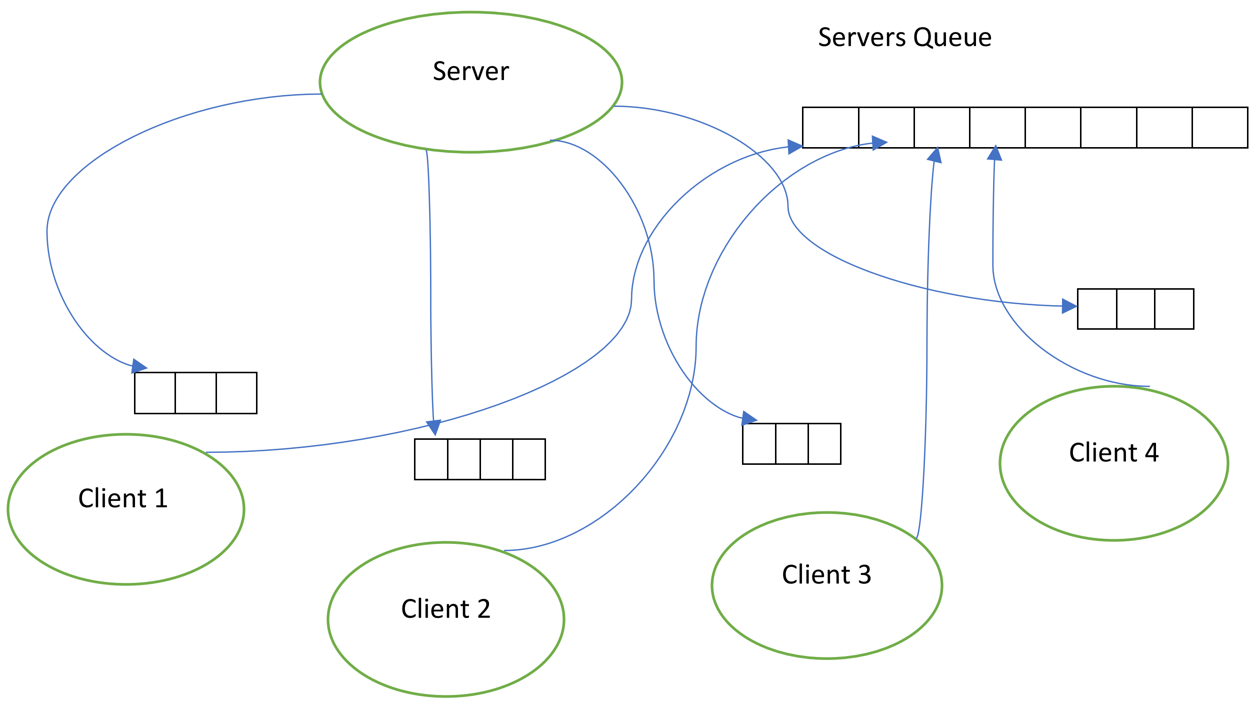
CS 6290 : Advanced Operating Systems

Project 2: Inter-process Communication Services

Sanjeev Rao(gt 903206772), Gauresh Vanjare(gt 903237912)

Design



The Amount of shared memory that can be made available to the application for compression is set by the admin who owns the server process. The client processes are responsible to create the shared memory according to the number of files it has and make sure that usage of the shared memory does not exceed the specified Cap size. Once the Cap for shared memory is set, the client processes will trigger a flow control mechanism that keeps track of the shared memory requested and total available shared memory for the service. Using a unique shared memory key the clients can copy their data onto the shared memory which is accessible by the server.

Based on the file sizes, shared memory is allocated dynamically and if the total requested size exceeds the shared memory size then we stall the application until the memory becomes free or deny service in some cases.

The Server Process that is running in the background will have its own message queue on which the clients will queue up their tasks. In the request message sent to the client, we encapsulate three fields, key of the shared memory segment of the file, client queue name and size of the file. The server starts processing the requests in the order of which the requests were made. Once the server is done with the compression, it returns a response to the client indicating the same using client's message queue. Each client's queue is identified using a client name descriptor tagged to the client, which is passed to the server using message queues.

Implementation

Shared Memory:

In order to facilitate the communication across processes, we are using shared memory segments created using System V library.

shmget(key_t *key*, **size_t** *size*, **int** *shmflg*): returns the identifier of the System V shared memory segment associated with the value of the argument *key.*

shmat(int *shmid*, **const void ****shmaddr*, **int** *shmflg***) :** attaches the System V shared memory segment identified by *shmid* to the address space of the calling process

int shmdt(const void ****shmaddr*):** detaches the shared memory segment located at the address specified by *shmaddr* from the address space of the calling process. The to-be-detached segment must be currently attached with *shmaddr* equal to the value returned by the attaching shmat() call.

int shmctl(int *shmid*, **int** *cmd*, **struct shmid_ds ****buf***):**performs the control operation specified by *cmd* on the shared memory segment whose identifier is given in *shmid*.

The first 3 API's are required to generate a unique identifier for the shared memory and to derive an address in order to access the shared memory.

Message Queues:

POSIX message queues for synchronization and message passing between the client and the server processes.

mq_open(const char ****name*, **int *oflag*, **mode_t** *mode*,**struct mq_attr ****attr***):** creates a new POSIX message queue or opens an existing queue.

mq_send(mqd_t *mqdes*, **const char ****msg_ptr*, **size_t** *msg_len*, **unsigned int** *msg_prio***):** adds the message pointed to by *msg_ptr* to the message queue referred to by the message queue descriptor *mqdes.*

mq_receive(mqd_t *mqdes*, **char ****msg_ptr*, **size_t** *msg_len*, **unsigned int ****msg_prio***):** removes the oldest message with the highest priority from the message queue referred to by the message queue descriptor *mqdes*, and places it in the buffer pointed to by *msg_ptr*.

A structure, which consists of a client generated identifier for the shared memory segment, size of the shared memory required and the name of the client message queue is passed between the client and server for the purpose of identification.

Pthread shared mutex:
As the admin sets the cap for the shared memory, we need to keep track of remaining shared memory in order to service further requests. We use a shared mutex and a size variable in order to maintain this information. When a request is issued, each client can access the shared size variable and reduce it using the lock. Once a client is done reading the compressed data from the shared memory, it can increase the available shared memory so that other stalled processes can issue their requests.

API’s exposed to the application:
The user can use simple API’s provided by our library service to compress the files required. Three API’s are developed in order to extend this functionality

1. BLOCK_API (char* filename, int Client Indentifier)
   This is a blocking call in which the user has to pass the filename and a client identifier which can be any number of the user’s choice. This is basically to distinguish which client is being processed. This call won’t return until the compression is done and saved on a predetermined folder.

2. NONBLOCK_API (Char* filename, int Indentifier)
   This is a non-blocking call in which the user has to pass the filename and client identifier which can be any number of the user’s choice. The call will return after sending the requests to the server. The user can wait on another call for the results.

3. Int WAIT_NONBLOCK_RESULT ( )
   This is a blocking call in which the user has to wait for the results from the server after it has issued the requests using a non-blocking call. This call will return with a 0 indicating the compression failed or 1 indicating that the compression was successful.

Safe Termination of the server and application
We have ensured that the server exits safely when ctrl+C is used. A signal handler has been implemented that will clear up all the shared memory created for mutex’s and any shared memory variable.

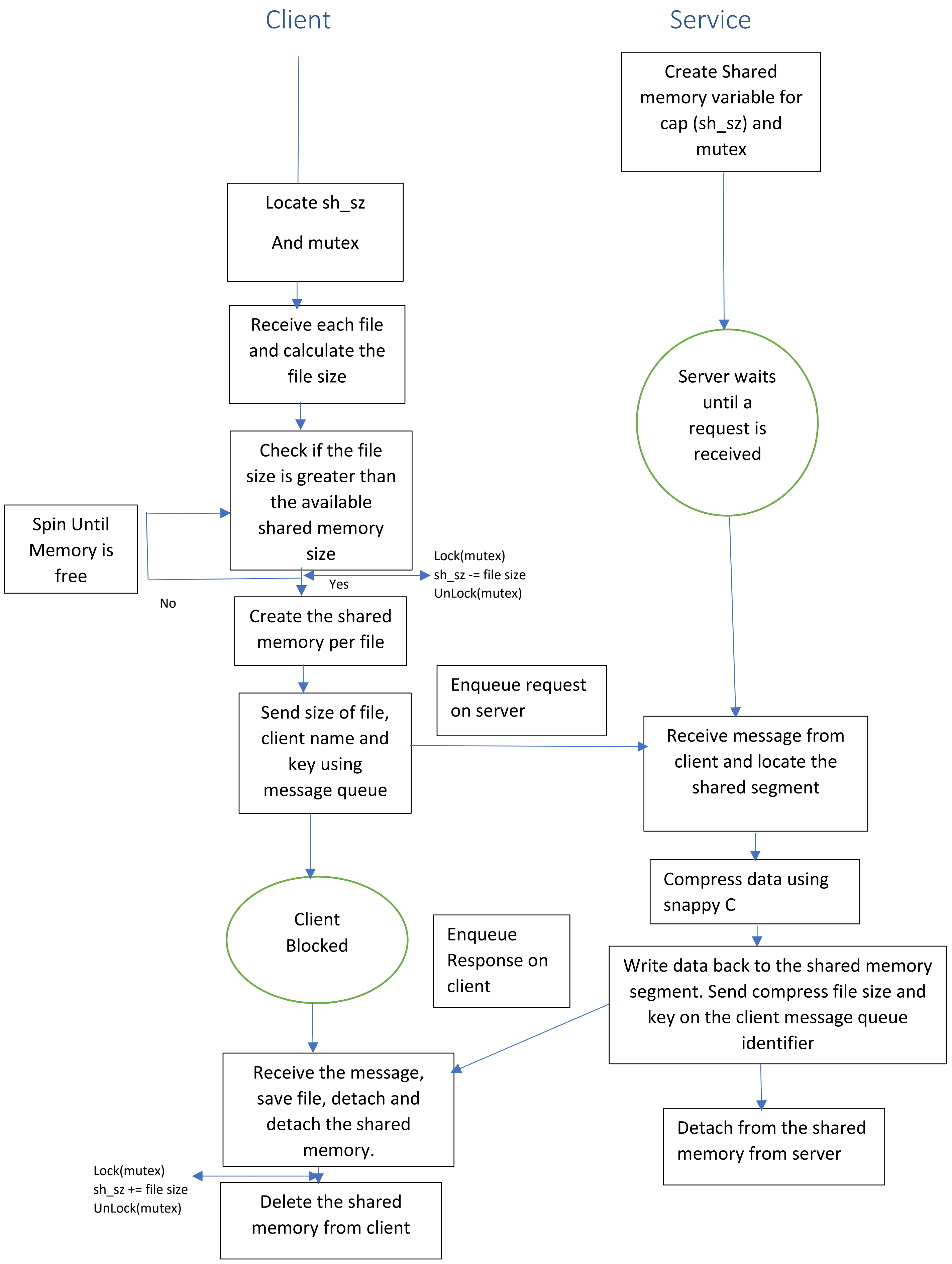
<u>Key notes about project:</u>

1. By using Dynamic memory allocation of the shared memory, we are avoiding unnecessary wastage of system memory.
2. Application processes requests are interleaved in the server queue to improve the throughput of the service.
3. A shared variable to track the total capacity using mutex gives necessary flow control.

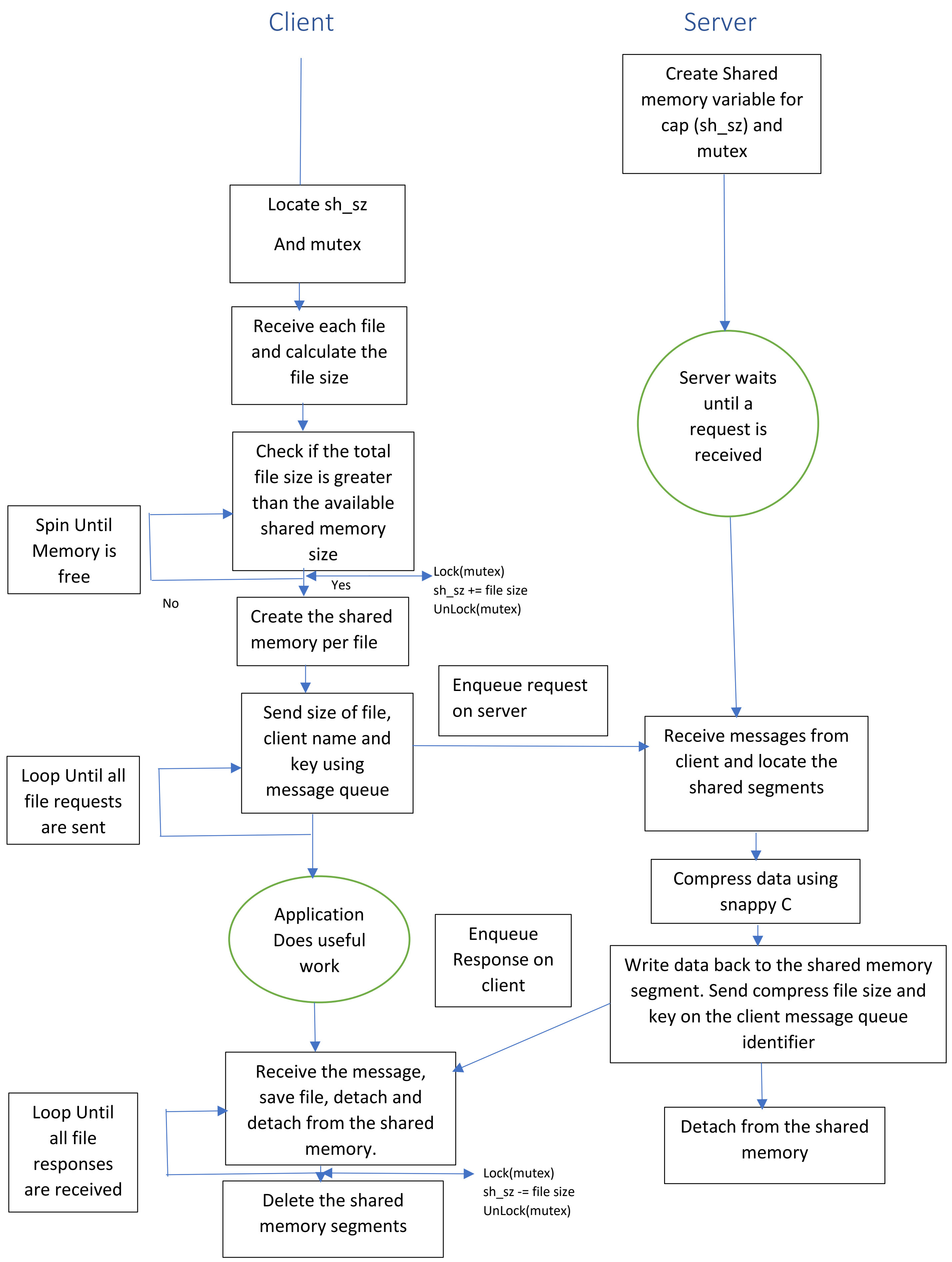
<u>Limitations:</u>

- ➔ The size of the files that require compression must be of length greater than 130 bytes(snappy limitation). As files that need to be compressed are usually large files, hence we do not support compression for small files.
- ➔ The Shared memory size that is configured by the admin should be of a greater size than the individual sizes provided by each client.

Flow Diagram of Synchronous API


Client
Service
Create Shared memory variable for cap (sh_sz) and mutex
Locate sh_sz
And mutex
Receive each file and calculate the file size
Server waits until a request is received
Check if the file size is greater than the available shared memory size
Spin Until Memory is free
No
Yes
Lock(mutex)
sh_sz -= file size
UnLock(mutex)
Create the shared memory per file
Enqueue request on server
Send size of file, client name and key using message queue
Receive message from client and locate the shared segment
Compress data using snappy C
Client Blocked
Enqueue Response on client
Write data back to the shared memory segment. Send compress file size and key on the client message queue identifier
Receive the message, save file, detach and detach the shared memory.
Detach from the shared memory from server
Lock(mutex)
sh_sz += file size
UnLock(mutex)
Delete the shared memory from client

Flow Diagram of Asynchronous API


Client
Server
Create Shared memory variable for cap (sh_sz) and mutex
Locate sh_sz
And mutex
Receive each file and calculate the file size
Server waits until a request is received
Check if the total file size is greater than the available shared memory size
Spin Until Memory is free
No
Yes
Lock(mutex)
sh_sz += file size
UnLock(mutex)
Create the shared memory per file
Enqueue request on server
Send size of file, client name and key using message queue
Receive messages from client and locate the shared segments
Loop Until all file requests are sent
Compress data using snappy C
Application Does useful work
Enqueue Response on client
Write data back to the shared memory segment. Send compress file size and key on the client message queue identifier
Receive the message, save file, detach and detach from the shared memory.
Loop Until all file responses are received
Detach from the shared memory
Lock(mutex)
sh_sz -= file size
UnLock(mutex)
Delete the shared memory segments

Evaluation

Test Cases

TEST1

Here we are testing Synchronous API (Blocking) with two clients and a server whose Cap is set to 100MB
Both the clients (App1 and App2 executable) have around 3MB of files.

Results and Comments:
Both the clients requests are handled atomically by the server as posix server message queue is atomic in nature. Hence Interleaving can occur between the clients requests to the server but each client will have to wait till the response is received for a request it sent to the server because of blocking API.

TEST2

Here we are testing Synchronous API (Blocking)with two clients and a server whose Cap is set to about 580 bytes
Both the clients (App1 and App2 executable)have gauresh.txt and Sanjeev.txt which are 577 bytes and 181 bytes respectively .

Results and Comments:
In this case the server can only handle one request from a client since the cap limit is approximately equal to file size of text file.
If one client is using the shared memory, the shared variable (size variable) is decreased and other client is blocked on creation of the shared memory.
When the server has serviced the client and when client reads back the data, shared variable (size variable) is incremented and the blocked client is allowed to use shared memory for the its file.

TEST3

Here we are testing ASynchronous API (Non Blocking) with two clients and a server whose Cap is set to about 10MB
Both the clients (App3 and App4 executable) have around 3MB of files.

Results and Comments:
In this case, only a single client has access to the server message queue to send all the shared segments keys, and file sizes. The application using this client stub will do meaningful work. The server will receive files one by one and process the request and put the response on the clients message queue. Whenever the Application is free it will call WAIT_NONBLOCK_RESULT() to collect the size and the compressed data.

At a time only one client is allowed to do the sending of files. Since the shared memory cap is 10MB, whenever a client sends all data to server, another client cannot acquire lock and send the data to the server Hence, in the later point of time, the applications will collect responses from their own message queue .

We are not allowing interleaving of requests here as there might be a case where both the clients put together might require more memory than the available shared memory. Instead of blocking both the clients at the same time, we can allow them to access the server one after the other.

TEST4

Here we are testing ASynchronous API (Non Blocking) with two clients and a server whose Cap is set to about 3MB
Both the clients (App3 and App4 executable) have less than 3MB of files.

Results and Comments:
Here if one client acquires the lock and sends all the files during this period, this will result in utilizing the entire cap size. Hence, other client's request is denied and the application puts up a message of server overloaded, please try again.

TEST5

Here we are testing ASynchronous API (Non Blocking) with two clients, Synchronous API (Blocking) with two clients and a server whose Cap is set to about 100MB
All the clients (App1, App2, App3 and App4 executable) have around 3MB of files.

Results and Comments:
This test is a mixture of both synchronous and asynchronous API running at the same time. Since cap size is very high, all the requests from the clients are processed by the service without denying any calls from clients. Also when asynchronous calls are putting files on the queue, the synchronous calls will not be able to put request on the server queue. But once the asynchronous call is done the synchronous API puts the data. Working on both the API gives desired results.

TEST6

Here we are testing ASynchronous API (Non Blocking) with two clients, Synchronous API (Blocking) with two clients and a server whose Cap is set to about 3MB
All the clients (App1, App2, App3 and App4 executable) have around 3MB of files.

Results and Comments:
This test is a mixture of both synchronous and asynchronous API running at the same time. Since cap size is less. All the requests from the clients are not processed by the service, this results in denying some requests from clients. Once Asynchronous API is done using its share of shared memory, Synchronous API will be given chance to proceed.

Note: sleep(sleep_time) variable is created in Client_API.h to observe the contention and interleaving between the application requests.(User can make it 0 also)