Distributed Shared Memory System

CS 425 MP3

Jon Tedesco – Ben Sommer – Ron Searl

tedesco1 – sommer3 – searl1

Design

In the design of our project, we made several simple assumptions that helped clearly decompose the problem, made sense semantically, and simplified our implementation.

Our Makefile generates two binary programs, *dsm*, and *coordinator*, representing the a node of the distributed shared memory system, and the coordinator node, respectively. For our purposes, we viewed a *coordinator* as simply a special DSM node, one that is designated with the task of ensuring mutual exclusion for our entire system. While this leaves the system vulnerable to coordinator failures, this design is easily extensible to a more robust implementation.

To represent commands, we created a *Command* structure that mimics the *command* design pattern, and as such allows very highly decoupled handling on commands. Our design provides the capability to allow the coordinator to function as a fully DSM node in and of itself, rather than a node entirely dedication to mutual exclusion control.

Initialization

The application starts up as follows:

1 The *dsm* binary is launched

1.1 The *dsm.cc* script parses the configuration file

1.2 Each DSM node is initialized and launched

1.3 Each DSM node begins listening on its incoming connection port, on its main thread

2 The *coordinator* binary is launched

2.1 The *coordinator.cc* reads the configuration file

2.2 The coordinator starts running

2.2.1 The coordinator broadcasts its port number to all DSM nodes

2.2.2 The coordinator begins listening for lock/unlock requests in a separate thread

Mutual Exclusion

Our implementation uses a simple central server mutual exclusion method. Whenever a DSM node discovers that it requires a lock, it sends a *LOCK* command to the coordinator, requesting a global lock. For this assignment, mutual exclusion is treated globally, so that a process requiring a lock on some variable *a* will also lock every other variable *b*. This is both by design and for simplicity, as it is clearly more simple to implement, but with the given set of potential commands, may often turn out to be a more efficient implementation than the more traditional and aggressive approach.

To improve the efficiency of our system, and to implement release consistency in our system, each node caches values locally when it holds the lock, rather than requesting data from other nodes continuously. We know that this can be safely done while a process holds the lock, because it can be confident that it alone has the right to alter data in our system.

Release Consistency

For our implementation of release consistency, we clearly upheld the three primary requirements

for release consistency:

- *The lock must be acquired before reads or writes are executed*

  Each DSM node, if it does not hold the lock, queues up all *ADD*, *PRINT*, and *UNLOCK* commands. Once the lock is obtained, it then traverses this queue of commands and executes them in order.

- *All reads and writes must be completed before release of the lock*

  When a DSM node obtains the lock, it initializes a cache of global data that remains valid while it holds the lock. Thus, when it reads a memory location from some node, it caches the value locally, and does not contact that node again until the cache is invalidated, or essentially, until the lock is released.

  When a memory location is read for the first time, the node sends out a message to get the value and stores it in its cache. This allows us to do writes locally before pushing them all back to memory once the node releases the lock, at which point we clear the local cache. To ensure the writes complete before another node can receive the lock, we make sure the nodes contacted for writing have closed the communication socket, signaling their writes have completed, before actually releasing the lock.

- *All processes must release the lock before another can acquire it*

  The coordinator ensures that this is always the case, by maintaining a global lock for the system. The coordinator will queue up all lock requests if the lock is currently held and will not grant the lock to any other node until the node holding the lock has sent an *UNLOCK* command to the coordinator.

Coordinator

The coordinator is initialized in much the same way as a DSM node, except that the coordinator also contains a queue in which it keeps track of requests to enter the critical section from various DSM nodes. As a direct result, and as an intentional consequence of our design, the coordinator thus contains the basic connection data structures as does the DSM node. This allows for significant code reuse, and thus simplifies our implementation substantially.

The coordinator is robustly implemented, and provides substantial error handling from when a command is parsed. When an invalid command is read, a detailed error message is output to the console, and the coordinator continues to parse the input, attempting to read the remaining valid portions of the input.

The coordinator is also the only multi-threaded component of the project. It uses a separate thread to listen on its incoming connection port, rather than its main thread. This allows for asynchronous processing of mutual exclusion requests and parsing of the command file, so that the coordinator may issue commands and handle lock/unlock requests simultaneously.

DSM

The DSM node is single threaded, and indefinitely waits, listening for incoming connections to its command port. When it discovers that it requires a lock for some command, it creates a queue of commands, and adds every command that must be synchronized to this queue as it arrives. When it finally receives a *LOCK_ACQUIRED* command from the coordinator, it executes each command in its queue.

However, to ensure that we do create a deadlock, the DSM node will only queue commands it receives from the coordinator. Subsequent commands generated by other DSM nodes are not blocked, as the sending node is assumed to have an exclusive lock on the system. If we did not

make this assumption, commands that create subsequent commands, such as the *ADD* command, would fail to terminate.

To improve the efficiency of our system, and to implement release consistency in our system, each node caches values locally when it holds the lock, rather than requesting data from other nodes continuously. We know that this can be safely done while a process holds the lock, because it can be confident that it alone has the right to alter data in our system.

Experiment 1

For this first experiment, we used the following command file, shown on the left, and produced the following output. Note that the individual lines do not correspond, as execution order varied from input commands based on mutual exclusion.

Input ***command1.txt***	**Output**
1000:0:1	Successfully launched 10 DSM nodes
1000:0:3:10:10:100	Node 0 requested the lock
2000:0:3:20:20:10:10	Node 0 acquired the lock
2000:1:1	Node 0 handled ADD command
1000:1:3:10:10:13	Node 0 handled ADD command
1000:0:4:20	Node 1 requested the lock
2000:0:3:10:20:100	The value 110 is stored in memory address 20 at node 0
2000:1:3:20:20:10:1	Node 0 handled ADD command
1000:1:4:20	The value 210 is stored in memory address 10 at node 0
3000:0:4:10	Node 0 handled ADD command
1000:0:3:30:30:10	The value 10 is stored in memory address 30 at node 0
3000:1:3:30:20:30:0	Node 0 handled ADD command
1000:1:4:30	The value 120 is stored in memory address 10 at node 0
1000:0:4:30	Node 0 released the lock
1000:0:3:10:20:30:0	Node 1 acquired the lock
1000:0:4:10	Node 1 handled ADD command
1000:0:2	Node 1 handled ADD command
1000:1:3:10:30:10:11	The value 244 is stored in memory address 20 at node 1
1000:1:4:10	Node 1 handled ADD command
	The value 254 is stored in memory address 30 at node 1
	Node 1 handled ADD command
	The value 142 is stored in memory address 10 at node 1

Total Number of Messages: 40

We can see that in this run, the number of messages used was relatively low. This makes sense, because the number of lock and unlock requests is relatively low compared to the other file. When a lock is acquired, many operations are done with this same lock, which yields the ideal performance for our implementation.

Experiment 2

For this second experiment, we used the following command file, shown on the left, and produced the following output, shown on the right. Note that the individual lines do not correspond, as execution order varied from input commands based on mutual exclusion.

Input ***command2.txt***	**Output**
1000:0:1 1000:0:3:10:10:100 2000:0:3:20:20:10:10 2000:1:1 1000:0:2 1000:1:3:10:10:13 1000:0:1 2000:1:3:20:20:10:1 2000:1:2 1000:0:4:20 2000:0:3:10:20:100 1000:1:1 1000:1:4:20 3000:1:3:30:20:30:0 1000:0:2 2000:1:2 1000:0:1 3000:0:4:10 1000:0:3:30:30:10 3000:1:1 1000:0:4:30 1000:1:4:20 3000:1:3:30:20:30:0 2000:1:2 1000:0:3:10:20:30:0 1000:0:2 1000:0:1 1000:0:4:10 1000:0:2 2000:1:1 1000:1:4:30 1000:1:3:10:30:10:11 1000:1:4:10 1000:1:2	Successfully launched 10 DSM nodes Node 0 requested the lock Node 0 acquired the lock Node 0 handled ADD command Node 0 handled ADD command Node 1 requested the lock Node 0 released the lock Node 1 acquired the lock Node 1 handled ADD command Node 0 requested the lock Node 1 handled ADD command Node 1 released the lock Node 0 acquired the lock The value 224 is stored in memory address 20 at node 0 Node 0 handled ADD command Node 1 requested the lock Node 0 released the lock Node 1 acquired the lock The value 224 is stored in memory address 20 at node 1 Node 1 handled ADD command Node 1 released the lock Node 0 requested the lock Node 0 acquired the lock The value 68 is stored in memory address 10 at node 0 Node 0 handled ADD command Node 1 requested the lock The value 234 is stored in memory address 30 at node 0 Node 0 handled ADD command Node 0 released the lock Node 1 acquired the lock The value 224 is stored in memory address 20 at node 1 Node 1 handled ADD command Node 1 released the lock Node 0 requested the lock Node 0 acquired the lock The value 202 is stored in memory address 10 at node 0 Node 0 released the lock Node 1 requested the lock Node 1 acquired the lock The value 202 is stored in memory address 30 at node 1 Node 1 handled ADD command The value 159 is stored in memory address 10 at node 1 Node 1 released the lock

Total Number of Messages: 93

We can see that in this run, the number of messages used was relatively high. This also makes sense, because the number of lock and unlock requests is relatively high compared to the other file. When a lock is acquired, very few operations are done with this same lock. This means that the overhead for locking and unlocking dominates the cost of concurrent operations here. This is not the ideal use case for our implementation, so performance somewhat suffers.

Division of Labor

Ron

Design, DSM functions *LOCK*, *UNLOCK*, and *WRITE*, writing back caches, queuing commands, testing, debugging

Ben

Design, DSM functions *ADD*, *PRINT*, and *READ*, create caches, update caches, testing, debugging

Jon

Design, project setup, project writeup, all coordinator functionality, main startup scripts (coordinator.cc, dsm.cc), initial decomposition