# Содержание

Введение ………………………………………………………………………. 2

Постановка задачи работы ...………………………………………………. 3

Команды протокола …………………………………………………………. 4

Способы и методы решения задачи ……………………………………... 7

Таймеры и ошибки ………………………………………………………... 9

Потокобезопасность ………………………………………………………. 10

Структуры данных ………………………………………………………… 11

Модель сети …………………………………………………………………… 12

Заключение …………………………………………………………………….. 13

## Введение

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

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 отдельной ноде, то теоретически можно ускорить расчёты пропорционально числу вовлечённых нод. 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 (ноды) 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.

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

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.

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

2 байта – смещение в блоке данных. Первый байт – младший.

2 байта – длина данных. Первый байт – младший.

[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 данных:

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

2 байта – порт цели. Первый байт – младший.

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

2 байта – смещение в блоке данных. Первый байт – младший.

2 байта – длина данных. Первый байт – младший.

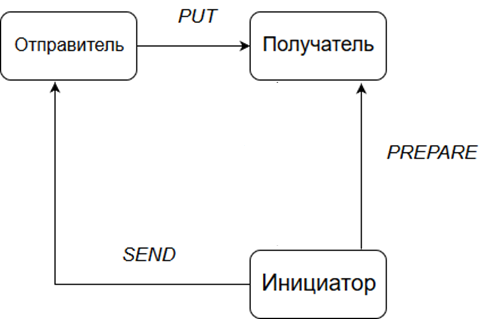
Код команды – 0x30.

схема работы протокола

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.

## *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.

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.

2 байта – номер блока. Первый байт – младший.

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. Серверная часть и consumer работают в отдельных потоках. Для работы с потоками используется стандартная библиотека *pthread*.

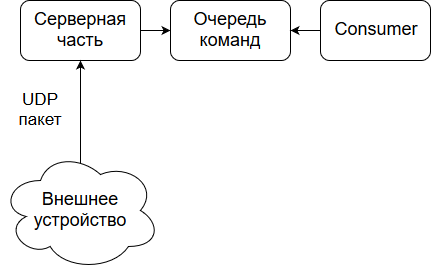


схема взаимодействия серверной части и consumer'а

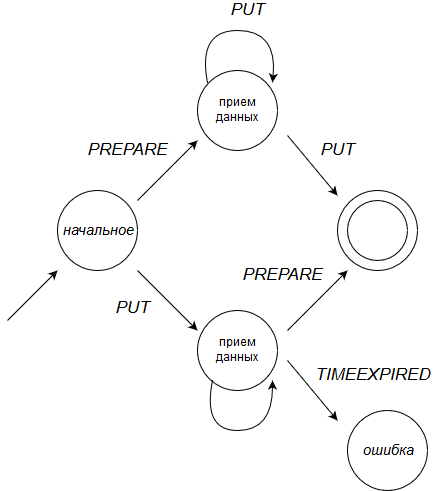
В процессе работы используются следующие глобальные переменные:

* *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*. Контекст хранит в себе номер блока, объем принятых данных, адрес инициатора, таймеры и другую информацию.

Схема работы consumer’а.

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.

В цикле consumer’a 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».

конечный автомат, представляющий работу consumer'а

Схема работы серверной части следующая:

1. *c =* верхний элемент в *BinaryHeap*.

Если в *BinaryHeap* пуста, *c* будет равно *NULL*.

1. Устанавливается время ожидания *t*. Если контекст *c* существует, то *t* равно значению времени истекания таймера контекста *c*. Иначе *t* = 10000. В итоге, в переменной *t* хранится время ожидания в миллисекундах.
2. 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)

- Смещение в блоке данных + длина 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 очередью контекстов. Для обеспечения 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 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.

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.

# Модель сети

## Описание модели

Для определения оптимальности выбранной стратегии была создана модель сети.

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

* Выполняется очередная задача. Для этого определяются ноды-отправители, на которых лежит нужный блок данных; из этих нод-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 нодами, имеет свою очередь (FIFO) пакетов. За один такт передается один пакет. Пакеты передаются по кратчайшему пути – пути, состоящем из наименьшего количества ребер. Во время инициализации модели ищутся кратчайшие пути между всем вершинами, на основе чего составляется список:

ДВ i: ДВ j ищем path, dict[i][j] = path[1]

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*. 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*.

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

# C:\Users\penguin\Downloads\m2p.pngC:\Users\penguin\Downloads\p2p.png

## Результаты моделирования

В ходе моделирования было установлено, что следующие параметры не влияли на процесс: количество начальных блоков данных, количество тактов.

Степень распределенности системы ускоряет работу m2p модели, однако после 5 увеличивать не умеет смысла, ибо производительность не меняется.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.

Метод “*tearDown*” будет выполняться по завершению каждого теста.

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:

* *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 элементами системы (нодами).

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:

- Б-дерево (вставка, удаление, поиск)

- Потокобезопасная очередь (вставка, взятие)

- Двоичная куча (вставка, удаление, взятие, взятие с удалением, перестраивание)