Отчёт по курсовому проекту

(Бородин Дмитрий Сергеевич, Федоров Павел Вячеславович)

12 января 2019 г.

Оглавление

Введение

Сервер

Задание. 6

Используется вызов select и рабочие процессы. Журналирование в отдельном процессе.

Цели и задачи

Цель: Разработать **SMTP-сервер** с использованием рабочих процессов и select.
Задачи:

- Проанализировать архитектурное решение
- Разработать подход для обработки входящих соединений и хранения входящих писем в maildir
- Рассмотреть **SMTP**-протокол
- Реализовать программу для получения писем по протоколу **SMTP**

Клиент

Задание. 26

Используется вызов pselect и рабочие потоки. Журналирование в отдельном потоки. Не обязательно пытаться отправлять все сообщения для одного MX за одну сессию.

Цели и задачи

Цель: Разработать **SMTP-клиент** с использованием рабочих потоков и pselect.
Задачи:

- Проанализировать архитектурное решение maildir
- Разработать подход для обработки писем в maildir
- Рассмотреть **SMTP**-протокол
- Проанализировать способы получения и обработки **MX**-записей
- Реализовать программу для отправки писем по протоколу **SMTP**

Аналитический раздел

Предметная область

ER-диаграмма предметной области

Согласно обозначенному протоколу в рамках данной работы, в системе устанавливаются отношения "отправитель - получатель причем отправитель может отправить несколько писем, указав себя в качестве источника сообщения (единственного). Основная единица данных, передаваемая по протоколу - письмо, которое включает в себя отправителя и получателя, причем получателей может быть несколько. Также письмо содержит в себе единственное тело, которое может быть использовано как для последующей передачи, так и для хранения на сервере. Таким образом, в рамках предметной области можно выделить 4 вида сущностей:

- 1. Отправитель
- 2. Получатель
- 3. Письмо
- 4. Тело письма

Зависимость между сущностями предметной области может быть описана следующей диаграммой (1):

Сервер

Преимущества и недостатки условия задачи

Согласно условию задачи, в работе сервера предлагается исполльзовать многопроцессную систему. Данный тип системы является самым простым в плане разработки при условии, что на каждую пользовательскую сессию или даже любой пользовательский запрос создается новый процесс. Данная архитектура имеет следующие преимущества:

- 1. Простота разработки. Фактически, мы запускаем много копий однопоточного приложения и они работают независимо друг от друга. Можно не использовать никаких специфически многопоточных API и средств межпроцессного взаимодействия.
- 2. Высокая надежность. Аварийное завершение любого из процессов никак не затрагивает остальные процессы.

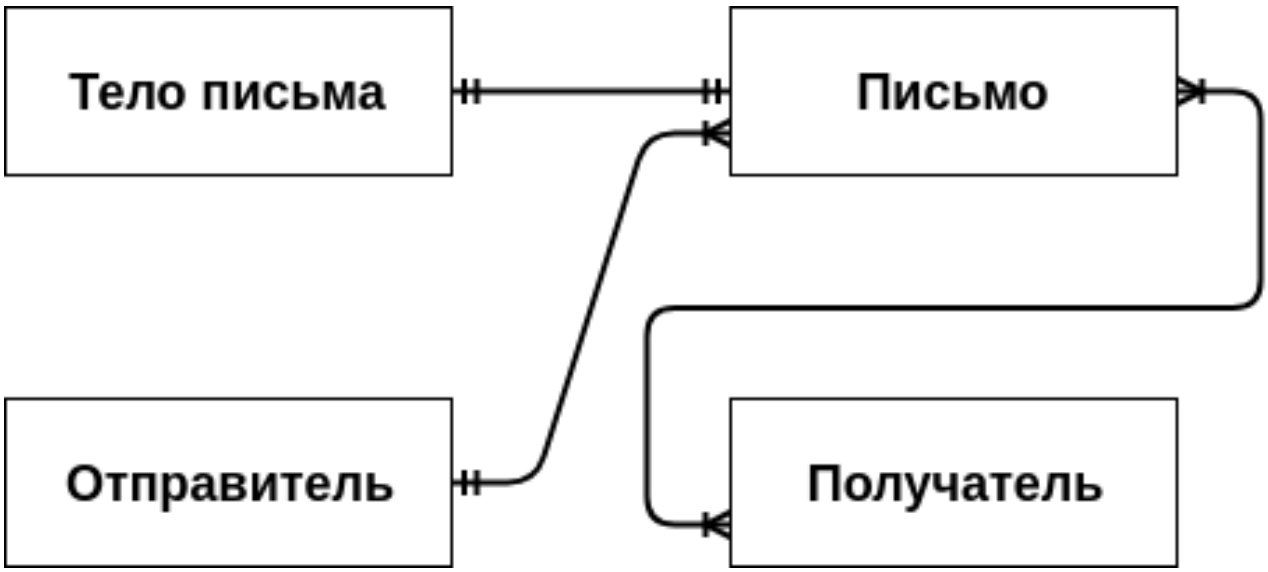

Рис. 1: ER-диаграмма сущностей

– 3. Хорошая переносимость. Приложение будет работать налюбой многозадачной ОС

– 4. Высокая безопасность. Разные процессы приложения могут запускаться от имени разных пользователей. Таким образом можно реализовать принцип минимальных привилегий, когда каждый из процессов имеет лишь те права, которые необходимы ему для работы. Даже если в каком-то из процессов будет обнаружена ошибка, допускающая удаленное исполнение кода, взломщик сможет получить лишь уровень доступа, с которым исполнялся этот процесс.

При этом данная архитектура имеет следующие недостатки:

– 1. Далеко не все прикладные задачи можно предоставлять таким образом. Например, эта архитектура годится для сервера, занимающегося раздачей статических HTMLстраниц, но совсем непригодна для сервера баз данных и многих серверов приложений.

– 2. Создание и уничтожение процессов – дорогая операция, поэтому для многих задач такая архитектура неоптимальна.

Поэтому для минимизации операций создания и уничтожения процессов предлагается архитектурное решение, представляющее собой пул процессов, создаанных заранее. Это позволит фиксировать число операций создания процесса. При этом слушающие сокеты сервера должны наследоваться каждым создаваемым процессом. Это делается для решения проблемы распределения соединений между процессами одной группы. В ходе исследования предметной области и реализации сервера с заданной архитектурой, было выяснено, что открытые файловые дескрипторы процесса не могут быть переданы посредством очередей сообщений. (при передаче открытого дескриптора возникает ошибка EBADF). Однако система с наследованием имеет недостаток в отсутствии возможности распределения соединений между процессами - соединение принимает тот, кто быстрее успел.

Конструкторский раздел

Сервер

Конечный автомат состояний сервера

Рис. 2 нагенерил самодельный *fsm2dot* из *autogen* и *dot2tex* на пару *dot*. Никто не мешает изменить параметры типа *rankdir* прямо в *fsm2dot*, если он будет лучше смотреться, например, сверху-вниз.

Синтаксис команд протокола

Ниже приведен формат команд сообщений протокола в виде регулярных выражений

1. **EHLO**: *EHLO [\w+]+*
2. **HELO**: *HELO [\w+]+*
3. **MAIL**: *MAIL FROM <[\w]+@[\w]+[\.\w]+>*
4. **RCPT**: *RCPT <[\w]+@[\w]+[\.\w]+>*
5. **DATA**: *DATA*
6. **NOOP**: *NOOP*
7. **RSET**: *RSET*
8. **QUIT**: *QUIT*

Представление данных

Ниже приведены диаграммы представления данных в системе - логическая и физическая.

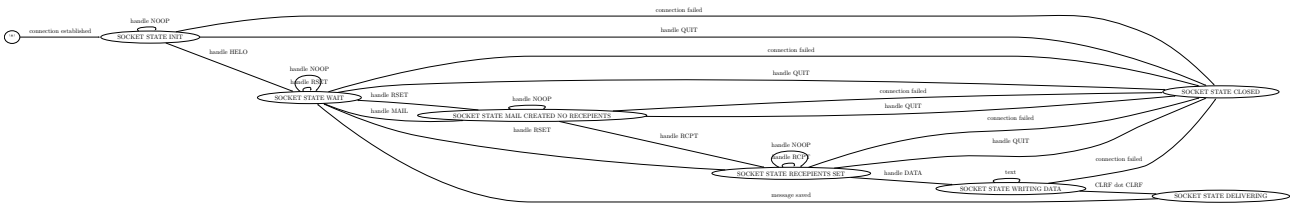

Рис. 2: Состояния сервера

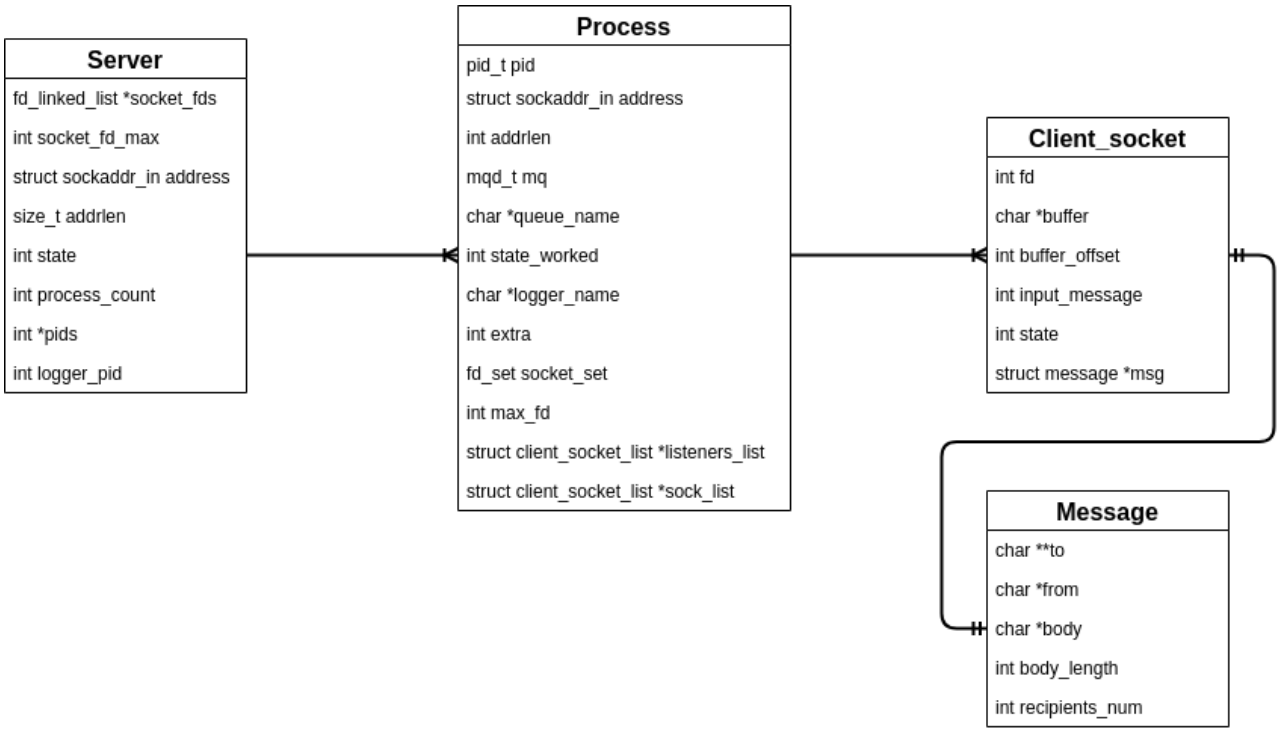

Рис. 3: Логическая диаграмма сущностей

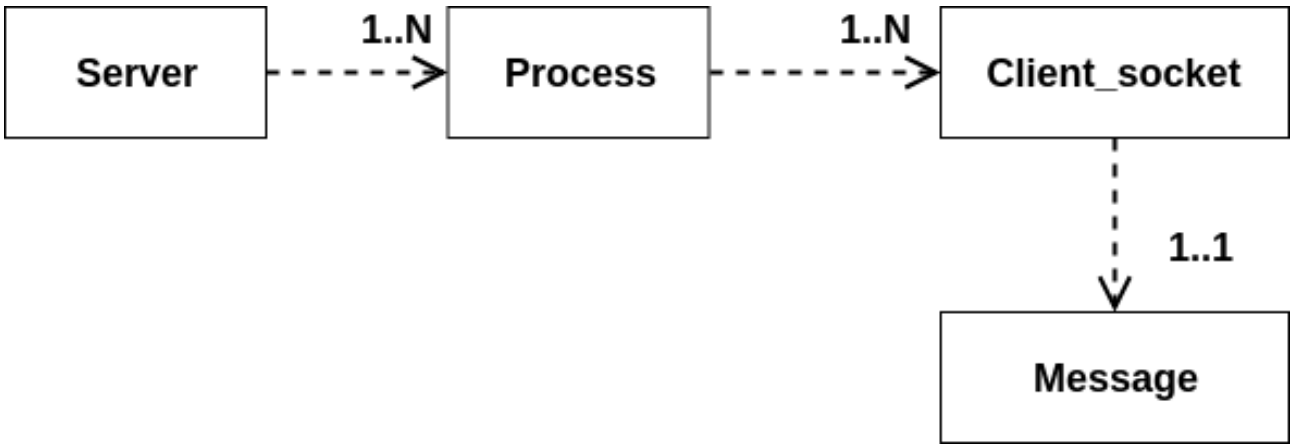

Рис. 4: Физическая диаграмма сущностей

Особенности реализации

- 1. Так как, согласно предложенному варианту задания на разработкаъу сервера, предлагается реализовать многопроцессную систему с журналированием в отдельном процессе, то имеет смысл объединить сущности логгера и обычного процесса с обязательной пометкой о том, что данный процесс - процесс журналирования;

- 2. Создание слушающих сокетов осуществляется через метод getaddrinfo() (стандарт gnu99). Этот метод позволяет получить все сокеты на конкретном адресе (в общем случае метод для локального хоста предоставляет одну структуру, описывающую сокет для адресов IPv4, одну - для адресов IPv6)

- 3. Обработка таймаутов для сокетов осуществляется через пометку всех клиентских сокетов, как закрытых. Это делается для освобождения ресурсов в ожидании новых соединений. Процессы при этом не закрываются, так как согласно выбранной архитектуре число процессов фиксированно - это сделано для минимизации числа вызовов fork(). Слушающие сокеты при этом также остаются открытыми, так как их число также ограничено.

- 4. Graceful closing осуществляется при помощи разделяемой памяти (очереди сообщений) и сигналов. В общем случае реализуются обработчики сигнала SIGINT для master-процесса и дочерних процессов. Для master-процесса реализуется ожидание закрытия всех дочерних процессов, после которого происходит освобождение системных ресурсов и удаление процесса, тогда как для дочерних процессов освобождение ресурсов осуществляется после получения сигнала SIGINT. Для родительского и дочерних процессов реализован схожий переход к освобождению ресурсов - через внутреннюю переменную, которая зануляется по сигналу.

- 5. В системе реализовано создание слушающих сокетов по числу сокетов (но в текущей версии приложения не используется, так как имеющихся сокетов при создании с помощью getaddrinfo() достаточно для функционального тестирования)

- 6. sleep() в родительском процессе обоснован тем, что в процессе разработки функциональность родительского процесса деградировала (см. Аналитическая часть), и для отсутствия захламеления журнала, родительский процесс с периодичностью сообщает, что жив. В случае расширения функциональности sleep() может быть убран.

- 7. В системе не реализовано использование конфигураций - зона ближайшего развития проекта.

- 8. Тестирование протокола осуществляется через мок-объекты с нулевым файловым дескриптором. Объект используется для непосредственного тестирования протокола, это позволяет гарантированно не зависать на моментах приема-передачи. Покрытие тестами неполное, так как это выходит за рамки проекта - вторая зона ближайшего развития проекта.

- 9. Генерация имен файлов сообщений осуществляется через структуру timeval - гарантирует относительно случайное имя.

– 10. Для родительского процесса аналогичным образом реализована система состояний. Но так как диаграмма состояний этого процесса малоинформативна из-за деградации функциональности, принято решение не включать ее в отчет.

– 11. Команда VRFY не обрабатывается, как отдельная команда - она обрабтывается так, если бы это был любой другой текст, используемый в качестве команды.

– 12. Команды должны посылаться на сервер строго как 4 символа в верхнем регистре - нет перевода между регистрами

Алгоритм обработки соединений в одном процессе

ПОКА (процесс$_{==1)}$
    Удалить из списка клиентских сокетов процесса сокеты с состоянием "SOCKET$_STATE_C$
    Обнулить сет читателей
    Добавить дескрипторы слушающих сокетов в сет читателей
    Добавить дескрипторы клиентских сокетов в сет читателей
    Добавить дескриптор очереди сообщений в сет читателей
    Ожидать соединения на одном из сокетов (время = 15с)
    ЕСЛИ (количество$_{==0)}$
        Обработать таймаут, переведя все клиентские сокеты в состояние "SOCKET$_STATE$
    ИНАЧЕ
        ЕСЛИ действие на очереди сообщений ТО
            Проверить запрос на graceful shutdown
            ЕСЛИ есть запрос ТО
                процесс$_{=0}$
        ДЛЯ каждого слушающего сокета
            ЕСЛИ действие на одном из слушающих сокетов ТО
                Принять новое соединение
                Инициализировать новый сокет
                Отправить приветствие
                Установить неблокирующий режим для принятого соединения
                Установить начальное состояние сессии (SOCKET$_STATE_INIT$)
            КОНЕЦ ЕСЛИ
        КОНЕЦ ДЛЯ
        ДЛЯ каждого клиентского сокета
            ЕСЛИ действие на одном из клиентских сокетов ТО
                Обработать действие в соотевтствии с протоколом
            КОНЕЦ ЕСЛИ
        КОНЕЦ ДЛЯ
    КОНЕЦ ЕСЛИ
КОНЕЦ ПОКА

Конечный автомат состояний сервера

Рис. 5 нагенерил самодельный *fsm2dot* из *autogen* и *dot2tex* на пару *dot*. Никто не мешает изменить параметры типа *rankdir* прямо в *fsm2dot*, если он будет лучше смотреться,

например, сверху-вниз.

Клиент

Принцип действия

Работу клиента можно разделить на 2 отдельные сущности: работа основного потока, работа рабочих потоков *(далее воркеры)* В основном потоке решено производить следующие действия

1. Создание при инициализации и удаление при завершении списка воркеров
2. Циклический поиск новых писем
3. Распределение писем по воркерам
4. Перехват сигналов на завершение работы

При запуске сервера, главный поток создает пул воркереров и помещает в него логгер и рабочие потоки. Количество рабочих потоков напрямую зависит от количества процессоров на устройстве. Однако, для устройств в которых процессоров меньше чем 3 создается обязательно 1 рабочих процесс Для *Изящного завершения* необходимо подписаться на перехват сигналов. Для более корректного и однозначного управления потоками программы необходимо, чтобы рабочие потоки не перехватывали общие сигналы с главным потоком. Для этого решено использовать 2 сигнала **SIGINT** для завершения программы – перехватывается главным потоком и **SIGUSR1** для управления воркерами. Во время корректного завершения главный поток оповещает всех воркеров по очереди в своем спике с помощью сигнала и ожидает их завершения. После завершения необходимо удалить данные воркера и освободить все ресурсы занимаемые главным потоком.

Рабочие процессы в данном случае выполняют роль связных. Их деятельность можно описать следующими действиями:

1. Контролирует список соединений
2. Получает и обрабатывает задания от главного потока
3. Отправляет письма

Воркер во время инициализации получает служебную информацию для поддержания связи с главным потоком. Так как в требованиях к заданию указывается, что соединение должно быть неблокируемым, каждый воркер обладает множествами пишущих, читающих и обработчиков прерываний. Помимо указаных множеств в данном блоке информации хранится список активных соединений (сокетов) Во время работы воркер ожидает на **pselect** готовых к работе соединений. Однако, перед этим необходимо распределить новые задания от главного потока. После обработки всех заданий воркер для каждого готового соединения передает управление SMTP контроллеру до тех пор, пока:

– Текущее состояние соединения - в процессе

– Текущее состояние соединения - заблокирован

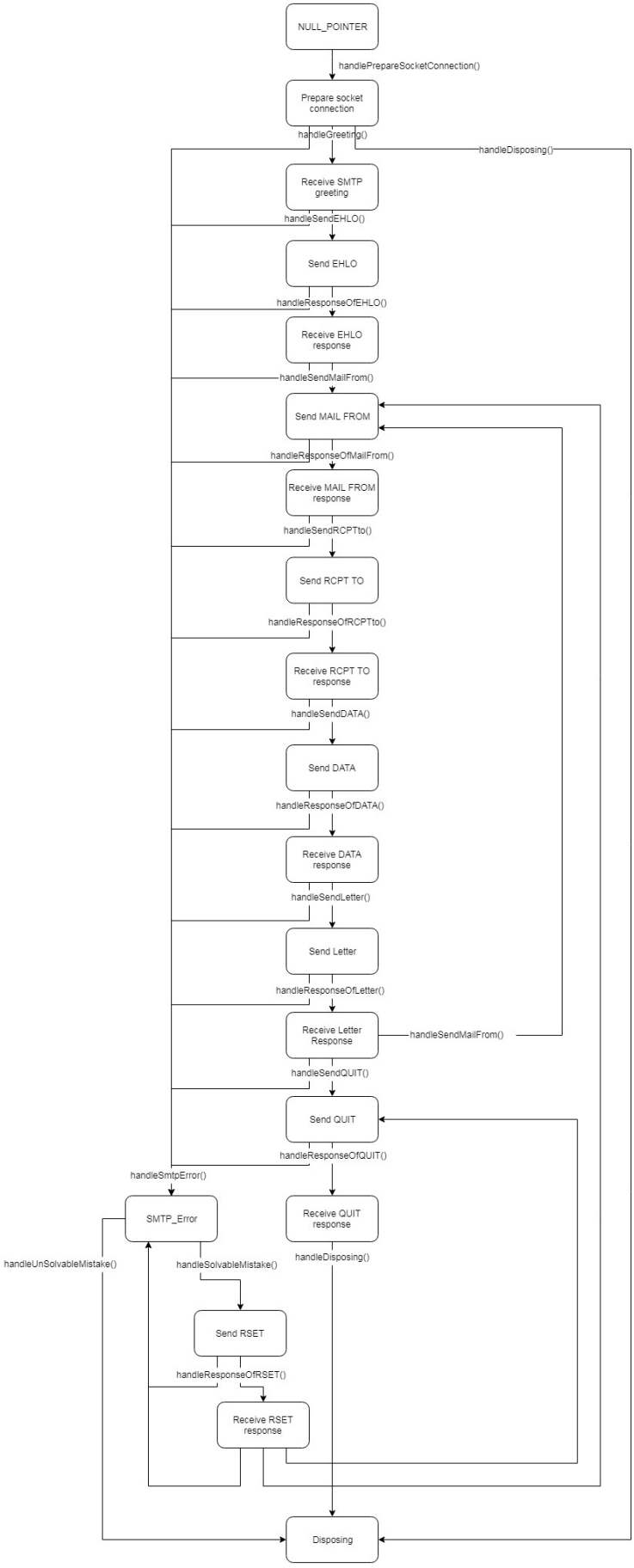

Рис. 5: Состояния сервера

– Дальнейших действий в SMTP контроллере по данному соединению произвести невозможно.

Если по текущемо соединению все задания на текущий момент были завершены, то активное соединение удаляется из списка. Стоит отметить что, для того чтобы новые заявки не накапливались во время ожидания активных соединений, решено использовать таймер. Таким образом, все новые заявки будут распределяться по своим соединениям даже когда они не готовы. Ранее, отмечалось, что воркер управляется главным потоком через сигнал. Он используется как для оповещения новых заданий, так и для завершения работы. Принцип *"Изящного завершения"* должен учитывать состояния соединений воркера. Поэтому заверешние воркера заключается в уменьшении текущих задач. Для всех соединений задания, которые необходимо будет выполнить(не текущее) удаляется из списка заданий, а само письмо переноситься в начальную директорию (/new) пользователя. В случае, если соединение еще не было корректно установлено (для данного соединения еще не было приветствия от SMTP сервера), тогда первое задание тоже удаляется, файл переноситься, а соединение закрывается и удаляется. Текущие же задания, если стадия "приветствия"прошла успешно обрабатываются до конца. После обработки всех оставшихся соединений, воркер освобождает занимаемые им ресурсы и сообщает о завершении работы.

Обработка новых писем

Для отправки писем, необходимо найти письма в maildir и отправить их воркерам на выполнение. Функцию поиска писем выполняет главный поток. После инициализации всех необходимых структур и данных главный поток проверяет maildir на наличие новых писем. Сначала формируется список пользователей SMTP клиента. После, в каждой директории из списка проверяется директория **new** на наличие новых писем. Если в данном каталоге есть файлы, необходимо выделить информацию о адресате письма. Адресат письма получается из поля **To** письма. На его основе формируется домен, ответственный за прием писем пользователей. Найденное письмо перемещается в директорию **tmp** и для него формируется заявка на отправку. В заявке указан ответственный домен и путь к письму. После производится поиск подходящего воркера и задание на оптравку делегируется. Когда список файлов на отправку освобождается главный поток ожидает 30 секунд и повторяет поиск заново. Для корректного распределения используется метрика загруженности воркеров. Она отображает сколько задач уже было делегировано воркеру.

Словарь

Для контроля распределения заявок между воркерами, а так же отправки писем через одну соединение, хоть это и не обязательно, решено использовать словарь. В данном словаре хранится информация о распределении доменов между воркерами. Во время поиска писем, главный поток узнает ответственный домен и смотрит свой словарь доменов. Если в нем нет воркера, который уже работает с данным доменом, то выбирается наимение загруженный воркер. В случае если есть воркер, то новое письмо делегируется ему. Рабочий процесс также взаимодействует с данным словарем. Когда список заданий для какого-либо определенного домена закончился, соединение переходит в режим "завершения"отправки писем. В данный момент и воркер удаляет запись в словаре для данного домена. Таким

образом, система старается(не гарантирует) отправить все письма адресованные одному домену через одно активное соединение.

Рабочий поток

Каждый рабочий поток обладает записью с необходимой информацией о себе. Данная информация передается ему при создании. С помощью общей с главным потоком блоком информации, воркер может получать новые задания узнавать о текущем состоянии работы программы. В данном блоке целесообразно хранить следующую информацию:

- ссылку на очередь новых заданий
- личный идентификатор (удобно для логгирования)
- состояние работы
- количество заданий

Доступ к очереди заданий рабочего процесса имею два потока: главный и сам рабочий поток. Для исключения коллизий, потери данных необходимо использовать объект синхронизации. В качестве такого объекта синхронизации является семафор. Для корректной работы, каждый воркер создает во время подготовки структуру, с помощью которой будет осуществляться контроль соединений и их состояний. Разграничение между состояниями соединений необходимо учитывать во время передачи данных. Для этого в данной структуре содержатся множества ожидающих на запись и ожидающих на чтение соединений.

Параметр состояния работы, указывает рабочему процессу на необходимость корректного и быстрого завершения работы. В связи с этим, рабочий процесс, узнав о необходимости завершения, убирает все задания, которые еще не выполнялись у каждого соединения. Также изменяется время ожидания активизации соединений. Это необходимость обусловлена неблокируемостью сокетов, из-за чего у соединения нет как такого *timeout* на получение или отправку. Стоит отметить, что у каждого соединения текущее задание остается для корректного завершения обмена информацией с серверами.

Каждое активное соединение обрабатывается с помощью SMTP контроллера

Активное соедиение

Каждое соедиенние должно быть однозначно определено. Для контроля соединений целесообразно описать соединение следующим образом

- идентификатор соединения
- домен сервера
- адрес сервера
- текущее состояние соединения
- предыдущее состояние соединения
- количество попыток отправки одного письма

– список писем

– данные текущего письма

Во время подготовки к соединению необходимо разрешить адрес сервера. Для этого для указанного домена необходимо запросить MX запись. Из этой записи получается доменное имя самого SMTP сервера, с которым необходимо обмениваться данными. Однако, для корректного обмена необходимо узнать его IP адрес в сети. Для этого полученной записи запрашивается список IP адресов и выбирается первый успешно обработанный. Для оптимальной обработки писем, целесообразно хранить в памяти только служебную информацию, которая необходима для передачи письма. Для этих целей выделена отдельный блок "данные письма". В данной блоке хранится адресат и адресант письма. Это позволяет сократить время на открытия файл и взаимодействия с ним во время установленного соединения

SMTP контроллер

SMTP контроллер отвечает за корректность общения с серверами. Для контроля обмена информацией данный контроллер ослеживает состояния общения и ведет их учет. Таким образом исключается возможность непредвиденных переходов, отправки некорректного сообщения, а так же исключается очередность отправки/получения сообщений между SMTP клиентом и SMTP сервером. Состояния соединений и переходы между ними отображены на Рис. 6

Стоит отметить, что во при переходе к завершению обмена данными по данному соединение именно SMTP контроллер удаляет запись из *словаря*. Это необходимо для исключения появления писем по закрывающемуся соединению. Таким образом, главный поток не будет делегировать письма воркеру, который завершает процесс передачи для данного домена и выберет наиболее свободного воркера. Отправка команд протокола SMTP сопровождается верификацией ответов от сервера. Каждый ответ сервера проверяется на соблюдение регламента передачи. Так, во время подключения к SMTP серверу и получения сообщения от сервера с доменным именем за который он отвечает, происходит верификация статуса ответа и самого домена. В случае нарушения соглашения: некорректный ответ или указанный в сообщение домен не совпадает с указанным в соединение, возникает ошибка, которая разрешается закрытием соединения и освобождению ресурсов.

Синтаксис поддерживаемых команд протокола

В данном разделе приведены поддерживаемые клиентом команды протокола SMTP.

– **EHLO**: *EHLO [w+]+*

– **MAIL**: *MAIL FROM: <[\w]+@[\w]+[\.\w]+>*

– **RCPT**: *RCPT To:<[\w]+@[\w]+[\.\w]+>*

– **DATA**: *DATA*

– **RSET**: *RSET*

– **QUIT**: *QUIT*

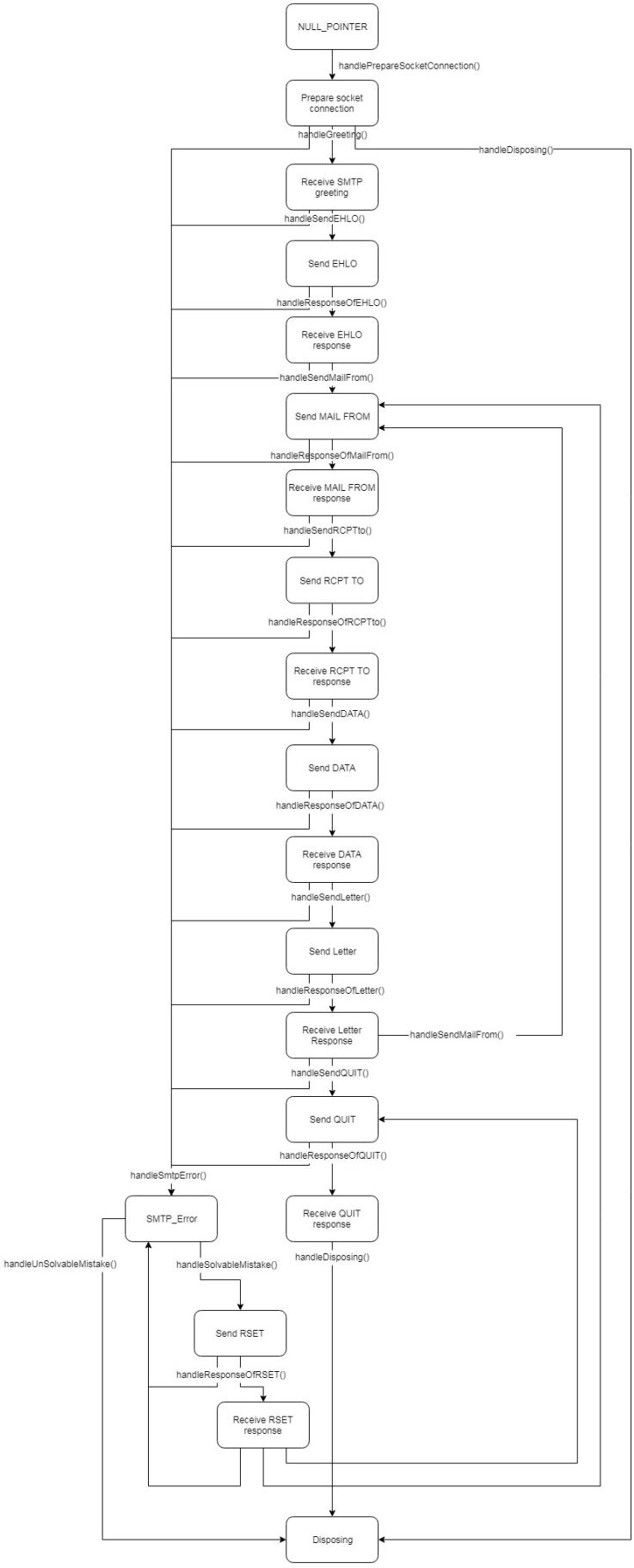

Рис. 6: Конечный автомат состояний соединений SMTP клиента

Технологический раздел

Сборка программы

Сборка программы, как для сервера, так и для клиента, осуществляется с помощью Makefile, объединяющего в себе несколько целей:

1. сборка исполняемого файла сервера/клиента (по умолчанию),
2. сборка модульных тестов,
3. генерация автомата состояний.

Генерация отчета по данному проекту осуществляется в единственном экземпляре и файл отчета содержит в себе данные как для сервера, так и для клиента. Для этой цели был создан отдельный Makefile с целями для генерации непосредственно отчета, а также включаемых данных, таких как, например, графы вызовов функций.

Сервер

Графы вызова функций

Поскольку функций много, графы вызовов разбиты на два рисунка. На рис. 9 показаны основные функции, на рис. 8 – функции обработки команд. Файл **cflow.ignore** содержит список функций (точнее, шабловнов поиска), использыемых программой *grep* для удаления малоинтересных стандартных функций[1].

Графы созданы с помощью *cflow*, *cflow2dot*, *dot*.

Клиент

Графы вызова функций

Основные данные

Стркутура рабочего процесса

```
struct worker
{
```

[1] Функции по работе с сокетами, ipc и привилегиями к малоинтересным ни в коем случае не относятся.

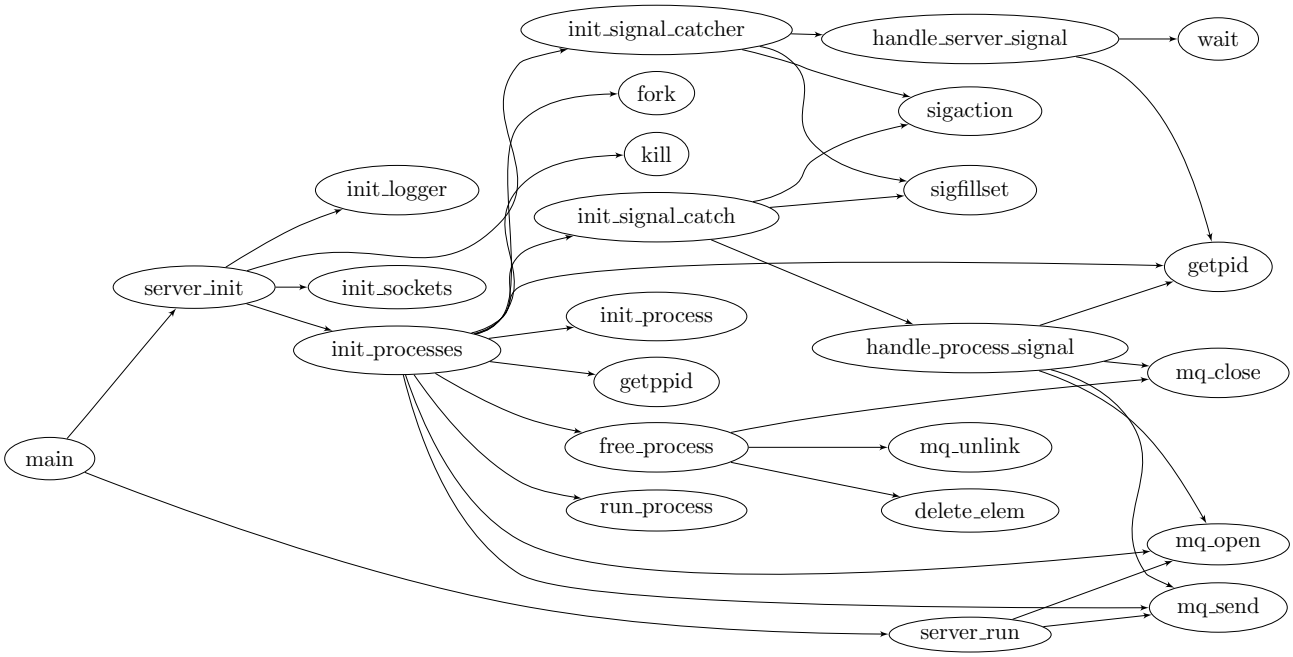

Рис. 7: Граф вызовов, функции обработки команд

```
    pthread_t thread;
    sem_t lock;
    struct worker_task *tasks;
    int workerId;
    int count_task;
    bool worked;
};
```

Структура соединения

```
struct FileDesc
{
    int id;
    char *domain;
    char *mx_record;
    struct sockaddr_in addr;
    int current_state;
    int prev_state;
    int attempt;
    struct worker_task *task_pool;
    struct letter_info meta_data;
};
```

Тестирование

Сервер

Модульные тесты создаются с помощью сценария unit_tests. Ниже приведён отчет о модульном тестировании (с использованием библиотеки CUnit):

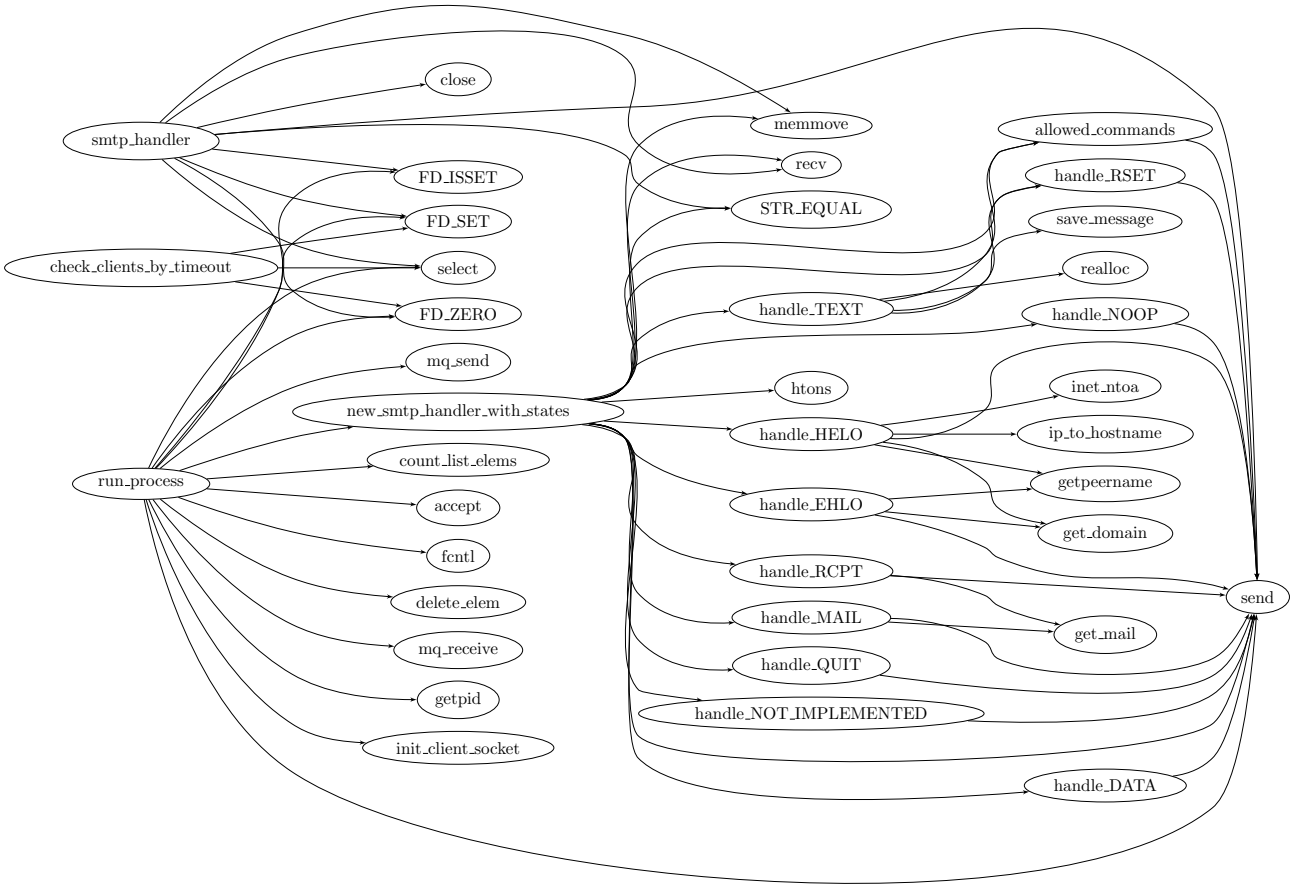

Рис. 8: Граф вызовов, функции обработки команд

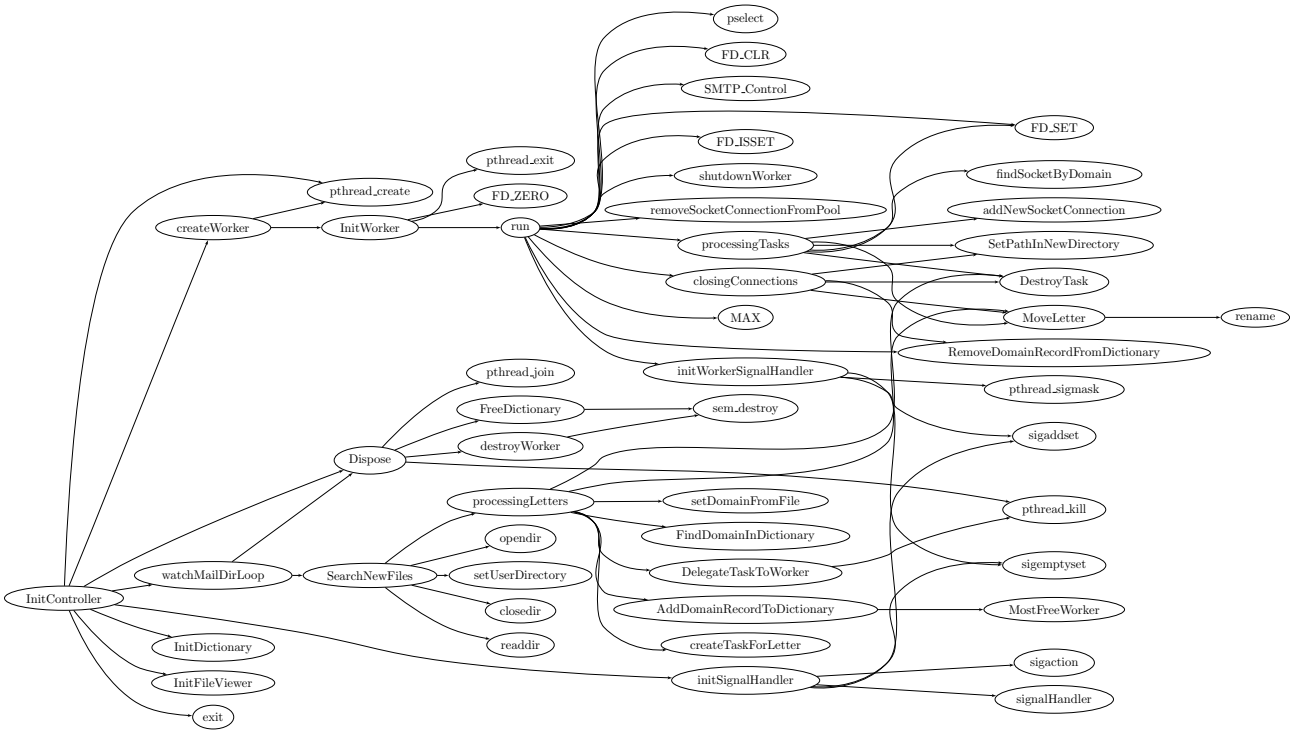

Рис. 9: Граф вызовов, основные функции

Сервер - модульные тесты

```
    CUnit - A unit testing framework for C - Version 2.1-2 \\
    http://cunit.sourceforge.net/ \\


Suite: suite_test \\
  Test: create_client_socket_no_message ...passed \\
  Test: create_client_socket_has_message ...passed \\
  Test: handle_helo_state_init ...Server: 0, HELO: 252 Ok \\
 myserver.com \\
passed \\
  Test: handle_helo_state_incorrect ...passed \\
  Test: handle_ehlo_state_correct ...Server: 0, EHLO: 250- myserver.com \\
passed \\
  Test: handle_ehlo_state_incorrect ...passed \\
  Test: handle_mail_correct ...Server: 0, MAIL: test@test.ru 250 Ok \\
passed \\
  Test: handle_mail_incorrect_command ...Server: 0, MAIL: 450 Requested mail action not
passed \\
  Test: handle_mail_incorrect_state ...passed \\
  Test: handle_rcpt_correct ...Server: 0, RCPT: 250 Ok \\
passed \\
  Test: handle_rcpt_incorrect_command ...Server: 0, MAIL: 450 Requested mail action not
passed \\
```

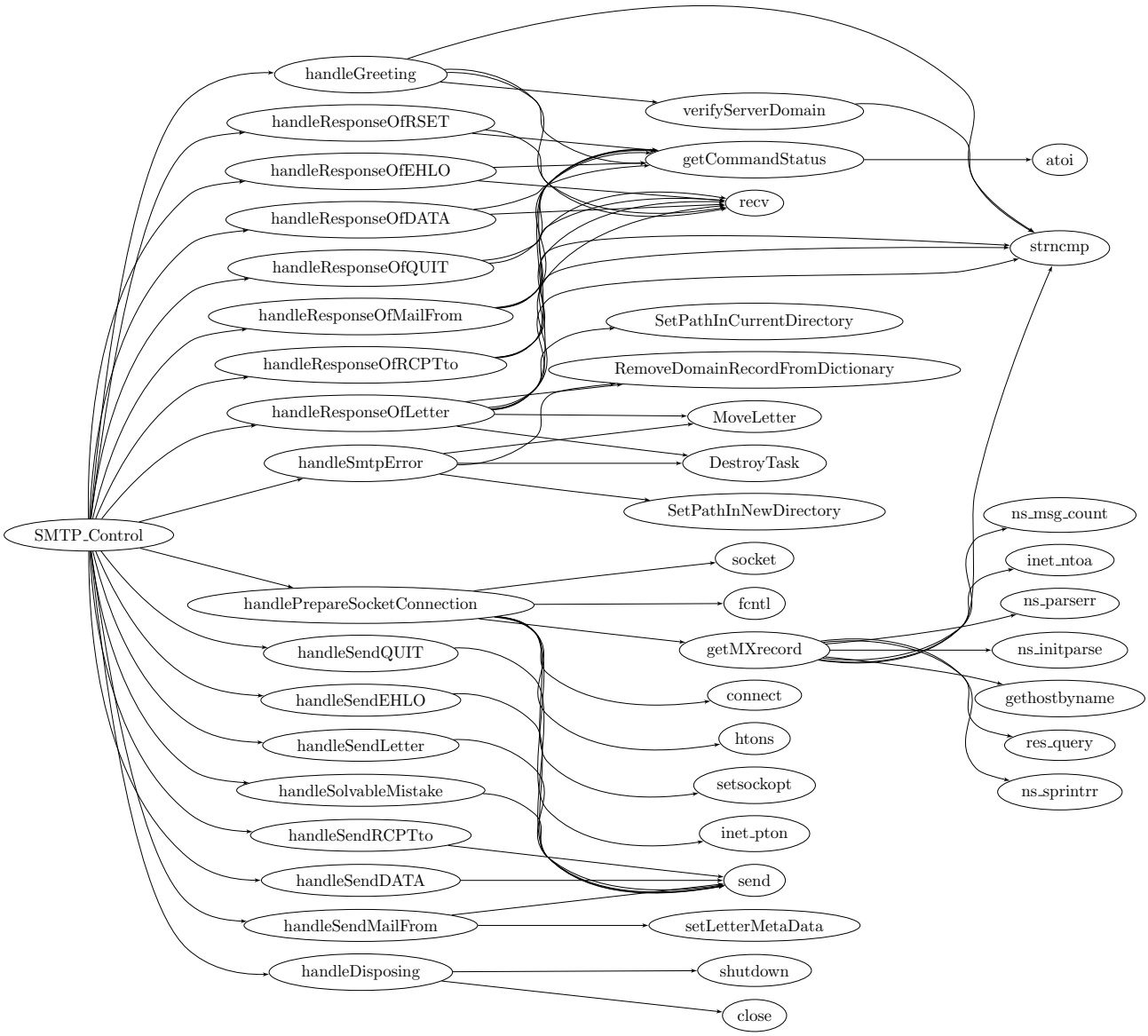

Рис. 10: Граф вызовов, взаимодействие с сервером

```
  Test: handle_rcpt_incorrect_state ...passed \\
  Test: handle_rcpt_exceeded_recepients ...Server: 0, RCPT: 250 Ok \\
Server: 0, RCPT: 250 Ok \\
Server: 0, RCPT: 250 Ok \\
Server: 0, RCPT: 250 Ok \\
Server: 0, RCPT: 250 Ok \\
Server: 0, RCPT: 250 Ok \\
Server: 0, RCPT: 250 Ok \\
Server: 0, RCPT: 250 Ok \\
Server: 0, RCPT: 250 Ok \\
Server: 0, RCPT: 250 Ok \\
Server: 0, RCPT: 451 Unable to complete command: number of recipients exceeded \\
passed \\
  Test: handle_rset_correct_no_recepients ...Server: 0, RSET: 250 Ok reset \\
passed \\
  Test: handle_rset_correct_has_recepients ...Server: 0, RSET: 250 Ok reset \\
passed \\
  Test: handle_rset_incorrect_state ...passed \\
  Test: test_prepare_client_before_getting_data_no_reset ...Server: 0, HELO: 252 Ok \\
 myserver.com \\
Server: 0, MAIL: test@test.ru 250 Ok \\
Server: 0, RCPT: 250 Ok \\
passed \\
  Test: handle_data_correct ...Server: 0, DATA: 354 Start mail input; end with <CRLF>.<C
passed \\
  Test: handle_data_incorrect_state ...passed \\
  Test: handle_quit_correct ...Server: 0, QUIT: 221 Ok \\
passed \\
  Test: handle_noop_correct ...Server: 0, NOOP: 250 Ok noop \\
passed \\
  Test: handle_not_implemented_correct ...Server: 0, NOT_IMPLEMENTED: 502 Command Not Im
passed \\
  Test: handle_text_not_dot ...Client: 0, In-MESSAGE: Test messsage passed \\
  Test: test_full_session_correct ...Server: 0, HELO: 252 Ok \\
 myserver.com \\
Server: 0, MAIL: test@test.ru 250 Ok \\
Server: 0, RCPT: 250 Ok \\
Server: 0, DATA: 354 Start mail input; end with <CRLF>.<CRLF> \\
Client: 0, In-MESSAGE: .Server: 0, MESSAGE: Server: 0, RSET: 250 Ok reset \\
Server: 0, QUIT: 221 Ok \\
passed \\
 \\
Run Summary:   \_ Type  \_Total \_   Ran Passed\_ Failed\_ Inactive \\
          \_    suites  \_    1   \_   1   \_ n/a    \_  0    \_    0 \\
          \_     tests  \_   24  \_   24    \_24     \_ 0    \_    0 \\
          \_   asserts  \_  126  \_  126  \_  126   \_   0  \_    n/a \\
```

```
Elapsed time =    0.003 seconds \\
```

Сервер - valgrind

Ниже приведен пример вывода для утилиты валгринд (число процессов = 2)

```
==6831== HEAP SUMMARY: \\
==6831==     in use at exit: 644 bytes in 4 blocks \\
==6831==   total heap usage: 106 allocs, 102 frees, 28,505 bytes allocated \\
==6831==  \\
==6831== Searching for pointers to 4 not-freed blocks \\
==6831== Checked 104,096 bytes \\
==6831==  \\
==6831== LEAK SUMMARY:\\
==6831==    definitely lost: 0 bytes in 0 blocks\\
==6831==    indirectly lost: 0 bytes in 0 blocks\\
==6831==      possibly lost: 0 bytes in 0 blocks\\
==6831==    still reachable: 644 bytes in 4 blocks\\
==6831==         suppressed: 0 bytes in 0 blocks\\
==6831== Reachable blocks (those to which a pointer was found) are not shown.\\
==6831== To see them, rerun with: --leak-check=full --show-leak-kinds=all\\
==6831== \\
==6831== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)\\
==6831== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)\\
\\
\\
==6830== HEAP SUMMARY:\\
==6830==     in use at exit: 40 bytes in 3 blocks\\
==6830==   total heap usage: 8 allocs, 5 frees, 1,060 bytes allocated\\
==6830== \\
==6830== Searching for pointers to 3 not-freed blocks\\
==6830== Checked 103,504 bytes\\
==6830== \\
==6830== LEAK SUMMARY:\\
==6830==    definitely lost: 0 bytes in 0 blocks\\
==6830==    indirectly lost: 0 bytes in 0 blocks\\
==6830==      possibly lost: 0 bytes in 0 blocks\\
==6830==    still reachable: 40 bytes in 3 blocks\\
==6830==         suppressed: 0 bytes in 0 blocks\\
==6830== Reachable blocks (those to which a pointer was found) are not shown.\\
==6830== To see them, rerun with: --leak-check=full --show-leak-kinds=all\\
==6830== \\
==6830== ERROR SUMMARY: 3 errors from 2 contexts (suppressed: 0 from 0)\\
\\
==6833== HEAP SUMMARY:\\
```

```
==6833==     in use at exit: 92 bytes in 5 blocks\\
==6833==   total heap usage: 12 allocs, 9 frees, 1,768 bytes allocated\\
==6833== \\
==6833== Searching for pointers to 5 not-freed blocks\\
==6833== Checked 103,592 bytes\\
==6833== LEAK SUMMARY:\\
==6833==    definitely lost: 52 bytes in 2 blocks\\
==6833==    indirectly lost: 0 bytes in 0 blocks\\
==6833==      possibly lost: 8 bytes in 1 blocks\\
==6833==    still reachable: 32 bytes in 2 blocks\\
==6833==         suppressed: 0 bytes in 0 blocks\\
==6833== Reachable blocks (those to which a pointer was found) are not shown.\\
==6833== To see them, rerun with: --leak-check=full --show-leak-kinds=all\\
==6833== \\
==6833== Use --track-origins=yes to see where uninitialised values come from\\
==6833== ERROR SUMMARY: 10 errors from 10 contexts (suppressed: 0 from 0)\\
\\
==6832== HEAP SUMMARY:\\
==6832==     in use at exit: 185 bytes in 7 blocks\\
==6832==   total heap usage: 33 allocs, 29 frees, 10,976 bytes allocated\\
==6832== \\
==6832== Searching for pointers to 8 not-freed blocks\\
==6832== Checked 103,592 bytes\\
\\
==6832== \\
==6832== LEAK SUMMARY:\\
==6832==    definitely lost: 132 bytes in 3 blocks\\
==6832==    indirectly lost: 13 bytes in 1 blocks\\
==6832==      possibly lost: 0 bytes in 0 blocks\\
==6832==    still reachable: 40 bytes in 3 blocks\\
==6832==         suppressed: 0 bytes in 0 blocks\\
==6832== Reachable blocks (those to which a pointer was found) are not shown.\\
==6832== To see them, rerun with: --leak-check=full --show-leak-kinds=all\\
==6832== \\
==6832== Use --track-origins=yes to see where uninitialised values come from\\
==6832== ERROR SUMMARY: 12 errors from 12 contexts (suppressed: 0 from 0)\\
==6832== \\

    CUnit - A unit testing framework for C - Version 2.1-3
     http://cunit.sourceforge.net/


    Suite: suite_test
    Test: domain_is_NULL ...passed
    Test: bad_domain ...passed
    Test: connection_is_blocked ...passed
    Test: greeting_bad_graph_state ...passed
```

```
Test: in_greeting_not_correct_domain ...passed
Test: send_ehlo_and_correct_result ...passed
Test: send_ehlo_and_change_state_on_error ...passed

Run Summary:    Type  Total    Ran Passed Failed Inactive
            suites      1      1    n/a      0        0
            tests       7      7      7      0        0
            asserts     24     24     24      0      n/a

Elapsed time =    0.004 seconds
```

Valgrind

```
==24216== LEAK SUMMARY:
==24216==    definitely lost: 326 bytes in 6 blocks
==24216==    indirectly lost: 1,056 bytes in 4 blocks
==24216==      possibly lost: 272 bytes in 1 blocks
==24216==    still reachable: 1,638 bytes in 4 blocks
==24216==         suppressed: 0 bytes in 0 blocks
==24216== Reachable blocks (those to which a pointer was found) are not shown.
==24216== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==24216==
==24216== For counts of detected and suppressed errors, rerun with: -v
==24216== Use --track-origins=yes to see where uninitialised values come from
==24216== ERROR SUMMARY: 33 errors from 21 contexts (suppressed: 0 from 0)
```

Пример работы относително сервер

```
11 Jan 2019 00:07:51 DEBUG org.subethamail.smtp.server.Session - Server: 220 samsung
11 Jan 2019 00:07:51 DEBUG org.subethamail.smtp.server.Session - Client: EHLO samsun
11 Jan 2019 00:07:51 DEBUG org.subethamail.smtp.server.Session - Server: 250-samsung
250-8BITMIME
250-AUTH LOGIN
250 Ok
11 Jan 2019 00:07:58 DEBUG org.subethamail.smtp.server.Session - Client: MAIL FROM:<
11 Jan 2019 00:07:58 DEBUG org.subethamail.smtp.server.Session - Server: 250 Ok
11 Jan 2019 00:07:58 DEBUG org.subethamail.smtp.server.Session - Client: RCPT To: <s
11 Jan 2019 00:07:58 DEBUG org.subethamail.smtp.server.Session - Server: 250 Ok
11 Jan 2019 00:08:01 DEBUG org.subethamail.smtp.server.Session - Client: DATA
11 Jan 2019 00:08:01 DEBUG org.subethamail.smtp.server.Session - Server: 354 End dat
11 Jan 2019 00:08:02 DEBUG org.subethamail.smtp.server.Session - Server: 250 Ok
11 Jan 2019 00:08:07 DEBUG org.subethamail.smtp.server.Session - Client: MAIL FROM:<
11 Jan 2019 00:08:07 DEBUG org.subethamail.smtp.server.Session - Server: 250 Ok
11 Jan 2019 00:08:07 DEBUG org.subethamail.smtp.server.Session - Client: RCPT To: <s
11 Jan 2019 00:08:07 DEBUG org.subethamail.smtp.server.Session - Server: 250 Ok
11 Jan 2019 00:08:08 DEBUG org.subethamail.smtp.server.Session - Client: DATA
11 Jan 2019 00:08:08 DEBUG org.subethamail.smtp.server.Session - Server: 354 End dat
```

```
11 Jan 2019 00:08:10 DEBUG org.subethamail.smtp.server.Session - Server: 250 Ok
11 Jan 2019 00:08:17 DEBUG org.subethamail.smtp.server.Session - Client: QUIT
11 Jan 2019 00:08:17 DEBUG org.subethamail.smtp.server.Session - Server: 221 Bye
```

Выводы

В рамках предложенной работы нами были реализованы два компонента современного SMTP-сервера, взаимодействующих между самой по самописному протоколу в соответствии со стандартами RFC. В ходе работы реализованы следующие задачи:

1. Сервер- Проанализировали архитектурное решение
2. Сервер- Разработали подход для обработки входящих соединений и хранения входящих писем в maildir
3. Сервер- Рассмотрели **SMTP**-протокол
4. Сервер- Реализовали программу для получения писем по протоколу **SMTP**
5. Клиент- Проанализировали архитектурное решение maildir
6. Клиент- Разработали подход для обработки писем в maildir
7. Клиент- Рассмотрели **SMTP**-протокол
8. Клиент- Проанализировали способы получения и обработки **MX**-записей
9. Клиент- Реализовали программу для отправки писем по протоколу **SMTP**
10. Общее- Рассмотрели работу с неблокирующими сокетами и их взаимодействие поверх протокола TCP
11. Общее- Разработали системы, работающие в многозадачном режиме, столкнувшись с проблемами взаимодействия потоков и процессов, которые были решены в ходе работы
12. Общее- Познакомились с утилитами автоматической сборки и тестирования
13. Общее- Разработали сценарии тестирования для модулей протокола обработки сообщений, организовали определенный процент покрытия
14. Общее- Познакомились с концепцией graceful shutdown, произвели ее реализацию в проекте.