A Transactional Key-Value store over YARN

BIG DATA | EPFL | 19 MAY 2015

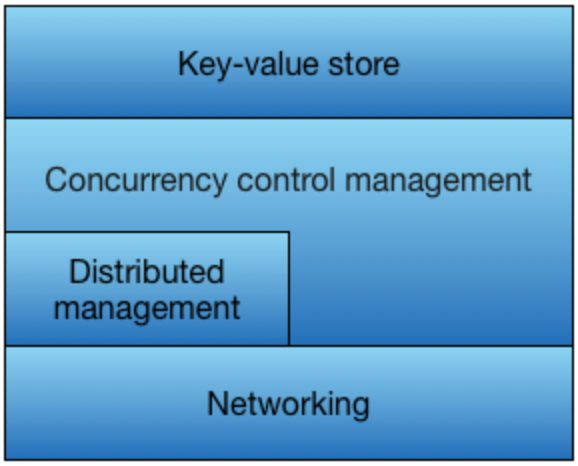
Andy Roulin
Egeyar Özlen Bagcioglu
Florian Depraz
Georgios Piskas
Philémon Favrod (Team leader)
Sachin Basil John

Table of Contents

Introduction

Key-value stores, among other NoSQL systems, generally favor eventual consistency to ACID properties. The main reason for this choice is the lack of scalability of strong concurrency control algorithms in large distributed systems. The classical example for this fact is the 2 communication rounds required by the 2 Phase Commit protocol that quickly becomes a nightmare for very large clusters. The goal of this project is to design a platform to evaluate concurrency control algorithms on a classical distributed NoSQL system, a key-value store. This is a first step in what could become an extended research topic.

One of the first requirements of such a system is flexibility. Software engineering theory - if such a thing exists - generally tells us that flexibility is the result of modularity and stratification. Avoiding coupling between the different functional layers was one of our first concerns. The figure illustrates the layered architecture of the resulting system. Let us briefly review them in a bottom-up manner. The networking layer is based on YARN to be compatible with cluster running Hadoop 2.0. It consists of message passing in JSON format. The concurrency control management layer includes a collection of algorithms that are known to ensure ACID properties - at the moment: **Strict 2PL**, **MVCC2PL** and **MVTO**. It is patched with a module ensuring distributed management of transactions and includes the protocols that are needed by the concurrency control algorithm layer to be executed in a distributed manner, e.g. **2PC**. The key-value store layer is a simple key value store that is kept in memory for simplicity since it was not the focus of this project.



The following sections describes each of these layers in greater details. The main goal of this report is to make the task of continuing the project easier. Therefore, we will keep the discussion at a high level and avoid being redundant with the code itself. Nonetheless, we will go into more details whenever deemed necessary. Also note that we assume that the reader knows the very basics of Yarn (ResourceManager, NodeManager, etc).

Network infrastructure

The network infrastructure consists of two main entities which form the skeleton of the application; the **AppMaster** (AM), a classical entity of Yarn applications, and the **TransactionManager** (TM). The AppMaster runs as the single point of access of the application while multiple instances of the TransactionManager run on different nodes. Another important entity is the client which comes in two flavors:

1. the **YarnClient** that is responsible for deploying the application on the cluster; and
2. the user clients comprising of applications that use the **UserTransaction** framework to submit transactions to be executed.

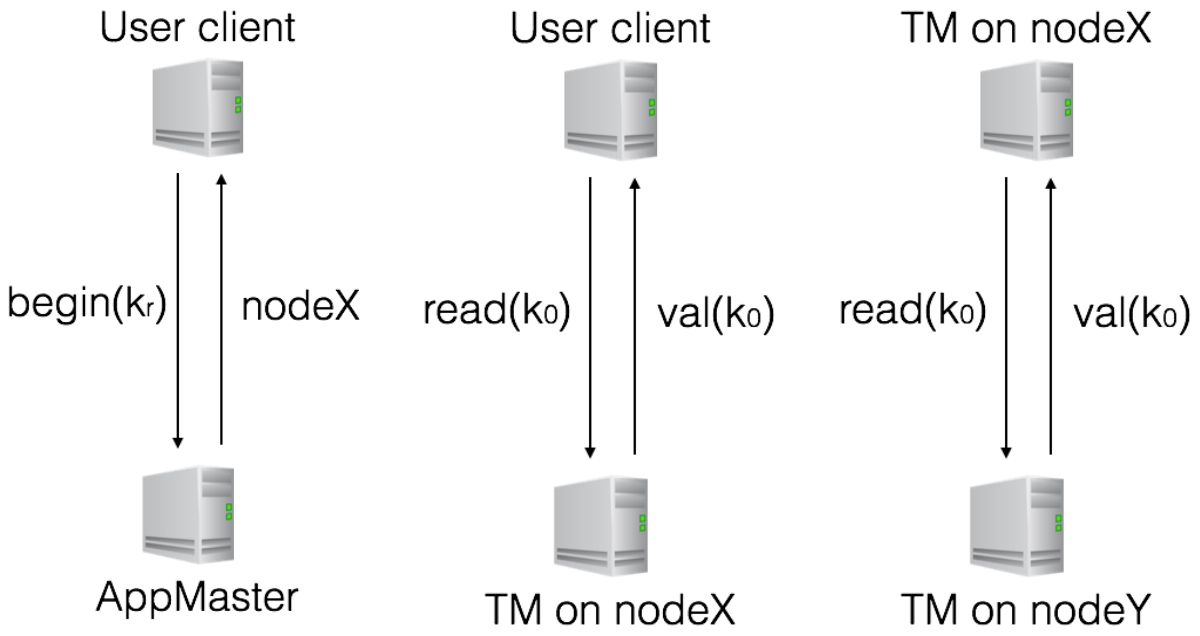
On startup, the AM is responsible for requesting containers from the Yarn ResourceManager that is running on the cluster. These containers are explicitly distributed on specific nodes[1] to enhance configurability. Once the ResourceManager is done allocating containers in the NodeManagers, the AM is responsible to start a TM daemon in each of them. From this point, we use the term TM to refer at the same time to the node as well as to the daemon that is running on them. As soon as TMs reply that they are ready, the AM starts acting as a server and listens for messages from users and TMs. On exit, it is responsible for gracefully stopping all active TMs if possible and killing those that cannot be gracefully stopped.

A TM is basically a server waiting for user requests and feeding them to a concurrency controller. Each TM executes and schedules transactions which are mapped to it by the AM. All TMs run the same instance of a concurrency control algorithm selected by the user before launching the system on the server[2].

[1] See the ./config/slaves file.

[2] The name of the algorithm (“simple_2pl”, “mvcc2pl” or “mvto”) should be written in the ./config/algorithm file for that purpose. Otherwise, MVTO will be selected by default.

The lifetime of a transaction in the cluster



Handshake
To begin a transaction, a user client informs the AM. Additionally, it gives a user-defined key that hints the AppMaster about the locality context of the transaction to be executed. The AM answers by sending the information needed by the user client to directly contact the TM that will handle the transaction.

Local operations
The next operations, such as read and write, that the user client will perform are first handled by a single TM which is referred to as **primary TM** that has been specified during the handshake. If the locality hint was accurate, most of the operations of the user client can be handled locally by the TM, i.e. the TM has the keys and value in its local memory and will be able to answer back as quickly as the concurrency controller allows.

TM-TM cooperation
In case the primary TM for that transaction cannot handle a request from the user client, a cooperation between TMs occurs. In other words, the primary TM communicates with another TM, referred to as a secondary TM for that transaction, on another node. The primary TM forwards the read or write request of a key to the secondary one that will answer back as quickly as its concurrency controller allows.

Concurrency control management

AppMaster and ICentralizedDecider

The AppMaster has two main purposes:

1. Users connect to AM to get a new transaction ID at the beginning of each transaction, as well as acquire the needed information regarding the TM which is responsible for this transaction.
2. It acts as an algorithm-specific central decider since it is aware of what happens on the cluster while the TMs are not.

The ICentralizedDecider is an interface for any functionality which requires a global view of the system. The AppMaster hosts a set of instances of central decider which handles algorithm-specific tasks that require such a view. One of the two centralized deciders currently implemented in the system is used to detect deadlocks. The other one determines the oldest live transaction in the system to have a garbage collection which is both safe and effective.

TransactionManager

Locking Unit

If the algorithm that is being used requires locks, each TM has access to an independent Locking Unit that it can use to lock keys that are stored locally. The Locking Unit is configurable to have different types of locks and internally supports local deadlock prevention. The TM initializes the Locking Unit by providing it with a lock compatibility table which specifies the lock types, and which pairs of locks are compatible.

Versioning Unit

Two different versioning units are used: one for **MVTO** and one for **MVCC2PL**. The **MVCC2PL** versioning unit is simpler as it needs to keep at most two versions per key and cannot cause a transaction to abort. Nevertheless, the two versioning units share similarities in the way they work. A version of a key is stored with a prefix which identifies the version in both cases. Therefore, to retrieve a particular key version, the versioning unit accesses the Key-Value Store with a new key that is the result of prefixing that key with the wanted version string.

Communication

Subclasses of `ch.epfl.tkvs.transactionmanager.communication.Message` are used for any communication between nodes. There are methods to convert these from and to JSON

format which is the actual format sent over the network. A TM, upon receiving a new message, spawns a new thread which processes the message and invokes the appropriate method of the concurrency controller. If the request can be performed locally, the concurrency controller does so in accordance with the algorithm. Otherwise, it is forwarded to the Remote Handler.

Any operation that requires co-operation between multiple TMs goes through the Remote Handler of the primary TM. Any read or write request for a key is forwarded to the TM which is responsible for that key. If not done before, a request to initialize this transaction on the other TM is sent before forwarding the actual request. The Remote Handler stores the secondary TMs associated with each transaction in order to avoid sending multiple begin requests to the same secondary TM for the same transaction. When the user wants to commit, the Remote Handler checks if the transaction can be prepared to commit in the primary TM and then sends prepare requests to all secondary TMs. If all such TMs are ready to commit, commit request is sent to all secondary TMs and the transaction is committed locally as well. At any time, if any operation results in an abort, the Remote Handler ensures that all TMs abort the transaction.

Algorithms

1. **MVTO** (uses transaction id as timestamp)
   The algorithm uses only an MVTO specialized versioning unit. The timestamp of each transaction is its unique transaction identifier and they write versions of keys with these timestamps. On read, the transaction reads the newest version that has an older or equal timestamp.
   It does the following for each of the methods.
   a. *begin* : creates a new transaction object and associates it with the given transaction id
   b. *read* : returns the value written by the same transaction if any, or the value of the version with the highest write timestamp which is smaller than the timestamp of the current transaction
   c. *write* : aborts if a transaction with higher timestamp has already read the value for the key, otherwise creates a new version of the key with the new value and the timestamp of that transaction.
   d. *prepare* : blocks if the transaction has read a “dirty” key, i.e. a version from a transaction that is still alive. If one of those read-from transactions aborts then this transaction also aborts. If all read-from transactions successfully commit, then the prepare method returns without exception, and sets the status of the transaction as prepared for commit.
   e. *commit* : removes the transaction from the list of active transactions

f. *abort* : removes the transaction from the list of active transactions and rolls back.
g. *checkpoint* : sends the list of active primary transactions to the centralized decider and performs garbage collection on versions based on the response.

2. Strict 2PL

The algorithm uses both a locking unit and a versioning unit. The versioning unit is used to only implement rollback in case of abort. If logging is implemented for recovery and rollback, this algorithm can interact directly with the key value store without using the versioning unit. The locking unit is initialized with read and write locks where only read locks are compatible with each other. If any operation results in deadlock, the transaction is aborted. In case of abort, all held locks are released before returning. It does the following for each of the methods.

a. *begin*: creates new transaction object and associates it with the given transaction id
b. *read*: obtains a read lock (if no read or write lock is already held) on the key and then returns the value given by versioning unit
c. *write*: obtains a write lock (if not already held) and write the new value using the versioning unit.
d. *prepare* : sets the status of the transaction as prepared for commit directly.
e. *commit*: asks the versioning unit to make the changes visible to other transactions and removes the transaction from the list of active transactions after releasing all its locks
f. *abort*: asks the versioning unit to rollback the changes, and removes the transaction from the list of active transactions after releasing all its locks
g. *checkpoint* : sends the local deadlock graph to the centralized decider.

3. MVCC2PL

The algorithm implemented is 2-version strict 2PL which uses both a locking and a versioning unit. The locking unit is initialized with read, write and commit locks where the commit lock is incompatible with anything and the write lock is incompatible with commit lock and itself, and every other pair is compatible. If any operation results in deadlock, the transaction is aborted. In case of abort, all held locks are first released before returning. It does the following for each of the methods.

a. *begin* : creates a new transaction object and associates it with the given transaction id
b. *read* : obtains a read lock (if not already held) on the key and then returns the value given by versioning unit
c. *write* : obtains a write lock (if not already held) and write the new value using the versioning unit.
d. *prepare* : obtains commit lock on all keys that were written and sets the status of the transaction as prepared for commit.

e. *commit* : asks the versioning unit to make the changes visible to other transactions and removes the transaction from the list of active transactions after releasing all its locks
f. *abort* : asks the versioning unit to rollback the changes, and removes the transaction from the list of active transactions after releasing all its locks
g. *checkpoint* : sends the local deadlock graph to the centralized decider.

Apart from what is mentioned above, all algorithms respond with failure in case of invalid state such as when transaction is not live, or the value does not exist for a read, or when another transaction exists with the same id for a begin, etc.

Client

Other than starting the application, Client is also responsible for running the read-evaluate-process-loop (REPL). Therefore, all the commands for REPL are in Client.
The *:help* command can be used to get a list of available commands.

User

The user has to define a new class which extends *Key* class that represents a key for the transactions. A transaction can then be performed by creating a new *UserTransaction* object specifying a key to hint the system about locality, and then invoking read, write, commit or abort methods on this object. An *AbortException* is thrown in case of failure, and it is up to the user to restart the transaction, if required.

Testing Framework

The project contains *UserClientScheduledTest* class which exports a test framework. This framework is well suited for executing a schedule of transactions and checking the expected result while taking care of scheduling, concurrency, blocking operations, deadlocks and other difficulties. In this framework, a transaction is represented by an array of *TransactionExecutionCommands* and a schedule is defined to be an array of transactions. Please refer to the Javadoc for detailed description of each command. The schedule is executed by passing it as an argument to *execute* method which also takes as an optional argument the list of keys to be initialized to the default value “00”.

Results

The system is not mature enough to consider the following results as true empirical facts about distributed concurrency control. Nevertheless, they help to get an idea of what has been done and, more importantly, to guide future work.

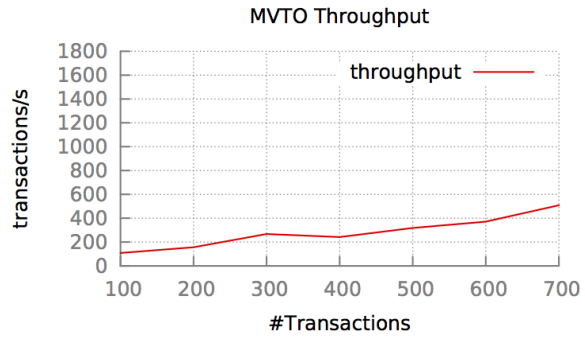
A benchmarking suite is also provided. It performs randomized benchmarks of the system given the following arguments:

- the number of concurrent transactions,
- their maximum number of operation per transaction,
- the read/write ratio,
- the locality percentage (what percentage of all the operations are handled by the transaction's primary TM), and
- the number of trials per experience.

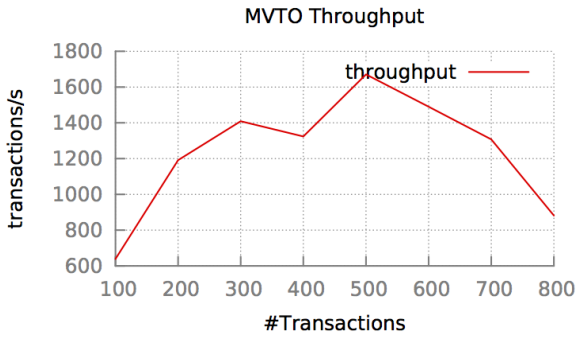
We tried to diminish the impact of central decision process as much as possible (see *AppMaster and ICentralizedDecider*). Since it is based on periodical checkpoint, it can be an important bias and introduces irregularities in our measurements.

The current benchmark suite only allows to benchmark from one user client at a time. This might however introduce some bias into the results as contention might occur at the client side.

Locality



Throughput of MVTO with a low locality percentage, i.e. 30%, given an increasing number of transactions.



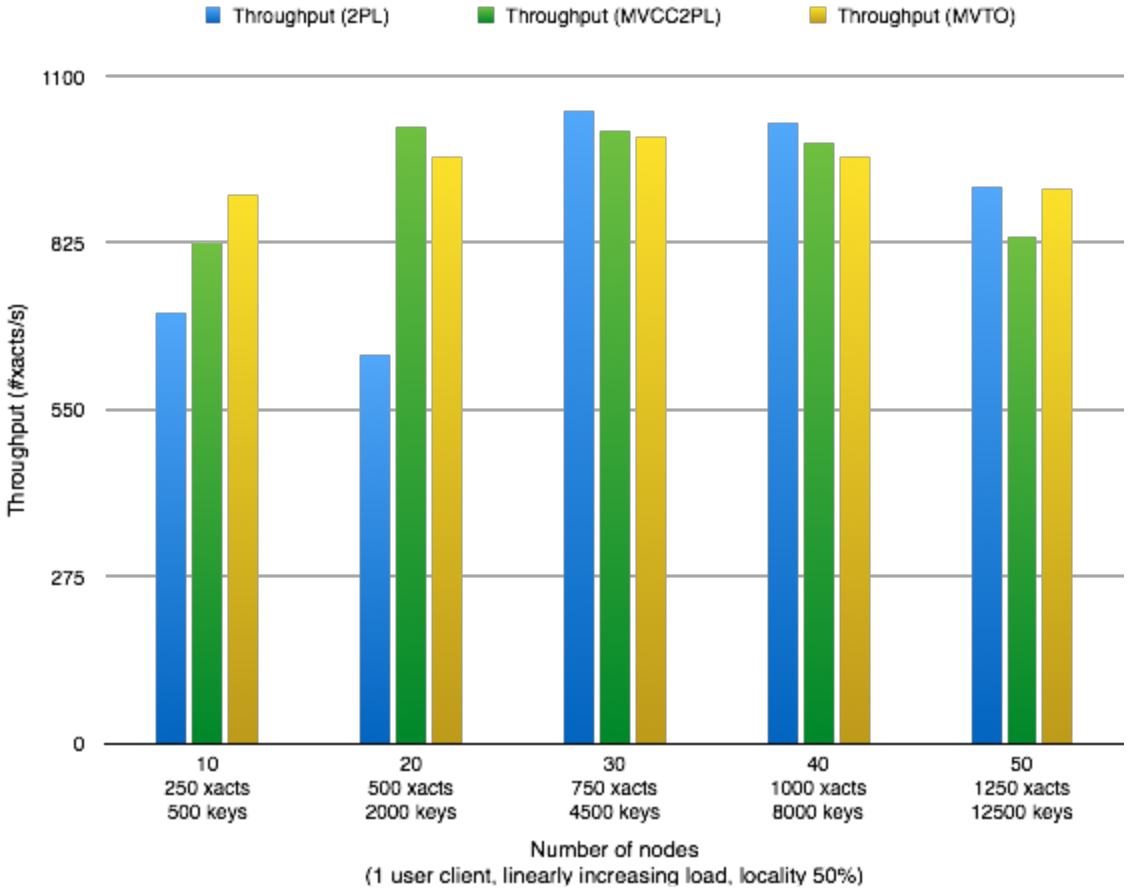
Throughput of MVTO with a high locality percentage, i.e. 80%, given an increasing number of transactions.

The design of the system relies on the locality principle. Whenever possible, it assumes that the user is able to hint the system with a good locality indicator so that most operations can be handled by a single TM. Validating that the system performs better when properly hinted

is therefore a primary concern. The two graphs depict the average transaction throughput with a given load (20 nodes, 30 reads per write, max. 10 requests per transactions). The graph on the left shows the case where 30% of operations are local while the one on the right shows the case where 80% are local. This results in a throughput up to 8 times higher compared to the poorly hinted case.

The decrease in the load observed at the end of the latter is supposed to be due to contention as outlined before. The system has reached its saturation point in the presence of a single user client, and pushing it further decreases the overall performance.

Throughput



The graph above shows the average number of concurrent transactions that commit successfully per second for a given number of nodes on which they are run. The workload is increased proportionally with the number of nodes. All transactions are launched by a single user client but are executed concurrently. To fairly compare the throughput of the concurrency control algorithms, the tests have been executed while making sure that no deadlock occurs in **MVCC2PL** nor in **2PL**. Garbage collection is also disabled in the case of **MVTO** to avoid performance irregularities.

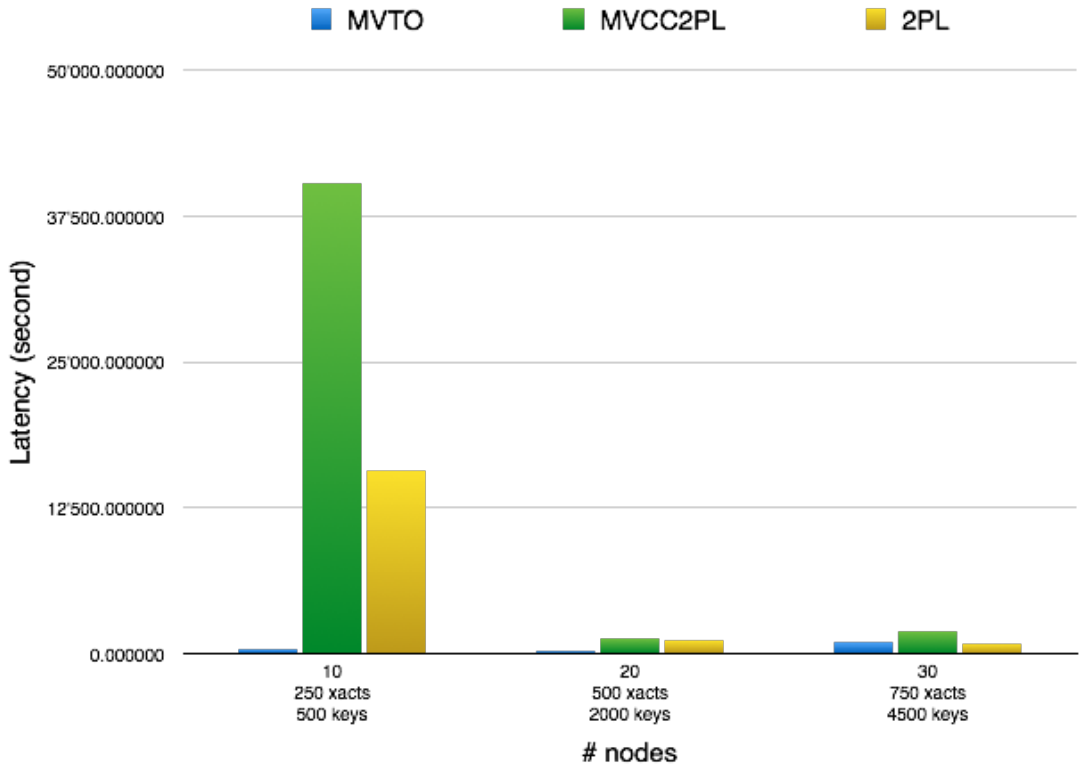
The two observations that we make regarding this graph are the following:

1. The average throughput seems to be above 800 transactions per second and remains pretty constant. Other cases seem to happen due to the lack of intrinsic parallelism, e.g. not having enough transactions.
2. The performance of the system seems to saturate near 1000 transactions per seconds.

Once again, the saturation might be because of the single-user-client problem: the number of messages increases and creates contention at the client-side. Testing it with multiple user clients might be an interesting next step.

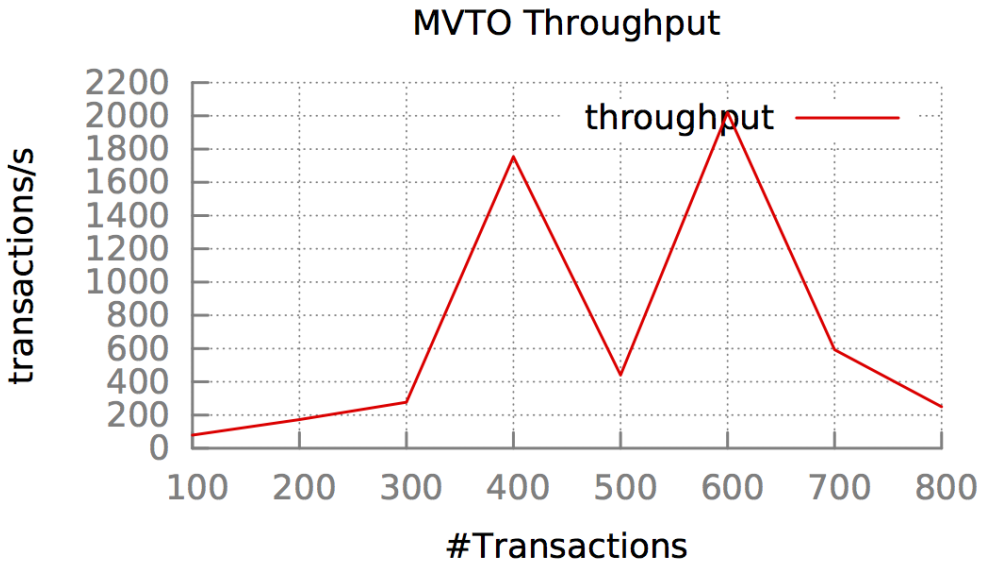
Background process irregularities

The impact of deadlock prevention on latency



The graph above shows the result of a benchmark of the latency in a conflicting environment where deadlocks could occur in the concurrency control algorithms that makes use of locking (2PL and MVCC2PL). This shows the unpredictable nature of the latency of such algorithms in our system. This is due to the fixed checkpointing deadlock detection system. Here, deadlocks are checked every 15 seconds. There exists a trade-off: increasing the checkpointing frequency diminishes the performance due to network contention while decreasing it creates irregularities. A future work might be to adjust the frequency dynamically as a function of the number of recent deadlocks detected. In this manner, only the first deadlocks would have an impact while the subsequent ones would have a diminished impact. Hopefully, this might slightly smooth the unpredictable spikes. As expected, one can see that MVTO has the best latency among all considered algorithms.

The impact of garbage collection on throughput



Lock-free concurrency control algorithms are not free from fluctuations. Whereas deadlocks do not occur in MVTO, garbage collection might be needed. The above graph shows the same benchmark as the one depicted before in the section about locality. However, this time, the garbage collection is enabled. The garbage collection overhead is significant because each TM needs to know which transactions are still alive. While doing garbage collection, it is important to ensure that versions are not removed prematurely. Removing a version when no transaction with a lower timestamp is active in that TM is incorrect. Even if a transaction was never active in a particular TM, it might come in the future to read a version as long as it is active in the primary TM. When garbage collection is triggered, all TMs contact a centralized service that knows about all live transactions and this introduces the overhead that can be seen as the valleys in the above graph.