**Overall Design**

The service is implemented as a daemon process with workers that run infinitely. The client is implemented as a class that exposes sync/async APIs that can be linked into applications.

**Communication Protocol**

**Registration**

When clients initialize, they first have to register themselves with the server in order to create per-client message queues for later use. The protocol is as follows:

1) Client connects to globally shared message queue.
2) Client sends a registration message on the globally shared message queue.
    a) The registration message contains a message type identifier
3) Server sends an initialization message on the globally shared message queue.
    a) The initialization message contains a message type identifier, unique client id, and the number, size, and virtual size of the shm segments.
4) Client creates a private message queue for sending requests to the server later.
5) Client sends a queue creation message on the globally shared message queue.
    a) The queue creation message contains a message type identifier and unique client id.
6) Server connects to the private message in order to receive client requests.

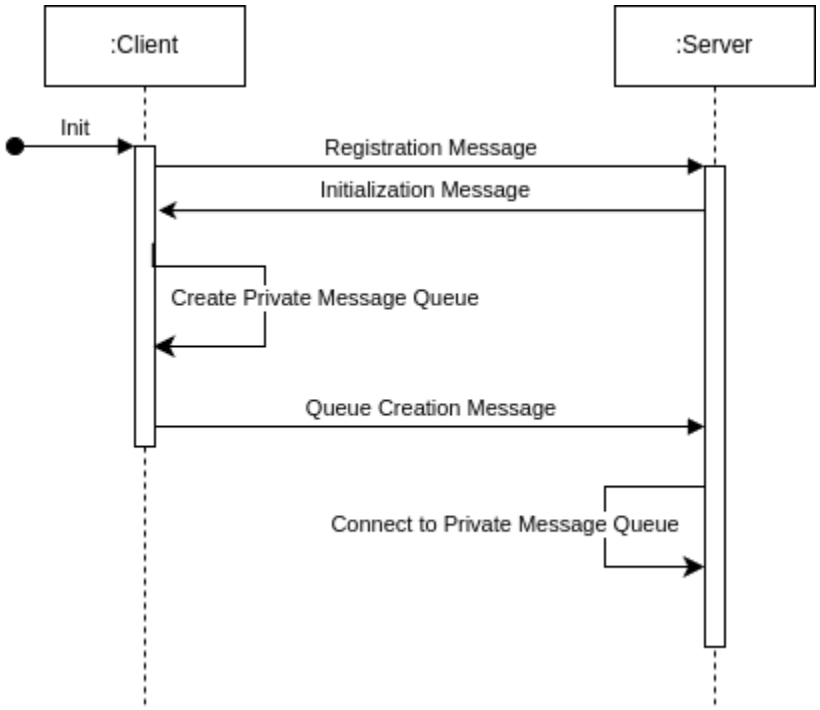


**Figure 1.** Sequence diagram for client registration upon initialization.

**File Compression**

After performing registration, the client can then send requests for file compression. The protocol is as follows:

1) Client sends a file compression request message on its private message queue.
    a) The file compression request message contains a message type identifier and file size.
2) Server allocates a corresponding buffer for storing the raw file and creates 3 semaphores for performing ping-pong synchronization with the client.
3) Server sends a file compression response message on the corresponding client's private message queue.
    a) The file compression response message contains a message type identifier and a list of virtual shm segment ids.
4) Client opens the virtual shm segments.
5) Client writes raw file contents onto shm.
6) Server reads raw file contents from shm onto local buffer.
7) Repeat 4-5 until the raw file is fully transferred.
8) Server performs file compression.
9) Server writes compressed file size onto shm.
10) Client reads compressed file size from shm and allocates a corresponding buffer for storing results.
11) Server writes compressed file contents onto shm.
12) Client reads compressed file contents from shm onto local buffer.
13) Repeat 11-12 until the compressed file is fully transferred.
14) Server closes semaphores.
15) Client closes and unlinks semaphores.

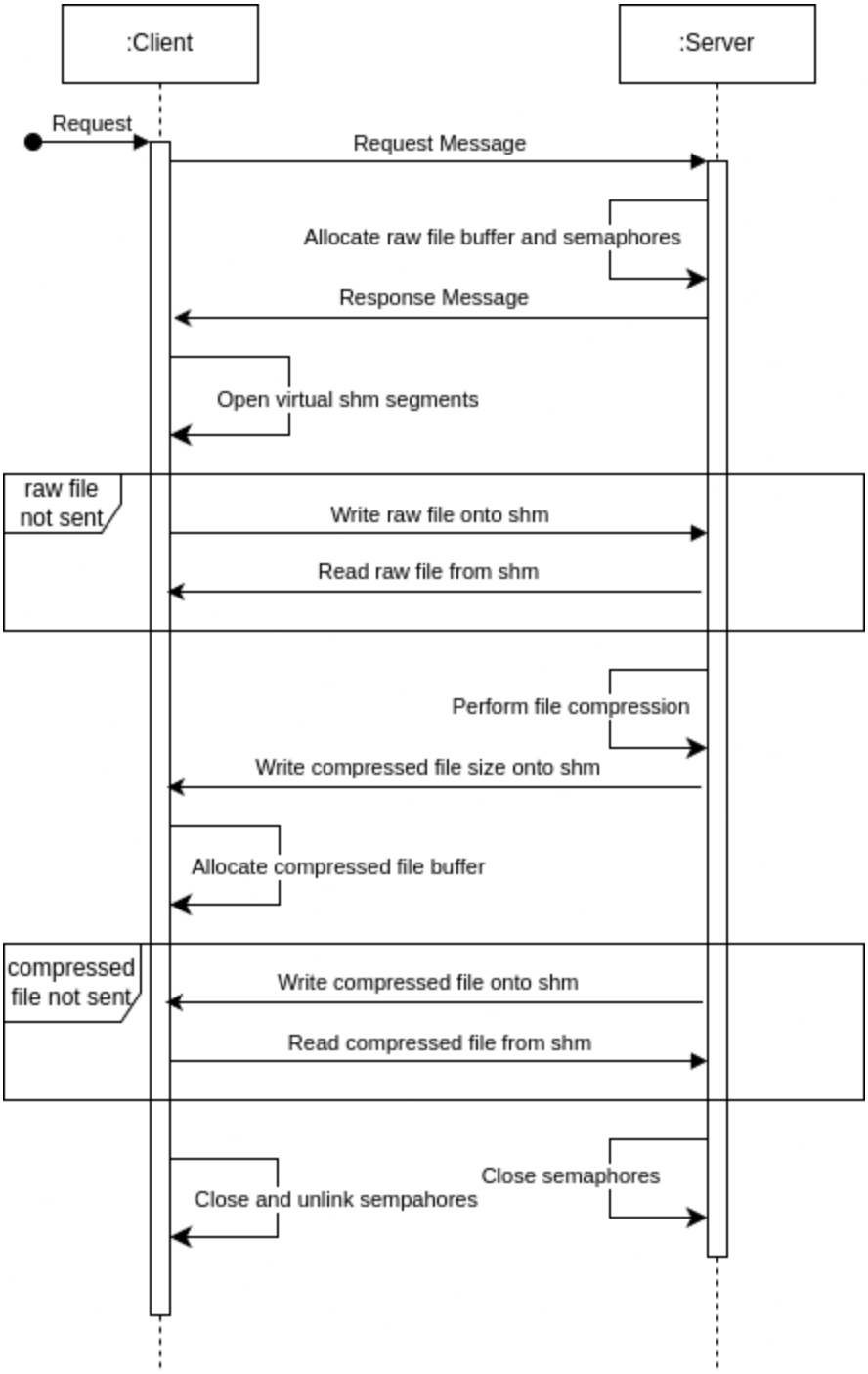


**Figure 2.** Sequence diagram for file compression request.

**Server Side Handling**

The server has one dedicated thread responsible for registering clients and scanning per-client queues for incoming requests. It simply alternates between these 2 tasks in an infinite loop. The server also has a configurable number of file compressor threads, which can process client requests. There is a singleton scheduler responsible for coordinating the registration and

worker threads. Each worker waits on a condition variable, which is signaled when new requests are received.

**Client Side Handling**

The client has one dedicated thread responsible for sending requests and receiving responses from the server on the message queue. The client also has a configurable number of file compressor threads, which can process requests by ping-ponging with the server. These are all contained within the Servicer class, which stores all necessary state. Each worker waits on a condition variable, which is signaled when new responses from the server are received. The client ensures that the number of requests in flight is at most the number worker threads it has, so it doesn't occupy shared memory without being able to process.

**Synchronous API**

The client exposes a synchronous API, where the caller supplies a pointer to data and the size of the data. The calling thread will place the request into a queue for scheduling and map for holding results inside the Servicer class. It will then wait on a condition variable, which is signaled when a worker has placed its results into the map. Note that this blocks the calling thread and waits for a worker to pick up the request. This adds additional scheduling overhead, but it was chosen for a simpler implementation. Otherwise, the client would have to demultiplex server responses when they're pulled off the message queue.

**Asynchronous API**

The client exposes an asynchronous API, where the caller supplies a pointer to data and the size of data. The calling thread will place the request into a queue for scheduling and map for holding results inside the Servicer class then return the request id. The calling thread is now free to perform other work until it wants to block and receive the compressed contents. The caller will then supply the request id and wait on a condition variable, which is signaled when a worker has placed its results into the map. Note that this also blocks the calling thread, similar to the synchronous API.

**Shared Memory Management**

Shared memory is managed by 32 byte virtual segments. There is a singleton shared memory manager that's responsible for distributing and reclaiming shared memory segments. Workers can request a size of shared memory, and the manager will provide a list of 32 byte virtual segments to fulfill the request. There is room for optimization here, since there are cases where the virtual segments are actually adjacent and could thus be compacted. There could also be more sophisticated allocation schemes that reduce fragmentation. However, it was determined that the overhead is minimal and not worth the additional complexity.

**Sample Application**

The sample application simply parses command line arguments and calls the corresponding function in the client side library. For async requests, it also prints a message after the request id has been returned but before it blocks to wait for the final result. The results were gathered by creating a benchmark file that listed Huge.jpg 10 times, so each function could be tested for multiple trials.

**Results**

The performance scales well with respect to the amount of shared memory allocated. Due to the virtual shared memory scheme, the performance scales with the total amount of shared memory because the virtual segments are treated uniformly. For example, the 3, 32-byte segment and 128-byte segment perform similarly. There are diminishing returns to increasing the amount of shared memory, because the reductions in overhead become smaller as there are less rounds of transfer.

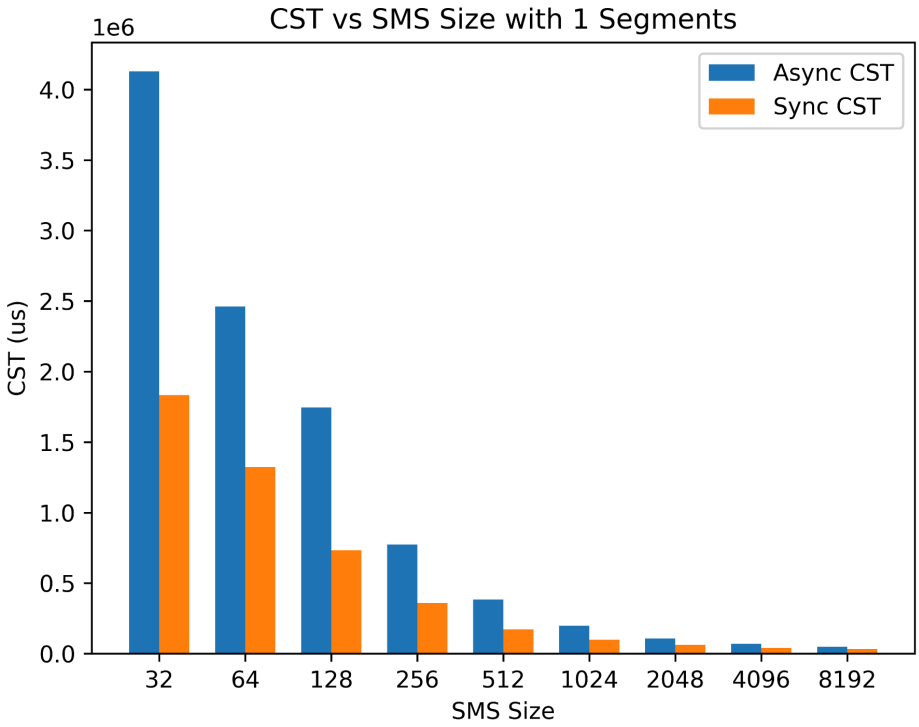


**Figure 3.** Mean client-side service time with 1 shared memory segment over 10 trials.

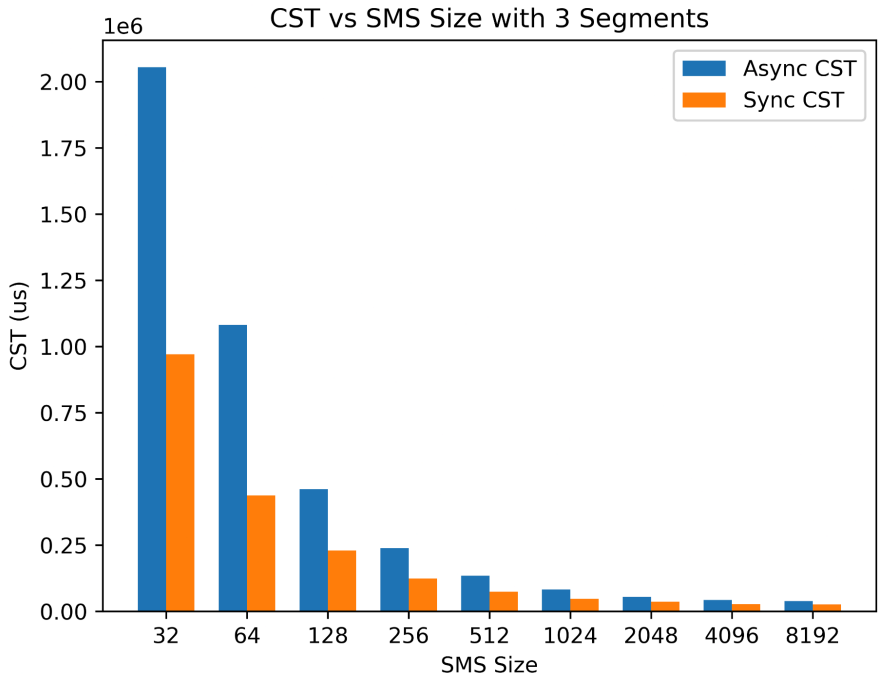


**Figure 4.** Mean client-side service time with 3 shared memory segments over 10 trials.

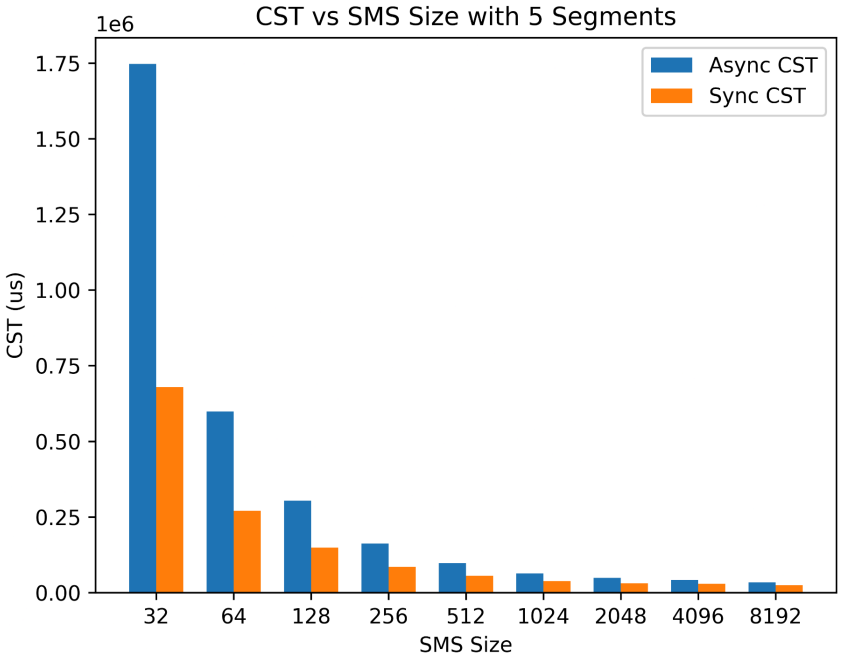


**Figure 5.** Mean client-side service time with 5 shared memory segments over 10 trials.

**Quality of Service**

The default scheduling policy is round robin. The runqueue is implemented as a map from client id to list of requests. The scheduler simply iterates over the keys in the map and

executes the first request in the next nonempty list. With this mechanism, it's impossible to ensure quality of service, because the scheduler doesn't take into account the size of requests. Clients with large file sizes will occupy more resources. Additionally, even if request size was uniform, there's no mechanism to achieve share based scheduling. Each client would get the same percentage of resources.

Quality of Service is implemented via credit scheduling. This ensures share based scheduling of clients with respect to the size of files that they process. Each client sends a priority to the server as part of its registration message. It will keep this credit weight for the rest of its lifetime. Credits are deducted and replenished based on the size of the file that each client requests.

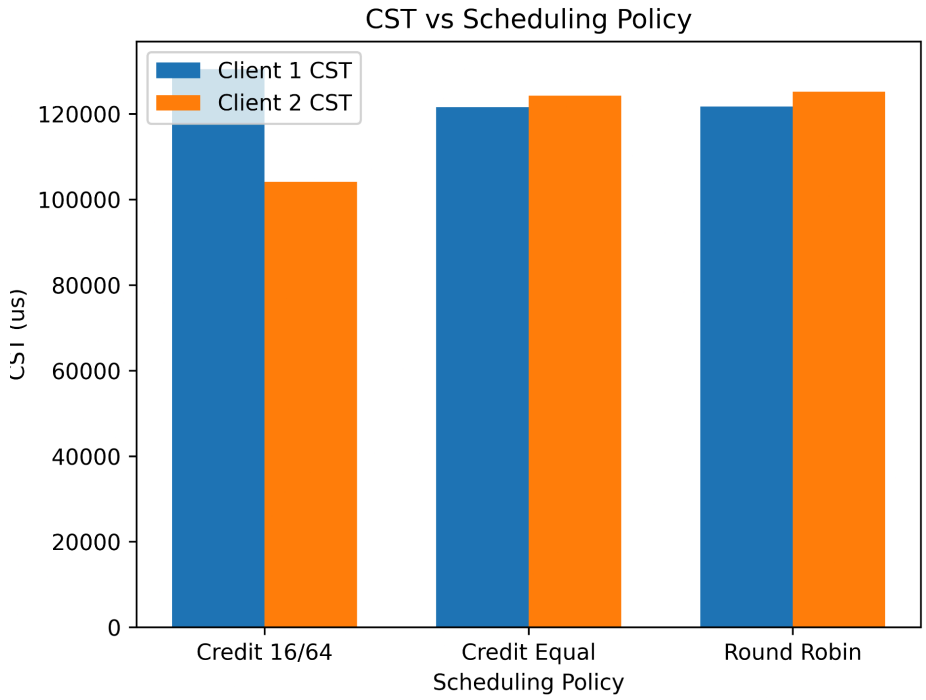


**Figure 6.** Mean client-side service time for various scheduling configurations over 10 trials.

There are two key observations to note. First, credit scheduling with equal weights and round robin achieve the same equal percentage distribution of resources. Second, credit scheduling with different weights achieves differential distribution of resources. Both are as expected. Note that the difference in service time is not linear in the difference in credits. This is likely caused by mistuned parameters in the scheduler, causing credits to go negative for both clients at times, resulting in round robin for some period. This could easily be addressed with better tuning. However, this was not completed due to time constraints.