Implementing a distributed transactional key-value store

Zhongmiao Li

zhongmiao.li@uclouvain.be

April 19, 2018

1 Introduction

Distributed key-value storage systems are the key building block of today's large-scale online servers. In this project, you should implement a simple transactional key-value storage, which allows *single-key update*, *multi-key snapshot read* and performs *garbage collection* to prune stale data. Among the various design choices, our data store will exploit the use of decentralized physical clocks to provide *snapshot isolation.*

2 Description

In this section, we will go through several key concepts in the design of the data store we are going to implement, in a rather high-level way. In the following text, we assume a key-value storage system, which stores two keys X and Y. The storage system consists of two data partitions Px and Py, which respectively stores X and Y (as shown in Figure 1).

Updating a key The most basic operation of the data store is to update a key's value (or if the key does not exist, creates it and inserts the value). While many data stores [4, 5] allow clients to issue *multi-key update transactions* to manipulate several data items in an atomic and consistent fashion, our system will only allow single-key update transaction. Furthermore, our data store will maintain multiple versions of each data item (the usage of multi-versioning will become clear later), which means the system does not really 'update' the value of a key, but instead appends a new timestamped version upon each update. Each version should be timestamped with the current physical time of the hosting data partition. As can be seen in Figure 1, if Px receives an insertion request for X, since its current time is 50, it will create version 50 for X.

Snapshot isolation Intuitively (more formal definitions can be found in [1, 2, 3]), a data store providing snapshot isolation executes a transaction as if it was executed under a snapshot of the data store, which is taken at the start of the transaction.

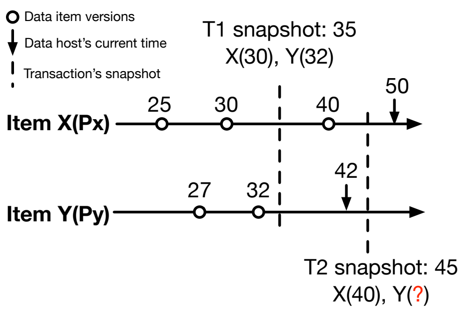

Figure 1: Demonstrating snapshot isolation. Item X is stored by partition Px and item Y is stored by partition Py.

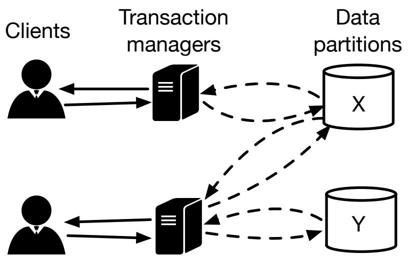

Figure 2: System architecture.

How can a transaction 'take a snapshot of the data store'? This is made possible due to the fact that we are already maintaining several timestamped versions of each data item! Basically, when a transaction starts, we assign it a snapshot time and allows the transaction to only read the *latest* item versions whose timestamps are smaller or equal to its snapshot time (even if there are newer versions!). As shown in Figure 1, transaction $T1$, whose snapshot time is 35, will read version 30 of X and version 32 of Y.

Snapshot unavailability A transaction may be blocked from reading a data item, if the snapshot it requires is still *unavailable* in the partition being read. Figure 1 shows that if $T2$, whose snapshot time is 45, tries to read Y, then it would be blocked since Py's current time is only 42. The blocking is necessary, because we do not know if Py will insert any newer version of Y between 42 and 45. Therefore, Py can only serve the read request of $T2$ until its physical time passes 45.

Garbage collection As you may have noticed, so far we keep creating new versions but do not remove old versions. Soon our data store will run out of space! So it is necessary for the data store to get rid of stale versions that are *useless*, i.e., *to perform garbage collection.* Though, the key challenge is to identify which versions are useless, namely the ones that will not be read by any transaction anymore. How do you think it should be implemented?

3 Implementing the system

After getting a high-level understanding of the system, in this section we will delve into more details that might be useful for you to implement the system. Note that you are not forced to obey all following descriptions, if you want to try other interesting design choices!

Logically, the system we are going to implement consists of three tiers of processes (as shown in Figure 2): the clients, transaction managers and data partitions. We first describe the two inner tiers and then the clients.

The data store Transaction managers are the proxy between clients and the data store, as clients are supposed to be (and they should be) oblivious of the internal structure of the data store. Transaction managers receive transactions from clients, coordinator and execute these transactions, and send back results to the clients. To forward a transaction to the responsible data partition, a transaction manager has to know how data is distributed among partitions. Despite various ways to achieve this, for simplicity we can use a deterministic hash function to decide the location of keys[1]. The hash function should be used both when updating and reading keys. Transaction managers should expose the following interfaces:

- UPDATE*(key::term(), value::term()) -> ok*: an update transaction takes a key and a value as its parameters. The receiving transaction manager forwards this operation to the data store partition hosting *key*, waits for reply and relays the reply back to the client. Since we assume there will be no failure, the update operation will never fail and always return *ok*.

- SNAPSHOT_READ*(keys::[term()]) -> [term()]*: upon receiving a snapshot read transaction, the transaction manager assigns *its current physical time* as the snapshot time of the transaction. Then, it sends a read operation for each key to its corresponding partition, accumulates the returned values in a list and replies the gathered values back to the client. Note that the order of values in the list should correspond to their keys in *keys* list.

Internally, the data store should consist of multiple data store partitions. We assume the data partitions are decided when the system is initialized, thus no partition can join or leave the system later. A data partition should expose the following interfaces:

- UPDATE*(key::term(), value::term()) -> ok*: when receiving an update request, the partition creates a new version of value for *key*, or creates the key and inserts the value if *key* does not exist. As described above, the partition should timestamp the created version with its current physical clock value. We will not discuss in detail here, but ensuring that a partition (which should be implemented as an Erlang process) gets monotonically increasing timestamp is crucial for the correctness of snapshot read. While this is not trivial[2], luckily you can invoke OS:TIMESTAMP() to achieve that. To efficiently maintain keys and values, you can use dictionary-like structures provided by Erlang[3].

- SNAPSHOT_READ*(snapshot_time::integer(), key::term()) -> term()*: as mentioned above, transaction managers will decompose a multi-key snapshot read transaction and send a read request for each key, thus a partition will only receive a snapshot read for a single key. If the specified snapshot time is smaller or equal to the partition's timestamp, the partition can directly fetch the corresponding version and reply with the value. Otherwise, the partition should somehow *buffer* this request and only serve it when its physical time is larger than the snapshot time.

[1] `https://en.wikipedia.org/wiki/Distributed_hash_table`
[2] `http://erlang.org/doc/apps/erts/time_correction.html`
[3] `http://erlang.org/doc/man/ets.html`, `http://erlang.org/doc/man/dict.html`

Although the above text distinguishes transaction managers and data partitions (as they are logically different), it is up to you to whether implement them differently: you can implement one type of process for each of them, or simply merge their roles by implementing data partitions that can act as transaction managers. Furthermore, we leave the design of garbage collection to you; implementing it correctly and efficiently can earn you extra bonus.

Clients The main purpose of having client processes is to evaluate the performance of the system. A client process issues an update/snapshot read/garbage collection transaction to the system, waits until the transaction completes and issues a new transaction. Ideally, you should start with one client process and count how many transactions you can complete per second (i.e. throughput), then spawn more and more processes, until the throughput does not increase any more. In order to not create too many client processes, you may allow a client process to send multiple requests to the system simultaneously.

Reading from input files To simplify the testing of the system, you can program a script that can read transactions from input files, execute them and output the results. Each line of the input file should be parsed as a transaction. The possible formats (along with the explanation) of a line are listed as follows:

- **up** key value: update *key* with *value*.
- **read** key1 key2 ... keyn: do a snapshot read for the given keys.
- **gc**: do garbage collection.
- **sleep** time: the input file parser sleeps for *time* milliseconds.

4 Submission and grading criteria

You should implement the system in Erlang, deploy the system and evaluate its performance and then write a report. Initially, you may deploy the system in a single machine; deploying and evaluating it in a distributed setting will earn you bonus. The report should cover 1) the design and implementation of the system, such as how each components interact and what data structures you use to maintain data etc., and 2) the experimental results, such as how the system is deployed, what workloads are used and the evaluated throughput. The length of the report should not exceed four pages.

Your should submit the source code of your program, a README file describing how to start your system and how to specify the input file (if it is possible) and the report. The detailed evaluation criteria are as follows:
Allowing insertion: 10 You can get 10 points, if you: implement clients, transaction managers and data partitions (as mentioned, you can combine the role of data partition and transaction manager); evaluate your system to check the throughput of insertion; write a report that describes the design and architecture of your system and report the evaluation results.
Allowing snapshot read: +2 Implement snapshot read and evaluate the throughput limit of snapshot reads. Describe how you implement it and the evaluation results in your report.

Reading from input files: +2 Implement this feature and describe how you implement it.

Note that to get the following bonus, your system must implement all above features, and the system should have at least two data partitions and two transaction managers.
Distributed deployment: +2 You should deploy the system in at least two physical machines. The report should describe how you manage to run them across machines and the deployment.
Well-designed garbage collection: +2 You should implement the garbage collection strategy, evaluate its impact on performance and memory usage and describe those in the report.
Extensive benchmark: +2 You should evaluate the system in depth. For example, you can evaluate the system under several mixes of transaction ratios (update/read/garbage collection), and for each mix present a graph of throughput versus latency, where each point on the curve is for given load.

References

[1] Jiaqing Du, Sameh Elnikety, and Willy Zwaenepoel. Clock-si: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *Reliable Distributed Systems (SRDS), 2013 IEEE 32nd International Symposium on*, pages 173–184. IEEE, 2013.

[2] Sameh Elnikety, Fernando Pedone, and Willy Zwaenepoel. Database replication using generalized snapshot isolation. In *Reliable Distributed Systems, 2005. SRDS 2005. 24th IEEE Symposium on*, pages 73–84. IEEE, 2005.

[3] Andrea Cerone and Alexey Gotsman. Analysing snapshot isolation. *Journal of the ACM (JACM)*, 65(2):11, 2018.

[4] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google?s globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[5] Deepthi Devaki Akkoorath, Alejandro Z Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno Preguiça, and Marc Shapiro. Cure: Strong semantics meets high availability and low latency. In *Distributed Computing Systems (ICDCS), 2016 IEEE 36th International Conference on*, pages 405–414. IEEE, 2016.