

VTL - Variable Transactional Layer

RÚBEN SILVA, Instituto Superior Técnico, Portugal

There is a variety of distributed transactional systems that differ in the models and mechanisms they employ — how a system writes an object, which concurrency control mechanisms they use to achieve transactional properties, and more. Despite the variety of the models among different transactional systems, most systems are designed with a single transactional model in mind. Thus, they provide little to no support for updating or swapping between different models. Additionally, we noticed that for most transactional systems, there is a common set of steps that need to be executed in order to begin and end a transaction. Given these two aspects — the lack of changeability and the shared components — we propose a framework that can modularize common and mandatory steps of distributed transactional systems with standard combinable interfaces. This would allow, with the aid of a configuration file, the user to interchange between different models of transaction at build time. This framework is targeted at a varied set of existing transactional systems that are representative of almost all existing different systems. The VTL framework is ported into a system where we implement two of its transactional models (pessimistic and optimistic) and evaluate them. This evaluation procedure revolves around comparing the results of the system with and without this additional layer. We demonstrate that the framework behaves exactly as the two distributed transactional models of the system, in terms of functionalities, despite the split into different blocks. Although the flexibility and changeability aspects are impressive, it still translates into a cost of around 60% of the throughput without the framework.

Additional Key Words and Phrases: Framework, Transactions, Distributed Transactional Systems, Multiple Transactional Models.

ACM Reference Format:

Rúben Silva. 2021. VTL - Variable Transactional Layer. 1, 1 (November 2021), 10 pages.

1 INTRODUCTION

In computer science there is constant change and evolution, almost everything is coded in a particular way with the objective of being easily adaptable and understandable. This aspect is vital for IT businesses, around 85-90% of its software budgets are spent in code evolution [16]. However, regarding transactional systems, most of them are designed with a single model in mind and provide little to no support for updating or re-configuring it [11; 12; 14; 15; 21; 22]. If the user decides to change a particular aspect of the model being used, most of the time he will just have to change to another system instead of changing a parameter on the system currently using. Consider the scenario where a user using Google Spanner [11]. After an analysis, the user verifies that conflicts are rare in their system,

and realizes that an optimistic approach would benefit the system's performance. Since Google Spanner does not provide an alternative concurrency control protocol, the user has to change to another system, for example, Centiman [12] which provides an optimistic concurrency control protocol. There is a huge variety of systems where each one of them try to explore different sets of properties (different levels of consistency, different concurrency control mechanisms, and others). System architects can get overwhelmed with having to choose the most efficient transactional model for their system. After choosing a model, if they realize that it is not the most efficient one, it might not even be worth it to swap it, due to all of the work that it is to use an entire different system. In the context of academia, researchers, when studying transactional models, might need to build all of the desired models from the group up, when often they just want to change small parts of a whole. It is hard to find solutions that offer a simple and quick way to swap between several different transactional models.

Systems that are distributed into several nodes and use transactions to process and execute user requests, distributed transactional systems, can differ significantly. They might use different concurrency control mechanisms with the intent of guaranteeing the ACID properties of the transactions. For example, Google Spanner uses two-phase locking [11] to obtain exclusive access to a transaction's data items, Centiman handles an optimistic concurrency control mechanism [12] to detect concurrent access to transaction's data and ecStore adopts a combination of an optimistic concurrency control with a multiversion mechanism [13] to provide a valid view of the data to the transaction. Isolation levels are also another aspect that systems normally vary. They consist in defining how is a transaction isolated from others that are executing concurrently. In the lowest isolation level, transactions can read uncommitted changes (and other consequences), which it is called dirty reads. At the highest isolation level, unless they are read-only transactions, transaction act like they are executed in sequence instead of concurrently. This guarantees that no transaction will access changes made by another one before it has committed. However, the increase of isolation levels will reduce the throughput of the system. Besides the concurrency control and the isolation levels, systems also vary in other aspects, such as in the replication scheme, how transactions are timestamped (using logical clocks, true time, etc), and more. Users are responsible for understanding and making a decision of which system is more suitable for them based on what properties they value more. If a user is trying to access his bank account details, he values its robustness and the consistency of its details much more than the time it takes to complete his order. On the other hand, (less critical systems, that value speed over consistency) value its availability or time instead. These types of aspects, that exist in every transactional system and that might differ depending on what the user or the service provider want, need to be taken in consideration in order to build our solution.

It is visible the abundance of distributed transactional models and how differently they behave. For each different model there

Author's address: Rúben Silva, rubenagsilva@outlook.com, Instituto Superior Técnico, Lisboa, Portugal.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2021 Association for Computing Machinery.

XXXX-XXXX/2021/11-ART \$15.00

<https://doi.org/>

is almost a distinct way of how an object is written, what to do after the write of an object, when the transaction commits, waits or aborts, and several other aspects. However, the steps that each distributed transactional system goes through are essentially the same for all systems. For instance, all of them need to replicate the data, validate the transaction, timestamp the transaction, etc. What changes between them is, how each step is done, to whom to replicate, how the replication is done, how is it timestamped, and more. Given the issues regarding changeability between different transactional models, and the common mandatory steps needed to execute transactions for most distributed transactional systems, we had the idea of creating for each step a generalized library that distinct protocols implement from and that allow to swap between different implementations with a simple configuration file or switch case.

Our goal for this thesis is to develop a framework that can modularize common and mandatory steps of distributed transactional systems with standard combinable interfaces, that allows interchanging different models of transaction. With this, system architects can simply use existing components, and explore how varied transactional guarantees can be constructed using these standard building blocks. For instance, assuming our VTL is added to Google Spanner, in order to change to another concurrency control mechanism, (to an optimistic locking mechanism) they only have to change the concurrency control interface, from a 2PL (2-Phase Locking) to a OCC (Optimistic Concurrency Control). In context of academia, this is absolutely useful, since most of the times when you are building a new transactional approach, you have to build everything from the ground up, when often you just want to change small parts of a whole. With the aid of our framework, you can simply do that by changing which interface of a particular module you want to change. And in case our framework does not yet support a particular algorithm, you can implement just the algorithm itself without having to implement the complete system.

2 RELATED WORK

2.1 Steps of a distributed transaction

From a user requesting a transaction to be executed, to the transaction finishing, numerous steps need to occur that are common to all systems. These steps might have different names, be aggregated inside other steps, or even done at different times. But they are mandatory in most distributed transactional systems. We defined them as:

- **Preparation.** It consists in preparing the execution of the transaction. Before a transaction is executed, the system might need to timestamp the transaction, replicate it to other nodes, etc. This phase is where all those operations are done. For example, we will view in section 2.2, that Calvin needs to timestamp transaction at arrival before they are executed. We consider that process to belong in the preparation step;
- **Execution** of all read and write operations requested by that specific transaction;
- **Validation.** This step consists in approving each operation of a transaction, in the sense of, guaranteeing that there are not

other conflict concurrent transactions. In other words, it revolves around mutual exclusion. Different algorithms do this step differently and at different times. Optimistic [12; 24–26] perform their validation after all operations of a transaction are executed. Pessimistic protocols [15; 27–30] on the other hand, do this step while they are executing the transaction. They guarantee, with the use of mutual exclusion algorithms (locks, for example), that no other transaction is accessing or writing that particular object. In both algorithms, the outcome of the validation selects what is done on the next step. If their validation is not approved the transaction will abort. If the validation is valid, the transaction commits;

- **Commit or Abort** step is done after the validation of a transaction and it results on the transaction being committed or aborted. The transaction will commit the changes to the database if it was successfully validated in the previous step - no conflict was found that negates any property of the system. As soon as the changes are durable, meaning that in any crash scenario, the changes done by the transaction will not be lost, the transaction is ready to finish. Alternatively, if some conflict invalidates a property defined by the system, the system failed to be validated and it is going to abort. Aborting means that the every modification and access that transaction made is reverted and discarded. After which, the transaction is ready to finish. For systems that have automatic retry, they would just jump to the execution step and retry to execute the transaction again;
- **Replication** step is essential to propagate the results and the operations that a given transaction has committed. Some systems even replicate the transaction's input to replicas [14];
- **Finish** This step is ran normally after the transaction committed or aborted. It consists in releasing any resource it still holds (releasing locks for example).

2.2 Discussion of existing systems

Our aim was to choose a varied set of existing transactional systems that are representative of a variety of different relevant protocols. We have chosen systems with optimistic concurrency control, 2-phase locking, deterministic locking and multiversion. Besides the concurrency control, we have also chosen systems that differ on other aspects. Such as, whether the replication is synchronized or not, does the transaction get timestamped and more. We analysed Calvin [14], Google Spanner [11], Centiman [12], Granola [15] and EcStore [13]. By analysing the above mentioned systems, it is visible that they share a notable amount of components, and that in some of them different mechanisms are used. Before jumping into the architecture of our solution, first, we must briefly discuss the various different components and mechanisms that are shared and mandatory in the systems.

Considering timestamping, the portion of the systems discussed timestamp the transaction before it commits [13–15]. Multiversion approaches also use timestamps in order to identify different object versions [11; 13]. Calvin [14] uses this timestamp as a concurrency control by pre-ordering the transactions, and executing them in that order. On the other hand, Granola [15] uses this timestamp

as a coordination mechanism to provide serializability for single-repository and independent transactions by propagating timestamp ordering constraints between repositories. Other systems timestamp transactions at commit time [11–13]. Google Spanner uses real time timestamps with Google True Time API. Centiman timestamps a transaction right when it is ready to commit to enter the validation step. EcStore [13] timestamps twice, once at the start of the transaction and on commit time.

All systems need to replicate their objects. In order to do that, they also need to know which nodes are present in the system and where they have to replicate to. Existing systems might want to replicate to only a set of nodes or to all of them. In Spanner [11], the applications can choose where to replicate to. In Granola [15], repositories are replicated using a set of $2f+1$ replicas to survive f crashes. Besides replicating to a different set of nodes, how the replication is done can also vary. Systems might only support synchronous replication using a consensus protocol like Paxos [11; 15]. In contrast, systems might only support asynchronous replication [13]. There are systems that even support both synchronous and asynchronous replication [12; 14].

Another aspect that is present in all distributed transactional systems is the concurrency control. Various systems use different concurrency control mechanisms: deterministic locking [14; 15], two-phased locking [11], optimistic control [12] and even a hybrid between optimistic and multiversion [13]. All of them perform the exact same set of operations (read, write, commit, abort, etc) but, as we can see, they can be done in distinct approaches.

3 ARCHITECTURE

As previously noted, especially in section 2.2, various different systems possess a variety of common steps.

Most of the systems need to timestamp transactions, replicate transactions or objects, communicate with other nodes in the network and manage the concurrency control of the transactions. The route taken to achieve each and every one of these steps can differ greatly, depending on the protocol the system has implemented. For example, as we have seen, relating concurrency control, Calvin [14] uses deterministic locking, while Centiman [12] employs an optimistic concurrency control. It is essential to note that, even though it exists high variability in distributed transactional systems, there are few options to aid changeability. In other words, systems normally only have one concurrency control protocols implemented [11; 12; 14; 15; 21; 22; 24; 25]. If we conclude it is more beneficial to change to another protocol, that would prove itself to be quite the task — unless we alternate to another system. We realized that the steps that most systems share can be generalized into white boxes which we referred to as modules, hereafter.

Since our goal is to implement a generalized modular framework that is able to represent a wide group of distributed transactional systems, we had to correspond the mutual phases into different modules. Each module will be a generalized interface that is always present and most of the time implemented differently depending on the system.

Different systems might use different ways to represent the nodes in the system — ring structures, tree structures, fully connected

network and more [13; 22; 31]. First and foremost, since this is a framework for distributed systems, it is important to have a component responsible for tracking the existing nodes in the system. It should also be in charge of initializing the communication between them. With this in mind, we defined the module group membership to fulfill these roles. Secondly, we confirmed that timestamping is crucial and that can be done differently depending on the distributed transactional system. Additionally, transactions might also be timestamped at different times. We have seen that for Calvin [14], they timestamp the transaction as it arrives which is then executed in order based on the arrival times. Thus, we defined an order module to be responsible for timestamping transactions and ordering them. The next module we found to be essential, is one that is in control of replicating objects and transactions to a set of nodes. The structure of the system dictates to which nodes the replication process is to occur. The process in itself, how it is to be done (passively or actively), is also dependant on the system, as well as the algorithm. We designated this module as the replication module. The last module we defined tackles the management of the concurrency control being used by the system. As we have seen, there is an extensive amount of distinct concurrency controls. However, since most of them share the same set of operations (all of them have to read and write operations, commit, abort, release resources, and more), we can represent them by a single generalized module. This is the Concurrency Control Manager module

3.1 Data Structures

Regarding data structures, it is crucial to fully grasp the concepts of **transaction**, **content**, **node** and **message**.

Transaction is an object which contains the write-set and read-set of its operations as well as the locks the transaction has detained. Each time an operation (read or write) of a particular transaction occurs, it will add to its read-set or write-set respectively. Whenever a transaction is being replicated, it includes the replication of both of these sets. This object is also timestamped during the flow of the execution of a new transaction. Some systems need to store the timestamp of arrival, but all are required to store when the transaction commits. Both of these timestamping values are stored in the object transaction.

Content essentially acts as a key/value store. It has a key and a value which corresponds to the object being stored. For systems that do not use key/value store, one has to convert their structure into a key/value store. To read and write contents, the operations need to have the content key to access or update the value of the object.

Node is a simple object which contains the information necessary to contact a particular node (for example, attributes such as socket ports, ID and more). It also has a list of contents which indicates the write operations of that particular node. For instance, suppose that node 1 executes a transaction with only one write operation. After the commit of the transaction, the changes to the database (this write operation) will get replicated. Every node that receives the replicate message, will add this new modified object to the list of their node 1, knowing that node 1 updated that particular object. This proves to be extremely useful if a node crashes, since the nodes

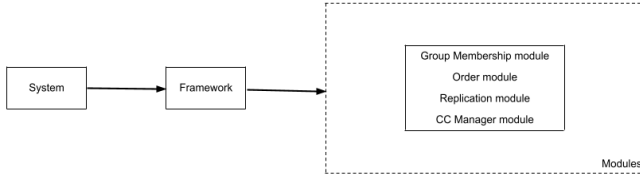


Fig. 1. **System overview** - An overview of the system with this layer implemented.

that received the replicate messages could take the responsibility of the objects that the failed node had. The Node object also contains a role enumerator which indicates if it is a timestamp or only a forwarder. Which indicates if a node is allowed to timestamp transactions. Timestampers are responsible for timestamping transactions, while forwarders are incapable of doing so, so they will instead resend the running transactions to a node that has the timestamp role. This allows two different timestamping models. On the one hand, we have a centralized approach where there is only one timestamp node and every other node needs to contact it to timestamp it. On the other hand, alternatively, we can opt for a decentralized approach where every node (or only a particular set of nodes) are timestampers.

Message is a simple serializable object that is utilized in the communication between different nodes, for example, in the replication of transactions. It contains the sender and receiver node ids, the transaction to replicate (and its changes to the database) and finally a message type. This message type has only two values: Replicate and Replicate Writes. The first has the purpose of replicating the transaction to other nodes, making them capable of running that transaction, if needed. The latter has the intent of replicating the changes done to the database by that transaction. This message object will be sent from one node's communication object to another node's communication object. We will observe in more detail how the communication is done in section 3.3.

3.2 Framework

The framework has two main objectives:

- (1) Function as a gateway between the system and the different modules, the rest of the architecture.
- (2) Establish a proper flow depending on the transactional model selected.

To fulfil the first objective we have to understand the system we are porting into, and find where it executes all the relevant methods to the framework. Then, we must call the respective methods from the framework. For instance, once we located where a new transaction is created, we had to call the `beginTransaction` method from the framework. Additionally, we also had to understand what the system does once a new transaction arrives, and portray that to the `beginTransaction` method of the framework. If it timestamps the transaction once it arrives and replicates it, then our `beginTransaction` needs to also timestamp it and replicate it, in order to comply with the transactional model displayed by the system. This analysis was essential for every method of the framework interface. We will

describe the steps of how to port the framework into a new system in the next chapter, in section 4.3.

The framework can be seen as a module of the system because like the modules, it should be specific for every different system and implement from a generic interface. This occurs due to the implementations of different systems and especially, its transaction models. For instance, a begin transaction of a locking model might not need (depending on the system) to timestamp the transaction once it arrives. However, in a system like Calvin [14], once the transaction arrives, it needs to be timestamped and executed in that arrival order (based on the timestamp).

Having that in mind, the methods we defined for the framework API were: *beginTransaction(txn)*, *read(key,txn)*, *write(key,value,txn)*, *validate(txn)*, *commit(txn)*, *abort(txn)*, *replicate(txn)* and *endTransaction(txn)*.

3.3 Modules

Module: Group Membership - The group membership module focuses on which nodes participate in the system and the roles they play. We defined two possible roles: timestamp and forwarder. Timestampers are responsible for timestamping transactions. In contrast, forwarders are incapable of doing so, so they will instead resend the running transactions to a node that fulfills a timestamp role, in order to timestamp the given transaction. As previously explained, this allows different types of timestamping models. A system that needs a centralized timestamp, for example, Google Spanner [11], only needs one coordinator. This node is solely responsible for timestamping every transaction, thus, every other node cannot timestamp them. Therefore, you would classify the coordinator as a timestamp and all the other nodes as forwarders. For leader election systems, leaders are classified as timestampers and slaves as forwarders. In these two scenarios, it makes sense to have the system composed by timestampers and forwarders. However, if you need a fully decentralized system, every node will only have the role of a timestamp. The methods defined for the group membership API were: *getNode(id)*, *getNodes()*, *getReplicationTargets()*, *getTimestamp()*.

Module: Ordering - This module is responsible for ordering transactions and timestamping them. It might be accessed to timestamp transactions at different times. It is normal for systems to timestamp only on commit. However, some need to timestamp the transactions on arrival to define an execution queue. In other words, transactions are ordered and then executed in that sequence [14]. This module has an object clock. This object is an interface and it depends on which algorithm the clock parameter was instantiated. This means that, the order module is independent of the type of algorithm the clock is launched with (logical clock, real time, etc). The methods defined for this module's API were: *timestampStartup(txn)*, *timestampCommit(txn)*, *compareTransaction(txn1, txn2)*, *lockGet(content)* and *lockRelease(content)*.

Module: Replication - The replication module focuses on the interactions regarding replicating transactions between different nodes in the system. Some systems might need to replicate the transaction once it arrives to other nodes in the network, to guarantee that the transaction is executed. Basically, the idea is that, when a

transaction first arrives, a node will be responsible for executing it. We replicate the transaction to other nodes allowing that, if the node responsible fails, the transaction can be executed by other nodes. After a transaction commits, systems require that their changes get replicated to other nodes to guarantee the durability property of that transaction to the database. This grants a feature in which all nodes that receive the replication message know the changes done to the database. Therefore an already existing node can now be responsible for the objects that the failed node was previously accounted for. Before executing any of the replication methods, the replication module first needs to gather the list of nodes to where it should replicate. Different algorithms might have different destination nodes (replicate to only one node, to all nodes, etc.). Group Membership module returns this list with `getReplicationTargets()` method. This module's API consists in: `replicate(txn, nodes)`, `replicateResult(txn, nodes)`.

Module: Concurrency Control Manager - This module is responsible for managing the resources based on the concurrency control mechanism. This module is responsible for managing the resources based on the concurrency control mechanism. By observing different systems we concluded that, independently of the concurrency control mechanism, all of them need to initialize their concurrency control mechanism, execute read and write operations, validate, commit, abort and finish transactions. If there are several different concurrency control protocols, we strongly recommend to have a module for each one of these mechanisms. When creating/importing the database it is important to initialize all necessary concurrency control mechanisms for each object in the database. For example, create the locks for each object in the database. Once that is done, operations (reads and writes) will get executed by a transaction, which will call these read and write methods, respectively. For each operation, this module will gather the necessary resources and release them if the isolation level selected requires to do so. Besides these methods, the manager will also call any validate, commit, abort and finish methods. For pessimistic approaches, the validate method should not be implemented nor called from the designated framework. All other methods are shared between every concurrency control protocol. These methods consist of this module's API: `initContent(key, object)`, `write(txn, key, object)`, `read(txn, key)`, `validate(txn)`, `commit(txn)`, `abort(txn)`, `finish(txn)`.

3.4 Communication

In a distributed environment it is essential that nodes communicate with each other. We need this functionality for the replication aspect and in case it is desirable to implement fault tolerance (in the sense of, the transactions that were about to be executed by a node, are executed by another in case of crash). Different systems will have the communication done differently, via sockets, via SOAP web services and other ways. Therefore, we also need to provide a level of flexibility in the communication interface to allow the implementation of a variety of different mechanisms. The communication between the nodes of the system rely on the use of the object Message previously described in 3.1. It basically contains a message type which indicates the receiving node what kind of operation we are doing. Depending on the message type, the node

receiving the message, will either store a new transaction or store the writes that transaction did. It is important to remember that, just like in the modules, if a developer does not want the communication with sockets and instead desires with SOAP web services, the developer only needs to implement the methods of the interface and the algorithm. The communication API used by each node has the following methods: `get(msg)`, `sendMessage(msg, targetCommunicationInterface)`, `initReceiver(nodeId, isClient)`, `initSender(nodeId, isClient)` and `recvTransaction()`.

The initialization of the communication of the nodes (`initReceiver()` and `initSender()`) is done when the Group Membership is creating every node. Each node will be able to send messages to every other node and also receive them. For the messages to get received, there must be a thread that is constantly looming for new messages.

The flow of the communication between nodes is rather simple. After they have been initialized with `initReceiver()` and `initSender()` in the group membership module, they are ready to send and receive messages. To send a message to a node, the node only needs to call the method `sendMessage(msg, targetCommunicationInterface)` through its communication attribute. The destination node has a running thread that is waiting for messages (method `recvTransaction()`). Once the message arrives, the thread will execute `get(msg)` which will read the type of the message and the message will get processed as intended. (For example, if the message type is to replicate the transaction, the get message will store that transaction).

4 IMPLEMENTATION

We next justify our procedure of choosing a system to port our framework into. In addition, we will discuss what scenarios we discarded for our solution.

4.1 Which systems and What Language

To demonstrate the potential of our solution, we had to choose a system that has multiple transactional models, different replication protocols, and so on. The alternative of finding one system that contains all these desired properties, is to port into two different systems that together contain what we desire. We value the first option more, due to the overhead of learning how the system works, before actually implementing and porting the framework. Besides valuing these aspects, we also had to take in consideration on what language the system is coded in and how complex is the code to understand and port the framework into. For every system we considered, we located where the relevant aspects of the framework were (where the replication occurred, how nodes are represented, concurrency control mechanisms and everything else). Once located, we decided to attribute a score (0-10) of how complex the system appeared to be. Although, the results show that Ignite [22] is the system with lowest score, it only supports multi-version concurrency control. Thus, we would had to pick an additional system to represent a different transactional model. The problem was that the other java systems that we considered, had quite bigger scores comparative to the others. The second on the list is Deneva [19], which represents a wide variety of different transactional models, plus, it has a decent

score difficulty associated. Given these reasons, we decided to port our framework into Deneva.

4.2 Assumptions

The system we ported our framework into, Deneva [19], has a in-memory distributed database that is partitioned between the nodes of the system. For every operation (read or write) being executed by a transaction, the system needs to contact the partition which contains that desired object, and execute the operation. Once the transaction finished executing all its operations, it is ready to commit or abort and to finish. To do so, the system needs to contact every partition and commit/abort their changes and release the resources that they had allocated.

Deneva does not have replication implemented. We had to decide between implementing replication or adding transaction recovery to our framework to retrieve a previous state in case of aborts. We opted for a simple replication implementation. How the replication is done was explained at the replication module section. By implementing replication we postponed the idea of transaction recovery in the case of transactions aborting. We know that this feature is essential in almost every transactional system but yet considered that replication has a bigger footprint. It is important to remember that we implemented a naive way of replication and do not wait for acknowledge messages. We do this so the values of throughput and latency are not as damaged, allowing us to still compare these results with the original Deneva.

We do not support node failure. However, the replication architecture and implementation puts us a step closer to supporting node failure. Basically, the idea is that, when a transaction first arrives, a node will be responsible for executing it. We replicate the transaction to other nodes allowing that, if the node responsible fails, the transaction can be executed by other nodes. Another idea is that, when a transaction commits, the changes of the database will also be replicated. This grants a feature that all nodes that receive the replication message, know the changes done to the database and therefore an already existing node can now be responsible for the objects that the failed node was responsible for.

4.3 Porting into a different system

To port the framework to a different system, you need to have the knowledge of the system and know how a transaction is handled. Every system behaves differently and handles the transactions distinctly. The first step is to implement the framework corresponding to that system which are basically calls to methods of the modules. For instance, a begin transaction in a system like Calvin, need to timestamp the transactions at arrival, therefore, in the framework, it would call the module Order to timestamp the new transaction. However, in Google Spanner [11], transactions do not need to be timestamped on arrival, so there will be no call of the module Order in begin transaction method of the framework. You need to figure out what happens to a transaction for each of the methods of the framework interface.

Algorithm	Iso. level	Txn 0	Txn 1	Txn 0 Commit	Txn 1 Commit	Original Deneva	Observation
One Server & One Client	Locking	Serializable	R	R	✓	✓	The transaction that first gets every lock will be able to commit, while the other aborts
			R	W	✓	✓	
			W	R	✓	✓	
			W	W	✓	✓	
	Locking	Read Committed	R	R	✓	✓	Read locks are released right away, allowing the next operations. Write locks are only released on commit (with this isolation level)
			R	W	✓	✓	
			W	R	✓	✓	
			W	W	✓	✓	
	Locking	Read Uncommitted	R	R	✓	✓	Every lock is released right away. This allows every operation to execute right after
			R	W	✓	✓	
			W	R	✓	✓	
			W	W	✓	✓	
	OCC	Serializable	R	R	✓	✓	The read cannot be validated, since at least a write has occurred on the same object by another transaction
			R	W	✓	✓	
			W	R	✓	✓	
			W	W	✓	✓	

Fig. 2. Validation of conflicting scenarios in different protocols implemented by our framework in regard to the original Deneva in a centralized environment. Two transactions executing concurrently varying which operations are done, the concurrency algorithm and the isolation levels. We showcase which transaction commits out of the two, and if the exact same result occurs in the original Deneva.

Besides knowing how a transaction is handled in that particular system, you also need to know where does the manipulation of the transaction occurs. In other words, you need to locate important operations done to a transaction and call the respective methods of the framework. These relevant operations are declared in the framework interface and consist of: *beginTransaction(txn)*, *read(key,txn)*, *write(key,value,txn)*, *validate(txn)*, *commit(txn)*, *abort(txn)*, *replicate(txn)* and *endTransaction(txn)*.

Additionally, the modules are also likely to be distinct from system to system, therefore, it is recommended to implement them given the target system. For example, the replication module, it states how and to whom does a transaction need to be replicated to. For a system it might need to replicate only to a set of nodes, while in another system, to every node or to no other node. Another important aspect when importing the framework to a different system, is if the algorithms it utilizes are already implemented. For example, the system might need to use logical clocks, which are not yet implemented, in the order module to timestamp the transactions. In these cases, you have to implement the algorithms in need and call them in the modules desired. It is important to note that, the framework, each one of the modules and algorithms have an interfaces that should be implemented and called from, from the respective modules. This allows to swap between algorithms by changing a variable on the configuration file. Plus, if there is a change to be made on the algorithm, everything else is unchanged. However, not every algorithm/module might fit perfectly with the interface. For example, the interface for the manager of the concurrency control has a method of validation, even though the locking mechanism does not validate the transaction. Once the framework, modules, necessary algorithms and calls to the framework are implemented, the only thing remaining is to add to these different alternatives for to the switch cases in the configuration class.

5 EVALUATION

We will discuss the results of our tests in this section. Since we ported our variable transactional layer into Deneva, it makes sense to compare each testing result with it. This means that we had to execute all the scenarios for the system without the additional layer,

	Algorithm	Iso. level	Txn 0		Txn 1		Txn0 Commit	Txn 1 Commit	Original Deneva	Observation
			R	W	R	W				
Multiple Servers & One Client	Locking	Serializable	R	R	✓	✓	✓	✓	✓	The transaction that first gets every lock will be able to commit, while the other aborts
			R	W	✓	✓	✓	✓	✓	
			W	R	✓	✓	✓	✓	✓	
	Locking	Read Committed	R	R	✓	✓	✓	✓	✓	Read locks are released right away, allowing the next operations Write locks are only released on commit (with this isolation level)
			R	W	✓	✓	✓	✓	✓	
			W	R	✓	✓	✓	✓	✓	
	Locking	Read Uncommitted	R	R	✓	✓	✓	✓	✓	Every lock is released right away. This allows every operation to execute right after
			R	W	✓	✓	✓	✓	✓	
			W	R	✓	✓	✓	✓	✓	
	OCC	Serializable	R	R	✓	✓	✓	✓	✓	The read cannot be validated, since at least a write has occurred on the same object by another transaction
			R	W	✓	✓	✓	✓	✓	
			W	R	✓	✓	✓	✓	✓	

Fig. 3. Validation of conflicting scenarios in different protocols implemented by our framework in regard to the original Deneva in a distributed environment. Two transactions executing concurrently varying which operations are done, the concurrency algorithm and the isolation levels. We showcase which transaction commits out of the two, and if the exact same result occurs in the original Deneva.

as well as, the system with the framework ported into. We will address Deneva without the framework as the original Deneva.

Firstly, we will discuss the correction tests. These tests try to showcase that our framework behaves exactly as the original Deneva in terms of functionalities, despite the split into different blocks. We focused on the more complicating scenarios, where transactions are executing concurrently, accessing and writing the same objects. For different algorithms and isolation levels, we will execute different operations and make sure our solution and the original Deneva make the same decisions on which transactions commit and abort.

Secondly, we will evaluate the system with our variable transactional layer in terms of throughput and latency and compare it with the original Deneva. We will discuss the overhead that our framework brings in relation with the original implementation. We will vary the amount of servers and the contention levels for both transactional models. In our evaluation the YCSB benchmark [20] will be used.

We deployed the framework on GSD Cluster hosted at IST-DSI. The amount of nodes will depend on how many clients and servers were used for each experiment. Each node consists of a CPU of eight cores with 2.13GHz speed and 40GB of memory. Before each test, table partitions are loaded on each server. The first 60 seconds is a warm-up period followed by another 60 seconds of measurements with a load of 10000 open client connections per server.

5.1 Correction tests

For this scenario of tests, there will be two transactions were executed simultaneously and concurrently. The operations done by each one are represented into R and W, corresponding into a read operation and a write operation, respectively. These operations are performed onto the same 4 objects concurrently. For example, each transactions will make an operation (Read or Write) to objects with identifiers of 1, 2, 3 and 4. To guarantee that the transactions are executing concurrently, besides being started at approximately the same time, each one of them will sleep for one second after doing each operation. This means that each transaction will take at least four seconds to execute. This disallows a transaction to finish quickly before the other one even began. **It is important to note that,** both transactions are being executed concurrently, but transaction

0 will always execute the given operation right before transaction 1. In other words, the transaction that gathers the first lock for the first object, will always gather first the lock of the second object and so on. For instance, Transaction0 will read object 0 and sleep for 2 seconds, transaction1 will read object0 and sleep for the same amount (2 seconds). After 2 seconds, transaction0 will read object1 and sleep again, transaction1 will read object1, sleep, and so forth.

We have two different transactional models, a pessimistic one and an optimistic one. On the pessimistic approach, three different isolation levels are implemented. Reads uncommitted, reads committed and serializability. In the optimistic approach only serializability is implemented. So we executed two transactions concurrently, accessing the same four objects and perform different operations on all of these different isolation levels for both transactional models. In addition, since this is a distributed partitioned approach, we did these tests for both a non-distributed system (executing only on one server) and for a distributed system (multiple servers).

Pessimistic approach - We can observe that in the pessimistic approach there are three different isolation levels. If the isolation level is serializability, the highest isolation level, we can view that if both transactions read the same objects, that both will commit successfully. Although, for the other three scenarios, one transaction will always abort. In these three scenarios, a transaction can successfully commit if, and only if, it can gather every lock that it requires before any other transaction executing concurrently. In this scenario, we considered that the transaction that gathers the first lock for the first object, will always gather first the lock of the second object and so on. In this particular scenario, the transaction that gathers all the locks first, will successfully commit and the other transaction will abort.

For read committed we can see the appearance of non-repeatable reads, meaning that, during the execution of a transaction, reading the same object could read different values. In this scenario, the locks that are shared (from read operations) are released right after the read operation is complete. Observing the table 2, we can continue to see that both transactions still commit when both execute read operations. When transaction 0 reads and transaction 1 writes, we can see that both are allowed to commit. However, when it is the other way around, only the write is allowed to commit. This occurs due to the read locks now being released right after the operation is executed, while the write locks are still only being released on commit and abort. This means that, once a transaction has acquired a write lock, no other concurrently executing transaction can execute any operation on that object.

For the isolation level of read uncommitted, the transaction are allowed to commit for every scenario. Here, the locks are released immediately after the operation is executed instead of waiting for the transaction to finish.

Optimistic approach - For the optimistic approach, only serializability is implemented and observed. Just like in the pessimistic approach, if both transactions make read-only operations, both will successfully commit. The rest of the results are quite different from the other transactional model. Analysing the scenario where the transaction0 performs read operations and transaction1 write operations, we observe that unlike the pessimistic approach, the transaction that first accesses the object does not commit. In this

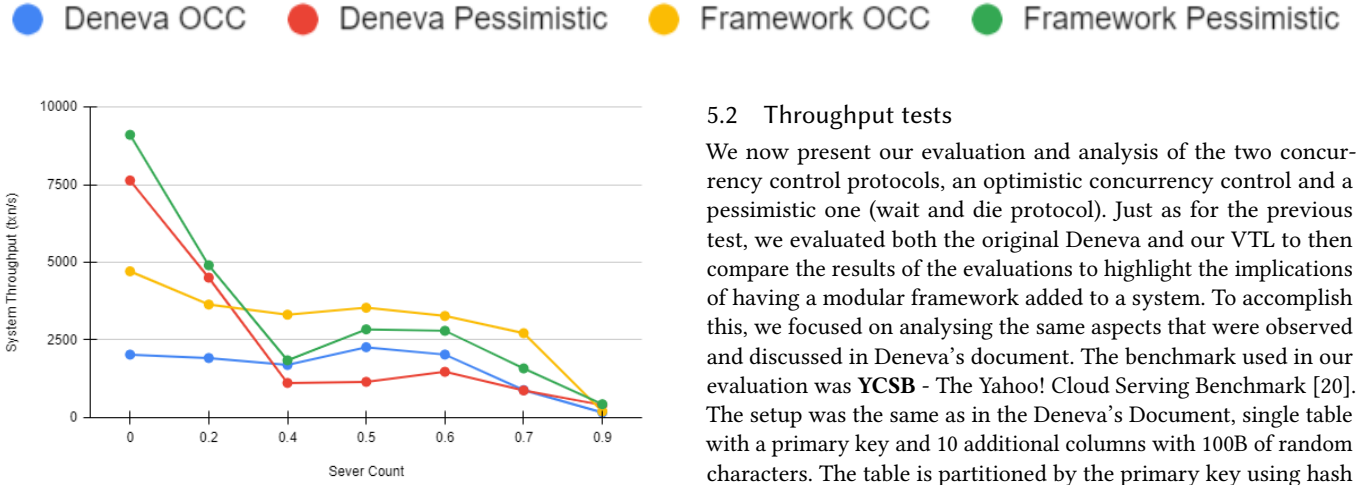


Fig. 5. **Contention** - Throughput measurements for each protocol varying the skew factor on 8 servers.

model, transactions make all the operations freely and validate the entire transaction at the end. In this case, transaction0 will not get validated due to another transaction modifying the value that it had accessed. However, the transaction that modified the value (transaction1), will validate successfully due to other transactions not conflicting with it. We can briefly say that, for this model, write operations cause conflict with other transactions accessing the same object, but read operations do not cause conflict. The same process happens for transaction0 writing and transaction1 reading, the write is allowed to commit, while the read is not. Lastly, if both transactions write the same objects, none of them will commit unlike the pessimistic model.

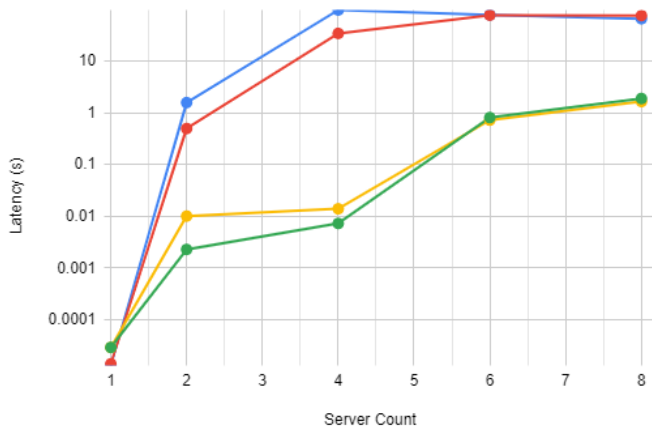


Fig. 6. **99th percentile Latency** - Latency from a transaction's first start to its final commit varying the cluster size.

5.2 Throughput tests

We now present our evaluation and analysis of the two concurrency control protocols, an optimistic concurrency control and a pessimistic one (wait and die protocol). Just as for the previous test, we evaluated both the original Deneva and our VTL to then compare the results of the evaluations to highlight the implications of having a modular framework added to a system. To accomplish this, we focused on analysing the same aspects that were observed and discussed in Deneva's document. The benchmark used in our evaluation was **YCSB** - The Yahoo! Cloud Serving Benchmark [20]. The setup was the same as in the Deneva's Document, single table with a primary key and 10 additional columns with 100B of random characters. The table is partitioned by the primary key using hash partitioning [19]. Each transaction will access ten records that can be either read or write operations that occur randomly, following a Zipfian distribution tuned by a skew parameter.

5.2.1 Contention: We first measured the influence of increasing the amount of contention in the system by tuning the skew parameter. Contention occurs when transactions make operations (read or write) to the same object. In this experiment, we configured the operations into 50% writes and 50% reads in a 8 servers (and 8 clients) setup. What is instantaneously noticeable by observing Figure 5 is that, the throughput clinically declines with the increment of the skew parameter. Unlike the original document, we can observe the decline sooner (0.2) than what they represented (0.6). At 0.0 contention, it is surprisingly that our solution shows a higher throughput in both algorithms, comparatively with their respective original protocols. This occurs due to the latency values which will be discussed further in the document. At low contention, both OCC protocols perform worse than the pessimistic protocols due to the overheads of validation [19]. However, once some the contention levels increase (at around 0.4) we can see that it overtakes the deterministic protocol. At 0.9, when the contention is highest, all of them reach their minimum throughput, as per expected.

5.2.2 Scalability: The previous experiment was mainly focused on varying the skew parameter and observing how the systems behaved regardless of the number of nodes. In contrast, we will now vary the number of nodes and have fixed workloads to evaluate how each protocol scale with more servers. We selected the same three scenarios as did Deneva: A read-only workload (0% of write operations) with no contention (skew = 0) - measures the maximum throughput for each protocols, given that there is no contention. A read-write workload (50% of write operations) with medium contention (skew = 0.6) and High contention (skew = 0.6).

No contention: By observing figure 7a, we can see that the protocols with the addition of VTL, have a big decrease in throughput in comparison with the original Deneva's protocols. We can also observe that, the throughputs are roughly similar independently of the protocol. This happens due to transactions only reading and not having to stop their execution since there is not contention.

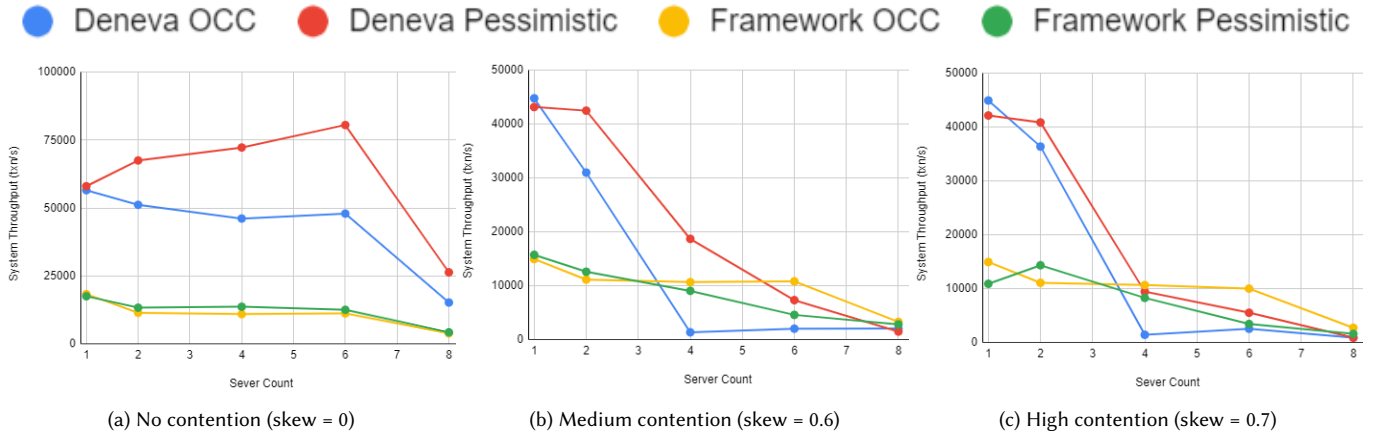


Fig. 7. **Scalability** - Throughput measurements for each protocol varying the workload and different cluster sizes.

The pessimistic approach having a slight bigger throughput is due to the OCC protocol having to copy items for reference during its validation phase.

If we compare these results, while having 8 nodes, with the ones from the previous graph (figure 5), where there is also no contention, we can clearly observe that they are not identical. This occurred due to, in figure 5, there was 50% of write operations, while here it was read-only transactions. If we compare them two, we can see a tremendously decrease of throughput just by adding writes to the system, without having any contention whatsoever. For example, OCC Read-Only has the throughput of 15k transactions per second while OCC Read-Write has 2k transactions per second.

Medium & High contention: Both the medium and high contention graphs look pretty similar (figures 7b and 7c). Just as it was seen in the previous graph, for only 1 server, we can see the overhead of the framework clearly. But, just like it was shown in the Deneva's document, the throughput of all the servers decrease with the increase of servers until a certain point. That point (around 16 servers) we could not showcase due to not having access to so many nodes in the GSD Cluster. One aspect that I should mention is that the framework OCC protocol does not have a sudden drop at 4 servers as does the original OCC protocol.

5.2.3 Latency: We will observe and compare the results of the latency of a transaction's start to its commit, between the framework protocols and original Deneva. We will focus on the 99th percentile, which means that 99% of the requests should be faster than the given latency (only 1% of the transactions can be slower).

In Figure 6, we observe that the 99th percentile latency increases with server size as expected. However, it is not expected to have a noticeable significant gap in latency between the protocols of the framework and the original Deneva. For example, for 4 servers, while the original OCC has a latency of 98s and original Pessimistic has 38s, the transactional models implemented by the framework have a latency of around 10 milliseconds. We realize that the following results might be incorrect. For the variable transactional layer we were expecting higher levels of latency than the original Deneva, to

help justify the low throughput of the previous examples. A reason that might explain these results is that, due to the low throughput of the framework, transactions take longer to finish. And since we are experimenting scenarios of 60s of warm up and 60s of testing, executing transactions that surpass this time period, will not have their statistics accounted for. With the distribution of the nodes and the partition of the database, for a transaction to commit, it needs to contact every node it accesses. We assume that this provokes the transactions to slow down that cause to surpass the experiment time. To test this hypothesis, we will do an experiment that only finishes once a certain amount of transactions are finished.

6 CONCLUSION

This research aimed at developing a framework that could represent a wide variety of distributed transactional systems. We researched a variety of distinct distributed transactional systems in order to determine all the common mandatory steps these systems have. Knowing what the majority of distributed transactional systems need, we built an architecture that allows to represent a big part of these systems. We chose to port our framework into a system that represents a wide range of transactional protocols, to try to showcase that we can represent an extent amount of protocols with our architecture.

We first demonstrate that our framework behaves exactly as the original Deneva in terms of functionalities, despite the split into different blocks. Secondly, we observed that, the addition of the variable transactional layer caused an overhead of around 60% in all centralized environments, regardless of the contention between transactions. This overhead is also observed, in a distributed environment, when there is no contention between transaction (Read-Only transactions). We also viewed that, when there exists contention, the increment of servers would result in drops of throughput due to the latency of a partitioned system. In the original Deneva's document, they showcase that by continuously incrementing the servers, the throughput would start to increase with the amount of servers. We seen that the framework appear to have basically no latency in comparison with original Deneva's protocols. We justify this by

the low throughput of the framework protocols. Meaning that, the transactions take so much time to commit that they surpass the duration of the test and are not accounted for. We are aware that 60% of throughput decrease is a lot and that it is unacceptable for commercial uses.

Since there is an abundance of different distributed transactional models, we believe that having a modular framework that can interchange between them is needed. Systems architects can simply use these existing components and swap between protocols, depending on what properties they desire at the moment. Otherwise, if you have the transactional model intertwined with the structure of the system, switching to a different protocol might just mean to switch to a complete different system. In context of academia, this is also useful, since most of the times when you are building a new transactional approach, you have to build everything from the ground up, when often you just want to change small parts of a whole. Although we just implemented two different transactional models, we trust that our architecture is able to represent much more protocols.

7 FUTURE WORK

For future work we have four main ideas. First, we think that is absolutely essential an optimization of the framework to increase the throughput levels.

Second, we unfortunately had to skip the implementation of transaction recovery which is absolutely essential for most distributed transactional systems. Therefore we think the addition of this feature would help represent a much bigger number of systems.

Third, since we only represented two different transactional models, a pessimistic and an optimistic one, we are interested in exploring more protocols such as MVCC [6], timestamped based concurrency control [7], and others.

And finally, we also think that it might be interesting to, depending on the workload being processed, change the transactional protocol being used at runtime to improve the throughput of the system.

ACKNOWLEDGMENTS

I would like to thank my supervisors for leading me in this path, as well as Instituto Superior Técnico for providing the necessary tools to evaluate our solution. I could not have done this without the continuous support of my parents, of my sister and especially my girlfriend, which i torment so much.

REFERENCES

- [1] S. Gilbert and N. Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. In: *ACM SIGACT News*, Jun. 2002
- [2] Ozsu, M., AND VALDURIEZ, P. Principles of Distributed Database Systems, Third Edition. pp. 335-363, 1999
- [3] ADYA, A. et al. Generalized isolation level definitions. In: *Proceedings of IEEE International Conference on Data Engineering*, IEEE Computer Society Press, Los Alamitos, CA, 2000.
- [4] ADYA, A. Weak consistency: A generalized theory and optimistic implementations of distributed transactions. In: PhD thesis, MIT, Cambridge, MA, Mar. 1999.
- [5] Jaffray, J. *What Does Write Skew Look Like?* In: <http://justinjaffray.com/what-does-write-skew-look-like/>, Mar. 2018 (visited on 08/01/2021)
- [6] Bernstein, P. A., and Goodman, N. Concurrency control in distributed database systems. In: *ACM Comput. Surv.* 13, 2, 1981.
- [7] Bernstein, P. and Goodman, N. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2), pp. 185-221, 1981.
- [8] Berenson, H. et al. A Critique of ANSI SQL Isolation Levels, In: *Proc. ACM SIGMOD 95*, San Jose CA, Jun. 1995.
- [9] B. Liskov and J. Cowling. Viewstamped replication revisited. In: Technical report, MIT CSAIL, Cambridge, MA, 2012.
- [10] Chang, F. et al. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 2008.
- [11] James C Corbett et al. Spanner: Google's Globally Distributed Database. In: *ACM Trans. Comput. Syst.*, 2012.
- [12] B. Ding et al. Centiman: elastic, high performance optimistic concurrency control by watermarking. In: *SoCC*, 2015
- [13] S. Wu et al. Towards elastic transactional cloud storage with range query support. In: *Int'l Conference on Very Large Data Bases (VLDB)*, 2010.
- [14] A. Thomson et al. Calvin: fast distributed transaction for partitioned database systems. In: *SIGMOD*, 2012.
- [15] J. Cowling and B. Liskov. Granola: low-overhead distributed transaction coordination. In: *USENIX ATC*, Jun. 2012.
- [16] Erlikh, L. Leveraging legacy system dollars for e-business. In: *IT Professional*, vol. 2, 2000
- [17] H. V. Jagadish et al. Baton: a balanced tree structure for peer-to-peer networks. In: *VLDB*, 2005
- [18] Hooda, P. *Comparison – Centralized, Decentralized and Distributed Systems* In: <https://www.geeksforgeeks.org/comparison-centralized-decentralized-and-distributed-systems/>, Apr, 2019
- [19] R. Harding, D. Van Aken, A. Pavlo, and M. Stonebraker An evaluation of distributed concurrency control. In: *Proc. VLDB Endow.*, 10(5), pp. 553–564, Jan, 2017.
- [20] B. Cooper et al, Benchmarking Cloud Serving Systems with YCSB. In: *ACM Symposium on Cloud Computing (SoCC)*, Indianapolis, Indiana, Jun, 2010.
- [21] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, et al CockroachDB: The Resilient Geo-Distributed SQL Database. In: *SIGMOD*, pp. 1493-1509. ACM, 2020.
- [22] Ivanov, N. Apache Ignite In-Memory Computing Platform. In: *InfoQ*. Retrieved Oct, 2017.
- [23] Justinmind In: <https://uxplanet.org/what-users-hate-most-about-your-app-according-to-google-c4a089ddfafa/>, Apr, 2018.
- [24] Gunne B. Design of Mimer SQL transaction handling system. 2001.
- [25] IDREOS, S., et al. Monetdb: Two decades of research in column-oriented database. In: *IEEE Data Engineering Bulletin*, 2012.
- [26] Crowe M. Transactions in the Pyrrho Database Engine. 2005.
- [27] Balakrishnan, M., Malkhi, et al. Tango: Distributed data structures over a shared log. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (pp. 325-340). Nov, 2013.
- [28] Mahmoud, Hatem A., Arora, V., et al. Maat: Effective and scalable coordination of distributed transactions in the cloud. In: *Proceedings of the VLDB Endowment*, 7(5), pp. 329-340, (2014).
- [29] Mu S., et al. Extracting more concurrency from distributed transactions. In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. pp. 479-494, 2014.
- [30] Zhang L., et al. Building consistent transactions with inconsistent replication. In: *ACM Transactions on Computer Systems (TOCS)*, 35.4: 1-37 2018.
- [31] Spear, M. F., Michael, M. M., & Von Praun, C. RingSTM: scalable transactions with a single atomic instruction. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures* (pp. 275-284), Jun. 2008.