GEORGIA INSTITUTE OF TECHNOLOGY

CS 6210 Project 2 Report: Shared Memory IPC

Manas George, Paras Jain

March 5, 2017

1 Project Goal

TinyFile is a service that accepts files through shared-memory IPC and then compresses/decompresses the data. Clients can communicate with the service and offload the work of compression to TinyFile. Behind TinyFile is an implementation of the `snappy` compression protocol.

In order to support inter-process communication between clients and the server, both a blocking and asynchronous interfaces are availible for end users. This interface allows for data to be efficiently shuttled between the client and server process through shared memory.

2 TinyFile Architecture

The TinyFile application is structured as a client-server program. The server is a standalone daemon that accepts requests to compress files from clients. The client is a library that is linked into client applications that need to use the application. Client applications call exported functions from the client library to send data they want compressed to the compression server running separately as a daemon, which responds with the compressed version of the file.

2.1 Server Design

The server `./bin/tinyd` allows users to start a daemon which listens for incoming compression requests. After initialization, the server alternates between servicing client requests to join/leave the service and actual compression requests.

Data Structures

The server maintains state for each client it is currently communicating with, and calls out to the snappy-c library to actually perform the requested compression. Client state is tracked in a linked-list structure that records, for each client, the private message queue and the shared memory segment used to perform client-server communication.

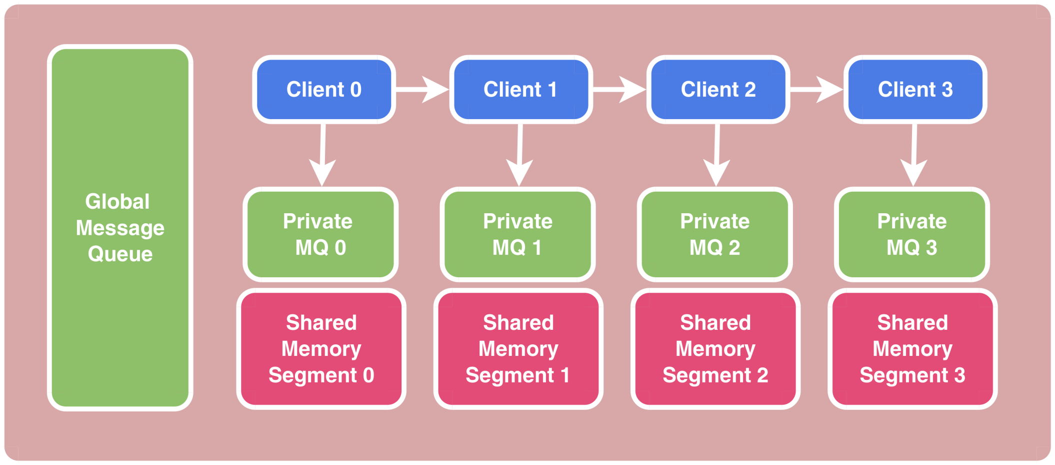

Figure 2.1: Server data structures

Apart from private message queues for each client, the server also has a global shared message queue that is available to all clients. This queue is used to initialize clients, as described in the following section, and for nothing else. Requests to join/leave the service can therefore be handled independently from actual service IPC traffic.

Client Initialization

When a new client comes online, it must register with the server using the `tiny_initialize` function exported by the library. The initialization function uses the global shared message queue to send an initialization request to the server, invoking a series of initialization procedures on the server side that comprise the initial client-server handshake.

1. Memory is allocated to store the new client's state.
2. The client is assigned new unique id and a temporary file.
3. The temporary file is used to derive an IPC key.
4. The IPC key is used to initialize a private message queue and shared memory.

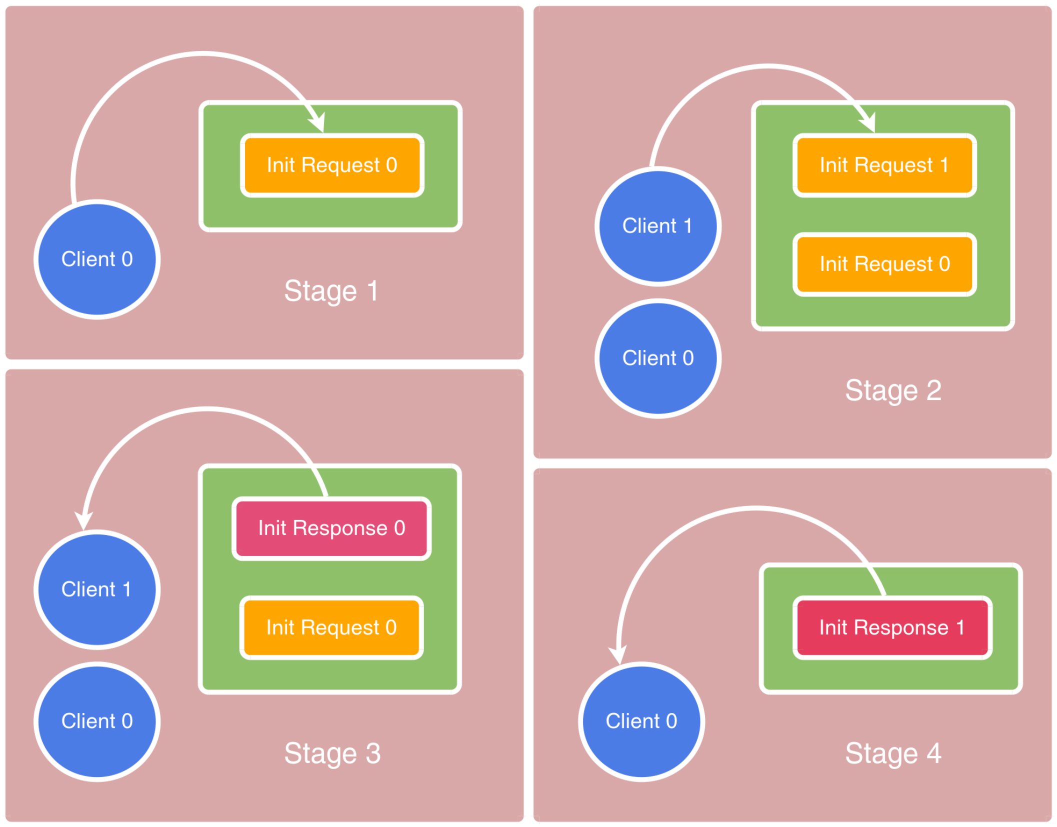

Figure 2.2: Initialization race condition

5. The new client is placed at the head of the global clients list.

Once all of this is done, the server responds with a message that contains the `client_key` created from the client's temporary file, which is the only information the client needs to connect to its private message queue and shared memory segment. The initialization request and response are sent over the global shared message queue common to all clients. This introduces the possibility of a race condition in since there is no guarantee that a particular response will reach exactly the client it was sent from. This is not really a problem, as the clients are essentially indistinguishable until the handshake is completed, and so a situation in which a client reads an initialization response message intended for another client is never problematic; the original client will simply receive a response message intended for yet another client, and all clients end up with unique IDs and keys in the end. Notice that mixing up of messages is not an issue once the handshake is done. The global shared message queue is only used for initialization; all subsequent messages between a server and client go through a per-client client-private message queue that is initialized during the handshake.

COMPRESSING/UNCOMPRESSING DATA

Once the client link is established, each client can communicate over its own message queue with the server. This private line contains requests from the client to the server to process a request. Service data is delivered through the shared memory and a response is finally communicated back through the shared store.

Specifically, the process of compressing a file using the blocking API is as follows:

1. The client reads the file into a buffer
2. Client calls `tiny_compress`, passing in the buffer and file size
3. The client library places the relevant arguments in the shared memory header `shm_header` and then copies the file buffer into the shared memory space
4. The client library then sends a message to the server to request decompression
5. When the server main loop services the relevant client queue, it invokes snappy to compress the buffer. It then copies the result back into shared memory.
6. The server notifies the client that compression occurred through shared memory
7. The client can copy the resulting compressed file from shared memory, thus freeing the region for use in the next call

CLEANING UP

Clients register themselves by sending a message on the main server IPC queue (not the private queue). The server recieves the message off the bus and then deallocated relevant data structures. This includes closing the message queue and marks the shared memory region for deletion. Clients can then quit. When a `SIGINT` signal is recieved, the system cleans up after each client before exiting safely.

3 API

3.1 Starting and running the server

The server can be started using `bin/tinyd`. This program takes several optional arguments: '`bin/tinyd -n N -s S`' where `N` represents the number of shared memory slots per client and `S` represents the size of each slot. The server now runs as a daemon until it is killed, during which time it will process requests.

3.2 Initialization and Cleanup API

Two functions control initialization and cleanup for the client:

- `tiny_initialize` initializes the connection with the daemon. Specificallt, this will load the main IPC message queue to the server, send a message to request to join the client group and then recieve private IPC information which contains a private shared memory mapping identifier, a semaphore for locking and an identifier for a private message queue channel.
- `tiny_finish` cleans up the shared memory region and unattaches it from its memory address space. Finally, it notifies the server so that various server-side data strutures and IPC mechanisms can be cleaned up.

3.3 Blocking API

There are several key functions in the blocking API for TinyFile:

- `tiny_compress` issues a request to compress some data and then blocks until the server has placed the results in shared memory. Finally, it copies the compressed data and returns the final buffer to the caller.
- `tiny_uncompress` operates similarly to compression, but requests the server uncompress the data rather than compress it.

This simple API allows users to quickly compress and decompress tasks in the server by simply linking a small client library. Performance tests of this library show that it is fast and efficient for the desired workload of 1MB files.

3.4 Asynchronous API

In addition to the calls provided by blocking API, we provide an asynchronous API which allows clients to continue to perform work while their file is being processed. The asynchronous API lets clients register callbacks when requesting a file be compressed or uncompressed. This made implementation less tricky while also keeping the API simpler. We provide two extra functions:

- `tiny_compress_async` which takes in a buffer and compresses the data, asynchronously invoking the callback argument when processing is complete.
- `tiny_uncompress_async` is very similar to compression.

Both functions call their respective blocking API counterparts. The calls are made asynchronous through the use of a call to `fork` which blocks on a separate thread. This made implementation simpler and also makes it easier for the server to manage requests as it doesn't need to keep around stale buffers whose clients have not retrieved their result.

3.5 TinyFile client application

All this functionality can be tested using the TinyFile compressor utility. This tool takes in a list of files as arugments and will compress the data and then uncompress the data again. Finally, it will verify the data is the same, thereby ensuring correctness of the compression algorithm. Timing events are printed which allow benchmarking the performance of the application.

To use the application, '`./bin/tiny [-a] file1 file2 file3 file4`' where `-a` is an optional argument that tests the asynchronous API. Files need not be unique.

4 Testing

4.1 Testing methodology

To test the program, we mix a variety of synthetic and real datasets. These include both binary and text files. For text datasets, we use public domain books. For synthetic data, we generated various order-of-magnitude datasets using `/dev/urandom`. This data should be difficult to compress and allows for stress testing the system.

4.2 Correctness evaluation

The code was correct on all inputs tested, specifically: `1bfile`, `10bfile`, `100bfile`, `1kfile`, `10kfile`, `100kfile`, `1mfile`, `2mfile`, `alice.txt`, `pride.txt` and `sherlock.txt`.

4.3 Performance evaluation

Performance was tested across a variety of file sizes. This was to understand what the bottlenecks in the algorithm were across various file sizes. Figure 4.1 displays the results of this experiment.[1]

Generally, IPC times are very consistent across various file sizes. This is a little surprising initially, given that there is a difference of 6 orders of magnitude between the largest and smallest test case. From 1 byte to 100 kilobytes, IPC times are generally very consistent. Times increase significantly when processing 1 megabyte and times increase slighty when processing 2 megabytes.

The reasoning for generally consistent times is that the IPC is occuring in two stages – first is the control signal which is passed through a System V IPC queue and second is the data transfer which occurs through shared memory. The first stage of IPC is generally going to be very fast. Shared memory speeds up the second stage as much as possible. For small transfers (< 1 cache line), this transfer is going to be very quick. We can see that 1 byte and 10 byte transfers are quite speedy[2]. The increase in latencies for 1 megabyte and 2 megabyte files may be due to cache effects.

Uncompression times are generally less than the respective compression time for most file sizes. This is likely due to two factors

1. The data to be uncompressed was just compressed, so much of the data probably already lives in the server cache
2. Compressed files are a bit smaller than the uncompressed files so there is a little less data to transfer.

[1] Asynchronous and blocking IPC latencies were very similar given our efficient implementation of the asycnhronous API using callbacks. Thus, Figure 4.1 only displays results for the blocking API.

[2] For such small transfers, it might be better to send data directly through the message queue. But it is unlikely that users care about compressing such small messages given the storage overhead of snappy.

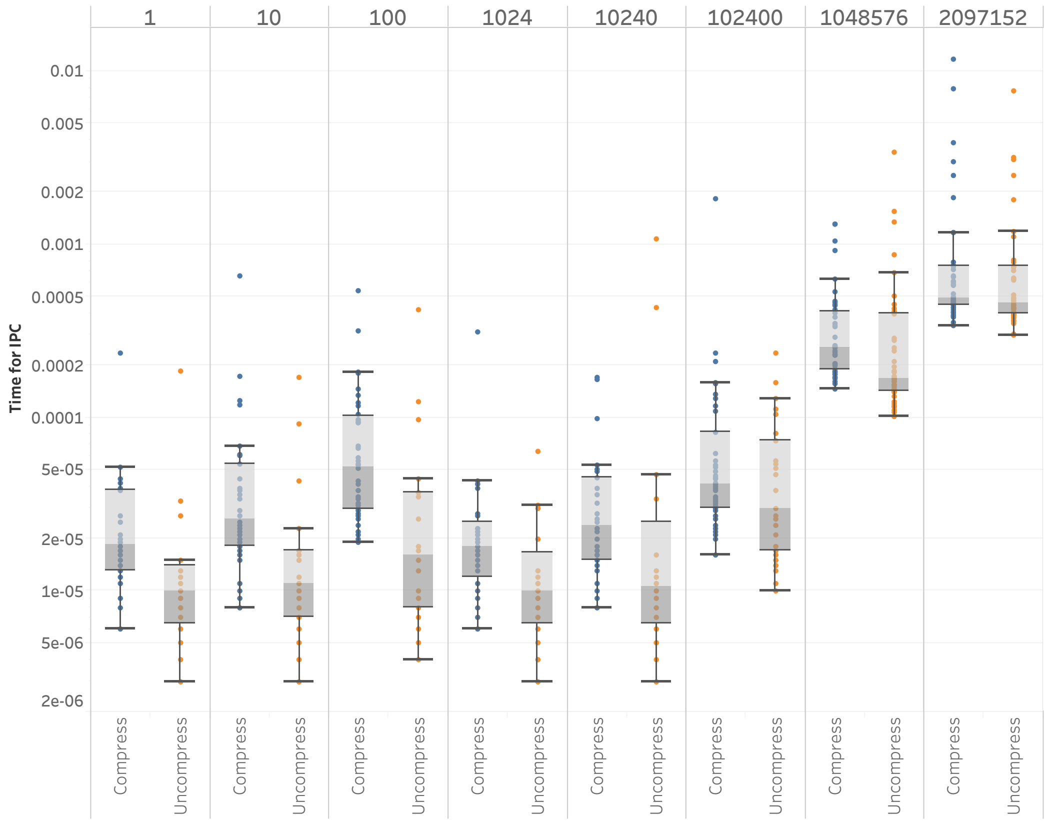

Figure 4.1: Round-trip IPC latencies for compression and decompression, across various file sizes. File size in bytes is labeled above. Times are in seconds. Note that the vertical axis is logarithmic.

5 Analysis of Engineering/Design Decision

Several key design decisions made during implementation include the use of:

1. **Multiple private message buffers**: Allows for simpler client/server communication, instead of multiplexing messages over a single message buffer to make sure they reach the right client. This also makes implementing QoS easier; the list of message queues just becomes a priority queue, so that higher priority clients are serviced first. The cost of this is that we now need to iterate over all of the message queues to find out if any of them contain a request to service, incurring a cost linear in the number of clients. In practice, it is unlikely that the service will see enough clients to make this an issue, although it would be possible to shift to a more complex single queue approach given more time.

2. **Multiple shared memory segments**: Given the guarantee of small file sizes (at most 1MB), using multiple shared memory segments for a single client is wasteful, and so we initialize a single relatively large (10MB by default) shared memory segment per client. We could have used a single segment across all clients, but this would result in large multiplexing/demul-

tiplexing overheads and result in additional complexity. This could also open the service up to security vulnrabilities with decreased isolation (i.e. buffer overflow attacks).

3. **Callback API for asynchronous calls**: We chose to use callbacks instead of having the client call a function to retrieve results later as the implementation is simpler from both the client perspective and the service perspective. This is due to asynchronous calls being thin wrappers around the blocking calls.