования на ЭВМ. Минимум требуемой подготовки для выполнения работы определен контрольными вопросами, которыми снабжена каждая работа. К выполнению работы допускаются студенты, письменно ответившие на контрольные вопросы.
Требования к выполнению практических работ
Каждая практическая работа выполняется в соответствии с индивидуальными заданиями.
В результате выполнения работы составляется отчет в письменной или электронной форме. Отчет содержит описание выполнения индивидуальных заданий, графики моделирования, выводы, а также соответствующие заданиям скрипт-файлы. Отчет должен быть сдан в конце последнего занятия по данной практической работе. Нарушение срока сдачи без уважительной причины влечет за собой снижение оценки за текущий модуль. Опережение срока сдачи лабораторной работы, естественно, приводит к поощрительным баллам.
Подробые методические указания к выполнению практических работ представлены в Приложении 1.

3. Рубежный контроль осуществляется в виде сдачи контрольных работ.
Вопросы по контрольным работам представлены в разделе РПД "Фонд оценочных средств". Контрольные работы оформляются в письменном виде.

4. Методические рекомендации к курсовой работе представлены в Приложении 3.

Задание 1. Реализовать мульти-клиентный эхо сервер.

Цель задания

Цель этого задания – улучшить навыки работы с языком C# и вспомнить основы работы с TCP сокетами. В задании надо реализовать простой мульти-клиентый эхо сервер на C#: каждое сообщение, посланное клиентом должно отобразиться у всех остальных клиентов.

Описание задания

Характеристики сервера

Ваш мульти-клиентый эхо сервер должен иметь следующие характеристики:

1. Сервер должен управлять и конкурентно взаимодействовать с клиентами используя TCP каналы (сокеты). Множество клиентов должны иметь возможность соединится/отсоединится от сервера одновременно.
2. Все сообщения, которые приходят на сервер разделяются знаком окончания строки (\n). Когда сервер получает сообщение, заканчивающееся символом (\n), он должен разослать тоже сообщение, включая символ новой строки всем присоединенным клиентам, включая того клиента, который прислал сообщение.
3. Сервер должен так же учитывать «медленно-читающих» клиентов. Для лучшего понимая, что это значит, представим следующий сценарий, в котором клиент не «читает» канал от сервера продолжительное время. Если сервер продолжит рассылать сообщения, то буфер отправки может достичь максимума и рассылка может прекратится.

   Для отслеживания этого случая, сервер должен сохранять очередь из последних 100 исходящих сообщений. Если буфер исходящих сообщений достигает 100, он должен быть сброшен. Если «медленно-читающий» клиент начинает читать сообщения, сервер должен послать все сообщения из очереди этому клиенту. (Можно использовать buffered channel или написать свою реализацию буфера).

Требования к программе

Это задание открыто для обсуждения, и может иметь множество решений. Но есть несколько пожеланий и требований:

- Проект должен быть выполнен индивидуально. Вы не должны использовать любой код, которые вы не написали.
- Вся синхронизация должна быть реализована с помощью TCP сокетов.
- Можно использовать следующие компоненты (возможно обсуждение технологий):
  - System.Net.Sockets.TcpListener,
  - System.Net.Sockets.TcpClient.
- Для оформления кода пользуйтесь стандартами, например, от Microsoft.
- Приложение клиента и сервера должны быть в одном Solution в Visual Studio. Если есть еще проекты или библиотеки с исходными кодами, то их нужно добавить как проекты в Solution. Но Solution должен быть ОДИН.

Не предполагается, что решение будет очень сложным. Данные приложение вряд ли будут превышать 500 строк кода. Но рекомендуется проработать конкурентную модель, перед тем как выполнять задание. Ссылки на дополнительные материалы можно найти в конце документа.

Оценка

Реализация клиент-серверной модели	**10 баллов**

Оформление

Должно быть, минимум два модуля правильно оформленных (Client и Server). Логика должна находится в отдельных классах и быть тестируемой. На вход программе должна быть возможность подавать данные с консоли. Клиенты должны писать все сообщения в лог файл (можно указать в командной строке), в том виде котором они приходят от сервера.
Весь код должен быть покрыт модульными тестами. Ссылку на доступ к репозиторию можно прислать вместо архива.

Структура архива:

src – исходники, без каких либо бинарных файлов

lib – дополнительные библиотеки, которые используются в вашем решении (кроме стандартных). Если пользуетесь **.nuget**, то заливать библиотеки не надо

bin – в этой директории должен хранится zip архив с вашими исполняемыми файлами и библиотеками. После распаковки архива, программа должна запускаться без дополнительных действий. Если есть особенности, указывайте их в readme.txt

Не забудьте положить в архив файл **readme.txt**. В файле описать интерфейс программы (алгоритм работы с программой, пункты меню, управляющие клавиши)

!Внимание: обратите внимание на правильное именование и структуру работы, а также на содержание файла readme.txt. Если ваша работа не будет соответствовать требованиям, баллы могут быть снижены.

Результаты работы

Результаты смотрите в интернете. Совместная работа или обмен кусками кода, при условии, что факт командной работы не был указан в readme.txt, считается плагиатом и оценивается в -5 баллов всем участникам.

Дополнительный материал

- **https://msdn.microsoft.com/en-us/library/system.net.sockets%28v=vs.110%29.aspx**
- **http://www.codeproject.com/Articles/13232/A-very-basic-TCP-server-written-in-C**
- **http://www.codeproject.com/Articles/488668/Csharp-TCP-Server**

Задание 2. Map reduce

Цель задания

Целью лабораторной работы является моделирование системы удаленных вычислений, т.е. системы, предоставляющей свое машинное время для решения трудоемких задач по клиентским запросам. Имеются сервера, считающиеся системами с высокой вычислительной мощностью, и клиент, решающий некоторую задачу путем разделения ее на подзадачи, раздачи частей по серверам (map) и дальнейшего сокращения результатов (reduce).

В качестве решаемой задачи – решение задачи вычисления количества слов в тексте. Задача легко разделяется на более мелкие задачи
Используемые технологии: ввод – вывод из файлов, сетевое взаимодействие, многопоточность, сериализация, загрузчик классов, reflection
Требуется написать приложение типа client-server (WCF).
Приложение состоит из трех элементов: Master Server – менеджер вычислений и управления Worker'ами. Worker – приложения осуществляющие вычисления map/reduce в определенном контексте. Клиентское приложение – которое посылает Master Server'у сами функции Map и Reduce.
Сервер может поддерживать соединение одновременно с произвольным количеством клиентов, но одновременно исполняет задачи только одного клиента.

Описание задания

Требования к Master Server. Сервер умеет регистрировать сервисы Workers двух типов – Map и Reduce. Так же сервер принимает функции Map и Reduce для Workers от Клиента. Следит за исполнением и очередями задача для Workers. Загрузка файлов для Workers может быть осуществлена так же через Master и он должен будет разделить файлы каждому из Workers. Master Server передает Workers функции в виде сериализованных объектов.

Требования к Worker Servers. Уметь регистрироваться на Master Server отправкой своих параметров. Так же исполнять функции Map/Reduce присланные с Master вместе с контекстом (файлом для обработки). Передавать Master Server свое состояние.

Требования к Клиенту. Передавать Master Серверу функции Map/Reduce в определенном формате (DataContract)

Все приложения должны быть консольными и уметь выводит весь лог работы.

Map-Reduce

Алгоритм Map-Reduce не следует путать с одноименным фреймворком распределенных вычислений, хотя в определенном смысле целью лабораторной работы является построение упрощенной модели этого фреймворка.
Идея алгоритма состоит в том, что Master Server делит задачу на куски и рассылает Workers, которые генерируют ему ответы. Эти ответы Master Server склеивает в конечный результат. Например, перед Master Server стоит задача найти все простые числа от одного до ста. Допустим, у Master Server имеется три подключенных сервера вычислений. Master Server делит задачу,

например, на десять диапазонов по 10 чисел и рассылает серверам, собирая их ответы (шаг Map). Получившиеся ответы рассылаются северам Reduce для склеивания в конечный результат (шаг Reduce) и результат высылается Клиенту.

Ввод и вывод

Ввод/вывод осуществляется через чтение стандартных для .NET потоков и файлов (Stream и System.IO)

Установление сетевого подключения.

Сетевое подключение между двумя процессами устанавливается через протокол TCP или HTTP, через сервис WCF. Нужно иметь представление о механизм соединения WCF сервиса и понимать, что такое протокол SOAP и как стерилизуются объекты для передачи. Нужно знать отличия NetDataContractSerializer и DataContractSerializer.

Serialization и Reflection

Каким образом мы можем взять произвольный поток байт, преобразовать его в класс, создать экземпляр этого класса и выполнить его методы? Эта задача решается с помощью технологий serialization и reflection

Serialization – отвечает за преобразование потока байт в определение класса. В случае, если класс хранится каким-либо нестандартным методом – например, лежит в сети, в виде xml, зашифрован, а также в любых других нестандартных вариантах. Может потребоваться написание собственного загрузчика классов, способного к получению данных из нестандартного места. В нашем случае нам надо преобразовать только поток байт, пришедший по сети, и для этого мы можем использовать стандартный NetDataContractSerializer.

Reflection – В **информатике отражение** или **рефлексия** означает процесс, во время которого **программа** может отслеживать и модифицировать собственную структуру и поведение во время выполнения.

Оценка

Реализация клиент-серверной модели	**15 баллов**

Оформление

Логика должна находится в отдельных классах и быть тестируемой. На вход программе должна быть возможность подавать данные с консоли. Клиенты должны писать все сообщения в лог файл (должно быт можно указать в командной строке), в том виде котором они приходят от сервера.

Весь код должен быть покрыт модульными тестами. Ссылку на доступ к репозиторию можно прислать вместо архива.

Структура:

src – исходники, без каких либо бинарных файлов

lib – дополнительные библиотеки, которые используются в вашем решении (кроме стандартных). Если пользуетесь **.nuget**, то заливать библиотеки не надо

bin – в этой директории должен хранится zip архив с вашими исполняемыми файлами и библиотеками. После распаковки архива, программа должна запускаться без дополнительных действий. Если есть особенности, указывайте их в readme.txt

Не забудьте положить в архив файл **readme.txt**. В файле описать интерфейс программы (алгоритм работы с программой, пункты меню, управляющие клавиши)

!Внимание: обратите внимание на правильное именование и структуру работы, а также на содержание файла readme.txt. Если ваша работа не будет соответствовать требованиям, баллы могут быть снижены. Время сдачи работы оценивается по последнему комиту. Если у вас есть модификации к работе после последнего срока – делайте branch и заливайте изменения туда.

Результаты работы

Результаты смотрите в интернете. Совместная работа или обмен кусками кода, при условии что факт командной работы не был указан в readme.txt, считается плагиатом и оценивается в -5 баллов всем участникам.

Дополнительный материал

1. **https://msdn.microsoft.com/en-us/library/system.runtime.serialization.netdatacontractserializer(v=vs.110).aspx**
2. **https://docs.microsoft.com/en-us/dotnet/framework/wcf/whats-wcf**
3.

Задание 3. Raft

Введение

Это первая лабораторная из серии лабораторных, после выполнения которых получится отказоустойчивая система хранения типа ключ/значение. В этой лабораторной вы реализуете протокол Raft – протокол репликации машины состояний. В следующей лабораторной вы реализуете хранилище ключ/значение как сервис над протоколом Raft. Затем вы распределите (shard) ваш сервис на множество реплицируемых машин состояний для большей производительности.

Реплицируемый сервис (далее база данных ключ/значение) достигает отказоустойчивости за счет хранения копий данных на множестве реплик сервиса. Репликация позволяет службе продолжать работу, даже если на некоторых ее серверах возникают сбои (сбои, упавшая или нестабильная сеть). Проблема заключается в том, что из-за сбоев реплики могут содержать разные копии данных.

Raft управляет репликами состояния службы и, в частности, помогает службе определить правильное состояние после сбоев. Raft реализует реплицированный конечный автомат. Он организует клиентские запросы в последовательность, называемую журналом, и обеспечивает согласование всех реплик с содержимым журнала. Каждая реплика выполняет клиентские запросы из журнала в порядке их появления в журнале, применяя эти запросы к локальной копии реплики состояния службы. Поскольку все живые реплики видят одинаковое содержимое журнала, все они выполняют одинаковые запросы в одинаковом порядке и, таким образом, у них имеется одинаковое состояние службы. Если сервер выходит из строя, но позже восстанавливается, Raft позаботится о своевременном обновлении журнала. Пока большинство серверов живы и могут общаться друг с другом, Raft продолжает работать. Если такого большинства нет, Raft приостановит работу, но как только большинство сможет снова общаться, поймет, где остановился.

В этой лабораторной работе вы будете реализовывать Raft как объекты C# с методами, предназначенными для использования в качестве модуля в более крупном сервисе. Набор экземпляров Raft взаимодействует друг с другом с помощью RPC для поддержки реплицированных журналов. Ваш интерфейс Raft будет поддерживать неопределенную последовательность пронумерованных команд, также называемых записями журнала. Записи нумеруются номерами индексов. Запись журнала с заданным индексом будет в конечном итоге зафиксирована. В этот момент ваш Raft должен отправить запись журнала в более крупный сервис для его выполнения.

Только RPC может использоваться для взаимодействия между различными экземплярами Raft. Например, разные экземпляры вашей реализации Raft не могут совместно использовать переменные C#. Ваша реализация не должна использовать файлы вообще.

В этой лабораторной работе вы реализуете большую часть дизайна Raft, описанного в расширенном документе, включая сохранение состояния и чтение его после сбоя узла, а затем перезапуска. Вы не будете реализовывать изменения членства в кластере (Раздел 6) или сжатие / снимок лога (Раздел 7).

Вы должны прочитать статью о Raft (входит в задание) и пользоваться лекционным заметкам плота. Возможно, вам будет полезно взглянуть на этот совет, написанный для студентов, и это иллюстрированное руководство по Raft. Для более лучшего понимая можно прочитать про Paxos, Chubby, Paxos Made Live, Spanner, Zookeeper, Harp, Viewstamped Replication и Bolosky et al.

- Начните как можно раньше. Хотя объем кода для реализации невелик, заставить его работать правильно будет очень сложно. И алгоритм, и код сложны, и есть много крайних случаев для рассмотрения. В случае сбоя одного из тестов может потребоваться некоторое время, чтобы понять, в каком сценарии ваше решение не является правильным, и как это исправить.
- Перед началом работы прочитайте и поймите статью про Raft и лекционные заметки. Ваша реализация должна точно соответствовать описанию в статье, особенно на рисунке 2, поскольку тесты это учитывают.

Эта лаборатория состоит из трех частей. Вы должны представить каждую часть в соответствующий срок. В этой лабораторной работе не много кода, но параллелизм делает его потенциально сложным для отладки; надо начинать как можно раньше.

Политика сотрудничества

Вы должны написать весь код самостоятельно, который вы сдаете для этого курса, за исключением кода, который мы даем вам как часть задания. Вам не разрешено смотреть на чье-либо решение, вам не разрешено смотреть на код предыдущих лет, и вам не разрешено смотреть на другие реализации Raft. Вы можете обсуждать задания с другими учащимися, но не можете просматривать или копировать чужой код или разрешать кому-либо еще просматривать ваш код.

Пожалуйста, не публикуйте свой код и не делайте его доступным для нынешних или будущих студентов. Репозитории github.com по умолчанию являются общедоступными, поэтому, пожалуйста, не размещайте свой код там, пока вы не сделаете репозиторий закрытым. Возможно, вам будет удобно использовать Azure Devops, но обязательно создайте частный репозиторий.

Код

Your implementation must support the following interface, which the tester and (eventually) your key/value server will use.

```
// create a new Raft server instance:
Make(peers, me, persister, applyCh)

// start agreement on a new log entry:
Start(command interface{}) (index, term, isleader)

// ask a Raft for its current term, and whether it thinks it is leader
GetState() (term, isLeader)

// each time a new entry is committed to the log, each Raft peer
// should send an ApplyMsg to the service (or tester).
type ApplyMsg
```

A service calls `Make(peers,me,…)` to create a Raft peer. The peers argument is an array of established RPC connections, one to each Raft peer (including this one). The `me` argument is the index of this peer in the peers array. `Start(command)` asks Raft to start the processing to append the command to the replicated log. `Start()` should return immediately, without waiting for this process to complete. The service expects your implementation to send an `ApplyMsg` for each new committed log entry to the `applyCh` argument to `Make()`.

Your Raft peers should exchange RPCs using the WCF that we provide to you. It is modeled after .NET **rpc library**, but internally uses WCF channels rather than sockets. Procedure is sends an RPC (`sendRequestVote()`) and that handles an incoming RPC (`RequestVote()`).

This lab may be your first exposure to writing challenging concurrent code and your first implementation may not be clean enough that you can easily reason about its correctness. Give yourself enough time to rewrite your implementation so that you can easily reason about its correctness. Subsequent labs will build on this lab, so it is important to do a good job on your implementation.

Part 3A

Implement leader election and heartbeats (`AppendEntries` RPCs with no log entries). The goal for Part 2A is for a single leader to be elected, for the leader to remain the leader if there are no failures, and for a new leader to take over if the old leader fails or if packets to/from the old leader are lost.

- Add any state you need to the `Raft` struct.You'll also need to define a struct to hold information about each log entry. Your code should follow Figure 2 in the paper as closely as possible.
- C# marshals only the *public* fields in structure marked [DataContract] and [DataMember] passed over RPC. Public fields are the ones whose names start with capital letters. Forgetting to make fields public by naming them with capital letters is the single most frequent source of bugs in these labs.
- Fill in the `RequestVoteArgs` and `RequestVoteReply` structs. Modify `Make()` to create a background routine that will kick off leader election periodically by sending out `RequestVote` RPCs when it hasn't heard from another peer for a while. This way a peer will learn who is the leader, if there is already leader, or become itself the leader. Implement the `RequestVote()` RPC handler so that servers will vote for one another.
- To implement heartbeats, define an `AppendEntries` RPC struct (though you may not need all the arguments yet), and have the leader send them out periodically. Write an `AppendEntries` RPC handler method that resets the election timeout so that other servers don't step forward as leaders when one has already been elected.
- Make sure the election timeouts in different peers don't always fire at the same time, or else all peers will vote for themselves and no one will become leader.
- The tester requires that the leader send heartbeat RPCs no more than ten times per second.
- The tester requires your Raft to elect a new leader within five seconds of the failure of the old leader (if a majority of peers can still communicate). Remember, however, that leader election may require multiple rounds in case of a split vote (which can happen if packets are lost or if candidates unluckily choose the same random backoff times). You must pick election timeouts (and thus heartbeat intervals) that are short enough that it's very likely that an election will complete in less than five seconds even if it requires multiple rounds.
- The paper's Section 5.2 mentions election timeouts in the range of 150 to 300 milliseconds. Such a range only makes sense if the leader sends heartbeats considerably more often than once per 150 milliseconds. Because the tester limits you to 10 heartbeats per second, you will have to use an election timeout larger than the paper's 150 to 300 milliseconds, but not too large, because then you may fail to elect a leader within five seconds.
- You may find C# **time** and **rand** packages useful.
- If your code has trouble passing the tests, read the paper's Figure 2 again; the full logic for leader election is spread over multiple parts of the Figure.
- A good way to debug your code is to insert print statements when a peer sends or receives a message, and collect the output in a file with `2A > out`. Then, by studying the trace of messages in the `out` file, you can identify where your implementation deviates from the desired protocol. You might use any logger like NLog or Log4Net.

Be sure you pass the 2A tests before submitting Part 2A. Note that the 2A tests test the basic operation of leader election. Parts B and C will test leader election in more challenging settings and may expose bugs in your leader election code for which the 2A tests do not test.

Part 3B

We want Raft to keep a consistent, replicated log of operations. A call to `Start()` at the leader starts the process of adding a new operation to the log; the leader sends the new operation to the other servers in `AppendEntries` RPCs.

Implement the leader and follower code to append new log entries. This will involve implementing `Start()`, completing the `AppendEntries` RPC structs, sending them, fleshing out the `AppendEntry` RPC handler, and advancing the `commitIndex` at the leader.

- You will need to make your implementation robust against various kinds of failures. Many of the tests involve servers failing and the network losing RPCs.
- You will need to implement the election restriction (section 5.4.1 in the paper).
- While the Raft leader is the only server that initiates appends of new entries to the log, all the servers need to independently give each newly committed entry to their local service replica (via their own `applyCh`). You should try should try to keep the goroutines that implement the Raft protocol as separate as possible from the code that sends committed log entries on the `applyCh` (e.g., by using a separate goroutine for delivering committed messages). If you don't separate these activities cleanly, then it is easy to create deadlocks, either in this lab or in subsequent labs in which you implement services that use your Raft package. Without a clean separation, a common deadlock scenario is as follows: an RPC handler sends on the `applyCh` but it blocks because no goroutine is reading from the channel (e.g., perhaps because it called `Start()`). Now, the RPC handler is blocked while holding the mutex on the Raft structure. The reading goroutine is also blocked on the mutex because `Start()` needs to acquire it. Furthermore, no other RPC handler that needs the lock on the Raft structure can run.
- Give yourself enough time to rewrite your implementation because only after writing a first implementation will you realize how to organize your code cleanly. For example, only after writing one implementation will you understand how to write an implementation that makes it easy to argue that your implementation has no deadlocks.
- Figure out the minimum number of messages Raft should use when reaching agreement in non-failure cases and make your implementation use that minimum.

Be sure you pass the 2A and 2B tests before submitting Part 2B.

Part 3C

If a Raft-based server reboots it should resume service where it left off. This requires that Raft keep persistent state that survives a reboot. The paper's Figure 2 mentions which state should be persistent.

A “real” implementation would do this by writing Raft's persistent state to disk each time it changes, and reading the latest saved state from disk when restarting after a reboot. Your implementation won't use the disk; instead, it will save and restore persistent state from a `Persister` object. Whoever calls `Raft.Make()` supplies a `Persister` that initially holds Raft's most recently persisted state (if any). Raft should initialize its state from that `Persister`, and should use it to save its persistent state each time the state changes. Use the `Persister`'s `ReadRaftState()` and `SaveRaftState()` methods.

Implement persistence by first adding code that saves and restores persistent state to `persist()` and `readPersist()`. You will need to encode (or "serialize") the state as an array of bytes in order to pass it to the `Persister`.

You now need to determine at what points in the Raft protocol your servers are required to persist their state, and insert calls to `persist()` in those places. You must also load persisted state in `Raft.Make()`. Once you've done this, you should pass the remaining tests.

In order to avoid running out of memory, Raft must periodically discard old log entries, but you **do not** have to worry about this until the next lab.

- The encoder you'll use to encode persistent state only saves public fields (fields whose names start with upper case letters). Using small caps for field names is a common source of mysterious bugs, since C# doesn't warn you that those won't be saved.
- In order to pass some of the challenging tests towards the end, such as those marked "unreliable", you will need to implement the optimization to allow a follower to back up the leader's nextIndex by more than one entry at a time. See the description in the extended Raft paper starting at the bottom of page 7 and top of page 8 (marked by a gray line). The paper is vague about the details; you will need to fill in the gaps, perhaps with the help of the Raft lectures.

Be sure you pass all the tests before submitting Part 2C.

Handin procedure

Before submitting, please run *all* the tests one final time. Some bugs may not appear on every run, so run the tests multiple times.

Submit your code via the class's submission website.

Check the submission website to make sure you submitted a working lab!

You may submit multiple times. We will use the timestamp of your **last** submission for the purpose of calculating late days. Your grade is determined by the score your solution **reliably** achieves when we run the tester on our test machines.

Оценка

Реализация	**15 баллов**

Оформление

Вашу программу следует размести в Git (Желательно создать учетную Visual Studio Online (VSTS).. Весь код должен быть покрыт модульными тестами. Ссылку на доступ к репозиторию можно прислать вместо архива.

Структура:

src – исходники, без каких либо бинарных файлов

lib – дополнительные библиотеки, которые используются в вашем решении (кроме стандартных). Если пользуетесь **.nuget**, то заливать библиотеки не надо

bin – в этой директории должен хранится zip архив с вашими исполняемыми файлами и библиотеками. После распаковки архива, программа должна запускаться без дополнительных действий. Если есть особенности, указывайте их в readme.txt

Не забудьте положить в архив файл **readme.txt**. В файле описать интерфейс программы (алгоритм работы с программой, пункты меню, управляющие клавиши)

!Внимание: обратите внимание на правильное именование и структуру работы, а также на содержание файла readme.txt. Если ваша работа не будет соответствовать требованиям, баллы могут быть снижены. Время сдачи работы оценивается по последнему комиту. Если у вас есть модификации к работе после последнего срока – делайте branch и заливайте изменения туда.

Результаты работы

Результаты смотрите в интернете. Совместная работа или обмен кусками кода, при условии что факт командной работы не был указан в readme.txt, считается плагиатом и оценивается в -5 баллов всем участникам.

Задание 4. Fault-tolerant Key/Value Service

Introduction

In this lab you will build a fault-tolerant key-value storage service using your Raft library from **lab** 3. You will build your key-value service as a replicated state machine, consisting of several key-value servers that coordinate their activities through the Raft log. Your key/value service should continue to process client requests as long as a majority of the servers are alive and can communicate, in spite of other failures or network partitions.

Your system will consist of clients and key/value servers, where each key/value server also acts as a Raft peer. Clients send `Put()`, `Append()`, and `Get()` RPCs to key/value servers (called kvraft servers), which then place those calls into the Raft log and execute them in order. A client can send an RPC to any of the kvraft servers, but should retry by sending to a different server if the server is not currently a Raft leader, or if there's a failure. If the operation is committed to the Raft log (and hence applied to the key/value state machine), its result is reported to the client. If the operation failed to commit (for example, if the leader was replaced), the server reports an error, and the client retries with a different server.

This lab has two parts. In part A, you will implement the service without worrying that the Raft log can grow without bound. In part B, you will implement snapshots (Section 7 in the paper), which will allow Raft to garbage collect old log entries. Please submit each part by the respective deadline.

- This lab doesn't require you to write much code, but you will most likely spend a substantial amount of time thinking and staring at debugging logs to figure out why your implementation doesn't work. Debugging will be more challenging than in the Raft lab because there are more components that work asynchronously of each other. Start early.
- You should reread the **extended Raft paper**, in particular Sections 7 and 8. For a wider perspective, have a look at Chubby, Raft Made Live, Spanner, Zookeeper, Harp, Viewstamped Replication, and **Bolosky et al.**

Part A: Key/value service without log compaction

The service supports three RPCs: `Put(key, value)`, `Append(key, arg)`, and `Get(key)`. It maintains a simple database of key/value pairs. `Put()` replaces the value for a particular key in the database, `Append(key, arg)` appends arg to key's value, and `Get()` fetches the current value for a key. An `Append` to a non-existant key should act like `Put`.

You will implement the service as a replicated state machine consisting of several kvservers. Your kvraft client code (`Clerk`) should try different kvservers it knows about until one responds positively. As long as a client can contact a kvraft server that is a Raft leader in a majority partition, its operations should eventually succeed.

Your first task is to implement a solution that works when there are no dropped messages, and no failed servers. Note that your service must provide *sequential consistency* to applications that use its client interface. That is, completed application calls to the `Clerk.Get()`, `Clerk.Put()`, and `Clerk.Append()` methods must appear to all clients to have affected the service in the same order, even in there are failures and leader changes.

Completed calls should have exactly-once semantics. A `Clerk.Get(key)` should see the value written by the most recent `Clerk.Put(key, …)` or `Clerk.Append(key, …)` (in the total order).

A reasonable plan of attack may be to first fill in the `Op` struct in `server` with the "value" information that kvraft will use Raft to agree on, and then implement the `PutAppend()` and `Get()` handlers in `server`. The handlers should enter an `Op` in the Raft log using `Start()`, and should reply to the client when that log entry is committed. Note that you **cannot** execute an operation until the point at which it is committed in the log (i.e., when it arrives on the Raft `applyCh`).

You have completed this task when you **reliably** pass the first test in the test suite: "One client". You may also find that you can pass the "concurrent clients" test, depending on how sophisticated your implementation is.

Your kvraft servers should not directly communicate; they should only interact with each other through the Raft log.

- After calling `Start()`, your kvraft servers will need to wait for Raft to complete agreement. Commands that have been agreed upon arrive on the `applyCh`. You should think carefully about how to arrange your code so that your code will keep reading `applyCh`, while `PutAppend()` and `Get()` handlers submit commands to the Raft log using `Start()`. It is easy to achieve deadlock between the kvserver and its Raft library.
- Your solution needs to handle the case in which a leader has called Start() for a client RPC, but loses its leadership before the request is committed to the log. In this case you should arrange for the client to re-send the request to other servers until it finds the new leader. One way to do this is for the server to detect that it has lost leadership, by noticing that a different request has appeared at the index returned by Start(), or that the term reported by Raft.GetState() has changed. If the ex-leader is partitioned by itself, it won't know about new leaders; but any client in the same partition won't be able to talk to a new leader either, so it's OK in this case for the server and client to wait indefinitely until the partition heals.
- You will probably have to modify your client Clerk to remember which server turned out to be the leader for the last RPC, and send the next RPC to that server first. This will avoid wasting time searching for the leader on every RPC, which may help you pass some of the tests quickly enough.
- A kvraft server should not complete a `Get()` RPC if it is not part of a majority (so that it does not serve stale data). A simple solution is to enter every `Get()` (as well as each `Put()` and `Append()`) in the Raft log. You don't have to implement the optimization for read-only operations that is described in Section 8.
- It's best to add locking relatively early in development because the need to avoid deadlocks sometimes affects overall code design. Check that your code is race-free using `go test -race`.

In the face of unreliable connections and server failures, a client may send an RPC multiple times until it finds a kvraft server that replies positively. If a leader fails just after committing an entry to the Raft log, the client may believe that the request failed, and re-send it to another leader. Each call to `Clerk.Put()` or `Clerk.Append()` should result in just a single execution, so you will have to ensure that the re-send doesn't result in the servers executing the request twice.

Add code to cope with duplicate client requests, including situations where the client sends a request to a kvraft leader in one term, times out waiting for a reply, and re-sends the request to a new leader in another term. The client request should always execute just once. To pass part A, your service should **reliably** pass all tests through `TestPersistPartitionUnreliable()`.

- You will need to uniquely identify client operations to ensure that the key/value service executes each one just once.
- Your scheme for duplicate detection should free server memory quickly, for example by having each RPC imply that the client has seen the reply for its previous RPC. It's OK to assume that a client will make only one call into a clerk at a time.

XXX submission instructions for Part A

Part B: Key/value service with log compaction

As things stand now with your lab code, a rebooting server replays the complete Raft log in order to restore its state. However, it's not practical for a long-running server to remember the complete Raft log forever. Instead, you'll modify Raft and kvraft to cooperate to save space: from time to time kvraft will persistently store a "snapshot" of its current state, and Raft will discard log entries that precede the snapshot. When a server restarts (or falls far behind the leader and must catch up), the server first installs a snapshot and then replays log entries from after the point at which the snapshot was created. Section 7 of the **extended Raft paper** outlines the scheme; you will have to design the details.

You should spend some time figuring out what the interface will be between your Raft library and your service so that your Raft library can discard log entries. Think about how your Raft will operate while storing only the tail of the log, and how it will discard old log entries. You should discard them in a way that allows the Go garbage collector to free and re-use the memory; this requires that there be no reachable references (pointers) to the discarded log entries.

The kvraft tester passes `maxraftstate` to your `StartKVServer()`. `maxraftstate` indicates the maximum allowed size of your persistent Raft state in bytes (including the log, but not including snapshots). You should compare `maxraftstate` to `persister.RaftStateSize()`. Whenever your key/value server detects that the Raft state size is approaching this threshold, it should save a snapshot, and tell the Raft library that it has snapshotted, so that Raft can discard old log entries.

Your `raft` probably keeps the entire log. Modify it so that it can be given a log index, discard the entries before that index, and continue operating while storing only log entries after that index. Make sure you pass all the Raft tests after making these changes.

Modify your kvraft server so that it detects when the persisted Raft state grows too large, and then saves a snapshot and tells Raft that it can discard old log entries. Save each snapshot with persister.SaveSnapshot() (don't use files).

Modify your Raft leader code to send an InstallSnapshot RPC to a follower when the leader has discarded the log entries the follower needs. When a follower receives an InstallSnapshot RPC, your Raft code will need to send the included snapshot to its kvraft. You can use the `applyCh` for this purpose — see the `UseSnapshot` field. A kvraft instance should restore the snapshot from the persister when it re-starts. Your solution is complete when you pass the remaining tests **reliably**.

The `maxraftstate` limit applies to the encoded bytes your Raft passes to `persister.SaveRaftState()`.

- Think about when a kvserver should snapshot its state and what should be included in the snapshot. You must store each snapshot in the persister object using `SaveSnapshot()`, **not** along with the other Raft state in `SaveRaftState()`. You can read the latest stored snapshot using `ReadSnapshot()`.
- Remember that your kvserver must be able to detect duplicate client requests across checkpoints, so any state you are using to detect them must be included in the snapshots. Remember to capitalize all fields of structures stored in the snapshot.
- Make sure you pass `TestSnapshotRPC` before moving on to the other Snapshot tests.
- A common source of bugs is for the Raft state and the snapshot state to be inconsistent with each other, particularly after re-starts or InstallSnapshots.

Handin procedure

Check the submission website to make sure you submitted a working lab!

You may submit multiple times. We will use the timestamp of your **last** submission for the purpose of calculating late days. Your grade is determined by the score your solution **reliably** achieves when we run the tester on our test machines.

Оценка

Реализация	**15 баллов**

Оформление

Вашу программу следует размести в Git (Желательно создать учетную Visual Studio Online (VSTS).. Весь код должен быть покрыт модульными тестами. Ссылку на доступ к репозиторию можно прислать вместо архива.

Структура:

src – исходники, без каких либо бинарных файлов

lib – дополнительные библиотеки, которые используются в вашем решении (кроме стандартных). Если пользуетесь **.nuget**, то заливать библиотеки не надо

bin – в этой директории должен хранится zip архив с вашими исполняемыми файлами и библиотеками. После распаковки архива, программа должна запускаться без дополнительных действий. Если есть особенности, указывайте их в readme.txt

Не забудьте положить в архив файл **readme.txt**. В файле описать интерфейс программы (алгоритм работы с программой, пункты меню, управляющие клавиши)

!Внимание: обратите внимание на правильное именование и структуру работы, а также на содержание файла readme.txt. Если ваша работа не будет соответствовать требованиям, баллы могут быть снижены.

Результаты работы

Результаты смотрите в интернете. Совместная работа или обмен кусками кода, при условии что факт командной работы не был указан в readme.txt, считается плагиатом и оценивается в -5 баллов всем участникам.

Задание 5. Sharded Key/Value Service

Introduction

In this lab you'll build a key/value storage system that "shards," or partitions, the keys over a set of replica groups. A shard is a subset of the key/value pairs; for example, all the keys starting with "a" might be one shard, all the keys starting with "b" another, etc. The reason for sharding is performance. Each replica group handles puts and gets for just a few of the shards, and the groups operate in parallel; thus total system throughput (puts and gets per unit time) increases in proportion to the number of groups.

Your sharded key/value store will have two main components. First, a set of replica groups. Each replica group is responsible for a subset of the shards. A replica consists of a handful of servers that use Raft to replicate the group's shards. The second component is the "shard master". The shard master decides which replica group should serve each shard; this information is called the configuration. The configuration changes over time. Clients consult the shard master in order to find the replica group for a key, and replica groups consult the master in order to find out what shards to serve. There is a single shard master for the whole system, implemented as a fault-tolerant service using Raft.

A sharded storage system must be able to shift shards among replica groups. One reason is that some groups may become more loaded than others, so that shards need to be moved to balance the load. Another reason is that replica groups may join and leave the system: new replica groups may be added to increase capacity, or existing replica groups may be taken offline for repair or retirement.

The main challenge in this lab will be handling reconfiguration -- changes in the assignment of shards to groups. Within a single replica group, all group members must agree on when a reconfiguration occurs relative to client Put/Append/Get requests. For example, a Put may arrive at about the same time as a reconfiguration that causes the replica group to stop being responsible for the shard holding the Put's key. All replicas in the group must agree on whether the Put occurred before or after the reconfiguration. If before, the Put should take effect and the new owner of the shard will see its effect; if after, the Put won't take effect and client must re-try at the new owner. The recommended approach is to have each replica group use Raft to log not just the sequence of Puts, Appends, and Gets but also the sequence of reconfigurations. You will need to ensure that at most one replica group is serving requests for each shard at any one time.

Reconfiguration also requires interaction among the replica groups. For example, in configuration 10 group G1 may be responsible for shard S1. In configuration 11, group G2 may be responsible for shard S1. During the reconfiguration from 10 to 11, G1 and G2 must use RPC to move the contents of shard S1 (the key/value pairs) from G1 to G2.

Only RPC may be used for interaction among clients and servers. For example, different instances of your server are not allowed to share Go variables or files.

This lab uses "configuration" to refer to the assignment of shards to replica groups. This is not the same as Raft cluster membership changes. You don't have to implement Raft cluster membership changes.

This lab's general architecture (a configuration service and a set of replica groups) follows the same general pattern as Flat Datacenter Storage, BigTable, Spanner, FAWN, Apache HBase, Rosebud, Spinnaker, and many others. These systems differ in many details from this lab, though, and are also typically more sophisticated and capable. For example, the lab doesn't evolve the sets of peers in each Raft group; its data and query models are very simple; and handoff of shards is slow and doesn't allow concurrent client access.

Your Lab 5 sharded server, Lab 5 shard master, and Lab 4 kvraft must all use the same Raft implementation. We will re-run the Lab 3 and Lab 4 tests as part of grading Lab 5, and your score on the older tests will count towards your total Lab 5 grade.

As this lab is due at the end of term, you will **not** be able to use late hours when submitting part B of this lab.

Part A: The Shard Master

First you'll implement the shard master and client.

The shardmaster manages a sequence of numbered configurations. Each configuration describes a set of replica groups and an assignment of shards to replica groups. Whenever this assignment needs to change, the shard master creates a new configuration with the new assignment. Key/value clients and servers contact the shardmaster when they want to know the current (or a past) configuration.

Your implementation must support the RPC interface, which consists of `Join`, `Leave`, `Move`, and `Query` RPCs.

The `Join` RPC is used by to add new groups to your system. It is given a set of mapping from unique, non-zero replica group identifiers (GID) to lists of server names. The shardmaster should react by creating a new configuration that includes the new replica groups. The new configuration should divide the shards as evenly as possible among the groups, and should move as few shards as possible to achieve that goal.

The `Leave` RPC's arguments are the GIDs of previously joined groups. The shardmaster should create a new configuration that does not include those groups, and that assigns those groups' shards to the remaining groups. The new configuration should divide the shards as evenly as possible among the groups, and should move as few shards as possible to achieve that goal.

The `Move` RPC's arguments are a shard number and a GID. The shardmaster should create a new configuration in which the shard is assigned to the group. The purpose of `Move` is to allow us to test your software. A `Join` or `Leave` following a `Move` will likely un-do the `Move`, since `Join` and `Leave` re-balance.

The `Query` RPC's argument is a configuration number. The shardmaster replies with the configuration that has that number. If the number is -1 or bigger than the biggest known configuration number, the shardmaster should reply with the latest configuration. The result of `Query(-1)` should reflect every `Join`, `Leave`, or `Move` RPC that the shardmaster finished handling before it received the `Query(-1)` RPC.

The very first configuration should be numbered zero. It should contain no groups, and all shards should be assigned to GID zero (an invalid GID). The next configuration (created in response to a `Join` RPC) should be numbered 1, &c. There will usually be significantly more shards than groups (i.e., each group will serve more than one shard), in order that load can be shifted at a fairly fine granularity.

Your task is to implement the interface specified above in `client` and `server`. Your shardmaster must be fault-tolerant, using your Raft library from Lab 3/4. Note that we will re-run the tests from Lab 3 and 4 when grading Lab 5, so make sure you do not introduce bugs into your Raft implementation. You have completed this task when you pass all the tests.

- Start with a stripped-down copy of your kvraft server.

- You should implement duplicate client request detection for RPCs to the shard master. The shardmaster tests don't test this, but the shardkv tests will later use your shardmaster on an unreliable network; you may have trouble passing the shardkv tests if your shardmaster doesn't filter out duplicate RPCs.
- C# maps are references. If you assign one variable of type map to another, both variables refer to the same map. Thus if you want to create a new `Config` based on a previous one, you need to create a new map object (with `make()`) and copy the keys and values individually.

Part B: Sharded Key/Value Server

Now you'll build shardkv, a sharded fault-tolerant key/value storage system.

Each shardkv server operates as part of a replica group. Each replica group serves `Get`, `Put`, and `Append` operations for some of the key-space shards. Use `key2shard()`. in `client` to find which shard a key belongs to. Multiple replica groups cooperate to serve the complete set of shards. A single instance of the `shardmaster` service assigns shards to replica groups; when this assignment changes, replica groups have to hand off shards to each other, while ensuring that clients do not see inconsistent responses.

Your storage system must provide a linearizable interface to applications that use its client interface. That is, completed application calls to the `Clerk.Get()`, `Clerk.Put()`, and `Clerk.Append()` methods in `client` must appear to have affected all replicas in the same order. A `Clerk.Get()` should see the value written by the most recent `Put`/`Append` to the same key. This must be true even when `Get`s and `Put`s arrive at about the same time as configuration changes.

Each of your shards is only required to make progress when a majority of servers in the shard's Raft replica group is alive and can talk to each other, and can talk to a majority of the `shardmaster` servers. Your implementation must operate (serve requests and be able to re-configure as needed) even if a minority of servers in some replica group(s) are dead, temporarily unavailable, or slow.

A shardkv server is a member of only a single replica group. The set of servers in a given replica group will never change.

We supply you with `client` code that sends each RPC to the replica group responsible for the RPC's key. It re-tries if the replica group says it is not responsible for the key; in that case, the client code asks the shard master for the latest configuration and tries again. You'll have to modify client.go as part of your support for dealing with duplicate client RPCs, much as in the kvraft lab.

Your server should not call the shard master's `Join()` handler. The tester will call `Join()` when appropriate.

Your first task is to pass the very first shardkv test. In this test, there is only a single assignment of shards, so your code should be very similar to that of your Lab 4 server. The biggest modification will be to have your server detect when a configuration happens and start accepting requests whose keys match shards that it now owns.

Now that you solution works for the static sharding case, it's time to tackle the problem of configuration changes. You will need to make your servers watch for configuration changes, and when one is detected, to start the shard migration process. If a replica group loses a shard, it must stop serving requests to keys in that shard immediately, and start migrating the data for that shard to the replica group that is taking over ownership. If a replica group gains a shard, it needs to wait for the previous owner to send over the old shard data before accepting requests for that shard.

Implement shard migration during configuration changes. Make sure that all servers in a replica group do the migration at the same time so that they all either accept or reject concurrent client requests. You should focus on passing the second test ("join then leave") before working on the later tests. You are done with this task when you pass all tests up to, but not including, `TestDelete`.

Your server will need to periodically poll the shardmaster to learn about new configurations. The tests expect that your code polls roughly every 100 milliseconds; more often is OK, but much less often may cause problems.

Servers will need to send RPCs to each other in order to transfer shards during configuration changes. The shardmaster's `Config` struct contains server names, but you need a `labrpc.ClientEnd` in order to send an RPC. You should use the `make_end()` function passed to `StartServer()` to turn a server name into a `ClientEnd` contains code that does this.

- Add code to `server` to periodically fetch the latest configuration from the shardmaster, and add code to reject client requests if the receiving group isn't responsible for the client's key's shard. You should still pass the first test.
- Your server should respond with an `ErrWrongGroup` error to a client RPC with a key that the server isn't responsible for (i.e. for a key whose shard is not assigned to the server's group). Make sure your `Get`, `Put`, and `Append` handlers make this decision correctly in the face of a concurrent re-configuration.
- Process re-configurations one at a time, in order.
- If a test fails, check for gob errors (e.g. "gob: type not registered for interface ..."). Go doesn't consider gob errors to be fatal, although they are fatal for the lab.
- Be careful about implementing at-most-once semantics (duplicate detection) for client requests. When groups move shards, they need to move some duplicate detection state as well as the key/value data. Think about how the receiver of a shard should update its own duplicate detection state. Is it correct for the receiver to entirely replace its state with the received one?
- Think about how the shardkv client and server should deal with `ErrWrongGroup`. Should the client change the sequence number if it receives `ErrWrongGroup`? Should the server update the client state if it returns `ErrWrongGroup` when executing a `Get`/`Put` request?
- After a server has moved to a new configuration, it is acceptable for it to continue to store shards that it no longer owns (though this would be regrettable in a real system). This may help simplify your server implementation.
- When group G1 needs a shard from G2 during a configuration change, does it matter at what point during its processing of log entries G2 sends the shard to G1?
- You can send an entire map in an RPC request or reply, which may help keep the code for shard transfer simple.
- If one of your RPC handlers includes in its reply a map (e.g. a key/value map) that's part of your server's state, you may get bugs due to races. The RPC system has to read the map in order to send it to the caller, but it isn't holding a lock that covers the map. Your server, however, may proceed to modify the same map while the RPC system is reading it. The solution is for the RPC handler to include a copy of the map in the reply.
- If you put a map or a slice in a Raft log entry, and your key/value server subsequently sees the entry on the `applyCh` and saves a reference to the map/slice in your key/value server's state, you may have a race. Make a copy of the map/slice, and store the copy in your key/value server's state. The race is between your key/value server modifying the map/slice and Raft reading it while persisting its log.
- During a configuration change, a pair of groups may need to move shards in both directions between them. If you see deadlock, this is a possible source.

Challenge exercises

For this lab, we have two challenge exercises, both of which are fairly complex beasts, but which are also essential if you were to build a system like this for production use.

Garbage collection of state

When a replica group loses ownership of a shard, that replica group should eliminate the keys that it lost from its database. It is wasteful for it to keep values that it no longer owns, and no longer serves requests for. However, this poses some issues for migration. Say we have two groups, G1 and G2, and there is a new configuration C that moves shard S from G1 to G2. If G1 erases all keys in S from its database when it transitions to C, how does G2 get the data for S when it tries to move to C?

Modify your solution so that each replica group will only keep old shards for as long as is absolutely necessary. Bear in mind that your solution must continue to work even if all the servers in a replica group like G1 above crash and are then brought back up. You have completed this challenge if you pass `TestChallenge1Delete` and `TestChallenge1Concurrent`.

- `gob.Decode` will **merge** into the object that it is given. This means that if you decode, say, a snapshot into a map, any keys that are in the map, but not in the snapshot, will **not** be deleted.

Client requests during configuration changes

The simplest way to handle configuration changes is to disallow all client operations until the transition has completed. While conceptually simple, this approach is not feasible in production-level systems; it results in long pauses for all clients whenever machines are brought in or taken out. A better solution would be if the system continued serving shards that are not affected by the ongoing configuration change.

Modify your solution so that, if some shard S is not affected by a configuration change from C to C', client operations to S should continue to succeed while a replica group is still in the process of transitioning to C'. You have completed this challenge when you pass `TestChallenge2Unaffected`.

While the optimization above is good, we can still do better. Say that some replica group G3, when transitioning to C, needs shard S1 from G1, and shard S2 from G2. We really want G3 to immediately start serving a shard once it has received the necessary state, even if it is still waiting for some other shards. For example, if G1 is down, G3 should still start serving requests for S2 once it receives the appropriate data from G2, despite the transition to C not yet having completed.

Modify your solution so that replica groups start serving shards the moment they are able to, even if a configuration is still ongoing. You have completed this challenge when you pass `TestChallenge2Partial`.

Handing procedure

Before submitting, please run *all* the tests one final time.

Also, note that your Lab 5 sharded server, Lab 5 shard master, and Lab 4 kvraft must all use the same Raft implementation. We will re-run the Lab 3 and Lab 4 tests as part of grading Lab 5.

Check the submission website to make sure you submitted a working lab!

You may submit multiple times. We will use the timestamp of your **last** submission for the purpose of calculating late days. Your grade is determined by the score your solution **reliably** achieves when we run the tester on our test machines.

Оценка

Реализация	**15 баллов**

Оформление

Вашу программу следует размести в Git (Желательно создать учетную Visual Studio Online (VSTS).. Весь код должен быть покрыт модульными тестами. Ссылку на доступ к репозиторию можно прислать вместо архива.

Структура:

src – исходники, без каких либо бинарных файлов

lib – дополнительные библиотеки, которые используются в вашем решении (кроме стандартных). Если пользуетесь **.nuget**, то заливать библиотеки не надо

bin – в этой директории должен хранится zip архив с вашими исполняемыми файлами и библиотеками. После распаковки архива, программа должна запускаться без дополнительных действий. Если есть особенности, указывайте их в readme.txt

Не забудьте положить в архив файл **readme.txt**. В файле описать интерфейс программы (алгоритм работы с программой, пункты меню, управляющие клавиши)

!Внимание: обратите внимание на правильное именование и структуру работы, а также на содержание файла readme.txt. Если ваша работа не будет соответствовать требованиям, баллы могут быть снижены.

Результаты работы

Результаты смотрите в интернете. Совместная работа или обмен кусками кода, при условии что факт командной работы не был указан в readme.txt, считается плагиатом и оценивается в -5 баллов всем участникам.

ПРИЛОЖЕНИЕ 2

ШКАЛА ОЦЕНИВАНИЯ КУРСОВОЙ РАБОТЫ

№	Наименование показателя	Отметка (в %)
КАЧЕСТВО КУРСОВОЙ РАБОТЫ		
1	Соответствие содержание работы заданию	0 - 20
2	Грамотность изложения и качество оформление работы	30 - 50
3	Самостоятельность выполнения работы, глубина проработки материала, использование рекомендованной и справочной литературы	0 - 20
4	Обоснованность и доказательность выводов	0 - 10
Общая оценка за выполнение (текущий и рубежный контроль)		**Сумма %**
КАЧЕСТВО ДОКЛАДА		
1	Соответствие содержания доклада содержанию работы	40 – 60
2	Выполнение основной мысли работы	0 – 20
3	Качество изложения материала	0 – 20
Оценка за доклад (промежуточный контроль)		**Сумма %**
ОТВЕТЫ НА ДОПОЛНИТЕЛЬНЫЕ ВОПРОСЫ ПО СОДЕРЖАНИЮ РАБОТЫ		
1	Вопрос 1	0 – 25
2	Вопрос 2	0 – 25
3	Вопрос 3	0 – 25
4	Вопрос 4	0 – 25
Оценка за ответы на вопросы (промежуточный контроль)		**Сумма %**
Общая оценка за промежуточный контроль		**Среднее арифм. %**

ШКАЛА ОЦЕНИВАНИЯ ПРАКТИЧЕСКИХ (ЛАБОРАТОРНЫХ) РАБОТ (текущий контроль)

- 85-100 % - Демонстрирует полное понимание проблемы. Все требования, предъявляемые к заданию выполнены.
- 70-84 % - Демонстрирует значительное понимание проблемы. Все требования, предъявляемые к заданию выполнены.
- 60-69 % - Демонстрирует частичное понимание проблемы. Большинство требований, предъявляемых к заданию выполнены.
- 31-60 % - Демонстрирует небольшое понимание проблемы. Многие требования, предъявляемые к заданию не выполнены.
- 0-30 % - Демонстрирует непонимание проблемы и даже не было попытки решить задачу.

ШКАЛА ОЦЕНИВАНИЯ КОНТРОЛЬНЫХ РАБОТ (рубежный контроль)

- 85-100 % - Демонстрирует полное понимание проблемы. Все задания выполнены.
- 70-84 % - Демонстрирует значительное понимание проблемы. Все задания выполнены, но содержат некоторые неточности.
- 60-69 % - Демонстрирует частичное понимание проблемы. Большинство требований, предъявляемых к заданию, выполнены.
- 31-60 % - Демонстрирует небольшое понимание проблемы. Многие требования, предъявляемые к заданию, не выполнены.
- 0-30 % - Демонстрирует непонимание проблемы или нет ответа и даже не было попытки решить задачу.

ШКАЛА ОЦЕНИВАНИЯ ПИСЬМЕННОГО ОПРОСА
(промежуточный контроль – «ЗНАТЬ»)

Отметкой (7-10- баллов) оценивается ответ, который показывает прочные знания теоретических основ дисциплины, понимание и правильное применение терминологии, правильные ответы на 75-100% вопросов

Отметкой (5-7 баллов) оценивается ответ, который показывает знание теоретических основ дисциплины, но неполное понимание и не всегда правильное применение терминологии, даны правильные ответы на 50-74% вопросов, в ответах допущено некоторое количество неточностей.

Отметкой (3-4 баллов) оценивается ответ, свидетельствующий о знакомстве с некоторыми теоретическими основами дисциплины. Даны правильные ответы на 25-49% вопросов, допущены неточности и ошибки.

Отметкой (2 балла) оценивается ответ, обнаруживающий незнание теоретических основ дисциплины. Отмечается отсутствие логичности и последовательности в ответе. Менее 25% правильных ответов. Допущены серьезные ошибки в содержании ответа.

Отметкой (0-1 балл) оценивается ответ, при котором студент демонстрирует непонимание поставленных вопросов, или нет ответа.

ШКАЛА ОЦЕНИВАНИЯ ПРАКТИЧЕСКИХ ЗАДАНИЙ
(промежуточный контроль – «УМЕТЬ и ВЛАДЕТЬ»)

Отметкой (8-10 баллов) оценивается ответ, при котором студент правильно отвечает на поставленные вопросы, Демонстрирует полное понимание проблемы. Все требования, предъявляемые к заданию, выполнены.

Отметкой (5-7 баллов) оценивается ответ, при котором студент в основном правильно отвечает на поставленные вопросы. Демонстрирует значительное понимание проблемы. Большинство требований, предъявляемых к заданию, выполнены.

Отметкой (2-4 баллов) оценивается ответ, при котором студент в основном не правильно отвечает на поставленные вопросы. Демонстрирует частичное или небольшое понимание проблемы. Многие требования, предъявляемые к заданию, не выполнены.

Отметкой (0 -1 балл) оценивается ответ, при котором студент демонстрирует непонимание проблемы или нет ответа и даже не было попытки решить задачи.

ПРИЛОЖЕНИЕ 3

Курсовой проект: Создание собственной распределенной системы

1. Общий обзор

В курсовом проекте требуется разработать, внедрить и тщательно оттестировать распределенную систему посредством реализация вашего собственного приложения, такого как, например, многопользовательская игра, инструмент для совместной работы или другой системы с поддержкой распределенных транзакций. Возможна совместная работа над проектом (не более 2-х человек). Однако, прежде чем принимать какие-либо решения, пожалуйста, ознакомьтесь с данным документов и политикой сдачи проекта, чтобы избежать сложностей во время сдачи проекта и срыва сроков проекта.

2. Требования к проекту

Поскольку это курс распределенных систем, мы хотим, чтобы у вашего проекта были «интересные» функции с точки зрения распределенных систем. Вот некоторые важные свойства, которые должна иметь ваша система:

- Ваша система должна включать как протокол Paxos, так и поддерживать Two-Phase Commit.
- Система должна поддерживать несколько автономных агентов (как человеческих, так и автоматизированных), конкурирующих за общие ресурсы и выполняющих обновления в режиме реального времени общего состояния системы.
- Состояние системы должно быть распределено между несколькими клиентскими или серверными узлами. Единственным централизованным сервисом должен быть тот, который поддерживает пользователей, регистрирующих, добавляющих или удаляющих клиентов, серверы, и другие поддерживающие задачи.
- Система должна быть надежной. Она должна иметь возможность продолжить работу, даже если на одном из узлов произошел сбой. Должна быть возможность восстановить состояние узла после сбоя, чтобы она мог возобновить работу.
- Если у вас готовый проект или наработки, это не связано с Paxos или 2PC, но, они соответствует целям этого курса и проекта, то вы можете получить разрешение от преподавателя, но сначала должно быть обсуждение проекта.

Разрешается выбрать направление самим, предоставляется широкий выбор по реализации и технологиям, но есть общие требования, которым должна соответствовать система:

- Вы можете использовать любой язык программирования.
- Поскольку трудно предсказать, насколько сложно будет разработать новую систему, система должна быть представлена в виде набора «уровней», где базовый уровень — это то, в чем вы уверены, что можете завершить, а дополнительные уровни добавляют дополнительные функции, как на уровне приложения, так и на уровне системы. Например, ваши усовершенствования системы могут включать в себя повышение уровня отказоустойчивости, уменьшение времени восстановления или добавление функций для повышения масштабируемости или безопасности.
- Вы можете использовать внешние библиотеки, если хотите (за исключением расширенных распределенных системных пакетов, таких как библиотеки, которые реализуют протокол Paxos или Two-Phase Commit). Вы также можете использовать внешние пакеты и SDK для поддержки приложения или пользовательского интерфейса (например, графические интерфейсы / фреймворки GUI, пакеты обработки изображений / видео / аудио, API т.д.).

3. Оценка проекта

Ваша система будет оцениваться в первую очередь как распределенная система. Основная оценка будет зависеть от того, имеет ли ваша система следующие атрибуты:

- Это должна быть «интересная» распределенная система, использующая Paxos или Two-Phase Commit протокол (и, возможно, другие алгоритмы, которые мы рассмотрели в курсе,

например, для распределенной синхронизации, репликации, отказоустойчивости и восстановления, безопасности и т. д.).

- Программное обеспечение должно быть хорошо спроектировано и хорошо реализовано с точки зрения общей архитектуры и деталей реализации.
- Должны быть разработаны и применены систематические процедуры тестирования, такие как модульные тесты и системные тесты.
- Система должна работать надежно и с хорошей производительностью, особенно в части управления сбоями. У вас должен быть хотя бы один тестовый пример, демонстрирующий устойчивость к ошибкам, который вы можете показать в пояснительной записке к проекту.

Важно, но можно включать в проект во вторую очередь:

- Насколько интересно приложение? Это интересная игра, полезный инструмент или решение для реальной проблемы?
- Как хорошо выглядит приложение: есть ли у него приятный интерфейс или привлекательные визуальные эффекты?

Когда оценивается ваш проект, мы ожидаем увидеть исходный код, как он запускается, и посмотреть, как выполняются тесты, разработанные вами.

4. Идеи проектов

Следующий список проектов призван стимулировать ваше воображение. Можно подойти с чем-то совершенно другим. Обратитесь к преподавателю, если вы не уверены соответствии требованиям.

- Проекты с многопользовательским компонентом в режиме реального времени подходят для реализации протокола Paxos - вы можете либо использовать отдельных пользователей в кворуме Paxos, либо использовать Paxos для реплицированного состояния, в котором хранятся обновления состояний системы.
- Совместное редактирование документов в стиле документов Google. Система должна поддерживать редактирование и просмотр в режиме реального времени несколькими участниками. Для отказоустойчивости должны поддерживаться множественные реплики. Кэширование и/или миграция копий могут быть полезны для минимизации времени отклика приложения.
- Игры-симуляции, в стиле The Sims или Farmville. Состояние системы должно быть разделено пространственно, с репликацией для отказоустойчивости.
- Многопользовательская игра в режиме реального времени - здесь много возможностей.
- Система бронирования авиабилетов. Каждая авиакомпания будет поддерживать собственную коллекцию серверов с достаточной репликацией состояния, чтобы обеспечить автоматическое переключение. Должна быть возможность забронировать путешествие, в котором участвуют несколько авиакомпаний.
- Система регистрации с малой временной задержкой. Например, просмотр множества RSS-каналов и отправка всем подписчикам почтовых уведомлений при обновлении. Интерфейс взаимодействия как обычными RSS-каналами, так и с сервисом обновления новостей Google. Требуется реализация репликации и разбиения состояний системы мониторинга так, чтобы она могла масштабироваться и выдерживать сбои узлов.
- Проекты с участием множества агентов или пользователей, которые координируют свои обновления, и которые могут отображаться на карте (например, Карты Google). Можно интегрировать с Google Maps, Yandex Map, 2GIS и т.д.. API этих систем помогают упростить работу с графическим интерфейсом.
- Большие проекты, связанные с созданием приложений и игр для Android и iOS.
- Внедрение распределенной файловой системы, которая делает что-то интересное. Возможно, вы хотите хранить MP3-файлы или фильмы. Или, возможно, для чего-то совершенно другого. Какие ограничения и критерии оптимизации у вас есть?
- Решение проблемы, которая действительно вас задевает или что вы хотели бы видеть, - возможно, ваша новая идея стартапа!

5. Сроки проекта

Проект имеет четыре срока сдачи, каждый из которых представлен ниже:

5.1. Проектное предложение (3 декабря до 23:59)

Прежде чем отправить свой проект, вы должны представить проектное предложение, состоящее из следующих позиций:

- Описание приложения.
- Общая структура реализации.
- Распределенные системные функции и алгоритмы, которые вы намереваетесь реализовать.
- Ваш план тестирования системы.
- Как вы намереваетесь реализовать свою систему, представленную по уровням.
- График того, как вы планируете выполнять дизайн и реализацию проекта.

Этот документ должен составлять от трех до четырех страниц, и он будет составлять 25% от общей оценки. Рекомендуется задать вопросы на этапах планирования, чтобы получить некоторые отзывы об идеях до сдачи вашего предложения.

Чтобы отправить свое предложение, вы должны сдать документ в системе с названием proposal.zip с документом PDF, до 3 декабря, 23:59

5.2. Предварительная демонстрация и обзор кода (code review) (10-12 декабря)

В течение трех дней после подачи вашего проектного предложения будут примерно 30-минутные собеседования. Вы должны продемонстрировать, как работают некоторые части системы. Вы покажете свой код, и обсудите, как завершить проект. Этот обзор будет составлять 15% от общей оценки.

5.3. Финальная демонстрация и обзор кода (17-19 декабря)

После первой встречи планируется вторая 30-минутная встреча. Вы продемонстрируете приложение и проведете обзор кода, включая то, как вы проводили тестирование. Это будет составлять 60% от общей оценки.

5.4. Отправка готового приложения и презентации (24 декабря)

В этот день будет демонстрация своей работы (добровольно). Проводится для обмена опытом и апробирования приложения. Проекты, получившие наибольшее одобрение всех одногруппников, получат призы.

Вы должны сдать zip архив в системе до 24 декабря до 16.40. Архив должен содержать весь исходный код, который вы написали для проекта, а также файл README.txt, в котором объясняется, как запускать проект, запускать все написанные вами тесты и т. д.

Приложение 4.

Технологическая карта дисциплины

Проектирование распределенных информационных систем

Название модулей дисциплины согласно РПД	Контроль	Форма контроля	зачетный	зачетный	График контроля
Модуль 1. Базовые знания по распределенным системам	текущий	Сдача практических работ №1	4	8	4
	рубежный	Контрольная работа №1	7	7	5
Модуль 2. Масштабируемость, согласованность данных и транзакции	текущий	Сдача практических работ №2 и №3	6	14	9
	рубежный	Контрольная работа №2	7	9	10
Модуль 3. Обработка больших данных	текущий	Сдача практической работы №4, №5	5	9	14
	рубежный	Контрольная работа №3	8	9	15
ВСЕГО за семестр			40	70	
Промежуточный контроль (зачет)			20	30	
Семестровый рейтинг по дисциплине			60	100	

85 – 100 баллов– «отлично»

70 – 84 баллов– «хорошо»

60-69 баллов– «удовлетворительно»

менее 60 баллов– «неудовлетворительно»

ПРИЛОЖЕНИЕ 3

Курсовой проект: Создание собственной распределенной системы

1. Общий обзор

В курсовом проекте требуется разработать, внедрить и тщательно оттестировать распределенную систему посредством реализация вашего собственного приложения, такого как, например, многопользовательская игра, инструмент для совместной работы или другой системы с поддержкой распределенных транзакций. Возможна совместная работа над проектом (не более 2-х человек). Однако, прежде чем принимать какие-либо решения, пожалуйста, ознакомьтесь с данным документов и политикой сдачи проекта, чтобы избежать сложностей во время сдачи проекта и срыва сроков проекта.

2. Требования к проекту

Поскольку это курс распределенных систем, мы хотим, чтобы у вашего проекта были «интересные» функции с точки зрения распределенных систем. Вот некоторые важные свойства, которые должна иметь ваша система:

- Ваша система должна включать как протокол Paxos, так и поддерживать Two-Phase Commit.
- Система должна поддерживать несколько автономных агентов (как человеческих, так и автоматизированных), конкурирующих за общие ресурсы и выполняющих обновления в режиме реального времени общего состояния системы.
- Состояние системы должно быть распределено между несколькими клиентскими или серверными узлами. Единственным централизованным сервисом должен быть тот, который поддерживает пользователей, регистрирующих, добавляющих или удаляющих клиентов, серверы, и другие поддерживающие задачи.
- Система должна быть надежной. Она должна иметь возможность продолжить работу, даже если на одном из узлов произошел сбой. Должна быть возможность восстановить состояние узла после сбоя, чтобы она мог возобновить работу.
- Если у вас готовый проект или наработки, это не связано с Paxos или 2PC, но, они соответствует целям этого курса и проекта, то вы можете получить разрешение от преподавателя, но сначала должно быть обсуждение проекта.

Разрешается выбрать направление самим, предоставляется широкий выбор по реализации и технологиям, но есть общие требования, которым должна соответствовать система:

- Вы можете использовать любой язык программирования.
- Поскольку трудно предсказать, насколько сложно будет разработать новую систему, система должна быть представлена в виде набора «уровней», где базовый уровень — это то, в чем вы уверены, что можете завершить, а дополнительные уровни добавляют дополнительные функции, как на уровне приложения, так и на уровне системы. Например, ваши усовершенствования системы могут включать в себя повышение уровня отказоустойчивости, уменьшение времени восстановления или добавление функций для повышения масштабируемости или безопасности.
- Вы можете использовать внешние библиотеки, если хотите (за исключением расширенных распределенных системных пакетов, таких как библиотеки, которые реализуют протокол Paxos или Two-Phase Commit). Вы также можете использовать внешние пакеты и SDK для поддержки приложения или пользовательского интерфейса (например, графические интерфейсы / фреймворки GUI, пакеты обработки изображений / видео / аудио, API т.д.).

3. Оценка проекта

Ваша система будет оцениваться в первую очередь как распределенная система. Основная оценка будет зависеть от того, имеет ли ваша система следующие атрибуты:

- Это должна быть «интересная» распределенная система, использующая Paxos или Two-Phase Commit протокол (и, возможно, другие алгоритмы, которые мы рассмотрели в курсе, например, для распределенной синхронизации, репликации, отказоустойчивости и восстановления, безопасности и т. д.).
- Программное обеспечение должно быть хорошо спроектировано и хорошо реализовано с точки зрения общей архитектуры и деталей реализации.
- Должны быть разработаны и применены систематические процедуры тестирования, такие как модульные тесты и системные тесты.
- Система должна работать надежно и с хорошей производительностью, особенно в части управления сбоями. У вас должен быть хотя бы один тестовый пример, демонстрирующий устойчивость к ошибкам, который вы можете показать в пояснительной записке к проекту.

Важно, но можно включать в проект во вторую очередь:

- Насколько интересно приложение? Это интересная игра, полезный инструмент или решение для реальной проблемы?
- Как хорошо выглядит приложение: есть ли у него приятный интерфейс или привлекательные визуальные эффекты?

Когда оценивается ваш проект, мы ожидаем увидеть исходный код, как он запускается, и посмотреть, как выполняются тесты, разработанные вами.

4. Идеи проектов

Следующий список проектов призван стимулировать ваше воображение. Можно подойти с чем-то совершенно другим. Обратитесь к преподавателю, если вы не уверены соответствии требованиям.

- Проекты с многопользовательским компонентом в режиме реального времени подходят для реализации протокола Paxos - вы можете либо использовать отдельных пользователей в кворуме Paxos, либо использовать Paxos для реплицированного состояния, в котором хранятся обновления состояний системы.
- Совместное редактирование документов в стиле документов Google. Система должна поддерживать редактирование и просмотр в режиме реального времени несколькими участниками. Для отказоустойчивости должны поддерживаться множественные реплики. Кэширование и/или миграция копий могут быть полезны для минимизации времени отклика приложения.
- Игры-симуляции, в стиле The Sims или Farmville. Состояние системы должно быть разделено пространственно, с репликацией для отказоустойчивости.
- Многопользовательская игра в режиме реального времени - здесь много возможностей.
- Система бронирования авиабилетов. Каждая авиакомпания будет поддерживать собственную коллекцию серверов с достаточной репликацией состояния, чтобы обеспечить автоматическое переключение. Должна быть возможность забронировать путешествие, в котором участвуют несколько авиакомпаний.
- Система регистрации с малой временной задержкой. Например, просмотр множества RSS-каналов и отправка всем подписчикам почтовых уведомлений при обновлении. Интерфейс взаимодействия как обычными RSS-каналами, так и с сервисом обновления новостей Google. Требуется реализация репликации и разбиения состояний системы мониторинга так, чтобы она могла масштабироваться и выдерживать сбои узлов.

- Проекты с участием множества агентов или пользователей, которые координируют свои обновления, и которые могут отображаться на карте (например, Карты Google). Можно интегрировать с Google Maps, Yandex Map, 2GIS и т.д.. API этих систем помогают упростить работу с графическим интерфейсом.
- Большие проекты, связанные с созданием приложений и игр для Android и iOS.
- Внедрение распределенной файловой системы, которая делает что-то интересное. Возможно, вы хотите хранить MP3-файлы или фильмы. Или, возможно, для чего-то совершенно другого. Какие ограничения и критерии оптимизации у вас есть?
- Решение проблемы, которая действительно вас задевает или что вы хотели бы видеть, - возможно, ваша новая идея стартапа!

5. Сроки проекта

Проект имеет четыре срока сдачи, каждый из которых представлен ниже:

5.1. Проектное предложение (3 декабря до 23:59)

Прежде чем отправить свой проект, вы должны представить проектное предложение, состоящее из следующих позиций:

- Описание приложения.
- Общая структура реализации.
- Распределенные системные функции и алгоритмы, которые вы намереваетесь реализовать.
- Ваш план тестирования системы.
- Как вы намереваетесь реализовать свою систему, представленную по уровням.
- График того, как вы планируете выполнять дизайн и реализацию проекта.

Этот документ должен составлять от трех до четырех страниц, и он будет составлять 25% от общей оценки. Рекомендуется задать вопросы на этапах планирования, чтобы получить некоторые отзывы об идеях до сдачи вашего предложения.

Чтобы отправить свое предложение, вы должны сдать документ в системе с названием proposal.zip с документом PDF, до 3 декабря, 23:59

5.2. Предварительная демонстрация и обзор кода (code review) (10-12 декабря)

В течение трех дней после подачи вашего проектного предложения будут примерно 30-минутные собеседования. Вы должны продемонстрировать, как работают некоторые части системы. Вы покажете свой код, и обсудите, как завершить проект. Этот обзор будет составлять 15% от общей оценки.

5.3. Финальная демонстрация и обзор кода (17-19 декабря)

После первой встречи планируется вторая 30-минутная встреча. Вы продемонстрируете приложение и проведете обзор кода, включая то, как вы проводили тестирование. Это будет составлять 60% от общей оценки.

5.4. Отправка готового приложения и презентации (24 декабря)

В этот день будет демонстрация своей работы (добровольно). Проводится для обмена опытом и апробирования приложения. Проекты, получившие наибольшее одобрение всех одногруппников, получат призы.

Вы должны сдать zip архив в системе до 24 декабря до 16.40. Архив должен содержать весь исходный код, который вы написали для проекта, а также файл README.txt, в котором объясняется, как запускать проект, запускать все написанные вами тесты и т. д.

Приложение 4.

Технологическая карта дисциплины Проектирование распределенных информационных систем

Название модулей дисциплины согласно РПД	**Контроль**	**Форма контроля**	**зачетный**	**зачетный**	**График**
Модуль 1. Базовые знания по распределенным системам	текущий	Сдача практических работ №1	4	8	4
	рубежный	Контрольная работа №1	7	7	5
Модуль 2. Масштабируемость, согласованность данных и транзакции	текущий	Сдача практических работ №2 и №3	6	14	9
	рубежный	Контрольная работа №2	7	9	10
Модуль 3. Обработка больших данных	текущий	Сдача практической работы №4, №5	5	9	14
	рубежный	Контрольная работа №3	8	9	15
ВСЕГО за семестр			40	70	
Промежуточный контроль (зачет)			20	30	
Семестровый рейтинг по дисциплине			60	100	

85 – 100 баллов– «отлично»

70 – 84 баллов– «хорошо»

60-69 баллов– «удовлетворительно»

менее 60 баллов– «неудовлетворительно»