SMTP сервер№1

(Агеев А. В.)

24 декабря 2020 г.

Оглавление

Введение

Для функционирования компьютерных сетей, на оборудовании устанавливается программное обеспечение реализующий различные протоколы взаимодействия. Протоколы различаются по назначению. В данное время для обеспечения сети интернет используется стек протоколов TCP/IP, который состоит из протоколов выполняющий каждый свою задачу:

- Канальный уровень (например Ethernet) – беспечивают отправку и прием данных данных через среду передачи.
- Сетевой уровень (ip) – Канальный уровень работает с множеством устройств, которые объединены в одну группу (сеть). В данной группе устройства «видят» друг друга напрямую. Протоколы сетевого уровня предназанчены для обеспечения взаимодействия устройст из разных групп. Две сети объединяются маршрутизатором, а с помощью протокола сетевого уровня выплняется адресация устройст. В этом случае, между устройствами разных групп существует посредник – маршрутизатор
- Транспортный уровень (TCP, UDP) – на современном оборудовании работает множество программ, для определения того, какой программе адресованы прешедшые данные из сети, используются протоклы транспротного уровня. Их основная цель – адресация процессов на устройстве.
- Прикладной уровень – данные протоколы реализуются приложениями, которую выполняют некоторую задачу.

Целью курсовой работы является реализация протокола прикладоного уровня для получения и доставки электронной почты – Simple Mail Transfer Protocol (SMTP). А именно, части, которая выполняет прием почты и выполняет ее передачу на следующий этап – отправку почты.

Вариант 1 предпологает многопоточную реализацию сервера.

Глава 1

Аналитический раздел

1.1 Основные понятия протокола SMTP

SMTP протокол основан на клиент-серверной архитектуре. В данном случае клиентом выступает программа, которая хочет отправить почту, а сервером является программа для приема почты. Протокол поддреживает маршрутизацию почты, то есть серверу может придти письмо, которое адресовано клиенту на другом сервере. В этом случае серверное программное обеспечение принимает роль клиента и отправлет почту другому серверу.

Протокол состоит из текстовых сообщений, которые передают друг другу клиент и сервер при взаимодействии. Каждое сообщение прдеставляет из себя команду с параметрами, которые выполняются сервером. На какждую команду серверв выдает отклик. При организации надежного соединения (например посредством протокола TCP) клиент инициирует почтову транзакцию, которая состоит из последовательности команд, задающих отправителя и получателя сообщения, а так же передается содержательная часть письма. После чего клиент может завершить сеанс или начать новую почтовую транзакцию для передачи очередного письма.

Объекты электронной почты: ({Конверт; Содержимое})

- Конверт
  - Адрес отправителя – определяется командой MAIL FROM, которая так же начинает почтовую транзакцию.
  - Адрес получателей - с помощью команды RCPT TO определяется один получатель и маршрут почты до этого получателя (в RFC2821 указано, что лучше механизм маршрутизации почты игнорировать). Данная команда может быть передана несоклько раз для указания списка получателей одного письма.
  - Дополнительные заголовки. Протокол SMTP поддреживает расширения - добавление новых заголовков и параметров к стандартным заголовкам.

- Содержимое – передается после отправки команды DATA
  - Заголовок - список полей вида <ключ>:<значение>, спецификация которых описана в RFC5322
  - Тело сообщения - это непосредственное содержимое письма, которая представляет из себя текстовый набор данных соответсвущий спецификации форматов разны типов объектов MIME (Multipurpose Internet Mail Extensions)

Все элементы описваются с исопльзованием 7-битной кодировки US-ASCII, но это ограничение может быть снято с использованием расширения протокола 8BITMIE

Получатель и отправитель:

Протокол SMTP работает в 2 стороны. Получателем и отправителем может выступать как почтовая служба на сервере так и клиентское программное обеспечение. В протоколе выделяются следующие понятия:

- Клиент – Отправлющая сторона в текущей почтовой транзакции.
- Сервер – Принимающая сторона в текущей почтовой транзации.
- Агент доставки почты (Mail Transfer Agent, MTA) – Клиент и сервер SMTP обеспечивающее почтовый трансопртный сервис.
- Пользовательский почтовый агент (Mail User Agent, MUA) – Программное обеспечение выступающее в качетсве исходных отправителей и конечных получателей почтовых сообщений

$$MUA \rightarrow MTA \rightarrow MTA \rightarrow MUA$$

Типы агентов SMTP:

- Система отрпавки (originator) – Вносит сообщение в среду передачи данных, в котором находится транспортный сервис.
- Система доставки (delivery) – Принимает почту от транспортного серивса и передает ее пользовательскому агенту или размещает ее в хранилище.
- Транслятор (relay) – Получает почту от клиента и передает ее другому серверу.
- Шлюз (gateway) – Система получающие письма от одной транспортной среды и передающие письма сереверу находящейся в другой транспортной среде.

1.2 SMTP сеанс

При подключении клиента к серверу начинается SMTP сеанс, в течении которого выполняется взаимодействие клиента и сервера по доставки писем.

1. Инициирование соединения

   Клиент: создает соединение с сервером

   Сервер: Отправляет отклик

   - 220 в случае готовности
   - 554 в случае отказа в SMTP сервисе

2. Инициирование клиента (сеанса)

   Клиент: передает команду HELO/EHLO. HELO - содание SMTP сеанса. EHLO - создание SMTP сеанса с поддержкой расширений протокола (Extend Hello).

   Сревер: Отрпавляет отклик 250. Если бла отправлена команда EHLO, то сервер так же возращает список расширений, который он поддерживает (расширения далее не рассматриваются)

3. Почтовая транзакция (Транзакцию нельзя сделать вложеной в другую транзакцию)

   (a) Начало транзакции

   Клиент: Отправлет команду MAIL FROM. Команда говорит о запуске новой почтовой транзакции и передает адрес отправителя. Если в процессе передачи возникнет ошибка, на этот адрес будет отправлено уведомление.

   Сервер: Отклик 250

   (b) Определение спика получателей

   Для определение списка получателей клиент отправлет несколько команд RCPT TO, на каждую из которых сервер отправляет отклик 250.

   Если команда RCPT TO отправлена до начала почтовой транзакции, то сервер отправлет отклик 503

   (c) Передача тела письма

   Клиент: Отправлет команду DATA

   Сервер: отправлет отклик 354, что свидетельствует о том, что сервер готов принимать содержимое письма

   Клиент: Отрпавлет все почтовые данные. После завершения отправки тела письма, клиент должен отправить точку на отдельной строке (<CRLF>.<CRLF> – послеовательность кончания данных письма)

   Сервер: Должен воспринимать все присилаемые данные, как тело письма. Как только он получает последовательность конца данных (<CRLF>.<CRLF>) сервер должен инициировать процесс доставки письма. А клиенту отправить отклик 250

4. Завершение сеанса или новая транзакция

- Если клиент желает завершить работу с сервером, то он должен послеть команду QUIT, на которую сервер должен ответить откликом 221 и закрыть соединение.
- Если клиент желает продолжить работу с сервером, то он должен создать новую почтовую транзакцию. Для этого необходимо перейти на шаг 3a

Дополнительные команды:

1. VRFY
2. EXPN
3. RSET – прерывание текущей почтовой транзакции. Отклик сервера: 250
4. HELP
5. NOOP

1.3 Синтаксис команд

```
ehlo = "EHLO" SP Domain CRLF
helo = "HELO" SP Domain CRLF
ehlo-ok-rsp  = ("250" domain [SP ehlo-greet] CRLF)
             | ("250-" domain [SP ehlo-greet] CRLF
             | *("250-" ehlo-line CRLF)
             | ("250" SP ehlo-line CRLF)

ehlo-greet = 1*(%d0-9 / %d11-12 / %d14-127)
ehlo-line = ehlo-keyword *( SP ehlo-param )
ehlo-keyword = (ALPHA / DIGIT) *(ALPHA / DIGIT / "-")
ehlo-param   = 1*(%d33-127)
"MAIL FROM:" ("<>" / Reverse-Path) [SP Mail-parameters] CRLF
"RCPT TO:" ("<Postmaster@" domain ">" /
          "<Postmaster>" / Forward-Path)
          [SP Rcpt-parameters] CRLF
"DATA" CRLF
"RSET" CRLF
"VRFY" SP String CRLF
"EXPN" SP String CRLF
"HELP" [ SP String ] CRLF
"NOOP" [ SP String ] CRLF
"QUIT" CRLF
Reverse-path = Path
Forward-path = Path
Path = "<" [ A-d-l ":" ] Mailbox ">"
A-d-l = At-domain *( "," A-d-l )
```

```
At-domain = "@" domain
Mail-parameters = esmtp-param *(SP esmtp-param)
Rcpt-parameters = esmtp-param *(SP esmtp-param)
esmtp-param     = esmtp-keyword ["=" esmtp-value]
esmtp-keyword   = (ALPHA / DIGIT) *(ALPHA / DIGIT / "-")
esmtp-value     = 1*(%d33-60 / %d62-127)
Keyword  = Ldh-str
Argument = Atom
Domain = (sub-domain 1*("." sub-domain)) / address-literal
sub-domain = Let-dig [Ldh-str]
address-literal = "[" IPv4-address-literal /
                  IPv6-address-literal /
                  General-address-literal "]"
Mailbox         = Local-part "@" Domain
Local-part      = Dot-string / Quoted-string
Dot-string      = Atom *("." Atom)
Atom            = 1*atext
Quoted-string   = DQUOTE *qcontent DQUOTE
String          = Atom / Quoted-string
IPv4-address-literal    = Snum 3("." Snum)
IPv6-address-literal    = "IPv6:" IPv6-addr
General-address-literal = Standardized-tag ":" 1*dcontent
Standardized-tag        = Ldh-str
Snum                    = 1*3DIGIT
Let-dig                 = ALPHA / DIGIT
Ldh-str                 = *( ALPHA / DIGIT / "-" ) Let-dig
IPv6-addr               = IPv6-full / IPv6-comp / IPv6v4-full / IPv6v4-comp
IPv6-hex  = 1*4HEXDIG
IPv6-full = IPv6-hex 7(":" IPv6-hex)
IPv6-comp = [IPv6-hex *5(":" IPv6-hex)] "::" [IPv6-hex *5(":"IPv6-hex)]
IPv6v4-full = IPv6-hex 5(":" IPv6-hex) ":" IPv4-address-literal
IPv6v4-comp = [IPv6-hex *3(":" IPv6-hex)] "::"
            [IPv6-hex *3(":" IPv6-hex) ":"] IPv4-address-literal
```

1.4 Архитектура – Цикл событий

Для реализации поддержки многопоточности используется Циклы событий, смысл которого заключается в том, что существует бесконечный цикл, ожидающий событий на сокете через системный вызов poll. Данный цикл работает в отдельном потоке. Так же существует несколько других потоков, которые называются работниками. Они выполняют обработку возникающих событий. Сущесвует следующее множество событий, для которых можно назначить функцию обработчик:

1. Событие «Подключился клиент»

2. Событие «Выполнено чтение из сокета»
3. Событие «Выполнена запись в сокет»
4. Событие «Истек таймер»
5. Событие «Сокет был закрыт»

Основным преймуществом данного подхода заключается в том, что все потоки разделают одно адресное пространство – быстрое взаимодействие между ними. В отличие от многопроцессной архитектуры, в которой имеются накладные расходы на обмен информацией между процессами, так необходимо использоватеь соответсвующие системные вызовы, что заставлет процессор переключаться в режим ядра. Хотя постоянное использование механизмов синхронизации, при доступе к разделяемым ресурсам, так же замедляет работу приложения. А недостатком многопотончой архитектуры является ненадежность – если в одном из потоке произойдет критическая ошика (например SIGFAULT), то будет уничтожен процесс, соответсвенно все потоки приложения немедленно завершат свою работу. В этом случае многопроцессная архитектура имеет преймущество в виде надежности. Критическая ошибка в одном процессе не затрагивает другие процессы. Так же можно реализовать отдельный процесс, который будет выполнять мониторинг состояний рабочих процессов (watchdog-процесс), и вслучае экстренного завершения одного из процессов, watchdog-процесс должен будет выполнить восстановление завершившегося процесса (выполнение его повторного запуска).

Менеджер событий реализуется посредством струтуры event_loop, запускаемый функцией el_open в отдельном потоке. Структуре сопутсвуют функции для регистрации обработчиков на события. Регистрация на событие одноразовое, т.е. после вызова обработчика события, данный обработчик не будет реагировать на данное событие – его необходимо заново зарегистрировать.

Рабочим потокам, которые будут обрабатывать события, необхоимо в цилке вызывать функцию el_run, функция выполняет обработку одного события и завершает работу. Если в момент вызова, отсутсвовали какие либо события, то функция немедленно возращает управление, сообщая об отсутсвии событий в возращаемом статусе. Пример работы цикла событий показан на диаграмме последовательности (рис. 1.1).

Функция el_run только обрабатывает событие, не предполагает создание потока. Для реализации многопоточной обработки был реализована вспомогательная структура thread_pool, используюя функции которой создаются потоки. Внутри каждого потока в бесконечном цикле выполняется вызов функции el_run.

Для корректного завершения работы, бесконечные циклы, на самом деле проверяют условие «Необходиом ли дальше работать» (циклы логически бесконечны, так как они работают до тех пор, пока работает приложение). Для корректного заврешния работы программы, используется обработчик сигналов, который регистрируется в операционной системе. Если приходит сигнал SIGTERM или SIGINT, то обработчик вызывает функцию el_stop, которая изменяет флаг работы цикла событий с true на

false. Таким образом на очередной проверки условия работы цикла всеми потоками, учавствующие в цикле событий, будет произведен выход. Все потоки завершат свою работу и приложение остановится.

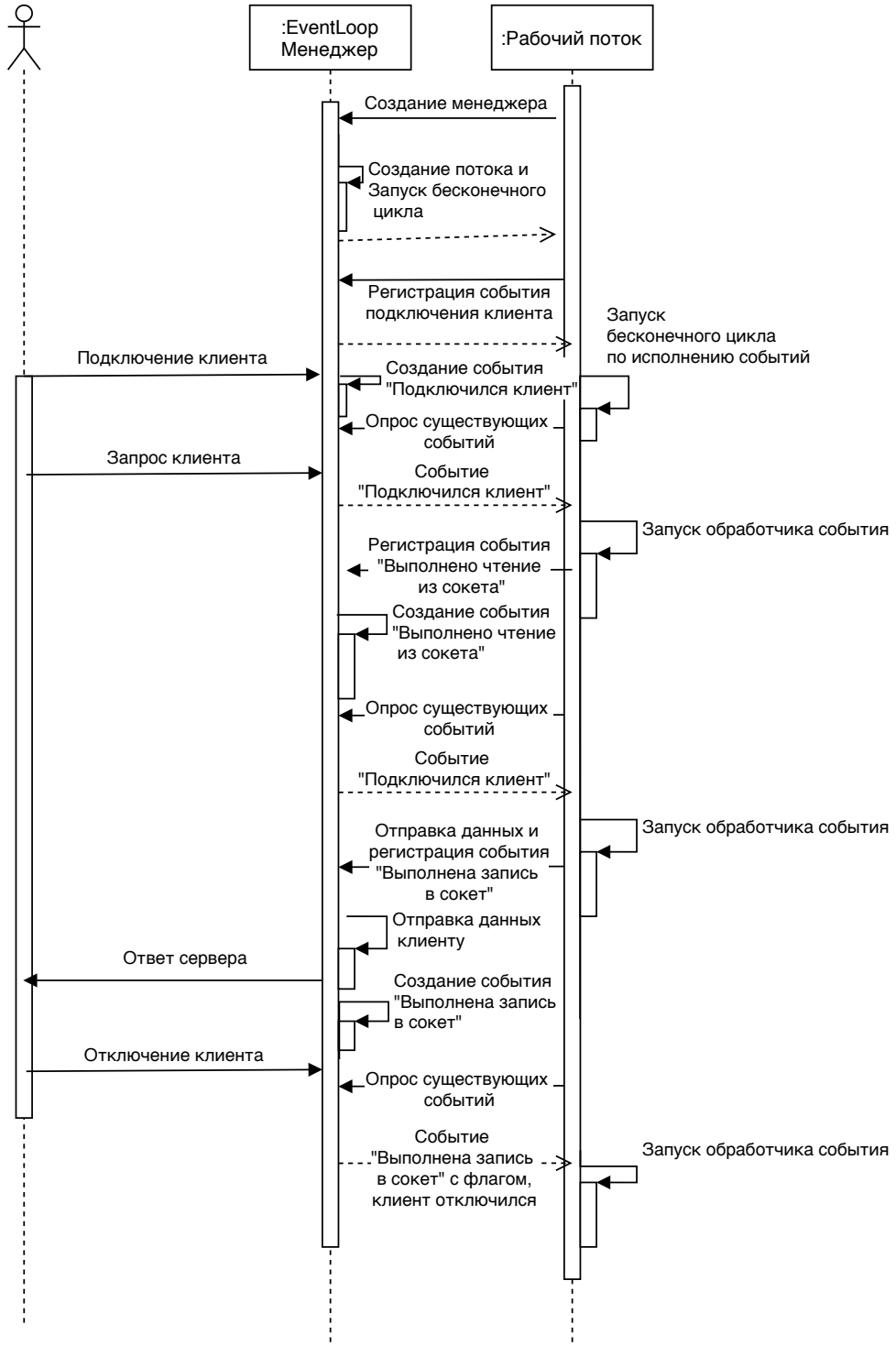

Рис. 1.1 Диграмма последовательности цикла событий сервера, описывающая подключение клиента, который отправлет запрос, ожидает ответ, а потом отключается. При отключении клиента, выставлется соответсвующий флаг, который обрабатывается в обработчике чтения из сокета или записи в сокет.

Глава 2

Конструкторский раздел

2.1 Реализаця протокола SMTP

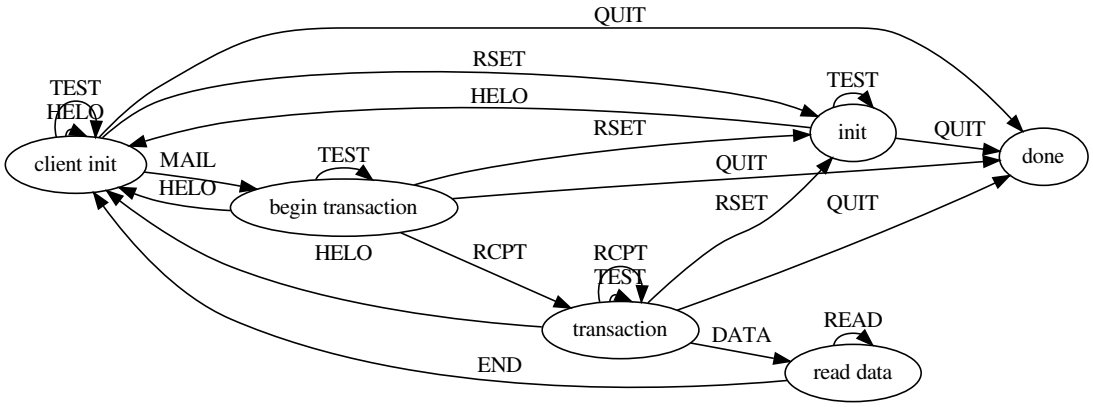

Рис. 2.1 Конечный автомат протокола SMTP

Обработка сеанса протокола SMTP выполняется на основе цикла событий. Для реализации конечного автомата и связанных с ним действий была реализована стркуктура smtp_state, которая содержит в себе конечный автомат, созданный посредством утилиты autofsm. На рисунке 2.1 показан конечный автомат, который описан в файле smtp-states.def. Овалами обозначены состояния, а метки ребер это команды. Таким образом каждая команда выполняет изменение состояния. Существует ряд меток, которые как команды, отсутсвуют в протоколе.

- TEST – это метка обозначает следующие команды: VRFY; EXPN; HELP NOOP;
- READ – это любая последовтаельность символов кроме .\r\n (точка на отдельной строке).

Состояния конечного автомата (рис. 2.1)

1. init – Начальное состояние, в котором находится соединение, когда клиент только подлкючился к серверу
2. client init – Состояние инициализированного smtp сеанса. В него выполняется переход после отправки команды HELO или EHLO
3. begin transaction – инициализация почтовой транзакции, которая происходит, когда клиент отправлет команду MAIL FROM
4. transaction – определение списка получателей, с помощью команды RCPT TO
5. read data – получение сервером тела письма. Данная стадия запускается командой DATA и продолжается до тех пор, пока не будет получена последовательность конца данных (точка на отдельной строке: .\r \n). На рисунке обозначена меткой END
6. done – завершение сенаса клиента с сервреом. Отправлется команда QUIT и сервер закрывает соединение с клиентом.

Автомат описывает только корректную последовательность команд, но если в некотором состоянии будет передана команда, которая не определена конечным автоматом, то состояние не изменится, а сервер сформирует отклик с кодом 503, означающий что клиент ввел неподходяющую команду (некорректная последовательность команд).

Для обработки команд SMTP протокола, синтаксис которых описан в разедел 1.3, исполльзовались регулярные выражения, которые входят в поставку вместе с компилятором языка C. Регулярные выражения описаны в следующем листинге:

```
RE_DOMAIN := ([[:alnum:]]+\\.)+[[:alnum:]]+
RE_IPv4 := ((25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\\.){3}(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)
RE_IPv6 :=
RE_GENERAL_ADDRESS_LITERAL :=
RE_ADDRESS_LITERAL := \\[ RE_IPv4 \\]
RE_DOMAIN_LITERAL := < RE_DOMAIN >
RE_AT_DOMAIN := @ RE_DOMAIN
RE_MAILBOX := [[:alnum:]]+ RE_AT_DOMAIN // TODO (ageev) если имя содеожит '_' то регуля
RE_ROUTE_PATH := (( RE_AT_DOMAIN [[:space:]]*,[[:space:]]*)* RE_AT_DOMAIN [[:space:]]*:
RE_PATH := < RE_ROUTE_PATH RE_MAILBOX >
RE_SERVER_NAME := ( RE_ADDRESS_LITERAL )|( RE_DOMAIN_LITERAL );
RE_HELLO := ((ehlo)|(helo))[[:space:]]+
RE_EMPTY_PATH := <>
RE_MAIL_FROM_PATH := (( RE_EMPTY_PATH )|( RE_PATH ))
RE_MAIL_FROM := mail[[:space:]]+from[[:space:]]*:[[:space:]]*
RE_RCPT_DOMAIN := <(postmaster RE_AT_DOMAIN )|(postmaster)|( RE_PATH )>
RE_RCPT_TO := rcpt[[:space:]]+to[[:space:]]*:[[:space:]]*
```

RE_DATA := data
RE_RSET := rset
RE_VRFY := vrfy
RE_EXPN := expn
RE_HELP := help
RE_NOOP := noop
RE_QUIT := quit

2.2 Maildir – формат хранения писем в файловой системе

Для хранения почты, получаемой сервером посрдеством почтовой транзакции, используется файловая система. Используемая струткутра каталогов взята из спецификации MAILDIR, которая описана в [???]. Формат maildir имеет следующую структуру каталогов:

```
- maildir_root
|
|- user_path
|    |- cur
|    |- tmp
|    |- new
|
|- user2_path
|    |- cur
|    |- tmp
|    |- new
```

Где maildir_root - кореньевая папка. user_path, user2_path - каталоги пользователей текущей почтовой службы. cur, tmp, new - папки содержащие письма. new - сюда попадают письма новые письма, которые пользователь не прочитал. cur - письма просмотренные пользователем. tmp - пиьсма находящиеся на стадии доставки, необходимость этой папки заключается в том, что запись данных в файл не является атомарной операцией. Если этой папки не будет то при создании файла, например в папке new, может произойти так, что программа для чтения локальной почты, попытается открыть этот новый файл, а программа доставки почты еще не успела туда записать данные. По этому используется каталог tmp для исключения таких случаев. Пока данные записываются в файл, тот находится в папке tmp, как только письмо полностью записано в файл, то программа доставки писем перенсит этот файл в каталог new.

При создании новых файлов писем, им необходимо даватб имена. Для создания уникального имени формат MAILDIR трактует слудющие правила:

<pid><sender_mailbox><timestamp><random_value>

Где <pid> - идентификатор процесса, выполняющий доставку почты; <sender_mailbox> - адрес электронной почты отправителя письма; <timestamp> - время UNIX; <random_value> - случайное целое число.

Поскольку формат MAILDIR разработан для доставки локальной почты, а по заданию необходимо обрабатывать и глобальную почту (почту адресованную пользователям на других серверах), то формат MAILDIR был модифицирован следующим образом:

```
- maildir_root
|
|- user_path
|    |- cur
|    |- tmp
|    |- new
|
|- user2_path
|    |- cur
|    |- tmp
|    |- new
|
|- .OTHER_SERVERS
|    |- tmp
|    |- new
```

Модификация maildir добавляет папку .OTHER_SERVERS, в которой слкадываются все письма, приходящие на данный сервер, но адресованные пользователям других почтовых служб. папки .OTHER_SERVERS/tmp и .OTHER_SERVERS/new имеют теже самые назначения, что и папки для пользователей. Папка .OTHER_SERVERS/cur отсутствует, так как после того как письма из папки .OTHER_SERVERS/new будет доставлено другому серверу будет удалена. Так же изменен формат создания уникального имени файла:

<timestamp>_<random_value>

Использую такой формат, нельзя определить по файлу от кого письмо и кому адресовано, по этому при записи письма в .OTHER_SERVERS к телу письма дописываются дополнительные заголовик, с помощью которых отправляющая программа сможет определить адресата и адресанта:

X-Postman-From: <mailbox>
X-Postman-Date: <timestamp>
X-Postman-To: <mailbox> [, <mailbox> [...]]
<пустая строка (\r\n)>
<тело письма полученное во время почтовой транзакции>

Обработка данной структуры в файловой системы реаизовано с помощью класса maildir. Дописыване дополнительных заголовков реализовано в обработчике почтовой транзакции, а не внутри класса maildir.

2.3 Логика программы

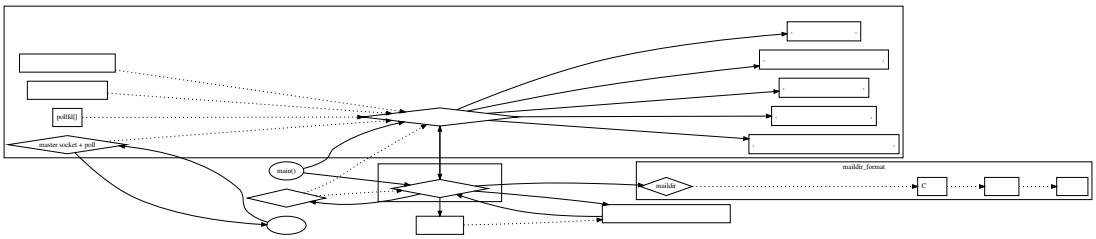

Рис. 2.2 Упращенная диаграмма взаимодействия модулей. EventLoop осуществлет обработку подключений. Рабоий поток осуществляет вызов обработчиков событий, которые EventLoop создает. Модуль mailidr обеспечивает сохраенние писем в файловой ситемы в соответсвующем формате

Программа написана с использованием Объектно Ориентированной парадигмы, хотя язык C напрямую ее не поддерживает. Для описания объектов используются структуры, для которых принято слоедующее соглашение:

- Поле считается приватным (private) если его название начинается с префика pr или _ (нижнее подчеркивание). Защищенные (protected) поля не используются.
- Функция (метод структуры) считается приватным, если его название начинается с префикса pr или _ (нижнее подчеркивание)

Так же используется некоторое подобие наследования в цилке событий, которе будет описано далее.

Работа программы начинается с того, что выполняется чтение конфигурационного файла, пример которого представлен в следующем листинге.

```
# Example application configuration file

server: {
        port: 8080
        host: "127.0.0.1"
        domain: "postman.local"
        maildir_path: "./maildir"
        worker_threads: 4
        timer: 20
        nice: 0
        log: {
                console_level: "DEBUG"
                files: (
                        {
                                path: "./server.log"
                                level: "DEBUG"
                        }
                )
        }
}
```

Чтение конфигурационного файла реализовано посредством бибилотеки libconfig. server_config_init реализует логику для заполнения глобальной структуры server_configuration содержащая все необходимые объекты для работы сервера.

После инициализации данной структуры, выполнятся создание экземпляра класса цилка событий $(\text{event}_{l}oopel\_init), , (master_{s}ocket, make\_server\_socket)..(- handler\_accept)event\_loop., , , event_{l}oop, .??., , .$

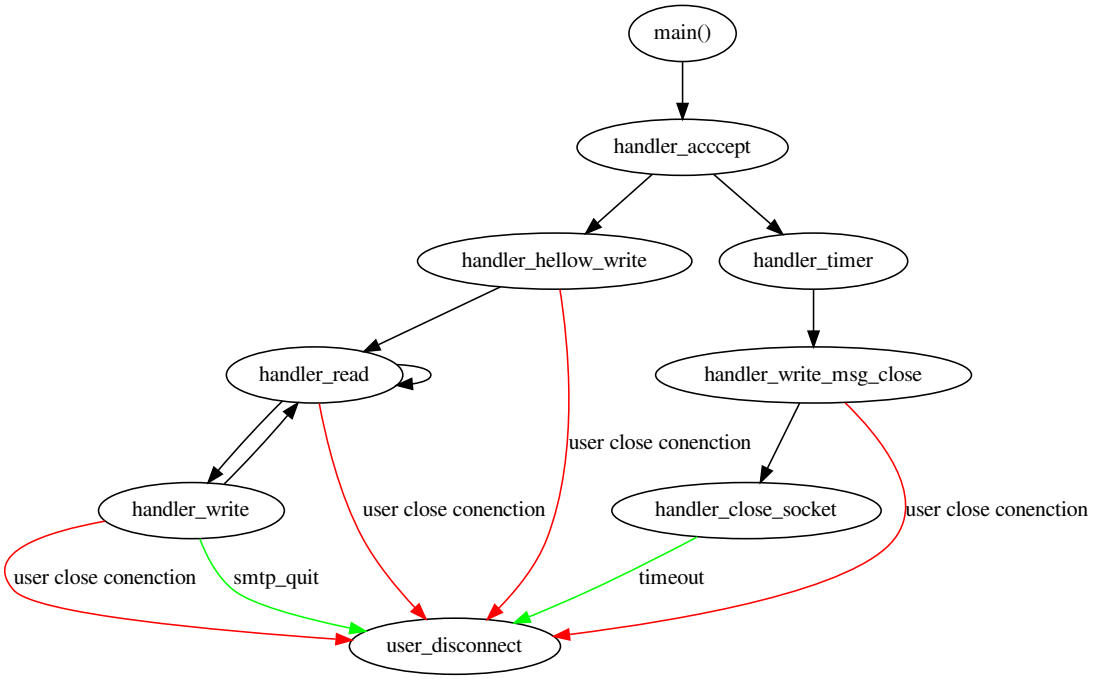

Рис. 2.3 Псоледовательность выполнения обработчиков событий. функции main и user_disconnet не являются обработчиками в смыле event_loop, а являются началом и концом работы. Стрелками обозначены возможные переходы во время работы. При этом две образовавшеся ветки, работают одновременно. Так как event_loop многопоточный. Зеленая стрелка означает корректный переход в конечное состояние, а красная – переход по возникшей ошибке

Жизненый цилк взаимодействия клиента с сервером начинает с обработчика handler_accept, который вызывается при подключении клиента к серверу. В этот момент регистрируются обработчики для отправки приветсвенного сообщения от сервера и таймер на ождинаие команд от пользователя. Левая ветка, описывает переходы между обработчиками во время передачи команд от клиента к серверу, и передачи откликов от сервера клиенту. Правая ветка следит за тем, что клиент успевая отправлть команды в зафиксированне промежутки времени. Но клиент, может выполнить отключение от сервера нарушая протокол SMTP (самостоятельно или по ошибке, такие переходы обозначены, красной линией). Завершение работы с клиентом выполняется в функции client_disconnect, в которой выоплняется очищение всех ресурсов, выделенных во время работы с клиентом.

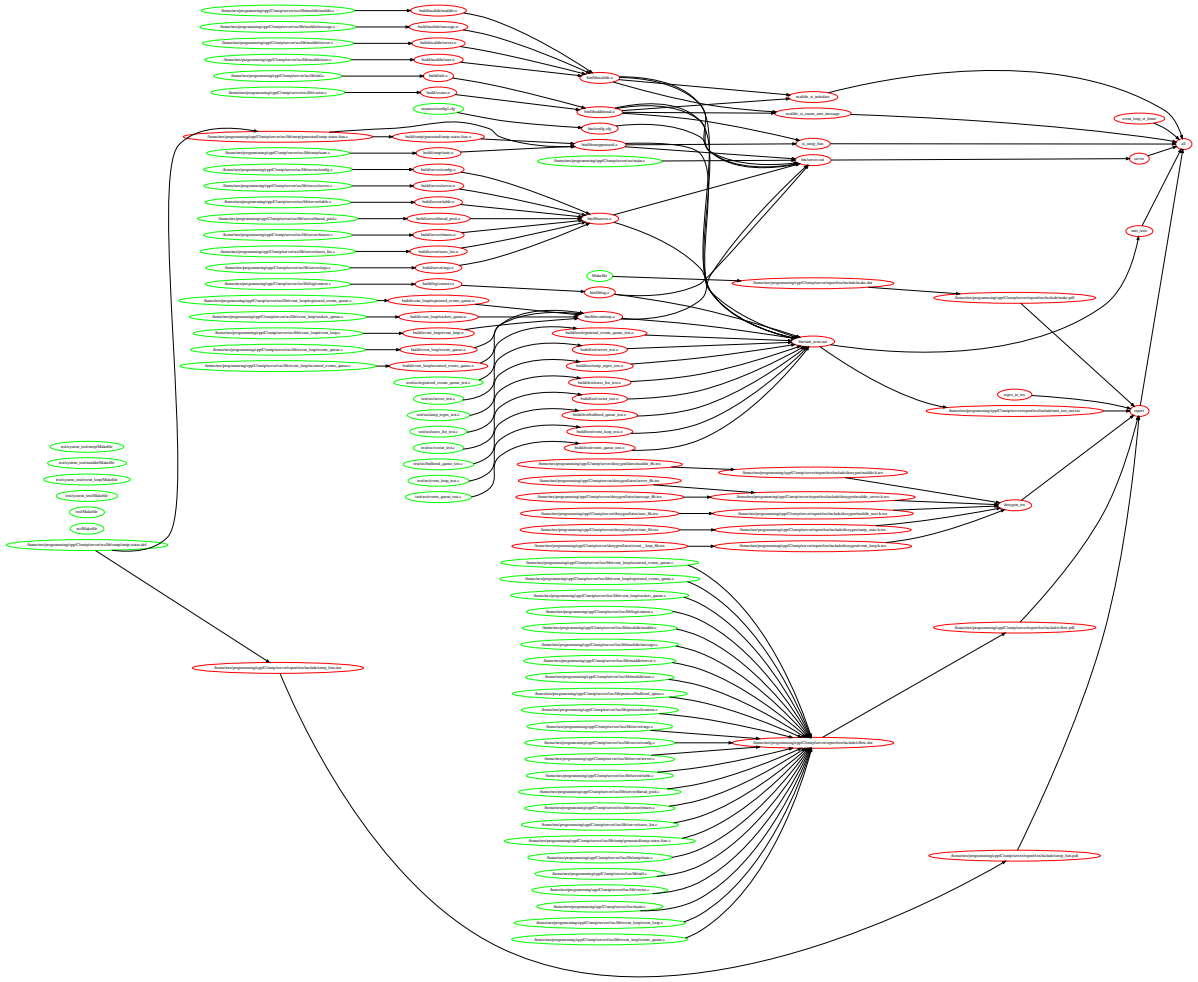

Рис. 2.4 Структура проекта.

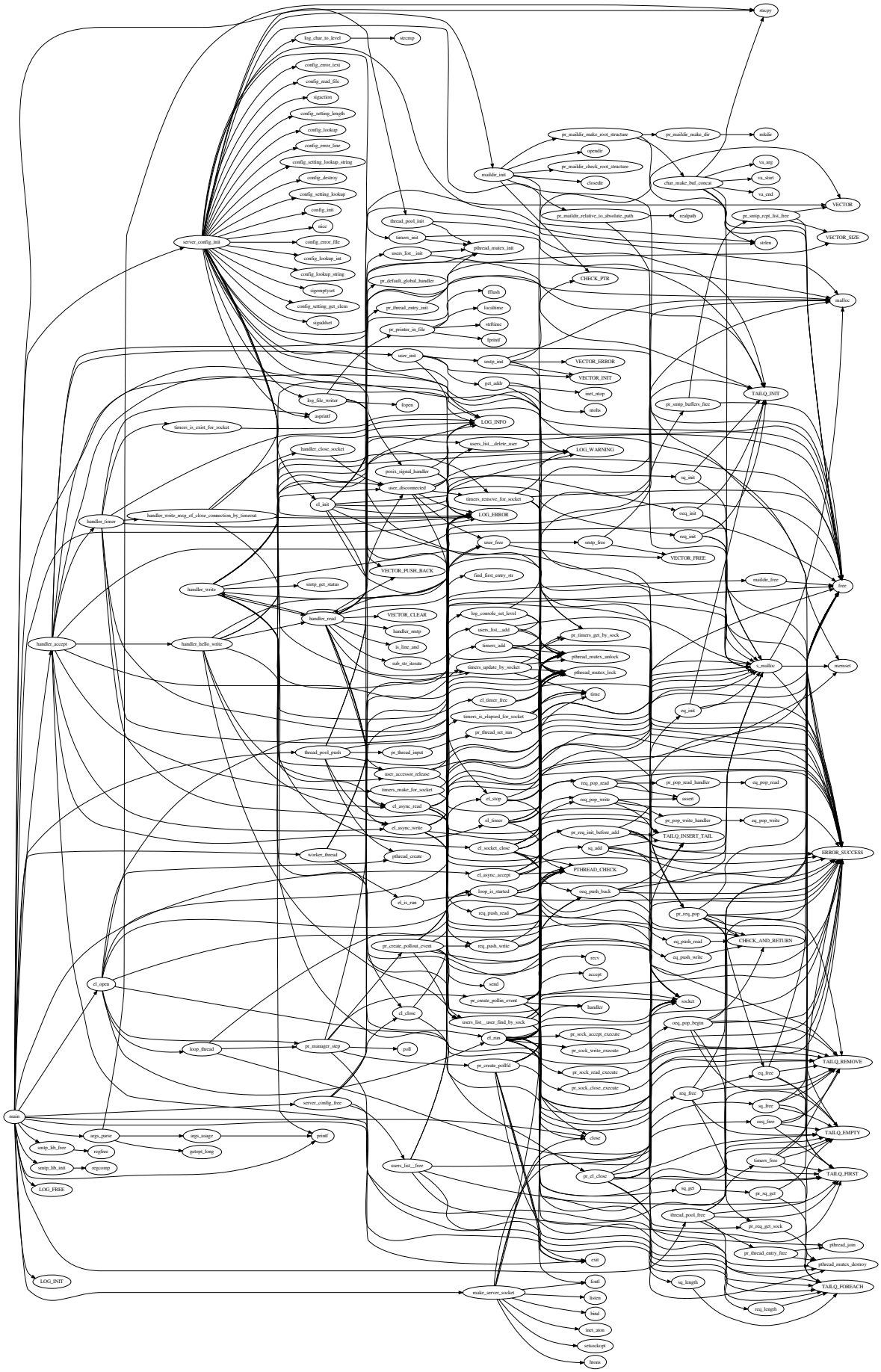

Рис. 2.5 Граф вызовов функций, который реализует всю логику.

2.4 Тестирование

Для тестирования отдельных модулей сервера, было написаны unit-тесты с использованием библиотеки cunit. Результат работы тестиованяи представлен в листинге

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


Suite: EventsQueue
  Test: add element in queue ...passed
  Test: add two elements of equals type in queue ...passed
  Test: pop element from queue ...passed
  Test: pop not existing element from queue ...passed
Suite: RegisteredEventsQueue
  Test: push element accept type in queue ...passed
  Test: push element read type in queue ...passed
  Test: push element write type in queue ...passed
  Test: pop element accept type in queue ...passed
  Test: pop element read type in queue ...passed
  Test: pop element write type in queue ...passed
  Test: get bitmask of registered events for socket ...passed
Suite: EventLoop
  Test: test init event loop ...passed
  Test: create pollfd from event_loop structure ...passed
  Test: process pollin ...passed
Suite: Vector
  Test: test init vector ...passed
  Test: test get element by wrong index ...passed
  Test: test create full copy ...passed
  Test: test create sub vector as first part ...passed
  Test: test create sub vector as second part ...passed
Suite: Smtp regex
  Test: regex hello ...passed
  Test: regex IPv4 ...passed
  Test: regex domain route list ...passed
  Test: mail from ...passed
  Test: rcpt to ...passed
  Test: parsing command 'hello' ...passed
  Test: parsing command 'mail from' ...passed
  Test: parsing command 'rcpt to' ...passed
  Test: check good command sequence ...passed
Suite: Users list
  Test: add element and find ...passed
```

```
Suite: Server
  Test: smtp session ...passed
  Test: substr iterate by sep ...passed

Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      7      7    n/a      0        0
               tests     31     31     31      0        0
             asserts    147    147    147      0      n/a

Elapsed time =    0.314 seconds

==24678== Memcheck, a memory error detector
==24678== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==24678== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==24678== Command: bin/unit_tests.out
==24678== Parent PID: 24677
==24678==
==24678==
==24678== HEAP SUMMARY:
==24678==     in use at exit: 0 bytes in 0 blocks
==24678==   total heap usage: 8,372 allocs, 8,372 frees, 1,220,980 bytes allocated
==24678==
==24678== All heap blocks were freed -- no leaks are possible
==24678==
==24678== For counts of detected and suppressed errors, rerun with: -v
==24678== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Так же было реализовано системное тестирование, на языке программирования python.

2.5 Основные функции программы

Данный раздел создан с помощью программы doxygen

2.5.1 Файл include/event_loop/event_loop.h

```
#include "event_t.h"
#include "sockets_queue.h"
#include "registered_events_queue.h"
#include "occurred_event_queue.h"
#include "protocol/buffered_queue.h"
#include <stdbool.h>
#include <sys/queue.h>
#include <pthread.h>
#include <poll.h>
#include <netinet/in.h>
```

Классы

- struct _async_error
- struct _event_loop

Макросы

- #define QUEUE_SIZE(entry_type, queue, field, res)

Определения типов

- typedef void(* error_global_handler) (int socket, err_t error, int line_execute, const char *function_execute)
- typedef enum __work_mode work_mode
- typedef enum _error_type error_type
- typedef struct _async_error async_error
- typedef struct _event_loop event_loop

Перечисления

- enum __work_mode { ONE_THREAD, OWN_THREAD }
- enum _error_type { NO_ERROR, ERROR }

Функции

- event_loop * el_init (err_t *error)
- bool el_open (event_loop *, work_mode mode, err_t *error)
- void el_close (event_loop *loop)
- bool el_run (event_loop *loop, err_t *error)
- bool el_async_accept (event_loop *loop, int sock, sock_accept_handler, err_t *error)
- bool el_async_read (event_loop *loop, int sock, char *buffer, int size, sock_read_handler, err_t *error)
- bool el_async_write (event_loop *loop, int sock, void *output_buffer, int bsize, sock_write_handler, err_t *error)
- bool el_socket_close (event_loop *loop, int sock, sock_close_handler, err_t *error)
- bool el_timer (event_loop *loop, int sock, unsigned int seconds, sock_timer_handler handler, timer_event_entry **descriptor, err_t *error)
- bool el_timer_free (event_loop *loop, timer_event_entry *descriptor)
- bool el_stop (event_loop *loop, err_t *error)
- bool el_is_run (event_loop *loop)

- bool el_reg_global_error_handler (event_loop *loop, error_global_handler hander, err_t *error)
- bool pr_create_pollfd (event_loop *loop, struct pollfd **fd_array, int *size, err_t *error)
- bool pr_create_pollin_event (event_loop *loop, struct pollfd *fd, int index, err_t *error)
- bool pr_create_pollout_event (event_loop *loop, struct pollfd *fd_array, int index, err_t *error)

2.5.1.1 Макросы

2.5.1.1.1 QUEUE_SIZE

```
#define QUEUE_SIZE(
                entry_type,
                queue,
                field,
                res )
```

Макроопределение:

```
do {                                \
    int i = 0;                      \
    entry_type *__ptr__ = NULL;     \
     TAILQ_FOREACH(__ptr__, queue, field) {  \
        i = i + 1;                  \
     }                              \
    *res = i;                       \
} while(0)\
```

2.5.1.2 Типы

2.5.1.2.1 async_error

typedef struct _async_error async_error

2.5.1.2.2 error_global_handler

typedef void(* error_global_handler) (int socket, err_t error, int line_execute, const char *function↩
_execute)

Описание глобального обработчика socket - файловый дескриптор (сокет) при работе с котроым возникал ошибка error - что произошло

2.5.1.2.3 error_type

typedef enum _error_type error_type

2.5.1.2.4 event_loop

typedef struct _event_loop event_loop

2.5.1.2.5 work_mode

typedef enum __work_mode work_mode

2.5.1.3 Перечисления

2.5.1.3.1 __work_mode

enum __work_mode

Элементы перечислений

ONE_THREAD	
OWN_THREAD	Менеджер очереди и обработка произошедших событий будет выполнятся в одном потоке Менеджер очереди будет выполнятся в отдельном потоке, для обработки событий необходимо вызывать функцию el_run в отдельном потоке

2.5.1.3.2 _error_type

enum _error_type

Элементы перечислений

NO_ERROR	
ERROR	

2.5.1.4 Функции

2.5.1.4.1 el_async_accept()

```
bool el_async_accept (
        event_loop * loop,
        int sock,
        sock_accept_handler ,
        err_t * error )
```

Регистрация обработчика осбытия "Подключение нового клиента" для сокета. Соект должен быть настроен, как неблокирующий и быть слушающим.

Аргументы

loop	цикл событий, в котором зарегистрировать данное событие
sock	- слушающий неблокирующийся сокет
error	- статус выполнения операции

Возвращает

true - операция заврешилась успешно; false - операция завершилась с ошибкой.

Прим.

thread save

2.5.1.4.2 el_async_read()

```
bool el_async_read (
        event_loop * loop,
        int sock,
        char * buffer,
        int size,
        sock_read_handler ,
        err_t * error )
```

Регистрация обработчика события "Чтение данных из сокета" для указанного сокета. Соект должен быть настроен, как неблокирующий. Если сокет был получен врезультате вызова обработчкиа события "подключение нового клиента" (

См. также

el_async_accept) то сокет уже имеет соответсвующие настройки. С ним ни чего делать не нужно.

Аргументы

loop	цикл событий, в котором зарегистрировать данное событие
sock	неблокирующий сокет, для которого регистрировать событие
buffer	Указатель на буффер, в который должно произойти запись данных при чтении
size	размер буфера
error	статус выполнения операции

Возвращает

true - операция заврешилась успешно; false - операция завершилась с ошибкой.

Прим.

thread save

2.5.1.4.3 el_async_write()

```
bool el_async_write (
        event_loop * loop,
        int sock,
```

void * output_buffer,
int bsize,
sock_write_handler ,
err_t * error)

Регистрация обработчика события "Запись данных в сокет" для указанного сокета. Соект должен быть настроен, как неблокирующий. Если сокет был получен врезультате вызова обработчкиа события "подключение нового клиента" (el_↩ async_accept) то сокет уже имеет соответсвующие настройки. С ним ни чего делать не нужно.

Аргументы

loop	цикл событий, в котором зарегистрировать данное событие
sock	неблокирующий сокет, для которого регистрировать событие
output_buffer	буфер из которого небходимо выполнить чтение при записи даннх в сокет
bsize	размер буфера
error	статус выполнения операции

Возвращает

true - операция заврешилась успешно; false - операция завершилась с ошибкой.

Прим.

thread save

2.5.1.4.4 el_close()

void el_close (
event_loop * loop)

Освобождение ресурсов

Аргументы

loop	- цикл событий из подкоторого необходимо освободить ресурсы

Прим.

No thread save

2.5.1.4.5 el_init()

```
event_loop* el_init (
            err_t * error )
```

Создание цикла событий

Аргументы

error	- статус выполнения операции

Возвращает

указатель на event_loop при успехе; NULL - если возникла ошибка

2.5.1.4.6 el_is_run()

```
bool el_is_run (
            event_loop * loop )
```

2.5.1.4.7 el_open()

```
bool el_open (
            event_loop * ,
            work_mode mode,
            err_t * error )
```

Инициализация цикла событий и его запуск. Поведение функции зависит от значение параметра mode:

- ONE_THREAD - в одном потоке будет работать менеджер событий и их обработчик. функция будет заблокирована до тех пор, пока цикл событий не будет остановлен (el_stop)
- OWN_THREAD - для менеджера событий будет создан отдельный поток. Функция сразу вернет управление. Обработка происходящих событий должна вестить вручную (вызовом функции el_run). Таким образом менеджер событий и обработчик соыбтий работают в разных потоках. Обработчиков событий может быть несколько

Аргументы

mode	режим работы
error	статус выполнения операции

Возвращает

true - операция заврешилась успешно; false - операция завершилась с ошибкой.

Прим.

Не предназначен для запуска из множества потоков. Создание несколько менеджеров событий не поддерживается.

2.5.1.4.8 el_reg_global_error_handler()

```
bool el_reg_global_error_handler (
        event_loop * loop,
        error_global_handler hander,
        err_t * error )
```

Регистрация глобального обработчика ошибок. Имеются набор ошибок, которые необходимо обрабатывать немедленно. Например, ошибка добавление нового события в очередь зарегистрированных событий. Данный обработчик вызывается (Если это возможно) при фатальных ошибках.

Аргументы

loop	- цикл событий
hander	- обрабочик
error	- возврат ошибок, которые могут произойти при добавлении обработчика

Возвращает

true - операция заврешилась успешно; false - операция завершилась с ошибкой.

2.5.1.4.9 el_run()

```
bool el_run (
        event_loop * loop,
        err_t * error )
```

Выполнение обработки одного произошедшего события. Если существует некоторое событие, то для него будет вызван обработчик в том потоке, в котором была вызвана данная функция. Если события отсутсвуют, то тогда функция вернет ошибку NO↩ T_FOUND в параметре error

Аргументы

loop	- цикл событий, для которого необходимо обработать произошедшие события
error	- статус выполнения операции

Возвращает

true - если был вызван обработчик события; false - если обработчик события не был вызван

Прим.

thread save

2.5.1.4.10 el_socket_close()

```
bool el_socket_close (
        event_loop * loop,
        int sock,
        sock_close_handler ,
        err_t * error )
```

2.5.1.4.11 el_stop()

```
bool el_stop (
        event_loop * loop,
        err_t * error )
```

Остановка цикла событий

Аргументы

loop	цикл событий, который необходимо остановить
error	статуст выполнения операции

Возвращает

true - операция заврешилась успешно; false - операция завершилась с ошибкой.

2.5.1.4.12 el_timer()

```
bool el_timer (
        event_loop * loop,
        int sock,
        unsigned int seconds,
        sock_timer_handler handler,
        timer_event_entry ** descriptor,
        err_t * error )
```

Регистрация обработчика события "Для соекта истек таймер"

Аргументы

loop	
sock	
ms	

Возвращает

2.5.1.4.13 el_timer_free()

```
bool el_timer_free (
        event_loop * loop,
        timer_event_entry * descriptor )
```

Отключение таймера и осовбождение памяти.

Аргументы

loop	
descriptor	

Возвращает

2.5.1.4.14 pr_create_pollfd()

```
bool pr_create_pollfd (
                event_loop * loop,
                struct pollfd ** fd_array,
                int * size,
                err_t * error )
```

2.5.1.4.15 pr_create_pollin_event()

```
bool pr_create_pollin_event (
                event_loop * loop,
                struct pollfd * fd,
                int index,
                err_t * error )
```

2.5.1.4.16 pr_create_pollout_event()

```
bool pr_create_pollout_event (
                event_loop * loop,
                struct pollfd * fd_array,
                int index,
                err_t * error )
```

2.5.2 Файл include/smtp/state.h

```
#include "smtp-states-fsm.h"
#include "error_t.h"
#include "vector.h"
#include "vector_structures.h"
```

Классы

- struct smtp_mailbox
- struct smtp_address
- struct d_smtp_state
- struct smtp_command

Макросы

- #define SMTP_COMMAND_END "\r\n"
- #define SMTP_COMMAND_END_LEN 2

Определения типов

- typedef struct smtp_mailbox smtp_mailbox
- typedef struct smtp_address smtp_address
- typedef enum d_smtp_status smtp_status
- typedef struct d_smtp_state smtp_state
- typedef struct smtp_command smtp_command

Перечисления

- enum smtp_address_type { SMTP_ADDRESS_TYPE_IPv4, SMTP_ADDRE↩ SS_TYPE_IPv6, SMTP_ADDRESS_TYPE_DOMAIN, SMTP_ADDRESS_↩ TYPE_NONE }
- enum d_smtp_status { SMTP_STATUS_ERROR, SMTP_STATUS_OK, SMTP_STATUS_WARNI↩ NG, SMTP_STATUS_CONTINUE, SMTP_STATUS_DATA_END, SMTP_STATUS_EXIT }
- enum smtp_command_type { SMTP_HELLO, SMTP_MAILFROM, SMTP_RCPTTO, SMTP_DATA, SMTP_RSET, SMTP_VRFY, SMTP_EXPN, SMTP_HELP, SMTP_NOOP, SMTP_QUIT, SMTP_INVALID_COMMAND }

Функции

- VECTOR_DECLARE (vector_smtp_mailbox, smtp_mailbox)
- void smtp_lib_init ()
- void smtp_lib_free ()
- bool smtp_init (smtp_state *smtp, err_t *error)
- void smtp_free (smtp_state *smtp)
- smtp_status smtp_parse (smtp_state *smtp, const char *message, char **buffer↩ _reply, err_t *error)
- char * smtp_make_response (smtp_state *smtp, size_t code, const char *msg)
- bool smtp_move_buffer (smtp_state *smtp, char **buffer, size_t *blen, err_↩ t *error)
- vector_smtp_mailbox * smtp_get_rcpt (smtp_state *smtp)
- smtp_mailbox * smtp_get_sender (smtp_state *smtp)
- smtp_status smtp_get_status (smtp_state *smtp)

2.5.2.1 Макросы

2.5.2.1.1 SMTP_COMMAND_END

#define SMTP_COMMAND_END "\r\n"

2.5.2.1.2 SMTP_COMMAND_END_LEN

#define SMTP_COMMAND_END_LEN 2

2.5.2.2 Типы

2.5.2.2.1 smtp_address

typedef struct smtp_address smtp_address

2.5.2.2.2 smtp_command

typedef struct smtp_command smtp_command

2.5.2.2.3 smtp_mailbox

typedef struct smtp_mailbox smtp_mailbox

2.5.2.2.4 smtp_state

typedef struct d_smtp_state smtp_state

2.5.2.2.5 smtp_status

typedef enum d_smtp_status smtp_status

2.5.2.3 Перечисления

2.5.2.3.1 d_smtp_status

enum d_smtp_status

Элементы перечислений

SMTP_STATUS_ERROR	
SMTP_STATUS_OK	Произошла ошибка во время обработки сообщения
SMTP_STATUS_WARNING	Сообщение полностью обработано
SMTP_STATUS_CONTINUE	Сообщение полностью обраотано, но ответ для клиента отрицательный
SMTP_STATUS_DATA_END	Сообщение состоит из множества строк, необходимо продолжить обрабатывать строки
SMTP_STATUS_EXIT	Тело письма завершено. письмо можно доставлять.

2.5.2.3.2 smtp_address_type

enum smtp_address_type

Элементы перечислений

SMTP_ADDRESS_TYPE_IPv4	
SMTP_ADDRESS_TYPE_IPv6	
SMTP_ADDRESS_TYPE_DOMAIN	
SMTP_ADDRESS_TYPE_NONE	

2.5.2.3.3 smtp_command_type

enum smtp_command_type

Элементы перечислений

SMTP_HELLO	
SMTP_MAILFROM	
SMTP_RCPTTO	
SMTP_DATA	
SMTP_RSET	
SMTP_VRFY	
SMTP_EXPN	
SMTP_HELP	
SMTP_NOOP	
SMTP_QUIT	
SMTP_INVALID_COMMAND	

2.5.2.4 Функции

2.5.2.4.1 smtp_free()

void smtp_free (
smtp_state * smtp)

2.5.2.4.2 smtp_get_rcpt()

vector_smtp_mailbox* smtp_get_rcpt (
smtp_state * smtp)

2.5.2.4.3 smtp_get_sender()

smtp_mailbox* smtp_get_sender (
smtp_state * smtp)

2.5.2.4.4 smtp_get_status()

smtp_status smtp_get_status (
smtp_state * smtp)

2.5.2.4.5 smtp_init()

bool smtp_init (
smtp_state * smtp,
err_t * error)

Инициализация состояния для протокола smtp

Аргументы

smtp	- описатель состояния
error	

Возвращает

статус выполнения операции

2.5.2.4.6 smtp_lib_free()

void smtp_lib_free ( )

Освобождение всех ресурсов из под библиотеки

2.5.2.4.7 smtp_lib_init()

void smtp_lib_init ( )

Инициализация библиотеки для обработки smtp протокола

2.5.2.4.8 smtp_make_response()

char* smtp_make_response (
smtp_state * smtp,
size_t code,
const char * msg)

2.5.2.4.9 smtp_move_buffer()

bool smtp_move_buffer (
smtp_state * smtp,
char ** buffer,
size_t * blen,
err_t * error)

Перенос буфера получателя сообщения. После вызова этой функции буфером владеет пользователь. Он должен освободить ресурсы

Аргументы

smtp	
buffer	
error	

Возвращает

2.5.2.4.10 smtp_parse()

smtp_status smtp_parse (
smtp_state * smtp,
const char * message,

```
char ** buffer_reply,
err_t * error )
```

Обработка протокольных сообщений SMTP. Функция выдает статус обрабокти и протокольный отклик на сообщение

Аргументы

smtp	- описатьель smtp контекста
message	- протокольное сообщение для обработки
buffer	- протокольный отклик. Указатель на указатель буфера - если размер буфера для отклика будет не достаточен, то буде выделена новый участок памяти, а старый будет освобожден
error	- описатель статуса выполнения операции

Возвращает

- статус обработки SMTP сообщения.
  - SMTP_ERROR - ошибка обработки сообщения, необходимо проверить error
  - SMTP_OK - сообщение полностью обработано
  - SMTP_CONTINUE - сообщение яявляется многострочным. Текущая часть сообщения успешно обработано, необходимо переадть оставшиеся части (отклик не формируется!) На каждыое действие в SMTP формируется протокольный отклик, если не указано иного

2.5.2.4.11 VECTOR_DECLARE()

```
VECTOR_DECLARE (
        vector_smtp_mailbox ,
        smtp_mailbox  )
```

2.5.3 Файл include/smtp/state.h

```
#include "smtp-states-fsm.h"
#include "error_t.h"
#include "vector.h"
#include "vector_structures.h"
```

Классы

- struct smtp_mailbox
- struct smtp_address
- struct d_smtp_state
- struct smtp_command

Макросы

- #define SMTP_COMMAND_END "\r\n"
- #define SMTP_COMMAND_END_LEN 2

Определения типов

- typedef struct smtp_mailbox smtp_mailbox
- typedef struct smtp_address smtp_address
- typedef enum d_smtp_status smtp_status
- typedef struct d_smtp_state smtp_state
- typedef struct smtp_command smtp_command

Перечисления

- enum smtp_address_type { SMTP_ADDRESS_TYPE_IPv4, SMTP_ADDRESS_TYPE_IPv6, SMTP_ADDRESS_TYPE_DOMAIN, SMTP_ADDRESS_TYPE_NONE }
- enum d_smtp_status { SMTP_STATUS_ERROR, SMTP_STATUS_OK, SMTP_STATUS_WARNING, SMTP_STATUS_CONTINUE, SMTP_STATUS_DATA_END, SMTP_STATUS_EXIT }
- enum smtp_command_type { SMTP_HELLO, SMTP_MAILFROM, SMTP_RCPTTO, SMTP_DATA, SMTP_RSET, SMTP_VRFY, SMTP_EXPN, SMTP_HELP, SMTP_NOOP, SMTP_QUIT, SMTP_INVALID_COMMAND }

Функции

- VECTOR_DECLARE (vector_smtp_mailbox, smtp_mailbox)
- void smtp_lib_init ()
- void smtp_lib_free ()
- bool smtp_init (smtp_state *smtp, err_t *error)
- void smtp_free (smtp_state *smtp)
- smtp_status smtp_parse (smtp_state *smtp, const char *message, char **buffer_reply, err_t *error)
- char * smtp_make_response (smtp_state *smtp, size_t code, const char *msg)
- bool smtp_move_buffer (smtp_state *smtp, char **buffer, size_t *blen, err_t *error)
- vector_smtp_mailbox * smtp_get_rcpt (smtp_state *smtp)
- smtp_mailbox * smtp_get_sender (smtp_state *smtp)
- smtp_status smtp_get_status (smtp_state *smtp)

2.5.3.1 Макросы

2.5.3.1.1 SMTP_COMMAND_END

#define SMTP_COMMAND_END "\r\n"

2.5.3.1.2 SMTP_COMMAND_END_LEN

#define SMTP_COMMAND_END_LEN 2

2.5.3.2 Типы

2.5.3.2.1 smtp_address

typedef struct smtp_address smtp_address

2.5.3.2.2 smtp_command

typedef struct smtp_command smtp_command

2.5.3.2.3 smtp_mailbox

typedef struct smtp_mailbox smtp_mailbox

2.5.3.2.4 smtp_state

typedef struct d_smtp_state smtp_state

2.5.3.2.5 smtp_status

typedef enum d_smtp_status smtp_status

2.5.3.3 Перечисления

2.5.3.3.1 d_smtp_status

enum d_smtp_status

Элементы перечислений

SMTP_STATUS_ERROR	
SMTP_STATUS_OK	Произошла ошибка во время обработки сообщения
SMTP_STATUS_WARNING	Сообщение полностью обработано
SMTP_STATUS_CONTINUE	Сообщение полностью обраотано, но ответ для клиента отрицательный
SMTP_STATUS_DATA_END	Сообщение состоит из множества строк, необходимо продолжить обрабатывать строки
SMTP_STATUS_EXIT	Тело письма завершено. письмо можно доставлять.

2.5.3.3.2 smtp_address_type

enum smtp_address_type

Элементы перечислений

SMTP_ADDRESS_TYPE_IPv4	
SMTP_ADDRESS_TYPE_IPv6	
SMTP_ADDRESS_TYPE_DOMAIN	
SMTP_ADDRESS_TYPE_NONE	

2.5.3.3.3 smtp_command_type

enum smtp_command_type

Элементы перечислений

SMTP_HELLO	
SMTP_MAILFROM	
SMTP_RCPTTO	
SMTP_DATA	
SMTP_RSET	
SMTP_VRFY	
SMTP_EXPN	
SMTP_HELP	
SMTP_NOOP	

Элементы перечислений

SMTP_QUIT	
SMTP_INVALID_COMMAND	

2.5.3.4 Функции

2.5.3.4.1 smtp_free()

void smtp_free (
smtp_state * smtp)

2.5.3.4.2 smtp_get_rcpt()

vector_smtp_mailbox* smtp_get_rcpt (
smtp_state * smtp)

2.5.3.4.3 smtp_get_sender()

smtp_mailbox* smtp_get_sender (
smtp_state * smtp)

2.5.3.4.4 smtp_get_status()

smtp_status smtp_get_status (
smtp_state * smtp)

2.5.3.4.5 smtp_init()

bool smtp_init (
smtp_state * smtp,
err_t * error)

Инициализация состояния для протокола smtp

Аргументы

smtp	- описатель состояния
error	

Возвращает

статус выполнения операции

2.5.3.4.6 smtp_lib_free()

void smtp_lib_free ( )

Освобождение всех ресурсов из под библиотеки

2.5.3.4.7 smtp_lib_init()

void smtp_lib_init ( )

Инициализация библиотеки для обработки smtp протокола

2.5.3.4.8 smtp_make_response()

```
char* smtp_make_response (
            smtp_state * smtp,
            size_t code,
            const char * msg )
```

2.5.3.4.9 smtp_move_buffer()

```
bool smtp_move_buffer (
            smtp_state * smtp,
            char ** buffer,
            size_t * blen,
            err_t * error )
```

Перенос буфера получателя сообщения. После вызова этой функции буфером владеет пользователь. Он должен освободить ресурсы

Аргументы

smtp	
buffer	
error	

Возвращает

2.5.3.4.10 smtp_parse()

```
smtp_status smtp_parse (
            smtp_state * smtp,
            const char * message,
            char ** buffer_reply,
            err_t * error )
```

Обработка протокольных сообщений SMTP. Функция выдает статус обрабокти и протокольный отклик на сообщение

Аргументы

smtp	- описатьель smtp контекста
message	- протокольное сообщение для обработки
buffer	- протокольный отклик. Указатель на указатель буфера - если размер буфера для отклика будет не достаточен, то буде выделена новый участок памяти, а старый будет освобожден
error	- описатель статуса выполнения операции

Возвращает

- статус обработки SMTP сообщения.
  - SMTP_ERROR - ошибка обработки сообщения, необходимо проверить error
  - SMTP_OK - сообщение полностью обработано
  - SMTP_CONTINUE - сообщение яявляется многострочным. Текущая часть сообщения успешно обработано, необходимо переадть оставшиеся части (отклик не формируется!) На каждыое действие в SMTP формируется протокольный отклик, если не указано иного

2.5.3.4.11 VECTOR_DECLARE()

```
VECTOR_DECLARE (
        vector_smtp_mailbox ,
        smtp_mailbox  )
```

2.5.4 Файл include/maildir/maildir.h

```
#include "error_t.h"
#include "vector_structures.h"
#include "server.h"
#include <stdbool.h>
#include <sys/queue.h>
#include <dirent.h>
```

Классы

- struct d_maildir_server_entry
- struct maildir_log_handlers
- struct d_maildir

Макросы

- #define SERVERS_ROOT_NAME ".OTHER_SERVERS"
- #define SERVERS_ROOT_NAME_PART "/.OTHER_SERVERS/"
- #define USER_PATH_CUR "cur"
- #define USER_PATH_TMP "tmp"
- #define USER_PATH_NEW "new"

Определения типов

- typedef struct d_maildir_user maildir_user
- typedef struct d_maildir_server maildir_server
- typedef struct d_maildir_users_list maildir_users_list
- typedef struct d_maildir_server_entry maildir_server_entry
- typedef struct d_maildir_servers_list maildir_servers_list
- typedef void(* maildir_log_handler) (char *message)
- typedef struct d_maildir maildir

Функции

- LIST_HEAD (d_maildir_servers_list, d_maildir_server_entry)
- bool maildir_init (maildir *md, const char *path, err_t *error)
- void maildir_free (maildir *md)
- bool maildir_release (maildir *md, err_t *error)
- bool maildir_get_self_server (maildir *md, maildir_server *server, err_t *error)
- bool maildir_get_server (maildir *md, maildir_server *server, err_t *error)
- bool maildir_delete_server (maildir *md, maildir_server *server, err_t *error)
- bool maildir_set_logger_handlers (maildir *md, struct maildir_log_handlers *handlers)

2.5.4.1 Макросы

2.5.4.1.1 SERVERS_ROOT_NAME

#define SERVERS_ROOT_NAME ".OTHER_SERVERS"

2.5.4.1.2 SERVERS_ROOT_NAME_PART

#define SERVERS_ROOT_NAME_PART "/.OTHER_SERVERS/"

2.5.4.1.3 USER_PATH_CUR

#define USER_PATH_CUR "cur"

2.5.4.1.4 USER_PATH_NEW

#define USER_PATH_NEW "new"

2.5.4.1.5 USER_PATH_TMP

#define USER_PATH_TMP "tmp"

2.5.4.2 Типы

2.5.4.2.1 maildir

typedef struct d_maildir maildir

2.5.4.2.2 maildir_log_handler

typedef void(* maildir_log_handler) (char *message)

2.5.4.2.3 maildir_server

typedef struct d_maildir_server maildir_server

2.5.4.2.4 maildir_server_entry

typedef struct d_maildir_server_entry maildir_server_entry

2.5.4.2.5 maildir_servers_list

typedef struct d_maildir_servers_list maildir_servers_list

2.5.4.2.6 maildir_user

typedef struct d_maildir_user maildir_user

2.5.4.2.7 maildir_users_list

typedef struct d_maildir_users_list maildir_users_list

2.5.4.3 Функции

2.5.4.3.1 LIST_HEAD()

```
LIST_HEAD (
        d_maildir_servers_list ,
        d_maildir_server_entry  )
```

2.5.4.3.2 maildir_delete_server()

```
bool maildir_delete_server (
        maildir * md,
        maildir_server * server,
        err_t * error )
```

2.5.4.3.3 maildir_free()

```
void maildir_free (
        maildir * md )
```

2.5.4.3.4 maildir_get_self_server()

```
bool maildir_get_self_server (
        maildir * md,
        maildir_server * server,
        err_t * error )
```

2.5.4.3.5 maildir_get_server()

```
bool maildir_get_server (
        maildir * md,
        maildir_server * server,
        err_t * error )
```

2.5.4.3.6 maildir_init()

```
bool maildir_init (
        maildir * md,
        const char * path,
        err_t * error )
```

2.5.4.3.7 maildir_release()

```
bool maildir_release (
        maildir * md,
        err_t * error )
```

2.5.4.3.8 maildir_set_logger_handlers()

```
bool maildir_set_logger_handlers (
        maildir * md,
        struct maildir_log_handlers * handlers )
```

2.5.5 Файл include/maildir/server.h

```
#include "error_t.h"
#include <stdbool.h>
#include <linux/limits.h>
```

Классы

- struct d_maildir_server

Определения типов

- typedef struct d_maildir_user maildir_user
- typedef struct d_maildir maildir
- typedef struct d_maildir_users_list maildir_users_list
- typedef struct d_maildir_server maildir_server

Функции

- bool pr_maildir_server_init (maildir_server *server, err_t *error)
- void maildir_server_default_init (maildir_server *server)
- void maildir_server_free (maildir_server *server)
- bool maildir_server_is_self (maildir_server *server, bool *res, err_t *error)
- bool maildir_server_domain (maildir_server *server, char **domain, err_t *error)
- bool maildir_server_create_user (maildir_server *server, maildir_user *user, const char *username, err_t *error)
- bool maildir_server_user (maildir_server *server, maildir_user *user, const char *username, err_t *error)

2.5.5.1 Типы

2.5.5.1.1 maildir

typedef struct d_maildir maildir

2.5.5.1.2 maildir_server

typedef struct d_maildir_server maildir_server

2.5.5.1.3 maildir_user

typedef struct d_maildir_user maildir_user

2.5.5.1.4 maildir_users_list

typedef struct d_maildir_users_list maildir_users_list

2.5.5.2 Функции

2.5.5.2.1 maildir_server_create_user()

```
bool maildir_server_create_user (
                maildir_server * server,
                maildir_user * user,
                const char * username,
                err_t * error )
```

2.5.5.2.2 maildir_server_default_init()

```
void maildir_server_default_init (
                maildir_server * server )
```

2.5.5.2.3 maildir_server_domain()

```
bool maildir_server_domain (
                maildir_server * server,
                char ** domain,
                err_t * error )
```

2.5.5.2.4 maildir_server_free()

```
void maildir_server_free (
                maildir_server * server )
```

2.5.5.2.5 maildir_server_is_self()

```
bool maildir_server_is_self (
                maildir_server * server,
                bool * res,
                err_t * error )
```

2.5.5.2.6 maildir_server_user()

```
bool maildir_server_user (
        maildir_server * server,
        maildir_user * user,
        const char * username,
        err_t * error )
```

2.5.5.2.7 pr_maildir_server_init()

```
bool pr_maildir_server_init (
        maildir_server * server,
        err_t * error )
```

2.5.6 Файл include/maildir/user.h

```
#include "vector_structures.h"
#include <stdbool.h>
#include <linux/limits.h>
#include <sys/queue.h>
```

Классы

- struct d_maildir_user
- struct d_maildir_users_entry

Определения типов

- typedef struct d_maildir_user maildir_user
- typedef struct d_maildir_users_entry maildir_users_entry
- typedef struct d_maildir_users_list maildir_users_list
- typedef struct d_maildir_message maildir_message
- typedef struct d_maildir_messages_list maildir_messages_list
- typedef struct d_maildir_server maildir_server

Функции

- LIST_HEAD (d_maildir_users_list, d_maildir_users_entry)
- void maildir_user_default_init (maildir_user *user)
- void maildir_user_free (maildir_user *user)
- bool maildir_user_login (maildir_user *user, char **login)
- bool maildir_user_server (maildir_user *user, maildir_server **server)
- bool maildir_user_create_message (maildir_user *user, maildir_message *message, char *sender_name, err_t *error)
- bool maildir_user_message_list (maildir_user *user, maildir_messages_list *msg_list, err_t *error)

2.5.6.1 Типы

2.5.6.1.1 maildir_message

typedef struct d_maildir_message maildir_message

2.5.6.1.2 maildir_messages_list

typedef struct d_maildir_messages_list maildir_messages_list

2.5.6.1.3 maildir_server

typedef struct d_maildir_server maildir_server

2.5.6.1.4 maildir_user

typedef struct d_maildir_user maildir_user

2.5.6.1.5 maildir_users_entry

typedef struct d_maildir_users_entry maildir_users_entry

2.5.6.1.6 maildir_users_list

typedef struct d_maildir_users_list maildir_users_list

2.5.6.2 Функции

2.5.6.2.1 LIST_HEAD()

LIST_HEAD (
d_maildir_users_list ,
d_maildir_users_entry)

2.5.6.2.2 maildir_user_create_message()

bool maildir_user_create_message (
maildir_user * user,
maildir_message * message,
char * sender_name,
err_t * error)

2.5.6.2.3 maildir_user_default_init()

void maildir_user_default_init (
maildir_user * user)

2.5.6.2.4 maildir_user_free()

void maildir_user_free (
maildir_user * user)

2.5.6.2.5 maildir_user_login()

```
bool maildir_user_login (
        maildir_user * user,
        char ** login )
```

2.5.6.2.6 maildir_user_message_list()

```
bool maildir_user_message_list (
        maildir_user * user,
        maildir_messages_list * msg_list,
        err_t * error )
```

2.5.6.2.7 maildir_user_server()

```
bool maildir_user_server (
        maildir_user * user,
        maildir_server ** server )
```

2.5.7 Файл include/maildir/user.h

```
#include "vector_structures.h"
#include <stdbool.h>
#include <linux/limits.h>
#include <sys/queue.h>
```

Классы

- struct d_maildir_user
- struct d_maildir_users_entry

Определения типов

- typedef struct d_maildir_user maildir_user
- typedef struct d_maildir_users_entry maildir_users_entry
- typedef struct d_maildir_users_list maildir_users_list
- typedef struct d_maildir_message maildir_message
- typedef struct d_maildir_messages_list maildir_messages_list
- typedef struct d_maildir_server maildir_server

Функции

- LIST_HEAD (d_maildir_users_list, d_maildir_users_entry)
- void maildir_user_default_init (maildir_user *user)
- void maildir_user_free (maildir_user *user)
- bool maildir_user_login (maildir_user *user, char **login)
- bool maildir_user_server (maildir_user *user, maildir_server **server)
- bool maildir_user_create_message (maildir_user *user, maildir_message *message, char *sender_name, err_t *error)
- bool maildir_user_message_list (maildir_user *user, maildir_messages_list *msg_list, err_t *error)

2.5.7.1 Типы

2.5.7.1.1 maildir_message

typedef struct d_maildir_message maildir_message

2.5.7.1.2 maildir_messages_list

typedef struct d_maildir_messages_list maildir_messages_list

2.5.7.1.3 maildir_server

typedef struct d_maildir_server maildir_server

2.5.7.1.4 maildir_user

typedef struct d_maildir_user maildir_user

2.5.7.1.5 maildir_users_entry

typedef struct d_maildir_users_entry maildir_users_entry

2.5.7.1.6 maildir_users_list

typedef struct d_maildir_users_list maildir_users_list

2.5.7.2 Функции

2.5.7.2.1 LIST_HEAD()

LIST_HEAD (
d_maildir_users_list ,
d_maildir_users_entry)

2.5.7.2.2 maildir_user_create_message()

bool maildir_user_create_message (
maildir_user * user,
maildir_message * message,
char * sender_name,
err_t * error)

2.5.7.2.3 maildir_user_default_init()

void maildir_user_default_init (
maildir_user * user)

2.5.7.2.4 maildir_user_free()

void maildir_user_free (
maildir_user * user)

2.5.7.2.5 maildir_user_login()

```
bool maildir_user_login (
        maildir_user * user,
        char ** login )
```

2.5.7.2.6 maildir_user_message_list()

```
bool maildir_user_message_list (
        maildir_user * user,
        maildir_messages_list * msg_list,
        err_t * error )
```

2.5.7.2.7 maildir_user_server()

```
bool maildir_user_server (
        maildir_user * user,
        maildir_server ** server )
```

2.6 Список источников и литературы

1. http://rfc.com.ru/rfc2821.htm
2. http://rfc.com.ru/rfc1123
3. https://www.protocols.ru/WP/rfc5322/
4. RFC 1035 DOMAIN NAMES — IMPLEMENTATION AND SPECIFICATION https://www.protocols.ru/WP/rfc1035/
5. dovecot maildir https://wiki.dovecot.org/MailLocation/Maildir
6. qmail maildir https://cr.yp.to/proto/maildir.html