Lab 3: Transactional Key-Value Database

Hand out: Nov 23th

Deadline: Dec 7th 23:59 (GMT+8)

We suggest you reading the whole content of this document before writing any code.

Get Ready

In this lab, you will implement a simple key-value database. It supports **get**, **put**, **del**, and the most important feature, **transaction**.

In Part1, you will implement the basic functionalities: get, put, del. Then, you will add transaction support, using **Two Phase Locking** (**2PL**, in **Part2**) or **Optimistic Concurrency Control** (**OCC**, in **Part3**) policy.

This lab is based on L**ab2 Part2**, so make sure you can pass the old tests.

If you have any questions, please ask TA: Huang Zheng (huangzheng@sjtu.edu.cn).

Getting Started

At first, save your solution of lab1++:

```
% cd lab-cse
% git commit -a -m "solution for lab1-plus"
```

Then, pull the lab3 branch:

```
% git pull
% git checkout lab3
```

Merge with your lab2:

```
% git merge lab2
```

If you don't want to be limited by the given Docker, and want to run this lab on your native environment, for Debian/Ubuntu, you just need to install the libfuse-dev package:

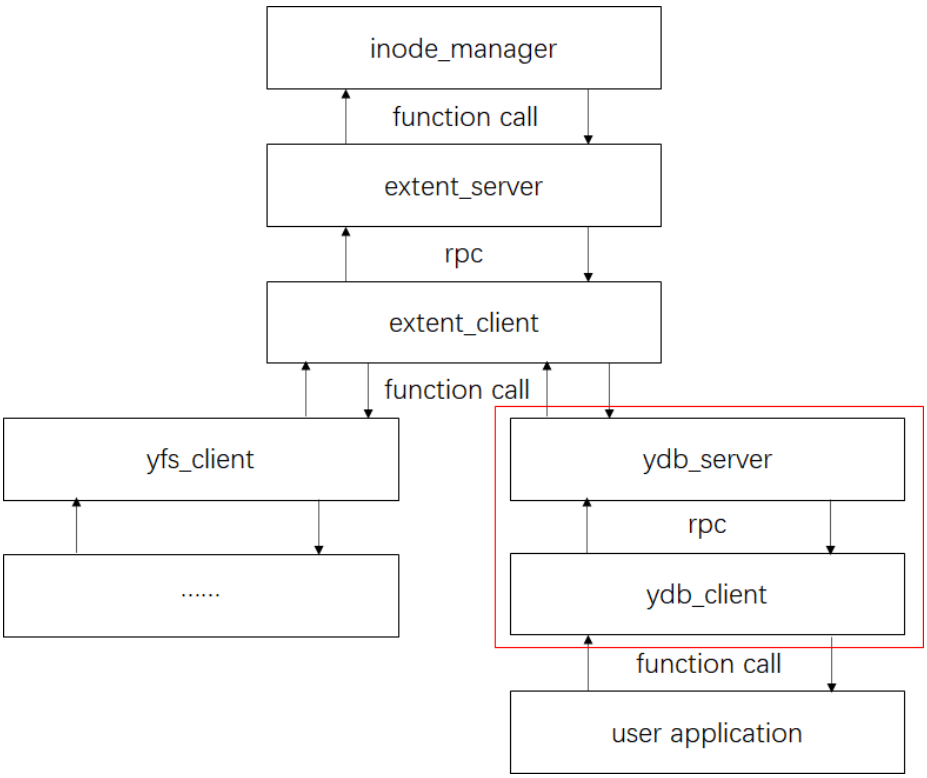
```
% sudo apt-get install libfuse-dev
```

Then:

```
% make clean
% make
```

Everything will OK now.

Part 1: Simple Database without Transaction



Recall the rpc interface between extent_server and extent_client, you may find it is already a key-value database prototype: the extentid is the key, and the inode content is the value.

A typical key-value database should support any byte sequence as key or value. One direct idea is using a hash function to map the byte sequence into an integer, which can be considered as extentid to get/put the inode content from extent_server.

See **ydb_server.cc**. To simplify your work, the length of key or value is no bigger than 32 during **all** tests, and you don't need to thought the inode size overflow problem because the test data is not to much.

(A better solution maybe enlarge the max inode size support of inode_manager, and use a dynamic allocator (like malloc, which you have implemented in ICS course lab) over the inode content to handle a very long key/value as well as saving storage space. This is not forced in this lab)

Also, keep compatible with the existing yfs system is necessary, so you should **not** add or change the protocol between extent_server and extent_client.

The tests never run ydb_server and yfs_client at same time, so we just create all inodes during ydb_server initialization, and you can invoke extent_server with only get and put in the later.

Part 2: Two Phase Locking

See **ydb_client.h**, we define six interfaces of ydb_client:

```
void transaction_begin();
void transaction_commit();
void transaction_abort();
std::string get(const std::string &key);
void set(const std::string &key, const std::string &value);
void del(const std::string &value);
```

The get/set/del operation must appear between transaction_begin and transaction_commit/transaction_abort.

There four basic characteristics of transaction are atomicity, consistency, isolation and durability, and also, we want all transactions can be seen as a serializable schedule. That means, 1) if one transaction repeats to **get** same key and itself does not **set** this key between these **get** operations, it should always see the same value, 2) a transaction should not **get** uncommitted **set** values from other transactions (but it can **get** uncommitted values that itself has **set**ted), 3) after the transaction committed, **all** the values it has setted should store into the backend extent_server, and can be seen by other transactions, 3) if the transaction abortted, **all** the values it has setted should not be seen by other transactions. See **test-lab3-part2-3-basic.cc** for some examples.

Part 2A: Simple Design

Two-phase locking: (See lec-09.pptx)

- Each shared variable has a lock (fine-grained locking)
- Before any operation on the variable, the transaction must acquire the corresponding lock
- After transaction releases a lock, it cannot acquire any other locks

See **ydb_server_2pl.h** and **ydb_server_2pl.cc**.

We only run **one** ydb_server together with **one** extent_server, so it is single machine transaction. But you should also use lock_server+lock_client as your lock implementation and do **NOT** use **ANY** local locks like pthread_* functions directly (to decouple with the implementation of lock) (We won't set RPC_LOSSY, but if you like, you can still use lock_server_cache+lock_client_cache). We will manually check your code.

You should not change the interface of ydb_client, because the tests will use them.

Part 2B: Deadlock Detecting

	T1	T2
1	lock(a)	lock(b)
2	lock(b)	
3		lock(a)

Assume there are two concurrency transactions T1 and T2 executing as the above table shows, T2's lock(a) at time3 will cause a deadlock.

Since we cannot predict future operations, we cannot imply a global order on locks to prevent deadlock. So if a lock attempt will cause deadlock, we should abort current transaction immediately.

To keep compatible with the existing yfs system, you should **not add** or **change** the lock protocol. We will use the same lock implementation to re-test your yfs system.

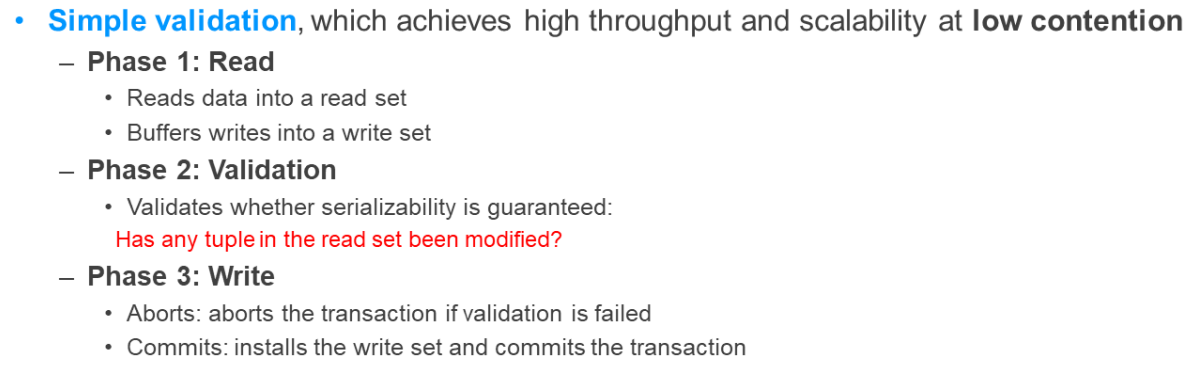
(NOTE: If an aborting occured during **get**/**put**/**del**/**transaction_commit**, the ydb server should return ydb_protocol::ABORT and then ydb client will throw such exception. But if the client calls **transaction_abort** actively, ydb server should just return ydb_protocol::OK after doing aborting successfully)

Part 3: Optimistic Concurrency Control

Optimistic Concurrency Control: (See lec-09.pptx)

- Concurrent local processing
- Validation in critical section
- Commit the results in critical section or abort

You can just use the **simple validation**, the tests allow false aborts.



See **ydb_server_occ.h** and **ydb_server_occ.cc**.

Test

To start ydb server without transaction support (part1, this will use class ydb_server):

```
% ./start_ydb.sh NONE
```

To start ydb server with 2PL transaction support (part2, this will use class ydb_server_2pl):

```
% ./start_ydb.sh 2PL
```

To start ydb server with OCC transaction support (part3, this will use class ydb_server_occ):

```
% ./start_ydb.sh OCC
```

To stop ydb server:

```
% ./stop_ydb.sh
```

Then, you can run each test separately:

- Durability test for part 1, part 2, part 3:

  ```
  % ./test-lab3-durability.sh <NONE/2PL/OCC>
  ```

- Basic functional test and exception test for part2 and part3: (the default ydb_server_listen_port is 4772, see start_ydb.sh and ydb_smain.cc):

  ```
  % ./test-lab3-part2-3-basic <ydb_server_listen_port>
  ```

- Special test a for 2PL, without deadlock (part 2A):

  ```
  % ./test-lab3-part2-a <ydb_server_listen_port>
  ```

- Special test b for 2PL, with deadlock (part 2B):

  ```
  % ./test-lab3-part2-b <ydb_server_listen_port>
  ```

- Special test a for OCC, without transaction conflict (part 3):

  ```
  % ./test-lab3-part3-a <ydb_server_listen_port>
  ```

- Special test b for OCC, with transaction conflict (part 3):

  ```
  % ./test-lab3-part3-b <ydb_server_listen_port>
  ```

- A complex test for part 2 and part 3:

  ```
  % ./test-lab3-part2-3-complex <ydb_server_listen_port>
  ```

The grade_lab3.sh script will run all the tests, and before starting each test, it will restart ydb server, so the result should equal to running each test separately.

When you pass all lab3 tests, re-test lab2 part by executing grade_lab2.sh. Make sure your implementation can pass both lab2 and lab3 tests.

Extra (voluntary, whether do it depends on you) :

- Replace rpc/librpc64.a with lab2's one and re-run the tests, can you pass again? Write down the reason.

  (NOTE: If you build with lab2's librpc64.a outside the docker and the g++ version is higher than 5, you should add `-no-pie -D_GLIBCXX_USE_CXX11_ABI=0` to CXXFLAGS and `-no-pie` to LDFLAGS to avoid build error，see GNUMakefile line 8, 9, 24, 25)
- Do further tests by writing more test cases.
- Analyze the performance and try to optimize it.

Handin Procedure

Write down your **student id**, your **name**, your **design** of all parts and **code explaination** in **lab3-doc_[your student id].[txt/md/pdf/docx/doc/...]**, we will manually check your doc and code.

After all above done:

```
% make handin
```

That should produce a file called lab3.tgz in the directory. Change the file name to your student id:

```
% mv lab3.tgz lab3_[your student id].tgz
```

Then upload **lab3_[your student id].tgz** file to **ftp://huangzheng:public@public.sjtu.edu.cn/upload/cse2020/lab3** (username: huangzheng, password: public) before the deadline. You are only given the permission to list and create new file, but no overwrite and read. So make sure your implementation has passed all the tests before final submit. (If you must re-submit a new version, add explicit version number such as "V2" to indicate).