Cloud transactions and caching for improved performance in clouds and DTNs

Dileep Mardham

CLOUD TRANSACTIONS AND CACHING FOR IMPROVED PERFORMANCE IN CLOUDS AND DTNS

by

DILEEP KUMAR MARDHAM

A THESIS

Presented to the Faculty of the Graduate School of the

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

In Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE IN COMPUTER SCIENCE

2018

PUBLICATION THESIS OPTION

This thesis consists of the following articles that has been published as follows:

Paper I, Pages 21- 44 have been accepted to the 33rd ACM Symposium on Applied Computing (SAC), 2018.

Paper II, Pages 45 - 66 have been accepted to the 14th Wireless On-demand Network Systems and Services Conference (IEEE/IFIP WONS), 2018.

ABSTRACT

In distributed transactional systems deployed over some massively decentralized cloud servers, access policies are typically replicated. Interdependencies ad inconsistencies among policies need to be addressed as they can affect performance, throughput and accuracy. Several stringent levels of policy consistency constraints and enforcement approaches to guarantee the trustworthiness of transactions on cloud servers are proposed. We define a look-up table to store policy versions and the concept of "Tree-Based Consistency" approach to maintain a tree structure of the servers. By integrating look-up table and the consistency tree based approach, we propose an enhanced version of Two-phase validation commit (2PVC) protocol integrated with the Paxos commit protocol with reduced or almost the same performance overhead without affecting accuracy and precision.

A new caching scheme has been proposed which takes into consideration Military/Defense applications of Delay-tolerant Networks (DTNs) where data that need to be cached follows a whole different priority levels. In these applications, data popularity can be defined not only based on request frequency, but also based on the importance like who created and ranked point of interests in the data, when and where it was created; higher rank data belonging to some specific location may be more important though frequency of those may not be higher than more popular lower priority data. Thus, our caching scheme is designed by taking different requirements into consideration for DTN networks for defense applications. The performance evaluation shows that our caching scheme reduces the overall access latency, cache miss and usage of cache memory when compared to using caching schemes.

ACKNOWLEDGMENTS

I would like to convey gratefulness to all those who helped me on this thesis. I would like to thank my advisor Dr. Sanjay Kumar Madria, who gave me a wonderful opportunity to work on this research. His suggestions, feedback and encouragement carried me through strenuous times. His intellect and expertise in this field of research is unparalleled. Secondly, I would also like to express my gratitude to Dr. Wei Jiang Yin and Dr. Fikret Ercal for serving on my thesis committee and for taking time to review this work.

I am grateful to all my lab mates and friends for their professional help, inputs. This project is partially funded by the Air Force Research Laboratory (AFRL) in Rome, New York and I express my gratitude to them as well.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

1. INTRODUCTION

1.1 INTRODUCTION AND MOTIVATION FOR CLOUD TRANSACTION

Cloud is a computing model that provides on-demand computing resources ranging from small – large scale applications to data centers over the internet on a pay for use basis. Cloud computing provides many features like elasticity, which is the ability of a system to automatically up-scale or down-scale the computing resources on demand as workload changes. Cloud based applications run on distant computers "in the cloud" that are owned and operated by others (Amazon, Google, Yahoo, Microsoft etc.,) and that connect to users' computers via the Internet. It also provides cloud-based environment to support the complete life cycle of building and delivering web-based applications without the cost and complexity of buying and managing the underlying hardware, software, provisioning and hosting.

One of the most appealing aspects of cloud computing is its elasticity, which provides an illusion of infinite, on- demand resources [1] making it an attractive environment for highly-scalable, multi-tiered applications. However, this can create additional challenges for back-end, transactional database systems, which were designed without elasticity in mind. Despite the efforts of key-value stores like Amazon's SimpleDB, Dynamo, and Google's Bigtable to provide scalable access to huge amounts of data, transactional guarantees remain a bottleneck [2].

The most important aspects of cloud computing are elasticity and scalability, which catalyzed much of the recent interest in cloud computing, raises many complex issues for back-end, transactional database systems. Cloud services often make heavy use of replication to ensure consistent performance and availability. A common trait characterizing the new generation of cloud data platforms is the adoption of weak consistency models, such as eventual consistency, restricted transactional semantics and non-serializable isolation levels.

Eventual consistency allows data to be inconsistent among some replicas during the update process, but ensures that updates will eventually be propagated to all replicas. The major problem associated with this approach is the difficulty in maintaining the

ACID guarantees, in which Consistency is compromised to provide reasonable availability.

Authorization policies describes the conditions under which users should be permitted access to resources and describes the relationships between the system principals, as well as the certified credentials that users must provide to attest to their attributes. The transactional database system is deployed in a highly distributed and elastic system, policies are typically replicated among multiple servers following the eventual consistency model. It therefore becomes possible based on policy-based authorization system is prone to make unsafe decisions using stale policies.

Interesting consistency problems can arise as transactional database systems are deployed in cloud environments and use policy-based authorization systems to protect sensitive resources. In addition to handling consistency issues amongst database replicas, we must also handle two types of security inconsistency conditions. First, the system may suffer from *policy inconsistencies* during policy updates due to the relaxed consistency model underlying most cloud services. For example, it is possible for several versions of the policy to be observed at multiple sites within a single transaction, leading to inconsistent (and likely unsafe) access decisions during the transaction. Second, it is possible for external factors to cause *user credential inconsistencies* over the lifetime of a transaction [3]. For instance, a user's login credentials could be invalidated or revoked after collection by the authorization server, but before the completion of the transaction. If the problems associated with policy consistency are not alleviated, the company or individual may face a potential risk. The company may leak information about customers and face harsh penalties and loss of credibility.

Cloud services are widely adopted by various organizations for resource sharing and elasticity. The vendors of cloud services generally lack consistency among the policies across all the servers. Due to this inconsistency, there is a possibility of transaction abort when a user tries to request with older policy. In this case, the user will not be able to access the data item as the policy doesn't match with the newer policy. A user who earlier had access to the data item will not be able to perform any operations on the data item. Such situations should be avoided, and we present the below mechanism to resolve such issues.

In addition to handling consistency issues among database replicas, policy inconsistencies and user credential inconsistencies needs to be addressed for which possible solutions have been proposed. But there are performance and latency issues with these solutions and we address them in this paper.

1.2 INTRODUCTION AND MOTIVATION FOR CACHING

Caching data within DTNs is discovered as a means for both persisting data locally to make the data both more readily available in close proximity, and to reduce resource usage through a reduction in hops to the high priority data sources. Additionally, DTNs present an even more significant challenge for data persistence as they are, by nature generally isolated and disconnected from a larger network or the Internet. Therefore, DTNs must cache mission critical useful data to share within the network. Though local device caching of data is a useful method to accomplish data persistence and increase availability, it comes with its own set of challenges and problems in DTNs. Situations can arise where specific nodes within the network become over utilized by caching too many data points due to the uncertainty of data transmission, thus multiple data copies need to be cached at different locations to ensure higher data dissemination. The difficulty in coordinating multiple caching nodes due to the lack of persistence network connectivity makes it hard to optimize the trade-off between data accessibility and caching overhead. Thus, each node has to decide what data to be cached as nodes cannot design a cooperative caching schemes because of not knowing which other nodes they can reach. A situation may come where specific nodes become over burdened by caching too many important messages. This results in the storage space for that specific device becoming over utilized and the battery life being exhausted due to the device carrying a disproportionate amount of data and forwarding requests for the data it holds to each peer it meets. Caching in DTNs can create a situation where the data might not be proximate to the nodes that need it. This results in an increase in the energy utilization due to the need to disseminate the data across many peers. Caching can consider the location of the node, POIs and queries performed by the node and then predicting the caching needs at the next time instants.

DTN networks use caching as a technique to store some of the transferring files in order to satisfy the future requests reduced data access delay. Each Network Operating System (NOS) which uses DTNs requires a unique kind of caching method to fulfil its particular needs. From all the data caching methods we have studied, file popularity and cooperative caching have always been the main motivation behind the file selection for caching to satisfy future requests and how cache replacement should to be done. Although these methods are efficient in many general scenarios where users transfer picture or other kinds of media files in a busy network using mobile devices, there are some scenarios where these methods of caching entirely fail to address the requirement. In situations like Military networks, or any other defense data networks, the data need that need to be cached follows a whole different priority. Some files despite of low popularity should be cached with highest priority which is completely against the conventional caching methods that we use today. Some other classified data files even with the increasing demand the data replication should be restricted to a certain number of times in order to prevent data theft or other security issues. So, a new caching technique has to be designed which takes different requirements into consideration to fulfil this necessity in defense data networks.

The simplest solution would be to just cache *everything* we can on every device and update it as we make connections to other nodes in the network, the problem with this is twofold: collisions, and space/energy constraints, so this approach is unrealistic. Because of this we must make decisions in exactly how we are going to cache data around this network. If an item is too popular a single node may become overwhelmed with requests for that data, so it should probably be replicated. In addition, data that is not cached locally must be retrieved with one or more hops around the network. This adds latency to the system, and consumes more power with the requests and responses for the data, so exactly where to store the data is another dimension of concern. And some data may not be as popular, but latency to it may be such a concern that it must be accessible quickly. These three elements: location, latency, and popularity of the data will be the primary qualities we use to judge data in how exactly to cache it.

2. RELATED WORK

2.1 RELATED WORK IN SECURE CLOUD TRANSACTIONS DOMAIN

Cloud providers lack services that guarantee both data and access control policy consistency across multiple data centers. In the article [3], several policy and user credential consistency problems are identified, and policy based authorization systems are used to enforce access control. In previous papers [4], [5], the confluence of data, policy, and credential inconsistency problems that can emerge using transactional database systems are addressed. In doing so, the authors made the following contributions:

- Initially, formalization of the concept of *trusted transactions* is presented. Trusted transactions are those transactions that do not violate credential or policy inconsistencies over the lifetime of the transaction. The authors then presented a more general term, *safe transactions*, that is used to identify transactions that conform to the ACID properties of distributed database systems *and* are trusted in terms of the validity of the policy evaluation.
- Since achieving ACID properties in distributed transactional databases has been extensively studied [4], [5], they focused on how to achieve trusted transactions. Accordingly, they defined different levels of policy consistency constraints as well as different enforcement approaches to guarantee the trustworthiness of transactions executing on cloud servers.
- The authors also proposed a solution that involves an adaptation of the Two-Phase Commit (2PC) protocol to enforce trusted transactions, which we refer to as Two-Phase Validation Commit (2PVC) protocol. The protocol ensures that a transaction is *safe*, as it ensures policy and credential consistency as well as data consistency.
- The authors presented a performance analysis study of the proposed approaches.

Proof of Authorization

We now present a formal definition of a proof of authorization. Let f_{si} = <q_i, s_i, $P_{si}(m(q_i))$, t_i, c> denote the proof of authorization evaluated at server s_i, where q_i is a query defined over a set of read/write requests submitted to that server. P_{si} denote the proofs of authorizations enforced by server s_i and belonging to the same administrative domain *A*. Function *m* is a mapping such that $m : Q \rightarrow 2D$, that is, *m* identifies the set of data items that are being touched by query *q*. Time t_i is the time instance at which the proof of authorization is being evaluated, and finally *c* is a set of credentials presented by the user to complete the proof of authorization such that $c \subseteq C$ which is the set of all credentials.

Let *F* denote the set of all proofs of authorizations, and the set *TS* contains all possible timestamps. The validity of each proof of authorization $f \in F$ at time instance *t* is evaluated using the predicate *eval*(*f*, *t*) such that $eval : F \times TS \rightarrow B$. The Boolean sign is true if the proof of authorization is valid. The validity of a proof of authorization is asserted in two cases:

1) *Credentials are syntactically and semantically valid:* According to the definitions in [4], a credential c_k is syntactically valid if the following conditions hold: (i) it is formatted properly, (ii) it has a valid digital signature, (iii) the start time of each transaction $\alpha(c_k)$ has passed, and (iv) the time at which the transaction finishes execution and is ready to commit $\beta(c_k)$ has not yet passed. A credential c_k issued at time t_i is semantically valid at time *t* if an online method of verifying c_k's status indicates that c_k was not revoked at time t' and $ti \leq t' \leq t$.
2) *The inference rules are satisfying:* A policy is a set of inference rules that are encoded by policy makers to capture system's access control regulations. Given policy *P*, and user credentials *C*, if the inference rules of the policy can be satisfied using the user credentials, then the proof of authorization is said to be valid and the access is granted accordingly.

Consistency Levels

Since transactions are executed over time, the state information of the credentials and the policies enforced by different servers are subject to changes at any time instance, therefore it becomes important to introduce precise definitions for different consistency levels that could be achieved within transaction's lifetime. These consistency models strengthen the notion of trusted transaction by defining the environment in which policy versions are consistent relative to the rest of the system. Before we do that, we define a transaction's *view* in terms of different proofs of authorizations evaluated during the lifetime of a transaction.

Two consistency models have been proposed to handle consistency within transactions:

- *View Consistency:* A view $V^T = \{<q_i, s_i, P_{si}(m(q_i)), t_i, c>, \ldots, <q_n, s_n, P_{sn}(m(q_n)), t_n, c>\}$ is *view consistent*, or Φ-consistent, if V^T satisfies a predicate Φ-consistent that places constraints on the versioning of the policies such that Φ-consistent(V^T) $\leftrightarrow \forall_{i,j} : ver(P_{si}) = ver(P_{sj})$ for all policies belonging to the same administrator *A*, where function *ver* is defined as $ver : P \rightarrow \mathrm{N}$ where P is set of all policies and N is the infinite set of natural numbers.

  The policy versions should be internally consistent among all servers executing the transaction. With a view consistency model, the policy versions should be internally consistent across all servers executing the transaction. The view consistency model is weak in that the policy version agreed upon by the subset of servers within the transaction may not be the latest policy version *v*. It may be the case that a server outside *S* servers has a policy *P* that belongs to the same administrative domain *A* and with a version $v' > v$. A more strict consistency model is the global consistency and is defined as follows.

- *Global Consistency:* A view $V^T = \{<q_i, s_i, P_{si}(m(q_i)), t_i, c>, \ldots, <q_n, s_n, P_{sn}(m(q_n)), t_n, c>\}$ is *global consistent*, or Ψ-consistent, if V^T satisfies a predicate Ψ-consistent that places constraints on the versioning of the policies such that Ψ-consistent(V^T) $\leftrightarrow \forall i : ver(P_{si}) = ver(\mathrm{P})$ for all policies belonging to the same administrator *A*, and function *ver* follows the same aforementioned definition, while *ver*(P) refers to the latest policy version.

  The policies used to evaluate the proofs of authorization during a transaction execution among S servers should match the latest policy version among the entire policy set P, for all policies enforced by the same administrator *A*.

Trusted Transactions

Trusted transactions are defined as transactions which do not fail to agree with the credential or policy inconsistencies during execution. Given a transaction $T = \{q_1, q_2, \ldots,$

q_n} and its corresponding view V^T , T is *trusted* iff $\forall f_{si} \in V^T$: $eval(f_{si}, t)$, at some time instance t : $\alpha(T) \leq t \leq \beta(T) \wedge (\Phi\text{-consistent}(V^T) \vee \Psi\text{-consistent}(V^T))$.

Based on the consistency models, the term *safe* transaction is defined which is a trusted transaction and satisfies all data integrity constraints imposed by the database management system. In practical, a transaction will commit if it is safe and any unsafe transaction is forced to rollback.

Trusted Transactions Enforcement

A variety of light-weight proof enforcement and consistency models (Deferred, Punctual, Incremental, and Continuous models) have been proposed that enforced increasingly strong protections with minimal runtime overheads. A brief description of different proofs of authorization models is given below:

- *Deferred proofs of authorization* are evaluated simultaneously at commit time to decide whether the transaction is trusted. A transaction T and its corresponding view V^T , T is trusted under the deferred proofs of authorization approach, iff at commit time $\alpha(T)$, $\forall f_{si} \in V^T$: $eval(f_{si},\beta(T)) \wedge (\Phi\text{-consistent}(V^T) \vee \Psi\text{-consistent}(V^T))$. By employing deferred proofs of authorizations, transactions are most likely to execute faster but on the expense of risking a transaction to be forced to rollback after it has proceeded till the commit time in case of violation of the trusted transaction condition.
- *Punctual proofs of authorization* are evaluated instantaneously whenever a query is being handled by a server. This is a proactive approach and facilitates early detection of unsafe transactions, saving system from getting into costly undo operations. Proofs of authorization are re-evaluated at the commit time. Given a transaction T and its corresponding view V^T, T is trusted under the Punctual proofs of authorization approach, iff at any time instance t_i : $\alpha(T) \leq t_i \leq \beta(T)$ $\forall$ $f_{si} \in V^T$: $eval(f_{si}, t_i) \wedge eval(f_{si}, \beta(T)) \wedge (\Phi\text{-consistent}(V^T) \vee \Psi\text{-consistent}(V^T))$.

  Punctual proofs of authorization do not impose any restrictions on the freshness of the policies used by the servers to evaluate the proofs during the transaction execution. It is only at commit time when the proofs of authorization are re-evaluated while enforcing view consistency or global consistency. Hence, due to the weak consistency paradigm on which cloud servers operate, a server might evaluate a proof based on an old version of a policy and in that case there is no guarantee that the decision made by that server is valid or invalid. Consequently, servers might have false negative decisions and deny access to queries, and on the other hand, false positive decisions could also be made.

- *Incremental Punctual proofs of authorization* doesn't allow a transaction to proceed unless each server achieves the desired level of the policy consistency with all previous servers. All participating servers are forced to be in consistent view with the first executing server and if a newer policy version shows up at a later server, the transaction aborts. Given a transaction T and its corresponding view V^T , T is trusted under the Incremental Punctual proofs of authorization approach, iff at any time instance ti : $\alpha(T) \leq t_i \leq \beta(T)$, $\forall f_{si} \in V^T_{ti}$: $eval(f_{si}, t_i) \wedge$ (Φ-consistent(V^T_{ti}) $\vee$ Ψ-consistent(V^T_{ti})).

  If the first server s_1 does not have the latest version, the proof of authorization at that server is risked to be evaluated using an older policy. Note that in this scenario if any of the other servers has a newer policy version, the consistency condition will not be satisfied and the transaction will be forced to rollback, saving the transaction from doing any further untrusted authorizations.

- *Continuous proofs of authorization* are evaluated throughout a transaction's lifetime and if a newer version of the policy is found at any of the participating server, all previous proofs have to be re-evaluated after updating policies on all inconsistent servers. A transaction T is declared trusted under the Continuous approach, iff $\forall_{1\leq i\leq n}\forall_{1\leq j\leq i}$: $eval(f_{si}, t_i) = true \wedge eval(f_{sj}, t_i) = true \wedge$ (Φ-consistent(V^T_{ti}) $\vee$ Ψ-consistent(V^T_{ti})) at any time instance t : $\alpha(T) \leq t_i \leq \beta(T)$.

  In Continuous proofs of authorizations, at every time instance when an evaluation of a proof of authorization is being made, all previous proofs of authorizations are forced to be re-evaluated before the transaction can proceed. If any of the evaluations fail at any time instance, the entire transaction is forced to rollback.

Two-Phase Validate Commit Algorithm

A two-phase validation commit protocol (2PVC) is proposed which is an improvised version of the two-phase commit protocol (2PC) and the two phase validation protocol (2PV). In 2PVC validation of the authorization policy and credentials are carried out compared to 2PC in which data integrity check is performed. 2PVC will evaluate the policies and authorizations within the voting phase. That is, when the TM sends out a Prepare-to-Commit message for a transaction, the participant server has three values to report: (1) the YES or NO reply for the satisfaction of integrity constraints as in 2PC, (2) the TRUE or FALSE reply for the satisfaction of the proofs of authorizations as in 2PV, and (3) the version number of the policies used to build the proofs (v_i, p_i) as in 2PV.

Algorithm: Two-Phase Validation Commit 2PVC

```
1  Send "Prepare-to-Commit" to all participants
2  Wait for all replies (Yes/No, True/False, and a set of policy versions for each unique policy)
3  If any participant replied No for integrity check
4      ABORT
5  Identify the largest version for all unique policies
6  If all participants utilize the largest version for each unique policy
7      If any responded False
8          ABORT
9      Otherwise
10         COMMIT
11 Otherwise, for participants with old policies
12     Send "Update" with the largest version number of each policy
13     Wait for all replies
14     Goto 5
```

The process given in Algorithm 2 is for the TM under view consistency. It is very similar to that of 2PV with the exception of handling the YES or NO reply for integrity constraint validation and having a decision of COMMIT rather than CONTINUE. The

TM enforces the same behaviour as 2PV in that it identifies policy inconsistency, sends Update messages to create consistency, and re-executes the first phase. The same changes to 2PV can be made here to provide global consistency. That is, the global 2PVC does not need to determine the latest version number from the participant votes. Instead, it simply asks some master server on the system which knows the latest policy version at Step 5.

Complexity

The cost of 2PC is typically measured in terms of log complexity (i.e., the number of times the protocol forcibly logs for recovery) and message complexity (i.e., the number of messages sent). We add another metric, namely the number of proof evaluations. These metrics are given with respect to the number of participants involved with the decision, *n*, the number of queries, *u*, and the number of voting rounds, *r*. The log complexity of 2PVC is no different than normal 2PC, which has a log complexity of 2*n* + 1 [4]. Table 2.1 shows the complexity—in terms of the maximum number of messages and proofs—for each proof of authorization scheme for both view and global consistency.

Simulated workloads have been used to experimentally evaluate implementations of the consistency models relative to three core metrics: transaction processing performance, accuracy and precision.

Table 2.1. Contrasting The Various Proofs Of Authorization

	Deferred		Punctual		Incremental		Continuous	
	View	Global	View	Global	iew	Global	View	Global
Messages	*2n + 4n*	*2n + 2nr + r*	*2n + 4n*	*2n + 2nr + r*	*4n*	*4n + u*	*u(u + 1) + 4n*	*u(u + 1) + u + 2n + 2nr + r*
Proofs	*2u - 1*	*ur*	*u+ 2u − 1*	*u + ur*	*u*	*u*	*u(u+1)/2*	*u(u+1)/2 + ur*

2.2 RELATED WORK IN DTN OPPORTUNISTIC CACHING DOMAIN

The paper [20] deals with improving the coordination between the multiple caching nodes in the delay tolerant networks (DTN's) to optimize the data accessibility and caching overhead. The main challenges that are dealt in this paper are how to select the nodes that are optimal to cache the data. How to overcome the limited buffer space in central nodes to improve the total caching limit, how to prioritize the data that need to be cached are all issues plagued by any research into this topic.

To select the Network Central Locations (NCL's) three types of NCL selection methods are used namely NCL selection metric, Trace-based validation and Practical NCL selection. In NCL selection metric the nodes which participate most in connecting all the nodes using the shortest path with less cost are selected and made as NCLs. In trace-based validation we select the nodes based on their popularity. The nodes are selected using realistic DTN traces. These traces are observed for a certain amount of time and each node popularity is decided on how well they are connected to the other nodes. In practical NCL selection we select 'K' best NCLs that are available in the network. In this method we use NCL selection metric that was used before, but we try to limit the NCL's to K number. Two ways to make this selection are global selection where the nodes are selected recursively to support the previous nodes until K nodes are selected and distributed selection where the k nodes are selected independent of the global network knowledge.

To expand the cache buffer space if all the NCLs cache is full then the nodes that are close to NCLs are selected and they are used to store the less popular data. To improve the coordination between the NCLs whenever two NCL’s encounter each other they compare and share their respective information to optimize the information. So, with the increase in time all the NCLs might encounter each other and the information they carry becomes optimized.

Coordination between different NCLs and optimization of data of every NCL can become an endless optimization cycle which consumes a lot of energy if the network is very big and can never be optimized. To overcome this drawback, we think the NCL optimization should be limited to certain range and the range depends on how diverse the nodes are and how big the network is. From this paper we can understand how NCL placement in the network is critical for the efficient data transmission and how caching space limitations can be improved.

In the paper [21], we consider a delay-tolerant content sharing network built over a network of mobile users and wireless access points, where the users download content opportunistically from each other via short-range communications (e.g. Bluetooth or Wi-Fi). If the requested content is not found within the prescribed time, the users will download it through the more expensive 3G network.

This paper discusses the factors that contribute for caching in DTNs are Cache capacity within each node, User mobility, Density of the access points, how well the requests are distributed within the nodes to find the data and contact duration i.e. at least for long should the connection exist to transfer the complete file. The ultimate goal is to send the query request to all the nodes and retrieve the data back to requested node within a certain amount of time.

Cooperative caching is the method used in this paper and to achieve that cooperation between the nodes we use Zipf's law. According to Zipf's law every file has a certain probability of getting requested and the files with higher probability should be replicated more to improve the data retrieval time. Due to the additional time constraint i.e. to retrieve the data within a certain amount of time, four different pushing methods are used to flood the nodes with all the available data.

1) *Random Pushing:* Each mobile device randomly stores K files in their cache.
2) *K-most Popular:* The K files with highest probability of request are cached.
3) *Optimal:* Files are distributed according to optimal file distribution algorithm.
4) *Pushing Algorithm:* Files are distributed because of the above selective pushing algorithm.

From the analysis, to improve the overall average hit ratio of the cache, the files which are popular should be replicated more and less popular files should be less replicated. By storing more copies of the popular files and less copies of the unpopular files, the overall hit rate could possibly improve. The miss ratio decreases exponentially with the patience time. Random push strategy has the highest miss ratio compared to others whereas k-most popular miss ratio didn't change much with increase in file number. Optimal file pushing strategy got less miss ratio with the increase in file number.

One assumption made in this paper by authors is that all the nodes has the same amount of cache space and all the nodes are well and uniformly connected to each other. So, there is no discussion about the selection of NCLs in the network. This clearly is a drawback because in a practical situation no two nodes behave similarly and NCLs play a huge role in DTN caching data retrieval.

The main contribution of the paper [22] is to provide a formal framework for the characterization of the performance of optimal in-network caching in ICNs, as well as opportunistic in-network caching at the edge of ICNs—i.e., close to the end-users. The authors used Independent Reference Model (IRM), which assumes that the objects are equal sized whose references occur independently, to study the benefits of using universal caching compared to a simple policy of caching only at the edge of the network assuming a simple hierarchical caching structure.

The authors consider a hierarchy of LRU (Least Recently Used) caches in the form of a tree with its root acting as the content source. The paper assumes that the source stores permanent copies of all the information objects in the system. Alternatively, the source can be considered as a collection of all possible content hosts that are logically collapsed into one single entity as the root of the tree in our model. The tree comprises L + 2 levels. The content subscribers (i.e., users or information requesters) are at the 0th level, while the content source is at level L + 1. Subsequently, there exist L levels of nodes with caching capabilities between users and the content source which are sequentially labelled from bottom (level 1) to the top (level L).

The caching paradigm used to optimize is called “on-path caching” which works as follows. When a request for an object is raised at level 0, it is forwarded along the (unique) path of intermediate caches towards the root until a cache hit occurs. If all cache accesses are missed along the path, the request will be fulfilled by fetching a copy of the object directly from the source (root). Once located, the object is transferred on the reverse path back to the requester and a local copy is also stored on each and every node along the path.

The results using this model demonstrate that, while optimal caching naturally tends towards the edge with an increased caching budget, higher degrees of reference locality further accelerate this transition. The results indicate that the optimal caching approach based on universal caching provides only marginal benefits over the simple policy of caching only at the edge routers of the ICN. The paper failed to take into consideration the following aspects:

- More realistic topologies and Develop model for random networks.
- Verify synthetic traces for locality of reference model with real traffic traces.
- New approaches to integrate routing with edge caching.

The paper [23] discusses about locating the Network Central Locations (NCL) within all the DTNs based on the node's social levels and how well it is connected to other nodes. Selected data which has great popularity in a region is cached on a node at that location. Even though the most popular data of the network has many duplicate copies within the NCLs the popularity of the data is gradually decreased depending on that node's distance from the node where that data has highest popularity. Caching space limitation is improved by utilizing the cache space of the other selective nodes near the NCLs. The cache of the NCLs is replaced periodically with new data based on the frequency of data usage, user requests and freshness of data access. To find the best nodes to be selected as NCLs within all DTNs K-Means Clustering algorithm is used.

This paper proposes a new method called social based forwarding approach for content retrieval. This method follows several steps which are as follows.

Compute Social-Tie Relationship: Two nodes are said to have a strong tie if they have met frequently in the recent past. We compute the social tie be1tween two nodes using the history of encounter events.

Compute centrality: Each node maintains a social-tie table that contains the social distances from the current node to all other encountered nodes. During the encounter period, the social-tie table is exchanged and merged into the other node's social-tie table. Based on the social-tie table, a node can compute each other node's centrality. We estimate the centrality by considering both the average social-tie values and their distribution. Namely, we favor nodes with high, uniformly distributed social ties to all other nodes.

Compute Social Level: Nodes that have similar centrality tend to have similar level of contacts with other nodes and thus similar knowledge on content providers. To reduce the forwarding cost of the content query phase, we propose to group together nodes with similar centrality into the same cluster. Interest packets are only forwarded from one cluster to another cluster. There is no Interest forwarding within a cluster. Each cluster represents a social level in the network.

Content Name Digest Convergence: To facilitate content query, each content provider actively announces its content name digest (a list of names of contents a node owns) to nodes in higher centrality clusters. Each node maintains a local data structure called digest table (which maps the provider ID to the digest) to store the received digests from lower centrality nodes. Furthermore, when nodes encounter each other, the digest table will be sent to the node with the higher centrality. Throughout this process, the content name digests from each content provider are converged toward higher centrality nodes. Subsequently, higher centrality nodes have broad knowledge of which node owns which content in the network.

Interest Packet Forwarding: The Interest packet is carried by the requester and is forwarded to the first encountered node that has a higher social level than the requester itself. Subsequently, the requester keeps a copy of the Interest packet and forwards it to the next encountered node that has an even higher social level than the last relay node.

After a node receives an Interest packet from other nodes it encountered, it will first check its local digest table to see if there is any matched name. If no matched name is found, it will continue forwarding the Interest packet. Each relay node performs the same strategy; forwarding the Interest packet to the next relay node that has a higher social level than the last relay node. Following this strategy, the Interest packet is forwarded upward, level by level, toward the most popular node in the centrality hierarchy.

Data Packet Forwarding: After the Interest packet reaches the content provider, the content provider will social-tie route the data packet back to the requester. The content provider only responds once to the same Interest packet that originates from the same requester. Subsequent received duplicate Interest packets are ignored. The prominent issues faced during caching schema are what data is needed to be cached, where the data needed to be cached and how the cached data should be replaced when new data comes into play.

Cached Data Selection: Intuitively, popular data is a good candidate for caching. We compute the content popularity (relative to the current node) by considering both the frequency and freshness of content requests arriving at a node over a history of request arrivals.

Cache Location: If each node has unlimited cache space, then it is trivial to identify suitable caching locations, as data can be cached everywhere. Given that each node has limited space for caching, we follow a conservative approach and only cache data at nodes satisfying the following conditions:

1) Selected nodes are on the query forwarding paths.
2) They are traversed through by many common requests.
3) Caching in neighbors of central nodes, whose caches are heavily utilized, is another optimization implemented in this scheme.

Cache Replacement: When the cache buffer is full, existing data must be evicted from the cache, to accommodate new data. There are two related issues: 1) Determining the amount of data to evict. 2) Identifying particular data to evict. For the first issue, we need to evict as much data as the size of the new data. Regarding the second issue, we propose to remove data from the cache that is identified as least popular. That is, we consider both the frequency and freshness of data access.

Caching Protocol: Nodes periodically advertise their spare cache capacity to each other. This allows central nodes to opportunistically make decisions regarding which cached data to move to neighboring nodes so that central nodes have more space to cache new popular data.

Cooperative caching is the main key to find which data should be stored at which NCL so that the data retrieval is easier and query requests are processed efficiently. Since the file popularity decreases gradually the query search can be stopped after certain range from the NCL where the data was popular. From this paper we can understand how the file popularity helps to cache a file for a long time and how modifying this file popularity index for a file manually can improve a file's life-time which has less requests but of high value for some users. This paper fails to address the following issues.

- Trying to update the social tie values and local digest table involving all the nodes increases the nodes power consumption.

- How to preserve the data which is less popular but very important for some set of users.

In paper [24] content distribution is defined as the task of providing requested data to the clients. Requests are constructed by the client and transmitted to a node capable of servicing them, usually a server or repository. This is different from caching, which ideally is used to provide the same data item to a large number of clients and thereby involves individual nodes keeping a copy of the data item so that they can fulfil requests just as well as the server. A mobile repository, sometimes called a data store or a throwbox, contains a large collection of related items, thus allowing the device to fulfil most requests.

In DTNs, social context can be used to improve the efficiency of certain tasks such as routing or caching. Clients' content requests tend to be influenced by social patterns. In general, if a client's contacts request a specific data item, that client has a greater probability of requesting the item as well. To take advantage of this tendency, a node first must identify the consistent contacts of a node. In a dynamic environment in which nodes are added and removed, the social structure must accommodate such changes.

The purpose of the Social Content Distribution (SCD) schema is to locate the optimal position for mobile repositories. This is accomplished by evaluating the existing social structure of the network, identifying which nodes and groups issue frequent content requests, and locating the devices most capable of fulfilling these requests. Initially, a node identifies its frequent contacts, forming small groups. These groups are merged, joined, and left by nodes to accurately reflect the network's social structure. The end result is the positioning of repositories in close proximity to requests, increasing content availability and reducing the average delivery time.

If a set of nodes meet frequently, the algorithm will assume the nodes are in a group. The request predictions are based on historical patterns; if a node requests data from a throwboxes frequently or are in a social group whose members frequently request content, the throwboxes assumes it will do so in the future. Based on the nodes' predicted content request, the throwboxes relocates itself to serve these requests quickly.

Grouping: In this context, a social group is a collection of nodes that have regular contact with each other. Nodes within a social group maintain group data, including group membership and the metrics of all group members. All nodes maintain their group list in one of following three ways:

i) Two nodes having regular contact with one another form a new social group: The first step in determining any groups formed by nodes is to establish a metric by which to measure the distance between two nodes. The SCD schema calculates the percentage of time spent in direct contact, using an exponential moving average formula to adjust the current estimate. With this information, a cumulative estimate of the contact strength is calculated and compared to a group formation threshold. When the contact strength exceeds a control threshold, the nodes are considered close enough to form a new group.

ii) Two social groups with similar members merge: By tracking the contact strength between nodes, a series of two-node social groups can be formed. The next step is to integrate these links into larger groups by merging similar groups. Nodes that are joint members of two groups will periodically review the group data for merges. If the joint node determines that the two groups are similar enough, then a SUGGEST message is sent to the group head of the smaller of the two groups. This message indicates that the node believes that merging the two groups is justifiable. Performing merges in this manner updates groups in a limited environment. The drawback is that it does not ensure that all nodes of the group are strongly connected to all members of the new group. To address this issue, nodes can resign from groups to which they no longer have a strong attachment.

iii) Nodes resign from groups in which they no longer participate: Periodically, nodes will review their group list to ensure that they are still participating. The average contact strength to all group members is calculated, and if it is beneath ψ, the node resigns from the group. To avoid fragmentation, the sending node will not remove any group data until the message has been confirmed.

Content Repository Positioning:

i) Request Frequency: To determine the optimal position for a repository, it is necessary to identify nodes which frequently request data items. Whenever a node makes a user request, the update process estimates the average time between requests based on the previous estimate and the control variable threshold, which determines how much emphasis, is placed on historical data. This process allows nodes to maintain up to-date estimates adjusted to reflect their individual request patterns.

ii) Group Request Score: A node's ranking depends on its own ability, as well as the ability of its contacts, to deliver a content item to requesting nodes. A node can calculate its Request with Group Score (RGS) which is the time required for a node or a neighboring node to serve a data request.

iii) Repository Position Ranking: At this stage of the process, nodes are aware of the frequency with which they contact other nodes (Request Frequency) and how often these nodes will request content, either on their own behalf or that of their neighbors (Group Request Score). The ranking algorithm establishes a metric, the Rank Position Score (RPS), which is the sum of another node's chance of requesting the data times the chance of encountering node. If RPS of Node B is greater than RPS of Node A, this indicates that Node B is closer than Node A to other nodes that make more frequent requests. A message updating the data owner is sent, and the entire repository shifts.

This paper presented a social algorithm, identifying groups dynamically by measuring the contact intervals. It then used this data to accurately identify which nodes served as optimal positions for mobile repositories.

PAPER

I. CLOUD TRANSACTIONS ADHERE TO STRICT POLICY CONSISTENCY FOR IMPROVED PERFORMANCE

Dileep Mardham[1], Sanjay Madria[1], James Milligan[2] and Mark Linderman[2]

[1]Missouri University of Science and technology, Rolla, MO, USA

[1]Air Force Research Lab, Rome, NY, USA

ABSTRACT

In distributed transactional systems deployed over some massively decentralized cloud servers, access policies are typically replicated. Interdependencies and inconsistencies among policy version replicas can affect performance, throughput and accuracy, which can increase the transaction failure rate and cause long delays. Thus, policy and user credential inconsistencies need to be addressed. Several stringent levels of policy consistency constraints and enforcement approaches to guarantee the trustworthiness of transactions on cloud servers are proposed. However, there are performance issues associated with the policies proposed while retrieving the latest policy versions present in various cloud servers. In this paper, first, we define a look-up table in which the policy versions used for authorization is stored and updated on a regular basis and this information can be easily retrieved by the transaction manager. Next, we use the concept of "Tree-Based Consistency" approach to maintain a tree structure of the servers where a particular data item is replicated. By integrating look-up table for policy versions and the consistency tree based approach, finally, we propose an enhanced version of Two-phase validation commit (2PVC) protocol integrated with the Paxos commit protocol to increase the number of commits of transactions with reduced or almost the same performance overhead (transaction execution time) without affecting accuracy and precision, but reducing the number of transaction aborts in comparison with a most recent work.

1. INTRODUCTION

Cloud provides on-demand [1] resources for highly-scalable multi-tiered applications to reduce the latency, and provide resiliency as needed. However, this can create additional challenges for back-end transactional database systems, which were designed without elasticity. Despite the efforts of key-value stores like Dynamo [7] and Bigtable [8] to provide scalable access to huge amounts of data, transactional guarantees remain a bottleneck [2].

Cloud services make use of replication to ensure consistent performance and availability. A common trait characterizing the new generation of cloud data platforms is the adoption of weak consistency models, such as eventual consistency, restricted transactional semantics and non-serializable isolation levels. Eventual consistency allows data to be inconsistent among some replicas during the update process, but ensures that updates will eventually be propagated to all replicas. The major problem associated with this approach is the difficulty in maintaining the ACID guarantees, in which consistency is compromised to provide reasonable availability.

Authorization policies describe the conditions under which users should be permitted access to resources and describes the relationships between the system principals, as well as the certified credentials that users must provide to attest to their attributes. The transactional database system is deployed in a highly distributed and elastic system; policies are typically replicated among servers following the eventual consistency model. It therefore becomes possible, based on policy-based authorization system, to make unsafe decisions using stale policies.

Interesting consistency problems can arise as transactional database systems are deployed in cloud environments and use policy-based authorization systems to protect sensitive resources. In addition to handling consistency issues amongst database replicas, we must also handle two types of security inconsistency conditions. First, the system may suffer from *policy inconsistencies* during policy updates due to the relaxed consistency model underlying most cloud services. For example, it is possible for several versions of the policy to be observed at multiple sites within a single transaction, leading to inconsistent (and likely unsafe) access decisions during the transaction. Second, it is possible for external factors to cause *user credential inconsistencies* over the lifetime of a transaction [3]. For instance, a user's login credentials could be invalidated or revoked after collection by the authorization server, but before the completion of the transaction. If the problems associated with policy consistency are not alleviated, the company or individual may face a potential risk. The company may leak information about customers and face harsh penalties and loss of credibility.

Due to inconsistency among the policies across the servers, there is a possibility of transaction abort when a user tries to request with older policy. In this case, a user will not be able to access the data item as the policy doesn't match with the newer policy. A user who earlier had access to the data item will not be able to perform any operation on the data item. Such situations should be avoided.

In addition to handling consistency issues among database replicas, policy inconsistencies and user credential inconsistencies need to be addressed for which possible solutions have been proposed [4,5]. These papers address the data, policy, and credential inconsistency problems in transactional database systems deployed in the cloud. The notion of trusted transactions when dealing with proofs of authorization is defined. Accordingly, it proposes several stringent levels of policy consistency constraints, and present different enforcement approaches to guarantee the trustworthiness of transactions executing on cloud servers. This work also formalized the concept of trusted transactions; transactions that do not violate credential or policy inconsistencies over the lifetime of the transaction. It defines several different levels of policy consistency constraints and corresponding enforcement approaches that guarantee the trustworthiness of transactions executing on cloud servers. The authors used Two-Phase Validation Commit (TPVC) protocol as a solution, which is a modified version of the traditional Two-Phase Commit protocol. It ensures that a transaction is safe by checking policy, credential, and data consistency during transaction execution. The previous work presented several approaches for checking and enforcing consistency during transaction execution. However, it never considered enforcing any level of consistency proactively into the system. The whole cloud system is based on Eventual Consistency and as a result, there is a high chance of rollbacks after spending a considerable time executing a transaction. These rollbacks increase significantly if either the policy updates occur too frequently or the average transaction time is more. Another issue is that their proposed solution is based on traditional 2PC, which is not non-blocking, and thus, any failure in the system will result in a transaction abort.

In our work proposed here, we use a look up table in which the policy versions of the different servers used for authorization is stored and updated on a regular basis, which can be easily retrieved by the transaction manager. To reduce the interdependency among replica servers and maintain strict policies, we integrate it with the concept of Tree-Based Consistency [6] to maintain a tree structure of the nodes where a data item is replicated. By making use of look-up table and the tree based approach, we propose an enhanced version of Two-phase validation commit protocol (TPVC) integrated with the Paxos commit protocol which ensures safety of the transaction by checking policy, credential and data consistency during transaction execution and reduces the performance overhead in case of failures without affecting accuracy and precision during transaction executions. The performance using simulation study shows an increase in transaction throughput and commit ratio compared with recent work [5] with update rate of 1150ms. This is due to increase in the number of commits and decrease in the number of aborts, which are generally caused by lack of consistency and fault tolerance in [5]. Stricter consistency reduces the number of aborts caused due to inconsistencies, and integration of Paxos makes sure that no transaction is aborted due to faults in the system. Thus, we are able to enforce strict-consistency with increased reduction in aborts, which can reduce latency and cost of overhead of acquiring resources and re-executing those distributed transactions again.

2. LITERATURE REVIEW AND PRELIMINARIES

Many database solutions have been proposed for the cloud environment like Dynamo [7], BigTable [8], and Cassandra [9]. Cloud providers lack services that guarantee both data and access control policy consistency across multiple data centers. Such consistency models add a new dimension to the complexity in the design of large-scale applications, and introduce a new set of consistency problems [11]. In [12], the authors presented a model that allows users to express consistency and concurrency constraints on their queries that can be enforced by the DBMS at runtime. On the other hand, [13] introduces a dynamic consistency rationing mechanism, which automatically adapts the level of consistency at runtime. Our work focuses on attaining both data and policy consistency, while both of these works [12,13] and [10] focus on data consistency.

CloudTPS provides full ACID properties with a scalable transaction manager designed for a NoSQL environment [14]. However, CloudTPS is primarily concerned with providing consistency and isolation upon data without regard to considerations of authorization policies. In [19], it provides a cure, a causal consistency model for performance improvement.

The work in [15] proactively ensures that data stored at a particular site conforms to the policy stored at that site. If the policy is updated, the server will scan the data items and throw out any that would be denied based on the revised policy. It is obvious that this will lead to an eventually consistent state where data and policy conform, but this work only concerns itself with local consistency of a single node, not with transactions that span multiple nodes.

The consistency of distributed proofs of authorization has previously been studied, though not in a dynamic cloud environment (e.g., [3]). The authors develop protocols that enable various consistency guarantees to be enforced during the proof construction process to minimize these types of security issues. These consistency guarantees are similar to notions of safe transactions in [4]. However, our work addresses the case in which policies—in addition to credentials—may be altered or modified during a transaction.

In [4], several policy and user credential consistency problems are identified and policy based authorization systems are used to enforce access control. In [4, 5], the confluence of data, policy, and credential inconsistency problems that can emerge using transactional database systems are addressed. Since achieving ACID properties in distributed transactional databases has been extensively studied [3,16], they focused on how to achieve trusted transactions.

Two consistency models have been proposed to handle transactions. A transaction is said to be *View Consistent* if all the servers participating in transaction execution have the same policy version. A transaction is said to be *globally Consistent* if all the servers participating in transaction execution have the same policy version as the latest policy in the entire cloud. *Trusted transactions* are defined as transactions which do not fail to agree with the credential or policy inconsistencies during execution. The term *safe* transaction is defined as a trusted transaction, which satisfies all data integrity constraints imposed by the database management system. In practical, a transaction will commit only if it is safe and any unsafe transaction is forced to rollback.

The authors proposed a solution involving an adaptation of the Two-Phase Commit (2PC) to enforce trusted transactions, which is referred to as Two-Phase Validation Commit (2PVC) protocol. The protocol ensures that a transaction is *safe*, as it ensures policy and credential consistency as well as data consistency. Authors introduced trusted transactions; that do not violate credential or policy inconsistencies over the lifetime of the transaction. The authors then presented a more general term, *safe transactions*, that is used to identify transactions that conform to the ACID properties of distributed database systems *and* are trusted in terms of the validity of the policy evaluation.

3. SYSTEM ASSUMPTIONS AND PROBLEM STATEMENT

3.1 SYSTEM MODEL

The cloud infrastructure assumed here consists of a set of Servers S, at least one *Transaction Manager* (TM) and an optional Master Policy Server. Each server is responsible for hosting a subset D of data items belonging to a specific application domain. The users interact with the system by submitting queries or update requests adhering to ACID properties. That is, each transaction T = q_1, q_2,…q_n is a set of sequential queries/updates and each query $q_i \in$ Q (the set of queries) will act on one or more data items. A TM is responsible for executing queries submitted by users.

TMs coordinate transaction execution across servers. In case of increase system workload, multiple TMs could be invoked for load balancing, but each transaction is handled by one TM. Each TM maintains a lookup table consists of version numbers pertaining to all the data items on a server. This is the major component of the system model introduced to implement strict consistency.

In this model, we assume heart beat signals are sent to the master server by each node to inform about its availability at regular intervals. We use these heart beat signals sent by the data nodes to the master server to update the policy versions in the table structure. These signals are sent in regular intervals to the master server. We also assume the same structure in case of Global transactions, which will be executed over different servers on different clouds. These assumptions do not affect the correctness or the validity of the consistency definitions by the authors in [4,5].

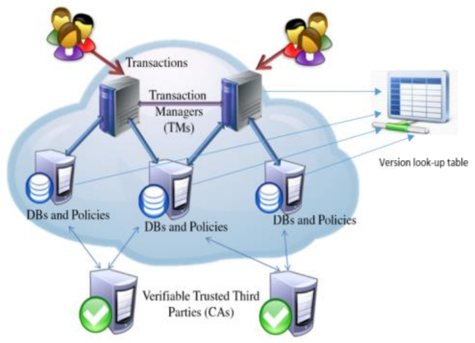

Figure 1. Interaction Among System Components

Further, the transaction manager uses a “Tree-Based Consistency” approach in order to reduce the interdependency among replica servers to minimize the response of cloud databases and maximize the performance of the applications [6]. The transaction manager is responsible to generate a weighted graph G(V,E) with replica servers as vertices V, and connections among these servers as edges E. The Transaction Manager (TM) in addition to assigning the transactions to the servers will (1) periodically communicate with the replica servers, (2) handle server failures, (3) integrates the servers which join after recovery, (4) synchronizes with other replica servers, and (5) maintains service logs used to build the tree. The replica servers hold the data items with them and help serve transactions. Upon calculating the weights of the replica servers and using the look up table, TM builds the consistency tree. The replica server with highest weight will have child nodes connected to it. The number of child nodes connected to it is limited, in order to reduce the load on the replica server. Algorithm such as Dijikstra is used to measure the shortest path between the replica servers and their child nodes. This tree based structure of the replica servers is then used by the TM to select the servers, in which the policy updates has to be carried out based on priority for providing reliable and accurate transaction processing in the cloud.

Let **P** denote the set of authorization policies and each authorization policy $P \in \mathbf{P}$ enforced by a server *si* governing the access to the subset of data items D is defined as P_{si} (D), where the policy P is a mapping such that $P : S \times 2^{D} \rightarrow 2^{R} \times A \times \mathrm{N}$. The value R indicates the set of inference rules to define the authorization policy. A refers to authorization policy of the administrator who is in charge of dictating an application’s policy to the cloud servers, and N is the set of natural numbers used to identify policy version *v*.

3.2 PROBLEM MOTIVATION

The eventual consistency policy to maintain consistency throughout the data replicas is also used to enforce policy consistency in the cloud. This paves way for inconsistent policies across different servers in the cloud. Stringent levels of consistency have to be applied to carry out transactions without any false positive access to the data items in the server.

As we discussed in the previous section, there are four different approaches used to enforce trusted transactions in the cloud [4]. These four models are evaluated on Precision, Accuracy and Performance. The precision is high in case of all the approaches, whereas the accuracy and performance factors vary based on the frequency of the policy updates and application complexity.

Accuracy in deferred and punctual models are low-medium there by maintaining low consistency levels during transaction execution. But these two approaches provide higher performance as the policy consistency check won’t interrupt the execution of the transaction. The accuracy in Incremental and Continuous approaches is medium-high as the policy consistency is checked and enforced at each and every level of transaction execution. Though these approaches provide high levels of security, there is a noticeable latency in transaction execution. The performance is highly affected in global transaction setup where multiple cloud domains are involved.

The delay may also vary based on which level of consistency we are trying to achieve. In case of view consistency, where servers of a single cloud are involved, the delay will be negligible. The complexity increases as we try to achieve global consistency over servers from different clouds. This will cause great delay in transaction execution when using incremental and continuous approaches.

To reduce this delay, we can maintain considerable level of strict policy consistency across servers. We can define a cloud system as *strict policy consistent* if updates to the policies are applied to all the servers, which contains unique copies of data items, which belong to a particular domain. A cloud system C is said to be *strict policy consistent (Σ - Consistent)* if there exists a server S for every data item $D \in Đ$ (set of all data items) with a policy P where $ver(P)=max(ver(Đ))$.

In the following sections, we discuss how to impose strict policy consistency by using policy version look-up table and a tree structure. At the same time, we show how transaction mangers can access this table to reduce the latency issues while implementing the consistency enforcement approaches and maintaining same levels of accuracy and precision.

4. OUR PROPOSED APPROACH

In this section, we present an approach for implementing strict policy consistency using a look-up table and a tree structure. We show how this design will help in increasing policy consistency and performance of the transactions executing in the cloud without compromising on either security or performance. Later, we propose a revised Two-Phase Validation Commit protocol, which uses the same table to reduce the latency in transaction execution due to multiple validation checks while using incremental and continuous approaches.

Strict Policy Consistency: As mentioned in the earlier sections, the Master server or Transaction Manager contains a look-up table, which holds the policy versions of data items that are present on a server. The transaction managers use this table to keep track of the servers whose policies are not updated.

We are assuming that the cloud servers send heart-beat signal informing the master server of its availability. We tweak this functionality so that the servers send policy versions along with the heart-beat signals at regular intervals. This enables transaction managers in selecting the servers to execute the transactions submitted by the users with great accuracy. This helps in avoiding multiple aborts, in case of any inconsistencies between policy versions.

The data items required to execute a transaction are replicated among two or more servers. If we can make sure that at least one of these servers has the most recent policy version then the latency that can occur due to consistency checks can be avoided. Whenever a policy update arrives for specific data items, we will have to update at least one of the servers, which is responsible for replicating the data items associated with the corresponding data items.

The master server or the transaction manager will coordinate with other servers hosting the data items to keep the policies updated across them. Whenever a policy update arrives at a master server, it will start finding all the servers where the respective data items are replicated.

The transaction manager builds the consistency tree and stores information such as total number of servers, connection path between servers, the probability of failure of the connection paths and the failure rate of servers. The following steps are involved while building the tree:

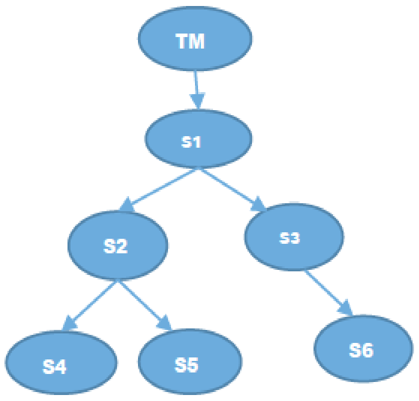

Figure 2. Consistency Tree Of Replica Servers

Connection Graph: Initially, transaction manager prepares the weighted connection graph G (V, E). V stands for the set of vertices, which are the server nodes, hosting the data items. E stands for the set of edges, which are the connections among those servers. Network path reliability is considered as the weight of the edges, since G is a weighted connection graph.

Root of the tree: The transaction manager will choose the most reliable server as the primary server. It acts as the parent node and the other servers connected to the primary server are the child nodes.

Consistency Tree: The transaction manager builds the consistency graph by applying Dijkstra’s single source shortest path algorithm to select the maximum reliable path to every child node. Consolidating all the paths will form the consistency tree. Then based on the tree structure the master server will decide which server will be updated first to ensure availability with strict consistent policy. Thus, the master server or a transaction manager will make at least one copy of a data item is available with latest policy in the cluster.

Update Operation: The transaction manager informs all servers about its immediate descendants and each server is responsible for its own descendants. Each server in the tree stores two flags:

Partially Consistent Flag: Whenever there is an update of policy version, the transaction manager sends a notification containing information about update to all its descendants. All the descendants will send an acknowledgement to the transaction manager indicating the receipt of update information. After receiving the acknowledgement, the parent node updates the policy with latest version and stores the operation sequence number as partially consistent flag. The intermediate nodes propagate the update operations in the same way.

Fully Consistent Flag: When the update operation propagates to the leaf nodes, there will be an empty list of descendants to send the notification. The leaf nodes store the operation sequence number as the partially consistent flag and fully consistent flag. The leaf node sends a notification to its immediate ancestor to store the fully consistent flag. The intermediate node receives the notification from the leaf node to store the fully consistent flag and stores the operation sequence number as fully consistent flag and informs its immediate ancestor. This propagates until the primary server is reached and it cannot commit the transaction until it sets the last operation sequence number as its fully consistent flag.

Failure Recovery: Due to heavy load or technical issues the server might go down at any time. The transaction manager also handles these failures and resolves by recovering it. Three types of situations can arise in this scenario:

Primary Server is Down: The transaction manager initially connects to the parent server. If it discovers that there is a failure in the parent node it communicates with its immediate descendants about its partially and fully consistent flags. If they are the same, the transaction manager will select the maximum reliable server as the root. If not, the transaction manager will find the latest updated servers by querying its immediate descendants for the partially consistent flag. If there is a match, then the transaction manager will select it as the root. The connection graph is reconfigured with the latest updated servers and the consistency tree is built.

Other server or communication path is down: When an intermediate node reports unresponsive behavior of its child node, the transaction manager tries to contact the child node. If the connection is not established, then the child node is considered as down and the consistency tree is reconfigured without the failed node. In this way, a new consistency tree is built and the other servers are informed about the new structure.

If the communication path is down, the transaction manager tries to contact the server through another path and the transaction manager can reconfigure the connection graph with the server and build the consistency tree with the same root.

The tree based consistency approach reduces the dependency between the servers as the communication is between the server and its immediate descendant. Hence, the risk of transaction failure is minimized to a greater extent even in highly unreliable network.

Modified Two-Phase Validation Commit Algorithm The original Two-Phase Validation Commit algorithm can be used to ensure the data and policy consistency of safe transactions. The validation is carried out in two phases; first the voting phase and then the validation phase. During both the phases there are several rounds of communication carried out between the servers and the transaction managers, which creates additional latency.

As we are maintaining strict consistent policy in our system using the design presented above, there is no need to check if the policy is consistent across all the participating servers. The transaction manager can select the servers with latest policies for the data items needed to execute a transaction. This proactive step will drop the additional step of checking the versions of the servers before committing. It is highly unlikely that a policy update will arrive during the lifetime of a transaction. Unless the average execution time of a transaction is higher than the average interval between each update, there is very little chance of such scenarios. Now we propose a new modified version of Two-Phase Validation Commit algorithm (2PVC-Modified), which reduces the latency and increases the performance.

We focus on reducing this latency using policy look-up table that is implemented in the system. Instead of collecting the policy versions from all the servers, we have them listed in the transaction manager. We can look-up this table to check for a server with latest version of policies pertaining to the data items required executing the transaction. By this we will make sure that servers participating in transaction execution is already policy consistent. This ensures increased performance while enforcing safe transactions with great precision and accuracy. This algorithm works very effectively for clouds where the average transaction execution time is less than the average of update intervals.

Algorithm 1: Modified Two-Phase Validation Commit 2PVC

```
1 Send "Prepare-to-Commit" to all participants
2 Wait for all replies (Yes/No, True/False) //[Additional Proof of Authorization]
3 If any participant replied No for integrity check
4         ABORT
5 If any participant replied FALSE for Proof of Authorization
6         ABORT
7 Send "Update" with the largest version number of each policy
8 Re-run the transaction
9 If any new policy updates arrived during transaction execution time
```

```
10        ABORT
11 For participants with old policies
12 Send “Update” with the largest version number of each policy
13 Wait for the update to finish
14 Re-run the transaction
15 Otherwise
16        COMMIT
```

If the average transaction length is greater than the average update interval, the traditional 2PVC may lead to aborts. In clouds, with this kind of scenario, we can implement another level of validation of proofs of authorization at individual servers where the transaction is being executed. This makes sure that the transaction won’t abort at the end after completing all the operations. The Algorithm 1 shows the modified steps where we have added a new level of proof of authorization to be performed at each server in the steps 5-8. If any of the servers returns FALSE for the proof of authorization, then the transaction is aborted and the update is sent to the servers. The updates then are propagated down the consistency trees associated with the data items used to run the transaction. Once the update is finished, the transaction is executed again in the servers.

Advantages: Some of the advantages of using the proposed algorithm over the original algorithm are: (a) Improved policy consistency across the cloud, (b) Minimized trade-off between performance and accuracy, (c) Effective Updates and failure recovery, (d) Reduced communication overhead between servers, (e) Reduced latency during transaction execution, and (f) Minimized aborts due to policy updates.

Non-blocking Paxos Commit

Problem with 2PC: In a transaction commit protocol, if one or more RMs fail, the transaction is usually aborted. For example, in the Two-Phase Commit protocol, if the TM does not receive a *Prepared* message from some RM soon enough after sending the *Prepare* message, then it will abort the transaction by sending *Abort* messages to the other RMs. However, the failure of the TM can cause the protocol to block until the TM is repaired. In particular, if the TM fails right after every RM has sent a *Prepared* message, then the other RMs have no way of knowing whether the TM committed or aborted the transaction. The solution comes in the form of a very popular non-blocking protocol called Paxos Algorithm.

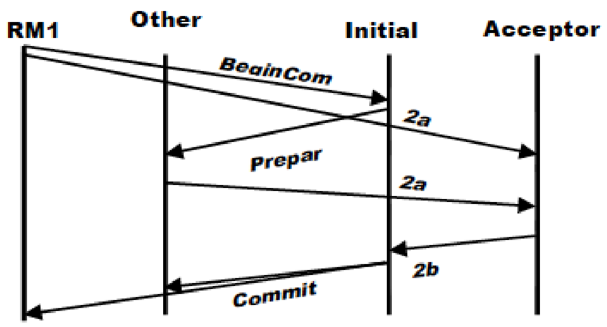

Figure 3. The Message Flow For Paxos Commit In The Normal Failure-Free Case

Paxos Commit: The Paxos algorithm is a popular asynchronous consensus algorithm [17]. Paxos Commit uses a separate instance of the original Paxos consensus algorithm to come to an agreement on the decision each RM makes of whether to prepare or abort—a decision represented by the values Prepared and Aborted. So there is one instance of the consensus algorithm for each RM. The transaction is committed iff each RM's instance chooses Prepared; otherwise the transaction is aborted. A set of acceptors and a leader play the role of Transaction Manager / Coordinator. Let there be N resource managers, and to survive F failures we need 2F+1 acceptors. The message flow for commit in the normal failure free-case is shown below in Table 1. RM1 is the first RM to enter the prepared state, thus initiating the round.

Algorithm 2: Two-Phase Validation with Paxos Commit

```
1 Initiate Paxos commit algorithm along with 2PV using Continuous Proof of Authorization
2 Wait for all replies (True/False)
3 If any acceptor was unable to commit
4     ABORT
5 If any participant replied FALSE for Proof of Authorization
```

```
6       ABORT
7 Send “Update” with the largest version number of each policy
8 Re-run the transaction
9 If any new policy updates arrived during transaction execution time or during Paxos
10      ABORT
11 For participants with old policies
12 Send “Update” with the largest version number of each policy
13 Wait for the update to finish
14 Re-run the transaction
15 Otherwise
16      COMMIT
```

Table 1. Tradeoff Between 2PC and Paxos

Two-Phase Commit	**Paxos Commit** *N* processes agree on a value Tolerates ***F*** faults
3N+1 messages	3N+ 2F(N+1) +1 messages
N+1 stable writes	N+ 2F +1 stable writes
4 message delays	5 message delays
2 stable-write delays	2 stable-write delays

Same algorithm when F=0 and TM=Acceptor

Trade-off discussion: The tradeoff between 2PC and Paxos commit is shown in Table 1. Both the algorithms are same if there is no failure and if one of the acceptors act as TM. Some of the advantages of using the Paxos Commit algorithm over 2PC are: it adds Fault Tolerance at the transaction level, and allows us commit on partial transactions, and allows us to handle Conflicting Policy Versions.

Trade-off discussion based on update interval:

- *Transaction Length < Update Interval*: The first version of the algorithm proposed is useful for these type of cloud systems. The proof of authorization is done at the TM, where it selects the servers for execution based on the consistency tree. So there are less chances of transaction aborts. We just need to make sure that there are no policy updates that arrived during the transaction execution. The log complexity of this version of 2PVC in these systems is 2n+1, which is same as a traditional 2PC. This is essentially same as the 2PC algorithm but an extra step added to check if any policies arrived during the transaction execution.
- *Transaction Length > Update Interval*: In these systems an additional step of validating Proof of authorization can be used when a query arrives at each server. In these systems there is a high probability of getting an update before the transaction execution completes. The additional step we included is useful to avoid aborts that occur at the end of execution. To achieve high accuracy, we can use continuous proof of authorization approach proposed in [4,5]. We can also use other approaches such as punctual and incremental, but the latter is more accurate and precise. The log complexity of this version of 2PVC in these systems varies based on enforcement approach used to validate proof of authorization.

5. SIMULATION DESIGN AND EXPERIMENTS

In the simulation, we had two major requirements i) communication over a network and ii) the ability to spawn threads. We chose the Java simulator from [4]. We generate random transactions of a varying number of database READ or WRITE operations. We make use of the consistency tree generated and select the servers required for executing each transaction. Each *CloudServer* has network path reliability as the weight, which is used to construct the tree. The tree is a binary tree, which will have the most reliable node as the root. The next two reliable nodes will be children of the root and so on. A maximum degree of concurrency can be set in the parameters file. We use Deferred and Continuous methods for proof of authorization.

The Deferred method can be used in a system where the transaction duration is greater than the policy update interval and the Continuous method for systems with transaction duration is lesser than the policy update interval. When a new policy version is issued and stored, it pushes the policy version to all available *cloud servers*. The *policy updater* will update the root of the tree first and then rest of the nodes. This makes sure that we have at least one node available with the latest policy at all times. The rest of the nodes will be updated subsequently.

Simulation 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 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 run in a simulation.
- *minOperations* and *maxOperations* are the minimum and maximum number of operations to be performed in a single transaction. Short transactions are defined as 8-15 operations, medium as 16-30 operations, and long as 31-50 operations.
- *maxServers* is the total number of cloud server instances to be created for the simulation.
- *maxDegree* is the degree of parallelism, or the maximum number of concurrent threads 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 test-bed.
- *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 at the beginning of the simulation. This parameter can be set 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.

Variables and Fixed Parameters: For initial experiments, we determine *transaction cost* in terms of time, successful *commit ratio,* and *throughput* of committed transactions as a function of policy update frequency. We use the same set of variables and fixed parameters for comparison as in [4,5].

To establish a series of policy update frequencies, we began with an initial 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 ran experiments using 1,150ms as the policy update frequency. 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), medium (16-30 operations), and long (31-50 operations) transaction.
- Five validation protocols were used: 2PC, 2PC with local authorization checks, view consistency, global consistency, and Paxos Commit.

With a combination matrix of these variables, we simulate 5 runs of 1,000 transactions for each combination and averaged their results. For the purposes of our experiments, several parameters (see Table 2) were constant throughout experiments but can be varied in future experiments to further explore the performance of all the protocols. Though quite capable of implementation over a variety of networks, for simplicity the experiments will be carried out entirely on a single computer running separate terminal windows for each Java class client/server participant.

Table 2. Fixed Parameters

Variable name	*Value*
maxServers	5
maxDegree	10
latencyMin	5ms
latencyMax	25ms
integrityCheckSuccessRate	1.000
localAuthorizationSuccessRate	0.995

The experiments utilize 5 cloud servers processing a maximum of ten transactions concurrently. The network latency parameters were set to simulate a local area network (LAN) environment. The local authorization success rate was set at 0.995, or 99.5%. Though this rate may initially appear artificially high, it is worth noting that it comes out to, on average, 1 out of every 200 operations failing local authorization. If four long transactions of 50 operations each are run, it is probable that one of them will fail local authorization and aborted.

Performance evaluation: In all the graphs shown below, policy update frequency value is found along the X-axis. We begin with a policy update frequency of 1,150ms and doubled the frequency for each subsequent set of runs up to 36,800ms.

Transaction Cost: We represent transaction cost as the time in milliseconds that a transaction requires for completion. We perform simulations of all protocols. for short transactions, medium transactions, and long transactions. We can find the transaction costs in ms along the Y-axis. For each simulation, we averaged the duration of each successfully committed transaction to provide a value of cost for view consistency, global consistency, and Paxos Commit.

As seen in the graphs (Figure 4 & 5), 2PC (as in [4]) and in our implementation of Paxos Commit have the least transaction costs when compared to View and Global consistency validation. View consistency comes second and as because there is a re-evaluation of proofs in Global consistency during the commit phase (both in 2 PC [4] as well as our Paxos Commit), it has the highest transaction costs. We can see that the transaction cost with our integrated tree consistency with Paxos commit is approximately 40-50 ms higher than with 2PC [4] because there is one more message delay (5 vs. 4) in Paxos commit when compared to 2PC. It also includes the costs for constructing the consistency tree before starting the transaction.

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 view consistency, global consistency, 2PC, and Paxos commit, for short transactions, medium transactions, and long transactions. The transaction throughput values are along the Y-axis.

Transaction throughput (Figure 6 & 7) is higher in ours when compared to [4,5] because the number of commits is more when we use tree consistency and Paxos Commit. There is an approximate increase of 5% more commits when we use tree consistency and Paxos in all the consistency models. The increase is due to executing the transactions on reliable and updated servers rather than selecting random servers as in [4,5], and also due to the fault tolerant nature of Paxos.

Successful Commit Ratio: We represent the commit ratio as the number of successful commits divided by the total number of transactions attempted. We perform simulations of all consistency models for short, medium, and long transactions. We can find the commit ratios along the Y-axis in the graphs below (Figure 8). The commit ratio is constant for 2PC and Paxos as in both we are not enforcing policy consistency checks. In case of view and global consistencies, the view consistency commit ratio is a bit lower then global consistency in both cases.

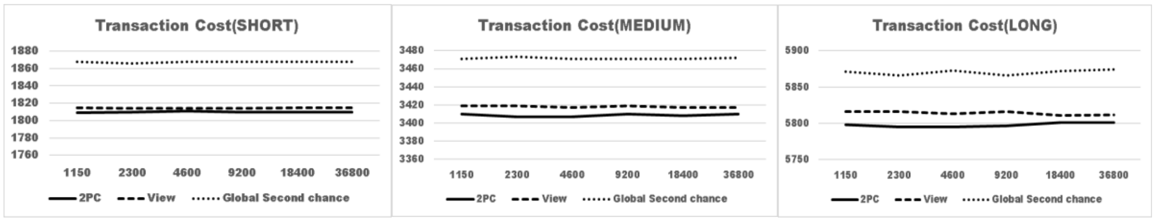

Figure 4. Transaction Cost Without Consistency Tree and 2PC

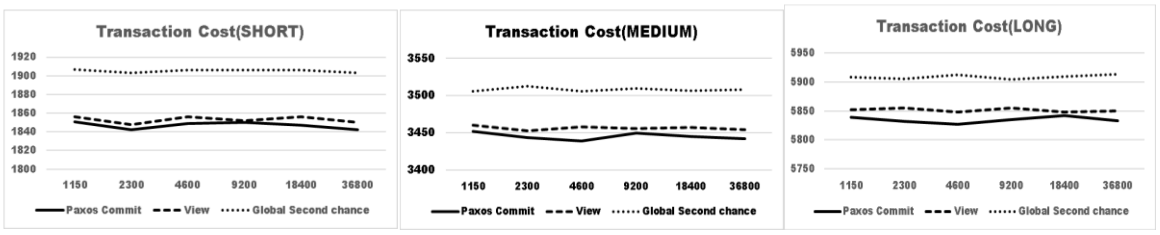

Figure 5. Transaction Cost With Consistency Tree And Paxos Commit

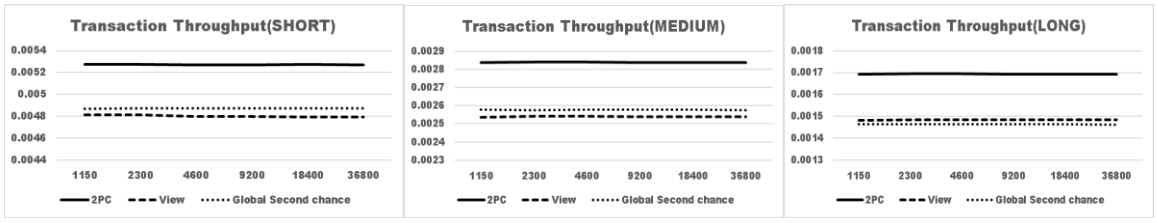

Figure 6. Transaction Throughput Without Consistency Tree And 2PC

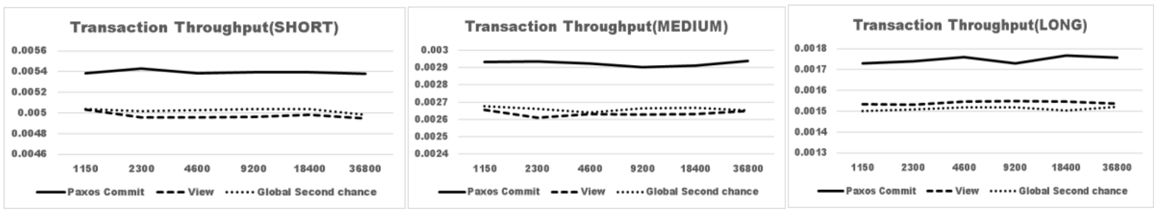

Figure 7. Transaction Throughput With Consistency Tree And Paxos Commit

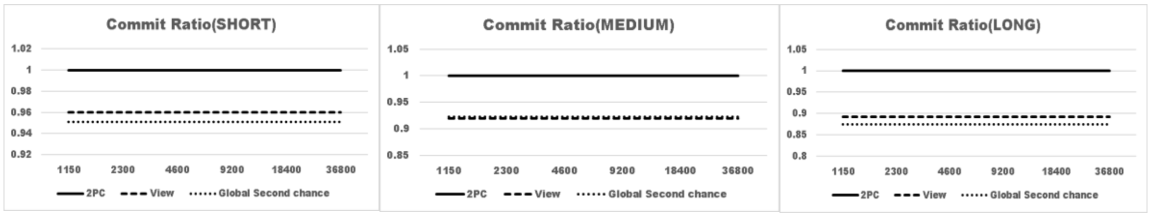

Figure 8. Commit Ratio Without Consistency Tree And 2PC

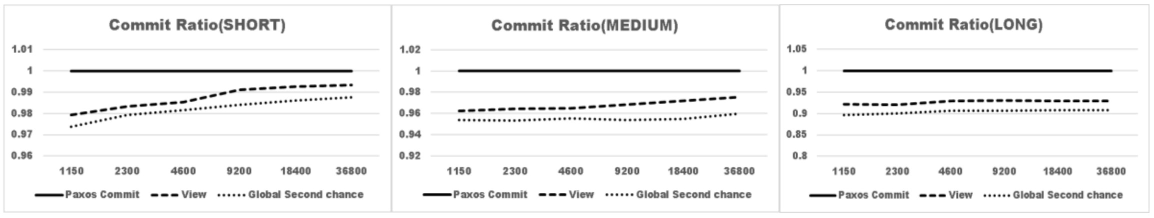

Figure 9. Commit Ratio With Consistency Tree And Paxos Commit

The number of aborts in the second chance global consistency model is less (Figure 8), which leads to increase in the number of committed transactions. The commit ratio is higher in every scenario when compared to [4,5] which does not use tree consistency and Paxos commit. The commit ratio increases steadily as the update interval increases. This is due to less number of aborts at higher intervals due to policy consistencies. We can see the same trend between short, medium and long transactions as well.

6. CONCLUSION AND FUTURE WORK

In this paper, *we* proposed a new approach to enforce *strict policy consistency* among cloud servers integrated with Two-Phase Validation Commit (2PVC) and *Paxos commit,* and *server consistency tree* structure to addresses the performance issues that can arise in cloud based transactional systems. Our approach offers stricter consistency with an increase of only 40-50 ms in execution time, but reduces the number of aborts caused due to policy inconsistencies, and use of Paxos makes sure that no transaction is aborted due to faults in the system. We noticed an increase in the number of commits by increasing the throughput and commit ratio by approximately 5% compared to [4] and still provide strict-consistency.

In future, we will consider consistency rationing to implement more reliable ways of constructing the policy consistency tree and enforcing strict policy consistencies for nested transactions. More work is needed to handle high priority transactions without aborts in case of policy inconsistencies.

REFERENCES

[1] M. Armbrust et al., "Above the clouds: A Berkeley view of cloud computing," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28, Feb. 2009.

[2] S. Das, D. Agrawal, and A. El Abbadi, Elastras: an elastic transactional data store in the cloud, in USENIX HotCloud, 2009.

[3] A. J. Lee and M. Winslett, "Safety and consistency in policy-based authorization systems", in ACM CCS, 2006.

[4] M. K. Iskander, D. W. Wilkinson, A. J. Lee, and P. K. Chrysanthis, "Enforcing policy and data consistency of cloud transactions," in IEEE ICDCS-SPCC, 2011.

[5] Marian K. Iskander, Panos K. Chrysanthis, et al., Balancing Performance, Accuracy, and Precision for Secure Cloud Transactions", IEEE Transactions on Parallel & Distributed Systems, Feb. 2014.

[6] Islam, M.A., Vrbsky, S.V., Tree-Based Consistency Approach for Cloud Databases," in IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom), 2010.

[7] G. DeCandia et al., "Dynamo: amazons highly available key-value store," in ACM SOSP, 2007.

[8] F. Chang, et al., "Bigtable: A distributed storage system for structured data," in USENIX OSDI, 2006.

[9] A. Lakshman and P. Malik, "Cassandra- a decentralized structured storage system," in ACM SIGOPS, Apr. 2010.

[10] Anand Tripathi et al. "A Transaction Model for Management of Replicated Data with Multiple Consistency Levels," IEEE International Conference on Big Data (Big Data), 2015.

[11] W. Vogels, "Eventually consistent," in Communication of ACM, Jan. 2009.

[12] H. Guo, P.-A. Larson, R. Ramakrishnan, and J. Goldstein, "Relaxed currency and consistency: how to say "good enough" in sql," in ACM SIGMOD, 2004.

[13] T. Kraska M. Hentschel, G. Alonso, and D. Kossmann,"Consistency rationing in the cloud: pay only when it matters," in Proc. VLDB, Aug. 2009.

[14] Z. Wei, G. Pierre, and C.-H. Chi, "Scalable transactions for web applications in the cloud," in Euro-Par, Aug. 2009.

[15] T. Wobber, T. L. Rodeheer, and D. B. Terry, "Policy-based access control for weakly consistent replication," in ACM EuroSys, 2010.

[16] P. K. Chrysanthis et al., Recovery and performance of atomic commit processing in distributed database systems, in Recovery Mechanisms in Database Systems. Prentice Hall PTR, 1998.

[17] Leslie Lamport. Paxos made simple. ACM SIGACT News (Distributed Computing Column), 32(4):18{25}, December 2001.

[18] Md Ashfakul Islam, Susan V. Vrbsky, and Mohammad A. Hoque. Performance Analysis of a Tree-Based Consistency Approach for Cloud Databases, IEEE International Conference on Networking and Communications (ICNC), 2012.

[19] D. Akkoorath, A. Z. Tomsic, et al. Cure: Strong semantics meets high availability and low latency. In Proceedings of 36th IEEE ICDCS, June 2016.

II. OPPORTUNISTIC DISTRIBUTED CACHING FOR MISSION-ORIENTED DELAY-TOLERANT NETWORKS

Dileep Mardham[1], Sanjay Madria[1], James Milligan[2] and Mark Linderman[2]

[1]Missouri University of Science and technology, Rolla, MO, USA

[1]Air Force Research Lab, Rome, NY, USA

ABSTRACT

In this paper, a new caching scheme has been proposed which takes into consideration Military/Defense applications of Delay-tolerant Networks (DTNs) where data that need to be cached follows a whole different priority levels. In these applications, data popularity can be defined not only on the basis of request frequency, but also based on the importance like who created and ranked POIs (point of interest) in the data (images), when and where it was created; higher rank data belonging to some specific location may be more important though frequency of those may not be higher than more popular lower priority data. Thus, our caching scheme for DTNs is designed by taking different requirements into consideration for DTN networks for defense applications so that access latency for more important but lesser accessed data is reduced. The performance evaluation shows that our caching scheme reduces the overall access latency, cache miss and usage of cache memory when compared to other caching schemes.

1. INTRODUCTION

Caching data within Delay Tolerant Networks (DTNs) presents many challenges [1,2]. DTNs are, by design, volatile as certain nodes may or may not be available all the time in the network (disconnected or not reachable). So, how should individual nodes, and indeed the network at large choose what data to cache, where to cache, and how much to cache and thereafter, on what basis to do cache replacement?

Caching data within DTNs is discovered as a means for both persisting data locally available in close proximity to reduce latency and accuracy/consistency, and to reduce resource usage (bandwidth and energy) through a reduction in hops to the high priority data sources. DTNs present an even more significant challenge for data persistency as they are, by nature generally isolated and disconnected from a larger network. Therefore, DTNs must cache mission-oriented critical useful data to share at the strategic locations. The simplest solution to cache everything on every device and update it as we make connections to other nodes in the network will increase collisions as nodes will duplicate data, and waste buffer space and energy. Situations can arise where specific nodes within the network become over utilized by caching too many data points due to the uncertainty of data transmission, thus, multiple data copies need to be cached at different locations to ensure better data dissemination. The difficulty in coordinating multiple caching nodes due to the lack of persistence network connectivity in DTN makes it hard to optimize the trade-off between data accessibility (proximate to the nodes) and caching overhead (energy utilization due to the need to disseminate the data across many peers). Thus, each node must decide what data to be cached as nodes cannot design a cooperative caching schemes [1,2] because of not knowing which other nodes they can reach.

Motivation. From all the data caching methods in DTNs we have studied, file popularity [2], social-relationships [4] and cooperative caching [1] have always been the main motivation behind the file selection for caching to satisfy future requests and how cache replacement should be done. Although these methods are efficient in many general scenarios where users transfer picture or other kinds of media files in networks using mobile devices, there are some scenarios where these methods of caching entirely fail to address the requirement.

Consider situations like Mission-oriented networks, or Military/Defense applications where data that need to be cached follows a whole different priority levels. Some data despite of low access frequencies (only few nodes access) should be cached with a highest priority (as those nodes which access are decision-makers) at strategic locations which is completely different from the conventional DTN caching methods. Some other classified data files even with the increasing demand, the data caching should be restricted to a certain number (may be within a certain radius) to prevent possible data theft or other security issues. In addition, some data may not be as popular (based on access frequency), but latency to access it may be such a concern that it must be accessible quickly. Data popularity can be defined not only on the basis of request frequency, but based on who (created) ranked those data, when and where it was created; higher rank data belonging at some specific location may be more important though frequency of those may not be high at this time than lower rank data. These three elements: location, access latency, data creator’s ranking, and popularity of the data based on ranking by intermediate nodes will be the primary qualities use to judge data in how exactly to cache it. Caching can consider the location/rank of the node who created it, resources at that location, perimeter of the nodes, POIs (point of interest) in the data (images), and requests by different nodes and then predicting the caching needs at the next time instant and its location. Thus, a new caching technique for DTNs needed to be designed which takes these different requirements into consideration to fulfil the necessity of defenese applications in DTN networks.

2. RELATED WORK

Below, we review some of the caching schemes which are close to our objective.

What data to cache? The problem of selecting the data to be cached is addressed by the authors in [2, 4]. In [2], the authors proposed a Selective Pushing algorithm which caches data with probability calculated using the Zipf-like distribution. The authors assume that the network is a homogeneous environment where all users share the same mobility or centrality statistic. In [4], they compute the content popularity (relative to the current node) by considering both the frequency and freshness of content requests arriving at a node over a history of request arrivals. Then they select the most popular data to cache at a network location. But whenever there is a change in cached data all the cache nodes have to update data, the social tie and digest tables which consumes time and energy. This article focuses on forming interest groups, which are set of nodes interested in a data item, which is closer to real time scenarios. We consider this approach in our design for selecting data to be cached with less overhead during updates. For example, in

DTN, updates at the friendly nodes are easier, at the few hopes away from the creator are fast, so nodes which experience more frequent updates should be closer to the owner who created.

Where to cache it? This problem is addressed in [1, 4, 5]. In [1], nodes are selected to be NCLs (Network Central Locations) based on three approaches. These NCLs will be used to cache popular data that is needed by the nodes in the network. When NCLs buffer is filled, it will start caching the data on the nearby reliable nodes. Coordination between different NCLs and optimization of data of every NCL can become an endless optimization cycle which consumes a lot of energy if the network is very big and can never be optimized. In [4], clusters/groups of nodes are formed as interest groups. The POIs of the nodes in those groups are cached at the NCLs nearby to those group of nodes. In [5], node groups are formed based on the frequency of contact between the nodes, and the nodes are merged or resigned from the group if the POIs of the nodes in a group changes.

How much to cache? The [3] does a good job of showing the performance differences between optimal and opportunistic 'Edge' caching. The results show that opportunistic caching performs on par with the optimistic caching in all most all the scenarios. We will be using opportunistic caching in our model to cache data at the Edge of the network i.e., near the end user. This will save the memory space across all the nodes in the network avoiding unnecessary data caching.

3. OUR MISSION-ORIENTED NETWORK MODEL

The mission-oriented military network systems consist of tactical vehicles such as Warfighter Information Network-Tactical (**WIN-T**) or High Mobility Multipurpose Wheeled Vehicle (**HMMWV**) which act as Network Central Locations (**NCLs**). These NCLs are always active and they have long-range wireless connections to each other and individually, they may also have secure and reliable satellite connection. We assume here that DTN behavior is observed by the ground troops and Humvees and control base can also have more like MANET connectivity using WiFi.

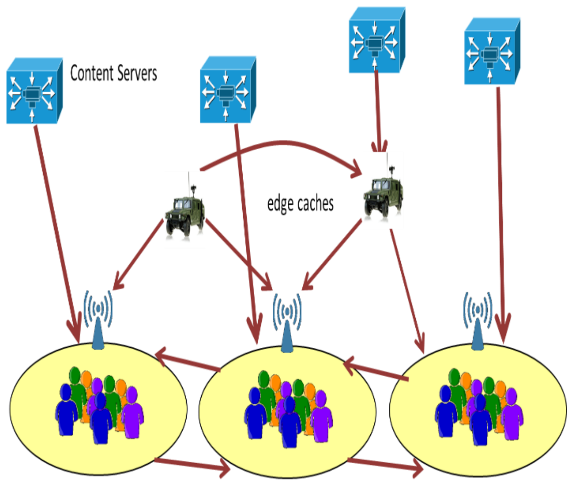

Figure 1. Content Caches

Content Caches. The network consists of devices, which are used to cache data and move in the network independently. These devices can range from tactical vehicles to custom-built systems just for caching. Some of these devices are mobile and some may be stationary. These devices can communicate with each other and exchange cached content between themselves using different wireless communication medium such as Bluetooth, WIFI-direct, etc. These devices will transfer data between them based on the demand around the nodes nearby. Some of these devices act as NCLs in the network and the nodes nearby can request cached data, and then cache data of their own. For example, in a disaster application, raspberry-pi can be a such device, which can be planted at some safe location and can remain stationary.

This network system consists of mobile nodes, which could be combat vehicles or ground forces, which move across the entire network freely. These ground forces need to send/receive messages which each other or should be able to send messages to tactical bases frequently that they are active and should be able to communicate with other tactical bases for further. Note that all the mobile nodes act as a delay-tolerant and use opportunistic ways to communicate.

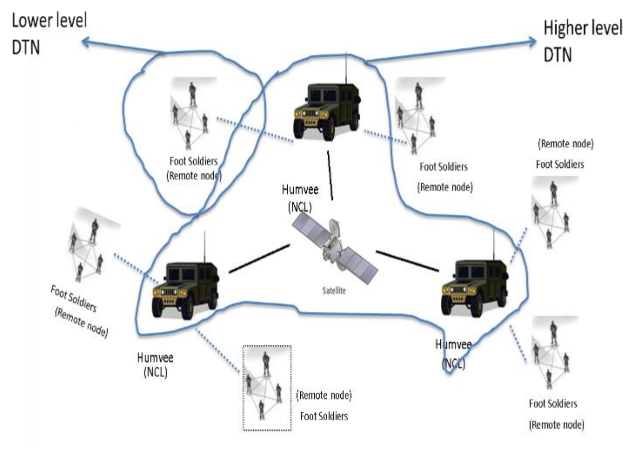
Lower level DTN
Higher level DTN
Foot Soldiers (Remote node)
Humvee (NCL)
Foot Soldiers (Remote node)
(Remote node) Foot Soldiers
Satellite
Foot Soldiers (Remote node)
Humvee (NCL)
Humvee (NCL)
(Remote node) Foot Soldiers
Foot Soldiers (Remote node)

Figure 2. Network Model

4. CACHING USING CONTENT CLASSIFICATION

We need to deal with two main problems: (1) how to define the method for caching by considering different requirements of defense applications, and (2) a method to follow to replace the cached data when the cache memory becomes full. For caching, depending on the situation and requirement of the system, the data can be classified into different categories with different kinds of priority and ranking. In our case where the data needed to be cached belongs to a mission-oriented network such as military. There exists a lot of data that might not be popular based on frequency of access but very important for the specified users like for the decision-makers (high rank nodes). These kinds of data should be cached at nodes, even if the popularity is quite low in the network currently, at some specific locations from where data can be retrieved faster and data is secure/available (not very far from the nodes taking decisions). Some data items may have very high priority as defined by the ranked nodes and its overall impact is adjusted by the combination of rank and priority.

We define a time and spatial decay functions to classify data items and use the same to replace them when cache memory is full. Each node in the network will have a set of data items $\{(W_i, l(v_i) \mid i \in I\}$, where for each item $i \in I$, $W_i > 0$ (varies between 0 and 1) is the weight of the item and $l(v_i) \in V$ is the location of the item cached at node v, where V is a set of nodes in the network. Here, I is the set of data items in the network.

The node defines weight of a data item where the item resides based on some parameters (including the space needed and considering all other data items it hosts) using the formula given below.

We denote by DIST(*l(v), l(u)*) the distance (number of hops) between the location of the two nodes *v* and *u* in the network. A decay function is a non-increasing function and it determines the *weight* of a remote item as a function of its distance and some varying attributes, such as time or number of requests to access the items. The spatial weight of an item i ∈ I at the node v as viewed by a node u ∈ V is

$$d_{uv} = DIST\big(l(v), l(u)\big)$$

The weight of an item of interest at location *v*, w_v^i, might be modelled as attribute values, w_U^i associated with other location *u*, weighted by the inverse of the distance separating locations *u* and *v*, d_{uv} raised to a power β and the difference between the current time T and the time of creation of the item t_i and $t_i \leq T$:

$$w_v^i = r_u^i * p_u^i * \frac{w_u^i}{\left(d_{uv}^i\right)^{\beta}(T - t_i)^{\alpha}}, \beta \geq 0, \alpha \geq 0$$

The exponent β has the effect of reducing the influence of other locations as the distance to these increases. With β=0 distance has no effect, whilst with β = 1 the impact is linear. Values of β >>1 rapidly diminishes the contribution to the expression from locations that are more remote. The exponent α has the same effect as β. There will be some threshold defined for w_v^i to decide about cache replacement of an item, and the space is available.

Note that the weight takes into consideration a priority p (a value of high (0.8), medium (0.6) and low (0.4) between 0 and 1) based on the category of data items defined by the ranker r (a value of high (0.9), medium (0.6) and low (0.3) between 0 and 1). Rank defines the importance of the node (0.9 means most important) and priority defines the importance assigned to the data item (0.8 being most important) by that node. The rank of a node r is fixed, whereas priorities p change based on the node who has assigned the p value. r values do not decay but p decays with distance and/or time as they are defined by the creator. Therefore, the parameters β and α can be adjusted to reflect data items defined by high ranked nodes, and thus, will be decaying slowly with respect to the frequency of access and time. Therefore, as explained earlier, due to slow decay, some items created by high ranked nodes may be cached as needed even if they do not have very high access frequency or created much earlier.

We can use the same decay equations to calculate the decay in weight based on other attributes like number of requests for an item instead of time. The function will be as follows.

$$w_v^i = r_u^i * p_u^i * \frac{w_u^i}{\left(d_{uv}^i\right)^{\beta}(Nt_1 - Nt_2)^{\alpha}}, \beta \geq 0, \alpha \geq 0$$

N_{t2} and N_{t1} are the number of requests received at timestamps t_2 and t_1. The exponent α behaves the same way as it did for time. It will make this function relevant for some of the classes of data items in the network.

We calculate spatial-time/request decaying weight sum for an item, *i*, using the following for a group of nodes *g*, which is a subset of all the nodes, *V*, in the network.

$$w_g^i = \sum_{v \in g} r_v^i * p_v^i * w_v^i, \forall g \subseteq V$$

We categorize the data in the network into five different categories.

Class 0: Emergency Data (high rank, high priority and high access frequency)

The emergency data broadcasts are quite rare but are the most important data. This kind of data can be classified as important and urgent with much higher priority. This data should be stored at each node that is connected to the network to provide instant access to that data, but its expiry will be much faster. In case of low buffer space, the data that needs to be removed is chosen by priority. First the least important data is removed and if space is still needed, then somewhat important but less frequently used data is removed followed by important and frequent data. After the end of emergency broadcasts, all the data deleted should be restored from the neighboring nodes. An example of emergency data is a sandstorm report for next 4 hours or area maps.

The Class 0 data items decays exponentially with time and there may be no spatial decay for these data items. Therefore, the exponent β will be 0 for these items and the exponent α will be 2. The priorities will be 1 for this class of data items, and rank depends on the importance of node who has created the item and given its rank value between 0 and 1. As a result, if a node with high rank has created the Class 0 data item then they will be cached everywhere in the network until the value of the item decays with time/frequency reaches some pre-defined threshold. After that items will no longer be cached in the network or declassified to be one of the lower classes as explained below.

Class I: Highly Important (both high to medium ranked as well as priority) and frequently used data

There are some types of data like log files, artillery count, injured-list, location-images for situational-awareness etc. that are important for certain nodes (military/doctors) and are frequently accessed by only some selective nodes and updated, and to make sure that nothing gets stolen or lost from the battlefield. We assume that soldiers and army vehicles act as nodes, where these kinds of files are accessed by everyone to inform other troops of their status that they are still functional or how many soldiers got injured and their current status. Thus, these kinds of data need to be replicated at every node or most of the nodes (in case if some group of nodes are well connected). The need and priority of this data comes from the importance (ranking and priority) but less frequent usage data. The disadvantage with these data items is to maintain data consistency; all the file copies at all the nodes need to be frequently updated and therefore, the access latency should be minimum. Frequent updating at few hops away drains battery power of the intermediate nodes, which might become crucial for the ground troops to make emergency contacts. Therefore, to minimize these drawbacks, the data should be stored with a few hops from the source called cached-radius, as well as update time limit called update-frequency, and this update-frequency is dynamic within the cached-radius to best fit the situation. These data can consider POIs, and quality of POIs as defined by the image quality and tags. Data, if stored at far reaching nodes may get the updates slowly and therefore, may be of poor quality in terms of POIs.

The Class I items decay both with space and time. As mentioned above, access latency needs to be minimum for these items. We should try to cache the items near the POIs which will be selected based on the spatial weight sum. Their weights will decay exponentially with space but only linearly with time. Thus, the exponents β and α will be 2 and 1, respectively.

Class II: Important and secure (high to low rank and priority) but less frequently used data

Many data files which are classified, and files with limited restriction comes under this category. These files are not accessed all the times since they have limited access thus, make them less popular and are mostly accessed from limited locations. However, caching these kinds of files are very important because they need to be accessed without much latency. Thus, these data files should be stored at NCLs and the data copies can be sent to other nodes upon request. These highest priority files can consume more buffer space like area maps for combat operations, POIs represent aggregated data, and thus, trying to replicate these files at every node reduces the caching efficiency of the network as well as can put the mission in danger. Therefore, trying to store these data at many nodes may pose a security threat. Thus, NCLs should be chosen well which has better caching space and has the best connectivity with other nodes. This kind of data ranked higher and has the highest priority of access by decision-making nodes. This data may be used for example in taking important decisions such as path for troop movement, which qualifies as class 0 data.

The Class II items, just like Class I, decays both with time and space but linearly. Thus, the exponents β and α will be 1 and 1, respectively. Weight sum will be used to choose the NCLs to cache, which includes both priority and ranking of the data items as well.

Class III: Not very important (medium to low rank and medium to low priority) but frequently accessed data

This data that may not look very important from a broader perspective, but plays a key role during a battlefield situation. For example, information about some activities like local news, sudden sandstorm, forest fire, etc. that need to be shared immediately with many nodes. These data become insignificant with time. This problem is similar to caching using file popularity at a certain location. These kinds of data need to be replicated at certain locations where it is crucial even though the data originated at some other location. These data during caching is temporary and should be easily replaceable. The priority of this data is after the first two types of data, Class I and II.

The Class III items just like Class 0 decays with time and don’t use spatial weights. The time decay is linear, and number of requests can be used to decide on caching the items or promoting the item to Class 0. Thus, the exponents β and α will be 0 and 1, respectively. Here also ranking and priory can rebalance the values to take a decision to cache replacement.

Class IV: Not very high ranked data (low rank, and medium to low priority), but accessed on popularity (higher access frequency)

At last there is some data, which is not important for the military networks (low ranked) but needs to be stored for general purposes (medium to low priority). These kind of data is cached using normal caching methods like file popularity, least recently used data etc. The priority of this data is way below the other kinds of data to provide easy access to important kinds of data. The frequency of data updating is kept low to maintain the update costs with very minimum overhead to preserve the battery power.

The Class IV does not decay neither with space nor with time. The exponents β and α will be 0 and 0, respectively. We can use the linear decay based on the number of requests to decide on caching the item on an NCL or a node in the network. The ranking of such data is not high though priority can be very medium to low as defined by nodes.

5. CACHING ALGORITHM

5.1 INITIAL DISTRIBUTION OF CACHING DATA

When data is first created at the source, the data needs to be classified by the source as one of the five categories mentioned above. A source can be either a remote node or a moving NCL. If the source is a remote node, then the data is sent to all its neighboring nodes within its nearest NCL range for further validation. Additionally, while the data is sent to the NCL via remote nodes, it can be re-classified by the NCL. The heuristic for how they re-classify is up to them; for example, if the data is an image, and other nodes are getting similar images with POIs they may raise the classification. The important aspect is that prior to validation, remote nodes can also classify based on their understanding. This can impact/help in NCLs decision, once data reaches there.

It is important to send this data to an NCL because they are expected to have a larger cache size and wireless communication range. Additionally, if they is a Humvee, they may have additional data to influence the classification of the data in question. For example, if a photo of ambulance was taken, and to a solider it may look like an ordinary ambulance. However, if that is stolen by the enemy, it suddenly has become far more important, and that was not possible without the additional data that the base (and its operators) had.

After an initial validation by the NCL and all its remote nodes, if the average class data is classified as class I or class II, then the data is sent to other NCLs for further validation. If most NCLs classify the data as class I, then that data is cached at some of the well-connected NCL locations who have validated it. Restricting the replication of such kind of data prevents data theft by enemy.

If most of the NCLs categorizes the data as class II, i.e., important and Urgent, then this kind of data needs to be further evaluated using local popularity. For example, using the geographical location of this data’s popularity. Finding local popularity and locally caching data is important since it is classified as urgent. To find the local popularity, the NCL nodes which classified the data as class II, are selected and requested for local popularity evaluation. If there are N active local nodes for each NCL and if the data item gets more than N/2 positive responses then that data is popular in that place, and the data is cached in that NCL and some nearby local nodes depending on the hop distance between the nodes. Trying to avoid the involvement of remote nodes until the last step helps in reducing communication and preserving the battery.

If the data is initially classified as class III or class IV by the source and/or its nearest NCL then the data caching follows normal caching methods such as file popularity and other popular methods. If the data is initially classified as class 0 i.e. the data as an emergency message, then the data is sent to all NCLs to cache and after caching the data is evaluated for classification. This process is like class I process but now we cache the data and evaluate later.

Algorithm 1. Distribution of Caching Data

```
Input: Data item created at remote node or a moving NCL
Output: Classify the input data item and cache it in various locations based on its class.
            Classify(data)
            if source is not an NCL then
              Validate(data, source.NCL)
            if data.classification is Class0 then
  Cache(NCLs, data)
            if data.classification is ClassI then
  for ncl in source.NCLs
                Cache(ncl, data)
            if data.classification is ClassII and #NCLs classifying data as Class II >
#NCLs/2 then
               Cache(source.NCL, data)
            If data.clssification is ClassIII or ClassIV then
              Cache(source, data)
```

5.2 CACHE REDISTRIBUTION UPON REQUESTS

Data is cached with the intent that it will be requested later, and if it is found at an intermediate node along the path to the request it can be returned faster, or immediately if locally cached. It is because of that the cache should be re-evaluated upon a fulfilled request.

Once the data is cached after initial distribution, the data should be replicated based on further user requests. When a data item is attempted to be retrieved from a source, the source either has it cached and can return it immediately (latency = 0), or it must hop to other nodes to retrieve the item (latency > 0). The amount of time it takes to retrieve the items via hops is that data item's latency. If the latency priority is lower (meaning the priority is to retrieve that data item in less time that it took to first retrieve it), then we cache it locally.

We send a request to multiple DTN nodes to get an item, and take the latency of the node that returned the data as the maximum latency for that item. We would update the latency on future requests to the item if it changes and cache it on different nodes if needed to provide the reduced access latency. In case, the item requested already has the best latency then we will try to cache based on the popularity in the neighbouring nodes.

Algorithm 2. Cache Redistribution

```
Input: Data item requested by a node
Output: The input data item is either cached at the source of request or cached based on
popularity
if Cached(source, data) then
        return GetCached(source, data)
       startTime = Now
       request = BroadcastRequestFor(data)
// Time passes while waiting for request response from remote node
       endTime = Now
if endTime–startTime > data.maxLatency then
  Cache(source, request)
else
  CacheByPopularity(source, request)
```

5.3 CACHE REPLACEMENT

If a new data item needs to be cached at a node and if the cache space becomes full then some cache data needs to be removed to make room for the new data item. Cache space is divided into 2 levels with dynamic space allocation to each level. One level of cache memory is used to store class I, class II and class 0 types data and second level of cache memory is used for storing class III and class IV data. There is no separate memory space for class 0 data since class 0 data is very rare when compared to other types of data and allocating one more level just for class 0 data might deteriorate the cache performance of the network.

If cache space becomes full and the new data item to be cached is of class I or class II then data from level 2 is removed and is cached in the neighboring nodes. While removing level 2 data, data in level 1 is declassified and moved to level 2 making space for the new item in level 1. If the space is still not sufficient then level 1 data is removed and must be cached in the neighboring nodes. If the new data item is of class III or class IV then level 2 data is removed and cached in neighboring nodes, until we have enough space to cache the new item.

Algorithm 3. Cache Replacement

```
Input: New data item
Output: New data item will replace the cached items either in Level1 or Level2 based on
its class.
if newData.classification ≥ ClassII then
 for data in node do
  if data.classification ≤ ClassIII then
        // some other node may want to cache it, but we need to remove it
    BroadcastToCache(data)
    Remove(node, data)
        if cache.spaceAvailible ≥ newData.size then
          Cache(node, newData)
    break
 // end for
 if not newData in node then
  BroadcastToCache(newData)
```

If data of class 0 needs to be cached and if the cache space is full then level 2 data is removed to replace this data. Unfortunately, if level 1 is filled, then class 2 is removed and cached in neighboring node to accommodate class 0 data. Class 0 data decays faster with time and can be removed later to get back the class II data. Class I data is not removed because this data is critical and will always be cached at well-connected NCL nodes.

6. SIMULATION AND EXPERIMENT

To evaluate our caching algorithm, we used the ONE, a Java DTN simulator [6]. The ONE is suited for DTN simulations, with extensive use in modelling routing and caching algorithms. We use the ONE's default epidemic routing algorithm and extend the *ActiveRouter* Class to cache messages as per our algorithm. The ONE DTN Simulator exposes many different options and variables to tweak in for different results, and most of the results are dependent upon the setup of the nodes. For all our simulations, we will use the same initial nodes with roads as paths taken from a subset of the real world New Dubai map.

The nodes are all chosen in a militaristic setting:

- 6 groups of 10-40 Soldiers each spread across the map.

- 4 Humvees starting on a random point on the map.
- A lone Base is in the center and Humvees will commonly drive through the area.

Soldiers and Humvees move around randomly. All randomness throughout the simulation has the same initial seed so all runs will have the same "randomness". We have 6 groups of soldiers in the simulation. We increase the node count for each group in increments of 10. Soldiers in each group move around in a circle of 500 meters based on dense or sparse areas accordingly in different locations of the map. Soldiers are not tied to roads, but Humvees are. Bases do not move. All nodes simulate a wireless interface akin to Wi-Fi, with varying ranges based on node type. Soldiers generate the data. Humvees and Bases are intended to hold a lot of cached data as they do not have to be carried by mobile devices and can afford to have larger storage and computing power, so they act as NCLs.

All simulations are run for 12 hours in simulation time. Messages are generated and requested randomly throughout the simulation. Messages have a random initial classification, and can be adjusted by remote nodes (Soldiers) and NCLs (Humvees and Bases) based on the decaying weights during the simulation. We let the simulation run for half time (6 hours), and then during the other half aggregate the latencies of fulfilled requests.

We compared 3 versions of our own algorithm to 2 most commonly used caching schemes (cache by popularity and LRU) and a simulation run without any caching. The 3 versions of our algorithm are as follows.

Classification. We classify the data items created on the network and is cached based on the class (as discussed in section 4) assigned by the source and remains same till the end of the simulation.

Dual classification without weights. This version also starts with the classification like the previous but during the simulation uses cache redistribution and replacement based on our algorithm to move around items in the network.

Dual classification with weights. This is same as the previous version of our algorithm but also includes decaying weights which are calculated as discussed in the previous section 4. These weights are used to declassify items during cache redistribution and replacement parts of our algorithm.

Table 1. Simulation Parameters

Parameters		Values
No of Soldier Groups		6
Soldier Count per Group		10 - 40
Transmission Speed	Soldiers	5 Mbps
	Humvee	10 Mbps
	Base	15 Mbps
Transmission Range	Soldiers	200 meters
	Humvee	500 meters
	Base	1000 meters
Cache Size	Soldiers	1 GB
	Humvee	10 GB
	Base	50 GB
Simulated Time		12 Hours

For cache by popularity, we use the simple case that ONE simulator supports where more "popular" (P) messages have a higher P, where $P = H * N$, with H being the number of hops that message has travelled, and N being the number of times the node, deciding to cache, has seen that message.

We also used ONE's default caching scheme such as Least Recently Used (LRU). By default, with LRU, the least recently used messages are evicted from the end of the cache until buffer space is created for the new messages. Finally, we also ran simulations using the same parameters without any caching to compare and see how a caching scheme affects the access latency in a network.

We poll latencies, for all these 5 scenarios, to see how the caching schemes compare in different mission critical situations. Note that our proposed dual caching scheme can be integrated with schemes like in [2, 4] for environments other than defense networks.

7. RESULT ANALYSIS

All numbers measured are in seconds inside ONE DTN Simulator and 5 runs of each combination is averaged.

The results of the latencies have been presented into two tables, with Table 2 being the recorded average access latencies, for 6 different caching schemes discussed above. Table 3 has the percentage values of cache size used for 3 specific scenarios.

Table 3 seems very useful to draw conclusions about the improvement of access latencies between different caching schemes at different node counts. We can clearly see that the five caching schemes vs. no caching as a base line clearly have a significant speedup. This makes sense as with data cached at more places, more requests can be fulfilled quicker.

Once we examine the differences in the caching schemes, we can clearly see that our dual algorithm out performs all other caching schemes. There is a 58% speedup when we compare our algorithm with LRU and 12% speedup when we compare it with cache by popularity (which is used by most DTN algorithms). We can see a speed up of 20% from dual without weight and 23% from dual with weights when compared to simple classification. So, moving around the cached data based on popularity, and decaying weights shows a significant effect on the access latency.

Only caching based on classification can sometimes be slower than caching with popularity, and that is probably due to our design, which prioritizes highly classified data. Another interesting observation is that although dual with weights version of our caching scheme should make the retrieval of critical items much faster, we only see around a 3% speedup over dual without weights (see Table 2).

Next, we want to see how our algorithm will affect cache hit/miss and cache memory used by each node during the simulation. We can see the cache miss percentage plot in figure 3 for different node counts for dual with and without weights.

Table 2. Access Latencies Across Different Algorithms For Different Node Counts

Soldier Count per Group	**Dual with Weight**	**Dual Without Weight**	**Classification**	**Popularity**	**LRU**	**No Caching**
10	461.90	479.66	604.59	523.00	1119.59	2070.19
20	462.97	468.71	627.27	535.50	1155.85	2110.75
30	465.85	472.94	649.13	556.47	1209.96	2275.59
40	471.33	497.39	684.66	573.96	1314.75	2486.17

Table 3. Cached Size Used Percentage

Soldier Count per Group	**Dual with Weight**	**Dual Without Weight**	**Popularity**
10	41.77%	54.37%	81.34%
20	40.82%	49.79%	76.92%
30	35.30%	46.60%	72.57%
40	31.10%	45.71%	64.49%

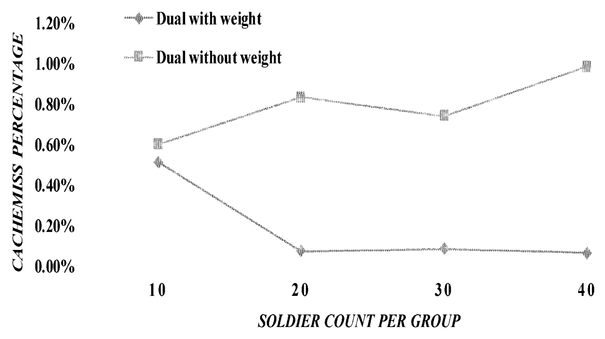

Figure 3. Cache Miss

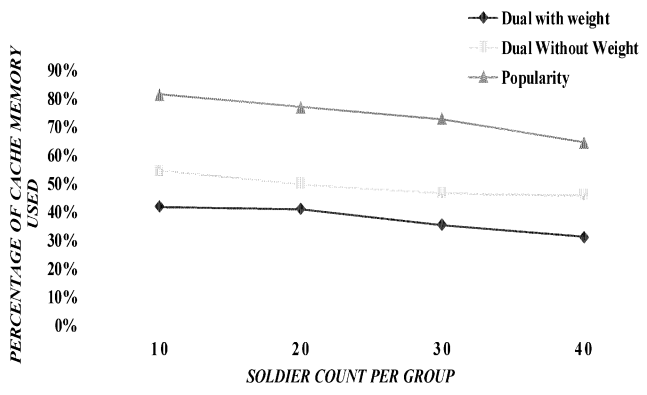

Figure 4. Cache Memory Used

As the number of nodes increases, the decaying weight help the network to make room for the incoming new messages by declassifying the unwanted data items and caching the new messages on NCLs and Base. The algorithm without decaying weights will just keep only the highly classified data leaving no space for the newly arriving low class data. This will result in the increase of cache miss, thereby increasing the percentage of misses.

We can also see the average percentage of cache used on all the nodes in the network in Table 3 and plot in figure 4. We can clearly see that dual algorithm with decaying weights uses less cache memory than the one without weights and popularity caching. Again, the declassification and removal of data items which decays with time and distance helps free up the cache space. Our dual with weights is using less than 50% of the cache space used by popularity caching scheme and 15% less than dual without weights.

From the results above, we can clearly see our dual algorithm with or without weights performs much better than any other common caching scheme in an environment like defense networks. By using decaying weights, we can achieve more consistent access latencies and improve cache hit/miss and reduce the total cache memory used.

8. CONCLUSION

In this paper, we propose a new caching scheme to opportunistically caches data for Mission-oriented Delay-tolerant Networks. The basics of this implementation demand that highly classified data is cached often across the network, even if it is not locally popular, and low classified items are cached based on popularity. Simulations using The ONE DTN Simulator show that our solution does lead to significant latency improvements for data access, over other caching schemes in mission oriented networks. Finally, our caching algorithm performs best by mixing data classification, redistribution and replacement and decaying weights.

Our Dual caching scheme does not currently consider power usage, but instead it tries to best manage the cache, however in the future it could be extended to do so. The ONE could be made to expose the power usage and future research could gleam if the dual scheme also leads to power savings.

REFERENCES

[1] W. Gao; G. Cao; A. Iyengar, M. Srivatsa. “Cooperative Caching for Efficient Data Access in Disruption Tolerant Networks” IEEE Transactions on Mobile Computing, vol./13, no. 3, pp.611- 625, March 2014.

[2] Tiance Wang, Pan Hui, Sanjeev R. Kulkarni, Paul Cuff “Cooperative Caching based on File Popularity Ranking in Delay Tolerant Networks”. CoRR, 2014.

[3] A. Dabirmoghaddam, M. Barijough, and J. Garcia-Luna-Aceves. "Understanding optimal caching and opportunistic caching at the edge of information-centric networks." in ACM Proceedings of the 1st international conference on Information-centric networking, pp. 47-56., 2014.

[4] Tuan Le; You Lu; Gerla, M., "Social caching and content retrieval in Disruption Tolerant Networks (DTNs)," International Conference on in Computing, Networking and Communications (ICNC), pp.905-910, 16-19 Feb. 2015.

[5] Cabaniss, R.; Madria, S., "Content distribution in Delay-Tolerant Networks using social context," in Wireless and Mobile Networking Conference (WMNC), 2014 7th IFIP, vol., no., pp.1-8, 20-22 May 2014.

[6] Ari Kernen, Jrg Ott, and Teemu Krkkinen. 2009. The ONE simulator for DTN protocol evaluation. In Proceedings of the 2nd International Conference on Simulation Tools and Techniques (Simutools ’09). ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Article 55, 10 pages.

3. CONCLUSION

We proposed a new approach to enforce *strict policy consistency* among cloud servers which reduces the number of aborts caused due to policy inconsistencies, and faults in the system. The proposed approach increases the number of commits by increasing the throughput and commit ratio by approximately 5% and still provide strict-consistency. we will consider consistency rationing to implement more reliable ways of constructing the policy consistency tree. Future work includes enforcing strict policy consistencies for nested transactions and ways to handle high priority transactions without aborts in case of policy inconsistencies.

We propose a new caching scheme to opportunistically caches data for Mission-oriented DTNs. Our implementation demands that highly classified data should be cached, even if it is not locally popular. Our solution leads to significant latency improvements for data access, over other caching schemes in mission oriented networks. Our Dual caching scheme does not currently consider power usage, but instead it tries to best manage the cache, however in the future it could be extended to do so.

REFERENCES

[1] M. Armbrust et al., “Above the clouds: A Berkeley view of cloud computing,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28, Feb. 2009.

[2] S. Das, D. Agrawal, and A. El Abbadi, Elastras: an elastic transactional data store in the cloud, in USENIX HotCloud, 2009.

[3] A. J. Lee and M. Winslett, “Safety and consistency in policy-based authorization systems”, in ACM CCS, 2006.

[4] M. K. Iskander, D. W. Wilkinson, A. J. Lee, and P. K. Chrysanthis, “Enforcing policy and data consistency of cloud transactions,” in IEEE ICDCS-SPCC, 2011.

[5] Marian K. Iskander, Panos K. Chrysanthis, et al., Balancing Performance, Accuracy, and Precision for Secure Cloud Transactions", IEEE Transactions on Parallel & Distributed Systems, Feb. 2014.

[6] Islam, M.A., Vrbsky, S.V., Tree-Based Consistency Approach for Cloud Databases," in IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom), 2010.

[7] G. DeCandia et al., “Dynamo: amazons highly available key-value store,” in ACM SOSP, 2007.

[8] F. Chang, et al., “Bigtable: A distributed storage system for structured data,” in USENIX OSDI, 2006.

[9] A. Lakshman and P. Malik, “Cassandra- a decentralized structured storage system,” in ACM SIGOPS, Apr. 2010.

[10] Anand Tripathi et al. "A Transaction Model for Management of Replicated Data with Multiple Consistency Levels," IEEE International Conference on Big Data (Big Data), 2015.

[11] W. Vogels, “Eventually consistent,” in Communication of ACM, Jan. 2009.

[12] H. Guo, P.-A. Larson, R. Ramakrishnan, and J. Goldstein, “Relaxed currency and consistency: how to say ”good enough” in sql,” in ACM SIGMOD, 2004.

[13] T. Kraska M. Hentschel, G. Alonso, and D. Kossmann,“Consistency rationing in the cloud: pay only when it matters,” in Proc. VLDB, Aug. 2009.

[14] Z. Wei, G. Pierre, and C.-H. Chi, “Scalable transactions for web applications in the cloud,” in Euro-Par, Aug. 2009.

[15] T. Wobber, T. L. Rodeheer, and D. B. Terry, “Policy-based access control for weakly consistent replication,” in ACM EuroSys, 2010.

[16] P. K. Chrysanthis et al., Recovery and performance of atomic commit processing in distributed database systems, in Recovery Mechanisms in Database Systems. Prentice Hall PTR, 1998.

[17] Leslie Lamport. Paxos made simple. ACM SIGACT News (Distributed Computing Column), 32(4):18{25}, December 2001.

[18] Md Ashfakul Islam, Susan V. Vrbsky, and Mohammad A. Hoque. Performance Analysis of a Tree-Based Consistency Approach for Cloud Databases, IEEE International Conference on Networking and Communications (ICNC), 2012.

[19] D. Akkoorath, A. Z. Tomsic, et al. Cure: Strong semantics meets high availability and low latency. In Proceedings of 36th IEEE ICDCS, June 2016.

[20] W. Gao; G. Cao; A. Iyengar, M. Srivatsa. “Cooperative Caching for Efficient Data Access in Disruption Tolerant Networks” IEEE Transactions on Mobile Computing, vol. 13, no. 3, pp.611- 625, March 2014.

[21] Tiance Wang, Pan Hui, Sanjeev R. Kulkarni, Paul Cuff “Cooperative Caching based on File Popularity Ranking in Delay Tolerant Networks”. CoRR, 2014.

[22] A. Dabirmoghaddam, M. Barijough, and J. Garcia-Luna-Aceves. "Understanding optimal caching and opportunistic caching at the edge of information-centric networks." in ACM Proceedings of the 1st international conference on Information-centric networking, pp. 47-56., 2014.

[23] Tuan Le; You Lu; Gerla, M., "Social caching and content retrieval in Disruption Tolerant Networks (DTNs)," International Conference on in Computing, Networking and Communications (ICNC), pp.905-910, 16-19 Feb. 2015.

[24] Cabaniss, R.; Madria, S., "Content distribution in Delay-Tolerant Networks using social context," in Wireless and Mobile Networking Conference (WMNC), 2014 7th IFIP, vol., no., pp.1-8, 20-22 May 2014.

[25] Ari Kernen, Jrg Ott, and Teemu Krkkinen. 2009. The ONE simulator for DTN protocol evaluation. In Proceedings of the 2nd International Conference on Simulation Tools and Techniques (Simutools ’09). ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Article 55, 10 pages.

VITA

Dileep Kumar Mardham is from Tirupati, India. He received distinction in Bachelor of Engineering degree in Computer Science and Engineering from Jawaharlal Nehru Technological University, India in 2011. After working at one of the top five software companies in Hyderabad for three years, he has been a graduate student in the Computer Science Department at Missouri University of Science and Technology since August 2015 and worked as a Graduate Research Assistant under Dr. Sanjay Kumar Madria from January 2016 to December 2017 and as a Graduate Teaching Assistant for the Computer Science Department. He received his Master's in Computer Science at Missouri University of Science and Technology in May 2018.