Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н. Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

ОТЧЕТ

по лабораторной работе № 5

по курсу «Моделирование»

на тему: «Моделирование работы информационного центра»

Студент ИУ7-71Б _____ Мицевич М. Д.
(Группа) (Подпись, дата) (И. О. Фамилия)

Преподаватель _____ Рудаков И. В.
(Подпись, дата) (И. О. Фамилия)

2022 г.

1 Теоретический раздел

Переменные и уравнения имитационной модели

Эндогенные переменные: время обработки задания i-ым оператором, время решения этого задания j-ым компьютером.

Экзогенные переменные: число обслуженных клиентов и число клиентов, получивших отказ.

Структурная схема представлена на рисунке 1.1, на нем К1-К3 моделируют работу операторов, К4-К5 – компьютеров, Г – генераторы времени поступления заявок и их обработки операторами.

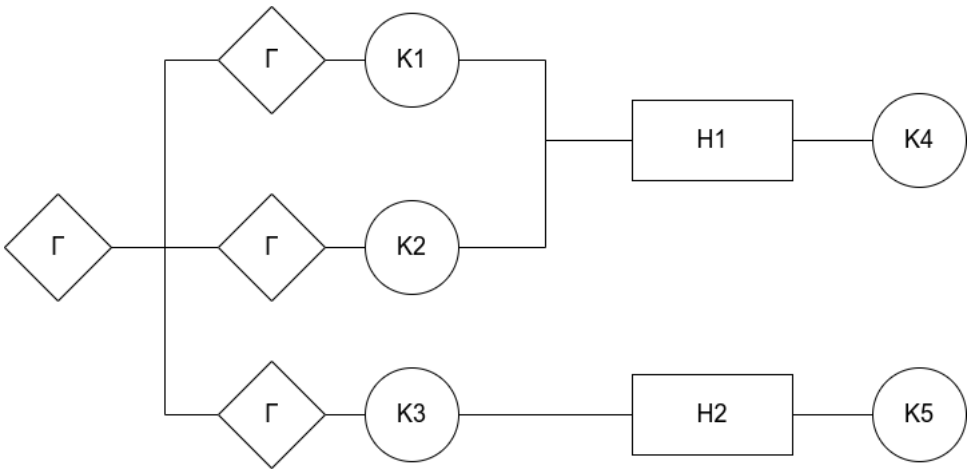

Рисунок 1.1 – Структурная схема модели информационного центра

2 Практическая часть

Результат работы программы представлен на рисунке 2.1.

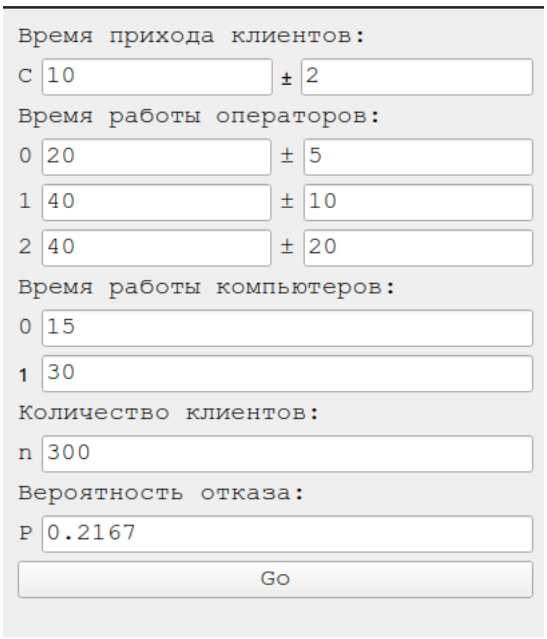

Рисунок 2.1 – Результат работы программы

На листинге 2.1 представлен код моделирования работы информационного центра.

Листинг 2.1 – Моделирование работы информационного центра

```
import numpy.random as nr


class UniformGenerator:
    def __init__(self, m, d):
        self._a = m - d
        self._b = m + d
        if not 0 <= self._a <= self._b:
            raise ValueError('Параметры␣должны␣удовлетворять␣
               условию␣0␣<=␣a␣<=␣b')

    def next(self):
        return nr.uniform(self._a, self._b)


class ConstGenerator:
    def __init__(self, m):
        if m <= 0:
```

```
            raise ValueError('Параметр должен быть больше 0')
        self._m = m

    def next(self):
        return self._m


class RequestGenerator:
    def __init__(self, generator):
        self._generator = generator
        self._receivers = []
        self._generated_requests = 0
        self._next_event_time = 0

    @property
    def next_event_time(self):
        return self._next_event_time

    @next_event_time.setter
    def next_event_time(self, time):
        self._next_event_time = time

    @property
    def generated_requests(self):
        return self._generated_requests

    def add_receiver(self, receiver):
        if receiver not in self._receivers:
            self._receivers.append(receiver)

    def remove_receiver(self, receiver):
        try:
            self._receivers.remove(receiver)
        except KeyError:
            pass

    def generate_time(self):
        return self._generator.next()

    def emit_request(self):
        self._generated_requests += 1
```

```
        for receiver in self._receivers:
            if receiver.receive_request():
                return receiver
        else:
            return None


class RequestProcessor(RequestGenerator):
    def __init__(self, generator, *, max_queue_size=0,
       reenter_probability=0.0):
        super().__init__(generator)
        self._generator = generator
        self._queued_requests = 0
        """Current count of requests in queue of processor"""
        self._max_queue_size = max_queue_size
        """Max queue size of processor"""
        self._queue_size = max_queue_size
        """Holds total queue size of processor"""
        self._processed_requests = 0
        self._reenter_probability = reenter_probability
        self._reentered_requests = 0

    @property
    def processed_requests(self):
        return self._processed_requests

    @property
    def queue_size(self):
        """
        Returns total queue size
        """
        return self._queue_size

    @queue_size.setter
    def queue_size(self, size):
        self._queue_size = size

    @property
    def queued_requests(self):
        """
        Returns number of queued requests
```

```
        """
        return self._queued_requests

    @property
    def reentered_requests(self):
        return self._reentered_requests

    def process(self):
        if self._queued_requests > 0:
            self._processed_requests += 1
            self._queued_requests -= 1
            self.emit_request()
            if nr.random_sample() < self._reenter_probability:
                self._reentered_requests += 1
                self.receive_request()

    def receive_request(self):
        if self._max_queue_size == 0:
            if self._queued_requests >= self._queue_size:
                self._queue_size += 1
            self._queued_requests += 1
            return True
        elif self._queued_requests < self._queue_size:
            self._queued_requests += 1
            return True
        return False


def event_based_modelling(client_m, client_d,
                          op0_m, op0_d, op1_m, op1_d, op2_m,
                             op2_d,
                          comp0_m, comp1_m, c_count):
    client_gen = RequestGenerator(UniformGenerator(client_m,
       client_d))
    op0 = RequestProcessor(UniformGenerator(op0_m, op0_d),
       max_queue_size=1)
    op1 = RequestProcessor(UniformGenerator(op1_m, op1_d),
       max_queue_size=1)
    op2 = RequestProcessor(UniformGenerator(op2_m, op2_d),
       max_queue_size=1)
    comp0 = RequestProcessor(ConstGenerator(comp0_m))
```

```
comp1 = RequestProcessor(ConstGenerator(comp1_m))

client_gen.add_receiver(op0)
client_gen.add_receiver(op1)
client_gen.add_receiver(op2)
op0.add_receiver(comp0)
op1.add_receiver(comp0)
op2.add_receiver(comp1)

devices = [client_gen, op0, op1, op2, comp0, comp1]
for device in devices:
    device.next_event_time = 0

dropped_requests = 0
client_gen.next_event_time = client_gen.generate_time()
op0.next_event_time = op0.generate_time()
while client_gen.generated_requests < c_count:
    # Find next event
    current_time = client_gen.next_event_time
    for device in devices:
        if 0 < device.next_event_time < current_time:
            current_time = device.next_event_time

    for device in devices:
        if current_time == device.next_event_time:
            if not isinstance(device, RequestProcessor):
                assigned_processor = client_gen.emit_request
                   ()
                if assigned_processor is not None:
                    assigned_processor.next_event_time =
                            (current_time +
                            assigned_processor.generate_time
                               ())
                else:
                    dropped_requests += 1
                client_gen.next_event_time = current_time +
                   client_gen.generate_time()
            else:
                device.process()
                if device.queued_requests == 0:
                    device.next_event_time = 0
```

```
                else:
                    device.next_event_time = current_time +
                        device.generate_time()

    return dropped_requests / c_count
```