|  |  |  |
| --- | --- | --- |
| **КУРСОВОЙ ПРОЕКТ** | | |
| МЕТОДЫ ИМИТАЦИОННОГО МОДЕЛИРОВАНИЯ | | |
| по дисциплине «Архитектура программных систем» | | |
| Выполнил | | |
| студент гр. 5130904/20104 |  | Шелковников Д.С. |
| Руководитель | | |
|  |  | Гончаров А.В. |
| «\_\_\_\_» декабря 2024г. | | |

**Оглавление**

[Введение 3](#_Toc185607703)

[Исходные данные 4](#_Toc185607704)

[Маппинг 5](#_Toc185607705)

[Архитектура приложения 6](#_Toc185607706)

[Диаграмма классов 6](#_Toc185607707)

[Сиквенс-диаграммы 7](#_Toc185607708)

[BPMN диаграмма 8](#_Toc185607709)

[Ограничения и требуемые характеристики 9](#_Toc185607710)

[Модульная структура 10](#_Toc185607711)

[Описание работы программы 12](#_Toc185607712)

[Результаты работы 14](#_Toc185607713)

[Анализ результатов: 14](#_Toc185607714)

[Стоимость компонентов системы 14](#_Toc185607715)

[Вывод 15](#_Toc185607716)

# Введение

Целью практической курсовой является создание модели ВС или ее компонентов на некотором уровне детализации, описывающей и имитирующей ее структуру и функциональность.

Каждый реальный объект ВС обладает огромной сложностью, определяемой множеством состояний, множеством внутренних и внешних связей, множеством анализируемых характеристик. Модель дает приближенное описание объекта с целью получения требуемых результатов с определенной точностью и достоверностью. Степень приближения модели к описываемому объекту может быть различной и зависит от требований задачи.

Существуют различные типы моделей ВС: аналитические, аналоговые, физические и имитационные. В данной работе будет использоваться имитационная модель ВС. Одним из подходов к построению имитационной модели является построение ее в виде системы массового обслуживания (СМО).

# Исходные данные

16. ИБ ИЗ2 ПЗ1 Д10З1 Д10О3 Д2П2 Д2Б2 ОР1 ОД1

**ИСТОЧНИКИ**

ИБ — источник бесконечный

ИЗ1 — равномерный закон генерации заявок

**ПРИБОРЫ**

ПЗ2 — экспоненциальный закон обслуживания заявок

**ДИСЦИПЛИНЫ БУФЕРИЗАЦИИ**Д10З1 — заполнение буфера по кольцу

**ДИСЦИПЛИНЫ ОТКАЗА**Д10О3 — самая старая заявка в буфере получает отказ в случае переполнения

**ДИСЦИПЛИНЫ ВЫБОРА ПРИБОРА**

Д2П2 — выбор прибора для обслуживания по кольцу

**ДИСЦИПЛИНА ВЫБОРА ЗАЯВКИ НА ОБСЛУЖИВАНИЕ**  
Д2Б2 — выбор заявки на обслуживание по принципу LIFO (последний пришел — первый обслужен)

**ДИНАМИЧЕСКОЕ ОТРАЖЕНИЕ РЕЗУЛЬТАТОВ**  
ОД2 — формализованная схема модели, текущее состояние;

**ОТОБРАЖЕНИЕ РЕЗУЛЬТАТОВ**ОР1 — сводная таблица результатов;

# Маппинг

Система обработки заявок в автоматизированной системе хранения и выдачи товаров на складе с роботизированными устройствами для перемещения грузов.

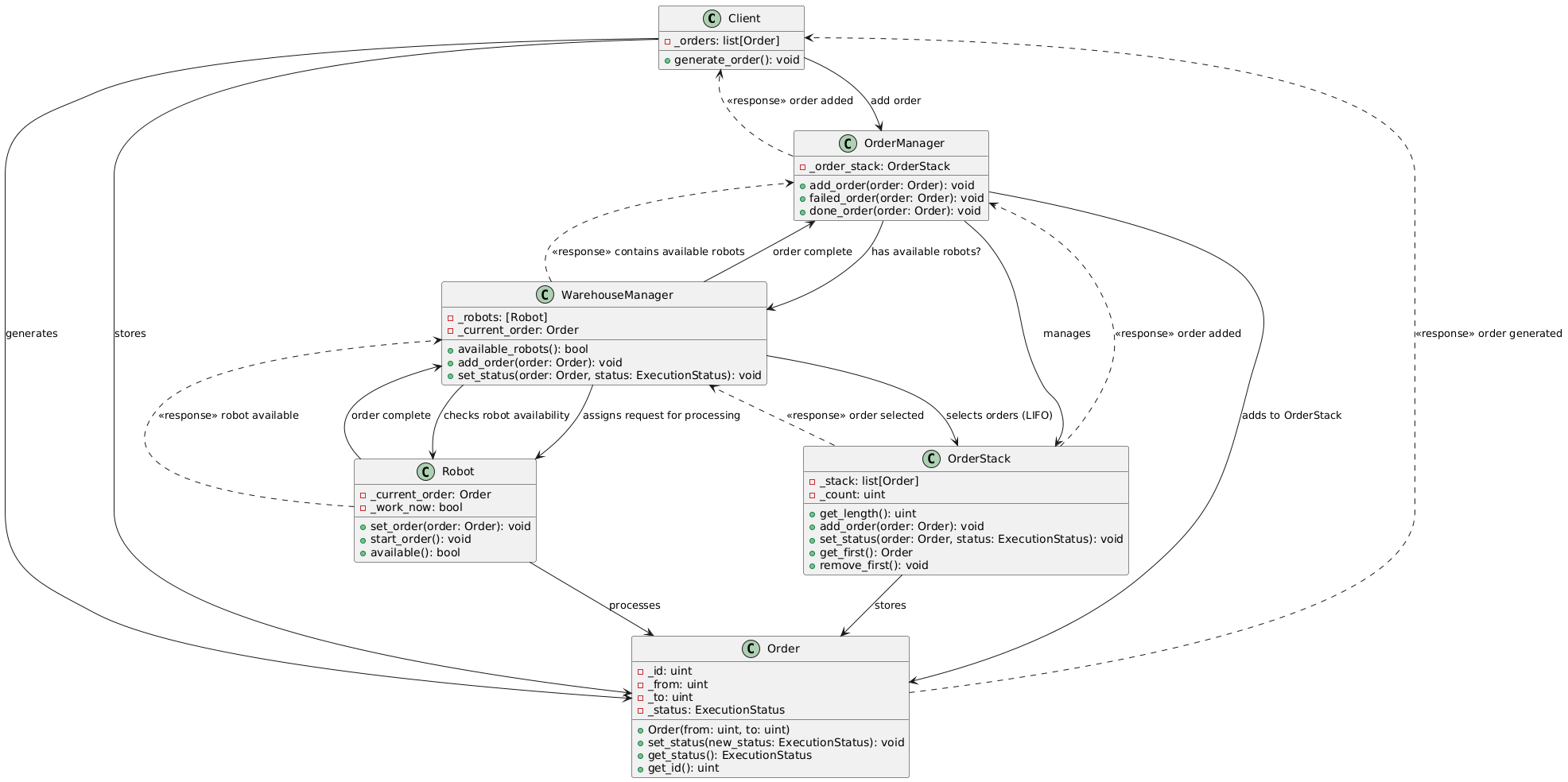
Задача заключается в эффективном распределении заявок между исполнителями.

В случае высокой загрузки и большого количества заявок могут происходить отказы в обслуживании.

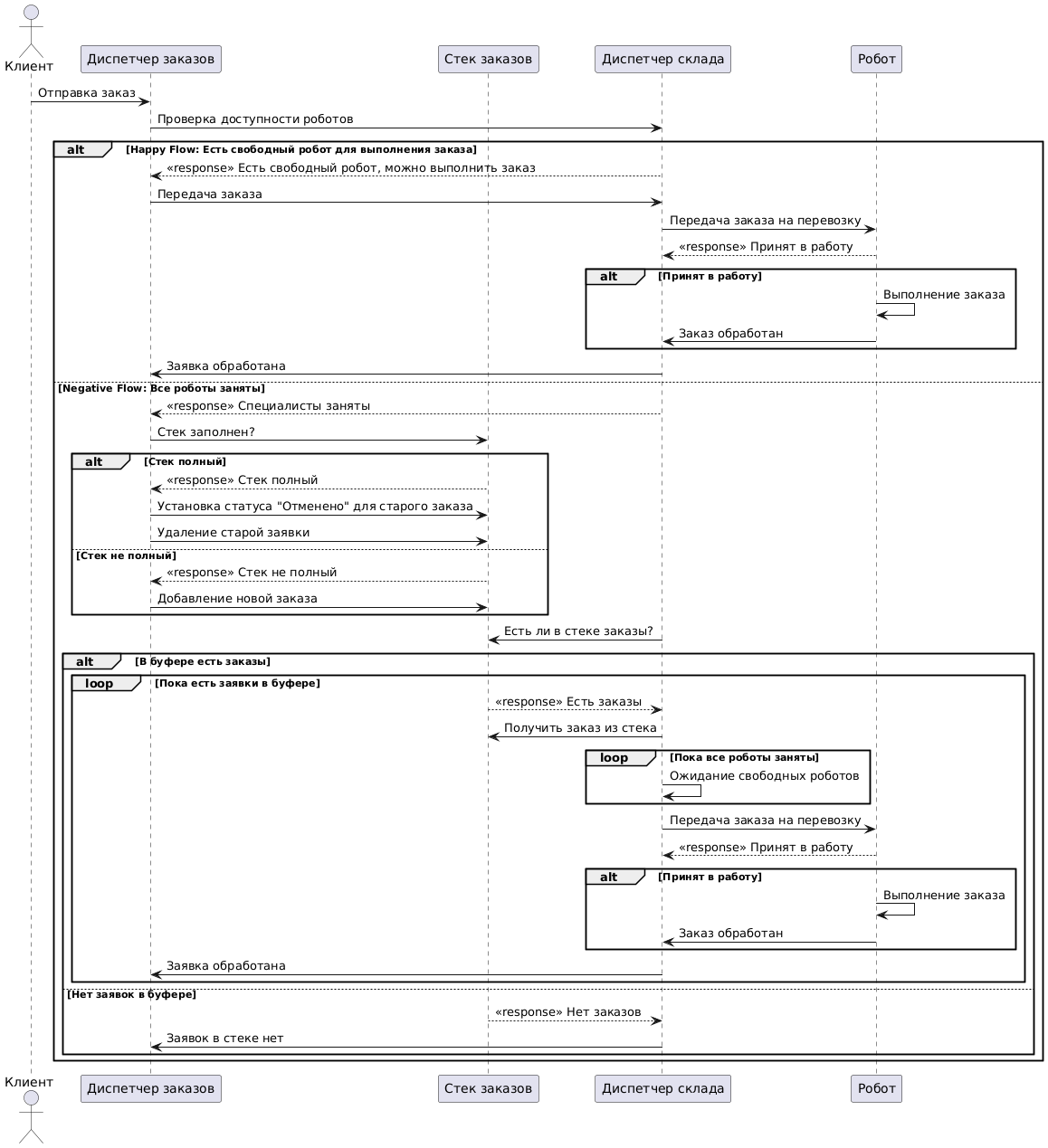
1. Источник — (заказчик) заказчик, создающий запрос на перемещение товара на складе.
2. Заявка — запрос на перемещение товара из одной зоны склада в другую.
3. Диспетчер постановки — (диспетчер заказов) система управления очередью, которая заполняет очередь. В случае переполнения очереди удаляет самую старую заявку, выдавая ей статус «Отменено».
4. Буфер — (стек заказов) виртуальная очередь заявок на перемещение товаров (LIFO — последняя заявка обрабатывается первой).
5. Диспетчер выборки — (диспетчер склада) система, выбирающая заявки для роботов (по кольцу), чтобы распределять нагрузку равномерно между ними.
6. Прибор — (робот) роботизированный манипулятор, перемещающий товар (время обработки заявки распределено по экспоненциальному закону).

# Архитектура приложения

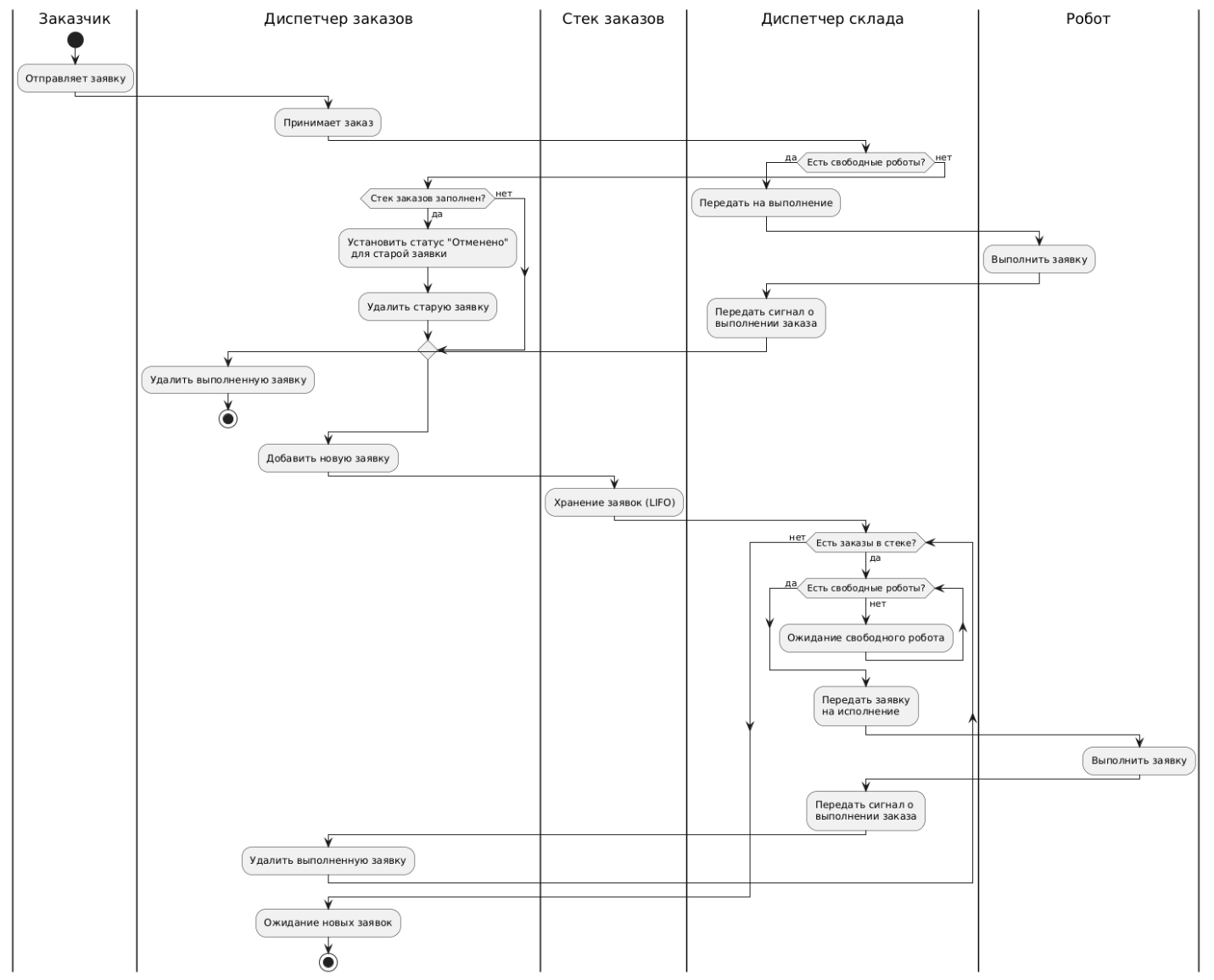
### Диаграмма классов



### Сиквенс-диаграммы



### BPMN диаграмма



# Ограничения и требуемые характеристики

* Вероятность отказа должна составлять не более 10%.
* Загрузка приборов более 90%.

Время пребывания заявки в системе не ограничено, т.к. в зависимости от присланных данных, заявка может обрабатываться длительное время для получения верного результата.

# Модульная структура

Разработка производилась в языке С++ с использованием графической библиотеки Qt6 для визуализации. Приложение является объектно-ориентированным и содержит следующие классы:

1. Класс **Client**  
   **Описание:** Класс, описывающий клиента, создающего заказы  
   **Приватные поля класса:**
   * **\_orders: list[Order]** - массив текущих заказов клиента

**Публичные методы:**

* + **generate\_order(): void** - метод, создающий рандомный заказ на перемещение

1. Класс **Order**  
   **Описание:** Класс, описывающий заказ на перемещение  
   **Приватные поля класса:**
   * **\_id: uint** - уникальный идентификатор заказа
   * **\_from: uint** - идентификатор места, откуда нужно переместить
   * **\_to: uint** - идентификатор места, куда нужно переместить
   * **\_status: ExecutionStatus** - текущий статус заказа

**Публичные методы:**

* + **Order(from: uint, to: uint)** - конструктор класса с двумя полями: откуда и куда положить
  + **set\_status(new\_status: ExecutionStatus): void** - установить новый статус для заказа
  + **get\_status(): ExecutionStatus** - получить текущий статус
  + **get\_id(): uint** - получить идентификатор заказа

1. Класс **OrderManager**  
   **Описание:** Диспетчер заказов, управляет очередью  
   **Приватные поля класса:**
   * **\_order\_stack: OrderStack** - стек, в котором хранятся заказы

**Публичные методы:**

* + **add\_order(order: Order): void** - добавляет заказ в очередь
  + **failed\_order(order: Order): void** - устанавливает статус **"Не выполнен"** для заказа и удаляет его из очереди
  + **done\_order(order: Order): void** - устанавливает статус **"Выполнен"** для заказа и удаляет его из очереди

1. Класс **OrderStack**  
   **Описание:** Стек в котором хранятся все заказы  
   **Приватные поля класса:**
   * **\_stack: list[Order]** - стек, в которой лежат все заказы
   * **\_count: uint** - размер стека

**Публичные методы:**

* + **get\_length(): uint** - возвращает длину стека
  + **add\_order(order: Order): void** - добавляет заказ в начало стека
  + **set\_status(order: Order, status: ExecutionStatus): void** - устанавливает статус для заказа
  + **get\_first(): Order** - возвращает последний добавленный заказ
  + **remove\_first(): void** - удаляет последний добавленный заказ

1. Класс **WarehouseManager**  
   **Описание:** Менеджер склада, управляет роботами и назначает им заказы  
   **Приватные поля класса:**
   * **\_robots: [Robot]** - список роботов
   * **\_current\_order: Order** - текущий заказ, который требуется обработать

**Публичные методы:**

* + **available\_robots(): bool** - проверка доступных роботов
  + **add\_order(order: Order): void** - добавить заказ
  + **set\_status(order: Order, status: ExecutionStatus): void** - установить статус для заказа

1. Класс **Robot**  
   **Описание:** Робот, выполняющий заказ  
   **Приватные поля класса:**
   * **\_current\_order: Order** - текущий заказ, выполняемый роботом
   * **\_work\_now: bool** - работает ли сейчас робот

**Публичные методы:**

* + **set\_order(order: Order): void** - установить заказ для выполнения
  + **start\_order(): void** - начать выполнение заказа
  + **available(): bool** - доступен ли для выполнения заказов

# Описание работы программы

Изображение выглядит как снимок экрана, текст, программное обеспечение, дисплей

Автоматически созданное описаниеПри запуске программы запускается графический интерфейс, который позволяет управлять системой в режиме реального времени.

Изображение выглядит как электроника, текст, снимок экрана, дисплей

Автоматически созданное описаниеВ качестве пошагового отображения результатов реализовано логирование:

Изображение выглядит как текст, Шрифт, число, линия

Автоматически созданное описаниеИзображение выглядит как текст, Шрифт, число, снимок экрана

Автоматически созданное описаниеВ автоматическом режиме данные собираются, обрабатываются с помощью скрипта и выводятся такие таблицы:

# Результаты работы

**Определение количества реализаций:**

Количество реализаций, необходимое для получения нужной точности при заданной доверительной вероятности, можно оценивать по формуле:

, где p — вероятность отказа заявкам в обслуживании,

=1.643 для α = 0.9,

= 0.1 — относительная точность.

По результатам работы программы получено, что в большинстве случаев для достижения заданной точности необходимо около 2000 заявок. Однако, в случаях, когда p мало (<0.05) для достижения точности в 10% может потребоваться существенно больше заявок (10000–20000).

# Анализ результатов:

Т. к. целью моделирования является выбор конфигурации системы, требующей наименьшее количество ресурсов и обрабатывающей максимальный поток информации, то начнем с проверки конфигурации с макс. числом источников и минимальным числом приборов и мин. размером буфера.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Число источников | Число приборов | Размер буфера | Lambda | Загруженность | P отк | T в системе |
| 10 | 20 | 20 | 7 | 70% | 4% | 372 мс |
| 10 | 15 | 15 | 7 | 94% | 10% | 352 мс |
| 10 | 17 | 2 | 7 | 99% | 11% | 309 мс |

# Стоимость компонентов системы

|  |  |  |
| --- | --- | --- |
| Наименование | Характеристики | Цена |
| Сервер iRU Rock w9208p с процессором 2х Intel Xeon Gold 6126, 2.6 ГГц | Время обслуживания заявки: ~300 мс | 490 780 рублей |
| Сервер Сервер iRU Rock c2212p, 2U | Время обслуживания заявки: ~200мс | 774 860 рублей |
| Память DDR4 Kingston KSM26RD4/32HDI 32ГБ DIMM, ECC, registered, PC4-21300, CL19, 2666МГц | Объем: 32768 МБ | 13 990 рублей |

# Вывод

В ходе курсовой работы была написана система массового обслуживания на языке C++. С помощью данной программы была проанализирована реальная система и подобрана максимально выгодная комплектация данной системы. Оптимальными вариантами оказались вариант с 17 инструментами с размером стека 2 при 20 источниках.