MP3: Simple Distributed File System

Aarya Bhatia <aaryab2@illinois.edu>

Architecture

Each node in my distributed file system comprises three layers of servers implemented in *Go* language: a *failure detector* on UDP port 6000, a *backend server* on TCP port 5000 and a *frontend server* on TCP port 4000. These servers run inside one process as different *goroutines* (threads) at each node. The client process communicates with only the frontend layer to initiate an upload or download. The frontend always gets the leader node from the backend and then communicates with the leader to schedule the task. The leader blocks the client (frontend) based on its read-write policies. Later, the leader sends the client a list of replicas and the frontend directly communicates with them to upload or download file "blocks". This helps to balance the load of the leader and shifts the bottleneck from the leader to the bandwidth of the frontend nodes.

Each node maintains a full membership list, which is updated by the internal gossip membership protocol (from *MP2*). The node with the lowest address is chosen as a leader. The leader node uses a read-and-write queue for each file (rather than each client). A read or write request is immediately enqueued on arrival. The queue has two modes. In the "reading mode", the leader only permits read access to at most two clients in parallel. Similarly, in "writing mode", the leader permits write access to at most one client. The mode is switched when either there are no more requests in that queue or if there were 4 consecutive operations of the same type. This strategy prevents read-write starvation and allows mutual exclusion per file.

Sharding: Each file is shared to blocks of size 16MB (configurable). Each file is associated with a version number (initially 1), indicating the latest update of the file. Each block is stored on disk as "filename:version:blockNum". So, we can distinguish between stale file blocks on nodes and allow caching of file blocks (future work). The leader knows which replicas currently store each block of the file.

The nodes that store the file metadata (such as name, size, version, replica-block mapping) are the backup leader or master nodes. The nodes that store the block data on disk, are the replica or slave nodes. Each metadata and block is replicated to atleast 4 other nodes since we assume to have at most 3 simultaneous failures. The backup leaders are selected by the lowest IDs (address = host:port). To select the replica nodes we perform the following algorithm: We compute the absolute difference between the hash of the filename and the hash of the node address, and choose the 4 nodes with the lowest values. Any node can act as both master and slave simultaneously (for simplicity).

To upload a file, the client (frontend) sends a UPLOAD_FILE command to the leader. On receiving the replica list for new file (format: blockName replicas[,...]), the client sends UPLOAD commands to each replica and multiplexes the connections to upload all blocks in one shot rather than reestablishing TCP connections many times. The replicas write the block to disk and send an OK message back to the client. When all blocks are uploaded, the client sends an ack to the leader. The leader replicates the block metadata to 3 other backup nodes. Finally, the leader can dequeue the next task from the queue forward and update the file metadata as required. An upload is aborted if any replica fails to acknowledge at any point. The client can retry with an exponential back-off, allowing time for failure handling at the leader.

To download a file, the client sends a DOWNLOAD_FILE command to leader. On receiving the replica list, the client can choose any replica for each block and send a DOWNLOAD block command to initiate transfer. All the blocks are concatenated and written to the local file at frontend node. Again, the client acknowledges the download to the leader, and the leader can mark the read finished.

Deletes are treated in the same way as writes (uploads). The client uses the DELETE_FILE command to begin the deletion. The leader itself sends a DELETE command to all the replicas storing either data or metadata and waits for an acknowledgment. On success, the leader removes the file entry.

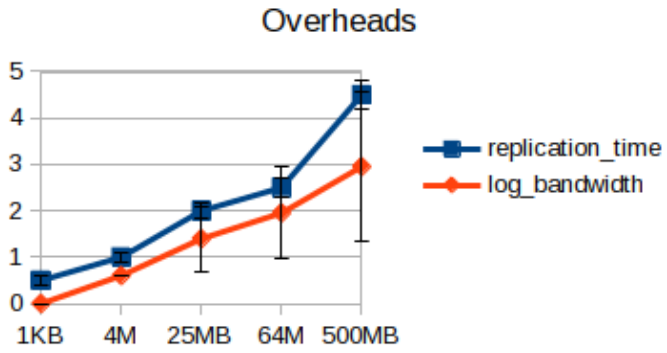
The leader nodes perform a replication task in the background. Periodically, they check if a replica is missing a block. The ADD_BLOCK command is used to inidicate to the replica that it should get the block from any of the replicas (listed in the message). The node performs a DOWNLOAD task and acknowledges the leader when it succeeds. For updating metadata, the SETFILE and SETBLOCK command is used.

The failure detector informs the backend node on joins, leaves or failures, through a local function call. For these events, the backend server has to update all of the metadata associated to the node, and possibly re-replicate some blocks via the rebalancing routine.

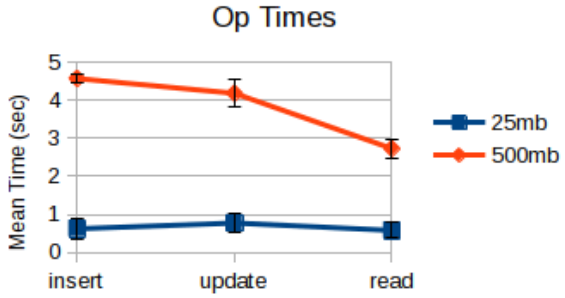
My modified version of *MP1*, a distributed "*shell*", was extremely useful to run arbitary shell coammnds across all nodes such as tailing log files while the server ran in the background. I also used this shell to send start/stop commands to the servers while testing.

Experiments

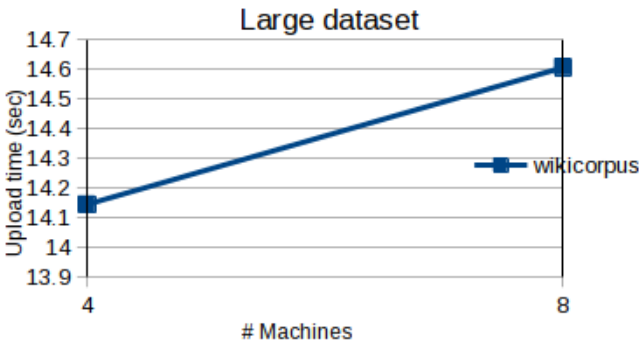
The following plot measures the overhead of replication time in seconds and background bandwidth in MB/sec using the "iftop" program. We observe a steady increase in overhead with increasing file sizes.



The next experiment measures the time for three operations with two file sizes. As expected, downloads are faster than uploads.



I measured the upload speeds for the entire 1.3 GB Wikipedia archive with different numbers of machines. A small linear growth is observed in the following plot given the overhead of establishing more TCP connections and possibly longer routes between some machines.



The following experiments were performed with a 6.0 GB file using 10 machines. A typical read time for this file is 32 seconds and a typical write time is 73 seconds. This large difference occurs because the frontend servers need to replicate each block 4 times. On the other hand, the downloading is much faster because each block is downloaded from a single replica source only.

The read-wait plot measures the finish time for n readers on a logarithmic y-axis. The overhead is observed to increase with the number of readers. For the write-read experiment, we store 6.0 GB non-identical files on each of the 10 machines, and perform a read operation from one VM, and a write operation from n other VMs in parallel. The following plot measures the finish time for n writers and 1 reader on a logarithmic y-axis. In this case, the overhead is increasing more rapidly than the read-wait plot, because for write operations there is a load from the background replication tasks and deletion of older files.