**Основные отличия нового фазового волнового алгоритма от предыдущего:**

1. **Децентрализованность:**

* В новом алгоритме каждый узел принимает самостоятельные решения
* Узлы обмениваются информацией через систему сообщений
* Нет централизованного управления балансировкой

1. **Система сообщений:**

* Два типа сообщений:
* wave - для распространения информации о нагрузке
* transfer - для переноса нагрузки
* Каждый узел имеет очередь сообщений
* Узлы обрабатывают сообщения асинхронно

1. **Фазовый подход:**

* Процесс балансировки разделен на фазы
* Каждая фаза волны передается по сети
* Узлы обрабатывают волну и решают о переносе нагрузки

1. **Логика балансировки:**

*В предыдущем алгоритме:*

* Прямой перенос нагрузки между максимальным и минимальным узлами
* Проверка порога в 20%

*В новом алгоритме:*

* Постепенное перераспределение нагрузки через соседей
* Каждый узел решает о переносе нагрузки самостоятельно
* Учет максимальной нагрузки при переносе

1. **Процесс балансировки:**

*В предыдущем алгоритме:*

* Нахождение максимального и минимального узлов
* Проверка порога
* Прямой перенос нагрузки

*В новом алгоритме:*

* Инициация волны от узла с максимальной нагрузкой
* Распространение волны по сети
* Обработка волновых сообщений
* Расчет и перенос нагрузки
* Обновление состояния узлов

Новый алгоритм более масштабируемый и устойчивый к отказам, так как не зависит от централизованного управления и учитывает реальную топологию сети через систему соседей.

import asyncio

import os

import json

import logging

from aiogram import Bot, Dispatcher, types

from aiogram.types import Message, BotCommand

from dotenv import load\_dotenv

from datetime import datetime

load\_dotenv()

TOKEN = os.getenv("TELEGRAM\_BOT\_TOKEN")

bot = Bot(*token*=TOKEN)

dp = Dispatcher()

logging.basicConfig(*level*=logging.INFO, *format*='%(asctime)s - %(levelname)s - %(message)s')

NODE\_LOADS\_FILE = "node\_loads.json"

*class* Node:

*def* \_\_init\_\_(*self*, *id*, *max\_load*):

*self*.id = *id*

*self*.max\_load = *max\_load*

*self*.load = 0

*self*.neighbors = []  # Соседние узлы

*self*.visited = False

*self*.message\_queue = []

*def* update\_load(*self*, *amount*=1):

        """Увеличивает нагрузку"""

*self*.load = min(*self*.load + *amount*, *self*.max\_load)

*def* decrease\_load(*self*, *amount*=1):

        """Уменьшает нагрузку"""

*self*.load = max(*self*.load - *amount*, 0)

*def* get\_status(*self*):

        """Получает текущее состояние узла"""

        return {

            'id': *self*.id,

            'load': *self*.load,

            'max\_load': *self*.max\_load,

            'neighbors': [n.id for n in *self*.neighbors]

        }

*def* process\_message(*self*, *message*):

        """Обрабатывает входящее сообщение"""

        if *message*['type'] == 'wave':

*self*.handle\_wave(*message*)

        elif *message*['type'] == 'transfer':

*self*.handle\_transfer(*message*)

*def* handle\_wave(*self*, *message*):

        """Обрабатывает волновое сообщение"""

        if not *self*.visited:

*self*.visited = True

*self*.message\_queue.extend(*self*.create\_wave\_messages())

*self*.message\_queue.extend(*self*.create\_transfer\_messages())

*def* handle\_transfer(*self*, *message*):

        """Обрабатывает сообщение о переносе нагрузки"""

        if *message*['source'] != *self*.id:

            transfer\_amount = *message*['amount']

*self*.decrease\_load(transfer\_amount)

*self*.message\_queue.extend(*self*.create\_transfer\_messages())

*def* create\_wave\_messages(*self*):

        """Создает волновые сообщения для соседей"""

        return [{

            'type': 'wave',

            'source': *self*.id,

            'status': *self*.get\_status()

        } for neighbor in *self*.neighbors]

*def* create\_transfer\_messages(*self*):

        """Создает сообщения о переносе нагрузки"""

        messages = []

        for neighbor in *self*.neighbors:

            transfer\_amount = *self*.calculate\_transfer\_amount(neighbor)

            if transfer\_amount > 0:

                messages.append({

                    'type': 'transfer',

                    'source': *self*.id,

                    'target': neighbor.id,

                    'amount': transfer\_amount

                })

        return messages

*def* calculate\_transfer\_amount(*self*, *neighbor*):

        """Вычисляет количество нагрузки для переноса"""

        max\_transfer = (*self*.load - *neighbor*.load) // 2

        return min(max\_transfer, *self*.max\_load // 4)

*async* *def* propagate\_wave(*node*):

    """Распространяет волну по сети"""

    try:

        logging.info(*f*"Начало волны от узла {*node*.id}")

*node*.visited = True

        messages = *node*.create\_wave\_messages()

        for message in messages:

            await process\_message(*node*, message)

        while any(node.message\_queue for node in NODES):

            for *node* in NODES:

                while *node*.message\_queue:

                    message = *node*.message\_queue.pop(0)

                    await process\_message(*node*, message)

    except *Exception* as e:

        logging.error(*f*"Ошибка при распространении волны: {e}")

*async* *def* process\_message(*node*, *message*):

    """Обрабатывает сообщение в асинхронном режиме"""

    try:

        target\_node = next(n for n in NODES if n.id == *message*['target'])

        target\_node.process\_message(*message*)

    except *Exception* as e:

        logging.error(*f*"Ошибка при обработке сообщения: {e}")

*async* *def* balance\_load():

    """Запускает фазовый волновой алгоритм балансировки"""

    try:

        # Выбираем узел с максимальной нагрузкой как источник волны

        source\_node = max(NODES, key=*lambda* *node*: *node*.load)

        logging.info(*f*"Запуск волнового алгоритма от узла {source\_node.id}")

        # Сбрасываем флаги посещения

        for node in NODES:

            node.visited = False

        # Запускаем волну

        await propagate\_wave(source\_node)

        # Сохраняем состояние нагрузки

        save\_node\_loads()

    except *Exception* as e:

        logging.error(*f*"Ошибка при балансировке нагрузки: {e}")

*async* *def* periodic\_balancing():

    """Периодическая балансировка нагрузки"""

    while True:

        try:

            await balance\_load()

            await asyncio.sleep(60)  # Проверяем каждую минуту

        except *Exception* as e:

            logging.error(*f*"Ошибка при периодической балансировке: {e}")

@dp.message(Command("node\_status"))

*async* *def* node\_status(*message*: types.Message):

    """Обработчик команды /node\_status"""

    try:

        status\_text = "📊 Статус узлов:\n\n"

        for node in NODES:

            status = node.get\_status()

            status\_text += (

*f*"Узел {status['id']}:\n"

*f*"Загрузка: {status['load']}%\n"

*f*"Максимальная загрузка: {status['max\_load']}%\n"

*f*"Соседи: {', '.join(*map*(*str*, status['neighbors']))}\n\n"

            )

        await *message*.answer(status\_text)

    except *Exception* as e:

        logging.error(*f*"Ошибка при получении статуса узлов: {e}")

        await *message*.answer("❌ Произошла ошибка при получении статуса узлов")

*def* save\_node\_loads():

    """Сохраняет текущее состояние нагрузки узлов"""

    try:

        with open(NODE\_LOADS\_FILE, 'w') as f:

            json.dump({node.id: node.load for node in NODES}, f)

    except *Exception* as e:

        logging.error(*f*"Ошибка при сохранении нагрузки: {e}")

*def* load\_node\_loads():

    """Загружает состояние узлов из файла"""

    try:

        with open(NODE\_LOADS\_FILE, 'r') as f:

            loads = json.load(f)

            for node in NODES:

                node.load = loads.get(*str*(node.id), 0)

    except *Exception* as e:

        logging.error(*f*"Ошибка при загрузке нагрузки: {e}")

# Инициализация сети

NODES = [

    Node(0, 100),  # Узел 0 с максимальной нагрузкой 100%

    Node(1, 100),  # Узел 1 с максимальной нагрузкой 100%

    Node(2, 100)   # Узел 2 с максимальной нагрузкой 100%

]

# Устанавливаем топологию сети

NODES[0].neighbors = [NODES[1], NODES[2]]

NODES[1].neighbors = [NODES[0], NODES[2]]

NODES[2].neighbors = [NODES[0], NODES[1]]

*async* *def* main():

    # Загружаем состояние узлов

    load\_node\_loads()

    # Запускаем периодическую балансировку в отдельном таске

    asyncio.create\_task(periodic\_balancing())

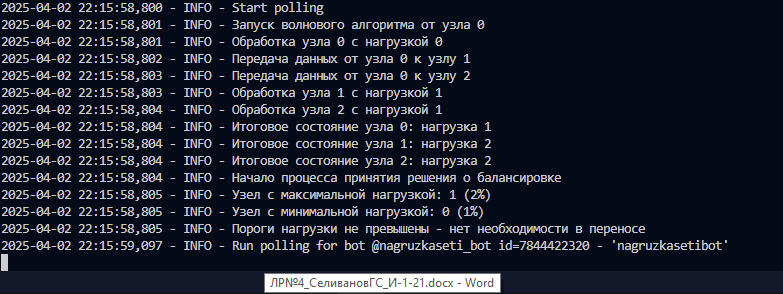
    # Запускаем бота

    await dp.start\_polling(bot)

if \_\_name\_\_ == "\_\_main\_\_":

    asyncio.run(main())

Вывод (логи) в консоли:



Вывод (логи) в консоли через 1 минуту:

