CSE 511 Project 2 Report

Distributed Shared Memory System

Name- Niramay Vaidya
PSU ID- 939687597

Table of Contents

1. Overall logic of the implementation-

Distributed mutex locks- Every participating node will first call a poll RPC to every other participating node (based on the entries in the node_list file, except the last one since that is not of use here) to check if everyone is up and running or not. Every node will also check first if it is a valid node or not (the system it is running on has its hostname in the node_list file or not). All of this (including the start of the server), opening of the log file and start of the client happens in start_lock. init_lock just initializes the context specific to the mutex lockno passed in (all the required variables and data structures). The actual Ricart Agrawala algorithm is covered in the mutex_lock and mutex_unlock functions. Within mutex_lock, all the required steps (setting requesting_cs, updating seq_num, etc.) are done before a recv_request RPC call is made to every other node, where its implementation contains updating highest_seq_num and then the main condition check which determines whether to respond or to defer. Either way, the RPC returns with an appropriate value. Back in the current node, it is continually checked if responses from all the other nodes have been received or not, and if they have been received, return from mutex_lock. Within mutex_unlock, there is a sequential check for seeing if there were any replies deferred previously for other nodes, and if there are any, send_def_reply RPC calls are made to those nodes (along with this, requesting_cs is unset). In the implementation for this RPC, the local specific responses data structure is updated by all the other nodes. Since there is a requirement for multiple locknos to be supported, all the local variables and data structures required per lock are expanded to now be keyed with a lockno instead (essentially, maps are used with locknos as keys). A simple test application for correctness of this implementation contains the critical section surrounded by mutex_lock and mutex_unlock to print current timestamp, sleep for a fixed amount of time, then print the hostname followed by the current timestamp again. The test for correctness is that the period between the timestamps should not be overlapping across nodes.

Distributed shared memory system- As explained before, there is a poll RPC here too, but in this case every node polls to checked if the directory node is up and running or not. Until it is up, keep polling so as to wait before proceeding with further logic (registering its shared data segment, etc.). Here, a separate directory server node has to be run too, along with the other participating nodes. Within the dsm_register_datasegment, the SIGSEGV handler is registered and an mprotect is done on the to be shared memory in the data segment with read permissions. After this, the reg_data_seg RPC call is made to the directory. Within the implementation for this RPC at the directory, the directory data structure is updated with the calling node's information i.e. the shared page addresses are stored along with the hostname. Now, if any application tries to write to any location within its shared pages, a write fault will occur which will be handled by the registered SIGSEGV handler. Within the handler, it is checked whether the page fault is due to a write or a read. If it's a write, then a dir_upd RPC call is made to the directory (before the RPC call, the handler also does an mprotect on this page address for single page to give read and write permissions). Within the implementation of this RPC, the directory data structure is updated to indicate that the calling node is now the owner of this page, and then it sends invalidate RPC calls to all nodes which have this shared page with them too. In the implementation for the invalidate RPC, an mprotect is done the corresponding page address for a single page by giving no permissions. After being invalidated, if any node tries to do a read of any location within the invalidated page, it gets a read fault which is again handled by the SIGSEGV handler. In this case, the handler makes a get_latest call to the directory to request the updated version of this page. In the implementation of this RPC, the directory further makes a fetch_latest call to the corresponding owner node of that page to get the updated page (before the RPC call, the handler also does an mprotect on this page address for single page to give read and write permissions, since the updated page will have to be written). Within the implementation for this RPC, the owner copies the entire page in the response and sends it back to the directory. Subsequently upon receiving the page, the directory also copies this page in the response and sends it back to the requester node. Here, the directory does not store the updated version of the page with itself (it actually does not store any actual page within itself at all). As a result, if any other node that was invalidated for the same page does a read, it has to follow the same path as explained above. Back in the current node, upon receiving the response, it copies the page data obtained in the response to the actual page and then in the handler, an mprotect is done again to downgrade this page's permissions to read-only. Also, instead of having page level locking, the directory currently just has a single lock for the entire directory data structure maintaining the dsm data.

Update- Malloc has also been added with the general flow being the same as above apart from some implementation specific changes (the structure maintained at the directory is different since in this case, associated names have to be tracked too).

Note- This dsm implementation has a non-blocking dsm_register_datasegment call in place (register with the directory and return immediately). The directory also sends invalidates to only those nodes that have a particular data segment registered with them, rather than sending invalidates to all the other participating nodes. Both these aspects are according to the actual working of the 3-state invalidate protocol.

Map reduce framework- mr_setup sets the thread id and the number of threads (actually, it is process id and number of processes instead). Then it does start_lock since one distributed mutex is going to be required, and dsm_register_datasegment for the page aligned thread count variable which needs to be shared between all the participating nodes for the barrier implementation. Then it does init_lock. Instead of using the dsm implementation for storing the intermediate data produced by map and to be consumed by reduce using dsm_malloc, a shared intermediate file has been used instead (works since W135 machines by themselves act as a shared memory system). Specific to the word count application, mr_map reads in all the lines from the input file and computes the current node's line offset and number of lines to process within this file. It then calls the word count specific map function passed in as a function pointer. Within the mapper_wc function, for its own chunk of lines within the file, a partial word count operation is performed and key values pairs are stored in the intermediate file by all the participating nodes. mutex_lock and mutex_unlock are wrapped around all file operations for this intermediate file. At the end of this function, barrier is called. Within barrier, the thread count variable is incremented/decremented by one (based upon the passed in direction parameter), followed by a while on this variable being equal to the total number of threads (processes here) or 0. Again, accesses to this variable are surrounded by mutex_lock/unlock. After this is done, the mr_reduce function reads in the intermediate file entirely, along with the output file (if there is already any data present in it), followed by the computation for num of lines and offset within the intermediate file. It then calls the word count specific reduce function, using the passed in function pointer. Within the reducer_wc function, the current node's key value pairs from the intermediate file are looked at, checked if they are already in the output file or not (if they have already been accounted for by some other node which executed reduce before the current node), and if not, do a global word count for these key value pairs across the entire intermediate file's data and write it into the output file. Basically, the intermediate step of shuffle and sort between map and reduce is handled by reduce itself here. Again, mutex_lock/unlock is used for both the intermediate file and the output file. At the end of this function, barrier is called again, followed by the print of the output file by the node having thread id (process id) equal to 0. The main application is straightforward, wherein the thread id and the number of threads (processes) are passed in as command line parameters (similar to what is done in case of the merge sort application), and then init_mr, mr_map and mr_reduce are called sequentially.

K-means on map reduce- The implementation of k-means is very similar to that of the word count application from the map reduce framework point of view. Here, since there are only 3 centroids, there is a need of only 4 threads (processes really). The map phase determines the chunk of input points to process for the current node, and then calculates the closest centroid to each of these points, which is stored into the intermediate file by each node. In the reduce phase, each node reads in the entire intermediate file, and based upon its thread id (process id), picks all points that have one of the centroids closest to them. After this, each node computes the new centroid location for one of the centroids it is responsible for (this can be done since now this node knows all the points having this centroid closest to them). Finally, each node stores its own updated centroid location to the common output file.

2. Difficulties faced and how they were overcome-

Distributed mutex locks- The test application was not working upon addition of the support for multiple locknos in the sense that the output from each node was showing overlapping time durations, since probably multiple nodes were being able to get into the critical section at the same time. In terms of this test application, this was resolved by adding local mutex locks at certain places to avoid out of order RPC call handling which was ideally not expected to happen. As a result of this, the test application now works irrespective of the order of execution between multiple participating nodes.

Distributed shared memory system- Similar to the case above, certain local mutex locks were required to get at least the sequential consistency's base case working (though, it's still unreliable as fails for x out of y executions). Another issue faced was the certain RPC calls were not returning at all, even though at the server side, the function had executed still the return point (debugged using prints). Because of this, specifically for the invalidate RPC call, a gRPC deadline has been set, which eventually causes the RPC to timeout. This is acceptable in case of this specific RPC, since invalidate doesn't return anything in the response. For the merge sort application, the stress test has been made more simpler with just one page of actual data to be merged being shared instead of 16 pages. With this simplification, it still seems to work rather unreliably with just 2 processes. The debug process for this is still ongoing.

Map reduce framework- Instead of using dsm_malloc to store the intermediate state in the dsm implementation, a shared file has been used instead (which is definitely not as per the stated project requirements, but at least the thread count variable required for barrier has been stored in the dsm). Along with this, certain timing issues with respect to RPC call handling (execution and returning responses) are being observed as of now, which results in the overall execution to work unreliably (just gets stuck x out of y times, with success rate being very low). This is most probably an issue with the RPCs in the mutex implementation rather than those in the dsm implementation. As a result, this is still under investigation to find the root cause. Also, only the first input has been tried out yet.

K-means on map reduce- Similar issues as stated above are being observed here too with just the first test case being tried out as of now. With the use of the intermediate file, at times, the file buffering issue (there is a certain delay in writes to files) used to crop up, which was then solved by adding a flush immediately after the writes, not waiting for the file to close for this operation to reflect.

3. Special cases that the code does not work for-

Disclaimer- As mentioned in the README for the git repo, the following details are with respect to my implementation only. My partner Aman's code works better for certain parts and as a result, that implementation should be referred to for these parts. For the rest of the parts, refer to my code (see more details in the README).

Note- It would be advisable to run all the parts with DEBUG mode on, because when it gets stuck midway during execution, partial output to the terminal can still indicate that there is some progress made. View README of the git repo for more details regarding how to execute.

Currently, this is the state of affairs in regards to the executions of all the parts-

Distributed mutex locks- The test application works fine (though, it has just been tested with 3 nodes).

Distributed shared memory system- The sequential consistency application works most of the times, but only in the p3 p2 p1 execution order, not otherwise. Testing with arbitrary sleep times has not been done. With respect to the merge sort application, with single page shared data (instead of 16), and with just two processes, it works unreliably (succeeds x out of y times) i.e. success rate is very low.

Map reduce framework- As stated before, there are certain issues being observed with the expected behavior and order of execution of RPC calls (this is being observed in the merge sort application too) due to which, it works correctly x out of y times for the first test case, but success rate is again very low. The second test case has not been tried.

K-means on map reduce- This has a similar state as above.

P.S. Honest efforts were made to implement all the parts to the best of my knowledge and I am still trying to actively fix all the issues being faced, attempts to resolve these problems are still underway.

4. Distribution of work between the team members-

Both me and my partner (Aman Pandey) have tried to implement all the requirements of the project individually even though we were working as a group. I have the code for all 4 parts in place, whereas he has the code for the first 2 parts completed. Though, my implementation of dsm is working unreliably (both tests, seq consistency and merge sort give successful executions x out of y times, but get stuck for the rest of the executions, with seq consistency working more reliably than merge sort) but his implementation of dsm is working as expected and reliably (for seq consistency, and merge sort to some extent). Since my dsm implementation is unreliable, it cascades down to parts 3 and 4 too, where both word count and k-means applications have a low success rate, working x out of y times (the same issue arises here, where the execution gets stuck somewhere midway and not always at the same point, and there is no progress after that; this could also potentially be an issue in my part 1 implementation when tested aggressively, related to out of order RPC call handling than what the expected order is). Hence, we have tried to integrate my parts 3 and 4 with his dsm and mutex implementation instead, to get it in a reliable working state.