Московский государственный технический университет им. Н.Э. Баумана

Курсовая работа

по курсу

«Протоколы вычислительных сетей»

Создание SMTP-сервера, обеспечивающего локальную доставку и добавление в очередь удаленной доставки.

студент Стройкова Ксения Александровна

группа ИУ7–39

Москва, 2014

Содержание

Введение

Целью работы является проектирование, реализация и тестирование почтового SMTP-сервера, поддерживающего команды протокола SMTP:

— HELO;

— MAIL;

— RCPT;

— DATA;

— QUIT;

Сервер должен обеспечивать локальную доставку и добавление в очередь удаленной доставки.

Согласно варианту 4 необходимо:

— использовать вызов poll

— использовать рабочие процессы

— журналирование производить в отдельном процессе

— проверять обратную зону днс

Решаемые конструкторские задачи:

— описание конечного автомата состояний протокола на основе его спецификации

— проектирование взаимодействия подсистем

— проектирование алгоритмов обработки соединений в одном потоке выполнения и их распределелением между потоками

Решаемые технологические задачи:

— описание процесса кодогенерации

— написание исходного кода и его документирование для создания программной документации

— создание сценария сборки ПО, создание записки, проведение тестов

— создание системы системного тестирования

1. Аналитический раздел

В данном разделе приведено описание решения всех аналитических задач.

1.1. Развязывание потоков выполнения сервера

По заданию в данной работе используются рабочие процессы. В [1] рассмотрены возможные схемы обслуживание клиентов:

1) Обслуживание по очереди в одном потоке
2) Многопоточный сервер (по одному потоку на клиента)
3) Вариация с пулом потоков или процессов
4) Использование select() или poll() в одном процессе
5) Опрос неблокирующих сокетов

В соответствии с поставленным заданием была выбрана следующая схема:

При запуске сервера создается определенное в конфигурации число рабочих процессов, т.е. процессы создаются статически один раз при запуске сервера и уничтожаются при его выключении. Это позволяет не тратить системные ресурсы на создание и уничножение процессов (в отличие от схемы с динамическим созданием процессов), а также предохраняет от чрезмерного потребления системных ресурсов (в отличие от схем, где на каждого клиента создается отдельный поток или процесс).

Каждый процесс имеет свою очередь обслуживаемых подключений, все сокеты подключений переведены в неблокирующий режим, это позволяет тратить процессу минимум времени на обсуживание одного подключения и не блокироваться при ожидании сообщений, таким образом обслуживая много клиентов единовременно.

В цикле обслуживания каждый процесс использует в соответствии с заданием вызов poll(), что позволяет прослушивать одновременно все подключения процесса и не тратить ресурсы на их циклический опрос.

1.2. Разделение нагрузки между рабочими процессами

Каждый рабочий процесс в цикле осуществляет вызов poll() и при возврате проверяет listen сокет, если на нем есть новое подключение, процесс пытается принять его вызовом accept(). Таким образом один из рабочих процессов принимает подключение, а остальные получают код ошибки и продолжают обслуживание своей очереди заявок, т.к. все сокеты неблокирующие. Таким образом рабочий процесс выбирается операционной системой из тех, что осуществляют в момент появления подключения вызов poll().

1.3. Сущности предметной области

Выделение сущностей предметной области представлено на рис. 1.1

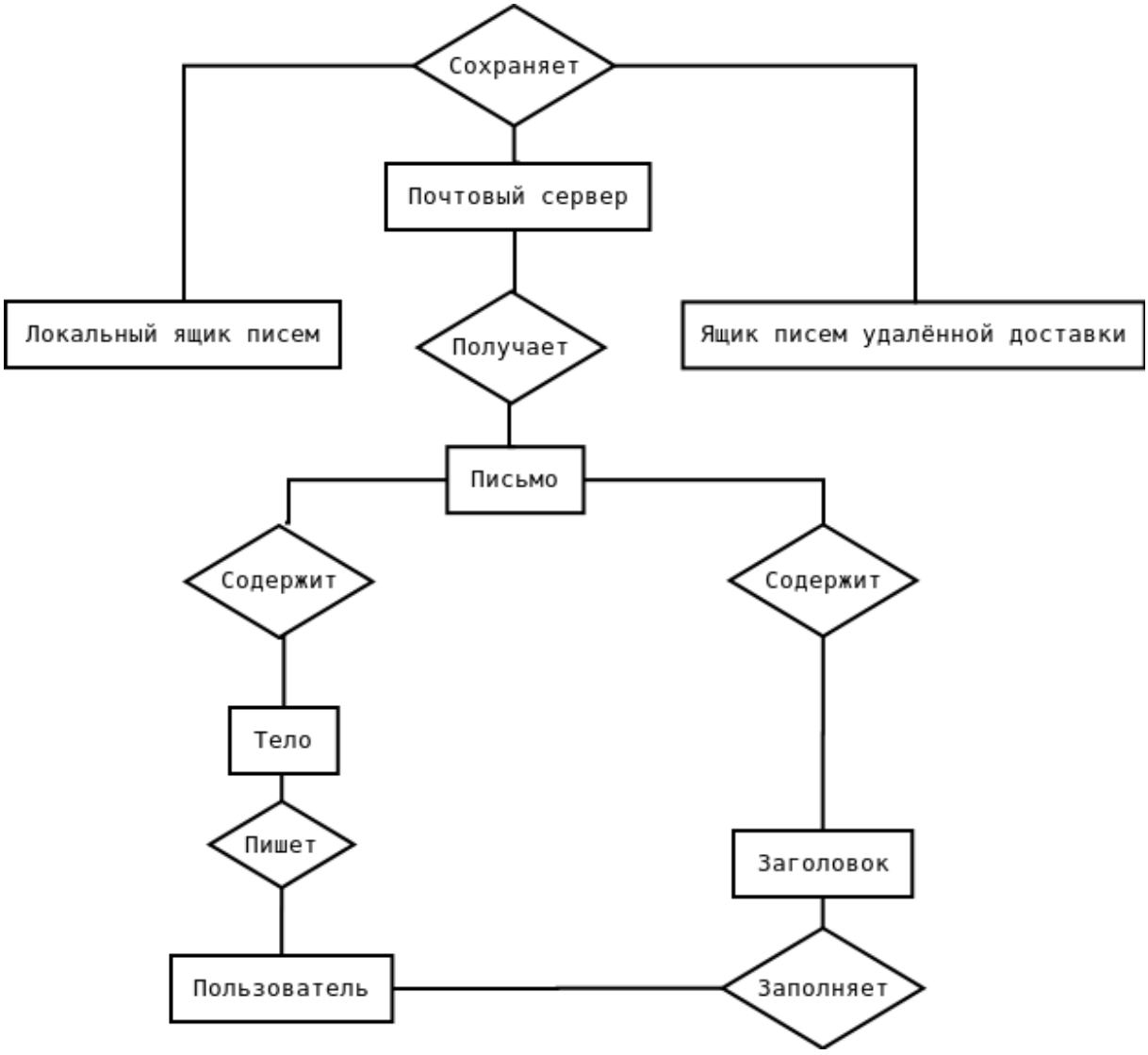


Рисунок 1.1. Сущности предметной области

2. Конструкторский раздел

В данном разделе приведено описание решения всех конструкторских задач.

2.1. Конечный автомат состояний сервера

На рис. 2.1 приведен конечный автомат, описывающий состояния SMTP-сервера.

Состояния автомата:

1) **START** – начальное состояние автомата

2) **INITIALIZED** – клиенту отправлена строка приветствия, состояние готовности приема команд, все буферы очищены

3) **MAIL_FROM_ENTERED** – от клиента получена информация об отправителе почты

4) **RCPT_TO_ENTERED** – от клиента получена информация о получателе почты

5) **DATA_ENTERED** – состояние ожидания содержимого письма

6) **POINT_ENTERED** – от клиента получена команда завершения передачи содержимого письма

7) **END** – завершающее состояние, получена команда **QUIT** или произошел таймаут

2.2. Проектирование алгоритмов обработки соединений

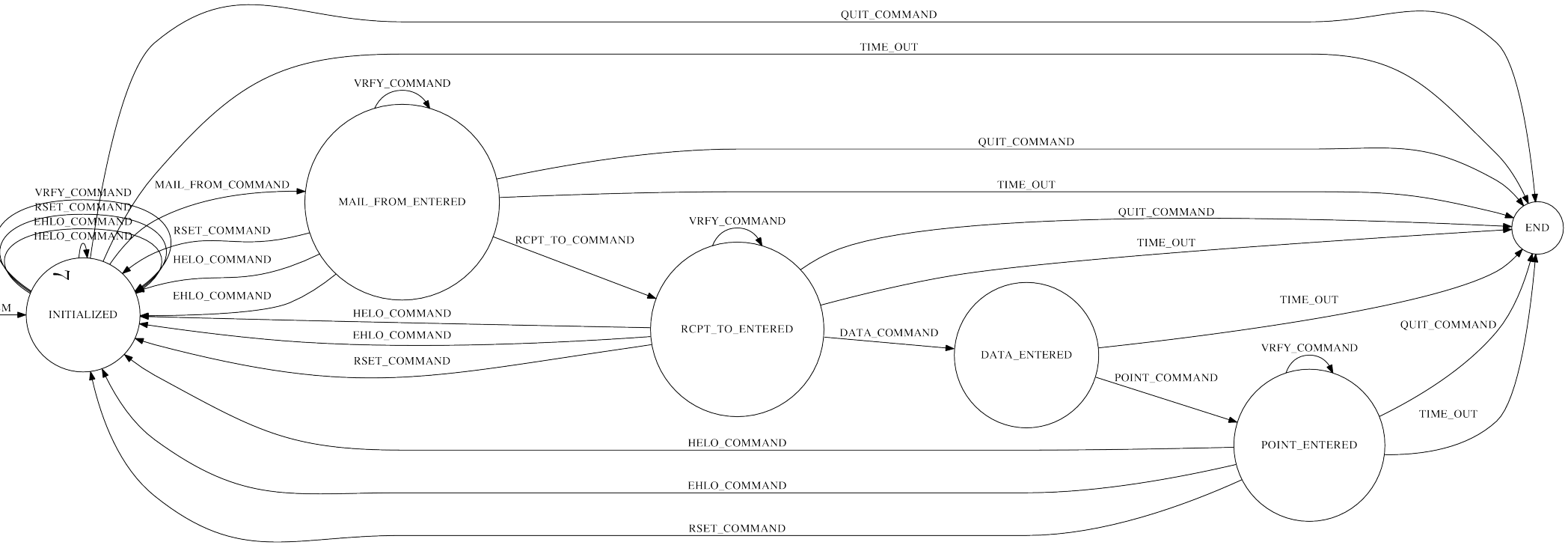
Сервер состоит из следующих процессов: основной процесс сервера, несколько рабочих процессов.

2.2.1. Логика работы основного процесса

Основной процесс системы инициализирует необходимые структуры данных, создает сокет для прослушивания и запускает остальные процессы. В листинге 2.1 приведен псевдокод, описывающий логику работы основного процесса сервера.

Листинг 2.1. Логика работы основного процесса

```
1 int main()
2 {
3     int result = read_config_file();
4     if (result)
5         return (EXIT_FAILURE);
6
7     result = mkdir_if_not_exists(mail_dir_path);
8     if (result)
```


QUIT_COMMAND
TIME_OUT
VRFY_COMMAND
QUIT_COMMAND
TIME_OUT
VRFY_COMMAND
RSET_COMMAND
EHLO_COMMAND
HELO_COMMAND
MAIL_FROM_COMMAND
RSET_COMMAND
HELO_COMMAND
EHLO_COMMAND
MAIL_FROM_ENTERED
RCPT_TO_COMMAND
VRFY_COMMAND
QUIT_COMMAND
TIME_OUT
END
INITIALIZED
HELO_COMMAND
EHLO_COMMAND
RSET_COMMAND
RCPT_TO_ENTERED
DATA_COMMAND
DATA_ENTERED
TIME_OUT
QUIT_COMMAND
VRFY_COMMAND
POINT_COMMAND
POINT_ENTERED
TIME_OUT
HELO_COMMAND
EHLO_COMMAND
RSET_COMMAND

```
9           fprintf(stderr, "mkdir_if_not_exists failed: %s\n", mail_dir_path);
10
11      Logger logger = Logger_create(log_file_name);
12      log_info(logger, "Server started\n");
13
14      char buffer[1024];
15      char buffer_temp[1024];
16      int rc;
17      int len;
18      int new_sd = -1;
19      int end_server = 0, compress_array = 0;
20      int close_conn;
21      int timeout;
22      int listen_sd = create_socket();
23      int nfds = 1, current_size = 0, i, j;
24
25      struct contexts head;
26
27      LIST_INIT(&head.list);
28
29      int fork_index;
30      for (fork_index = 0; fork_index < workers_count; ++fork_index){
31          int pid = fork();
32          switch(pid) {
33          case -1:
34              perror("fork");
35              return 1;
36          case 0:{
37              struct pollfd fds[200];
38              /*Initialize the pollfd structure*/
39              memset(fds, 0 , sizeof(fds));
40
41              /*Set up the initial listening socket*/
42              fds[0].fd = listen_sd;
43              fds[0].events = POLLIN;
44
45              /*Initialize the timeout to 3 minutes. If no
46              activity after 3 minutes this program will end
47              timeout value is based on milliseconds*/
48              timeout = (3 * 60 * 1000);
49              srand(getpid());
50              /*Loop waiting for incoming connects or for incoming data
51              on any of the connected sockets*/
52              do{
53                  /*Call poll() and wait 3 minutes for it to complete.*/
54                  printf("Waiting on poll()...\n");
55                  rc = poll(fds, nfds, timeout);
```

```
56                  ...
57              } while (end_server == FALSE); /* End of serving running.*/
58
59              /* Clean up all of the sockets that are open*/
60              for (i = 0; i < nfds; i++){
61                  if(fds[i].fd >= 0)
62                      close(fds[i].fd);
63              }
64          }
65
66          default:
67              continue;
68          }
69      }
70
71      while(1);
72      return 0;
73  }
```

2.2.2. Логика работы рабочего процесса

Рабочий процесс получает дескриптор сокета для прослушивания и запускает цикл обслуживания. В листинге 2.2 приведен псевдокод, описывающий логику работы рабочего процесса.

Листинг 2.2. Логика работы рабочего процесса

```
1  do{
2                  /*Call poll() and wait 3 minutes for it to complete.*/
3                  printf("Waiting on poll()...\n");
4                  rc = poll(fds, nfds, timeout);
5                  /*Check to see if the poll call failed.*/
6                  if (rc < 0){
7                      perror("  poll() failed");
8                      break;
9                  }
10
11                 /*Check to see if the 3 minute time out expired.*/
12                 if (rc == 0){
13                     printf("  poll() timed out.  End program.\n");
14                     break;
15                 }
16
17                 /*One or more descriptors are readable.  Need to
18                 determine which ones they are.*/
19                 current_size = nfds;
20                 for (i = 0; i < current_size; i++){
```

```
21                  /*Loop through to find the descriptors that returned
22                  POLLIN and determine whether it's the listening
23                  or the active connection.*/
24                  if(fds[i].revents == 0)
25                      continue;
26
27                  /*If revents is not POLLIN, it's an unexpected result,
28                  log and end the server.*/
29                  if(fds[i].revents != POLLIN){
30                      printf("  Error! revents = %d\n", fds[i].revents);
31                      end_server = TRUE;
32                      break;
33                  }
34
35                  if (fds[i].fd == listen_sd){
36                      /*Listening descriptor is readable.*/
37                      printf("  Listening socket is readable\n");
38
39                      /*Accept all incoming connections that are
40                      queued up on the listening socket before we
41                      loop back and call poll again.*/
42                      do{
43                          /*Accept each incoming connection. If
44                          accept fails with EWOULDBLOCK, then we
45                          have accepted all of them. Any other
46                          failure on accept will cause us to end the
47                          server.*/
48                          new_sd = accept(listen_sd, NULL, NULL);
49                          ...
50
51                          /*Loop back up and accept another incoming
52                          connection*/
53                      } while (new_sd != -1);
54                  }
55
56                  /* This is not the listening socket, therefore an
57                  existing connection must be readable*/
58
59                  else{
60                      printf("  Descriptor %d is readable\n", fds[i].fd);
61                      close_conn = FALSE;
62
63                      /* Receive all incoming data on this socket
64                      before we loop back and call poll again.*/
65                      struct context *c = get_context_from_list(&head,
                            fds[i].fd);
66
```

```
67                      assert(c);
68                      assert(c->fsm);
69
70                      rc = -1;
71                      do{
72                          /*Receive data on this connection until the
73                          recv fails with EWOULDBLOCK. If any other
74                          failure occurs, we will close the
75                          connection.*/
76                          rc = recv(fds[i].fd, buffer, sizeof(buffer),
                                MSG_DONTWAIT);
77                          if (rc < 0){
78                              if (errno != EWOULDBLOCK){
79                                  perror("  recv() failed");
80                                  close_conn = TRUE;
81                              }
82                              printf("broken\n");
83                              break;
84                          }
85
86                          /*Check to see if the connection has been
87                          closed by the client*/
88                          if (rc == 0){
89                              printf("  Connection closed\n");
90                              close_conn = TRUE;
91                              break;
92                          }
93
94                          /*Data was received*/
95                          len = rc;
96                          ...
97                      } while(rc > 0);
98
99
100
101                     /* If the close_conn flag was turned on, we need
102                     to clean up this active connection. This
103                     clean up process includes removing the
104                     descriptor.*/
105                     if (close_conn){
106                         close(fds[i].fd);
107                         fds[i].fd = -1;
108                         compress_array = TRUE;
109                     }
110
111
112                 }  /* End of existing connection is readable*/
```

```
113                 } /* End of loop through pollable descriptors*/
114
115                 /* If the compress_array flag was turned on, we need
116                 to squeeze together the array and decrement the number
117                 of file descriptors. We do not need to move back the
118                 events and revents fields because the events will always
119                 be POLLIN in this case, and revents is output.*/
120                 if (compress_array){
121                     compress_array = FALSE;
122                     for (i = 0; i < nfds; i++){
123                         if (fds[i].fd == -1){
124                             for(j = i; j < nfds; j++)
125                             fds[j].fd = fds[j+1].fd;
126                             nfds--;
127                         }
128                     }
129                 }
130
131             } while (end_server == FALSE); /* End of serving running.*/
```

2.3. Выбор и описание основных используемых структур данных

2.3.1. Массив контекстов обслуживаемых подключений

Для хранения состояния подключений была реализована структура данных **struct context**.

```
struct context{
    int socket;
    char *mail_from;
    char *rcpt_to;
    char *data;
    char *buffer;
    struct fsm_struct *fsm;
    LIST_ENTRY(context) entries;
};
```

Контекст описывает структуру данных, содержащую дескриптор сокета, состояние конечного автомата, принятые от клиента данные. Для хранения списка контекстов подключений была реализована структура данных **struct contexts**, использующая библиотеку **<sys/queue.h>**.

```
struct contexts {
    LIST_HEAD(context_list, context) list;
};
```

2.3.2. Способ хранения писем для локальных пользователей и для очереди отправки

В корневом каталоге Maildir создаются подкаталоги для почты локальных пользователей и подкаталог «outgoing» для хранения писем, предназначенных для удаленной доставки. Для локальных пользователей именем подкалалога является имя пользователя.

2.3.2.1. Структура подкаталога для хранения писем

— «cur» - содержит просмотренную почту;

— «new» - содержит новую почту;

— «tmp» - содержит временные файлы.

2.3.2.2. Структура письма

Имя файла письма составляется из почтового адреса получателя, времени получения письма и случайного числа.

Файл письма состоит из шапки письма, содержащей поля From и To, и тела письма.

3. Технологический раздел

Раздел содержит описание решения технологических задач.

3.1. Описания файла конфигурации, параметров командной строки

Рассмотрим минимальный список поддерживаемых параметров командной строки при запуске SMTP-сервера:

MailDirPath — корневой каталог для сообщений в формате Maildir;

LogFileName — имя файла журнала (лог);

WorkersCount — число рабочих процессов;

Port — порт привязки сервера SMTP;

DefaultFolder — имя папки для хранения почты для удаленной доставки;

3.2. Требования к системе

Ниже указаны версии основных использованных утилит и библиотек:

— GNU C Compiler 4.5.3.

— GNU Make 3.81.

— cfsm

3.3. Тестирование программы

Далее приведены описание и результаты системного тестирования.

3.3.1. Результаты системного тестирования

3.3.1.1. Успешная отправка письма

```
receive:
rc = 20
sock = 3
220 Service ready
Send:
HELO
receive:
250 OK
Send:
MAIL FROM: rcpt@domain.name
250 OK
Send:
```

```
RCPT TO: rcpt@domain.name
250 OK
Send:
DATA
250 OK
Send:
123
250 OK
Send:
123
250 OK
Send:
.
250 OK
Send:
QUIT
250 OK
```

Заключение

В рамках курсового проекта были достигнуты следующие цели:

— описан конечный автомат состояний протокола на основе его спецификации и с учётом отслеживания таймаутов;

— спроектировано взаимодействия подсистем;

— спроектированы алгоритмы обработки соединений в одном потоке выполнения и их распределелеление между потоками;

— написан исходный код программы;

— создан сценарий сборки системы, создана записка, проведены тесты;

— созданы системы системного тестирования.

Основным результатом работы является реализация SMTP-сервера, обеспечивающего локальную доставку и добавление в очередь удаленной доставки.

Список использованных источников

1. *В.А. Крищенко, А.О. Крючков.* Стек сетевых протоколов TCP/IP: теория и практика компьютерных сетей / А.О. Крючков В.А. Крищенко. — 2012.