Рассчётно-пояснительная записка по курсовой работе "Разработка почтового сервера"

Кисленко Максим Германович

20 января 2018 г.

Оглавление

Введение

Курсовая работа предполагает разработку собственного почтового сервера, который будет использоваться для пересылки электронных писем (email-ов) в IP-сети. Вся работа поделена на две приблизительно одинаковые части - разработка почтового сервера, осуществляющего приём писем, и разработка почтового клиента, который переотправит письма другим почтовым серверам в зависимости от получателей-адресатов. Эти части разрабатываются независимо друг от друга двумя студентами, а потом интегрируются в единую систему. Команде студентов дополнительно присваивается вариант, который определяет основные требования к реализации или функциональности.

При реализации подразумевается, что используются низкоуровневые системные вызовы и, что сервер работает под управлением POSIX-совместимой операционной системы. Последнее условие необходимо, чтобы обеспечить переносимость созданного ПО.

В данном документе будет рассмотрена разработка почтового сервера, с условиями от варианта 12. Условия включают в себя следующие пункты:

- сервер должен обрабатывать входящие соединения в единственном рабочем потоке с применением вызова poll. Такое условие подразумевает использование сокетов в режиме неблокирующего ввода-вывода, так как иначе, единственный поток бы блокировался на одном клиенте и не мог бы считаться полноценным сервером;
- логирование событий и ошибок в работе сервера должно происходить в отдельном процессе. Такое техническое решение увеличивает сложность реализации, но и делает сервер более производительным (отсутствуют операции ввода-вывода на жесткий диск);
- проверка обратной зоны dns не предполагается.

Независимо от условий варианта, при постановке задачи заданы требования к выполнению данной работы. А именно:

- в качестве протокола пересылки электронных сообщений предписано использовать протокол SMTP (simple mail transfer protocol). Вся необходимая информация по нему доступна в RFC 5321;
- для локального хранения писем использовать механизм Maildir;
- конфигурация сервера должна быть вынесена в отдельный конфигурационный файл;
- необходимо разработать автоматическое тестирование созданного ПО (системные, unit-тесты);

– необходимо автоматизировать сборку проекта из исходников.

В поставленной задаче довольно жестко регламентированы используемые технологии. В качестве основного языка программирования - язык Си стандарта 99-го года. Для автоматизации тестирования - cunit или скриптовые языки (python, ruby или другое). Для автоматизации сборки - только GNU Make. Также готовое ПО необходимо протестировать на наличие утечек памяти с помощью утилиты valgrind.

Готовая программа должна обрабатывать входящие соединения, обслуживать их в соответствии с правилами сессии SMTP, сохранять письма локально в Maildir вместе с необходимыми заголовками, и передавать письма на дальнейшую отправку SMTP-клиенту. Далее в этом документе детально рассмотрены:

– Пользовательское взаимодействие с сервером, его достоинства, недостатки и существующие аналоги в аналитическом разделе;

– Проектирование и принятые программные решения при создании ПО в конструкторском разделе;

– Используемые инструменты и технологии, порядок эксплуатации созданного ПО в технологическом разделе;

Глава 1

Аналитический раздел

1.1 Предметная область

В результате проведённого исследования были выделены следующие сущности предметной области (рисунок 1.1). Здесь и далее все диаграммы выполняются в нотациях UML, если иное не указано явно.

1.2 Достоинства и недостатки реализуемого сервера

Достоинства реализованного ПО проистекают из его архитектурных решений:

1. благодаря однопоточной реализации, в сервере нет потерь времени на переключение контекстов, ожидание блокировок и проблем с разделяемой памятью;

2. благодаря неблокирующему вводу-выводу, сервер может с очень высокой производительностью обслуживать запросы клиентов, при условии, что сама обработка занимает мало времени. Последнее обеспечивается архитектурой протокола SMTP, который изначально построен по принципу: "многих быстрых этапов";

3. благодаря логированию в отдельном процессе, сервер не тормозится на IO операциях с жестким диском. Основной процесс просто отправляет буфер в очередь и продолжает исполнение, а записью в файл занимается отдельный процесс. Такое решение хорошо и тем, что уменьшает связность и зависимость программных компонент.

Как и у любой системы, наряду с достоинствами есть и недостатки.

1. из-за однопоточности сервера затруднено решение вопросов с отказоустойчивостью, так как при появлении фатальной ошибки в основном процессе, весь сервер "падает"(с потерей всех данных о текущих клиентах) и перестаёт обслуживать клиентов;

2. также затруднено обеспечение масштабируемости при росте нагрузки. Единственное решение - поднимать несколько демонов-серверов на разных портах и производить балансировку, что очень накладно по расходуемым ресурсам.

Данные недостатки уменьшены в существующих решениях, к примеру в сервере Nginx, благодаря более сложной архитектуре. Она включает не только мастер-процесс для IO-операций на неблокирующих сокетах, но и процессы worker-ы.

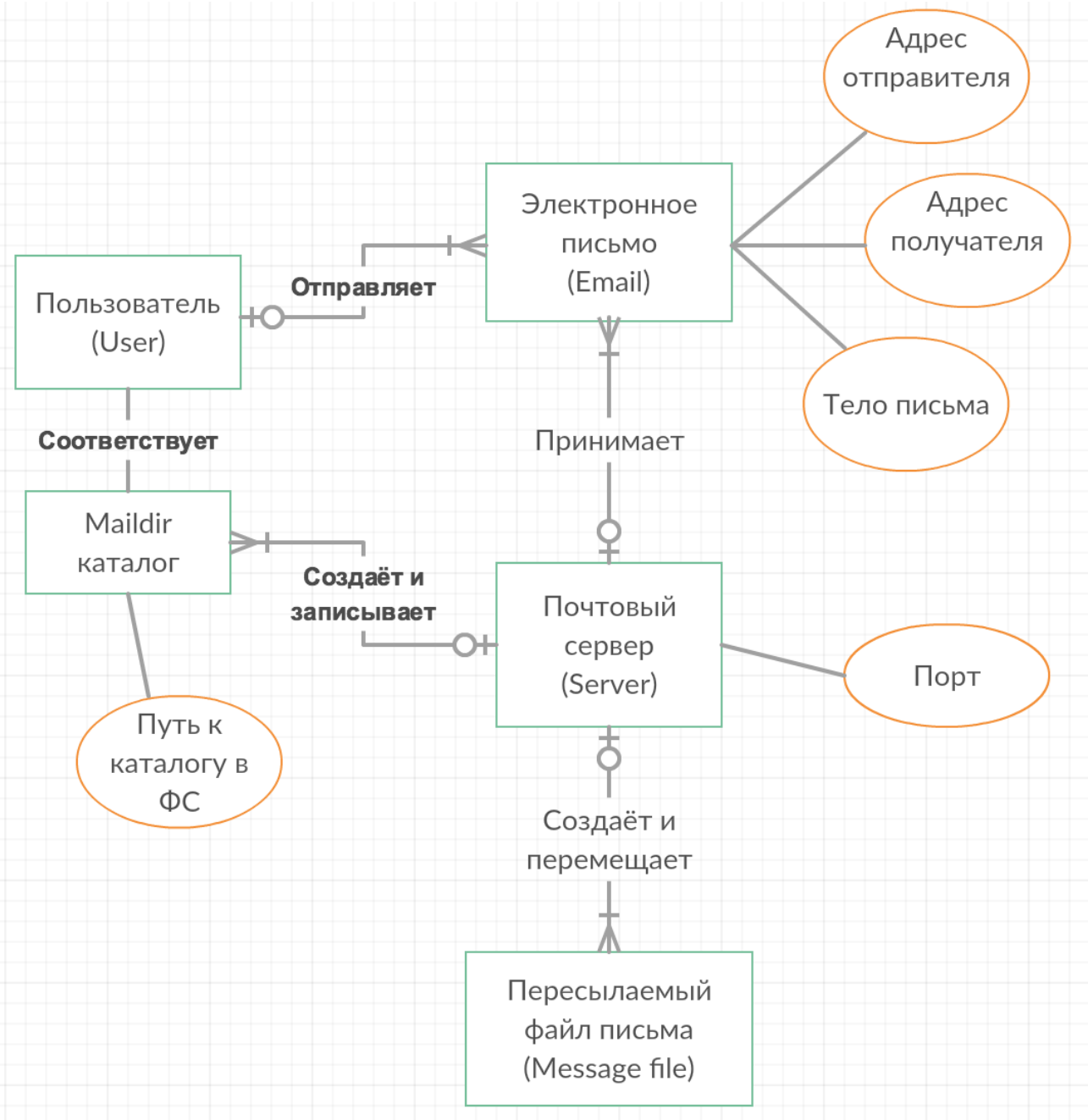

Рис. 1.1: Основные сущности предметной области

1.3 Программы-аналоги

Схожим функционалом обладают две широко-используемые программы: exim и postfix.

Глава 2

Конструкторский раздел

2.1 Архитектура сервера

Реализация почтового сервера разделена на несколько функциональных модулей с собственными заголовочными файлами. В server.h и server.c выполняется установка соединений с клиентами, получение от них текстовых команд и вызов SMTP-парсера для формирования ответа. Он также хранит для каждого из соединений запись о текущем состоянии (это связано с тем, что SMTP является stateful-протоколом). Сервер является однопроцессным, однопоточным и работает с сокетами клиентов в режиме неблокирующего ввода-вывода. Для получения событий о готовности ввода-вывода применяется системный вызов **poll()**. Подобная архитектура применяется в http-серверах nginx (с процессами-воркерами для обработки запросов) и node.js.

Опишем алгоритм работы сервера с помощью python-подобного псевдокода. Пусть **fd** будет основным сокетом, на котором сервер сделал вызовы **bind()** и **listen()**, тогда:

```
sockets = []
sockets.append({fd: fd, events: POLLIN})
active_cnt = 1

while True:
    code = poll(sockets, active_cnt, timeout)
    if code == 0:
        # timeout expired
        return 0

    quit = []
    for sock in sockets:
        if sock == fd:
            accept_new_conns(fd)
        else:
            state = get_state(sock)
            code = handle_known_conn(state)
            if code < 0:
                # received QUIT command
                quit.append(sock)
```

```
    if quit:
        for sock in quit:
            sockets.remove(sock)
```

Здесь и далее в листингах не показываются обработки ошибок, только основные действия. Чтобы принять новые соединения сервер выполняет внутри функции **accept_new_conns** системный вызов **accept**.

```
while True:
    new_sock = accept(fd)
    if new_sock < 0:
        return new_sock
    else:
        create_state(new_sock)
        send(new_sock, 'Welcome!')

        sockets.append(new_sock)
        active_cnt += 1
```

Если поступают данные от клиента, то сервер вызовет **handle_new_conn**, предварительно восстановив состояние этого клиента по сокету. Для получения данных используется вызов **recv**. Для анализа полученных от клиента данных функция SMTP-парсера **handle_request**.

```
input = ''

while True:
    chunk = recv(sock, input, 1000)
    if chunk:
        input += chunk
    else:
        break

if input:
    resp = handle_request(input)
    send(sock, resp)
```

2.2 Обработка команд протокола

Далее SMTP-парсер, анализирует полученную строку и в зависимости от неё и текущего состояния конкретного клиента выбирает выполняемое действие, новое состояние и нужный ответ клиенту. Парсер реализован в отдельном модуле из файлов smtp.h и smtp.c. По сути он представляет собой конечный автомат и может быть описан диаграммой состояний переходов (рисунок 2.1).

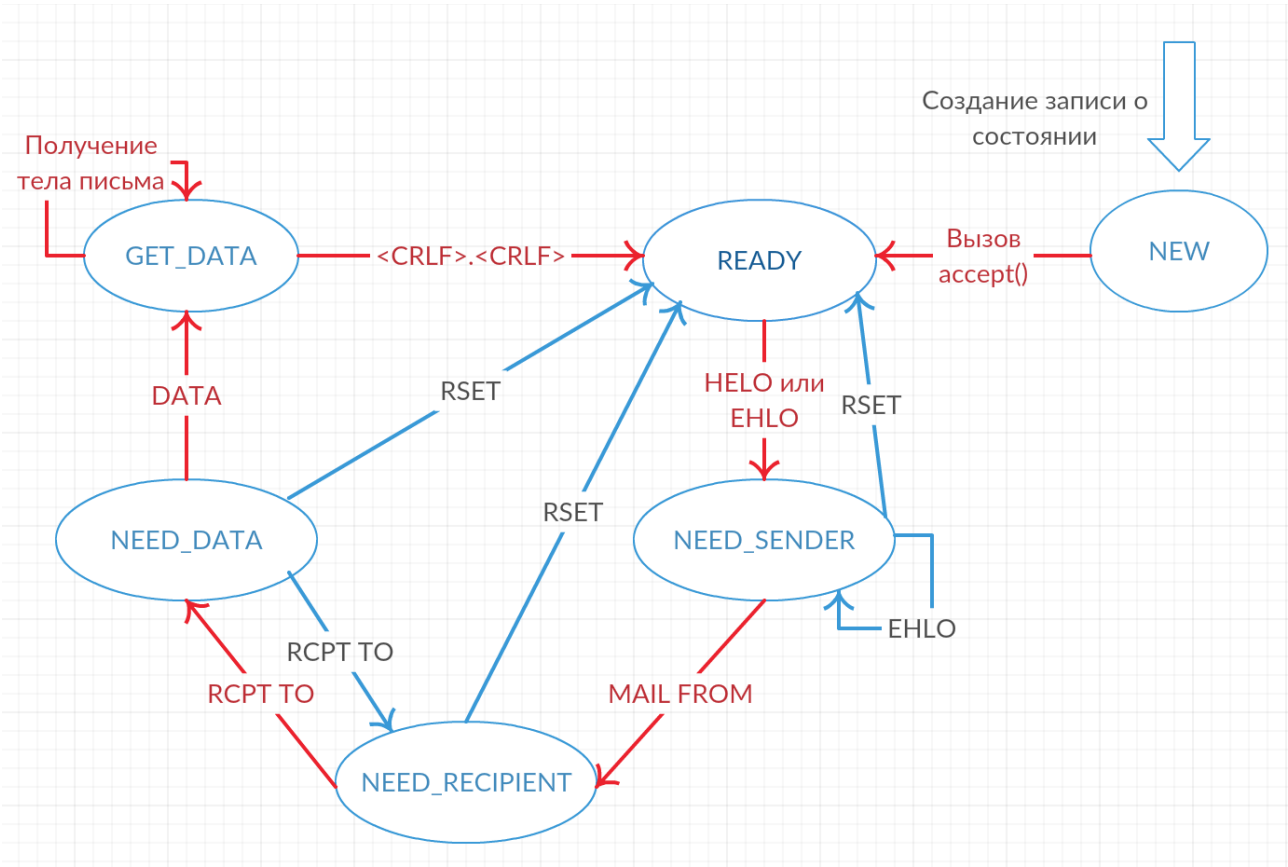

Рис. 2.1: Конечный автомат протокола SMTP

На диаграмме красным цветом выделен путь по графу, который соответствует обычной отправке письма с одним получателем. Синим - другие возможные переходы. На диаграмме также не показано несколько команд, которые не меняют состояния - это QUIT (команда серверу о закрытии соединения), NOOP (команда ничего не делать - no operation) и VRFY (доступна для вызова в состояниях READY и NEED_SENDER).

Некоторые из команд используют параметры. Рассмотрим их подробнее:

- **HELO domain** и **EHLO domain**, где domain определяется с помощью регулярного выражения **([a-zA-Z0-9]([a-zA-Z0-9]0,61[a-zA-Z0-9])?)+[a-zA-Z]2,6)**. Команды инициируют smtp-сессию. В ответ на EHLO ожидается развернутый ответ со списком поддерживаемых расширений;
- **MAIL FROM: <email>** и **RCPT TO: <email>**, где email определяется с помощью регулярного выражения **([-A-z0-9.]+@([A-z0-9][-A-z0-9]+)+[A-z]2,4)**. Команды задают отправителя и получателя. Получателей может быть несколько.
- **VRFY email_part**, где email_part определяется по **([A-z0-9.@-_]+)**. Команда возвращает информацию по известным для сервера пользователям в формате <полное имя> <email>, в email-е которых встретилась передаваемая параметром последовательность символов.

На команды клиента сервер отвечает предопределенными в спецификации SMTP кодами. Далее приведем список поддерживаемых команд.

- 221 Closing transmission channel - закрытие соединения по инициативе сервера;
- 250 OK - команда корректна и принята к исполнению;
- 354 Start mail input; end with <CRLF>.<CRLF> - ответ при переходе в состояние GET_DATA;
- 451 Requested action aborted: error in processing - ошибки при выполнении дисковых операций с maildir;
- 455 Server unable to accommodate parameters - возвращается в случае, когда получателей больше максимума;
- 500 Invalid command - неизвестная команда SMTP;
- 501 Invalid argument - отсутствие или некорректный аргумент в команде;
- 503 Bad sequence of commands - неожиданная команда для текущего состояния;
- 550 Sender unknown - email отправителя не является известным для smtp-сервера (известные записаны в файле usersinfo.txt);
- 552 Requested mail action aborted: exceeded storage allocation - ошибка выделения памяти, например для тела письма;

2.3 Работа с Maildir

Maildir - распространенный формат хранения электронной почты, который обеспечивает целостность передаваемых писем и самих почтовых ящиков за счет того, что операции по блокировке и перемещению отданы локальной файловой системе. Каждое письмо в maildir представлено отдельным файлом с уникальным именем (в работе используется UUID), который при получении записывается в подкаталог tmp, затем по окончанию записи помещается в new, где его найдет почтовый клиент. Схема Maildir представлена на рисунке 2.2.

Отдельный каталог Maildir создаётся для каждого получателя. Если получателей у одного письма несколько, то запишется несколько файлов с одинаковым содержимым для разных пользователей. Формат файла включает помимо тела письма ещё и SMTP-заголовки, а также дополнительные поля Subject (берется первая строка тела) и Date (дата получения письма сервером).

2.4 Логирование в отдельном процессе

Процесс, в котором происходит логирование, стартует из основного по вызову **fork()**. Соответственно, он является дочерним и разделяет с родителем обработчики прерываний (к примеру, SIGINT, послылаемый при нажатии Ctrl-C в терминале) и глобальную переменную-указатель на прочитанную конфигурацию.

Взаимодействие между процессами организовано с помощью System V IPC. Этот механизм более древный, чем Posix MQ и более распространен среди операционных систем (к примеру, posix mq не поддерживается OSX и MacOS). Реализация логирования для сервера расположена в модуле с файлами logger.h и logger.c.

```
#include <sys/ipc.h>
#include <sys/msg.h>

key_t key = ftok("/tmp", 'S');
int msg_queue = msgget(key, 0644 | IPC_CREAT);
FILE *log_file = fopen(log_file_name, "w");

char buffer[512];
while (1) {
    int res_code = msgrcv(msg_queue, &buffer, sizeof(buffer), 0, 0);
    if (strcmp(buffer, "Stop") == 0) {
        break;
    }

    char now[40];
    formatted_now(now, 40);
    fprintf(log_file, "[%s] %s\n", now, buffer);
    fflush(log_file);
}
```

Для того чтобы отправить сообщение в лог, основной процесс должен сначала подключиться к очереди.

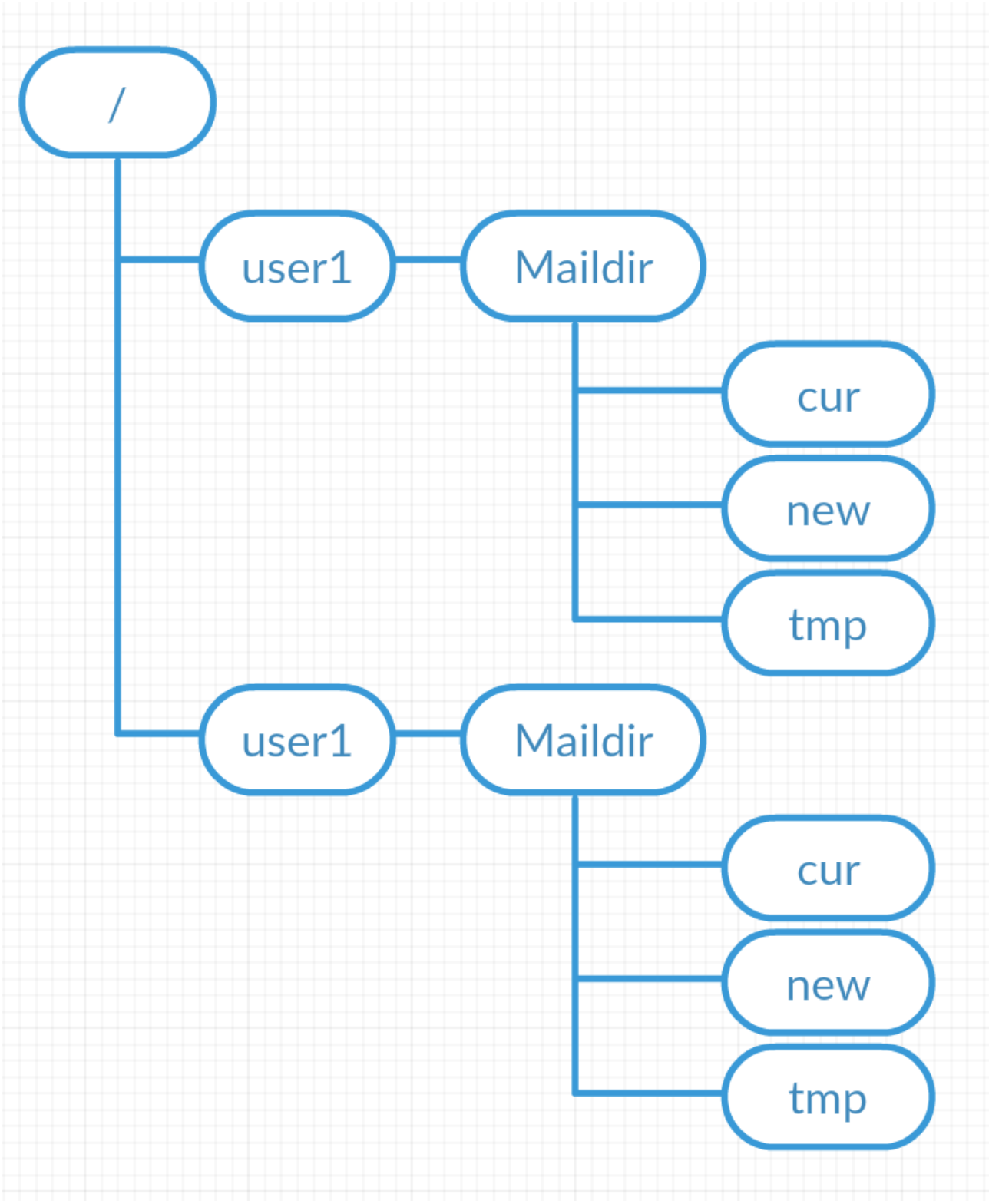

Рис. 2.2: Организация maildir

```
key_t key = ftok("/tmp", 'S');
int msg_queue = msgget(key, 0644 | IPC_CREAT);
int res_code = msgsnd(msg_queue, msg, strlen(msg), 0);
```

Процесс логирования останавливается по инициативе основного при посылке строки "Stop". Это используется, если сервер завершил выполнение по таймауту.

Глава 3

Технологический раздел

3.1 Используемые технологии и аппаратура

Разработанное ПО разрабатывалось и тестировалось на компьютере Macbook PRO модели 2015 года с процессором Intel Core i5, 8 GB оперативной памяти и 128 GB SSD, под управлением операционной системы MacOS High Sierra 10.13.

Для того чтобы собрать исполнемый файл из исходных текстов необходимо наличие в системе GNU make (>= 3.81) и компилятора языка си (>= llvm 9.0, если установлен clang и >= 4.2.1, если используется gcc). В проекте также приложена конфигурация gmake (файл smtp_server.pro), который использовался для быстрого запуска при разработке. Запущенный сервер пишет в консоль идентификатор собственного процесса, процесса логирования и строку "Server started". Подробнее про сборку в следующей секции.

Для чтения файла конфигурации сервер использует библиотеку libconfig версии >= 1.7.2, которая также должна быть установлена в системе для корректного запуска. Путь к заголовочным файлам библиотеки нобходимо явно указать в Makefile в переменной **INCPATH**. Значение по умолчанию - /usr/local/Cellar.

Чтобы убедиться в корректной работе собранного сервера, можно запустить системные тесты. Они написаны на языке python и требуют версию интерпретатора >= 3.4. Для запуска достаточно просто выполнить **python3 as user.py host port**, заменив параметры командной строки на те хост и порт, на которых запущен сервер. Подробнее про тестирование ниже.

3.2 Сборка программы

Сборка осуществляется с помощью утилиты make. Конфигурационный файл с описанием целей приложен в каталоге проекта /build. Для его исполнения достаточно команды **make all**.

Помимо создания исполняемого файла, Makefile прикладывает файлы конфигурации. Для очистки предусмотрена цель **make clean**.

3.3 Конфигурация в отдельном файле

Пример файла конфигурации для почтового сервера в conf/settings.cfg. Содержимое поделено на три секции - настройки для собственно сервера, smtp-парсера и для процесса

логирования.

```
version = "1.0";

server = {
    port = 9091;
    timeout_sec = 200;
    max_clients = 100;
};

smtp = {
    name = "smtp.maxim.ru";
    maildir = "/Users/maksimkislenko/smtp_env/mails";
    userinfo = "/Users/maksimkislenko/smtp_env/smtp_server/conf/userinfo.txt";
};

logger = {
    path = "/Users/maksimkislenko/smtp_env/log.txt";
};
```

Чтение и разбор файла конфигурации осуществляет библиотека libconfig при запуске основного процесса. По умолчанию ищется файл settings.cfg в текущей директории. Однако можно поменять это поведение передав параметром командной строки путь к собственным настройкам. Также есть возможность выводить доп. информацию в терминал - для этого при запуске достаточно указать –**verbose**.

3.4 Автоматические тесты

Тесты сгруппированы в сценарии. Каждый состоит из одного соединения и нескольких посылаемых команд. Полученные ответы сравниваются с ожидаемыми (ищутся вхождения ключевых слов и необходимый smtp-код). Если хоть одно из вхождений не найдено, будет сгенерировано исключение с информативным сообщением.

Код сценариев приведен в следующем листинге:

```
def script_1():
    print '=== Script 1 started (Test connection - HELO, QUIT, then write) ==='

    with SMTP_Client(HOST, PORT) as client:
        test_smtp_output('hello', client.hello(), ['250', 'ok'])
        test_smtp_output('quit', client.quit(), ['221', 'closing'])
        try:
            client.hello()
        except Exception as e:
            test_smtp_output('after close', str(e), ['errno 54', 'connection reset'])
    print ''
```

```
def script_2():
    print '=== Script 2 started (Test EHLO, RSET, NOOP commands) ==='

    with SMTP_Client(HOST, PORT) as client:
        test_smtp_output('1st ehlo', client.hello_extended(), ['250-', 'pipelining', '8b
        test_smtp_output('1st noop', client.no_operation(), ['250', 'ok'])
        test_smtp_output('rset', client.reset(), ['250', 'ok'])
        test_smtp_output('hello', client.hello(), ['250', 'ok'])
        test_smtp_output('2nd noop', client.no_operation(), ['250', 'ok'])
        test_smtp_output('2nd ehlo', client.hello_extended(), ['250-', 'pipelining', '8b
        test_smtp_output('quit', client.quit(), ['221', 'closing'])
    print ''

def script_3():
    print '=== Script 3 started === (Test ordinal mailing)'

    with SMTP_Client(HOST, PORT) as client:
        test_smtp_output('hello', client.hello(), ['250', 'ok'])
        test_smtp_output('sender', client.mail_from(), ['250', 'ok'])
        test_smtp_output('receiver', client.recipient_to(), ['250', 'ok'])
        test_smtp_output('data', client.data(), ['250', 'ok'])
        test_smtp_output('quit', client.quit(), ['221', 'closing'])
    print ''

def script_4():
    print '=== Script 4 started === (Test many recipients)'

    with SMTP_Client(HOST, PORT) as client:
        test_smtp_output('hello', client.hello(), ['250', 'ok'])
        test_smtp_output('sender', client.mail_from(), ['250', 'ok'])

        test_smtp_output('receiver', client.recipient_to(email='<test1@mail.ru>'), ['250
        test_smtp_output('receiver', client.recipient_to(email='<test2@mail.ru>'), ['250
        test_smtp_output('receiver', client.recipient_to(email='<test3@mail.ru>'), ['250

        test_smtp_output('data', client.data(), ['250', 'ok'])
        test_smtp_output('quit', client.quit(), ['221', 'closing'])
    print ''

def script_5():
    print '=== Script 5 started === (Errors 5xx)'

    with SMTP_Client(HOST, PORT) as client:
        test_smtp_output('mail_from before hello', client.mail_from(), ['503', 'bad', 's
        test_smtp_output('invalid hello', client.hello(cmd='HELLO'), ['500', 'invalid',
        test_smtp_output('hello', client.hello(), ['250', 'ok'])
```

```
        test_smtp_output('invalid sender argument', client.mail_from(email='test'), ['50
        test_smtp_output('unknown sender', client.mail_from(email='<test@mail.ru>'), ['5
        test_smtp_output('sender', client.mail_from(), ['250', 'ok'])
        test_smtp_output('hello after mail_from', client.hello(), ['503', 'bad', 'sequen

        test_smtp_output('no receiver', client.recipient_to(email=''), ['501', 'invalid
        test_smtp_output('quit', client.quit(), ['221', 'closing'])
    print ''

def script_6():
    print '=== Script 6 started === (Verify command)'

    with SMTP_Client(HOST, PORT) as client:
        test_smtp_output('hello', client.hello(), ['250', 'ok'])
        test_smtp_output('verify known', client.verify(), ['250', 'chesalin', 'denis',
        test_smtp_output('verify unknown', client.verify(email_part='test@mail.ru'), ['5
        test_smtp_output('quit', client.quit(), ['221', 'closing'])
    print ''
```

В результате запуска успешно-пройденные тесты отмечаются плюсами. Ошибки отмечаются минусами.

3.5 Проверка на наличие утечек памяти

Для проверки использовалась утилита Valgrind версии 3.13.0 с ручными изменениями в файле ./configure, необходимыми для успешной компиляции утилиты из исходников.

Результаты запуска команды **valgrind –leak-check=yes ./smtp_server**:

...

Выводы

В результате выполнения курсового проекта достугнута поставленная цель, а именно разработано приложение smtp-сервера, обеспечивающего локальную доставку и добавление в очередь удаленной доставки.

В ходе выполнения работы были получены и закреплены следующие навыки:

- проектирование и реализация сетевого протокола по имеющейся спецификации;
- реализация smtp-сервера на языке программирования Си;
- автоматизированное системное тестирование ПО сетевой службы;
- тестирования утечек памяти;
- создание сценариев сборки ПО;
- использование latex и сценариев сборки для автогенерации расчетно-пояснительной записки.

Из-за ограниченного времени и ресурсов в данной работе не были реализованы следующие важные пункты:

- проверка обратной зоны DNS, как базовый механизм от спам-рассылок (в принципе это не требовалось по условиям варианта);
- постановка сообщений в очередь почтового клиента для удаленной рассылки (из-за отсутствия команды);
- генерация документации утилитой doxygen.

Приложение 1. Основные функции программы

Здесь хотелось бы документацию doxygen

Приложение 2. Графы вызова функций

Здесь хотелось бы результат работы cflow