Project 2

This project's main goal was to create a server and client parts that communicate with each other through shared memory. The server compresses files using a third-party program snappy. The client, in turn, wants to get a compressed version of its files. Let's take a closer look at the design.

Queue.

In this implementation a program manages one queue for accepting and sending messages. The queue is created when tinyfiledeamon starts using the startup() function.

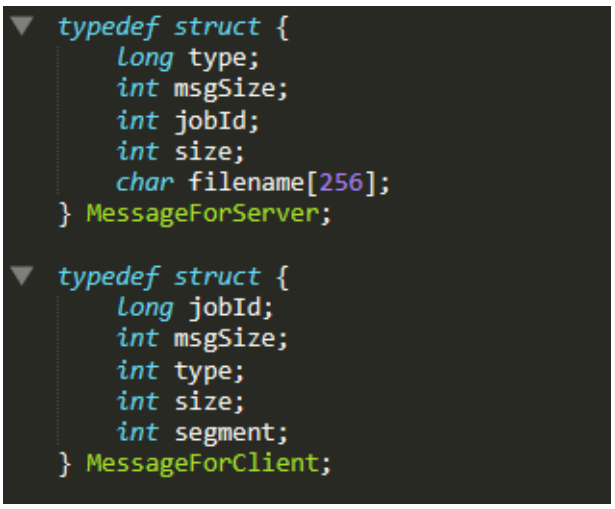
To create a queue, the functions ftok (to create a unique key) and msgget (to create a queue identifier) were used. Since all clients and the server must write to the queue and read from it, a public file "/usr/bin" and a randomly chosen proj_id = 42 were used. The msgctl function was used to delete the queue (Figure 1).

```
int open_msg_queue()
{
    key_t key = ftok(QUEUE_FILENAME, PROJECT_ID);
    if (key < 0) report_and_exit("couldn't get key...");

    int qid = msgget(key, 0666 | IPC_CREAT);
    if (qid < 0) report_and_exit("couldn't get queue id...");

    return qid;
}

void delete_msg_queue(int qid)
{
    if (msgctl(qid, IPC_RMID, NULL) < 0)  /* NULL = 'no flags' */
      report_and_exit("trouble removing queue...");
}
```

Figure 1. Queue operations

At the beginning of the startup() function, the queue is opened, deleted, and reopened. This ensures that the queue is empty before starting work and does not contain old information.

Since only one queue is used, we need to make sure that the server sees messages from all clients, and clients only see messages addressed to them. Because of that, we use two different structures MessageForServer and MessageForClient (Figure 2), and filter by first 'long' in the message. Communication between server and client takes place through the msgrcv and msgsnd functions.

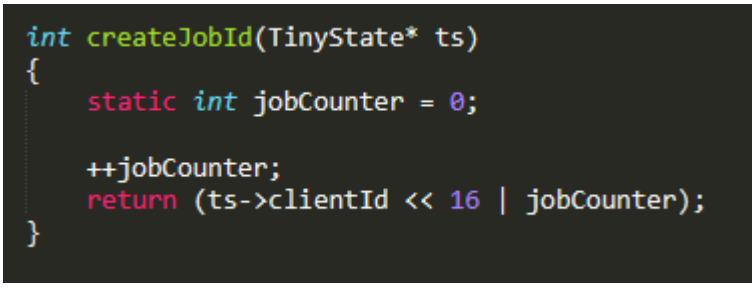


```
typedef struct {
    long type;
    int msgSize;
    int jobId;
    int size;
    char filename[256];
} MessageForServer;

typedef struct {
    long jobId;
    int msgSize;
    int type;
    int size;
    int segment;
} MessageForClient;
```

Figure 2. Message structures for client and server

In this case, the server will receive all types of messages (a total of 4 types for this project (figure 9)) from all clients since the structure's *type* variable has the long type. The server is waiting for messages: msgrcv(qid, msg, sizeof (MessageForServer) - sizeof (long), -CLIENT_ID_BASE, MSG_NOERROR), which means the server receives all messages with type less than CLIENT_ID_BASE (by default it is 1000 because we can have a lot of the types of messages).

In turn, the client will receive messages explicitly addressed to him because the filtering will go on the jobId variable. The client is waiting for messages: msgrcv (qid, msg, sizeof (MessageForClient) - sizeof (long), jobId, MSG_NOERROR), which means that the client only receives messages with its jobId. So, jobIds have to be unique between clients, so each client will use clientId (should be unique between all active clients) as a base to generate them (Figure 3).



```
int createJobId(TinyState* ts)
{
    static int jobCounter = 0;

    ++jobCounter;
    return (ts->clientId << 16 | jobCounter);
}
```

Figure 3. Creating jobId

Using this technique, we can use only one queue for receiving and transmitting messages, which is more convenient, faster, and easier to implement since the client and server do not have to switch between queues.

Segment.

To work with segments, we introduce two new structures, TinyState and TinyJob (Figure 4). TinyState stores main objects (queue, shared memory segments) and is used by clients and server. TinyJob stores information about one file. In this project, we consider that each file transferred uses one segment(job).

```c
typedef struct
{
    int segmentCount;
    int segmentSize;
    int clientId;

    int fd;
    int qid;
    char* memptr;
} TinyState;
```

```c
typedef struct
{
    char filename[512];

    char* incomingBytes;
    int incomingSize;
    char* outgoingBytes;
    int outgoingSize;

    // Depending on mode: offset to next byte to send/receive.
    // Or number of received or sent bytes, as those are the same.
    int offset;
    int id;

    // Index of the segment used by the job
    int segment;
} TinyJob;
```

Figure 4. New structures

A shared memory object is created using the shm_open function and map shared memory to process address space (Figure 5). For a shared memory object ftruncate() sets the object's size to size (segmentCount * segmentSize).

```c
void create_segment(char* name, int size, int* out_fd, char** out_memptr)
{
    int fd = shm_open(name, O_RDWR | O_CREAT, SEGMENT_ACCESS_PERM);  /* empty to begin */
    if (fd < 0) report_and_exit("Can't get file descriptor...");

    ftruncate(fd, size); /* get the bytes */

    /* get a pointer to memory */
    void* memptr = mmap(
        NULL,       /* let system pick where to put segment */
        size    ,   /* how many bytes */
        PROT_READ | PROT_WRITE, /* access protections */
        MAP_SHARED, /* mapping visible to other processes */
        fd,         /* file descriptor */
        0);         /* offset: start at 1st byte */
    if ((void*) -1 == memptr)
    {
        close(fd);
        report_and_exit("Can't access segment...");
    }

    memset(memptr, 0, size);

    *out_fd = fd;
    *out_memptr = memptr;
}
```

Figure 5. Segment creation

The client opens this particular shared memory object using the open_segment function (open a segment).

```
void open_segment(char* name, int size, int* out_fd, char** out_memptr)
{
    int fd = shm_open(name, O_RDWR, SEGMENT_ACCESS_PERM);  /* empty to begin */
    if (fd < 0) report_and_exit("Can't get file descriptor...");

    /* get a pointer to memory */
    void* memptr = mmap(
        NULL,       /* let system pick where to put segment */
        size    ,   /* how many bytes */
        PROT_READ | PROT_WRITE, /* access protections */
        MAP_SHARED, /* mapping visible to other processes */
        fd,         /* file descriptor */
        0);         /* offset: start at 1st byte */
    if ((void*) -1 == memptr)
    {
        close(fd);
        report_and_exit("Can't access segment...");
    }

    *out_fd = fd;
    *out_memptr = memptr;
}
```

Figure 6. Segment opening

Now let's figure out how the file data is transferred between the client and the server. Each file can only use one segment. It will transfer its contents in chunks equal to the segment's size until it has moved the entire file. This is implemented through the fillBuffer function (Figure 7).

First, we compare the file size and the segment size. If the file size is larger than the segment size, then the first part of the file is transferred, and the offset value is changed to this value. If the file size (or the remainder of the file) is less than the segment size, then the file or its rest is transferred.

```
    // Copy first part of the outgoing file into shared segment
    int bytesSent = fillBuffer(segPtr(ts, job), ts->segmentSize, job->outgoingBytes, job->offset, job->outgoingSize);
    job->offset += bytesSent;
```

```
int fillBuffer(char* dest, int destSize, const char* src, int srcOffset, int srcSize)
{
    int bytesToCopy = MIN(srcSize - srcOffset, destSize);
    memcpy(dest, src + srcOffset, bytesToCopy);
    return bytesToCopy;
}
```

Figure 7. Sending a file piece by piece

To adjust the number of our segments, we use the tryToStartNewJob() function (Figure 8). If the number of simultaneously sent files is greater than the number of free segments, then the client will receive the TINYMSG_SERVER_IS_FULL message and will no longer process this request.

Thus, we have saved the project's logic using only a certain number of segments with a given size. Of course, if we have only one file sent, only one segment will be used, which is not very productive. But there will be no situation when one client came with a large file and occupied all the segments, leaving no room for other clients.

```
TinyJob* tryToStartNewJob(TinyState* ts, int jobId, int size, const char* filename)
{

    for (int i = 0; i < ts->segmentCount; ++i)
    {
        if (isFree(&jobs[i]))
        {
            // Start of a new job
            strcpy(jobs[i].filename, filename);
            jobs[i].id = jobId;
            jobs[i].segment = i;
            jobs[i].incomingSize = size;
            jobs[i].incomingBytes = malloc(size);

            return &jobs[i];
        }
    }

    return NULL;
}
```

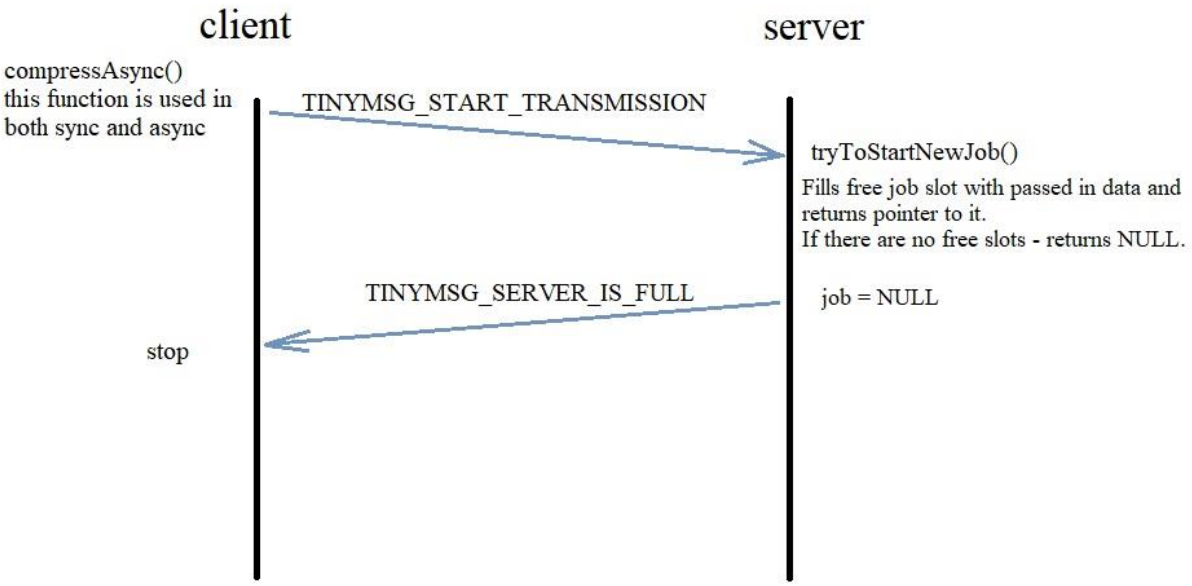
Figure 8. tryToStartNewJob() function

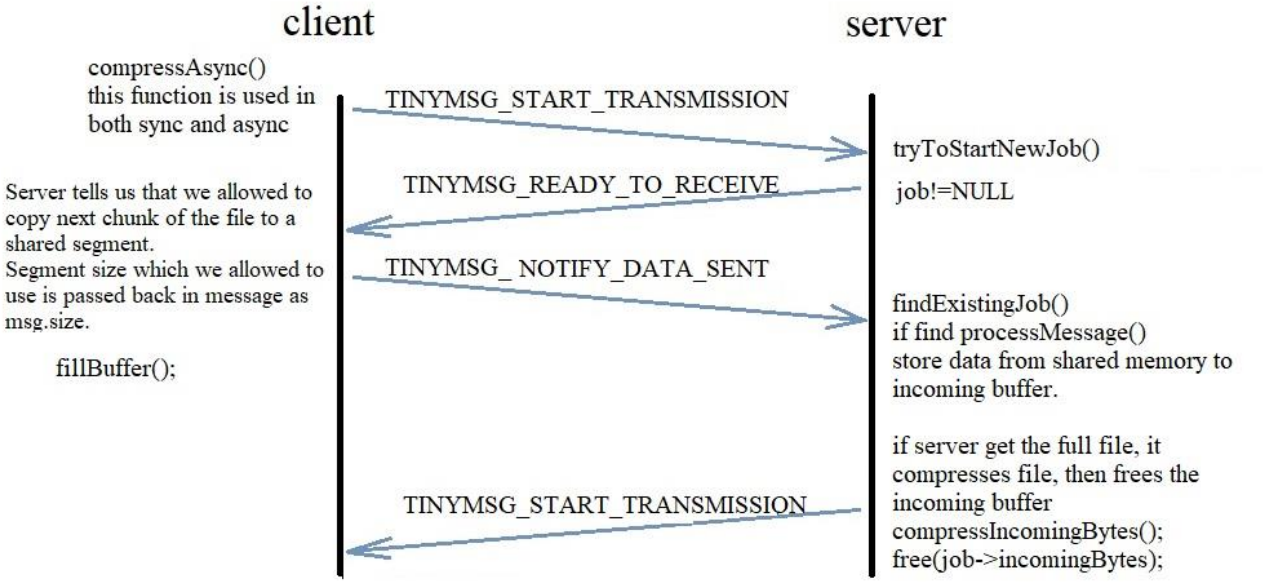
Exchange of messages between client and server.

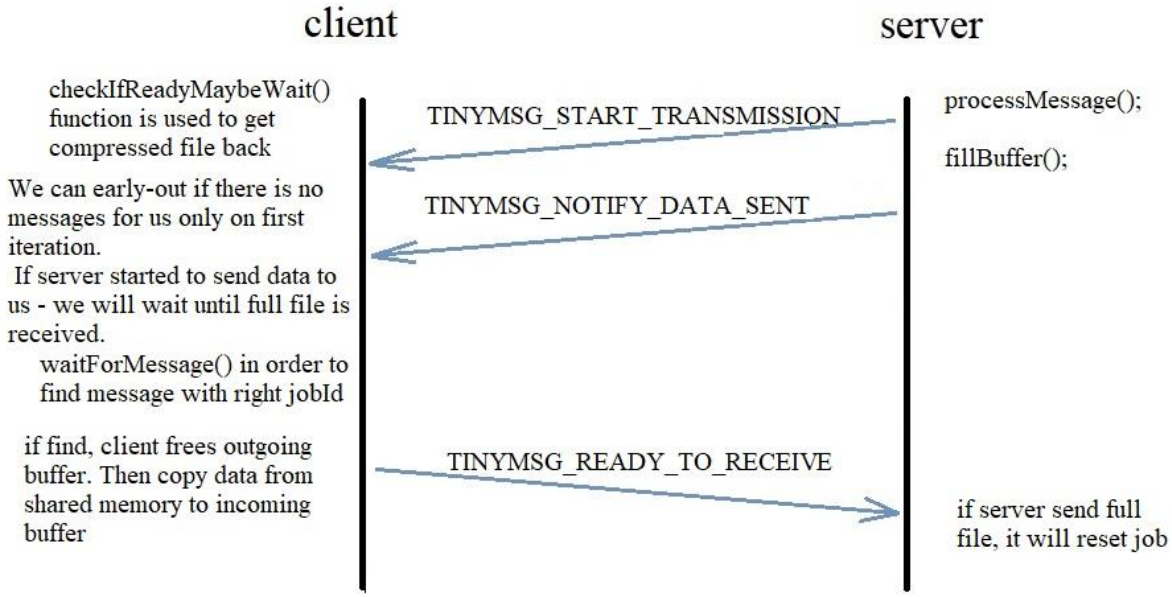
There are four types of messages in total (Figure 9). Before starting to send the file, the client sends a TINYMSG_START_TRANSMISSION message. If it receives a TINYMSG_READY_TO_RECEIVE response, the client begins transferring the file in pieces equal to the segment size, writing it to the shared memory object from an outgoing buffer. After that, the client sends a TINYMSG_NOTIFY_DATA_SENT message, and the server starts copying data from memory to the incoming buffer. Once the server has received the entire file, it will begin to compress it using snappy. Then it will send the file in the same way. Examples of communications are shown below (Figures 10-12).

```
// Sender wants to start transmission of a file of a specified size.
#define TINYMSG_START_TRANSMISSION 1
// Sent by server when it has free segment and ready to receive data.
// Or by client when he read chunk of data from a segment and ready for the next.
#define TINYMSG_READY_TO_RECEIVE 2
// Sender filled a segment with data. Size field tells how much bytes are there.
#define TINYMSG_NOTIFY_DATA_SENT 3
// Sent by server when there are no free slots for new jobs.
#define TINYMSG_SERVER_IS_FULL 4
```

Figure 9. Message types


client
server
compressAsync()
this function is used in
both sync and async
TINYMSG_START_TRANSMISSION
tryToStartNewJob()
Fills free job slot with passed in data and
returns pointer to it.
If there are no free slots - returns NULL.
TINYMSG_SERVER_IS_FULL
job = NULL
stop


client
server
compressAsync()
this function is used in
both sync and async
TINYMSG_START_TRANSMISSION
tryToStartNewJob()
Server tells us that we allowed to
copy next chunk of the file to a
shared segment.
Segment size which we allowed to
use is passed back in message as
msg.size.
TINYMSG_READY_TO_RECEIVE
job!=NULL
TINYMSG_ NOTIFY_DATA_SENT
fillBuffer();
findExistingJob()
if find processMessage()
store data from shared memory to
incoming buffer.
if server get the full file, it
compresses file, then frees the
incoming buffer
compressIncomingBytes();
free(job->incomingBytes);
TINYMSG_START_TRANSMISSION



Figures 10-12. Server and client communication

Synchronous and Asynchronous modes.

The synchronous mode will work exactly as shown above. Until the client has fully received its compressed file, it will wait for messages from the server (Figure 13).

```
void waitUntilReady(TinyState* ts, TinyJob* job)
{
    checkIfReadyMaybeWait(ts, job, true);
}

// Sync API just utilizes blocking method which will wait for results.
bool compressSync(TinyState* ts, TinyJob* job)
{
    if (compressAsync(ts, job))
    {
        waitUntilReady(ts, job);
        return true;
    }

    return false;
}

// To make things easier collect all files we need to send into single array
```

Рисунок13. Работа синхронного типа

In asynchronous mode we trying to show potential async nature. But because we are running on a single thread, we can't really get a lot of benefits. If we pretent, that sending and receiving is fast, but compression on server is slow (which is not true) - then we might want to send few files, do something usefull, then check results. This code will try to emulate this. So, we use the

compressAsync() function to send the file to the server, then emulate the client's work (in our case, we display the message), and then check if our compressed file has returned using the waitUntilRead() function (Figure 14). And we output the result in reverse order. jobCount is the number of files sent in parallel.

```
void sendFilesAsync(TinyState* ts, char** allFileNames, int fileCount)
{

    #define jobCount 2
    TinyJob parallelJobs[jobCount] = {0};
    double durations[jobCount] = {0};

    int fileIndex = 0;
    while (fileIndex < fileCount)
    {
        // Send out up to jobCount requests to server
        int i = 0;
        while (i < jobCount && fileIndex < fileCount)
        {
            if (createJob(ts, &parallelJobs[i], allFileNames[fileIndex]))
            {
                clock_t start = clock();
                if (!compressAsync(ts, &parallelJobs[i]))
                {
                    fatalError("We don't handle overloaded server scenario here.\n");
                }
                durations[i] = getDuration(&start);

                printf("Sent '%s' for processing\n", allFileNames[fileIndex]);

                ++i;
            }

            ++fileIndex;
        }

        // Here we pretend to do some useful work while server processed our data.
        printf("Working really hard...\n");

        // And then retrieve results. For fun let's do it in reverse order.
        for (int i = jobCount - 1; i >= 0; --i)
        {
            if (!isFree(&parallelJobs[i]))
            {
                clock_t start = clock();
                waitUntilReady(ts, &parallelJobs[i]);
                durations[i] += getDuration(&start);

                printf("CST for %s was %.3f ms\n", parallelJobs[i].filename, durations[i]);
            }
        }
    }

    #undef jobCount
}
```

Figure 14. Asynchronous mode

Examples and calculations.

```
vmakhambayeva3@advos-05:~/lab/tinyfiledaemon$ bin/release/tinyfiledaemon --n_sms 3 --sms_size 32
Current folder: bin/release/tinyfiledaemon
Parameters:
  - # of segments: 3
  - size of a segment: 32

TinyFile server starting...
  - msg queue opened...
  - shared memory segments allocated...
  - snappy initialized...
  - job slots allocated...
Done.


Received full file (33920 bytes).
Snappy compression: 33920 --> 3057 bytes.
Starting to send processed file (3057 bytes).
Received full file (33920 bytes).
Snappy compression: 33920 --> 3057 bytes.
Starting to send processed file (3057 bytes).
Received full file (233024 bytes).
Snappy compression: 233024 --> 232957 bytes.
Starting to send processed file (232957 bytes).
Received full file (233024 bytes).
Snappy compression: 233024 --> 232957 bytes.
Starting to send processed file (232957 bytes).
Received full file (532542 bytes).
Snappy compression: 532542 --> 479229 bytes.
Starting to send processed file (479229 bytes).
Received full file (532542 bytes).
Snappy compression: 532542 --> 479229 bytes.
Starting to send processed file (479229 bytes).
Received full file (1048575 bytes).
Snappy compression: 1048575 --> 877796 bytes.
Starting to send processed file (877796 bytes).
Received full file (1936994 bytes).
Snappy compression: 1936994 --> 1924917 bytes.
Starting to send processed file (1924917 bytes).
Received full file (33920 bytes).
Snappy compression: 33920 --> 3057 bytes.
Starting to send processed file (3057 bytes).
Received full file (33920 bytes).
Snappy compression: 33920 --> 3057 bytes.
Starting to send processed file (3057 bytes).
Received full file (233024 bytes).
Snappy compression: 233024 --> 232957 bytes.
Starting to send processed file (232957 bytes).
Received full file (233024 bytes).
Snappy compression: 233024 --> 232957 bytes.
Starting to send processed file (232957 bytes).
Received full file (532542 bytes).
Snappy compression: 532542 --> 479229 bytes.
```

Figure 15. Server works

```
vmakhambayeva3@advos-05:~/lab/tinyfileclient$ bin/release/tinyfileclient --client_id 777 --n_sms 3 --sms_size 32 --files filelist.txt --mode async
Current folder: bin/release/tinyfileclient
Parameters:
  - client id: 777
  - # of segments: 3
  - size of a segment: 32
  - mode: async
  - list of input files: filelist.txt

TinyFile client starting...
  - msg queue opened...
  - shared memory segments allocated...
Done.

Sent 'Tiny.txt' for processing
Sent 'Small.jpg' for processing
Working really hard...
Job (Small.jpg) is done. Received 232957 bytes.
CST for Small.jpg was 179.626 ms
Job (Tiny.txt) is done. Received 3057 bytes.
CST for Tiny.txt was 7.167 ms
Sent 'Medium.pdf' for processing
Sent 'Large.txt' for processing
Working really hard...
Job (Large.txt) is done. Received 877796 bytes.
CST for Large.txt was 676.026 ms
Job (Medium.pdf) is done. Received 479229 bytes.
CST for Medium.pdf was 342.538 ms
Sent 'Huge.jpg' for processing
Working really hard...
Job (Huge.jpg) is done. Received 1924917 bytes.
CST for Huge.jpg was 1323.710 ms
```

```
vmakhambayeva3@advos-05:~/lab/tinyfileclient$ bin/release/tinyfileclient --client_id 777 --n_sms 3 --sms_size 32 --files filelist.txt --mode sync
Current folder: bin/release/tinyfileclient
Parameters:
  - client id: 777
  - # of segments: 3
  - size of a segment: 32
  - mode: sync
  - list of input files: filelist.txt

TinyFile client starting...
  - msg queue opened...
  - shared memory segments allocated...
Done.

Job (Tiny.txt) is done. Received 3057 bytes.
CST for Tiny.txt was 9.473 ms
Job (Small.jpg) is done. Received 232957 bytes.
CST for Small.jpg was 138.749 ms
Job (Medium.pdf) is done. Received 479229 bytes.
CST for Medium.pdf was 345.261 ms
Job (Large.txt) is done. Received 877796 bytes.
CST for Large.txt was 682.155 ms
Job (Huge.jpg) is done. Received 1924917 bytes.
CST for Huge.jpg was 1561.038 ms
```

Figure 16-17. Clients work

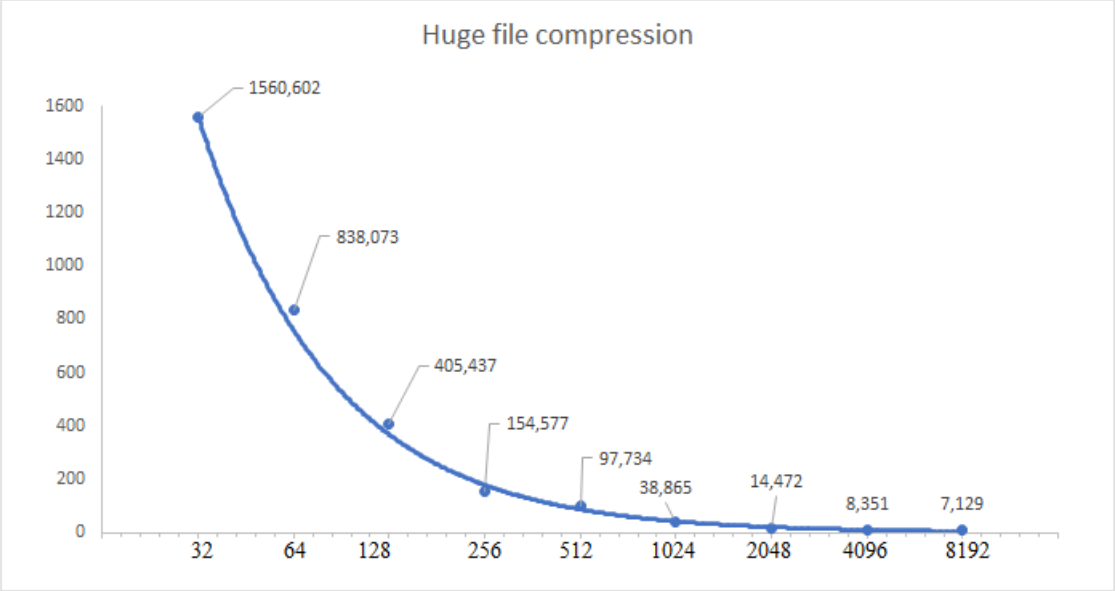


Figure 18. CST fluctuations along with segment size and the number of segments

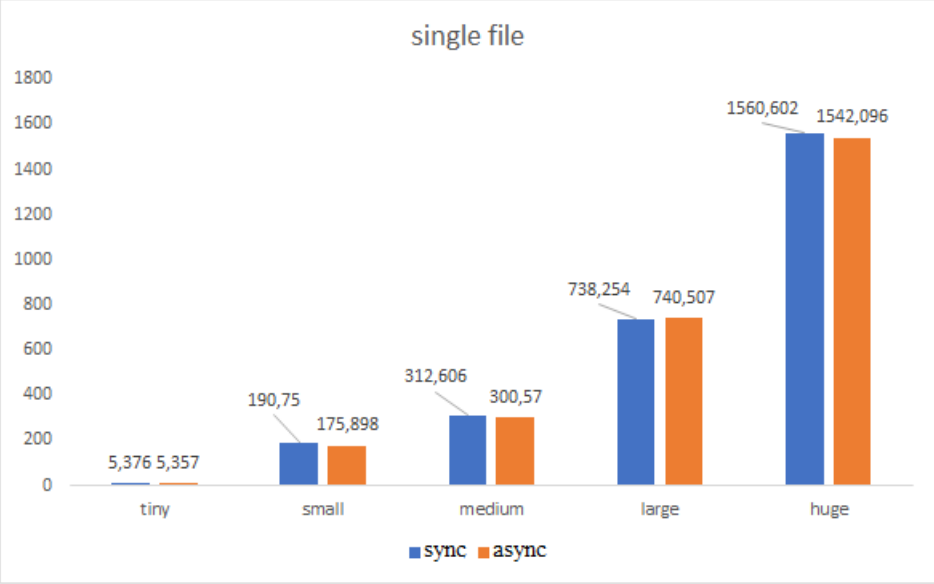


Figure 19. Processing time for one file depending on sync/async modes

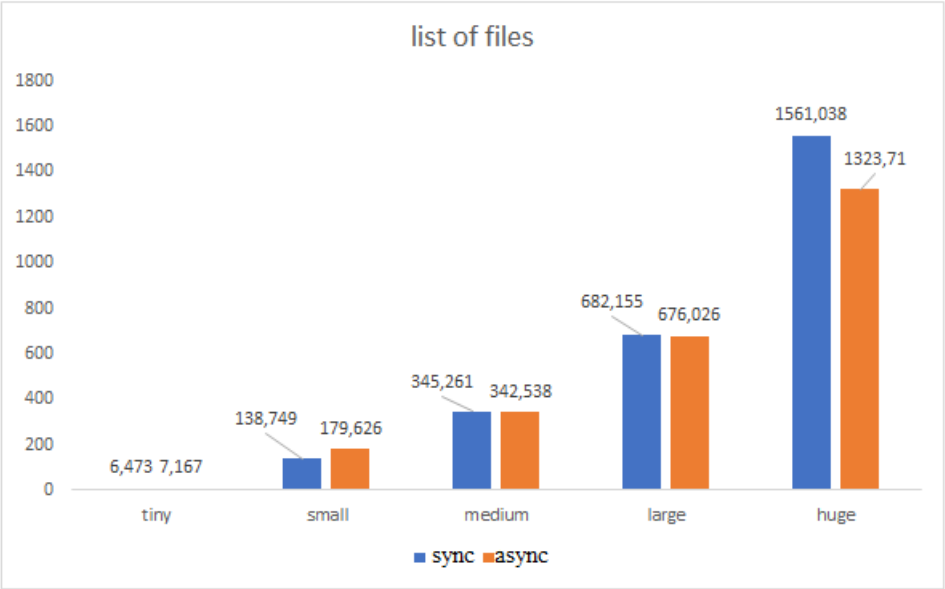


Figure 19. Processing time for list of files depending on sync/async modes

Conclusion.

In the experiments, I found that with an increase in the segment's size, the processing time decreases. With this design, the number of segments does not affect the processing time since each file occupies only one segment.

When sending one file, the difference in contrast to the transportation modes (synchronous, asynchronous) is not particularly big since the files are compressed quickly.

When sending a list of files, we can notice that large files are processed a little faster since, in the asynchronous mode, we can transfer several files in parallel. But because we are running on a single thread, we can't really get a lot of benefits.

This project has several features/disadvantages that I would like to highlight:

- I added a verbose global flag to common.h file. By default, it is set to false, and there is less noise on the screen when the code is running. It is easier to find, for example, records about how long it took to send the file. But if something breaks, you can enable verbose = true, and then it should be easier to find the reasons because it outputs all messages on both sides.

- It is necessary to set clientId on the client-side. ClientId can be figured out between server and client, but it will complicate protocol. For now, I expect each client to have a unique id assigned by the user on start.

- The server does not get into the shutdown () function now, so there is a need to manually exit (ctrl + C) and start it again. This may slightly affect the result, since the close_segment, delete_msg_queue, snappy_free_env functions are used on shutdown().

- The application is not split from the library and directory structure is changed. The entire process of how to run a program is described in the Readme file. There was not enough time for me to do it properly, I apologize for this. The make files were taken from open sources and modified slightly.