Joshua Lew, Larry Liu
NetIDs: jtlew2, lliu65
ECE 428 Spring 2017
May 1, 2017

MP3 - Distributed Transactions: Design Documentation

System Design Considerations:

Handling Concurrency and Isolation
We handle concurrency and isolation by using two phase locking and temporary storage for uncommitted data. Two phase locking allows us to safely process multiple transactions at once. We use separate locks for each object, which allows transactions with no overlapping object operations to run concurrently. Since the locks for a transaction are held until `COMMIT`/`ABORT`, this stops multiple transactions from editing the same object at once.

Using modified Read-Write Lock
Python does not have a built-in read-write lock, so we created our own version. Our class uses exclusively non-blocking methods, because we want the acquiring of any lock to be aborted if a deadlock is detected. Our class also keeps track of the current owner/owners of a lock, which is used to notify the coordinator which client our transaction is waiting on. To maintain the two-phase locking protocol, we needed a method to "promote" a read lock to a write lock. Thus, we created a non-blocking method that tries to promote a read lock to a write lock by checking if the user is the only owner of the read-lock.

Deadlock Design
For our deadlock detection system, each server sends a message to the coordinator whenever they fail to acquire a lock, indicating which client they are waiting on. After acquiring the lock, the server will send another message indicating that they are not waiting on any client (see `op` 5). The coordinator compares all these "waiting-on" relationships and tells every server to abort the transaction of the highest VM# client involved in the deadlock. Since there are only 3 clients/transactions up at a time, there are only five cases to check. The servers will abort the chosen transaction, and that client will be notified of the abort by the coordinator.

Processing Client Commands and Client Interface:
When processing messages, we utilized basic Python string parsing to examine user-typed commands. A client is allowed to type another command after receiving the acknowledgement that the previous had been processed and completed. General descriptions for how we handled each of the required functions are provided below:

`SET`:
First, we check if the key exists in the server's storage of (key, value) pairs. If it is, then we perform another check to see if it owns the lock for the input key. If the lock is owned, and is currently a read lock, then we will promote it to ownership of a write lock. If it does not own the key, we will continue to wait until we acquire the write lock. After owning the write lock, the process is then able to safely set the (key, value) pair in a local dictionary of uncommitted keys. After doing so, it will finally send the acknowledgement to the coordinator that key has been set. These are not transferred to the main storage data structure until the client calls `COMMIT` (discussed later).

`GET`:
For our get functionality, we first check if the input key currently exists in our local dictionary of uncommitted keys. If it exists, we wait until we acquire the read lock for the key (a blocking operation) and then obtain the value of the key that exists in the uncommitted keys structure. This must be the first structure to check because it contains the most updated value of a key. The next structure we check if the key does not exist in the uncommitted keys dictionary is storage. If the key does not exist, we will return a "*NOT FOUND*" message to the coordinator (which will eventually force the server to `ABORT`). On the other hand, if the key does exist in storage, we wait until acquiring the read lock, then return the value associated with the key to the coordinator.

`COMMIT`:
When receiving the commit message, it tells our server that all operations for the transactions should be written to disk (in this case, simply the storage data structure we've been referring to). This will make the results of the transaction visible to other transactions. We will iterate through all (key, value) pairs in the uncommitted keys structure, and set them in the storage structure. Then, for all locks that the process owns, we will release all of them so that they are free to use for other transactions. Finally, we clear all necessary structures and return the "*COMMIT OK*" message to the coordinator.

`ABORT`:
Our abort functionality works very similar to commit; however, we skip writing all (key, value) pairs from the uncommitted keys structure to the storage structure. All other steps are the same except we return an "*ABORT*" message to the coordinator.

Message Protocol:
The messaging protocol we used for passing messages is as follows:

```
op`server_number`key`value`client_id`transaction_id~
```

Client to Coordinator and *Coordinator to Server:*
`op`: 0 - `BEGIN` transaction request
1 - `SET` key-value request
2 - `GET` key request
3 - `COMMIT` transaction request
4 - `ABORT` transaction request

Server to Coordinator and *Coordinator to Client:*
`op`: 1 - `SET` key-value return value
2 - `GET` key return value
3 - `COMMIT` transaction return value
4 - `ABORT` transaction return value
5 - `LOCK_ACQUIRE`/`LOCK_RELEASE` ping

Note: For `op` 5, this message is only sent from server to coordinator. It indicates whether the server is currently waiting on a lock or has acquired a lock; thus, not waiting on anybody. For the `transaction_id`, if it is high, then the `ABORT` message sent from coordinator to server indicates that a deadlock has occurred and should release all locks for the included `client_id`. If it is low, then no deadlock exists. We utilize these aspects of our protocol to ensure that a deadlock is properly handled by the servers, and is appropriately communicated to the client.

Performance Challenges and Metrics:
When considering performance challenges, there were not that many that existed in this MP because we did not have to handle fault tolerance. Thus, our system was static in that there always existed 3 clients, 1 coordinator, and 5 servers. If we were to proceed with the not handling fault tolerance, our system is very scalable. It is very simple to extend our system to support more servers and client machines, which would require 1) modifying our acknowledgement count in our coordinator code, 2) mapping the representations of servers by letter ('A', 'B', etc…) to server locations, and 3) rewriting more cases for deadlock detection. Evidently, these are simple modifications for scaling our system. All `SET` and `GET` operations in transactions are implemented very efficiently, none of which block unless operations require waiting on a lock. Additionally, one of the more technical challenges was to handle deadlock detection in a clean manner. This involved pinging coordinators whenever servers attempt to wait on a lock or acquire a lock. Also, it required that a deadlock detection should be eventually communicated to the clients. We executed this by first communicating the deadlock to a server with an `ABORT` message, which would trigger proper removal of structures holding previous operations in the transaction. This would finally route a reply to the client indicating that the transaction was aborted. We isolated most of the deadlock detection logic in the coordinator

because it has view of all messages passed between clients to servers and servers to clients. One improvement for our system to increase performance would be to reduce the time the coordinator process waits for fully declaring deadlock detection. In our implementation, the coordinator program iterates through all potential deadlock scenarios, and if any of them are true, it would sleep for 1 second and check again before declaring a deadlock. This 1 second of period is to ensure that if a process has acquired the lock is able to send a message to the coordinator to indicate it is no longer waiting; thus, removing any chance of false positives. However, it's clear that if the time the coordinator process waits for is not the most precisely calculated time. If this time is reduced, then our time for detecting deadlock is reduced; and ultimately the results of fixing a deadlock would be communicated to the client faster.

Resources Utilized:
Provided below is a list of major resources we referenced to throughout the course of development of MP3. Major new concepts learned that were not introduced in lecture or previous classes include: Python programming with more complex data structures, TCP protocol, classes in Python, and locks in Python.

General Python Documentation:
https://docs.python.org/2/howto/sockets.html

Python Threading:
https://docs.python.org/2/library/threading.html

Python Thread Locking:
http://stackoverflow.com/questions/10525185/python-threading-how-do-i-lock-a-thread

Python Classes:
https://jeffknupp.com/blog/2014/06/18/improve-your-python-python-classes-and-object-oriented-programming/

ECE 428 Lecture 16 Slides on Transactions