

File scheduler

Generated by Doxygen 1.8.14

Thu Apr 12 2018 01:18:19

Contents

1	PYSSC scheduler.	2
2	Data Structure Index	2
2.1	Data Structures	2
3	File Index	2
3.1	File List	2
4	Data Structure Documentation	3
4.1	client_buffer Struct Reference	3
4.1.1	Detailed Description	3
4.1.2	Field Documentation	4
4.2	fd_struct Struct Reference	5
4.2.1	Detailed Description	5
4.2.2	Field Documentation	5
4.3	read_add Struct Reference	6
4.3.1	Detailed Description	6
4.3.2	Field Documentation	6
4.4	thread_data Struct Reference	7
4.4.1	Detailed Description	7
4.4.2	Field Documentation	8
5	File Documentation	8
5.1	file_scheduler.cpp File Reference	8
5.1.1	Macro Definition Documentation	9
5.1.2	Function Documentation	9
5.1.3	Variable Documentation	22
5.2	file_scheduler.cpp	23
5.3	README.md File Reference	33
5.4	README.md	33

[Index](#)

35

1 PYSSC scheduler.

This server is used to synchronize pssc workers that run at different nodes.

Main problem is that each of them have to know whether to generate 'reuse' file read it or wait while it is being generated by other worker(node).

This server recieves data about requested file and respond with 'suggestion': READ, WRIT, WAIT.

In short. If file does not exist and is not being generated - generate it (WRIT). If file exists - read it (READ). Read operation does not change the data, so can be done in parallel, anyway file will be cached into RAM. If file is being generated - WAIT for a next READ message.

Communication is done by epoll. It is possible to add any reasonable number of threads if needed, but for the environment it was codded for - two threads is more than enough. First thread accepts connections, second communicates with clients.

This server should be launched on one of the nodes. Other clients should know server's ip. Because of the asynchronous design, it produces very little overhead. When received SIGINT - all connection should be closed and program terminated. Check PYSSC git for a client version. Although this server was tested with many threads and for a long time, it may still have some error or space for improvement. I would be glad to hear any response.

Code sucessfully passedd PVS-Studio and Valgrind check.

2 Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

client_buffer	3
fd_struct	5
read_add	6
thread_data	7

3 File Index

3.1 File List

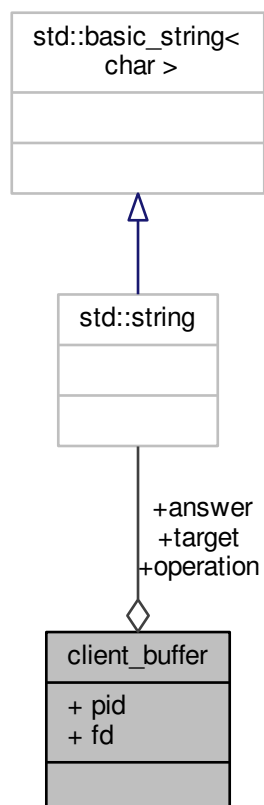
Here is a list of all files with brief descriptions:

file_scheduler.cpp	8
------------------------------------	---

4 Data Structure Documentation

4.1 client_buffer Struct Reference

Collaboration diagram for client_buffer:



Data Fields

- int `pid`
- int `fd`
- string `operation`
- string `target`
- string `answer`

4.1.1 Detailed Description

Definition at line 92 of file `file_scheduler.cpp`.

4.1.2 Field Documentation

4.1.2.1 answer

```
string client_buffer::answer
```

Definition at line 98 of file [file_scheduler.cpp](#).

4.1.2.2 fd

```
int client_buffer::fd
```

Definition at line 95 of file [file_scheduler.cpp](#).

4.1.2.3 operation

```
string client_buffer::operation
```

Definition at line 96 of file [file_scheduler.cpp](#).

4.1.2.4 pid

```
int client_buffer::pid
```

Definition at line 94 of file [file_scheduler.cpp](#).

4.1.2.5 target

```
string client_buffer::target
```

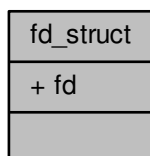
Definition at line 97 of file [file_scheduler.cpp](#).

The documentation for this struct was generated from the following file:

- [file_scheduler.cpp](#)

4.2 fd_struct Struct Reference

Collaboration diagram for fd_struct:



Data Fields

- int [fd](#)

4.2.1 Detailed Description

Definition at line [82](#) of file [file_scheduler.cpp](#).

4.2.2 Field Documentation

4.2.2.1 fd

```
int fd_struct::fd
```

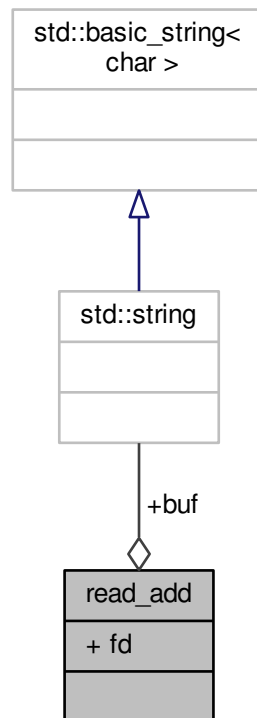
Definition at line [84](#) of file [file_scheduler.cpp](#).

The documentation for this struct was generated from the following file:

- [file_scheduler.cpp](#)

4.3 read_add Struct Reference

Collaboration diagram for read_add:



Data Fields

- int [fd](#)
- string [buf](#)

4.3.1 Detailed Description

Definition at line [101](#) of file [file_scheduler.cpp](#).

4.3.2 Field Documentation

4.3.2.1 buf

```
string read_add::buf
```

Definition at line [104](#) of file [file_scheduler.cpp](#).

4.3.2.2 fd

```
int read_add::fd
```

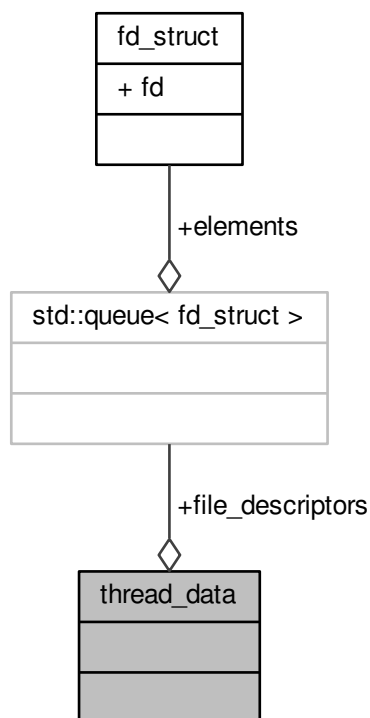
Definition at line 103 of file [file_scheduler.cpp](#).

The documentation for this struct was generated from the following file:

- [file_scheduler.cpp](#)

4.4 thread_data Struct Reference

Collaboration diagram for thread_data:



Data Fields

- `queue< fd_struct > file_descriptors`

4.4.1 Detailed Description

Definition at line 87 of file [file_scheduler.cpp](#).

4.4.2 Field Documentation

4.4.2.1 file_descriptors

`queue<fd_struct> thread_data::file_descriptors`

Definition at line 89 of file [file_scheduler.cpp](#).

The documentation for this struct was generated from the following file:

- [file_scheduler.cpp](#)

5 File Documentation

5.1 file_scheduler.cpp File Reference

```
#include <fcntl.h>
#include <netinet/in.h>
#include <pthread.h>
#include <stddef.h>
#include <string.h>
#include <sys/epoll.h>
#include <sys/socket.h>
#include <unistd.h>
#include <cerrno>
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <iterator>
#include <string>
#include <vector>
#include <sstream>
#include <queue>
#include <deque>
#include <fstream>
#include <ctime>
#include <iomanip>
#include <arpa/inet.h>
#include <csignal>
```

Include dependency graph for `file_scheduler.cpp`:



Data Structures

- struct [fd_struct](#)
- struct [thread_data](#)
- struct [client_buffer](#)
- struct [read_add](#)

Macros

- `#define MAXEVENTS 500`

Functions

- `size_t parse_buffer` (string str, deque< `client_buffer` > *client_buf, int fd)
- `int secure_send` (`client_buffer` *client_buf)
- `void check_client_errors` (deque< `client_buffer` > *processed_client_buf, deque< `client_buffer` > *client_buf, const ssize_t fd)
- `void * read_and_respond` (void *threadarg)
- `int accept_connections` (uint16_t port, queue< `fd_struct` > *clients)
- `void signalHandler` (int signum)
- `int main` ()

Variables

- `pthread_mutex_t lock`
- `bool time_to_exit` = false
- `int exit_code` = 0

5.1.1 Macro Definition Documentation

5.1.1.1 MAXEVENTS

```
#define MAXEVENTS 500
```

Definition at line 79 of file `file_scheduler.cpp`.

5.1.2 Function Documentation

5.1.2.1 accept_connections()

```
int accept_connections (
    uint16_t port,
    queue< fd_struct > * clients )
```

Initializes socket on port 'port' and waits for connections. On incoming connection sets associated socket to async mode and stores in `fd_struct` structure which is shared with processing thread.

Parameters

in	<i>port</i>	Port used to create listening socket.
in	<i>clients</i>	queue that stores file descriptors(sockets) accepted on port 'port'.

Returns

status code

Definition at line 668 of file [file_scheduler.cpp](#).

```

00669 {
00670     std::string rcv;
00671     int listen_fd, comm_fd;
00672     struct sockaddr_in servaddr;
00673     std::ofstream log_main;
00674     log_main.open("incoming.log", std::ios::out | std::ios::app);
00675     #ifdef DEBUG
00676         auto t = time(nullptr);
00677         auto tm = *localtime(&t);
00678         log_main << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Thread created\n";
00679     #endif
00680
00681     listen_fd = socket(AF_INET, SOCK_STREAM, 0);
00682
00683     if (listen_fd == -1)
00684     {
00685         cout << "Can't create file descriptor." << endl;
00686         exit_code = 1;
00687         time_to_exit = true;
00688         sleep(10);
00689         exit(1);
00690     }
00691
00692     memset(&servaddr, 0, sizeof(servaddr));
00693     servaddr.sin_family = AF_INET;
00694     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
00695     servaddr.sin_port = htons(port);
00696     #ifdef DEBUG
00697         t = time(nullptr);
00698         tm = *localtime(&t);
00699         log_main << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Attempting to listen on " << port << " port\n";
00700     #endif
00701     int my_timer = 20;
00702
00703     while(my_timer > 0)
00704     {
00705         if(bind(listen_fd, (struct sockaddr *)&servaddr, sizeof(servaddr)) < 0)
00706             sleep(10);
00707         else
00708             break;
00709         my_timer--;
00710     }
00711
00712     if(my_timer == 0)
00713     {
00714         cout << "Binding to socket error." << endl;
00715         exit_code = 1;
00716         time_to_exit = true;
00717         sleep(10);
00718         exit(2);
00719     }
00720     #ifdef DEBUG
00721         t = time(nullptr);
00722         tm = *localtime(&t);
00723         log_main << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Successful bind to the socket\n";
00724     #endif
00725
00726     fd_struct temp;
00727
00728     while(!time_to_exit)
00729     {
00730         listen(listen_fd, 60);
00731         #ifdef DEBUG
00732             t = time(nullptr);
00733             tm = *localtime(&t);
00734             log_main << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Waiting for incoming connections.\n";
00735             log_main.flush();
00736         #endif
00737         sockaddr_in clientAddr;
00738         socklen_t sin_size=sizeof(struct sockaddr_in);
00739         comm_fd = accept(listen_fd, (struct sockaddr*)&clientAddr, &sin_size);
00740
00741         if(comm_fd == -1)
00742         {
00743             cout << "Connection acceptance error." << endl;
00744             // exit(3);
00745         }

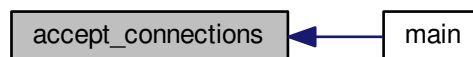
```

```

00746
00747     #ifdef DEBUG
00748         t = time(nullptr);
00749         tm = *localtime(&t);
00750         char loc_addr[INET_ADDRSTRLEN+1];
00751         inet_ntop(AF_INET, &(clientAddr.sin_addr), loc_addr, INET_ADDRSTRLEN);
00752         log_main << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Incoming connenction on descriptor " <<
comm_fd << " from " << loc_addr << ":" << clientAddr.sin_port << "\n";
00753     #endif
00754     // make nonblocking
00755     auto flags = fcntl (comm_fd, F_GETFL, 0);
00756     if (flags < 0)
00757     {
00758         perror ("fcntl");
00759         return -1;
00760     }
00761
00762     flags |= O_NONBLOCK;
00763     auto s = fcntl (comm_fd, F_SETFL, flags);
00764
00765     if (s < 0)
00766     {
00767         perror ("fcntl");
00768         time_to_exit = true;
00769         log_main.close();
00770         return -1;
00771     }
00772     #ifdef DEBUG
00773         t = time(nullptr);
00774         tm = *localtime(&t);
00775         log_main << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Switched " << comm_fd << " to nonblocking
mode.\n";
00776     #endif
00777
00778     temp.fd = comm_fd;
00779
00780     pthread_mutex_lock(&lock);
00781     clients->push(temp);
00782     pthread_mutex_unlock(&lock);
00783     #ifdef DEBUG
00784         t = time(nullptr);
00785         tm = *localtime(&t);
00786         log_main << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Successfully pushed into queue for
further processing.\n";
00787     #endif
00788 }
00789
00790 #ifdef DEBUG
00791 log_main.close();
00792 #endif
00793
00794 return 0;
00795 }

```

Here is the caller graph for this function:



5.1.2.2 check_client_errors()

```

void check_client_errors (
    deque< client_buffer > * processed_client_buf,
    deque< client_buffer > * client_buf,
    const ssize_t fd )

```

Definition at line 208 of file [file_scheduler.cpp](#).

```

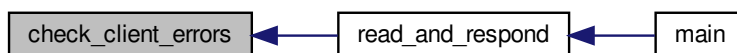
00209 {
00210     string error_target = "";
00211
00212     for(auto iter = client_buf->begin(); iter != client_buf->end(); ++iter)
00213     {
00214         if(iter->fd == fd)
00215             return;
00216     }
00217
00218     for(auto iter = processed_client_buf->begin(); iter != processed_client_buf->end(); ++iter)
00219     {
00220         if(iter->fd == fd )
00221         {
00222             error_target = iter->target;
00223             cerr << "Broken client removing: " << iter->fd << " " << iter->operation << " " << iter->target
00224             << endl;
00225             processed_client_buf->erase(iter);
00226             break;
00227         }
00228         if(error_target.length() > 0)
00229         {
00230             for(auto iter = processed_client_buf->begin(); iter != processed_client_buf->end(); ++iter)
00231             {
00232                 if(iter->target == error_target && iter->answer == "WAIT")
00233                 {
00234                     iter->answer = "WRIT";
00235                     cerr << "PID " << iter->pid << " advised to WRIT";
00236                     if(secure_send(&*iter) != 0)
00237                         cerr << "ERROR in secure send";
00238                     break;
00239                 }
00240             }
00241         }
00242     }

```

Here is the call graph for this function:



Here is the caller graph for this function:



5.1.2.3 main()

```
int main ( )
```

Nothing fancy. Creates a thread and launches connection accepting function

Returns

status code to OS

Parameters

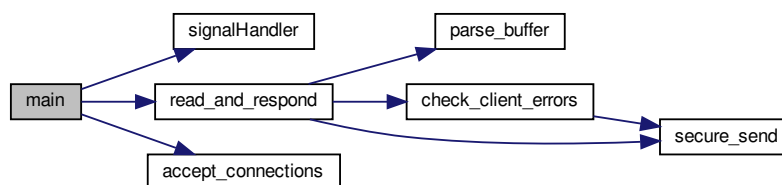
<i>clients</i>	data structure to store fd
----------------	----------------------------

Definition at line 802 of file [file_scheduler.cpp](#).

```

00803 {
00804     queue <fd_struct> clients;
00805     pthread_t threads[1];
00806
00807     signal(SIGINT, signalHandler);
00808
00809     if (pthread_mutex_init(&lock, NULL) != 0)
00810     {
00811         printf("\n mutex init failed\n");
00812         return 1;
00813     }
00814
00815     int rc = pthread_create(&threads[0], NULL, read_and_respond, (void *)&clients);
00816     if (rc)
00817     {
00818         cout << "Error:unable to create thread," << rc << endl;
00819         exit_code = -1;
00820         time_to_exit = true;
00821         sleep(10);
00822         return exit_code;
00823     }
00824
00825     accept_connections(1987, &clients);
00826
00827     pthread_join(rc, NULL);
00828     pthread_mutex_destroy(&lock);
00829
00830     return exit_code;
00831 }
00832 }
```

Here is the call graph for this function:



5.1.2.4 parse_buffer()

```

size_t parse_buffer (
    string str,
    deque< client_buffer > * client_buf,
    int fd )
```

Parses input buffer and stores parsed messages in queue It may parse more than one message(stored in str) and if last message is incomplete - returns how many characters to save in external buffer for future processing.

Parameters

in	<i>str</i>	Input buffer with data recieved in socket fd.
in	<i>client_buf</i>	Structure that keeps all parsed messages.
in	<i>fd</i>	file descriptor associated with passed buffer data.

Returns

how many characters to save in external buffer for future processing

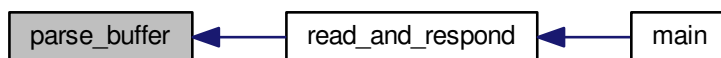
Definition at line 136 of file [file_scheduler.cpp](#).

```

00137 {
00138     bool done = false;
00139     bool error = false;
00140     client_buffer temp;
00141     vector <string> info_block;
00142     string current_str;
00143     size_t found = 0;
00144     string token;
00145
00146     while(str.length() > 0 && !done && !error)
00147     {
00148         found = str.find_first_of("#");
00149         if(found == std::string::npos)
00150         {
00151             error = true;
00152             continue;
00153         }
00154
00155         unsigned fut_len = stoi(str.substr(0, found));
00156         if(str.length() < found + 1 + fut_len)
00157             break;
00158
00159         current_str = str.substr(found+1, fut_len);
00160         str = str.substr(found + 1 + fut_len);
00161
00162         std::istringstream iss(current_str);
00163         info_block.clear();
00164         while (getline(iss, token, '#'))
00165             info_block.push_back(token);
00166
00167         current_str.clear();
00168
00169         temp = {};
00170         if(info_block.size() > 0 )
00171             temp.pid = stoi(info_block[0]);
00172
00173         if(info_block.size() > 1 )
00174             temp.operation = info_block[1];
00175
00176         if(info_block.size() > 2 )
00177             temp.target = info_block[2];
00178
00179         temp.fd = fd;
00180
00181         if(temp.operation == "DONE")
00182             client_buf->push_front(temp);
00183         else
00184             client_buf->push_back(temp);
00185
00186         if(str.length() < 3)
00187             done = true;
00188     }
00189
00190     return str.length();
00191 }

```

Here is the caller graph for this function:



5.1.2.5 read_and_respond()

```
void * read_and_respond (
    void * threadarg )
```

Registers new sockets in epoll function(waits on data in async mode). Reads data from sockets in async mode. Calls parse function and makes decision according to the processed requests.

Parameters

in	<i>threadarg</i>	Structure that contains address of queue with file descriptors.
----	------------------	---

Definition at line 248 of file [file_scheduler.cpp](#).

```

00249 {
00250     auto my_data = (thread_data *) threadarg;
00251     queue <fd_struct> *fds = &my_data->file_descriptors;
00252     struct epoll_event event;
00253     struct epoll_event *events;
00254     auto efd = epoll_create1 (0);
00255     vector <int> fd_to_remove;
00256     // vector <size_t> local_fd;
00257     deque <client_buffer> client_buf;
00258     deque <client_buffer> processed_client_buf;
00259     vector <read_add> buff_add;
00260     read_add ra;
00261     events = (epoll_event*)calloc (MAXEVENTS, sizeof event);
00262     event.events = EPOLLIN | EPOLLET;
00263     size_t fd_event_counter = 0;
00264     int n;
00265     std::ofstream log_processing;
00266     log_processing.open("processing.log", std::ios::out | std::ios::app);
00267     #ifdef DEBUG
00268         auto t = time(nullptr);
00269         auto tm = *localtime(&t);
00270         log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Thread created\n";
00271     #endif
00272     while(!time_to_exit)
00273     {
00274         #ifdef DEBUG
00275             if(fd_to_remove.size() > 0)
00276             {
00277                 t = time(nullptr);
00278                 tm = *localtime(&t);
00279                 log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Descriptors to clean: " <<
00280 fd_to_remove.size() << "\n";
00281             }
00282             #endif
00283             for(unsigned j=0; j < fd_to_remove.size(); ++j)
00284             {
00285                 for(size_t k = 0; k < buff_add.size(); ++k)
00286                 {

```



```

00287         if(buff_add[k].fd == fd_to_remove[j])
00288         {
00289             #ifdef DEBUG
00290                 t = time(nullptr);
00291                 tm = *localtime(&t);
00292                 log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Removing " <<
fd_to_remove[j] << endl;
00293             #endif
00294             buff_add.erase(buff_add.begin() + k);
00295             break;
00296         }
00297     }
00298
00299     for(auto iter = processed_client_buf.begin(); iter != processed_client_buf.end(); )
00300     {
00301         if(iter->fd == fd_to_remove[j])
00302             iter = processed_client_buf.erase(iter);
00303         else
00304             iter++;
00305     }
00306
00307 }
00308
00309 fd_event_counter -= fd_to_remove.size();
00310 fd_to_remove.clear();
00311
00312 #ifdef DEBUG
00313     if(!fds->empty())
00314     {
00315         t = time(nullptr);
00316         tm = *localtime(&t);
00317         log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Processing new connections\n";
00318         if(!fds->empty())
00319             log_processing.seekp(-1, std::ios_base::cur);
00320     }
00321 #endif
00322
00323 while(!fds->empty())
00324 {
00325     pthread_mutex_lock(&lock);
00326     event.data.fd = fds->front().fd;
00327     fds->pop();
00328     pthread_mutex_unlock(&lock);
00329     ++fd_event_counter;
00330     #ifdef DEBUG
00331         log_processing << ".";
00332     #endif
00333
00334     if( epoll_ctl(efd, EPOLL_CTL_ADD, event.data.fd, &event) == -1)
00335     {
00336         perror ("epoll_ctl");
00337         cerr << "EPOLL ERROR\n";
00338         time_to_exit = true;
00339         #ifdef DEBUG
00340             log_processing.seekp(0, std::ios_base::end);
00341             t = time(nullptr);
00342             tm = *localtime(&t);
00343             log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "EPOLL ERROR\n";
00344             log_processing.close();
00345         #endif
00346         pthread_exit(NULL);
00347     }
00348 }
00349
00350 #ifdef DEBUG
00351     log_processing.seekp(0, std::ios_base::end);
00352 #endif
00353
00354 if(fd_event_counter > 0)
00355 {
00356     n = epoll_wait(efd, events, MAXEVENTS, 1000);
00357     for(int i = 0; i < n; ++i)
00358     {
00359         if (( &events[i] != NULL) && ((events[i].events & EPOLLERR) || (events[i].events & EPOLLHUP
) || (!(events[i].events & EPOLLIN))))
00360         {
00361             cerr << "epoll error\n";
00362             close (events[i].data.fd);
00363             fd_to_remove.push_back(events[i].data.fd);
00364             continue;
00365         }
00366         else
00367         {
00368             int done = 0;
00369             ra.buf = "";
00370
00371             for(unsigned k = 0; k < buff_add.size(); ++k)

```

```

00372         {
00373             if(buff_add[k].fd == events[i].data.fd)
00374             {
00375                 ra.buf = buff_add[k].buf;
00376                 break;
00377             }
00378         }
00379         #ifdef DEBUG
00380         log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Reading from socket " <
< events[i].data.fd << endl;
00381         #endif
00382         while (1)
00383         {
00384             ssize_t count;
00385             char buf[512];
00386             memset(buf,0, sizeof buf);
00387             count = recv(events[i].data.fd, buf, sizeof buf, 0);
00388             #ifdef DEBUG
00389                 t = time(nullptr);
00390                 tm = *localtime(&t);
00391                 log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Recieved: " <<
string(buf) << endl;
00392             #endif
00393             if (count == -1)
00394             { // If errno == EAGAIN, that means we have read all data. So go back to the main
loop.
00395                 if (errno != EAGAIN)
00396                 {
00397                     perror ("read");
00398                     cerr << "Count error";
00399                     #ifdef DEBUG
00400                         t = time(nullptr);
00401                         tm = *localtime(&t);
00402                         log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "ERROR: Count
error in epoll" << endl;
00403                     #endif
00404                     done = 1;
00405                 }
00406                 break;
00407             }
00408             else if (count == 0)
00409             { // End of file. The remote has closed the connection.
00410                 done = 1;
00411                 break;
00412             }
00413             ra.buf += string(buf);
00414             #ifdef DEBUG
00415                 t = time(nullptr);
00416                 tm = *localtime(&t);
00417                 log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Parsing message of
length " << ra.buf.length() << endl;
00418                 log_processing << "Message: " << string(ra.buf) << endl;
00419             #endif
00420             size_t char_left = parse_buffer(ra.buf, &client_buf, events[i].data.
fd);
00421             #ifdef DEBUG
00422                 t = time(nullptr);
00423                 tm = *localtime(&t);
00424                 if(client_buf.size() > 0)
00425                 {
00426                     for(auto iter = client_buf.begin(); iter != client_buf.end(); ++iter)
00427                     {
00428                         log_processing << "PID " << iter->pid << " on " << iter->fd << " requested
" << iter->operation << " " << iter->target << endl;
00429                     }
00430                 }
00431                 else
00432                 {
00433                     log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "WARNING: Client
buffer is empty\n";
00434                 }
00435             #endif
00436             if(char_left > 0)
00437             {
00438                 #ifdef DEBUG
00439                     t = time(nullptr);
00440                     tm = *localtime(&t);
00441                     log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Not all
information was received. Missing " << char_left << " chars\n";
00442                 #endif
00443                 ra.fd = events[i].data.fd;
00444                 unsigned k = 0;
00445                 for(; k < buff_add.size(); ++k)
00446                 {
00447                     if(buff_add[k].fd == events[i].data.fd)
00448                     {
00449                         buff_add[k].buf += ra.buf;

```

```

00450             break;
00451         }
00452     }
00453     if(k == buff_add.size())
00454         buff_add.push_back(ra);
00455 }
00456 else
00457 {
00458     #ifdef DEBUG
00459         t = time(nullptr);
00460         tm = *localtime(&t);
00461         log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Success in
reading full message\n";
00462     #endif
00463     for(unsigned k = 0; k < buff_add.size(); ++k)
00464     {
00465         if(buff_add[k].fd == events[i].data.fd)
00466         {
00467             buff_add.erase(buff_add.begin() + k);
00468             break;
00469         }
00470     }
00471 }
00472 }
00473
00474 if (done)
00475 {
00476     #ifdef DEBUG
00477         t = time(nullptr);
00478         tm = *localtime(&t);
00479         log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Closing connection on
descriptor " << events[i].data.fd << endl;
00480     #endif
00481
00482     check_client_errors(&processed_client_buf, &client_buf, events[i
].data.fd);
00483
00484     close (events[i].data.fd); // Closing the descriptor will make epoll remove it from
the set of descriptors which are monitored.
00485     fd_to_remove.push_back(events[i].data.fd);
00486 }
00487 }
00488 }
00489
00490 #ifdef DEBUG
00491 if(!client_buf.empty())
00492 {
00493     t = time(nullptr);
00494     tm = *localtime(&t);
00495     log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Begin processing requests.\n";
00496 }
00497 #endif
00498
00499 #ifdef DEBUG
00500 log_processing << "processed_client_buf before processing new requests \n*****\n";
00501 for(auto iter = processed_client_buf.begin(); iter != processed_client_buf.end(); ++iter)
00502 {
00503     log_processing << "PID " << iter->pid << " from socket " << iter->fd << " requested " <<
iter->operation << " " << iter->target << " advised " << iter->answer << endl;
00504 }
00505 log_processing << "*****\n end\n";
00506 #endif
00507
00508 while(!client_buf.empty())
00509 {
00510     if(client_buf.front().operation == "READ")
00511     {
00512         #ifdef DEBUG
00513             t = time(nullptr);
00514             tm = *localtime(&t);
00515             log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "PID: " << client_buf.
front().pid << " from socket " << client_buf.front().fd << " wants to read " << client_buf.front().target <<
endl;
00516         #endif
00517         client_buf.front().answer = "READ";
00518
00519         for(auto iter = processed_client_buf.begin(); iter != processed_client_buf.end(); ++
iter)
00520         {
00521             if(iter->target == client_buf.front().target )
00522             {
00523                 if(iter->answer == "WRIT" || iter->answer == "WAIT")
00524                 {
00525                     client_buf.front().answer = "WAIT";
00526                     break;
00527                 }
00528                 else if(iter->answer == "READ")

```

```

00529         {
00530             client_buf.front().answer = "READ";
00531             break;
00532         }
00533     }
00534 }
00535
00536 if(secure_send(&client_buf.front()) == 0)
00537     processed_client_buf.push_back(client_buf.front());
00538 else
00539     cerr << "ERROR in secure send";
00540
00541 #ifdef DEBUG
00542     t = time(nullptr);
00543     tm = *localtime(&t);
00544     log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "RESPONSE to PID: " <<
client_buf.front().pid << " from socket " << client_buf.front().fd << ": " << client_buf.front().answer <<
endl;
00545 #endif
00546 }
00547 else if(client_buf.front().operation == "WRIT")
00548 {
00549     #ifdef DEBUG
00550         t = time(nullptr);
00551         tm = *localtime(&t);
00552         log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "PID: " << client_buf.
front().pid << " from socket " << client_buf.front().fd << " wants to write " << client_buf.front().target <<
endl;
00553     #endif
00554     client_buf.front().answer = "WRIT";
00555     for(auto iter = processed_client_buf.begin(); iter != processed_client_buf.end(); ++
iter)
00556     {
00557         if(iter->target == client_buf.front().target )
00558         {
00559             if(iter->answer == "WRIT" || iter->answer == "WAIT")
00560             {
00561                 client_buf.front().answer = "WAIT";
00562                 break;
00563             }
00564             else if(iter->answer == "READ")
00565             {
00566                 client_buf.front().answer = "READ";
00567                 break;
00568             }
00569         }
00570     }
00571
00572 if(secure_send(&client_buf.front()) == 0)
00573     processed_client_buf.push_back(client_buf.front());
00574 else
00575     cerr << "ERROR in secure send";
00576
00577 #ifdef DEBUG
00578     t = time(nullptr);
00579     tm = *localtime(&t);
00580     log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "RESPONSE to PID: " <<
client_buf.front().pid << " from socket " << client_buf.front().fd << ": " << client_buf.front().answer <<
endl;
00581 #endif
00582 }
00583 else if(client_buf.front().operation == "DONE")
00584 {
00585     #ifdef DEBUG
00586         t = time(nullptr);
00587         tm = *localtime(&t);
00588         log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "PID: " << client_buf.
front().pid << " from socket " << client_buf.front().fd << " finished working with " << client_buf.front().
target << endl;
00589     #endif
00590
00591     for(auto iter = processed_client_buf.begin(); iter != processed_client_buf.end(); )
00592     {
00593         if(iter->target == client_buf.front().target )
00594         {
00595             if(iter->pid == client_buf.front().pid)
00596             {
00597                 #ifdef DEBUG
00598                     t = time(nullptr);
00599                     tm = *localtime(&t);
00600                     log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "PID: " <<
iter->pid << " from socket " << iter->fd << ": is DONE - SELF-DESTRUCTION" << endl;
00601                 #endif
00602                 iter = processed_client_buf.erase(iter);
00603                 continue;
00604             }
00605             else if(iter->answer == "WAIT")

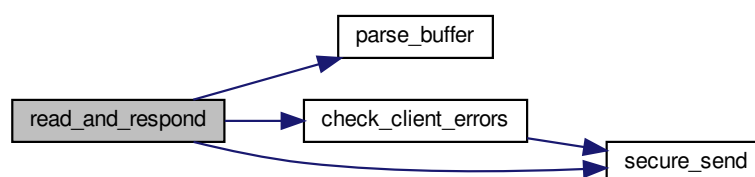
```

```

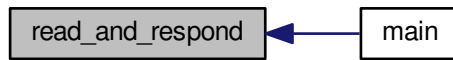
00606             {
00607                 iter->answer = "READ";
00608                 if(secure_send(&*iter) != 0)
00609                     cerr << "ERROR in secure send";
00610
00611                 #ifdef DEBUG
00612                     t = time(nullptr);
00613                     tm = *localtime(&t);
00614                     log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "RESPONSE to
PID: " << iter->pid << " from socket " << iter->fd << ": " << iter->answer << endl;
00615                     #endif
00616             }
00617         }
00618
00619         ++iter;
00620     }
00621 }
00622 else
00623     cerr << client_buf.front().operation << endl;
00624
00625     client_buf.pop_front();
00626 }
00627
00628 #ifdef DEBUG
00629 log_processing << "processed_client_buf after processing new requests \n*****\n";
00630 for(auto iter = processed_client_buf.begin(); iter != processed_client_buf.end(); ++iter)
00631 {
00632     log_processing << "PID " << iter->pid << " from socket " << iter->fd << " requested " <<
iter->operation << " " << iter->target << " advised " << iter->answer << endl;
00633 }
00634 log_processing << "*****\n end\n";
00635 #endif
00636 }
00637 else
00638     sleep(1);
00639 }
00640
00641 for(auto iter = processed_client_buf.begin(); iter != processed_client_buf.end(); )
00642 {
00643     iter->answer = "EXIT";
00644     // cerr <<
00645     secure_send(&*iter);
00646     close(iter->fd);
00647 }
00648 processed_client_buf.clear();
00649
00650 #ifdef DEBUG
00651 log_processing.seekp(0, std::ios_base::end);
00652 t = time(nullptr);
00653 tm = *localtime(&t);
00654 log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Time to exit\n";
00655 log_processing.close();
00656 #endif
00657
00658 pthread_exit(NULL);
00659 }

```

Here is the call graph for this function:



Here is the caller graph for this function:



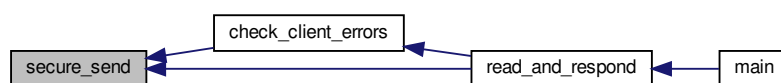
5.1.2.6 secure_send()

```
int secure_send (  
    client_buffer * client_buf )
```

Definition at line 193 of file [file_scheduler.cpp](#).

```
00194 {  
00195     for(size_t as = 0; as < client_buf->answer.length();)  
00196     {  
00197         auto sent = send(client_buf->fd, client_buf->answer.substr(as).c_str(), client_buf->  
00198         answer.substr(as).length(), 0);  
00199         if(sent < 0)  
00200         {  
00201             cerr << "Error on socket " << client_buf->fd << endl;  
00202             return -1;  
00203         }  
00204         as += sent;  
00205     }  
00206     return 0;  
00207 }
```

Here is the caller graph for this function:



5.1.2.7 signalHandler()

```
void signalHandler (  
    int signum )
```

Definition at line 117 of file [file_scheduler.cpp](#).

```
00118 {  
00119     cerr << "Interrupt signal (" << signum << ") received.\n";  
00120     cerr << "Programm has 30 seconds to finish or it will be forced to exit.\n";  
00121     exit_code = signum;  
00122     time_to_exit = true;  
00123     sleep(30);  
00124     cerr << "Looks like program does not respond. Killing.\n";  
00125     exit(signum);  
00126 }
```

Here is the caller graph for this function:



5.1.3 Variable Documentation

5.1.3.1 exit_code

```
int exit_code = 0
```

Definition at line 115 of file [file_scheduler.cpp](#).

5.1.3.2 lock

```
pthread_mutex_t lock
```

Definition at line 113 of file [file_scheduler.cpp](#).

5.1.3.3 time_to_exit

```
bool time_to_exit = false
```

Definition at line 114 of file [file_scheduler.cpp](#).

5.2 file_scheduler.cpp

```

00001 // This is a personal academic project. Dear PVS-Studio, please check it.
00002 // PVS-Studio Static Code Analyzer for C, C++ and C#: http://www.viva64.com
00003 /** @file file_scheduler.cpp */
00004 /** This server is used to synchronize pssc workers that run at different nodes.
00005 **
00006 ** Main problem is that each of them have to know whether to generate 'reuse' file
00007 ** read it or wait while it is being generated by other worker(node).
00008 **
00009 ** This server recieves data about requested file and respond with 'suggestion':
00010 ** READ, WRIT, WAIT.
00011 **
00012 ** In short. If file does not exist and is not being generated - generate it (WRIT).
00013 ** If file exists - read it (READ). Read operation does not change the data, so can be
00014 ** done in parallel, anyway file will be cached into RAM.
00015 ** If file is being generated - WAIT for a next READ message.
00016 **
00017 ** Communication is done by epoll. It is possible to add any reasonable number
00018 ** of threads if needed, but for the environment it was coded for -
00019 ** two threads is more than enough.
00020 ** First thread accepts connections, second communicates with clients.
00021 **
00022 **
00023 ** This server should be launched on one of the nodes. Other clients should
00024 ** know server's ip. Because of the asynchronous design, it produces
00025 ** very little overhead.
00026 ** When received SIGINT - all connection should be closed and program terminated.
00027 ** Check PYSSC git for a client version.
00028 ** Although this server was tested with many threads and for a long time,
00029 ** it may still have some error or space for improvement. I would be glad to hear
00030 ** any response.
00031 **
00032 ** Code sucessfully passedd PVS-Studio and Valgrind check.
00033 */
00034 //=====
00035 // Name      : file_scheduler.cpp
00036 // Author    : Ivan Syzonenko
00037 // Version   :
00038 // Copyright : MIT
00039 // Description : This server is used to synchronize pssc workers that ran at different nodes
00040 //=====
00041 #include <fcntl.h>
00042 #include <netinet/in.h>
00043 #include <pthread.h>
00044 #include <stddef.h>
00045 #include <string.h>
00046 #include <sys/epoll.h>
00047 #include <sys/socket.h>
00048 #include <unistd.h>
00049 #include <cerrno>
00050 #include <cstdio>
00051 #include <cstdlib>
00052 #include <iostream>
00053 #include <iterator>
00054 #include <string>
00055 #include <vector>
00056 #include <sstream>
00057 #include <queue>
00058 #include <deque>
00059 #include <fstream>
00060 #include <ctime>
00061 #include <iomanip>
00062 #include <arpa/inet.h>
00063 #include <csignal>
00064
00065 using std::string;
00066 using std::cerr;
00067 using std::cout;
00068 using std::cin;
00069 using std::endl;
00070 using std::vector;
00071 using std::queue;
00072 using std::deque;
00073 using std::put_time;
00074
00075 // enables log files
00076 // #define DEBUG
00077 // One server generates around 20-25 events.
00078 // For 4 nodes I expect 100 events for cluster
00079 #define MAXEVENTS 500
00080
00081 // just wrapper for better understanding.
00082 struct fd_struct
00083 {
00084     int fd; /// just wrapper for better understanding.

```



```

00085 };
00086
00087 struct thread_data
00088 {
00089     queue <fd_struct> file_descriptors;
00090 };
00091
00092 struct client_buffer
00093 {
00094     int pid;
00095     int fd;
00096     string operation;
00097     string target;
00098     string answer;
00099 };
00100
00101 struct read_add
00102 {
00103     int fd;
00104     string buf;
00105 };
00106
00107 size_t parse_buffer(string str, deque <client_buffer> *client_buf, int fd);
00108 int secure_send(client_buffer* client_buf);
00109 void check_client_errors(deque <client_buffer> *processed_client_buf, deque
    <client_buffer> *client_buf, const ssize_t fd);
00110 void *read_and_respond(void * threadarg);
00111 int accept_connections(uint16_t port, queue <fd_struct> *clients);
00112
00113 pthread_mutex_t lock;
00114 bool time_to_exit = false;
00115 int exit_code = 0;
00116
00117 void signalHandler( int signum )
00118 {
00119     cerr << "Interrupt signal ( " << signum << " ) received.\n";
00120     cerr << "Program has 30 seconds to finish or it will be forced to exit.\n";
00121     exit_code = signum;
00122     time_to_exit = true;
00123     sleep(30);
00124     cerr << "Looks like program does not respond. Killing.\n";
00125     exit(signum);
00126 }
00127
00128 /*!
00129 Parses input buffer and stores parsed messages in queue
00130 It may parse more than one message(stored in str) and if last message is incomplete - returns how many
    characters to save in external buffer for future processing.
00131 \param[in] str Input buffer with data recieved in socket fd.
00132 \param[in] client_buf Structure that keeps all parsed messages.
00133 \param[in] fd file descriptor associated with passed buffer data.
00134 \return how many characters to save in external buffer for future processing
00135 */
00136 size_t parse_buffer(string str, deque <client_buffer> *client_buf, int fd)
00137 {
00138     bool done = false;
00139     bool error = false;
00140     client_buffer temp;
00141     vector <string> info_block;
00142     string current_str;
00143     size_t found = 0;
00144     string token;
00145
00146     while(str.length() > 0 && !done && !error)
00147     {
00148         found = str.find_first_of("#");
00149         if(found == std::string::npos)
00150         {
00151             error = true;
00152             continue;
00153         }
00154
00155         unsigned fut_len = stoi(str.substr(0, found));
00156         if(str.length() < found + 1 + fut_len)
00157             break;
00158
00159         current_str = str.substr(found+1, fut_len);
00160         str = str.substr(found + 1 + fut_len);
00161
00162         std::istringstream iss(current_str);
00163         info_block.clear();
00164         while (getline(iss, token, '#'))
00165             info_block.push_back(token);
00166
00167         current_str.clear();
00168
00169         temp = {};

```

```

00170         if(info_block.size() > 0 )
00171             temp.pid = stoi(info_block[0]);
00172
00173         if(info_block.size() > 1 )
00174             temp.operation = info_block[1];
00175
00176         if(info_block.size() > 2 )
00177             temp.target = info_block[2];
00178
00179         temp.fd = fd;
00180
00181         if(temp.operation == "DONE")
00182             client_buf->push_front(temp);
00183         else
00184             client_buf->push_back(temp);
00185
00186         if(str.length() < 3)
00187             done = true;
00188     }
00189
00190     return str.length();
00191 }
00192
00193 int secure_send(client_buffer* client_buf)
00194 {
00195     for(size_t as = 0; as < client_buf->answer.length();)
00196     {
00197         auto sent = send(client_buf->fd, client_buf->answer.substr(as).c_str(), client_buf->
00198             answer.substr(as).length(), 0);
00199         if(sent < 0)
00200         {
00201             cerr << "Error on socket " << client_buf->fd << endl;
00202             return -1;
00203         }
00204         as += sent;
00205     }
00206     return 0;
00207 }
00208 void check_client_errors(deque <client_buffer> *processed_client_buf, deque
00209 <client_buffer> *client_buf, const ssize_t fd)
00210 {
00211     string error_target = "";
00212     for(auto iter = client_buf->begin(); iter != client_buf->end(); ++iter)
00213     {
00214         if(iter->fd == fd)
00215             return;
00216     }
00217     for(auto iter = processed_client_buf->begin(); iter != processed_client_buf->end(); ++iter)
00218     {
00219         if(iter->fd == fd )
00220         {
00221             error_target = iter->target;
00222             cerr << "Broken client removing: " << iter->fd << " " << iter->operation << " " << iter->target
00223 << endl;
00224             processed_client_buf->erase(iter);
00225             break;
00226         }
00227     }
00228     if(error_target.length() > 0)
00229     {
00230         for(auto iter = processed_client_buf->begin(); iter != processed_client_buf->end(); ++iter)
00231         {
00232             if(iter->target == error_target && iter->answer == "WAIT")
00233             {
00234                 iter->answer = "WRIT";
00235                 cerr << "PID " << iter->pid << " advised to WRIT";
00236                 if(secure_send(&*iter) != 0)
00237                     cerr << "ERROR in secure send";
00238                 break;
00239             }
00240         }
00241     }
00242 }
00243
00244 /*!
00245 Registers new sockets in epoll function(waits on data in async mode). Reads data from sockets in async
00246 mode. Calls parse function and makes decision according to the processed requests.
00247 \param[in] threadarg Structure that contains address of queue with file descriptors.
00248 */
00249 void *read_and_respond(void * threadarg)
00250 {
00251     auto my_data = (thread_data *) threadarg;
00252     queue <fd_struct> *fds = &my_data->file_descriptors;
00253     struct epoll_event event;

```

```

00253     struct epoll_event *events;
00254     auto efd = epoll_create1 (0);
00255     vector <int> fd_to_remove;
00256     // vector <size_t> local_fd;
00257     deque <client_buffer> client_buf;
00258     deque <client_buffer> processed_client_buf;
00259     vector <read_add> buff_add;
00260     read_add ra;
00261     events = (epoll_event*)calloc (MAXEVENTS, sizeof event);
00262     event.events = EPOLLIN | EPOLLET;
00263     size_t fd_event_counter = 0;
00264     int n;
00265     std::ofstream log_processing;
00266     log_processing.open("processing.log", std::ios::out | std::ios::app);
00267     #ifdef DEBUG
00268         auto t = time(nullptr);
00269         auto tm = *localtime(&t);
00270         log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Thread created\n";
00271     #endif
00272
00273     while(!time_to_exit)
00274     {
00275         #ifdef DEBUG
00276             if(fd_to_remove.size() > 0)
00277             {
00278                 t = time(nullptr);
00279                 tm = *localtime(&t);
00280                 log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Descriptors to clean: " <<
fd_to_remove.size() << "\n";
00281             }
00282             #endif
00283             for(unsigned j=0; j < fd_to_remove.size(); ++j)
00284             {
00285                 for(size_t k = 0; k < buff_add.size(); ++k)
00286                 {
00287                     if(buff_add[k].fd == fd_to_remove[j])
00288                     {
00289                         #ifdef DEBUG
00290                             t = time(nullptr);
00291                             tm = *localtime(&t);
00292                             log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Removing " <<
fd_to_remove[j] << endl;
00293                         #endif
00294                         buff_add.erase(buff_add.begin() + k);
00295                         break;
00296                     }
00297                 }
00298             }
00299             for(auto iter = processed_client_buf.begin(); iter != processed_client_buf.end(); )
00300             {
00301                 if(iter->fd == fd_to_remove[j])
00302                     iter = processed_client_buf.erase(iter);
00303                 else
00304                     iter++;
00305             }
00306         }
00307
00308         fd_event_counter -= fd_to_remove.size();
00309         fd_to_remove.clear();
00310
00311         #ifdef DEBUG
00312             if(!fds->empty())
00313             {
00314                 t = time(nullptr);
00315                 tm = *localtime(&t);
00316                 log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Processing new connections\n";
00317                 if(!fds->empty())
00318                     log_processing.seekp(-1, std::ios_base::cur);
00319             }
00320         #endif
00321
00322         while(!fds->empty())
00323         {
00324             pthread_mutex_lock(&lock);
00325             event.data.fd = fds->front().fd;
00326             fds->pop();
00327             pthread_mutex_unlock(&lock);
00328             ++fd_event_counter;
00329             #ifdef DEBUG
00330                 log_processing << ".";
00331             #endif
00332
00333             if( epoll_ctl(efd, EPOLL_CTL_ADD, event.data.fd, &event) == -1)
00334             {
00335                 perror ("epoll_ctl");
00336                 cerr << "EPOLL ERROR\n";
00337

```

```

00338         time_to_exit = true;
00339         #ifdef DEBUG
00340             log_processing.seekp(0, std::ios_base::end);
00341             t = time(nullptr);
00342             tm = *localtime(&t);
00343             log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "EPOLL ERROR\n";
00344             log_processing.close();
00345         #endif
00346         pthread_exit(NULL);
00347     }
00348 }
00349
00350 #ifdef DEBUG
00351     log_processing.seekp(0, std::ios_base::end);
00352 #endif
00353
00354 if(fd_event_counter > 0)
00355 {
00356     n = epoll_wait(efd, events, MAXEVENTS, 1000);
00357     for(int i = 0; i < n; ++i)
00358     {
00359         if ((events[i] != NULL) && ((events[i].events & EPOLLERR) || (events[i].events & EPOLLHUP
00360 ) || (!(events[i].events & EPOLLIN))))
00361         {
00362             cerr << "epoll error\n";
00363             close (events[i].data.fd);
00364             fd_to_remove.push_back(events[i].data.fd);
00365             continue;
00366         }
00367         else
00368         {
00369             int done = 0;
00370             ra.buf = "";
00371
00372             for(unsigned k = 0; k < buff_add.size(); ++k)
00373             {
00374                 if(buff_add[k].fd == events[i].data.fd)
00375                 {
00376                     ra.buf = buff_add[k].buf;
00377                     break;
00378                 }
00379             }
00380             #ifdef DEBUG
00381                 log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Reading from socket " <
00382 < events[i].data.fd << endl;
00383             #endif
00384             while (1)
00385             {
00386                 ssize_t count;
00387                 char buf[512];
00388                 memset(buf, 0, sizeof buf);
00389                 count = recv(events[i].data.fd, buf, sizeof buf, 0);
00390                 #ifdef DEBUG
00391                     t = time(nullptr);
00392                     tm = *localtime(&t);
00393                     log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Recieved: " <<
00394 string(buf) << endl;
00395                 #endif
00396                 if (count == -1)
00397                 { // If errno == EAGAIN, that means we have read all data. So go back to the main
00398 loop.
00399                     if (errno != EAGAIN)
00400                     {
00401                         perror ("read");
00402                         cerr << "Count error";
00403                         #ifdef DEBUG
00404                             t = time(nullptr);
00405                             tm = *localtime(&t);
00406                             log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "ERROR: Count
00407 error in epoll" << endl;
00408                         #endif
00409                         done = 1;
00410                     }
00411                     break;
00412                 }
00413                 else if (count == 0)
00414                 { // End of file. The remote has closed the connection.
00415                     done = 1;
00416                     break;
00417                 }
00418                 ra.buf += string(buf);
00419                 #ifdef DEBUG
00420                     t = time(nullptr);
00421                     tm = *localtime(&t);
00422                     log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Parsing message of
00423 length " << ra.buf.length() << endl;
00424                     log_processing << "Message: " << string(ra.buf) << endl;

```

```

00419                                     #endif
00420                                     size_t char_left = parse_buffer(ra.buf, &client_buf, events[i].data.
fd);
00421                                     #ifdef DEBUG
00422                                     t = time(nullptr);
00423                                     tm = *localtime(&t);
00424                                     if(client_buf.size() > 0)
00425                                     {
00426                                         for(auto iter = client_buf.begin(); iter != client_buf.end(); ++iter)
00427                                         {
00428                                             log_processing << "PID " << iter->pid << " on " << iter->fd << " requested
" << iter->operation << " " << iter->target << endl;
00429                                         }
00430                                     }
00431                                     else
00432                                     {
00433                                         log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "WARNING: Client
buffer is empty\n";
00434                                     }
00435                                     #endif
00436                                     if(char_left > 0)
00437                                     {
00438                                         #ifdef DEBUG
00439                                         t = time(nullptr);
00440                                         tm = *localtime(&t);
00441                                         log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Not all
information was received. Missing " << char_left << " chars\n";
00442                                         #endif
00443                                         ra.fd = events[i].data.fd;
00444                                         unsigned k = 0;
00445                                         for(; k < buff_add.size(); ++k)
00446                                         {
00447                                             if(buff_add[k].fd == events[i].data.fd)
00448                                             {
00449                                                 buff_add[k].buf += ra.buf;
00450                                                 break;
00451                                             }
00452                                         }
00453                                         if(k == buff_add.size())
00454                                             buff_add.push_back(ra);
00455                                     }
00456                                     else
00457                                     {
00458                                         #ifdef DEBUG
00459                                         t = time(nullptr);
00460                                         tm = *localtime(&t);
00461                                         log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Success in
reading full message\n";
00462                                         #endif
00463                                         for(unsigned k = 0; k < buff_add.size(); ++k)
00464                                         {
00465                                             if(buff_add[k].fd == events[i].data.fd)
00466                                             {
00467                                                 buff_add.erase(buff_add.begin() + k);
00468                                                 break;
00469                                             }
00470                                         }
00471                                     }
00472                                     }
00473                                     if (done)
00474                                     {
00475                                         #ifdef DEBUG
00476                                         t = time(nullptr);
00477                                         tm = *localtime(&t);
00478                                         log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Closing connection on
descriptor " << events[i].data.fd << endl;
00480                                         #endif
00481                                         check_client_errors(&processed_client_buf, &client_buf, events[i
].data.fd);
00482                                     }
00483                                     close (events[i].data.fd); // Closing the descriptor will make epoll remove it from
the set of descriptors which are monitored.
00484                                     fd_to_remove.push_back(events[i].data.fd);
00485                                     }
00486                                     }
00487                                     }
00488                                     }
00489                                     #ifdef DEBUG
00490                                     if(!client_buf.empty())
00491                                     {
00492                                         t = time(nullptr);
00493                                         tm = *localtime(&t);
00494                                         log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Begin processing requests.\n";
00495                                     }
00496                                     #endif
00497

```

```

00498
00499     #ifdef DEBUG
00500     log_processing << "processed_client_buf before processing new requests \n*****\n";
00501     for(auto iter = processed_client_buf.begin(); iter != processed_client_buf.end(); ++iter)
00502     {
00503         log_processing << "PID " << iter->pid << " from socket " << iter->fd << " requested " <<
iter->operation << " " << iter->target << " advised " << iter->answer << endl;
00504     }
00505     log_processing << "*****\n end\n";
00506     #endif
00507
00508     while(!client_buf.empty())
00509     {
00510         if(client_buf.front().operation == "READ")
00511         {
00512             #ifdef DEBUG
00513             t = time(nullptr);
00514             tm = *localtime(&t);
00515             log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "PID: " << client_buf.
front().pid << " from socket " << client_buf.front().fd << " wants to read " << client_buf.front().target <<
endl;
00516             #endif
00517             client_buf.front().answer = "READ";
00518
00519             for(auto iter = processed_client_buf.begin(); iter != processed_client_buf.end(); ++
iter)
00520             {
00521                 if(iter->target == client_buf.front().target )
00522                 {
00523                     if(iter->answer == "WRIT" || iter->answer == "WAIT")
00524                     {
00525                         client_buf.front().answer = "WAIT";
00526                         break;
00527                     }
00528                     else if(iter->answer == "READ")
00529                     {
00530                         client_buf.front().answer = "READ";
00531                         break;
00532                     }
00533                 }
00534             }
00535
00536             if(secure_send(&client_buf.front()) == 0)
00537                 processed_client_buf.push_back(client_buf.front());
00538             else
00539                 cerr << "ERROR in secure send";
00540
00541             #ifdef DEBUG
00542             t = time(nullptr);
00543             tm = *localtime(&t);
00544             log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "RESPONSE to PID: " <<
client_buf.front().pid << " from socket " << client_buf.front().fd << ": " << client_buf.front().answer <<
endl;
00545             #endif
00546         }
00547         else if(client_buf.front().operation == "WRIT")
00548         {
00549             #ifdef DEBUG
00550             t = time(nullptr);
00551             tm = *localtime(&t);
00552             log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "PID: " << client_buf.
front().pid << " from socket " << client_buf.front().fd << " wants to write " << client_buf.front().target <<
endl;
00553             #endif
00554             client_buf.front().answer = "WRIT";
00555             for(auto iter = processed_client_buf.begin(); iter != processed_client_buf.end(); ++
iter)
00556             {
00557                 if(iter->target == client_buf.front().target )
00558                 {
00559                     if(iter->answer == "WRIT" || iter->answer == "WAIT")
00560                     {
00561                         client_buf.front().answer = "WAIT";
00562                         break;
00563                     }
00564                     else if(iter->answer == "READ")
00565                     {
00566                         client_buf.front().answer = "READ";
00567                         break;
00568                     }
00569                 }
00570             }
00571
00572             if(secure_send(&client_buf.front()) == 0)
00573                 processed_client_buf.push_back(client_buf.front());
00574             else
00575                 cerr << "ERROR in secure send";

```

```

00576
00577         #ifdef DEBUG
00578             t = time(nullptr);
00579             tm = *localtime(&t);
00580             log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "RESPONSE to PID: " <<
client_buf.front().pid << " from socket " << client_buf.front().fd << ": " << client_buf.front().answer <<
endl;
00581         #endif
00582     }
00583     else if(client_buf.front().operation == "DONE")
00584     {
00585         #ifdef DEBUG
00586             t = time(nullptr);
00587             tm = *localtime(&t);
00588             log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "PID: " << client_buf.
front().pid << " from socket " << client_buf.front().fd << " finished working with " << client_buf.front().
target << endl;
00589         #endif
00590
00591         for(auto iter = processed_client_buf.begin(); iter != processed_client_buf.end(); )
00592         {
00593             if(iter->target == client_buf.front().target )
00594             {
00595                 if(iter->pid == client_buf.front().pid)
00596                 {
00597                     #ifdef DEBUG
00598                         t = time(nullptr);
00599                         tm = *localtime(&t);
00600                         log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "PID: " <<
iter->pid << " from socket " << iter->fd << ": is DONE - SELF-DESTRUCTION" << endl;
00601                     #endif
00602                     iter = processed_client_buf.erase(iter);
00603                     continue;
00604                 }
00605                 else if(iter->answer == "WAIT")
00606                 {
00607                     iter->answer = "READ";
00608                     if(secure_send(&*iter) != 0)
00609                         cerr << "ERROR in secure send";
00610
00611                     #ifdef DEBUG
00612                         t = time(nullptr);
00613                         tm = *localtime(&t);
00614                         log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "RESPONSE to
PID: " << iter->pid << " from socket " << iter->fd << ": " << iter->answer << endl;
00615                     #endif
00616                 }
00617             }
00618
00619             ++iter;
00620         }
00621     }
00622     else
00623         cerr << client_buf.front().operation << endl;
00624
00625     client_buf.pop_front();
00626 }
00627
00628 #ifdef DEBUG
00629 log_processing << "processed_client_buf after processing new requests \n*****\n";
00630 for(auto iter = processed_client_buf.begin(); iter != processed_client_buf.end(); ++iter)
00631 {
00632     log_processing << "PID " << iter->pid << " from socket " << iter->fd << " requested " <<
iter->operation << " " << iter->target << " advised " << iter->answer << endl;
00633 }
00634 log_processing << "*****\n end\n";
00635 #endif
00636 }
00637 else
00638     sleep(1);
00639 }
00640
00641 for(auto iter = processed_client_buf.begin(); iter != processed_client_buf.end(); )
00642 {
00643     iter->answer = "EXIT";
00644     // cerr <<
00645     secure_send(&*iter);
00646     close(iter->fd);
00647 }
00648 processed_client_buf.clear();
00649
00650 #ifdef DEBUG
00651 log_processing.seekp(0, std::ios_base::end);
00652 t = time(nullptr);
00653 tm = *localtime(&t);
00654 log_processing << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Time to exit\n";
00655 log_processing.close();

```

```

00656     #endif
00657
00658     pthread_exit(NULL);
00659 }
00660
00661 /*!
00662 Initializes socket on port 'port' and waits for connections. On incoming connection sets associated socket
    to async mode
00663 and stores in fd_struct structure which is shared with processing thread.
00664 \param[in] port Port used to create listening socket.
00665 \param[in] clients queue that stores file descriptors(sockets) accepted on port 'port'.
00666 \returns status code
00667 */
00668 int accept_connections(uint16_t port, queue <fd_struct> *clients)
00669 {
00670     std::string rcv;
00671     int listen_fd, comm_fd;
00672     struct sockaddr_in servaddr;
00673     std::ofstream log_main;
00674     log_main.open("incoming.log", std::ios::out | std::ios::app);
00675     #ifdef DEBUG
00676         auto t = time(nullptr);
00677         auto tm = *localtime(&t);
00678         log_main << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Thread created\n";
00679     #endif
00680
00681     listen_fd = socket(AF_INET, SOCK_STREAM, 0);
00682
00683     if (listen_fd == -1)
00684     {
00685         cout << "Can't create file descriptor." << endl;
00686         exit_code = 1;
00687         time_to_exit = true;
00688         sleep(10);
00689         exit(1);
00690     }
00691
00692     memset(&servaddr, 0, sizeof(servaddr));
00693     servaddr.sin_family = AF_INET;
00694     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
00695     servaddr.sin_port = htons(port);
00696     #ifdef DEBUG
00697         t = time(nullptr);
00698         tm = *localtime(&t);
00699         log_main << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Attempting to listen on " << port << " port\n";
00700     #endif
00701     int my_timer = 20;
00702
00703     while(my_timer > 0)
00704     {
00705         if(bind(listen_fd, (struct sockaddr *)&servaddr, sizeof(servaddr)) < 0)
00706             sleep(10);
00707         else
00708             break;
00709         my_timer--;
00710     }
00711
00712     if(my_timer == 0)
00713     {
00714         cout << "Binding to socket error." << endl;
00715         exit_code = 1;
00716         time_to_exit = true;
00717         sleep(10);
00718         exit(2);
00719     }
00720     #ifdef DEBUG
00721         t = time(nullptr);
00722         tm = *localtime(&t);
00723         log_main << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Successful bind to the socket\n";
00724     #endif
00725
00726     fd_struct temp;
00727
00728     while(!time_to_exit)
00729     {
00730         listen(listen_fd, 60);
00731         #ifdef DEBUG
00732             t = time(nullptr);
00733             tm = *localtime(&t);
00734             log_main << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Waiting for incoming connections.\n";
00735             log_main.flush();
00736         #endif
00737         sockaddr_in clientAddr;
00738         socklen_t sin_size=sizeof(struct sockaddr_in);
00739         comm_fd = accept(listen_fd, (struct sockaddr*)&clientAddr, &sin_size);
00740

```



```

00741         if(comm_fd == -1)
00742         {
00743             cout << "Connection acceptance error." << endl;
00744             //      exit(3);
00745         }
00746
00747         #ifdef DEBUG
00748             t = time(nullptr);
00749             tm = *localtime(&t);
00750             char loc_addr[INET_ADDRSTRLEN+1];
00751             inet_ntop(AF_INET, &(clientAddr.sin_addr), loc_addr, INET_ADDRSTRLEN);
00752             log_main << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Incoming connenction on descriptor " <<
comm_fd << " from " << loc_addr << ":" << clientAddr.sin_port << "\n";
00753         #endif
00754         //      make nonblocking
00755         auto flags = fcntl (comm_fd, F_GETFL, 0);
00756         if (flags < 0)
00757         {
00758             perror ("fcntl");
00759             return -1;
00760         }
00761
00762         flags |= O_NONBLOCK;
00763         auto s = fcntl (comm_fd, F_SETFL, flags);
00764
00765         if(s < 0)
00766         {
00767             perror ("fcntl");
00768             time_to_exit = true;
00769             log_main.close();
00770             return -1;
00771         }
00772         #ifdef DEBUG
00773             t = time(nullptr);
00774             tm = *localtime(&t);
00775             log_main << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Switched " << comm_fd << " to nonblocking
mode.\n";
00776         #endif
00777
00778         temp.fd = comm_fd;
00779
00780         pthread_mutex_lock(&lock);
00781         clients->push(temp);
00782         pthread_mutex_unlock(&lock);
00783         #ifdef DEBUG
00784             t = time(nullptr);
00785             tm = *localtime(&t);
00786             log_main << put_time(&tm, "[%H:%M:%S %d-%m-%Y] ") << "Successfully pushed into queue for
further processing.\n";
00787         #endif
00788     }
00789
00790     #ifdef DEBUG
00791         log_main.close();
00792     #endif
00793
00794     return 0;
00795 }
00796
00797 /*!
00798 Nothing fancy. Creates a thread and launches connection accepting function
00799 \returns status code to OS
00800 \param clients data structure to store fd
00801 */
00802 int main()
00803 {
00804     queue <fd_struct> clients;
00805     pthread_t threads[1];
00806
00807     signal(SIGINT, signalHandler);
00808
00809     if (pthread_mutex_init(&lock, NULL) != 0)
00810     {
00811         printf("\n mutex init failed\n");
00812         return 1;
00813     }
00814
00815     int rc = pthread_create(&threads[0], NULL, read_and_respond, (void *)&clients);
00816     if (rc)
00817     {
00818         cout << "Error:unable to create thread," << rc << endl;
00819         exit_code = -1;
00820         time_to_exit = true;
00821         sleep(10);
00822         return exit_code;
00823     }
00824 }

```

```
00825
00826     accept_connections(1987, &clients);
00827
00828     pthread_join(rc, NULL);
00829     pthread_mutex_destroy(&lock);
00830
00831     return exit_code;
00832 }
```

5.3 README.md File Reference

5.4 README.md

```
00001 # PYSSC scheduler.
00002
00003 This server is used to synchronize pssc workers that run at different nodes.
00004
00005 Main problem is that each of them have to know whether to generate 'reuse' file
00006 read it or wait while it is being generated by other worker(node).
00007
00008 This server recieves data about requested file and respond with 'suggestion':
00009 READ, WRIT, WAIT.
00010
00011 In short. If file does not exist and is not being generated - generate it (WRIT).
00012 If file exists - read it (READ). Read operation does not change the data, so can be
00013 done in parallel, anyway file will be cached into RAM.
00014 If file is being generated - WAIT for a next READ message.
00015
00016 Communication is done by epoll. It is possible to add any reasonable number
00017 of threads if needed, but for the environment it was codded for -
00018 two threads is more than enough.
00019 First thread accepts connections, second communicates with clients.
00020
00021
00022 This server should be launched on one of the nodes. Other clients should
00023 know server's ip. Because of the asynchronous design, it produces
00024 very little overhead.
00025 When received SIGINT - all connection should be closed and program terminated.
00026 Check PYSSC git for a client version.
00027 Although this server was tested with many threads and for a long time,
00028 it may still have some error or space for improvement. I would be glad to hear
00029 any response.
00030
00031 Code sucessfully passeddd PVS-Studio and Valgrind check.
```


Index

- accept_connections
 - file_scheduler.cpp, [9](#)
- answer
 - client_buffer, [4](#)
- buf
 - read_add, [6](#)
- check_client_errors
 - file_scheduler.cpp, [11](#)
- client_buffer, [3](#)
 - answer, [4](#)
 - fd, [4](#)
 - operation, [4](#)
 - pid, [4](#)
 - target, [4](#)
- exit_code
 - file_scheduler.cpp, [22](#)
- fd
 - client_buffer, [4](#)
 - fd_struct, [5](#)
 - read_add, [6](#)
- fd_struct, [5](#)
 - fd, [5](#)
- file_descriptors
 - thread_data, [8](#)
- file_scheduler.cpp, [8](#)
 - accept_connections, [9](#)
 - check_client_errors, [11](#)
 - exit_code, [22](#)
 - lock, [22](#)
 - MAXEVENTS, [9](#)
 - main, [12](#)
 - parse_buffer, [13](#)
 - read_and_respond, [15](#)
 - secure_send, [21](#)
 - signalHandler, [21](#)
 - time_to_exit, [22](#)
- lock
 - file_scheduler.cpp, [22](#)
- MAXEVENTS
 - file_scheduler.cpp, [9](#)
- main
 - file_scheduler.cpp, [12](#)
- operation
 - client_buffer, [4](#)
- parse_buffer
 - file_scheduler.cpp, [13](#)
- pid
 - client_buffer, [4](#)
- README.md, [33](#)
- read_add, [6](#)
 - buf, [6](#)
 - fd, [6](#)
- read_and_respond
 - file_scheduler.cpp, [15](#)
- secure_send
 - file_scheduler.cpp, [21](#)
- signalHandler
 - file_scheduler.cpp, [21](#)
- target
 - client_buffer, [4](#)
- thread_data, [7](#)
 - file_descriptors, [8](#)
- time_to_exit
 - file_scheduler.cpp, [22](#)