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.

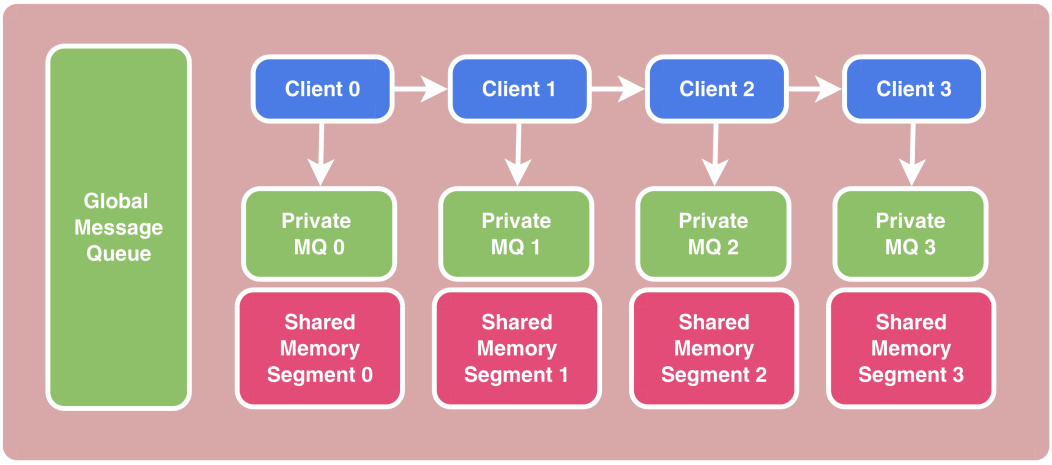

Figure 2.1: Server data structures

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.

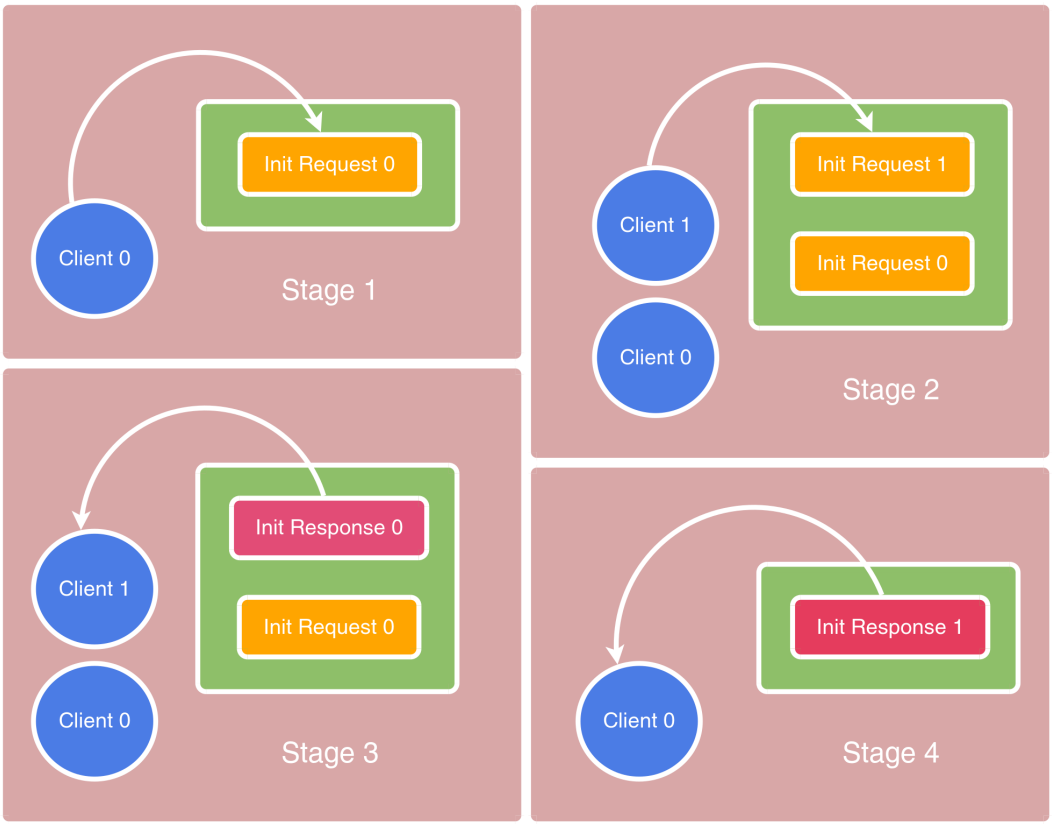

Figure 2.2: Initialization race condition

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 degister 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 and marking the shared memory region for deletion. Clients can then quit. When a `ctrl-c` signal is recieved, the system cleans up after each client before exiting safely.