Реферат

Хапов К. В., Высокопроизводительный сервер для p2p сети, выпускная квалификационная работа: стр. 48, рис. 10

Ключевые слова: сервер, потокобезопасность, сеть, Python, C, граф, распределенная система, узел.

В работе рассматривается написание программы для передачи блоков данных между узлами распределенной системы на языке C. Решаются задачи потокобезопасности и оптимальности программы. Обсуждается написание автотестов для полученного решения.

Также в работе сравниваются две модели передачи данных по сети: от точки к точке, от множества точек к точке.

Khapov K. V., High-performance server for p2p network, graduation qualification work: p. 48, Fig. 10

The paper deals with the writing of a program for transferring data blocks between nodes of a distributed system in the C language. The problems of flow safety and optimality of the program are solved. The writing of autotests for the solution obtained is discussed.

The paper also compares two models of data transmission over the network: from point to point, from multiple points to point.

Содержание

Обозначения и сокращения

p – вероятность появления ребра между двумя случайными вершинами графа

c – контекст операции

Введение

RiDE (RiDE is distributable environment) – исследовательский проект о том, как программировать большие распределенные системы [1].

Высокопроизводительные вычисления приходят на помощь в тех случаях, когда нужно сократить время расчётов или получить доступ к большему объёму памяти. Например, ваша программа может проводить необходимые вычисления в течение недели, но вам нужно получить результаты завтра. Если разделить эту программу на части и выполнять каждую из них на отдельном узле, то теоретически можно ускорить расчёты пропорционально числу вовлечённых узлов. Но это только теоретически, а на практике этому всегда что-то мешает (например, неоднородность системы).

Примерами высокопроизводительных вычислений могут быть: инженерные расчеты, доказательства теорем с помощью компьютера, научные вычисления. Эти задачи решаются с помощью распределенных систем (таких, как суперкомпьютер, например). RiDE помогает программировать такие системы. RiDE предоставляет удобный язык и набор утилит. Одна из составляющих RiDE – утилита по передаче данных с одного компьютера (узла) на другой.

Постановка задачи работы

Имеется сеть из нескольких узлов. На каждом из них хранятся некоторые данные, которые делятся на блоки по 1Мб. Требуется:

- Написать программу, с помощью которой можно передавать данные между двумя компьютерами. Данные, которые нужно передать, однозначно идентифицируются номером блока, смещением в блоке и длинной данных.
- Сравнить 2 способа организации транспортировки пакетов по сети: от одного узла к одному, от нескольких узлов к одному. Сравнить среднюю и максимальную загруженности сети.

Программа должна реализовывать следующий протокол. Программа принимает «команды» - сообщения, которые интерпретируются программой как команды на выполнение (см. ниже). Каждая команда идентифицируется кодом – первым байтом в сообщении. Для передачи команд используется протокол UDP.

В передаче данных участвуют 3 компьютера:

1. Инициатор – компьютер, который инициирует передачу данных. Он знает, где лежат данные и куда их нужно передать.
2. Отправитель – компьютер, который хранит нужные данные и который передает их другому компьютеру (получателю).
3. Получатель – конечная точка передачи данных.

Программа должна отслеживать таймауты операций, чтобы не тратить ресурсы на обработку ошибочных команд.

Команды протокола

PREPARE

Команда *PREPARE* – команда, которую получает получатель от инициатора. Сообщает, что получатель в скором времени получит данные. Подтверждает хранение и дальнейший прием данных этого блока. Так же указывает, когда получателю следует закончить прием. Формат команды:

1. 2 байта – номер блока данных. Первый байт – младший.
2. 2 байта – длина данных. Первый байт – младший.

Код команды – 0x10.

PUT

Команда *PUT* – команда, которую получает получатель от отправителя. Сообщает получателю данные и место их сохранения. Формат команды:

1. 2 байта – номер блока данных. Первый байт – младший.
2. 2 байта – смещение в блоке данных. Первый байт – младший.
3. 2 байта – длина данных. Первый байт – младший.
4. [Length] байтов – данные.

Код команды – 0x20.

Принцип работы: получая команду *PUT*, получатель кладет данные в указанное место в памяти. Контекст операции берется из дерева контекстов. Если контекста еще нет, значит, команда *PUT* пришла раньше команды *PREPARE*, поэтому создается новый контекст.

SEND

Команда *SEND* – команда, которую получает отправитель от инициатора. Сообщает отправителю, какие данные и куда нужно переслать.
Формат команды:

1. 4 байта – адрес цели. Представляет из себя IPv4 адрес. Первый байт – младший.

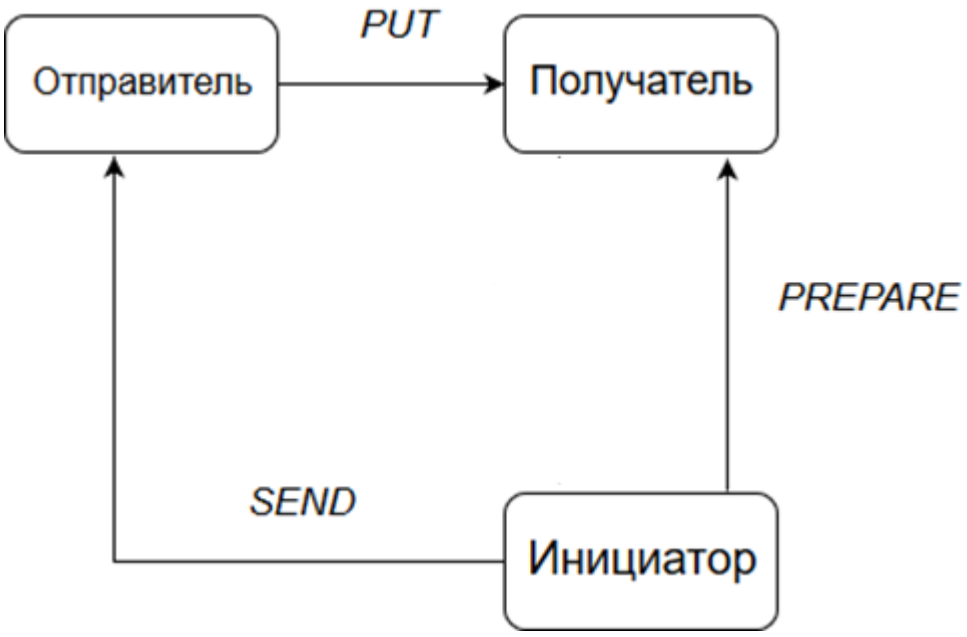

Рисунок 1.1 – Схема работы протокола

2. 2 байта – порт цели. Первый байт – младший.
3. 2 байта – номер блока данных. Первый байт – младший.
4. 2 байта – смещение в блоке данных. Первый байт – младший.
5. 2 байта – длина данных. Первый байт – младший.

Код команды – 0x30.

Принцип работы: получая команду *SEND*, отправитель понимает, что нужно отправить данные получателю. Для этого он шлет получателю команду PUT с данными, которые однозначно задаются номером блока памяти, смещением и размером данных. Схема взаимодействия между командами *SEND, PUT* и *PREPARE* показана на рисунке 1.1.

TIMEEXPIRED

Команда, сообщающая, что контекст имеет завершенные таймеры. Удаляет контекст, отправляет инициатору сообщение об ошибке. Данные, уже находящиеся в памяти, не обнуляются. Для программы эта команда является внутренней, т.е. предполагается, что она не может прийти от стороннего отправителя. Формат команды:

2 байта – номер блока данных. Первый байт – младший.

Код команды – 0x40.

RECEIVE

Команда, получающая данные с сервера. При обработке команды, сервер берет данные по указанному блоку и смещению и посылает на адрес отправителя команды.

1. 2 байта – номер блока. Первый байт – младший.
2. 4 байта – смещение в блоке.

Код команды – 0x50.

QUIT

Команда *QUIT* завершает работу приложения. Код команды – 0x00.

Способы и методы решения задачи

Команды как события, серверная часть, consumer и контексты

Программа состоит из 2-х частей – серверной части и consumer'а. Серверная часть занимается принятием и записыванием поступающих команд. Команды записываются в очередь команд. Consumer обрабатывает очередь команд – выполняет каждую команду поочередно (см. рисунок 2.1). Серверная часть и consumer работают в отдельных потоках. Для работы с потоками используется стандартная библиотека *pthread*.

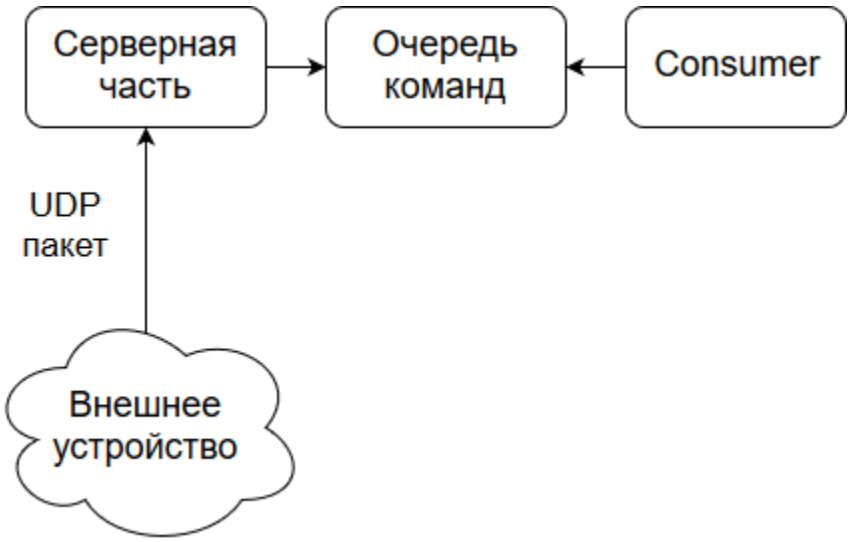

Рисунок 2.1 – Схема работы сервера

В процессе работы используются следующие глобальные переменные:

- *Queue* – очередь команд
- *BinaryHeap* – бинарная куча, содержащая таймеры контекстов (см. ниже)
- *Btree* – б-дерево, хранящее контексты (см. ниже)

Т.к. процесс передачи блока данных задействует несколько команд, то у него должен быть некоторый контекст, который описывает промежуточное

состояние процесса. Процесс можно представить в виде конечного автомата, в котором роль состояния выполняет контекст, а переходы между ними выполняются по командам *PUT*, *SEND*, *PREPARE* и *TIMEEXPIRED* (см. рисунок 2.2). Контекст хранит в себе номер блока, объем принятых данных, адрес инициатора, таймеры и другую информацию. Как написано выше, все контексты хранятся в переменной *Btree*.

Схема работы consumer'а (см. приложение 1).

1. Проверяется очередь команд *Queue*. Если она пуста, ничего не делаем и повторяем этот шаг.
2. Из очереди команд достаем верхнюю и передаем ее в соответствующий обработчик.
3. После обработки команды переходим на шаг 1.

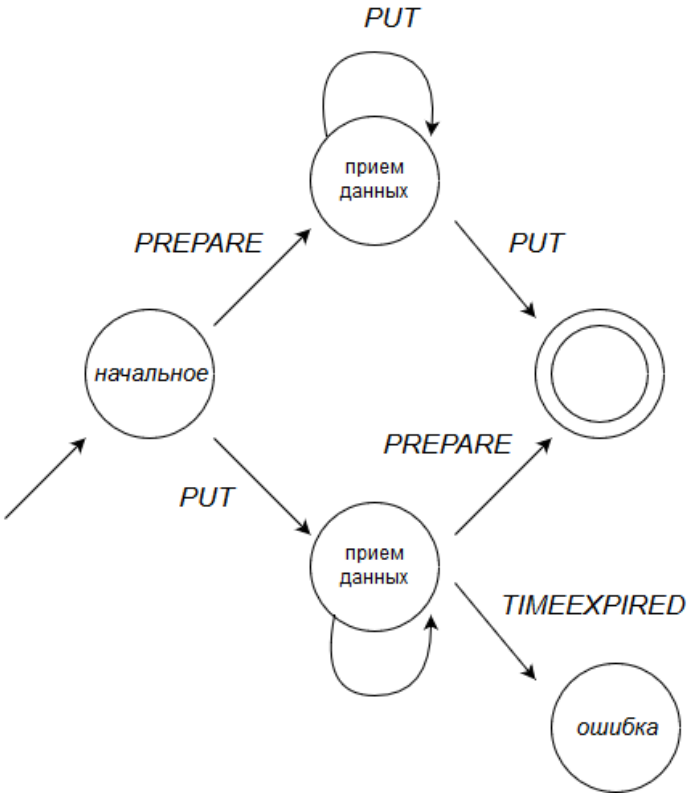

Рисунок 2.2 – Представление контекста операции как конечного автомата

В цикле consumer’а располагается «точка расширения» - чтобы добавить новый обработчик команды, достаточно вставить проверку на принадлежность команды вашему обработчику и его код. Таким образом, реализуется паттерн «цепочка обязанностей» [2].

Схема работы серверной части следующая (см. приложение 2):

1. *c* = верхний элемент в *BinaryHeap*.
   Если в *BinaryHeap* пуста, *c* будет равно *NULL*.
2. Устанавливается время ожидания *t*. Если контекст *c* существует, то *t* равно значению времени истекания таймера контекста *c*. Иначе *t* = 1000. В итоге, в переменной *t* хранится время ожидания в миллисекундах.
3. В течении времени *t* программа прослушивает порт, указанный на старте программы. Если за время *t* не пришло ни одной команды, то в очередь команд записывается новая команда *TIMEEXPIRED* с указанием контекста *c* (если он существует). Иначе новая команда записывается в очередь команд, после чего начинается следующая итерация цикла.

Для работы со временем и сетью используется стандартная функция *poll*. Она позволяется слушать файлы (в данном случае сокеты, которые на уровне ОС являются файлами) с определенным таймером, в качестве которого и передается время *t*.

Таймеры и ошибки

Для обработки некоторых исключительных ситуаций введены таймеры. Всего в программе предусмотрено 3 связи с таймерами – от отправителя к получателю (*команда PUT*), от инициатора к получателю (*команда SEND*), от инициатора к отправителю (*команда PREPARE*). Для определения ближайшего к истечению времени используется *BinaryHeap*.

В процессе работы могут возникать различные исключительные ситуации. Вот некоторые из них и их решения представлены ниже:

- UDP пакет не дошел до получателя. Решение – через некоторое время сработает таймер (отправитель-получатель).
- В команде записан неверный блок данных. Решение – таймаут (отправитель-получатель, инициатор-получатель)
- Сумма Смещения в блоке данных и длины данных превышает размер блока. Решение – обработчики команд валидируют данные, которые получают.
- Номер блока превышает допустимое значение. Решение – обработчики команд валидируют данные, которые получают.
- Ответ до инициатора не дошел. Решение – инициатор сам решает, что делать – сколько ждать, повторять ли операцию.
- Неправильные данные. Эту ситуацию сервер никак не обрабатывает.

Заметим, что отсутствие контекста в случае его ожидания ошибкой не является. Запрос несуществующего контекста заканчивается его инициализацией.

Потокобезопасность

Серверная часть и Consumer одновременно работают с очередью команд и деревом контекстов. Для обеспечения их потокобезопасности используется системная библиотека *pthread*. Структура данных *pthread_mutex_t*, определенная в этой библиотеке, описывает мьютекс – единицу синхронизации потоков. Метод *pthread_mutex_lock(pthread_mutex_t)* блокирует мьютекс, метод *pthread_mutex_unlock(pthread_mutex_t)* открывает мьютекс.

Т.к. каждый процесс передачи данных может длиться долго, нерационально ждать выполнения текущего, чтобы приступить к следующему. Первый вариант избежать ожидания – выполнять каждый процесс в отдельном потоке с помощью пула потоков (англ. thread pool). Однако это усложняет задачу, т.к. надо синхронизировать потоки между собой. Контексты позволяют избежать проблем с многопоточностью. Каждый процесс протекает несколькими командами, которые выполняются синхронно, т.е. в любой момент времени выполняется только одна команда.

Структуры данных

BinaryHeap – бинарная куча, алгоритмом сортировки которой является минимизация значения таймера контекста, а значениями – ссылки на контексты. При выполнении команд (*PUT, SEND, PREPARE*) значения таймеров обновляются. При обновлении элемента кучи контекст с наиболее близким к истеканию временем автоматически помещается наверх кучи. Для написания структуры данных использовались алгоритмы из [3] (см. приложение 3).

Для хранения контекстов используется *Btree* – б-дерево. Б-дерево представляет из себя сбалансированное, сильно ветвистое дерево. В *Btree* реализованы операции поиска, вставки и удаления элемента. Ключом является идентификатор контекста. Структура используется из-за высокой скорости доступа к данным. Недостатком структуры является отсутствие эффективного способа обхода дерева, что в данной ситуации не имеет значения. Структура реализована на основе алгоритмов, данных в [4] и [5] (см. приложение 4).

Взаимодействие двух независимых частей сервера ведется через *Queue* – потокобезопасную очередь (*FIFO*). Потокобезопасность обеспечивается с помощью библиотеки *pthread*, как было написано выше. Реализованы операции вставки, взятия, проверки на пустоту (см. приложение 5).

Модель сети

Описание модели

Для определения оптимальности выбранной стратегии была создана модель сети (см. приложение 6).

Модель представляет из себя невзвешенный неориентированный граф. Вершинами представлены узлы системы, ребрами – связи между ними. В модели выполняются «задачи». Задача представляет из себя абстракцию над передачей данных по сети. Задача имеет цель – номер узла, и номер блока данных. В каждый момент времени *t*:

- Выполняется очередная задача. Для этого определяются узлы-отправители, на которых лежит нужный блок данных; из этих узлов-отправителей определенным образом выбираются несколько (d>=1), на которых создаются пакеты – единицы передачи данных. Все задачи генерируются заранее и передаются на вход модели в качестве параметра (для обеспечения одинаковых условий работы для каждой модели).
- После создания задачи генерируются новые пакеты. Случайным образом выбирается вершина, на которой появится блок данных со следующим номером. Создание новых блоков данных нужно для более точного моделирования сети.

Каждое ребро в графе, представляющее связь между двумя узлами, имеет свою очередь (FIFO) пакетов. За один такт передается один пакет. Пакеты передаются по кратчайшему пути – пути, состоящем из наименьшего количества

Сравниваются 2 модели поведения сети, назовем их p2p-модель – m2p-модель. Отличаются они генераторами пакетов, которые передаются на вход модели. P2p-генератор генерирует последовательную передачу пакетов между вершинами. Т.е., если нам необходимо передать пакет размером *v* из вершины *n* на вершину *m*, то пакет преодолеет этот путь целиком, т.е. по сети пройдет один пакет объёмом *v* (см. рисунок 3.1). M2p-генератор генерирует параллельную передачу пакетов. Если мы хотим передать пакет объёмом *v* на вершину *m*, с *k* других попарно различных вершин на вершину *m* передается пакет, размером *v/k* (см. рисунок 3.1).

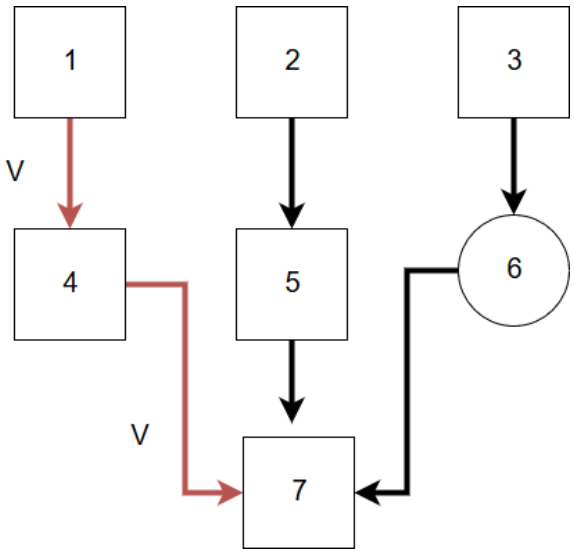

Рисунок 3.1 – Схема p2p-модели. На ребрах указан объем передаваемых данных.

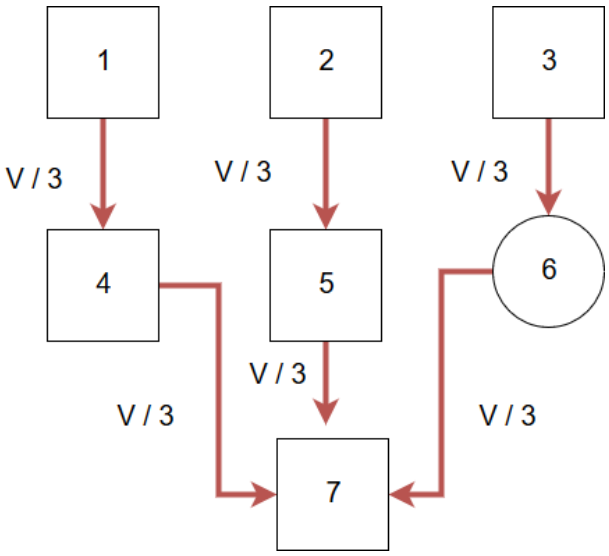

Рисунок 3.2 – Схема m2p-модели. На ребрах указан объем передаваемых данных.

Создание модели

Для создания графа и нахождения путей между вершинами используется библиотека *networkx*. Она представляет методы для генерации случайных графов, которые отличаются списком параметров и видами графов, которые получаются на выходе. Я пользовался методом *gnp_random_graph*, который

принимает на вход вероятность p появления ребра между двумя случайными вершинами, а также количество вершин в будущем графе.

Во время инициализации модели ищутся кратчайшие пути между всем вершинами, на основе чего составляется словарь путей между вершинами:

$\forall i \forall j, i \neq j: paths[i][j] = GetPath(i, j)[1]$,
где *GetPath* – функция поиска пути между двумя вершинами графа.

По итогу, переменная *paths* будет представлять из себя словарь, ключом которого будут являться вершины i и j, а значением будет вершина k в пути $i – k – … – j$.

Таким образом, всего в модель в качестве параметров передаются:

- Граф
- Время работы модели (количество тактов)
- Начальное количество блоков данных
- Заранее сгенерированный список задач
- Заранее сгенерированный список вершин, на которых в момент времени t будет происходить появление нового блока данных
- Генераторы пакетов
- Степень распределенности системы

Результаты моделирования

Эксперимент первый

- Количество вершин фиксировано, от 10 до 100 включительно
- Степень распределенности системы фиксирована, 5

- Количество тактов работы модели фиксировано, 500
- Вероятность *p* появления ребра меняется в пределах от 0.1 до 1 с шагом 0.1

При увеличении вероятности *p* появления ребра средняя и максимальная забитости узлов понижались. Причем, чем больше узлов *n* участвовало в эксперименте, тем меньшей была разница между максимальным и минимальным значениями каждого графика (см. рисунок 3.3 и рисунок 3.4).

Исключение составляют графики при *n* = 10. В этом случае при *p* = 0.2 и *p* = 0.3 значения графиков значительно превышают значения при других *p* (см. Рисунок 3.5).

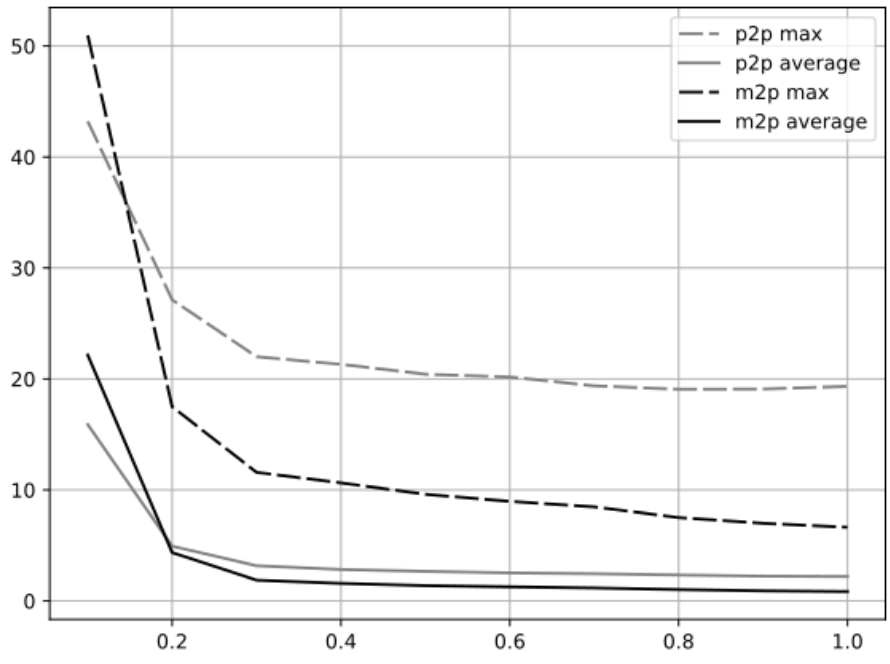

Рисунок 3.3. – Графики при n = 20

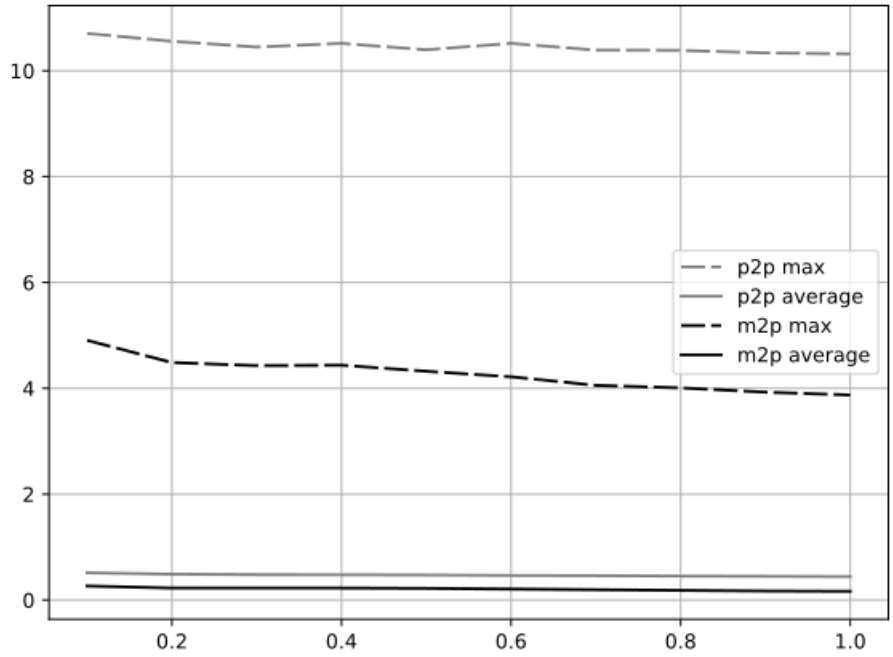

Рисунок 3.4 – Графики при n = 100

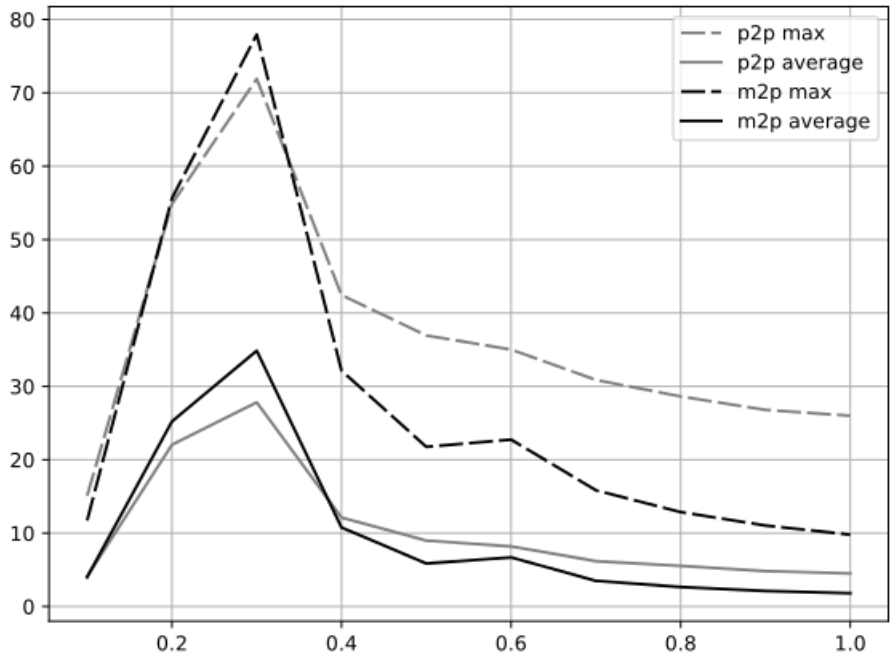

Рисунок 3.5 – Графики при n = 10

Эксперимент второй

- Количество вершин фиксировано, 500

- Количество тактов работы модели фиксировано, от 100 до 1000 включительно
- Вероятность *p* появления ребра фиксирована, 0.5
- Степень распределенности системы меняется от 1 до 20 включительно

При увеличении степени распределенности m2p-модель показывает лучшие результаты. Однако по графикам видно, что этот параметр не имеет смысла увеличивать больше 5 (см. рисунок 3.6).

Итоги моделирования

Анализируя полученные графики, можно сказать: организация передачи данных способом множество точек – точка является более выгодным, т.к. максимальная загруженность узлов меньше, чем максимальная загруженность узлов при способе организации точка – точка. Средняя загруженность, однако, почти не зависит от выбора способа передачи.

Также, в ходе экспериментов было установлено, что следующие параметры не влияют на результат моделирования: количество заранее созданных блоков данных, количество тактов работы модели (больше 100) (см. рисунки 3.6 и 3.7).

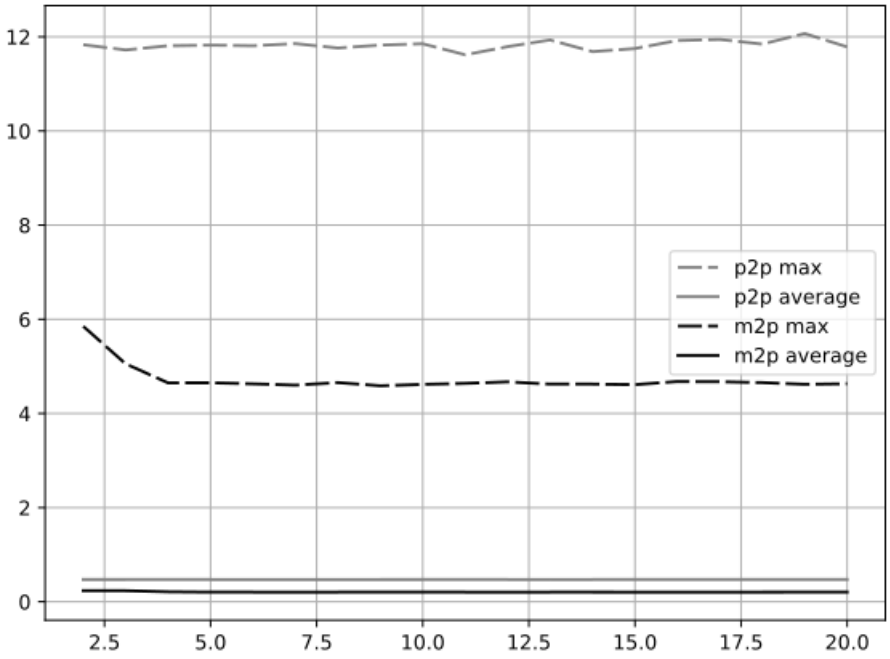

Рисунок 3.6 – Графики при t = 1000

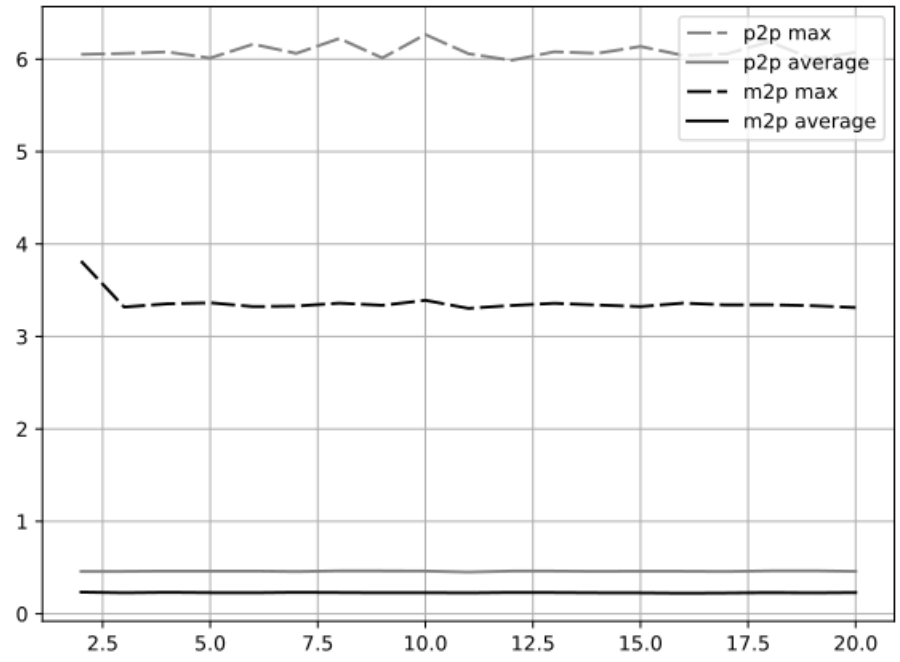

Рисунок 3.7 – Графики при t = 100

Тестирование

Для обеспечения работоспособности сервера необходимы тесты. Для этой цели хорошо подходит язык программирования Python. Для этого языка разработано большое количество библиотек, в том числе библиотек для тестирования.

Для написания тестов я использовал библиотеку *unittest*. Она входит в стандартный пакет библиотек Python. Чтобы написать тест с помощью этой библиотеку, достаточно создать класс-наследник *unittest.TestCase*. Методы этого класса, начинающиеся с "*test*", будут являться самими тестами. Если создать метод "*setUp*", он будет выполняться перед запуском каждого теста. В методе *setUp* выполняется старт тестовых серверов. Метод "*tearDown*" будет выполняться по завершению каждого теста. В методе *tearDown* выполняется закрытие всех тестовых серверов.

Для удобства и краткости кода создан класс *Client*. Экземпляр этого класса представляет собой клиент, с помощью которого можно отправлять команды сервера. На вход в конструктор подается адрес и порт, куда клиент будет слать команды. Клиент имеет несколько метод, каждый их которых представляет отправку соответствующей команды:

- Client.send
- Client.put
- Client.receive
- Client.quit

Для проверки работоспособности были написаны следующие тесты (см. приложение 7):

- *test_put_and_receive*

  Тест, проверяющий простой сценарий отправки данных на сервер с помощью команды *put* и дальнейшее получение этих данных командой *receive*.

  test_send_and_receive
- Тест, проверяющий простой сценарий отправки данных на сервер с помощью команды *send* и дальнейшее получение этих данных командой *receive*.

Заключение

Итак, в качестве системной утилиты для RiDE была разработана утилита передачи данных между узлами системы.

Программа написана на языке СИ стандарта СИ99. Особенности работы программы:

- Работа с многопоточностью с помощью системной библиотеки *pthread*
- Работа с сетью с помощью системной библиотеки *socket*

Структуры данных, реализованные в работе:

- B-дерево (вставка, удаление, поиск) (см. приложение 4)
- Потокобезопасная очередь (вставка, взятие) (см. приложение 5)
- Двоичная куча (вставка, удаление, взятие, взятие с удалением, перестраивание) (см. приложение 3)

Также, было произведено сравнение двух способов передачи пакетов данных по сети: способ точка – точка, способ множество точек – точка. Для этого была написана модель на языке Python. Анализируя полученные результаты, было выяснено, что второй способ загружает сеть в меньшей степени.

В качестве проверки работоспособности полученного решения были написаны автотесты.

Список использованных источников

1. *Бахтерев М.* (2016) Распределенное программирование в системе RiDE [Доклад]// YouTube. 29 декабря (https://www.youtube.com/watch?v=pysxEMkPcb0) Просмотрено: 29.03.2019.
2. *Фримен Э., Фримен Э., Сьерра К., Бейтс Б.* Паттерны проектирования. СПб.: Питер, 2011. – 656 с.: ил.
3. *Двоичная куча* [Статья] // Википедия университета ИТМО. URL: https://neerc.ifmo.ru/wiki/index.php?title=Двоичная_куча (дата обращения: 02.04.2019)
4. *B-дерево* [Статья] // Википедия университета ИТМО. URL: https://neerc.ifmo.ru/wiki/index.php?title=B-дерево (дата обращения: 04.04.2019)
5. *B-tree* [Статья] // «Хабр» - Ресурс для IT-специалистов. URL: https://habr.com/ru/post/114154/ (дата обращения: 05.04.2019)

Приложение 1 (consumer.c)

```
#include <stdio.h>
#include <stdlib.h>

#include "../headers/consumer.h"
#include "../headers/event_handlers/confirm_handler.h"
#include "../headers/event_handlers/prepare_event_handler.h"
#include "../headers/event_handlers/put_event_handler.h"
#include "../headers/event_handlers/send_event_handler.h"
#include "../headers/event_handlers/time_expired_handler.h"
#include "../headers/event_handlers/receive_event_handler.h"
#include "../headers/data_structures/queue.h"

void handle_event(StsHeader* queue, Event* event, Node* context_btree, Heap* context_heap,
void* memory)
{
   Context* context = NULL;
   if(is_put_command(event->commandId))
   {
      context = handle_put_event(queue, context_btree, event, context_heap, memory);
   }

   if(is_prepare_event(event->commandId))
   {
      context = handle_prepare_event(event, context_btree, context_heap);
   }

   if(is_send_event(event->commandId))
   {
      handle_send_event(event, memory);
   }

   if(is_receive_event(event->commandId))
   {
      handle_receive_event(context_btree, event, memory);
   }

   if(is_time_expired_event(event->commandId))
   {
      handle_time_expired_event(queue, context_btree, event);
   }

   if(context != NULL && need_confirm(context))
   {
```

```
        confirm(context);
    }
}

void* start_consumer(void* args)
{
    StsHeader* queue = ((ConsumerArgs*)args)->queue;
    Node* context_btree = ((ConsumerArgs*)args)->context_btree;
    unsigned char* memory = ((ConsumerArgs*)args)->memory;
    Heap* heap = ((ConsumerArgs*)args)->context_heap;

    int commandId = -1;
    do
    {
        if(StsQueue.empty(queue) == 0)
        {
            Event* event = (Event*) StsQueue.pop(queue);
            commandId = event->commandId;
            handle_event(queue, event, context_btree, heap, memory);
            free(event);
        }
    } while(commandId != 0);

    return 0;
}
```

Приложение 2 (server.c)

```
#include <netinet/in.h>
#include <stdio.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <string.h>

#include "../headers/client.h"
#include "../headers/data_structures/queue.h"
#include "../headers/event.h"
#include "../headers/server.h"
#include "../headers/data_structures/heap.h"

#define MAX_BUFFER_SIZE 256

void* start_server(void* args)
{
   StsHeader* queue = ((ServerArgs*)args)->queue;
   Heap* heap = ((ServerArgs*)args)->context_heap;

   unsigned char buffer[MAX_BUFFER_SIZE];
   struct sockaddr_in other_addr_port;
   socklen_t other_addr_len = sizeof(other_addr_port);

   while(1)
   {
      Context* context = BinaryHeap.peek(heap);
      int timeout;
      if (context != NULL)
         timeout = (context->time_for_expired - (unsigned int)time(0)) * 1000;
      else
         timeout = 1000;

      int res = client_recvfrom(buffer, MAX_BUFFER_SIZE, (struct sockaddr *) &other_addr_port,
&other_addr_len, timeout);

      if (res == 0)
      {
         if (context != NULL)
         {
            BinaryHeap.delete(heap, context);

            Event* event = (Event*) malloc(sizeof(event));
            event->commandId = 0x40;
            event->data[0] = (unsigned char) context->block_number;
            event->data[1] = (unsigned char) context->block_number << 8;
```

```
            StsQueue.push(queue, event);
        }

        continue;
    }

    if (res == -1)
    {
        printf("Error\n");
        continue;
    }

    Event* event = (Event*) malloc(sizeof(Event));
    event->sender_addr_port = other_addr_port;
    event->commandId = buffer[0];
    for(int i = 0; i < MAX_BUFFER_SIZE - 1; i++)
        event->data[i] = buffer[i+1];
    StsQueue.push(queue, event);

    if (event->commandId == 0)
        break;
  }

  int status;
  status = client_shutdown();
  //printf("shutdown status: %d\n", status);

  return 0;
}
```

Приложение 3 (heap.c)

```
#include <malloc.h>

#include "../../headers/data_structures/heap.h"

void add(Heap* heap, Context* item);
Heap* create_heap(size_t size);
void heapify(Heap* heap, int i);
Context* pop(Heap* heap);
Context* peek(Heap* heap);
void destroy(Heap* heap);
void delete(Heap* heap, Context* item);

// private
int greater(Context* original, Context* other);

void add(Heap* heap, Context* item)
{
   pthread_mutex_lock(&heap->lock);
   int i = heap->size;
   int parent = (i - 1) / 2;
   heap->data[i] = item;
   while(parent >= 0 && i > 0)
   {
      if(heap->data[i] > heap->data[parent])
      {
         Context* temp = heap->data[i];
         heap->data[i] = heap->data[parent];
         heap->data[parent] = temp;
      }
      i = parent;
      parent = (i - 1) / 2;
   }
   heap->size++;
   heapify(heap, 0);
   pthread_mutex_unlock(&heap->lock);
}

Heap* create_heap(size_t size)
{
   Heap* heap = (Heap*) malloc(sizeof(Heap));
   pthread_mutex_init(&heap->lock, NULL);
   for (int i = 0; i < size; i++)
   {
      heap->data[i] = 0;
   }
```

```
    return heap;
}

void heapify(Heap* heap, int i)
{
    int left, right;
    Context* temp;
    left = 2 * i + 1;
    right = 2 * i + 2;

    if (left < heap->size)
    {
        if(greater(heap->data[i], heap->data[left]))
        {
            temp = heap->data[i];
            heap->data[i] = heap->data[left];
            heap->data[left] = temp;
            heapify(heap, left);
        }
    }

    if (right < heap->size)
    {
        if(greater(heap->data[i], heap->data[right]))
        {
            temp = heap->data[i];
            heap->data[i] = heap->data[right];
            heap->data[right] = temp;
            heapify(heap, right);
        }
    }
}

Context* pop(Heap* heap)
{
    pthread_mutex_lock(&heap->lock);
    Context* item = heap->data[0];
    heap->size--;
    heap->data[0] = heap->data[heap->size];
    heap->data[heap->size] = NULL;
    heapify(heap, 0);
    pthread_mutex_unlock(&heap->lock);

    return(item);
}
```

```
Context* peek(Heap* heap)
{
   return heap->data[0];
}

void delete(Heap* heap, Context* item)
{
   pthread_mutex_lock(&heap->lock);
   for(int i = 0; i < heap->size; i++)
   {
      if (heap->data[i] == item)
      {
         heap->size--;
         heap->data[i] = heap->data[heap->size];
         heap->data[heap->size] = NULL;
         heapify(heap, i);
         pthread_mutex_unlock(&heap->lock);

         return;
      }
   }
}

void destroy(Heap* heap)
{
   pthread_mutex_destroy(&heap->lock);
}

const struct _BinaryHeap BinaryHeap = {
      create_heap,
      add,
      heapify,
      pop,
      peek,
      delete,
      destroy
};

int greater(Context* original, Context* other)
{
   if (original->time_for_expired > other->time_for_expired)
   {
      return 1;
   }

   return 0;
}
```

Приложение 4 (btree.c)

```
#include <stdlib.h>
#include "../../headers/data_structures/btree.h"
#include "../../headers/context.h"

// public
void insert(Node* root, Context* key);
const Node* search(const Node* node, int key);
Node* create_empty_node();
void remove(Node* root, int key);
Context* get_or_create(Node* root, int key);

// private
void divide_node(Node* node, Node* left, Node* right);
void internal_insert(Node* node, Context* key);
void insert_simple(Node *node, Context* key);
void insert_simple_internal(Node *node, Context* key, int with_children_moving); // костыль
void insert_child_instead_two(Node *node, Node *parent, int idx);
Node* remove_simple(Node *node, int hash);
Node* remove_internal(Node* node, Node* parent, int hash);
Node* remove_from_leaf(Node* node, Node* parent, int hash);
Node* union_simple(Node* left, Node* right);
Node* union_node_with_any_neighbour(Node* node, Node* parent);

int get_key_position(const Node *node, int hash);
Node* get_child_by_hash(const Node *node, int hash);
int get_key_position_to_insert(Node *node, Context* key);
int get_child_position(Node* node, Node* child);
int get_neighbour_for_remove(Node* node, Node* parent);

int get_hash(Context* context);
Context* get_context_by_block_number(const Node* node, int block_number);

void insert(Node* root, Context* key)
{
   internal_insert(root, key);
   if (root->key_length == 2 * M - 1)
   {
      Node* left = create_empty_node();
      Node* right = create_empty_node();
      Context* medium_key = root->keys[M -1 ];
      divide_node(root, left, right);
      Node* new_root = create_empty_node();
      insert_simple(new_root, medium_key);
      new_root->children[0] = left;
      new_root->children[1] = right;
```

```
        new_root->is_leaf = 0;
        free(root);
        *root = *new_root;
    }
}

const Node* search(const Node* node, int key)
{
    if (get_key_position(node, key) != -1)
        return node;

    Node* child = get_child_by_hash(node, key);
    if (child == NULL)
        return NULL;

    return search(child, key);
}

void remove(Node* root, int key)
{
    root = remove_internal(root, NULL, key);
    if (root->key_length == 0)
    {
        if (root->children[0] != NULL)
        {
            *root = *root->children[0];
        }
    }
}

Node* create_empty_node()
{
    Node* node = (Node*) malloc(sizeof(Node));
    node->key_length = 0;
    node->is_leaf = 1;
    for(int i = 0; i < 2 * M; i++)
    {
        node->children[i] = NULL;
        node->keys[i] = NULL;
    }

    return node;
}

Context* get_or_create(Node* root, int key)
{
    Context* context;
```

```
   const Node* node = Btree.search(root, key);
   if (node == NULL)
   {
      context = (Context*) malloc(sizeof(Context));

      context->received_length = 0;
      context->prepared = 0;
      context->expected_length = 0;
      context->block_number = key;
      context->confirmed = 0;
      context->has_timer = 0;
      context->time_for_expired = 0;

      Btree.insert(root, context);
   } else {
      context = get_context_by_block_number(node, key);
   }

   return context;
}

const struct _Btree Btree = {
      insert,
      search,
      create_empty_node,
      remove,
      get_or_create
};

// private

void internal_insert(Node* node, Context* key)
{
   if (node->is_leaf)
   {
      insert_simple(node, key);
   }
   else
   {
      Node* child = get_child_by_hash(node, get_hash(key));
      internal_insert(child, key);

      if (child->key_length == 2 * M - 1)
      {
         Node* left = create_empty_node();
         Node* right = create_empty_node();
         divide_node(child, left, right);
```

```
            Context* medium_key = child->keys[M - 1];
            insert_simple(node, medium_key);

            int insert_position = get_key_position(node, get_hash(medium_key));
            node->children[insert_position] = left;
            node->children[insert_position + 1] = right;
        }
    }
}

Node* remove_internal(Node* node, Node* parent, int hash)
{
    if (node->is_leaf)
    {
        return remove_from_leaf(node, parent, hash);
    }
    else
    {
        get_key_position(node, hash);
        Node* child = get_child_by_hash(node, hash);
        int idx = get_child_position(node, child);
        child = remove_internal(child, node, hash);
        if (child->key_length < M - 1)
        {
            union_node_with_any_neighbour(child, node);
            return node;
        }
        node->children[idx] = child;
        return node;
    }
}

// node1 and node2 - result of division
void divide_node(Node* node, Node* left, Node* right)
{
    for(int i = 0; i < M; i++)
    {
        if (i < M - 1)
        {
            left->keys[i] = node->keys[i];
            left->key_length++;
        }
        if (!node->is_leaf)
        {
            left->children[i] = node->children[i];
            left->is_leaf = 0;
        }
```

```
    }

    for(int i = M; i < 2 * M; i++)
    {
        if (i < 2 * M - 1)
        {
            right->key_length++;
            right->keys[i - M] = node->keys[i];
        }
        if (!node->is_leaf)
        {
            right->children[i - M] = node->children[i];
            right->is_leaf = 0;
        }
    }

    node->is_leaf = 0;
}

// node1 и node2 - ноды, которые получаются в результате переполнения
void insert_simple(Node *node, Context* key)
{
    insert_simple_internal(node, key, 1);
}

void insert_simple_internal(Node *node, Context* key, int with_children_moving)
{
    int insert_position = get_key_position_to_insert(node, key);

    for(int i = node->key_length + 1; i > insert_position; i--)
    {
        if (i < node->key_length + 1)
            node->keys[i] = node->keys[i - 1];
        if (!node->is_leaf && with_children_moving)
            node->children[i] = node->children[i - 1];
    }

    node->keys[insert_position] = key;

    node->key_length++;
}

Node* remove_from_leaf(Node* node, Node* parent, int hash)
{
    // if parent is NULL then node is root
    if (node->key_length > M - 1 || parent == NULL)
    {
```

```
        node = remove_simple(node, hash);
        return node;
    }
    else
    {
        int idx = get_child_position(parent, node);
        int pos = get_neighbour_for_remove(node, parent);
        if (pos == 0)
        {
            Node* new_node = union_node_with_any_neighbour(node, parent);

            remove_simple(new_node, hash);
            return new_node;
        }
        else
        {
            Context* neighbour_key;
            int parent_delimeter_key_idx;
            // если сосед - левый, берем его последний ключ
            if (pos == -1)
            {
                Node* neighbour = parent->children[idx - 1];
                neighbour_key = neighbour->keys[neighbour->key_length - 1];
                remove_simple(neighbour, get_hash(neighbour_key));
                parent_delimeter_key_idx = idx - 1;
            }
                // если сосед - правый, берем его первый ключ
            else
            {
                Node* neighbour = parent->children[idx + 1];
                neighbour_key = neighbour->keys[0];
                remove_simple(neighbour, get_hash(neighbour_key));
                parent_delimeter_key_idx = idx;
            }
            remove_simple(node, hash);
            insert_simple(node, parent->keys[parent_delimeter_key_idx]);
            parent->keys[parent_delimeter_key_idx] = neighbour_key;

            return node;
        }
    }
}

Node* remove_simple(Node *node, int hash)
{
    int remove_position = get_key_position(node, hash);
    for(int i = remove_position; i < node->key_length; i++)
```

```
    {
        node->keys[i] = node->keys[i + 1];
    }
    node->keys[node->key_length - 1] = NULL;
    node->key_length--;

    return node;
}

Node* union_simple(Node* left, Node* right)
{
    Node* result = create_empty_node();
    int i;
    for(i = 0 ; i < left->key_length; i++)
    {
        result->keys[i] = left->keys[i];
        result->key_length++;

        if (!left->is_leaf)
        {
            result->children[i] = left->children[i];
            result->is_leaf = 0;
        }
    }
    result->children[i] = left->children[i];

    for(i = 0; i < right->key_length; i++)
    {
        result->keys[i + left->key_length] = right->keys[i];
        result->key_length++;

        if (!right->is_leaf)
        {
            result->children[i + left->key_length + 1] = right->children[i];
            result->is_leaf = 0;
        }
    }
    result->children[i + left->key_length + 1] = right->children[i];

    return result;
}

Node* union_node_with_any_neighbour(Node* node, Node* parent)
{
    Node* left;
    Node* right;
    int parent_delimeter_key_idx;
```

```
    int idx = get_child_position(parent, node);
    if (idx > 0)
    {
        left = parent->children[idx - 1];
        right = node;
        parent_delimeter_key_idx = idx - 1;
    }
    else
    {
        right = parent->children[idx + 1];
        left = node;
        parent_delimeter_key_idx = idx;
    }

    Context* key_to_down = parent->keys[parent_delimeter_key_idx];
    Node* new_node = union_simple(left, right);
    insert_simple_internal(new_node, key_to_down, 0);
    remove_simple(parent, get_hash(key_to_down));
    // parent_delimeter_key_idx - чтобы вставлять на место левого
    insert_child_instead_two(new_node, parent, parent_delimeter_key_idx);

    return new_node;
}

// возвращает место между существующими ключами
int get_key_position_to_insert(Node *node, Context* key)
{
    int idx = 0;
    for(idx = 0; idx < node->key_length; idx++)
    {
        if (get_hash(node->keys[idx]) > get_hash(key))
            break;
    }

    return idx;
}

int get_child_position(Node* node, Node* child)
{
    for(int i = 0; i < node->key_length + 1; i++)
    {
        if (node->children[i] == child)
        {
            return i;
        }
    }
```

```
    return -1;
}

int get_key_position(const Node *node, int hash)
{
    for(int i = 0; i < node->key_length; i++)
    {
        if (get_hash(node->keys[i]) == hash)
            return i;
    }

    return -1;
}

Node* get_child_by_hash(const Node *node, int hash)
{
    if (node->key_length > 0 && hash < get_hash(node->keys[0]))
        return node->children[0];

    for(int i = 1; i < node->key_length; i++)
    {
        if (hash < get_hash(node->keys[i]))
            return node->children[i];
    }

    return node->children[node->key_length];
}

// возвращает -1, если сосед левый, 1- если правй, 0 - если не найден
int get_neighbour_for_remove(Node* node, Node* parent)
{
    int idx = get_child_position(parent, node);

    // вернуть ревого соседа
    if (idx > 0 && parent->children[idx - 1]->key_length > M - 1)
    {
        return -1;
    }

    // вернуть правого соседа
    if (idx < parent->key_length && parent->children[idx + 1]->key_length > M - 1)
    {
        return 1;
    }

    return 0;
}
```

```
void insert_child_instead_two(Node *node, Node *parent, int idx)
{
   parent->children[idx] = node;

   for(int i = idx + 1; i < 2 * M - 1; i++)
      parent->children[i] = parent->children[i + 1];
   parent->children[2 * M - 1] = NULL;
}

int get_hash(Context* context)
{
   return context->block_number;
}

Context* get_context_by_block_number(const Node* node, int block_number)
{
   for(int i = 0; i < node->key_length; i++)
   {
      if (node->keys[i]->block_number == block_number)
         return node->keys[i];
   }

   return NULL;
}
```

Приложение 5 (queue.c)

```
#include <stdlib.h>
#include <pthread.h>

#include "../../headers/data_structures/queue.h"

typedef struct StsElement {
   void *next;
   void *value;
} StsElement;

struct StsHeader {
   StsElement *head;
   StsElement *tail;
   pthread_mutex_t *mutex;
};

static StsHeader* create();
static StsHeader* create() {
   StsHeader *handle = malloc(sizeof(*handle));
   handle->head = NULL;
   handle->tail = NULL;

   pthread_mutex_t *mutex = malloc(sizeof(*mutex));
   handle->mutex = mutex;

   return handle;
}

static void destroy(StsHeader *header);
static void destroy(StsHeader *header) {
   free(header->mutex);
   free(header);
   header = NULL;
}

static void push(StsHeader *header, void *elem);
static void push(StsHeader *header, void *elem) {
   // Create new element
   StsElement *element = malloc(sizeof(*element));
   element->value = elem;
   element->next = NULL;

   pthread_mutex_lock(header->mutex);
   // Is list empty
   if (header->head == NULL) {
```

```
        header->head = element;
        header->tail = element;
    } else {
        // Rewire
        StsElement* oldTail = header->tail;
        oldTail->next = element;
        header->tail = element;
    }
    pthread_mutex_unlock(header->mutex);
}

static void* pop(StsHeader *header);
static void* pop(StsHeader *header) {
    pthread_mutex_lock(header->mutex);
    StsElement *head = header->head;

    // Is empty?
    if (head == NULL) {
        pthread_mutex_unlock(header->mutex);
        return NULL;
    } else {
        // Rewire
        header->head = head->next;

        // Get head and free element memory
        void *value = head->value;
        free(head);

        pthread_mutex_unlock(header->mutex);
        return value;
    }
}

static int empty(StsHeader *header);
static int empty(StsHeader *header)
{
    return header->head == NULL;
}

_StsQueue const StsQueue = {
        create,
        destroy,
        push,
        pop,
        empty
};
```

Приложение 6 (model.py)

```
import networkx.algorithms.shortest_paths as shortest_paths
import random
import math
import copy

PACKET_WEIGHT = 10


class Model:
    def __init__(self, node_count, block_count, time, graph, assigner, tasks, nodes_for_block_create):
        self.graph = graph
        self.time = time
        self.node_count = node_count
        self.block_count = block_count
        self.assigner = assigner
        self.tasks = tasks
        self.nodes_for_block_create = nodes_for_block_create
        self.paths = {}
        for node in range(self.node_count):
            self.paths[node] = {}
            for path_target in range(self.node_count):
                if node == path_target:
                    continue
                if shortest_paths.has_path(self.graph, node, path_target):
                    path = shortest_paths.unweighted.bidirectional_shortest_path(self.graph, node,
path_target)
                    if len(path) > 1:
                        self.paths[node][path_target] = path[1]

    def calculate(self):
        stats = Statistics()
        for node in range(self.node_count):
            stats.max[node] = stats.average[node] = 0

        for t in range(self.time):
            task = self.tasks[t]
            hosts = self.get_hosts(task.block_number)
            packets = self.assigner.assign(task, hosts)
            for packet in packets:
                if packet.task.target in self.paths[packet.sender]:
                    self.add_packet(packet.sender, packet)

            self.create_new_block(t)

            nodes = list(range(self.node_count))
```

```
            random.shuffle(nodes)
            for node in nodes:
                for link_node in self.graph.node[node]["packets"]:
                    queue = self.graph.node[node]["packets"][link_node]

                    if not queue:
                        continue

                    packet = queue.pop()
                    next_node = self.paths[node][packet.task.target]
                    if next_node != packet.task.target:
                        self.add_packet(next_node, packet)
                    else:
                        self.graph.node[next_node]["data"].add(packet.task.block_number)

                self.calculate_statistics(stats, node)
        return stats

    def get_hosts(self, block_number):
        hosts = []
        for node in range(self.node_count):
            if block_number in self.graph.node[node]["data"]:
                hosts.append(node)
        return hosts

    def add_packet(self, sender, packet):
        next_node = self.paths[sender][packet.task.target]
        self.graph.node[sender]["packets"][next_node].appendleft(packet)

    def create_new_block(self, t):
        new_block_number = self.block_count
        self.block_count += 1
        host = self.nodes_for_block_create[t]
        self.graph.node[host]["data"].add(new_block_number)

    def calculate_statistics(self, stats, node):
        current_packets = [p for packets in self.graph.node[node]["packets"].values() for p in packets]
        count = len(current_packets)
        stats.average[node] += count / self.time
        stats.max[node] = stats.max[node] if stats.max[node] > count else count
        # stats.max_max = stats.max_max if stats.max_max > stats.max[node] else stats.max[node]
        # stats.max_average = stats.max_average if stats.max_average > stats.average[node] else
stats.average[node]


class P2PAssigner:
    @staticmethod
```

```
    def assign(task, hosts):
        result = []
        if len(hosts) == 0:
            return result

        rnd = random.Random()
        i = rnd.randint(0, len(hosts) - 1)
        sender = hosts[i]
        for _ in range(PACKET_WEIGHT):
            result.append(Packet(sender, task))
        return result


class M2PAssigner:
    def __init__(self, duplication_degree):
        self.duplication_degree = duplication_degree

    def assign(self, task, hosts):
        result = []
        if len(hosts) == 0:
            return result
        elif len(hosts) <= self.duplication_degree:
            senders = hosts
        else:
            senders = copy.copy(hosts)
            random.shuffle(senders)
            senders = senders[:self.duplication_degree]

        for sender in senders:
            for _ in range(math.ceil(PACKET_WEIGHT / len(senders))):
                result.append(Packet(sender, task))
        return result


class Packet:
    def __init__(self, sender, task):
        self.sender = sender
        self.task = task


class Task:
    def __init__(self, block_number, target):
        self.block_number = block_number
        self.target = target


class Statistics:
```

```
def __init__(self):
    self.average = {}
    self.max = {}
    # self.max_max = 0
    # self.max_average = 0
```

Приложение 7 (tests.py)

```
import unittest
import socket
import subprocess
import time


class Client:
    def __init__(self, addr, port):
        self.client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.client_socket.settimeout(5.0)
        self.addr = str.encode(addr)
        self.port = port

    def start(self):
        subprocess.Popen(['/bin/bash', '-c', "cmake-build-debug/untitled -p %s" % self.port])

    def send(self, target_address, target_port, block_number, offset, length, data):
        send_data = bytes([0x30])

        send_data += bytes(target_address)
        send_data += bytes([target_port & 0xFF, target_port >> 8 & 0xFF])
        send_data += bytes([block_number & 0xFF, block_number >> 8 & 0xFF])
        send_data += bytes([offset & 0xFF, offset >> 8 & 0xFF, offset >> 16 & 0xFF,  offset >> 24 &
0xFF])
        send_data += bytes([length & 0xFF, length >> 8 & 0xFF])
        send_data += bytes(data)

        self.__send_data(send_data)

    def put(self, block_number, offset, length, data):
        put_data = bytes([0x20])
        put_data += bytes([block_number & 0xFF, block_number >> 8 & 0xFF])
        put_data += bytes([offset & 0xFF, offset >> 8 & 0xFF, offset >> 16 & 0xFF,  offset >> 24 &
0xFF])
        put_data += bytes([length & 0xFF, length >> 8 & 0xFF])
        put_data += bytes(data)

        self.__send_data(put_data)

    def receive(self, block_number, offset):
        receive_data = bytes([0x50])
        receive_data += bytes([block_number & 0xFF, block_number >> 8 & 0xFF])
        receive_data += bytes([offset & 0xFF, offset >> 8 & 0xFF, offset >> 16 & 0xFF,  offset >> 24
& 0xFF])
```

```
        self.__send_data(receive_data)

        return self.client_socket.recvfrom(65207)[0]

    def quit(self):
        self.__send_data(b"\0")

    def dispose(self):
        self.client_socket.close()

    def __send_data(self, data):
        self.client_socket.sendto(data, (self.addr, self.port))


class ServerTests(unittest.TestCase):
    def setUp(self):
        self.client1 = Client("127.0.0.1", 1111)
        self.client2 = Client("127.0.0.1", 2222)
        self.client1.start()
        self.client2.start()

    def test_put_and_receive(self):
        block_number = 7
        offset = 4
        data = [1, 2, 3]
        data_bytes = b"\x01\x02\x03"
        length = len(data)

        self.client1.put(block_number, offset, length, data)
        result = self.client1.receive(block_number, offset)[:length]

        self.assertEqual(data_bytes, result)

    def test_send_and_receive(self):
        block_number = 0
        offset = 0
        data = [1, 2, 3]
        data_bytes = b"\x01\x02\x03"
        length = len(data)
        self.client1.put(block_number, offset, length, data)

        target_address = b"\x01\x00\x00\x7f"
        target_port = self.client2.port
        self.client1.send(target_address, target_port, block_number, offset, length, data)

        result = self.client2.receive(block_number, offset)[:length]
        self.assertEqual(data_bytes, result)
```

```
    def tearDown(self):
        time.sleep(1)
        self.client1.quit()
        self.client1.dispose()
        self.client2.quit()
        self.client2.dispose()
        time.sleep(1)


if __name__ == "__main__":
    unittest.main()
```