Enforcing Policy and Data Consistency of Cloud Transactions: A Simulation

Tucker Trainor
Department of Computer Science, University of Pittsburgh
tmt33@pitt.edu

April 30, 2012

Abstract

With an increase in services offering data storage in the cloud comes an increase in scrutiny over the security of that data. Narrowing our focus on distributed transactional database systems deployed over cloud servers, we can look at how distribution of user authorization policies affects the trustworthiness of transactions. If policy versions are in an inconsistent state across the cloud servers utilized during that transaction, how can we be sure that the transaction is trusted? A proposed solution lies in the use of Two-Phase Validation protocols [1], a modified version of established Two-Phase Commit protocols. We can simulate the protocol's deferred proofs of authorization to quantify the cost of implementing the protocol in terms of time cost and commit success rate.

1. Model

To model the simulation, we begin by examining Two-Phase Validation (2PV) and deferred proofs of authorization and determine what we will require to accurately produce a scenario in which their performance can be measured. Deferred proofs of authorization define a situation where policy validation occurs at commit time, similar in spirit to deferred integrity checks on a database. Over the lifetime of a transaction, multiple servers are likely to be utilized to complete the transaction. Once a server is called upon to assist in completion of the transaction, it stores its most recent local (to the server) policy version for that transaction. The transaction's stored policy version is untouched until a commit is requested for the transaction. At this point there are two levels of policy consistency that can be enforced to determine if the transaction is to be considered trustworthy: view consistency and global consistency.

For view consistency to be true, each server which took part in the transaction must have the same stored policy, i.e., the policy versions should be internally consistent across

all servers involved in the transaction [1]. In 2PV, we use a collection phase to gather the policy versions from the servers, and a validation phase to determine whether or not the servers are in agreement. In our model, if the policy versions are not consistent amongst all servers, then the transaction is aborted. If the view is consistent, then the transaction is allowed to commit. View consistency is quick in terms of checking validity but suffers from an inherent weakness in terms of policy freshness. Once the policy used for validation is set at the beginning of each server's role in the transaction, any updates to policy versions that occur after that point are not taken into account. As is common in the realm of computer science, speed is gained at the cost of security.

Global consistency involves a more stringent approach to validation to eliminate the weakness noted above for view consistency. At the validation phase, the policy versions collected from each server are compared to the latest policy version globally available from the originator of policies. Using the latest policy for validation purposes is an improvement of the trust level that view consistency provides. Additionally, instead of aborting the transaction in the case of any server's policy not matching the latest policy version, we use the latest version to execute a new local authorization check on each operation performed during the transaction that used an earlier policy for the local authorization check during the initial run of the transaction. As each server records all operations, it can execute local authorizations from its records without repeating the entire transaction. If all operations in the transaction are locally authorized by the latest policy version, then the transaction is allowed to commit. Otherwise, the transaction is aborted. Due to the possibility of running a series of local authorizations again at commit time, global consistency is likely to be more costly in terms of time when compared to view consistency. However, by offering an alternative to an abort after server policy inconsistency, successful commits may improve under certain conditions.

We can also create a protocol that combines the strengths of both view consistency and global consistency. Called view consistency with second chance global consistency, the protocol allows a transaction to check for view consistency at first, and if that fails then it can invoke a global consistency check to possibly save the transaction from aborting. Thus, the transaction has the ability to quickly be validated for commit via view consistency, but is possibly saved from an initial abort by a global consistency check. We suspect that time cost may increase when transactions are subject to both consistency checks, but commit success rates will improve over view consistency checks only.

In order to benchmark these protocols we can simulate two-phase commit (2PC) and two-phase commit with local authorization checks (2PCLA). Both the view consistency and the global consistency protocols simulate a check of each operation in a transaction for local authorization (i.e., verify that the client has permission to perform the operation). 2PC will operate similar to view consistency but will omit all local authorization checks and the final view consistency check. In essence, it forgoes all policy validation. 2PCLA adds the per operation local authorization checks back in, but still omits the final view consistency check. These two baseline measures, 2PC and 2PCLA, will be the base cases

on which we will measure the performance of the 2PV protocols.

To benchmark these protocols, we will record the interaction between experimental clients and servers as they process sample transactions. A robot-controlled client will send transactions to one of several cloud data servers. This primary server will serve as a transaction manager for the entire transaction, routing operations requiring other cloud data servers to those servers as necessary. The transaction manager is also responsible for collection and validation phases of any 2PC and 2PV protocol invocations. There will also be a policy server which periodically updates the master policy version and pushes it to all available cloud servers. The policy server also will field requests from cloud servers for the most recent policy version.

2. Experimental Testbed

Design and Implementation

In designing the simulation, we had two major requirements: communication over a network and the ability to spawn threads. Since we recently were given access to Java source code of a simple client/server application that utilized threading, we went with Java over other equally suitable languages such as C or C++. Coding in Java, we implemented a simulation with three major classes (`Robot`, `CloudServer`, `PolicyServer`), each with a number of supporting classes.

The `Robot` class is the main driver of the program. It accepts the bulk of the test parameters and generates transactions to be handled by the `CloudServer`s. The `Robot` reads from a parameters file and generates random transactions of a varying number of database READ or WRITE operations. Each transaction is passed to a spawned thread of the `RobotThread` class. The `RobotThread` initiates a socket connection with the `CloudServer` that is designated as the transaction manager (TM). Communications between clients and servers are in the form of the `Message` class, a simple serializable class containing a single String variable. As acknowledgements of transaction processing are returned from the TM, the `RobotThread` adds data to a `TransactionLog`, which is a list of `TransactionData` items. As `RobotThread`s are completed, new `RobotThread`s are spawned, maintaining a maximum number of concurrent threads as specified in the parameters file and tracked in the `ThreadCounter` class. Once all transactions are complete, the `TransactionLog` is finalized and written to disk and the `Robot` exits.

The `Robot` requires command line input for the simulation to begin. At minimum it requires its first argument as the identifying number of the `CloudServer` that will be designated as the TM. Additional arguments can be added to override parameters set in the parameters file, which is useful if running shell scripts for multiple runs of simulations. The second argument is reserved for setting a seed value for the `Robot`'s random number generator. The third and fourth arguments are reserved for setting the minimum and maximum number of operations in a transaction, while the fifth argument is reserved

for setting the policy verification protocol (e.g., 2PC, view consistency, etc.). The third through fifth arguments, if set, are sent by the `Robot` to all available `CloudServer` instances before transaction processing begins.

The `CloudServer` class and its helper classes perform the bulk of the work in the simulation. The `CloudServer` class itself is mainly used for loading server parameters and listening for connections from `RobotThread`s. These connections are handed off to the `WorkerThread` class, which implements the protocols and returns progress data to its associated `RobotThread`. If the `WorkerThread` is passed an operation that needs to be processed on another server, it opens a new connection with that server if one has not been created already. Each new socket connection created by a `WorkerThread` is stored in that `WorkerThread`'s own instance of a `SocketGroup`. If any subsequent communications with a server that has had its connection stored in the `SocketGroup` can easily be reestablished by retrieving the stored socket from the `SocketGroup`. As a `WorkerThread` processes operations, it stores information about each one in a list of `OperationRecord`s. The list is necessary when performing global consistency checks, as the policy version used for local authorization is one of the data items stored in an `OperationRecord`.

Launching a `CloudServer` requires a single command line argument of an integer. The integer is a unique identifier for the server, as it refers to the IP address and port number for that server which the `Robot` and other servers will establish a connection with. The integers and their corresponding IP addresses and ports are stored in a configuration file named `serverConfig.txt` which is read in by all major classes.

The `PolicyServer` class and its associated classes are responsible for propagating policy version updates. The `PolicyServer` class itself is has two duties; one is accepting and responding to `CloudServer` requests for current policy versions, the other is launching the `PolicyUpdater` class that updates the policy version in a frequency passed by the parameters file. When a new policy version is issued and stored in the public `PolicyVersion` class, the `PolicyUpdater` spawns `PolicyThread`s to push the policy version to all available `CloudServer`s. Requests for the current policy version are handled by `PolicyRequestThread`s.

Parameters

The parameters used in the simulation are stored in a text file named `parameters.txt` which is read by all three major classes. The following are the variables that are set from the parameters file. Certain variables can be set by the `Robot` through command line input and override the values set by the parameters file; these variables are noted below.

- `maxTransactions` is the total number of transactions to be run in a simulation.
- `minOperations` and `maxOperations` are the minimum and maximum number of operations to be performed in a single transaction. These parameters can be set by the `Robot` via the command line. Short transactions are defined as 8-15 operations,

medium as 16-30 operations, and long as 31-50 operations. These parameters can be set by the `Robot` via the command line.

- `maxServers` is the total number of `CloudServer` instances to be created for the simulation.
- `maxDegree` is the degree of parallelism, or the maximum number of concurrent `RobotThread`s available to process transactions.
- `latencyMin` and `latencyMax` are the minimum and maximum amount of simulated delay in milliseconds caused by network latency. LAN latency is defined as 5-25ms in our testbed. These variables are used by all three major classes.
- `verificationType` is an integer value representing which protocol to use (e.g., 2PC, view consistency, etc.) at commit time. This variable is used by all `CloudServer` instances and is sent over the network by the `Robot` at the beginning of the simulation. This parameter can be set by the `Robot` via the command line.
- `integrityCheckSuccessRate` is a decimal value ranging from 0.0 to 1.0 for the rate at which a commit's integrity check is successful.
- `localAuthorizationSuccessRate` is a decimal value ranging from 0.0 to 1.0 for the rate at which an operation's authority to perform an action is allowed.
- `policyUpdateMin` and `policyUpdateMax` are integer values representing milliseconds between policy updates by the `PolicyServer`.
- `randomSeed` is a Long-type integer used to seed the `Robot`'s random number generator. This parameter can be set by the `Robot` via the command line.

Sample Transaction Flow

A typical transaction begins in the `Robot`, where a series of random operations are assembled and and passed to a spawned `RobotThread`. The `RobotThread` connects with the TM, which passes it to a `WorkerThread`. The `RobotThread` then begins sending the `WorkerThread` operations one at a time, in order. The `WorkerThread` will then determine if the operation is to be processed on its own server or on another server; if it is the latter, the `WorkerThread` passes the operation to the appropriate server and waits for the results of the operation. The `WorkerThread` handles READ and WRITE operations by performing a check of local authorization, which is simulated using the `localAuthorizationSuccessRate` for probability of granting authorization; a denial of authorization forces the transaction to ABORT. When the `WorkerThread` reaches a COMMIT operation, it simulates an integrity check against the `integrityCheckSuccessRate`, which if successful proceeds to the 2PV protocol as specified in the parameters.

If the transaction is to use validate using view consistency, after the integrity check the `WorkerThread` collects the policy version that each server involved in the transaction used for the lifetime of the transaction, and compares all that are collected to the current policy version on the `CloudServer` that hosts the TM. If any of the collected versions differ from the current version from the TM's server, the transaction is aborted. Otherwise, the transaction is allowed to commit and is complete.

If the transaction is to validate using global consistency, after the integrity check the `WorkerThread` has its TM request the current policy version from the `PolicyServer`. It then iterates through its `OperationRecord`s and if the operation was authorized with a policy version other than the current version, it runs a local authorization again for that operation with the current version. If all of the operations validate under the current policy version, the transaction is allowed to commit and is complete. Otherwise, the transaction is aborted at the first local authorization failure.

Finally, if the transaction is to validate using view consistency with second chance global consistency, it simple performs a view consistency check. If the view consistency check passes, then the transaction is allowed to commit and is complete. If the view consistency check fails, the `WorkerThread` then runs a global consistency check, with the same outcomes as the previous paragraph.

3. Experiments

With our testbed implemented, we carried out a series of experiments to benchmark the costs of the 2PV protocols. For our initial experiments we chose to determine transaction cost in terms of time, successful commit ratio, and throughput of committed transactions as a function of policy update frequency.

To establish a series of policy update frequencies, we began with an initial amount of time representing an average case of eight operations. Given a range of 75ms to 125ms for a READ operation and a range of 150ms to 225ms for a WRITE operation, we calculated the average duration of four READs and four WRITEs to be 1,150ms. We then ran experiments using 1,150ms as the policy update frequency. From there we continuously doubled the value, running experiments using the value as the policy update frequency, up through 36,800ms.

For each policy update frequency, we varied transaction length and validation protocol and ran experiments for every combination between the two. The variables were set as follows:

- Transaction lengths were divided into three ranges: short (8-15 operations per transaction), medium (16-30 operations per transaction), and long (31-50 operations per transaction).
- Five validation protocols were used: 2PC, 2PC with local authorization checks, view

consistency, global consistency, and view consistency with second chance global consistency.

State the fixed parameters, the varied parameters
explain all parameter settings

A. Performance evaluation

Transaction Cost

We represent transaction cost as the time in milliseconds that a transaction requires for completion. We performed simulations of all protocols for short transactions (see fig. 1), medium transactions (see fig. 2), and long transactions (see fig. 3). We began with a policy update frequency of 1,150ms and doubled the frequency for each subsequent set of runs up to 36,800ms. For each simulation we averaged the duration of each successfully committed transaction to provide a value of cost for view consistency, global consistency, and view consistency with second chance global consistency.

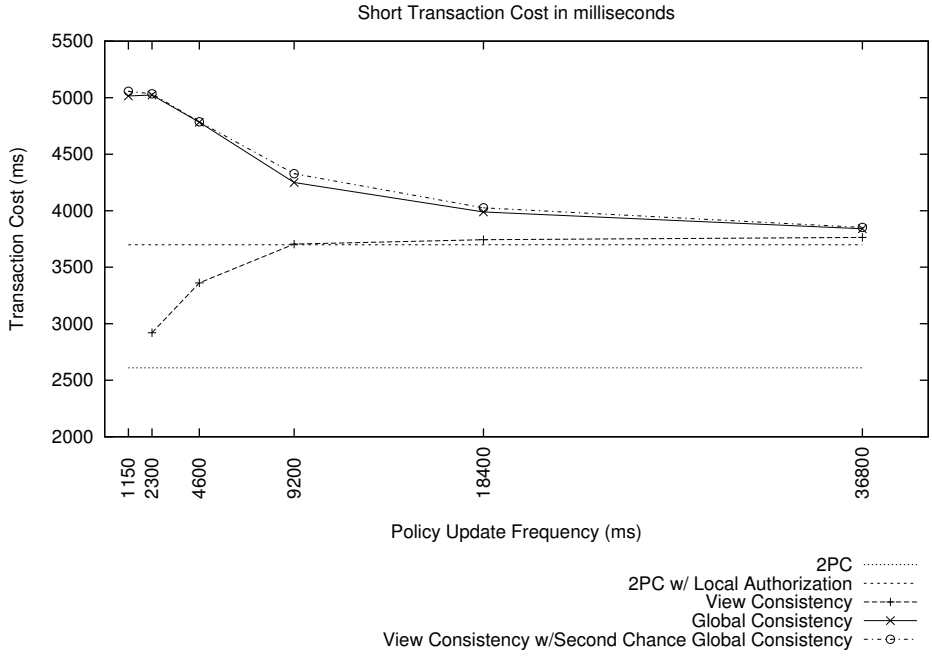

Figure 1: Short Transaction Cost

For short length transactions we observed a 2PC baseline cost of 2,610ms, rounded to the nearest integer. Using 2PC with a 99.5% local authorization success rate (2PCLA)

baseline, we observed a cost of 3,698 ms, rounded to the nearest integer. Figure 1 shows the results of the 2PV protocols against the baseline values. We observe view consistency perform no successful commits at 1,150ms but performing successful commits at 2,300ms and above. View consistency begins with a cost of 2,920ms, approximately 12% higher than the 2PC baseline and approximately 21% lower than the 2PCLA baseline. By the 9,200ms frequency, view consistency nearly equals the 2PCLA baseline (3,705 ms versus 3,698 ms) and only rises slightly for the remaining frequency points. Global consistency and view consistency with second chance show parallel performance throughout the graph, beginning at slightly more than 5,000ms (about 35% higher than the 2PCLA baseline) and closely approaching the 2PCLA baseline by the 36,800ms update frequency. At that frequency point, all three implementations are less then 5% above the 2PCLA baseline.

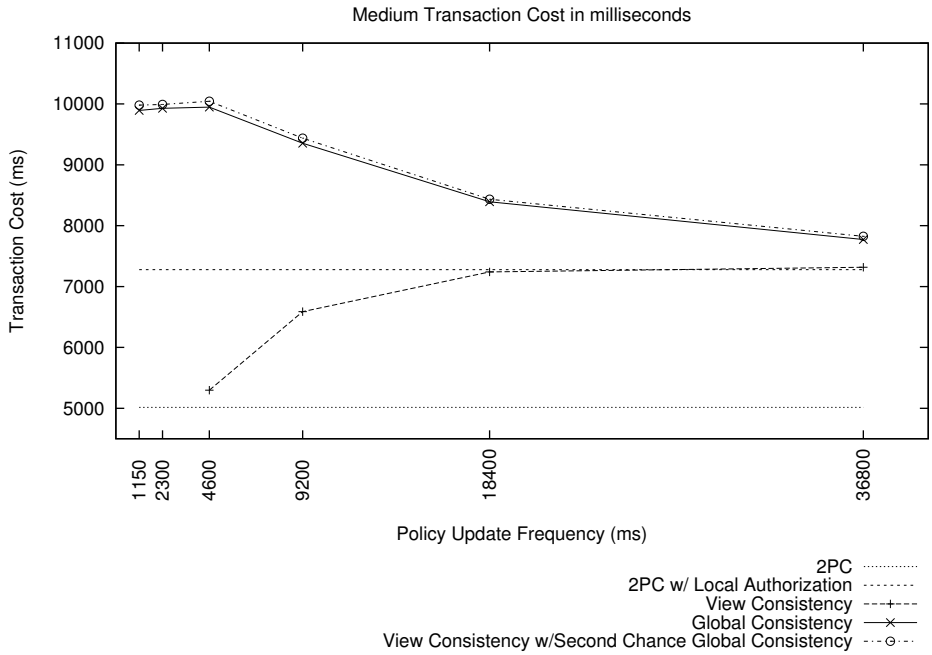

Figure 2: Medium Transaction Cost

For medium length transactions we observed a 2PC baseline cost of 5,015 ms, rounded to the nearest integer. For the 2PCLA baseline, we observed a cost of 7,279 ms, rounded to the nearest integer. Figure 2 shows the results of the 2PV protocols against the baseline values. View consistency first began performing successful commits at the 4,600ms update frequency, beginning with a cost of 5,297 ms, 6% higher than the 2PC baseline and 27% lower than the 2PCLA baseline. By the 18,400ms and 36,800ms marks, view consistency is within $\pm$40ms of the 2PCLA baseline. Global consistency and view consistency with

second chance again exhibit similar performance, with both beginning at the 1,150ms update frequency roughly 37% higher than the 2PCLA baseline, rising slightly at the 4,600ms update frequency, and then trending back towards the 2PCLA baseline, reaching a cost of about 7% of the 2PCLA baseline at the 36,800ms update frequency.

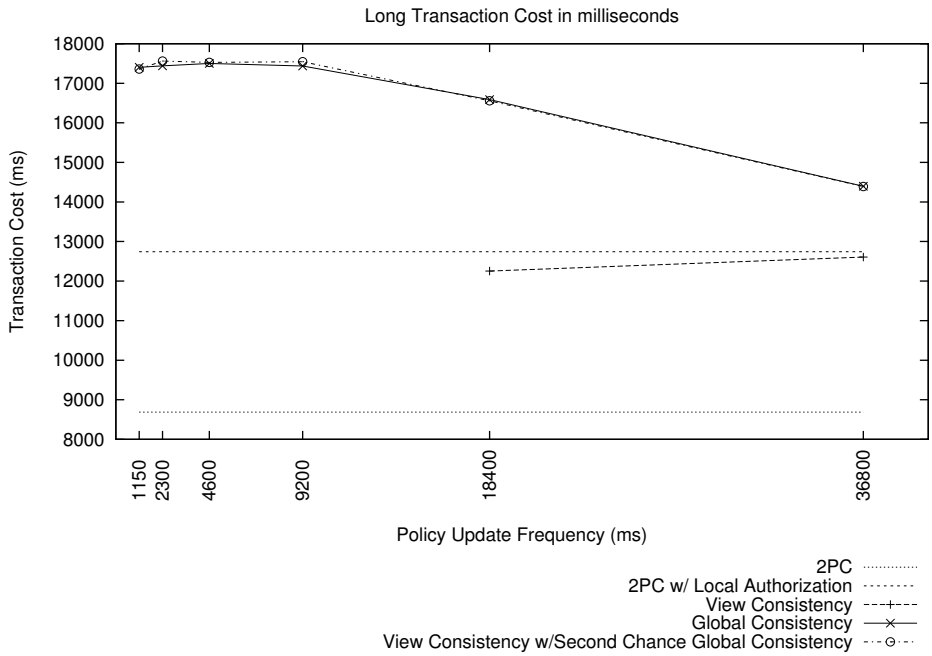

Figure 3: Long Transaction Cost

For long length transactions we observed a 2PC baseline cost of 8,688 ms, rounded to the nearest integer. For the 2PCLA baseline, we observed a cost of 12,742 ms, rounded to the nearest integer. Figure 3 shows the results of the 2PV protocols against the baseline values. View consistency increasingly shows negligible performance in the shorter update frequencies, only beginning to register successful commits at the 18,400ms mark. However, the performance does maintain comparable performance to the 2PCLA baseline, costing about 4% less at the 18,400ms mark and about 1% less at the 36,800ms mark. Global consistency and view consistency with second chance are again roughly equivalent in performance, with similarly steady results from the 1,150ms mark to the 9,200ms mark ranging between 36% and 38% higher than 2PCLA. The performance begins to trend towards the 2PLCA baseline at the 18,400ms and 36,800ms marks, reducing in cost to nearly 13% above the 2PCLA baseline.

Successful Commit Ratio

We represent the commit success ratio as the number of commits divided by the total number of transactions attempted. We performed simulations of all protocols for short transactions (see fig. 4), medium transactions (see fig. 5), and long transactions operations each (see fig. 6). We began with a policy update frequency of 1,150ms and doubled the amount for each subsequent set of runs up to 36,800ms.

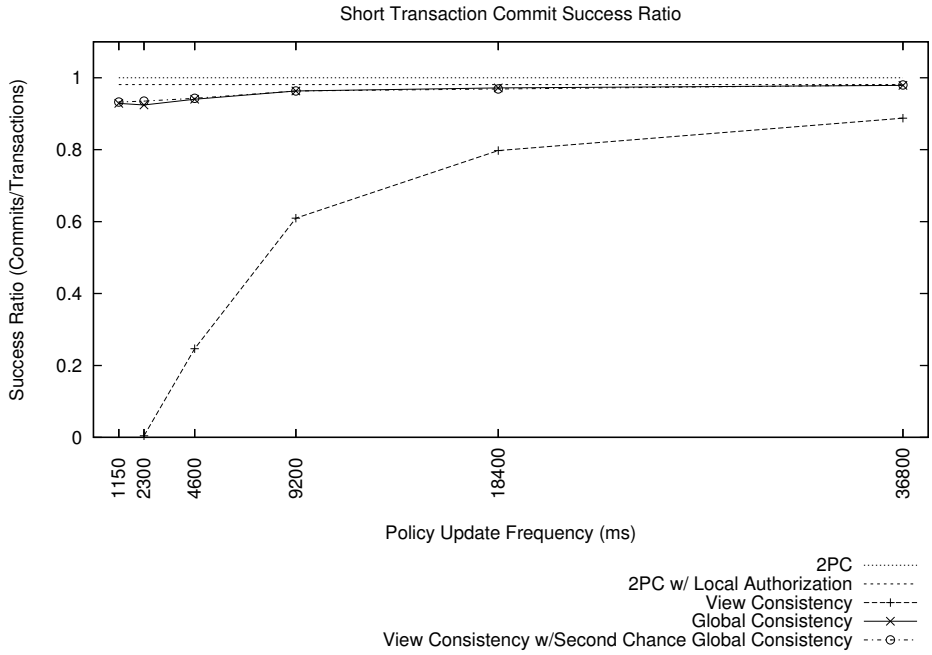

Figure 4: Short Transaction Commit Success Ratio

For short length transactions we observed a 2PC baseline success ratio of 1.0, or 100%. The perfect commit success was expected from our simulation setting for integrity constraint success of 1.0, which for 2PC solely determines the commit success rate. For the 2PCLA baseline, we observed a baseline success ratio of 0.981, or 98.1%. As 2PCLA is subject to transaction aborts due to the 99.5% local authorization success rate, we expected a small drop in successful commits from 100%. Figure 4 shows the results of the 2PV protocols against the baseline values. View consistency had no commits at the 1,150ms frequency update mark, and had only six commits out of three runs of 1,000 transactions for the 2,300ms mark, resulting in a success ratio of 0.43%. The view consistency success ratio improves as policy update frequency is lengthened, resulting in an 88.77% commit success ratio by the 36,800ms mark, roughly 10% below that of 2PCLA. Global consistency and

view consistency with second chance have nearly equal performance characteristics similar to what was observed in the transaction cost trials. Both protocols perform strongly in terms of commit success ratios, beginning at the 1,150ms mark with 93% ±0.2% success rates and improving to 97.9% ±0.1% success by the 36,800ms mark, nearly equal to the 2PCLA baseline of 98.1%.

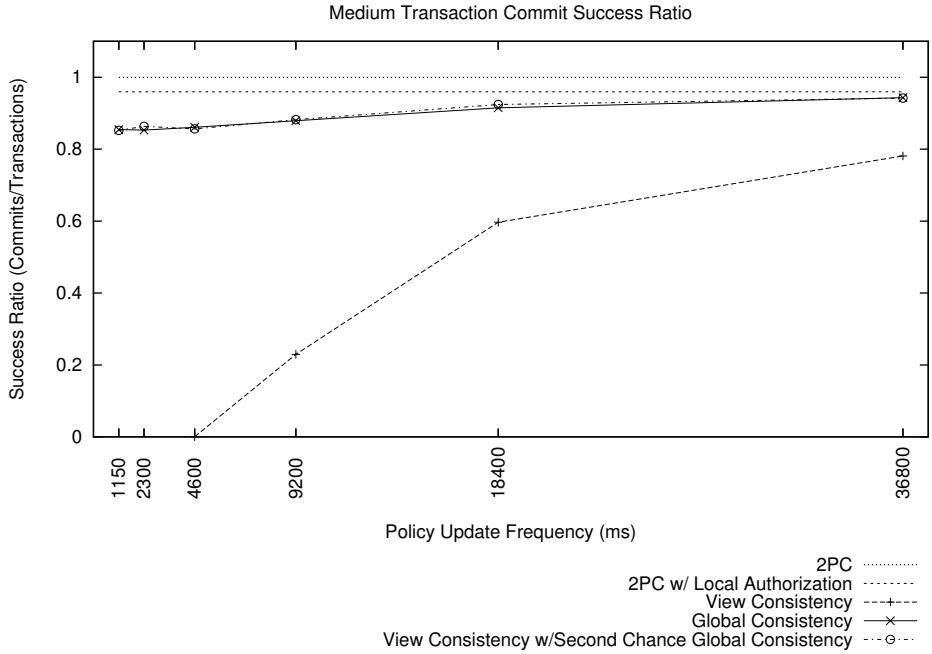

Figure 5: Medium Transaction Commit Success Ratio

For medium length transactions, the 2PC baseline success ratio remains 1.0, or 100%. For 2PCLA, we observed a baseline success ratio of 0.9597, or 95.97%. Figure 5 shows the results of the 2PV protocols against the baseline values. View consistency had no commits at either the 1,150ms or 2,300ms frequency update marks, and had just one commit out of three runs of 1,000 transactions for the 4,600ms mark, resulting in a success ratio of 0.03%. The view consistency success ratio again improves as policy update frequency is lengthened, resulting in an 78.1% commit success ratio by the 36,800ms mark, nearly 18% below that of 2PCLA. Global consistency and view consistency with second chance again show nearly identical performance characteristics. Both protocols perform well in terms of commit success ratios, beginning at the 1,150ms mark with 85.35% ±0.12% success rates and improving to 94.25% ±0.05% success by the 36,800ms mark, almost 4% below the 2PCLA baseline of 98.1%.

For long length transactions, the 2PC baseline success ratio remains 1.0, or 100%.

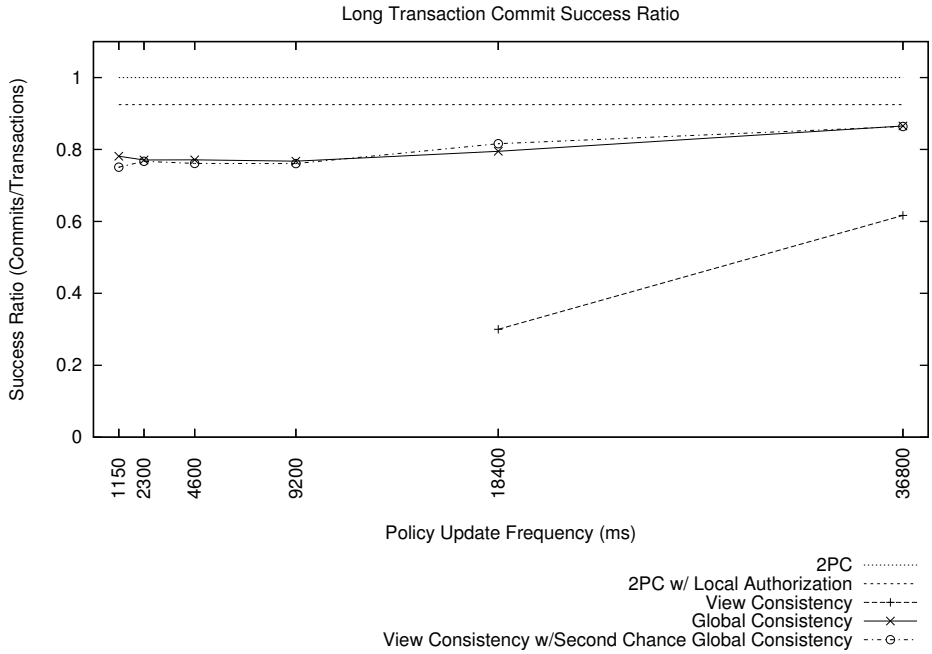

Figure 6: Long Transaction Commit Success Ratio

For 2PCLA, we observed a baseline success ratio of 0.9247, or 92.47%. Figure 6 shows the results of the 2PV protocols against the baseline values. View consistency began producing successful commits at the 18,400ms mark at a rate of 30%. At the 36,800ms mark, its success rate is slightly more then doubled to 61.67%, over a third less than the 2PCLA baseline. Global consistency and view consistency with second chance yet again show nearly identical performance characteristics. Both protocols again outperform view consistency in terms of commit success ratios, beginning at the 1,150ms mark with 78.13% and 75.07% success rates, respectively, and improving to 86.53% and 86.37% success by the 36,800ms mark, slightly more then 6% below the 2PCLA baseline of 92.47%.

Transaction Throughput

We represent transaction throughput as the number of commits in a simulation run divided by the time in milliseconds of the duration of that simulation run. We performed simulations of all protocols for short transactions (see fig. 7), medium transactions (see fig. 8), and long transactions (see fig. 9). We began with a policy update frequency of 1,150ms and doubled the amount for each subsequent set of runs up to 36,800ms.

For short length transactions we observed a 2PC baseline transaction throughput

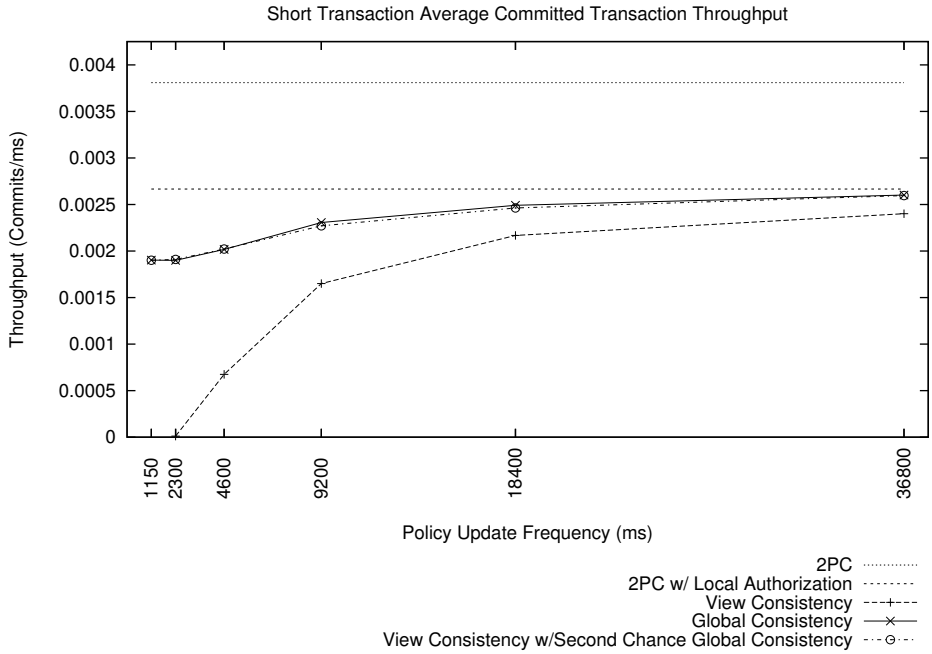

Figure 7: Short Transaction Throughput

of 0.003809 commits/ms. For the 2PCLA baseline, we observed a baseline transaction throughput of 0.002665 commits/ms. Figure 7 shows the results of the 2PV protocols against the baseline values. View consistency follows a similar trend to its performance in successful commit ratios (see fig 4, which is logical since the quantity of commits for throughput for each protocol is the number of successful commits for each protocol. View consistency produces a throughput of virtually zero at the 2,300ms update frequency mark due to there being only six successful commits at that update frequency. Its performance improves to 0.002400 commits/ms by the 36,800ms mark, about 11% less than the 2PCLA baseline. Global consistency and view consistency second chance fare better, beginning at the 1,150ms update frequency with 0.001903 commits/ms and 0.001900 commits/ms, respectively, showing performance about 29% below the 2PCLA baseline. By the 36,800ms mark, they improve to 0.002602 commits/ms and 0.002596 commits/ms, respectively, falling between 2.4% and 2.6% below the 2PLCA baseline.

For medium length transactions we observed a 2PC baseline transaction throughput of 0.001984 commits/ms. For the 2PCLA baseline, we observed a baseline transaction throughput of 0.001338 commits/ms. Figure 8 shows the results of the 2PV protocols against the baseline values. View consistency still performs poorly at the outset, with a negligible throughput at the 4,600ms update frequency, but improving to 0.001095 com-

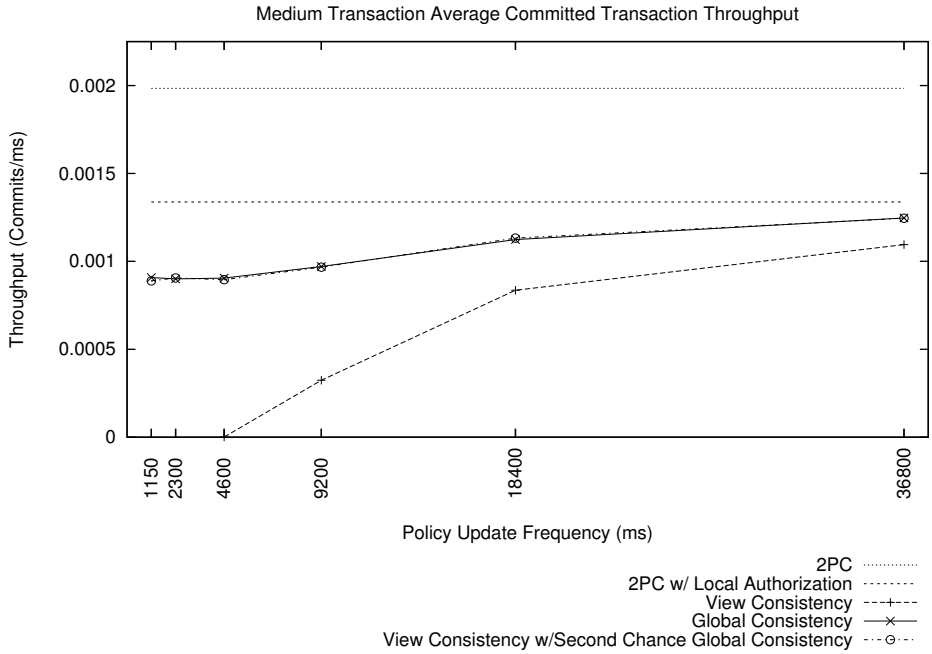

Figure 8: Medium Transaction Throughput

mits/ms by the 36,800ms mark, about 18% less than the 2PCLA baseline. Global consistency and view consistency second chance perform reasonably well at the outset, beginning at the 1,150ms update frequency with 0.000908 commits/ms and 0.000889 commits/ms, respectively, showing performance about 32% to 34% below the 2PCLA baseline. By the 36,800ms mark, they improve to 0.001246 commits/ms and 0.001245 commits/ms, respectively, nearly identical performance at 6.9% below the 2PCLA baseline.

For long length transactions we observed a 2PC baseline transaction throughput of 0.001145 commits/ms. For the 2PCLA baseline, we observed a baseline transaction throughput of 0.000752 commits/ms. Figure 9 shows the results of the 2PV protocols against the baseline values. View consistency echoes its commit success rate performance, producing throughput only at the 18,400ms and 36,800ms update frequency marks. Its throughput at the former was 0.000247 commits/ms and 0.000506 commits/ms at the latter. These values are, respectively, 67% less and 33% less tan the 2PCLA baseline. The throughput of global consistency and view consistency second chance is less impressive than in short and medium transactions but still outperforms view consistency for long transactions. Throughput for global consistency and view consistency second chance begin at 0.000481 commits/ms and 0.000468 commits/ms, respectively, for the 1,150ms update frequency, a performance of about 36% to 38% below the 2PCLA baseline. By the 36,800ms mark, they improve to

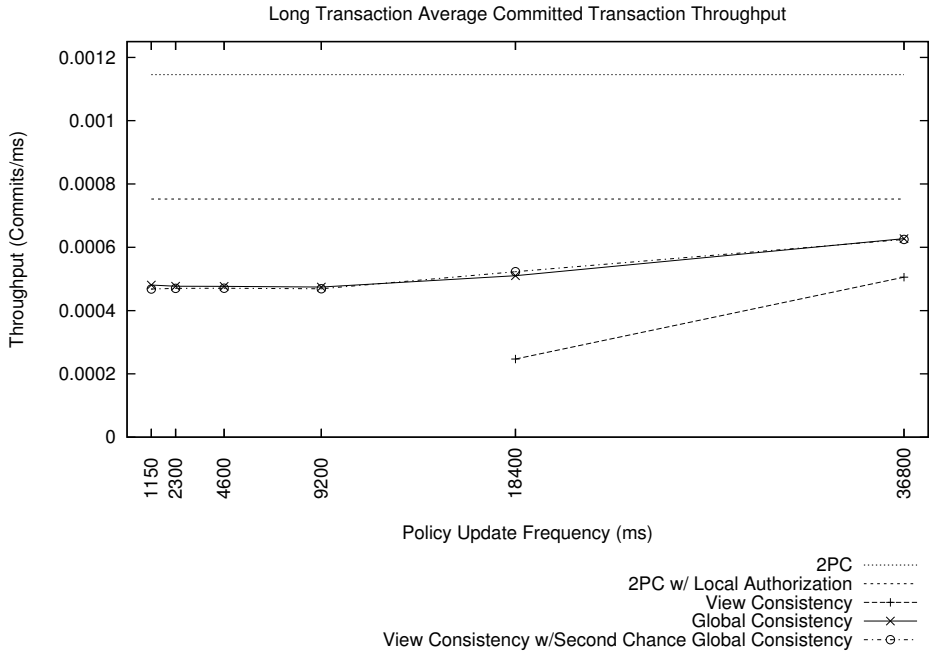

Figure 9: Long Transaction Throughput

0.000627 commits/ms and 0.000625 commits/ms, respectively, again nearly identical performance but at 16.6% and 16.9% below the 2PCLA baseline.

B. Sensitivity analysis

give analysis

2PC vs. 2PCLA? 65-70% less for throughput

all fall beneath 2PCLA, as could be expected

GC and VC2nd nearly identical, cost of an extra GC is offset by VC quickness if successful

Commit success: VC fares poorly for all lengths, success decreases with length but increases with update interval, suggesting that for longer frequencies may approach something good (cite paper on xn length?); GC and VC2nd do reasonably well as freq updates are longer and ten lengths are shorter

throughput is subject to successful commits

concurrency affects VC performance, as all that overlap a policy update will abort

4. Conclusions & Observations

In this section we summarize the findings from the simulation and explore the potential of the algorithms simulated. We will note any findings that suggest further simulation or experimentation. We will also recount any notable observations that occurred during implementation or simulation.

notes

Mention reduction of group size, object was to implement all proofs, instead doing half, leaving remainder for future research

References

[1] Iskander *et al.*, "Enforcing Policy and Data Consistency of Cloud Transactions," Department of Computer Science, University of Pittsburgh.