**Документация разработчика**

Данная документация написана для программы, которая симулирует работу сети массового обслуживания с двумя типами заявок и возможностью ошибочной классификации. Программа разработана на C++ и Qt с использованием дискретно-событийного подхода. Основными событиями в программе являются прибытие заявки и выход заявки из узла. В программе также используется сторонняя библиотека QCustomPlot для построения графиков. Далее будут описаны используемые в программе классы. Если в описании метода явно не указан модификатор доступа, то подразумевается публичный метод.

1. Класс **Job** описывает заявку.

*Атрибуты:*

* **job\_type\_** – тип заявки;
* **assigned\_node\_** – узел, в который была классифицирована заявка;
* **system\_arrival\_time**\_ – время прибытия заявки в систему;
* **node\_arrival\_time\_**  – время прибытия в узел системы. Это значение обновляется с каждым прибытием заявки в узел;
* **service\_start\_time**\_ – время начала обработки заявки;
* **service\_end\_time\_** – время конца обработки заявки;
* **queue\_start\_time\_** – время попадания в очередь;
* **queue\_end\_time\_**  – время конца ожидания в очереди;
* **operation\_time\_** – время, которая заявка пребывала в системе до выхода из сети.

*Конструктор:*

**Job**(int job\_type**,** doublesystem\_arrival\_time) – устанавливает тип заявки, время прибытия в систему и время пребывания в узел (так как заявка прибывает непосредственно в узел по прибытии в систему)

*Методы:*

* void **SetOperationTime**() – устанавливает переменную **operation\_time\_**  как разницу времени последнего окончания обслуживания и времени прихода в систему;

Сеттеры:

* void **SetNodeArrival**(double time);
* void **SetServiceStart**(double time);
* void **SetServiceEnd**(double time);
* void **SetQueueStart**(double time);
* void **SetQueueEnd**(double time);
* void **SetAssignedNode**(int node\_index);

Геттеры:

* int **GetAssignedNode**() const;
* int **GetJobType**() const;
* double **GetSystemArrival**() const;
* double **GetNodeArrival**() const;
* double **GetServiceStart**() const;
* double **GetServiceEnd**() const;
* double **GetQueueStart**() const;
* double **GetQueueEnd**() const;
* double **GetOperationTime**() const;

1. Перечисляемый класс **ProcessDistr** содержит используемые в программе распределения: { exp\_plain, exp\_scaled, uniform } – экспоненциальное, экспоненциальное с множителями и равномерное соответственно.
2. Класс **Node** представляет собой узел.

*Атрибуты:*

* int **node\_type\_ –** тип (номер) узла;
* double **mu\_ –** значение параметра , определяющего интенсивность обработки верно классифицированной заявки в узле ;
* std::queue<Job> **process\_jobs\_ –** хранит заявку, находящуюся в обработке;
* std::queue<Job> **queued\_jobs\_** – хранит заявки, находящиеся в очереди на обработку. Используемая структура данных – очередь;
* double **idle\_start\_** – переменная для вычисления времени простоя. Содержит начало последнего периода простоя;
* double **idle\_end\_** – переменная для вычисления времени простоя. Содержит конец последнего периода простоя;
* double **idle\_time\_** – переменная для вычисления времени простоя. Содержит общее время, проведенное в простое;
* double **total\_system\_time\_** – общее время использования узла (как в работе, так и в простое);
* double **current\_job\_count\_time\_** – переменная для вычисления долей времени, проведенных с определенным количеством заявок. Хранит время, в которое узел сменил состояние в последний раз (когда изменилось количество заявок);
* ProcessDistr **err\_distr\_ –** распределение, используемое при обработке неверно классифицированных заявок;
* double **scaling\_par\_ –** параметр , который используется в параметре распределения времени обработки ошибочных заявок в узле . Данное значение используется только во второй и третьей моделях сети;
* std::vector<double> **job\_count\_duration\_ –** вектор, содержащий доли времени, проведенные с определенным количеством заявок;
* std::vector<double> **queue\_waiting\_times\_ –** вектор, накапливающий времена ожидания в очереди;
* int **busy\_upon\_arrival\_count\_**  **–** счетчик, увеличивающий каждый раз, когда очередная заявка по прибытии в узел попадает в очередь;
* int **false\_arrival\_count\_**  **–** счетчик, увеличивающийся каждый раз, когда в узле встречается неверно классифицированная заявка;

*Конструктор:*

**Node(**intnode\_type**,** doublemu**,** ProcessDistrerr\_distr**,** doublescaling\_par) **–**принимает соответственно тип узла, параметр экспоненциального распределения для верно классифицированных заявок, распределение для ошибочно классифицированных заявок и множитель этого распределения (для первой модели автоматически устанавливается равным 1).

*Методы:*

* void **JobArrival**(Job& j, double time) **–** направляет заявку на непосредственную обработку, в случае если сервер свободен, иначе добавляет заявку в конец очереди;
* void **JobProcessing**(Job& j, double time) **–** обрабатывает заявку, устанавливает переданное время начала и рассчитанное время конца обслуживания. Это время конца обслуживания используется для определения времени вызова JobDeparture;
* Job **JobDeparture**() **–** удаляет заявку, которая была обработана сервером, и возвращает ее. Если очередь непустая, то направляет на сервер первую заявку из очереди.;
* bool **HasFreeServerAvailable**() const **–** возвращает, свободен ли сервер;

Геттеры:

* int **GetNodeType**() const;
* double **GetMu**() const;
* ProcessDistr **GetErrorDistr**() const;
* double **GetScalingPar**() const;
* double **GetIdleStart**() const;
* double **GetIdleEnd**() const;
* double **GetIdleTime**() const;
* double **GetTotalSystemTime**() const;
* double **GetAverageQueueLength**() const;
* std::queue<Job> **GetProcessJobs**() const;
* std::queue<Job> **GetQueuedJobs**() const;
* std::vector<double> **GetQueueWaitingTimes**() const;
* std::vector<double> **GetJobCountDuration**() const;
* int **GetBusyUponArrivalCount**() const;
* int **GetFalseArrivalCount**() const;

1. Класс **Model** относится к симуляции в целом. Наследуется от **QObject** для поддержки механизма сигналов и слотов.

*Атрибуты:*

* const int **job\_type\_count\_** – количество типов заявок (программа реализована с учетом двух типов заявок);
* const int **processing\_node\_count\_** – количество обрабатывающих узлов (программа реализована с учетом двух обрабатывающих узлов);
* std::vector<double> **lambda\_params\_** – вектор из параметров и . Используется для генерации времени прихода заявок в систему;
* std::vector<double> **mu\_params\_** – вектор из параметров и ;
* std::vector<double> **class\_errors\_** – вектор, содержащий вероятности ошибок классификации и . Используется для определения узла во время генерации и повторной классификации заявок;
* std::vector<double> **scaling\_params\_** – вектор из параметров и . Используется только во второй и третьей моделях сети;
* std::vector<double> **prev\_arrival\_times\_** – используется во время генерации заявок. Данный вектор содержит времена прибытия в систему последней заявки для каждого типа;
* ProcessDistr **err\_distr\_** – распределение времени обработки ошибочных заявок, используемое в узлах;
* std::vector<Node> **nodes\_** – вектор, содержащий объекты класса **Node**. Содержит только обабатывающие узлы (без классификатора);;
* std::vector<std::vector<double>> **system\_time\_stats\_** – двумерный вектор, первый и второй элементы которого – это векторы, содержащие времена пребывания для первого и второго типа заявок соответственно;
* std::vector<std::vector<double>> **final\_service\_waiting\_time\_stats\_** – двумерный вектор, первый и второй элементы которого – это векторы, содержащие времена ожидания верной обработки для первого и второго типа заявок соответственно;
* std::vector<std::vector<double>> **node\_time\_spent\_** – двумерный вектор, первый и второй элементы которого – это векторы, содержащие времена, которые заявки (вне зависимости от типа) проводят в первом и втором узле соответственно;
* std::queue<Job> **jobs\_to\_process\_** – хранит сгенерированные заявки;
* bool **simulation\_stop** – переменная, которая отвечает за прерывание симуляции;

*Конструктор:*

**Model**(int num\_of\_process\_nodes = 2, int num\_of\_job\_types = 2, std::vector<double> lambda\_params = { 1.0, 1.0 }, std::vector<double> mu\_params = { 2.0, 2.0 }, std::vector<double**>** class\_errors = { 0.0, 0.0 }, ProcessDistr err\_distr = ProcessDistr::exp\_plain, std::vector<double> scaling\_params = { 1, 1 }) – принимает соответственно количество обрабатывающих узлов (программа реализована для двух), количество типов заявок (программа реализована для двух), параметры , параметры , ошибки классификации , распределение для ошибочно классифицированных заявок, множители этого распределения. Инициализирует узлы с помощью переданных параметров.

*Методы:*

* bool **StartSimulation**(quint64 count) – запускает симуляцию. Выполняет обработку событий в хронологическом порядке. На каждой итерации определяет ближайшее событие и вызывает методы объектов класса Node. В качестве параметра принимает общее количество заявок, которые пойдут на вход системы.;
* bool **BatchArrival**(quint64 count) – создает заданное количество заявок на вход системы;
* void **ReclassifyJob**(Job& job) – выполняет повторную классификацию для заявки;
* int **GetRespectiveNodeId**(int job\_type) const – приватная функция, используемая в методах **BatchArrival** и **ReclassifyJob** для получения назначенного узла с учетом ошибки классификации;
* void **reset**() – сбрасывает результаты симуляции. Используется для повторного использования объекта класса **Model** в случае нескольких итераций симуляции;

Геттеры:

* std::vector<Node> **GetNodes**() const;
* int **GetJobTypeCount**() const;
* int **GetProcessingNodeCount**() const;
* std::vector<double> **GetLambdaParams**() const;
* std::vector<double> **GetClassErrors**() const;
* std::vector<double> **GetMuParams**() const;
* std::vector<double> **GetScalingParams**() const;
* ProcessDistr **GetErrDistr**() const;
* std::vector<std::vector<double>> **GetSystemTimeStats**() const;
* std::vector<std::vector<double>> **GetFinalServiceWaitingTimeStats**() const;
* std::vector<std::vector<double>> **GetNodeTimeSpentStats**() const;

Сигналы:

* void **progressChanged**(int value) – сигнал, который используется для демонстрации прогресса выполнения симуляции;
* void **processInfoChanged**(const QString& text) – сигнал, который используется для демонстрации текущего состояния выполнения симуляции (генерация заявок или обработка заявок);

Слоты:

* void **stop**() – слот, который отвечает за прерывание симуляции;

1. Класс **PlotWindow** отвечает за окно с графиками.

*Атрибуты:*

* Ui::PlotWindow \***ui –** интерфейс класса;
* QCPBars \***barsFirst** – переменная, представляющая собой столбчатую диаграмму долей времени в определенном состоянии для первого узла;
* QCPBars \***barsSecond** – переменная, представляющая собой столбчатую диаграмму долей времени в определенном состоянии для первого узла;

*Конструктор:*

explicit **PlotWindow**(QWidget \*parent = nullptr) – принимает родительский виджет, подготавливает графики;

*Методы:*

Сигналы:

* void **plotReady**() – вызывается после построения графиков, делает активным кнопку для показа графиков;

Слоты:

* void **onPlotDataChanged**(const QVector<double>& first, const QVector<double>& second) – публичный слот, вызывающийся при обновлении данных для построения графиков. Принимает соответственно значения для первого и второго графика;
* void **on\_save\_first\_plot\_clicked**() – приватный слот, вызывающийся при сохранении графика для первого узла;
* void **on\_save\_second\_plot\_clicked**() – приватный слот, вызывающийся при сохранении графика для второго узла;

1. Класс **MainWindow** относится к основному окну программы.

*Атрибуты:*

* int **iteration\_number –** переменная, хранящая текущую итерацию симуляции;
* int **iteration\_count –** переменная, хранящая общее количество итераций симуляции;
* Ui::MainWindow \***ui –** интерфейс класса;
* Model\* **model** – объект класса **Model**, хранит модель, описывающую текущую симуляцию;
* PlotWindow\* **plotWindow** – объект класса **PlotWindow**, хранит окно с графиками;

*Конструктор:*

**MainWindow(**QWidget **\***parent **=** nullptr**)**  – принимает родительский виджет, устанавливает валидаторы для всех полей ввода, инициализирует **plotWindow** и подключает сигналы;

*Методы:*

* bool **checkInputs**() – приватный метод, производящий проверку введенных данных. Также предупреждает о нестабильности системы в случае определенных входных данных;
* void **disableInput**() – приватный метод, который выполняется после запуска симуляции. Делает поля и кнопки недоступными для изменения, и меняет внешний вид окна для показа процесса симуляции;
* void **startSimulation**() – приватный метод, который считывает данные из полей, подготавливает переменные для хранения характеристик и выполняет симуляцию с указанным количеством итераций;
* void ***closeEvent***(QCloseEvent \*event) override – переопределенный защищенный метод, который останавливает симуляцию при закрытии окна;

Сигналы:

* void **iterationNumberChanged**() – вызывается в начале каждой итерации;
* void **outputTextChanged**(const QString& text) – вызывается для добавления текста в панель вывода;
* void **setUpPlotData**(const QVector<double>& first, const QVector<double>& second) – вызывается для передачи данных для графиков, которые принимает слот **PlotWindow**::**onPlotDataChanged**;

Слоты:

* void **on\_pushButton\_clicked**() – приватный слот, вызывается при нажатии кнопки *Старт*/*Стоп*, соответственно запускает/останавливает симуляцию;
* void **on\_distribution\_selection\_currentTextChanged**(const QString &arg1) – приватный слот, вызывается при изменении распределения для ошибочно классифицированных заявок. В случае изменения на экспоненциальное, делает недоступными параметры ;
* void **on\_show\_plot\_clicked**() – приватный слот, вызывается при нажатии *График состояний*;
* void **on\_clear\_output\_clicked**() – приватный слот, вызывается при нажатии *Очистить вывод*;
* void **enableInput**() – приватный слот, который делает доступными для ввода различные входные параметры симуляции;
* void **onOutputTextChange**(const QString& text) – приватный слот, который добавляет переданный текст в панель вывода;
* void **onProcessInfoChange**(const QString& text) – приватный слот, который устанавливает текущий этап итерации (*Генерация заявок*/*Обработка заявок*);
* void **onIterationNumberChange**() – приватный слот, обновляющий текст о номере итерации;
* void **onProgressChange**(int value) – приватный слот, обновляющий прогресс текущего этапа итерации;
* void **onPlotReady**() – приватный слот, устанавливающий доступным для показа *График состояний*;

Далее опишем основную часть программы. В функции *main* выполняется создание объекта класса QTranslator для перевода некоторых кнопок интерфейса. После этого создается объект класса **MainWindow**. Обратим внимание на некоторые строчки кода в теле конструктора:

QSizePolicy sp = ui->progressBar->sizePolicy(); **// получаем правила, относящиеся к положению и размеров для полоски с прогрессом**

sp.setRetainSizeWhenHidden(true); **// добавляем правило, при котором скрытие элемента не освобождает место, таким образом размещение элементов не будет нарушено**

ui->progressBar->setSizePolicy(sp); **// устанавливаем правило для полоски с прогрессом**

ui->progressBar->hide(); **// так как полоска прогресса показана только во время симуляции, изначально скрываем ее**

Далее через регулярные выражения устанавливаются валидаторы для полей ввода, чтобы пользователь не мог ввести недопустимые входные данные.

plotWindow = new PlotWindow(); **// присваиваем собственному полю окно для графиков**

connect(this, SIGNAL(outputTextChanged(const QString&)), this, SLOT(onOutputTextChange(const QString&))); **// привязываем к этому же объекту сигнал о добавлении текста в панель вывода, поскольку вызов сигнала будет производиться в другом потоке**

connect(plotWindow, SIGNAL(plotReady()), this, SLOT(onPlotReady())); **// сигнал о готовности графиков**

connect(this, SIGNAL(setUpPlotData(const QVector<double>&, const QVector<double>&)), **// сигнал при обновлении данных для графиков**

plotWindow, SLOT(onPlotDataChanged(const QVector<double>&, const QVector<double>&)));

При создании объекта **PlotWindow** производится подготовка графиков. Об использованных функциях подробно написано в документации QCustomPlot, поэтому здесь останавливаться на этом не будем.

С нажатием кнопки *Старт/Стоп:*

void MainWindow::**on\_pushButton\_clicked**() {

if (ui->pushButton->text() == "Старт") {

if (checkInputs()) { **// проверка входных данный**

disableInput(); **// отключение полей и смена некоторых элементов для демонстрации процесса симуляции**

ui->progressBar->show(); **// отображение полоски прогресса**

QFuture<void> result = QtConcurrent::run(startSimulation,

this); **// запуск симуляции в отдельном потоке**

QFutureWatcher<void> \*watcher = new QFutureWatcher<void>(this);

connect(watcher, SIGNAL(finished()), this, SLOT(enableInput())); **// возобновление полей для ввода по завершению симуляции**

connect(watcher, SIGNAL(finished()), watcher, SLOT(deleteLater())); **// удаление по завершению симуляции**

watcher->setFuture(result);

}

}

**// этот блок выполняется при нажатии на «Стоп»**

else if (model != nullptr) {

model->stop(); **// вызывает остановку симуляции, после чего срабатывают сигналы, подключенные сверху**

}

}

Опишем основные функций, относящиеся к симуляции. Так как методы класса **Job** не представляют особый интерес, начнем с класса **Node**:

void Node::**JobArrival**(Job& job, double arrival\_time) {

**// следующий блок if/else накапливает значения для вычисления времени, которое проводится с определенным количеством заявок. Если состояние (в котором узел перестанет быть по прибытии заявки) еще не достигалось, то добавляем новое значение в вектор job\_count\_duration, иначе прибавляем к уже существующему значению**

if (job\_count\_duration\_.size() < process\_jobs\_.size() + queued\_jobs\_.size() + 1) {

job\_count\_duration\_.push\_back(arrival\_time - current\_job\_count\_time\_);

}

else {

job\_count\_duration\_[process\_jobs\_.size() + queued\_jobs\_.size()] += arrival\_time - current\_job\_count\_time\_;

}

if (job.GetJobType() != node\_type\_)

++false\_arrival\_count\_; **// увеличение счетчика при несопадении типа узла и заявки**

**// если сервер свободен, то заявка туда попадает, иначе в очередь**

if (HasFreeServerAvailable()) {

queue\_waiting\_times\_.push\_back(0); **// в этом случае заявка не ждет в очереди, поэтому добавляем значение 0 в вектор**

idle\_end\_ = arrival\_time; **// при прибытии заявки узел перестает простаивать**

idle\_time\_ += idle\_end\_ - idle\_start\_; **// прибавляем накопленное с предыдущего раза время простаивания**

JobProcessing(job, arrival\_time);

}

else {

++busy\_upon\_arrival\_count\_; **// увеличиваем счетчик для случая занятого сервера**

job.SetQueueStart(arrival\_time);

queued\_jobs\_.push(job); **// помещаем заявку в очередь**

}

current\_job\_count\_time\_ = arrival\_time; **// устанавливаем начало времени с текущим состоянием узла**

total\_system\_time\_ = arrival\_time; **// обновляем общее время**

}

**/\* обработка заявки, а также определение времени окончания обработки**

**второй параметр здесь нужен для заявок, пришедших из очереди, так как для них время начала обработки равно времени ухода предыдущей заявки, и это нужно учитывать \*/**

void Node::**JobProcessing**(Job& job, double start\_time) {

std::random\_device rd;

std::mt19937 gen(rd());

std::exponential\_distribution<> distribution(mu\_);

**// если заявка верно классифицирована либо используется модель 1, то используем экспоненциальное распределение без доп. коэфф-ов**

if (node\_type\_ == job.GetJobType() || err\_distr\_ == ProcessDistr::exp\_plain) {

job.SetServiceStart(start\_time);

job.SetServiceEnd(start\_time + distribution(gen)); **// сразу устанавливаем время окончания обработки**

}

else {

**// случай ошибочной классификации для модели 2**

if (err\_distr\_ == ProcessDistr::exp\_scaled) {

std::exponential\_distribution<> err\_distribution(mu\_ \* scaling\_par\_);

job.SetServiceStart(start\_time);

job.SetServiceEnd(start\_time + err\_distribution(gen));

}

**// случай ошибочной классификации для модели 3**

else if (err\_distr\_ == ProcessDistr::uniform) {

std::uniform\_real\_distribution<> err\_distribution(0, 2 / (mu\_ \* scaling\_par\_));

job.SetServiceStart(start\_time);

job.SetServiceEnd(start\_time + err\_distribution(gen));

}

}

process\_jobs\_.push(job);

}

**// выход заявки из узла**

Job Node::**JobDeparture**() {

Job finished\_job = process\_jobs\_.front(); **// берем первую (и единственную) заявку из сервера**

**// аналогично JobArrival накапливает значения для вычисления времени, которое проводится с определенным количеством заявок**

if (job\_count\_duration\_.size() < process\_jobs\_.size() + queued\_jobs\_.size() + 1) {

job\_count\_duration\_.push\_back(finished\_job.GetServiceEnd() - current\_job\_count\_time\_);

}

else {

job\_count\_duration\_[process\_jobs\_.size() + queued\_jobs\_.size()] += finished\_job.GetServiceEnd() - current\_job\_count\_time\_;

}

process\_jobs\_.pop(); **// удаляет (единственную) убывающую заявку из сервере**

**// если очередь непуста, то освободившийся сервер занимает первая заявка оттуда**

if (queued\_jobs\_.size()) {

Job next = queued\_jobs\_.front();

queued\_jobs\_.pop();

double process\_start\_time = finished\_job.GetServiceEnd();

next.SetQueueEnd(process\_start\_time);

queue\_waiting\_times\_.push\_back(next.GetQueueEnd() - next.GetQueueStart());

JobProcessing(next, process\_start\_time); **// заявка из очереди начинает обработку, как только освободился сервер**

}

else {

idle\_start\_ = finished\_job.GetServiceEnd();

}

total\_system\_time\_ = finished\_job.GetServiceEnd();

current\_job\_count\_time\_ = total\_system\_time\_;

return finished\_job; **// возвращаем обработанную заявку для дальнейшего использования**

}

Теперь перейдем к методам класса **Model**:

**// функция-классификатор, возвращает номер назначенного узла**

int Model::**GetRespectiveNodeId**(int job\_type) const {

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_real\_distribution<> dis(0.0, 1.0);

double value = dis(gen);

if (value > class\_errors\_[job\_type]) **// ошибку классификации проведем сравнением случайной равномерной величины, если она больше ошибки классификации, то классификатор верно определяет заявку**

return job\_type;

else

return 1 - job\_type;

}

**// создание 'count' заявок в хронологической последовательности**

bool Model::**BatchArrival**(quint64 count) {

std::random\_device rd;

std::mt19937 gen(rd());

std::vector<int> node\_count(processing\_node\_count\_, 0);

std::vector<double> temp\_arrival\_times(job\_type\_count\_, 0); **// массив с предварительным временем прибытия для каждой заявки**

int prev\_job\_type = -1;

**// сначала рассчитываем предварительное время прибытия для каждой заявки**

for (int i = 0; i < job\_type\_count\_; ++i) {

std::exponential\_distribution<double> distribution(lambda\_params\_[i]);

temp\_arrival\_times[i] = prev\_arrival\_times\_[i] + distribution(gen);

}

int progress = 0;

for (int k = 0; k < count; ++k) {

if (simulation\_stop) { **// в случае прерывания системы возвращаем false**

return false;

}

if (prev\_job\_type != -1) { **// на первой итерации пропускаем этот шаг**

**// только для предыдущей заявки рассчитываем следующее предварительное время прибытия**

std::exponential\_distribution<double> distribution(lambda\_params\_[prev\_job\_type]);

temp\_arrival\_times[prev\_job\_type] = prev\_arrival\_times\_[prev\_job\_type] + distribution(gen);

}

**// получаем тип следующей в очереди заявки**

int next\_job\_type = distance(begin(temp\_arrival\_times),

min\_element(begin(temp\_arrival\_times), end(temp\_arrival\_times)));

double arrival\_time = temp\_arrival\_times[next\_job\_type];

Job job(next\_job\_type, arrival\_time);

int corresp\_node\_ind = GetRespectiveNodeId(next\_job\_type); **// классифицируем заявку**

job.SetAssignedNode(corresp\_node\_ind);

++node\_count[corresp\_node\_ind]; **// увеличиваем счетчик количества заявок, классифицированных в каждый узел**

prev\_arrival\_times\_[next\_job\_type] = arrival\_time;

jobs\_to\_process\_.push(job);

prev\_job\_type = next\_job\_type; **// обновляем значение предыдущего типа заявки**

if (progress < static\_cast<int>(static\_cast<double>(k) \* 100 / count)) {

progress = static\_cast<int>(static\_cast<double>(k) \* 100 / count);

emit progressChanged(progress); **// оповещаем о состоянии прогресса обработки заявок**

}

}

return true;

}

**// производит повторную классификацию заявки**

void Model::**ReclassifyJob**(Job& job) {

int corresp\_node\_ind = GetRespectiveNodeId(job.GetJobType()); **// повторно классифицируем заявку**

job.SetAssignedNode(corresp\_node\_ind);

job.SetNodeArrival(job.GetServiceEnd()); **// так как классификация мгновенная, сразу устанавливаем время прибытия в узел**

}

**// функция обработки заявок**

bool Model::**StartSimulation**(quint64 count) {

processInfoChanged("Генерация заявок..."); **// оповещаем о смене этапа текущей итерации**

bool generation\_check = BatchArrival(count); **// проводим генерацию заявок и прерываем симуляцию, если генерация была прервана**

if (!generation\_check) {

return false;

}

int check\_count = 0, progress = 0; **// для вывода прогресса**

double last\_timepoint = -1; **// для проверок адекватности работы программы**

processInfoChanged("Обработка заявок..."); **// оповещаем о смене этапа текущей итерации**

while (jobs\_to\_process\_.size()) { **// итерируемся, пока каждая из сгенерированных заявок не будет классифицирована в узел**

if (simulation\_stop) { **// в случае прерывания симуляции досрочно выходим, возвращая false**

return false;

}

Job first\_arrived\_job = jobs\_to\_process\_.front(); **// берем первую заявку из очереди сгенерированных заявок**

int num\_of\_node = first\_arrived\_job.GetAssignedNode();

Node& current\_node = nodes\_[num\_of\_node];

Node& other\_node = nodes\_[1 - num\_of\_node];

double first\_time = first\_arrived\_job.GetNodeArrival(); **// время прибытия текущей заявки**

**// происходит событие, которое должно произойти следующим по времени**

if ((current\_node.HasFreeServerAvailable() || first\_time < current\_node.GetProcessJobs().front().GetServiceEnd())

&& (other\_node.HasFreeServerAvailable() || first\_time < other\_node.GetProcessJobs().front().GetServiceEnd())) {

**// в данный блок попадаем, в случае если происходит событие прибытия заявки из очереди в узел**

current\_node.JobArrival(first\_arrived\_job, first\_time);

jobs\_to\_process\_.pop(); **// удаляем заявку из очереди сгенерированных**

++check\_count;

if (last\_timepoint > first\_time) {

qDebug() << "Ошибка: была нарушена хронологическая последовательность событий во время обработки заявок.";

return false;

}

last\_timepoint = first\_time;

}

else {

**// в этот блок попадаем, если окончание обслуживания заявки в каком-либо из узлов произошло раньше, чем прибытие первой заявки из очереди сгенерированных**

int departing\_job\_node\_type = -1;

**// вычисление номера узла, у которого произошло окончание обслуживания заявки**

if (nodes\_[num\_of\_node].HasFreeServerAvailable()) {

if (other\_node.HasFreeServerAvailable()) {

departing\_job\_node\_type = num\_of\_node;

}

else {

if (first\_time < other\_node.GetProcessJobs().front().GetServiceEnd()) {

departing\_job\_node\_type = num\_of\_node;

}

else {

departing\_job\_node\_type = 1 - num\_of\_node;

}

}

}

else {

if (other\_node.HasFreeServerAvailable()) {

departing\_job\_node\_type = num\_of\_node;

}

else {

if (nodes\_[num\_of\_node].GetProcessJobs().front().GetServiceEnd() <

other\_node.GetProcessJobs().front().GetServiceEnd()) {

departing\_job\_node\_type = num\_of\_node;

}

else {

departing\_job\_node\_type = 1 - num\_of\_node;

}

}

}

double next\_departure\_time = nodes\_[departing\_job\_node\_type].GetProcessJobs().front().GetServiceEnd(); **// получаем время следующего отправления из узла**

if (last\_timepoint > next\_departure\_time) {

qDebug() << "Ошибка: была нарушена хронологическая последовательность событий во время обработки заявок.";

return false;

}

last\_timepoint = next\_departure\_time;

Job processed\_job = nodes\_[departing\_job\_node\_type].JobDeparture();

if (next\_departure\_time != processed\_job.GetServiceEnd()) {

qDebug() << "Ошибка: время окончания обслуживания не совпадает с ожидаемым значением.";

return false;

}

node\_time\_spent\_[departing\_job\_node\_type].push\_back(processed\_job.GetServiceEnd() - processed\_job.GetNodeArrival()); **// добавляем время, проведенное в данном узле**

if (node\_time\_spent\_[departing\_job\_node\_type].back() < 0) {

qDebug() << "Ошибка: время пребывания в узле оказалось отрицательным.";

return false;

}

if (departing\_job\_node\_type == processed\_job.GetJobType()) {

**// сюда попадаем, если заявка была верно классифицирована**

processed\_job.SetOperationTime();

if (processed\_job.GetOperationTime() < 0) {

qDebug() << "Ошибка: время пребывания в системе оказалось отрицательным.";

return false;

}

system\_time\_stats\_[departing\_job\_node\_type].push\_back(processed\_job.GetOperationTime()); **// добавляем время пребывания заявки в системе**

final\_service\_waiting\_time\_stats\_[departing\_job\_node\_type].push\_back(processed\_job.GetServiceStart() - processed\_job.GetSystemArrival()); **// добавляем время, проведенное до ожидания верной обработки**

}

else {

**// сюда попадаем, если заявка была неверно классифицирована, поэтому проводим повторную классификацию**

ReclassifyJob(processed\_job);

nodes\_[processed\_job.GetAssignedNode()].JobArrival(processed\_job, processed\_job.GetNodeArrival()); **// сразу же вызываем прибытие заявки в назначенный узел, так как классификация мгновенная**

}

}

if (progress < int(double(check\_count) \* 100 / count)) {

progress = int(double(check\_count) \* 100 / count);

emit progressChanged(progress); **// оповещаем о прогрессе данного этапа итерации**

}

}

return true;

}

Наконец, опишем функцию из **MainWindow**, которая начинает симуляцию:

void MainWindow::**startSimulation**() {

**// считываем данные из полей**

double lambda1 = ui->lambda1\_input->text().replace(',', ".").toDouble();

double lambda2 = ui->lambda2\_input->text().replace(',', ".").toDouble();

double mu1 = ui->mu1\_input->text().replace(',', ".").toDouble();

double mu2 = ui->mu2\_input->text().replace(',', ".").toDouble();

double alpha1 = ui->alpha1\_input->text().replace(',', ".").toDouble();

double alpha2 = ui->alpha2\_input->text().replace(',', ".").toDouble();

double gamma1 = ui->gamma1\_input->text().replace(',', ".").toDouble();

double gamma2 = ui->gamma2\_input->text().replace(',', ".").toDouble();

quint64 arriving\_job\_count = ui->arriving\_job\_count->text().toULongLong();

iteration\_count = ui->iteration\_count->text().toUInt();

ProcessDistr err\_distr;

bool output\_every\_iteration = ui->checkBox->isChecked();

if (ui->distribution\_selection->currentText() == "Экспоненциальное") {

err\_distr = ProcessDistr::exp\_plain;

}

else if (ui->distribution\_selection->currentText() == "Экспоненциальное масштабированное") {

err\_distr = ProcessDistr::exp\_scaled;

}

else {

err\_distr = ProcessDistr::uniform;

}

**// подготавливаем текст про входные данные системы**

std::stringstream system\_info\_text;

system\_info\_text << "Параметры системы: " << "\n";

system\_info\_text << "λ1: " << lambda1 << '\t' << "λ2: " << lambda2 << "\n";

system\_info\_text << "μ1: " << mu1 << '\t' << "μ2: " << mu2 << "\n";

system\_info\_text << "α1: " << alpha1 << '\t' << "α2: " << alpha2 << "\n";

if (err\_distr != ProcessDistr::exp\_plain) {

system\_info\_text << "γ1: " << gamma1 << '\t' << "γ2: " << gamma2 << "\n";

}

system\_info\_text << "\nРаспределение обслуживания ошибочных заявок:\n";

if (err\_distr == ProcessDistr::exp\_scaled) {

system\_info\_text << "экспоненциальное с параметрами " << mu1 \* gamma1

<< " и " << mu2 \* gamma2 << "\n";

}

else if (err\_distr == ProcessDistr::uniform) {

system\_info\_text << "равномерное [0, " << 2 / (mu1 \* gamma1)

<< "] и [0, " << 2 / (mu2 \* gamma2) << "]\n";

}

else {

system\_info\_text << "экспоненциальное с параметрами " << mu1 << " и " << mu2 << "\n";

}

system\_info\_text << "\nКоличество заявок на вход системы: " << arriving\_job\_count;

system\_info\_text << "\nКоличество итераций: " << iteration\_count;

emit outputTextChanged(QString::fromStdString(system\_info\_text.str()));

**// инициализируем переменные, в которых будут собираться результаты характеристик с итераций, впоследствии они усредняются для вывода итоговых результатов симуляции**

std::vector<double> combined\_system\_time\_stats(2, 0);

std::vector<double> combined\_final\_service\_waiting\_time\_stats(2, 0);

std::vector<double> combined\_node\_time\_spent\_times(2, 0);

std::vector<double> combined\_queue\_lengths(2, 0);

std::vector<double> combined\_queue\_waiting\_times(2, 0);

std::vector<double> combined\_idle\_times(2, 0);

std::vector<double> combined\_job\_count(2, 0);

std::vector<std::vector<double>> combined\_job\_count\_durations(2);

model = new Model(

2, **// количество типов заявок**

2, **// количество обрабатывающих узлов**

{ lambda1, lambda2 }, **// интенсивности прибытия заявок**

{ mu1, mu2 }, **// параметры экспоненциального распределения для облуживания заявок**

{ alpha1, alpha2 }, **// ошибка классификации**

err\_distr, **// распределение обработки ошибочных заявок**

{ gamma1, gamma2 } **// значения, используемые для распределения обработки ошибочных заявках**

);

**// подключаем сигналы**

connect(model, &Model::progressChanged, this, onProgressChange, Qt::BlockingQueuedConnection);

connect(model, &Model::processInfoChanged, this, onProcessInfoChange);

connect(this, iterationNumberChanged, this, onIterationNumberChange);

bool simulation\_check; **// переменная, которая контролирует прерывание симуляции**

auto simulation\_start = std::chrono::steady\_clock::now(); **// фиксируем время начала симуляции**

iteration\_number = 0;

for (; iteration\_number < iteration\_count; ++iteration\_number) {

emit iterationNumberChanged(); **// оповещаем об изменении** **номера итерации**

auto iteration\_start = std::chrono::steady\_clock::now(); **// фиксируем время начала итерации**

simulation\_check = model->StartSimulation(arriving\_job\_count);

auto iteration\_finish = std::chrono::steady\_clock::now();

if (!simulation\_check) { **// если симуляция была прервана, то выходим из цикла**

break;

}

**// если при входных данных был выбран «вывод после каждой итерации», то выводим характеристики внутри цикла**

if (output\_every\_iteration) {

QString str = "----------------------------------------------------------------------";

str += "Результаты итерации #" + QString::number(iteration\_number + 1) + ":\n";

emit outputTextChanged(str);

}

**// получаем результаты симуляции**

std::vector<std::vector<double>> system\_time\_stats = model->GetSystemTimeStats();

std::vector<std::vector<double>> final\_service\_waiting\_time\_stats = model->GetFinalServiceWaitingTimeStats();

std::vector<std::vector<double>> node\_time\_spent\_stats = model->GetNodeTimeSpentStats();

std::vector<std::vector<double>> job\_count\_durations;

std::vector<Node> nodes = model->GetNodes();

for (int i = 0; i < nodes.size(); ++i) {

job\_count\_durations.push\_back(nodes[i].GetJobCountDuration());

}

std::stringstream results\_text; **// поток для вывода результатов**

**// получаем средние значения характеристик, которые накапливали за итерацию**

for (int i = 0; i < system\_time\_stats.size(); ++i) {

double average\_system\_time = std::accumulate(begin(system\_time\_stats[i]), end(system\_time\_stats[i]), 0.0)

/ system\_time\_stats[i].size();

double average\_final\_service\_waiting\_time = std::accumulate(begin(final\_service\_waiting\_time\_stats[i]), end(final\_service\_waiting\_time\_stats[i]), 0.0)

/ final\_service\_waiting\_time\_stats[i].size();

combined\_system\_time\_stats[i] += average\_system\_time;

combined\_final\_service\_waiting\_time\_stats[i] += average\_final\_service\_waiting\_time;

results\_text << "Среднее время пребывания заявок " << i+1 << "-го типа: " << average\_system\_time << "\n";

results\_text << "Среднее время ожидания верной обработки заявок " << i+1 << "-го типа: " << average\_final\_service\_waiting\_time << "\n";

}

for (int i = 0; i < nodes.size(); ++i) {

results\_text << "\n" << "Узел #" << i+1 << ":" << "\n";

double idle\_time = nodes[i].GetIdleTime();

double total\_system\_time = nodes[i].GetTotalSystemTime();

std::vector<double> queue\_waiting\_times = nodes[i].GetQueueWaitingTimes();

std::vector<double> job\_count\_duration = job\_count\_durations[i];

double avg\_node\_time\_spent = std::accumulate(begin(node\_time\_spent\_stats[i]), end(node\_time\_spent\_stats[i]), 0.0)

/ node\_time\_spent\_stats[i].size();

double avg\_queue\_time\_spent = std::accumulate(begin(queue\_waiting\_times), end(queue\_waiting\_times), 0.0)

/ queue\_waiting\_times.size();

double avg\_idle\_time = idle\_time / total\_system\_time;

double average\_job\_count = 0;

double avg\_queue\_length = 0;

**// вычисляем характеристики по формулам**

for (int j = 0; j < job\_count\_duration.size(); ++j) {

job\_count\_duration[j] /= total\_system\_time;

if (j > 1) {

avg\_queue\_length += job\_count\_duration[j] \* (j-1);

}

average\_job\_count += job\_count\_duration[j] \* j;

if (combined\_job\_count\_durations[i].size() < j + 1) {

combined\_job\_count\_durations[i].push\_back(job\_count\_duration[j]);

}

else {

combined\_job\_count\_durations[i][j] += job\_count\_duration[j];

}

}

results\_text << "Средняя доля времени в простое: " << avg\_idle\_time << "\n";

results\_text << "Средняя длина очереди: " << avg\_queue\_length << "\n";

results\_text << "Среднее время ожидания в очереди: " << avg\_queue\_time\_spent << "\n";

// results\_text << "Среднее время, проведенное в узле: " << avg\_node\_time\_spent << "\n";

results\_text << "Среднее количество заявок в узле: " << average\_job\_count << "\n";

combined\_node\_time\_spent\_times[i] += avg\_node\_time\_spent;

combined\_queue\_lengths[i] += avg\_queue\_length;

combined\_queue\_waiting\_times[i] += avg\_queue\_time\_spent;

combined\_idle\_times[i] += avg\_idle\_time;

combined\_job\_count[i] += average\_job\_count;

}

results\_text << "\nДлительность " << std::to\_string(iteration\_number+1) << "-й итерации: " <<

std::chrono::duration\_cast<std::chrono::milliseconds>(iteration\_finish-iteration\_start).count()

<< " мс";

if (output\_every\_iteration) {

emit outputTextChanged(QString::fromStdString(results\_text.str())); **// на каждой итерации обновляем вывод в панели, если был выбран «вывод после каждой итерации»**

}

model->reset(); **// после итерации возвращаем модель в исходное состояние**

}

if (!simulation\_check) { **// сюда попадаем, если симуляция была прервана**

QString simulation\_stop\_text = "";

simulation\_stop\_text += "=======================================\n";

simulation\_stop\_text += "Симуляция была прервана во время " + QString::number(iteration\_number + 1) + "-й итерации";

if (iteration\_number == 0) {

simulation\_stop\_text += "\n=======================================\n";

}

emit outputTextChanged(simulation\_stop\_text); **// оповещаем о прерывании симуляции**

}

if (simulation\_check || iteration\_number > 0) { **// попадаем в следующий блок, если не было прерываний или была выполнена хотя бы одна итерация**

auto simulation\_finish = std::chrono::steady\_clock::now();

std::stringstream simulation\_results\_text;

if (simulation\_check) {

simulation\_results\_text << "=======================================\n";

}

simulation\_results\_text << "Результаты симуляции:\n\n";

**// выводим общие характеристики за все проведенные итерации**

for (int i = 0; i < combined\_system\_time\_stats.size(); ++i) {

simulation\_results\_text << "Среднее время пребывания заявок " << i+1 << "-го типа: " <<

combined\_system\_time\_stats[i] / iteration\_number << "\n";

simulation\_results\_text << "Среднее время ожидания верной обработки заявок " << i+1 << "-го типа: " <<

combined\_final\_service\_waiting\_time\_stats[i] / iteration\_number << "\n";

}

for (int i = 0; i < combined\_node\_time\_spent\_times.size(); ++i) {

simulation\_results\_text << "\nУзел #" << i+1 << ":\n";

simulation\_results\_text << "Средняя доля времени в простое: " << combined\_idle\_times[i] / iteration\_number<< "\n";

simulation\_results\_text << "Средняя длина очереди: " << combined\_queue\_lengths[i] / iteration\_number << "\n";

simulation\_results\_text << "Среднее время ожидания в очереди: " << combined\_queue\_waiting\_times[i] / iteration\_number << "\n";

// simulation\_results\_text << "Среднее время, проведенное в узле: " << combined\_node\_time\_spent\_times[i] / iteration\_number << "\n";

}

simulation\_results\_text << "\nДлительность симуляции: " <<

std::chrono::duration\_cast<std::chrono::milliseconds>(simulation\_finish-simulation\_start).count() << " мс" <<

"\n"; // << "----------------------------------------------------------------------" << "\n" << "Конец симуляции";

simulation\_results\_text << "=======================================\n";

for (int i = 0; i < combined\_job\_count\_durations.size(); ++i) {

for (int j = 0; j < combined\_job\_count\_durations[i].size(); ++j) {

combined\_job\_count\_durations[i][j] /= iteration\_number;

}

}

emit outputTextChanged(QString::fromStdString(simulation\_results\_text.str())); **// оповещаем о добавлении текста в панель вывода**

**// инициализируем QVector состояний и передаем их для построения графиков**

QVector<double> first = QVector<double>(combined\_job\_count\_durations[0].begin(), combined\_job\_count\_durations[0].end());

QVector<double> second = QVector<double>(combined\_job\_count\_durations[1].begin(), combined\_job\_count\_durations[1].end());

emit setUpPlotData(first, second); **// вызываем сигнал с полученными данными**

}

if (model != nullptr) {

delete model; **// удаляем модель**

model = nullptr;

}

}