GFS-python: A Simplified GFS Implementation in Python

Andy Strohman

ABSTRACT

GFS-python is distributed network filesystem written entirely in python. There are no dependencies other than Python's standard library. The goal of the project is to become familiar with the distributed nature of GFS and implement features necessary for its intended workload. To that end, GFS-python implements chunk replication, server failure resilience, multi-chunk read/writes and producer/consumer record append functionality.

1. DESIGN OVERVIEW

There are three major components to this project; a master server implementation, a chunk server implementation, and a client library. The client library provides for a way to access the filesystem. Optionally, the programmer may use an record-based interface that can discard duplicates that result from failures that occur at secondaries during the atomic append operation. The chunk and master server implementations are multi-threaded with a RPC interface that is similar to what is described in the Google's GFS paper.

2. CLIENT API

The client's interface is centered around a file handle object. The user creates the file handle by specifying the master server name and a file path. Once the file handle is instantiated, the user can create and delete the file corresponding to the path used during object construction. The user also uses this file handle to read, write and append.

When reading, the user specifies an offset within the file and the size of data that he wants to read. The library will block until all the data is read or the end of file has been reached. If the user receives less data than what he requested, this means that he has reached the end of the file.

When writing, the user specifies an offset within the file and the data to be written. The user may specify any existing offset within the file for a write. But it is currently impossible to seek past the end of the file for a write unless the offset is within a chunk that already contains some data.

The atomic append works exactly as described in Google's paper. The size of the write has a size limit of 1/4 chunk. If the append would exceed the chunk size, all replicas are padded, and the client library is informed to retry the append on the next chunk. Like read(), write() and append() are blocking operations.

3. SYSTEM INTERACTIONS

The data flow is very similar to what is described in Google's paper. Clients ask for chunk IDs and their corresponding location(chunk servers) from the master.

For writes, the master arranges leases with primary chunk servers on a per-chunk basis. There is a global LRU cache on each chunk server that stores data to be written. How the write data is distributed, and how serial numbers are assigned is a bit different in GFS-python than Google's implementation. For simplicity's sake, the client first sends the data to the primary. In turn, the primary assigns a serial number for the write, and responds to the client with the assigned serial. The client then sends the data directly to all the secondaries, along with the serial number. After this point, things return to the standard GFS behaviour. The client instructs the primary to write the data. The primary writes the data, tells all the secondaries to write the data, and returns the outcome of these write attempts to the client. If the client sees a failure, it retries.

For reads, the client tries all the chunk servers for the desired chunk in a round-robin fashion until it succeeds. The chunk servers ensure that they are on the correct version of the chunk before responding.

For appends, the process begins similarly to writes. Clients distribute the chunk to be appended to primary, which assigns a serial. The client then transmits the data to all the secondaries. The only difference with a standard write operation at this point, is that the client has not specified an offset within the chunk to write. Subsequently, the client tells the primary to append by specifying the serial number. As in Google's implementation, the primary ensures that there is enough space in the last chunk for the append. If not, it pads the last chunk, tells all the secondaries to do the same, and informs the client to retry the append on the following chunk. When there is enough space available in the chunk, the primary appends to end of the chunk and tells the secondaries to write the data at the same offset within the chunk as it had written to. Finally, the master replies back to the client, telling it whether all replicas succeeded in writing the data or not, just as with a standard write operation. This feedback informs the client whether it needs to retry or not.

4. CLIENT OPERATION

The file handle object holds a cache of chunk ID location information which helps decrease client to master interactions. When the client detects an operation's failure due to chunk server failure, it will refresh its cache in hopes of being able to retry and complete the operation. It keeps retrying until it succeeds.

When reading records, the client always attempts to read as much data as it can from a chunk. It caches this greedy read opaquely within the handle. When the client asks to read the next

record, the library first tries to find the record in the cached data previously read. If the next record is not found within the cache, the client will again read as much as it can, and cache it. When reading successive records, the client library can detect when it has reached the end of the chunk. It will then move onto the next chunk.

When using the record based interface, record data is encapsulated within a header which includes a beginning of record marker("BOR\0"), the record length, and the record checksum. This, however, does not protect clients from seeing duplicate records that result from append failures.

For clients that cannot cope with duplicate records, there is yet another interface layered on top of the record interface, the de-duplicate record interface. This layer uses sequence numbers and producer IDs to discard duplicate records. Producer IDs are used in multi-producer scenarios. The idea is to enable multiple independent producers to append to the same file. The producer IDs allow producer's records to be distinguished from each other so that they can be tracked independently for the purpose of de-duplication.

5. MASTER OPERATION

As previously mentioned, the master process is the multithreaded RPC server. This means that each RPC request is serviced in its own thread to allow for concurrency. The master identifies all chunk servers during initialization by querying a special DNS name, "chunkservers." After the set of chunk servers has been determined, they are queried for all the chunks that they own. This way, the master builds the state of all chunk locations during initialization.

In addition to spawning threads on a per-RPC basis, the master runs a thread for each chunk server. These per chunk server threads periodically heartbeat and collect information about what chunks the chunk server holds. The master uses these threads to extend leases for primaries as well.

There is also a re-replication thread. This thread scans all chunks to ensure they are at the proper replication level. On a per chunk basis, this thread checks that all the chunk servers that are supposed to hold the chunk continue to do so, and are still alive. If not, it will update the version number of the chunk, so that writers will be able to mutate the chunk. If it sees that a chunk is below the desired system-wide replication level, it will take the lease for the chunk so that it may be safely re-replicated. Once it has secured the lease for the chunk, it instructs available chunk server to replicate.

During both chunk creation and re-replication, the master takes relative chunk usage into account when selecting a replica for storage. It attempts to balance the amount of chunks stored evenly across all available chunk servers.

The master stores two sets of metadata persistently on the disk. The first is chunk metadata, which just includes the chunk ID to version mapping. This is stored in an always increasing vector. When chunks are deleted, their ID's on the disk are set to -1 to signify they are no longer valid. There is no compaction of the log, but this could be easily implemented in future. When the master process initializes, it reads this chunk metadata and store the offset of where each chunk lives within the metadata vector file in an in-memory data structure. This way, chunk version numbers can quickly be updated persistently to disk.

The second set of metadata stored on disk contains information about all existing files, including what chunks belong to each file. The filesystem hierarchy is stored within a configurable directory. Each file within GFS is represented by actual directories and files within this configurable directory. So, for example, if the GFS-python administrator configures the master server to store this metadata at /gfs/data, and there was a GFS file with path /directoryName/fileName, then there there will be a file on the master server at location /gfs/data/directoryName/fileName, which holds all the chunk IDs for that file. When the master process initializes, it walks the directory for this metadata and loads all the file paths and corresponding chunk IDs into an in-memory data structure for fast lookup.

6. CHUNKSERVER OPERATION

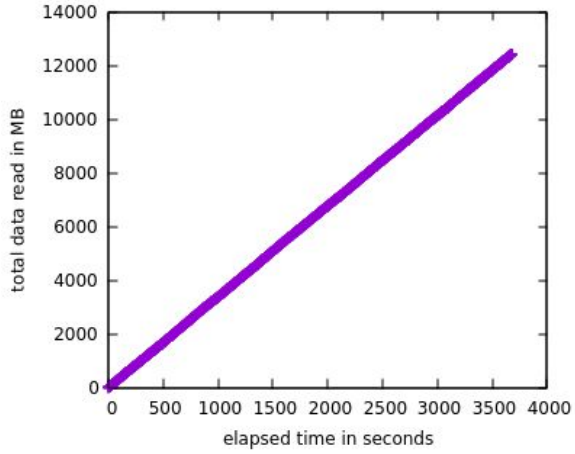
Like the master server, the chunk server is a RPC server that spawns a new thread for each remote call. Clients and other chunks servers make these calls. As with Google's implementation, each chunk is stored in its own file. Read and write access to chunks are serialized by a per-chunk lock to avoid corruption. Each time a chunk is updated, the corresponding chunk's checksum is updated. These checksums are stored persistently on disk. For every read and write operation, the checksum of the chunk is verified before allowing the operation to continue. If a bad checksum is discovered, the chunk is discarded. Unlike Google's implementation, there is no background thread that scans inactive chunks for bad checksums, but that functionality can easily be added in the future. During chunk server process initialization, metadata is read from the disk to an in-memory data structure which holds chunk ID, version and checksum information.

7. MEASUREMENTS

A test environment was created in order to understand the throughput in producer/consumer record append workloads. The test infrastructure consists of 9 Ubuntu VMs on a freeBSD VirtualBox host with an Intel(R) Core(TM) i5-4670K CPU @ 3.40GHz. Each VM was allocated 1GB of memory, 100GB of disk, and 1 virtual CPU. The

chunk size was set at 32MB and the chunk server LRU cache size was limited to 128MB. Of the 9 VMs, 6 acted as chunk servers, 2 as clients and one as the master. A producer client appends random sized verifiable records, while the consumer client reads these records. The consumer client tails the producer, reading new records as soon as they are available. It verifies that the record contents are correct, including an incrementing sequence number. The consumer keeps track of the total amount of data received over time. Figure 1 shows the the results.

Figure 1



8. FUTURE WORK

As the goal was to implement as many GFS features as possible and to only use the python standard library, many performance sacrifices were made. CPython's GIL prevents threads from running concurrently. However, this is not too bad, considering that threads are I/O bound. So it seemed desirable to move forward with a threaded design to compartmentalize functionality and allow for multi-chunk concurrency.

However, having the master spawn one thread per chunkserver does not scale. So, the implementation could benefit from more of an event-driven approach as opposed to multi-threaded. In addition to the scaling benefits of a event-driven strategy, locking could be simplified or perhaps removed altogether.

Python's standard library only provides one RPC interface, xmlrpc. This is a poor choice for this application, as huge amounts of binary data is being passed via RPC. The binary data is base64 encoded to be compatible with XML, which significantly increases the network load. Python's standard library does include a XDR implementation, but no corresponding RPC interface. Historically, there was a RPC demo for XDR included with CPython's source, but it was removed in early 2011.

Python's threading implementation does not include direct support for read-write locks. Although there are ample examples on how to create read-write locks using locks and condition variables, many of them suffer from the possibility of writer starvation. GFS-python would benefit from read-write locks to protect chunk data access.

9. CONCLUSIONS

GFS-python was created to better understand the distributed aspects of GFS. It includes master, and chunk

server processes along with a client library for interacting with the distributed filesystem. GFS-python has achieved the goals of implementing multi-chunk read/write and multi producer/consumer atomic append. It can also can survive chunkserver failures and master restarts.