POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Transactional key-value store

MSc in Computer Science and Engineering

Author: Vincenzo Greco

Academic year: 2021-2022

---

1. Introduction

This is an implementation of a transactional key-value store in a distributed system.
This implementation will follow the requirement and the domain assumption given by Prof. G. Cugola and A. Margara

2. Assignment

2.1. Requirements

Clients submit transactions. Each transaction is a list of read and write operations:

- write(k, v): inserts/updates value v for key k.
- read(k): returns the value associated to key k (or null if the key is not present)

The store is internally partitioned and replicated: assuming N nodes, each key (with its values) is replicated in exactly R nodes, with R<=N being a configuration parameter.
Clients can interact with the store by contacting one or more nodes and submitting their transactions, one by one. Nodes internally coordinate to offer the service.
The store should offer sequential replication consistency and serializable transactional isolation. Your design should favour solutions that maximize system performance, in terms of query latency and throughput.

2.2. Assumptions

You can assume reliable processes and reliable links (use TCP to approximate reliable links and assume no network partitions happens).

3. Analysis

Since the system will be intrinsically replicated this could give us performance improvements in terms of throughput, latency, availability and fault tolerance. The main drawback is that we have to deal with consistency

3.1. Consistency

What would happen if multiple replicas are updated concurrently?
Since no data are own by an user and we want to guarantee serializable transactional isolation we will use a "Data centric" consistency model that guarantee sequential consistency:

> The result is the same as if the operations by all processes were executed in some sequential order, and the operations by each process appear in this sequence in the order specified by its program

In order to achieve sequential consistency replicas need to agree on a given order of operations. This can be done through a single coordinator or a distributed agreement. The important assumption behind it is to have "Sticky" clients.
Single coordinator introduce high latency so leaderless protocols have to be used to achieve the requirements.

3.2. Concurrency

Since in a hypothetical data store there could be a huge amount of transaction, execute them serially would mean a huge waste of time: concurrency is required.
Concurrent transactions can cause anomalies. The possible anomalies that concurrent transaction can generate are:

1. Lost update: r1 - r2 - w2 - w1
   An update is applied from a state that ignores a preceding update, which is lost
2. Dirty read: r1 - w1 - r2 - abort1 - w2
   An uncommitted value is used to update the data
3. Non-repeatable read: r1 - r2 - w2 - r1
   Someone else updates a previously read value
4. Phantom update: r1 - r2 - w2 - r1
   Someone else updates data that contributes to a previously valid constraint
5. Phantom insert: r1 - w2(new data) - r1
   Someone else inserts data that contributes to a previously read datum

Anyway, since our domain is limited, not all the usual anomalies can occur. Let's analyze them one by one:

1. Lost update: since our system does not support operations with variables interleaving, reading a variable and consequentially updating it will never cause problems since a write don't consider the already existing values
2. Dirty read: if our scheduler never kills transactions in order to deal with deadlock there will never be rollbacks so the problem of dirty reads is, for now, marginal.
3. Non-repeatable read: this problem can actually occur in our scope
4. Phantom update: since our application don't support operations with multiple variables this anomaly is not part of the domain
5. Phantom insert: since our application don't support operations on groups this anomaly is not part of the domain

The concurrency control methods that can be used belong to 2 categories:

- Optimistic: based on the assumption that collisions will not arise often
- Pessimistic: based on the assumption that collisions will arise often

Pessimistic concurrency control methods are implemented through locks. 2PL is a pessimistic implementation that also deal with non repeatable reads. Anyway locking means the need to deal with deadlocks (so also dirty reads). Update lock is an alternative version of 2PL with 3 instead of 2 phases. This will avoid the most common type of deadlock: the one between only 2 concurrent transactions. Since we are in a distributed system detecting deadlocks is more challenging than in a non distributed configuration.

Optimistic concurrency control methods are implemented through timestamps. Each transaction has a timestamp representing the time at which the transaction begins so that transactions can be ordered by “birth date”: *smaller index* $\implies$ *older transaction*. Since the distributed configuration we should deal with the problem of achieving ordering between transactions. In this concurrency control method deadlocks never occurs but still transactions can be can be killed so we have to deal with rollbacks (so also dirty reads).
In our particular domain we can think about an optimization: since our application don’t support interleaving, we are allowed to reorder the operations in the transaction in such a way to have all the transaction of the same variable consecutively.
Since our optimization allow us to sort by key the operations in the transactions we can delete all the intermediate operations considering only the first read, if is performed as first operation of the considered transaction (since will read the value from a previous transaction), and the last write, since will be the only value visible to the next transaction once this one will terminate. Since, for 2PL, we cannot acquire another lock after releasing it (otherwise will not be ensured the serializability of the transactions) we still can encounter deadlocks.
To avoid them we can sort the keys in the same order for each transaction to be sure that if a transaction has a key k locked and wants to lock the key l (with key l successive in the order to k) no transaction can have a key l already locked with the will to lock k since the way the operations are ordered is always the same.
Another optimization can rely again on the ordering optimization: since, for each key, there will be at most one read before a write, is useless to require a shared lock for then update it on the real next transaction. So it can be feasible to rethink the lock system: if a transaction for a key has at least a write an exclusive lock will be request straight away, otherwise only the shared lock will be request as usual.

3.3. Replication

By requirement the data has to be replicated R times <= N nodes with R as parameter. This means that not all the nodes will have all the data stored. In order to understand who has to store a key we can either impose a balance for every node creating an agreement algorithm for storing a key or we can rely on some algorithm that, given a key, will return always the same node identifier for that key. With this we can be sure that, for a number of key high enough, all the node will store approximately the same number of keys.
Following the last idea, requesting a lock will mean only sending the request to a specific server without the need to find who has the key. Since is useless to replicate also the lock table for each replica that holds a specific key we can store the lock table only in one node.
So we need an algorithm that for each key will give us always one number that is associated to a server. This server will also hold the key locks. Then, in order to achieve replication, we can simply increase the number R - 1 times in order to identify a total of R servers that will hold the replica of that specific key.

4. Run-Time Architecture and implementation

Both client and server are implemented in Java. In order to select either to run a client or a server the application has to be run with -s (server) or -c (client)

4.1. Client

In order to properly execute the client a set of parameters has to be provided as arguments of the application:

```
java application_name.java -c localhost r(x),w(y)1,r(y)2,r(b)
```

or

```
java application_name.java -c 192.168.1.1 r(x),w(y)1,r(y)2,r(b)
```

where localhost or the IP address are the addresses of one of the servers. In order to handle the communication the client contact a server via socket always on the port 8080 with the transaction. Than it wait for the response and print it to terminal.

4.2. Server

In order to properly execute the server a set of parameters has to be provided as arguments of the application:

```
java application_name.java -s 3 2 localhost 192.168.1.1
```

where the first two numbers indicate respectively the total number of servers and the total number of replicas localhost and the IP address are the addresses of other servers. This list of addresses has to be the whole set of addresses already connected. This has be done since the purpose of the project is the key-value store so it will avoid to add complexity to the development.
Once the server have established the connection with every other server on port 4040, it will start to hear requests on port 8080 from the clients.
Each time a client will send a transaction the server will create a thread to handle it. This transaction received by the client will be already optimized client-side and the optimization will not be checked since we assumed reliable processes.
The thread for each key of the transaction (ordered alphabetically) will send an optimized operation (composed of at most one read independent from every write of the same transaction and at most one write that will not be overwritten by another write of the same transaction) to the node that holds the lock for that key. That node will compute the operation and will send the response back to the server. Once all the optimized operation are been performed the thread handling the transaction will send the input to release the lock on the keys to the holder of the lock table for each key. Before releasing the locks each holder will ask to every server holding a replica to update the value of the key (in case is needed). Since the value of the replicas will never get read or written we don't need our threads to wait till the fetching will be performed.

5. Conclusions

Since the format of the request and the optimizations achieved thanks to the domain assumptions, the replication is not helping to enhance the performances, and is, indeed, slowing down the system. In a real scenario the replicas would come useful to improve fault tolerance: once a process will fail, for whatever reason, the system would be able to reorganize himself, keeping on running even after the fail of a process.