CSEP 545 Project Design Document

Team members:
Mihnea Olteanu (mihneao@gmail.com)
Vincent T.C. Lung (Vincent.lung@gmail.com)

Summary

We have implemented all the required features in the project plus the following extra-credit features:

- Garbage collecting the outstanding transaction list in the TM
- Paged files
- Presumed Abort
- Weaker degree of isolation

Please refer to the last chapter for details on the extra-credit features.

Data structures

RM Durable Storage Data Structures

The durable storage in the RM is implemented using the various Storage* classes in the MyRM.csproj. The following section has a breakdown and description of the various classes.

Reservation.cs

This class implements an abstraction that represents an itinerary reservation made by a customer. The ID of this structure is a Customer object and the class stores a list of all resource IDs that are part of the reservation.
Introducing this abstraction allows us to use a single strategy/protocol to store both resources and customer reservation information in our persistent store.

StorageManager.cs

This class implements the transaction aware persistent storage mechanism. It manages reads and write from the physical disk file, implements the locking policy, manages and stores the contextual data for each transaction. The StorageManager relies on the following main components to implement its functionality:

- **Single Disk File:** a single file is used to store both the user data as well as book-keeping data needed by the manager

- **DB Root:** each data file will have a DB root instance stored on page 0 of the data file containing the addresses of the page table, resource indexes, free page list, and prepared transactions state table
- **Page Table:** each data file has one instance of a page table used to translate logical record addresses into physical addresses corresponding to the location of the record in the actual data file
- **Resource Index:** each data file has one instance of the resource index used to store a mapping of resource id (RID) to logical page address
- **Reservation Index**:each data file has one instance of the reservation index used to store a mapping of reservation id (Client) to logical page address
- **Free Page List:** each data file has one instance of a free page list used to recycle pages that have been freed up by the shadowing algorithm's moving of data to new physical location
- **Prepared Transactions State Table**: each data file has a one instance of the table storing the address of the location in persistent storage needed to commit active transactions.

StorageContext.cs

The class holds all the information required during the life-time of a transaction. It stores a pointer to the copies PageTable, ResourceIndex, and ReservationIndex the transaction is using, as well as lists of all StoragePages the transaction has allocated and freed.

StoragePage.cs

The class implements an abstraction of a 4kB storage page. Each storage page is implemented as a list of variable size records that this class is able to serialize and de-serialize.The serialized format looks as follows:

```
|RecordCount:Int32|Record "1" Size:Int32|Record "1" Data:Binary| …
|Record "n" Size:Int32|Record "n" Data:Binary|
```

New records can be added to the page and existing records can be modified. If the page runs out of space while writing a new record or trying to extend a record it throws and exception.

StoragePageManager.cs

Data: List of int values representing physical page indices.
Description: This class implements a helper object to keep track of "free" pages in the data file. "Free" pages are pages that do not contain data which are no longer holding the "master" copy of the data. This is used in order to make more efficient use of the data storage.
The class implements logic to serialize and de-serialize its data into records stored in StoragePage obejcts. This data is read from persistent storage only during start-up.

StoragePageTable.cs

Data: List of PageTableItem (MyRM\DS\PageTableItem.cs)
Description: This class implements a data-structure that maps a logical page number to a physical page number in the file stored on disk. Logical pages are assigned in increasing order starting at “0” when new data is written to the store.
The class implements the logic to serialize and de-serialize its data into records stored in StoragePage objects. In order to serialize and deserialize its data a StoragePageTable object only requires the index of the first page containing its data. When reading data from persistent storage into memory the StoragePageTable will merge the data with the contents of the data it already has in memory using “dirty” flags to determine which data already in memory has been modified and should not be overwritten. Upon serialization to durable storage the class provides mechanism to clear the “dirty” flags.

StoragePageIndex.cs

Data: List of IndexItem<T> objects (MyRM\DS\IndexItem.cs)
Description: The class implements a data-structure that maps a Resource or Reservation ID to a logical address (logical page & record). New mappings are added to this index as new Resources or Reservations are written to the store.
Similar to the StoragePageTable the class implements logic to serialize and deserialize its data into records stored in StoragePage objects. Similarly to the StoragePageTable the index data structure only requires the index of the first page containing its data. Once again similar to the StoragePageTable this class as well when reading data from the durable storage will merge it with the in memory data using “dirty” flags to indicate which values have changed.

StorageTransactionTable.cs

Data: List of TransItem objects (MyRM\DS\TransItem.cs)
Description: The class implements a data-structure that keeps track of the location in persistent storage (disk file) of the data needed to commit an active transaction. Data is written to persistent storage and a pointer stored in this table upon receipt of a “PREPARE-TO-COMMIT” message from the transaction manager.
Similarly to the StoragePageTable and StorageIndex this class knows how to serialize and de-serialize its data into records stored in StoragePage obejcts. Items in this table are never modified after the initial add, they are removed when the transaction commits. Unlike the StoragePageTable and StorageIndex table data read from persistent storage will replace any data already in main memory.

TM Durable Storage Data Structures

The only durable storage in the TM is the outstanding transaction log for keeping track of transactions that have not received Done from all its associated RMs.

MyTM.cs

Data: OutstandingTransactionValue class (MyTM\myTM.cs)
Description: The class implements a data-structure that describes the property of a outstanding transaction. It contains a transaction ID, a transaction type (commit or abort) and a list of strings representing the list of RMs associated with a transaction.

Data: OutstandingTransaction class (MyTM\myTM.cs)
Description: The class implements a data-structure that keep track of outstanding transactions for the TM. It uses a hashtable that maps a transaction ID to a OutstandingTransactionValue object. In addition, it provides functions for writing the hashtable to a file and populating the hashtable with data from a file.

Committing a transaction

TM Side

When the TM receives a Commit request for a given transaction, it first obtains the list of RMs associated with the transaction (let's call this the RM list for brevity). Then the TM will perform the following operations (in order):

1. Send a Request For Prepare message for each RM in the RM list. If any RM times out or replies with No, the TM will abort the transaction.
2. The TM will either commit or abort the transaction at this point based on the responses from the RMs. The TM writes the transaction id and the RM list to the outstanding transaction log. Outstanding transactions are transactions that have not received acknowledgement from all its associated RMs.
3. The TM sends commit or abort to all RMs in the RM list. If a RM replies with Done, the TM removes that particular RM from the RM list. Otherwise, the TM will wait until the operation times out.
4. If it is a commit operation and the RM list is not empty (ie. some RMs have timed out), the TM writes the transaction id and the RM list to the outstanding transaction log. The recovery operation (see next section) will re-issue commit to the remaining RMs that have not acknowledged the transaction. On the other hand, if it is an abort operation, the TM always writes the transaction id and an empty RM list to the outstanding transaction log to indicate that it can forget about the transaction. This is essentially how we implemented Presumed Abort.

RM Side

(The RM requires that “Prepare()” is called first before commit. If “Commit()” called before “Prepare()” was called then it will just reply done and ignore the request. It is easy to change this to do 1 phase commit.)
The RM will perform the 2-PC protocol as follow:

1. When it receives a “Prepare()” message it will write out the transaction context to disk inside the locked region. This includes the following:
    a. remove transaction from the “active transaction” list
    b. write the modified StoragePageTable to disk
    c. write the modified StorageIndex(es) to disk (both the resource and reservation index)
    d. write the allocated pages list to disk
    e. write the freed pages list to disk
    f. add transaction to the “prepared transactions” list
    g. write the global “prepared transaction” list to disk
    h. update the DB root to point to the new “prepared transaction” list
2. When it it receives a “Commit()” message it will install the transactions updates and forget the transaction by removing it from the “prepared transaction” list. The steps performed inside a locked section are as follows:

. remove the transaction from the “pre-pared” transaction list
    a. read in the transaction context data if needed
    b. merge the StoragePageTable
    c. write the StoragePageTable to disk
    d. merge the StorageIndex(es) (both the resource and reservation index)
    e. write the StorageIndex(es) to disk (both the resources and reservations)
    f. write the global “prepared transactions” list to disk
    g. update the “free” page list
    h. write the global “free” page list to disk
    i. update the DB root to point to the new page table, indexes, “free” page list, and “prepared transactions” list

The “prepared transaction” list and “free” page list are global structures in the StorageManager that are only modified by transactions during the “commit” stages in a synchronized code regions. As such they are always up to date and do not need to be merged.

Recovery

TM Side

The TM keeps a list of outstanding transaction in a file (see the Commit section above for more information). Outstanding transactions are transactions that have not received acknowledgement from all its associated RMs.

After the TM is up and running, it runs a recovery task every 30 seconds. When the task runs, it reads the outstanding transaction file and locks it for exclusive access. Then, it goes through each transaction in the log and resend commit or abort commands to the remaining unacknowledged RMs in the RM list. If the TM receives a Done message from the RM, it removes that RM from the RM list. Once the TM finished going through the log, it rewrites the log with the remaining outstanding transactions and their RMs. Since we have implemented Presumed Abort in the commit operation, there should be no abort transactions in the outstanding transactions log. The only way this will happen is that the TM crashes during Abort, specifically between the initial write to the outstanding transaction log with the transaction id and its RM list and before the write to the log with only the transaction id to indicate that the Abort was sent to the RMs.

RM Side

During recovery the RM only needs to finalize the all transactions in the "prepared transaction" list. It does this by calling the TM for each transaction and enquiring about the state.
Because we do not store the locks the transaction held when we persisted the transaction state we have to finalize all these transactions before we start accepting new requests. This means that if the TM is not available the RM will be completely blocked until the TM comes back up. Soring the locks would have allowed us to start accepting requests while waiting for the TM but we felt that without a running TM the RM would not do any useful work because no transactions are able to commit and the RM would probably also end up in a state with very high lock contentions were transactions continuously deadlock and are aborted.

Extra-credit

The extra credit items we have implemented are listed below with a brief description of how each one was implemented.

ii. Garbage collect the TM's committed transaction list.

In order to garbage collect the TM's commit list the TM will periodically resend "commit" decisions to those RMs that it has not received a "done" message from. In order to handle this on the RM side we have implemented the following protocol. Once the RM has successfully processed a "commit" or "abort" message from the TM it "forgets" the transaction by ensuring that is not present in either to "active transactions" list nor the "prepared transactions" list. If it receives a "commit" or "abort" message for a transaction it knows nothing about it replies with "done".
Once the TM has collected "done" messages from all RMs that were involved in the transaction it can delete the transaction from the outstanding transaction log since no RM will ask about it in the future.

iii. Presumed abort for two-phase commit

In order to reduce the book-keeping needed for 2 phase commit we have implemented presumed abort. The TM forgets about the transaction as soon as it makes the Abort decision. If an RM asks the TM about a transaction the TM knows nothing about the TM will respond with "Abort".

vi. Implement paged files

As described in the “Data Structures” section we have implemented a StoragePage abstraction and all data written to our persistent storage file is written in chunks of 4kB. We keep track of physical page addresses that are “free” by the data relocation steps performed by the shadowing algorithm and reuse them for storage needs of transactions that execute after the transaction that “freed” those pages has committed.
The DBRoot is stored in page “0” and is written as a single chunk of 4kB. If 4kB is not the size that is atomically flushed to disk by the OS we have the ability to easily change the page size. (For additional details on how reading and writing pages like the format please see the section on StoragePage.cs above).

vii. Implement weaker degrees of isolation

For the to commands that scan the entire data store for items: ListCustomers(...), ListRooms(...), ListCars(...), ListFlights(...) we have implements the “read committed” locking protocol. Basically those commands acquire the appropriate read locks (read lock on the page and read lock on the ID) for each individual item but release them as soon as they have read in the data for the item.
All these operation are implemented as a scan over the StorageIndex (see above for a description of how the StorageIndex is implemented) for either the resources or reservation. The index is read in from disk at the beginning of the operation, so the values that will be returned will be the ones that were written by transactions that already committed. Items added by transactions running concurrently with this transaction will not be seen by the scan, modifications to existing items, however, can be picked up if the concurrent transaction commits while the scan was happening.
We lose serializability of these “scan” operations but the trade-off is worth it given the alternative of locking pretty much our entire store for the duration of the scan.

Code modules

You can find the source code of our project under the various folders in our submitted zip file. Here is the directory structure of our project.

- CSEP545/ - Main executable for the project - contains the demo code
- MyRM/ - Contains the lock manager, resource manager and storage manager.
    - MyRM/DS/ - Contains helper classes needed to serialize data to file
- MyRMTests/ - Contains tests we created for classes in the MyRM folder.
- MyTM/ - Transaction manager
- MyTMTests/ - Contains tests we created for classes in the MyTM folder
- MyWC/ - Workflow coordinator
- TP/ - Transaction Processor interface

Other than the test code modules no new modules were added compared to the modules in the initial sample code.