Enforcing Policy and Data Consistency of Cloud Transactions

Marian K. Iskander Dave W. Wilkinson Adam J. Lee Panos K. Chrysanthis
Department of Computer Science, University of Pittsburgh
{marianky, dwilk, adamlee, panos}@cs.pitt.edu

Abstract—**In distributed transactional database systems deployed over cloud servers, entities cooperate to form proofs of authorizations that are justified by collections of certified credentials. These proofs and credentials may be evaluated and collected over extended time periods under the risk of having the underlying authorization policies or the user credentials being in inconsistent states. It therefore becomes possible for a policy-based authorization systems to make unsafe decisions that might threaten sensitive resources. In this paper, we highlight the criticality of the problem. We then present the first formalization of the concept of *trusted transactions* when dealing with proofs of authorizations. Accordingly, we define different levels of policy consistency constraints and present different enforcement approaches to guarantee the trustworthiness of transactions executing on cloud servers. We propose a Two-Phase Validation Commit protocol as a solution, that is a modified version of the basic Two-Phase Commit protocols. We finally provide performance analysis of the different presented approaches to guide the decision makers in which approach to use.**

Keywords-**Cloud databases, authorization policies, consistency, distributed transactions, atomic commit protocol**

I. Introduction

Cloud computing has recently emerged as a computing paradigm in which storage and computation can be outsourced from organizations to next generation data centers hosted by companies such as Amazon, Google, Yahoo, and Microsoft. Such companies have gained a remarkable success by providing multiple services and paradigms referred to as Infrastructure as a Service (IaaS), Database as a Service (DaaS), and Software as a Service (SaaS). This frees organizations that rely on the cloud from requiring expensive infrastructure and expertise in-house, and instead make use of cloud providers to maintain, support, and broker access to high-end resources. From an economics perspective, cloud consumers can save huge IT capital investments and be charged on the basis of a pay-only-for-what-you-use pricing model.

One of the most appealing aspects of cloud computing is its elasticity, which provides an illusion of infinite, on-demand resources [1]. This elasticity provides 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].

To provide scalability and elasticity, cloud services often make heavy use of replication to ensure consistent performance and availability. As a result, many cloud services rely on the notion of eventual consistency when propagating data throughout the system. This consistency model is a variant of weak consistency that allows data to be inconsistent among some replicas during the update process, but ensures that updates will eventually be propagated to all replicas. This makes it difficult to strictly maintain the ACID guarantees in the face of data replication over large geographic distance, as the 'C' (consistency) part of ACID is sacrificed to provide reasonable availability [3].

In systems that host sensitive resources, accesses are protected via authorization policies that describe the conditions under which users should be permitted access to resources. These policies describe relationships between the system principals, as well as the certified credentials that users must provide to attest to their attributes. In a transactional database system that is deployed in a highly distributed and elastic system such as the cloud, policies would typically be replicated—very much like data—among multiple sites, often following the same weak or eventual consistency model. It therefore becomes possible for a policy-based authorization system to make unsafe decisions using stale policies.

Interesting consistency problems can arise as transactional database systems are deployed in elastic 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 being made during the transaction. Second, it is possible for

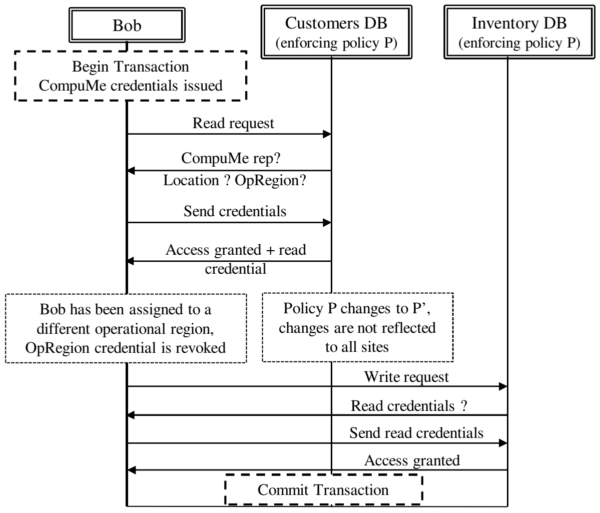
Bob
Customers DB (enforcing policy P)
Inventory DB (enforcing policy P)
Begin Transaction
CompuMe credentials issued
Read request
CompuMe rep?
Location ? OpRegion?
Send credentials
Access granted + read credential
Bob has been assigned to a different operational region, OpRegion credential is revoked
Policy P changes to P', changes are not reflected to all sites
Write request
Read credentials ?
Send read credentials
Access granted
Commit Transaction

Fig. 1. A graphical representation of Bob's interaction with the system

external factors to cause *user credential inconsistencies* over the lifetime of a transaction [4]. 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. *In this paper, we address this confluence of data, policy, and credential inconsistency problems that can emerge as transactional database systems are deployed to the cloud.* In doing so we make the following contributions:

- We highlight, for the first time in the literature, the criticality of the problem. We then describe possible problems that can arise in the context of a motivational example (Section II).
- We present the first formalization of the concept of *trusted transactions*. Trusted transactions are those transactions that do not violate credential or policy inconsistencies over the lifetime of the transaction. We then present a more general term, *safe transactions*, that we use to identify transactions that conform to the ACID properties of distributed database systems *and* are trusted in terms of the validity of the policies evaluation (Section III).
- Since achieving ACID properties in distributed transactional databases has been extensively studied [5], [6], our focus in this paper is how to achieve trusted transactions. Accordingly, we define different levels of policy consistency constraints as well as different enforcement approaches to guarantee the trustworthiness of transactions executing on cloud servers (Sections IV).
- We propose 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 resulting protocol ensures that a transaction is *safe*, as it ensures policy and credential consistency along with ensuring data consistency (Section V).
- We present a performance analysis study of our proposed approaches to guide the decision makers in which approach to use in practice (Section VI).

Finally, Section VII describes previous related work, while Section VIII presents our conclusions.

II. Motivating Example

Figure 1 illustrates one case in which inconsistencies among policies and/or credentials can cause unsafe authorizations to occur. In this scenario, Bob is attempting to access a customer database that requires him to prove that he is a sales representative for his company (CompuMe), that he is currently assigned to sell within a particular geographical region, and that he is currently located within that region. Bob constructs such a proof of authorization, which is then verified by the customer database. The database then permits access, and returns Bob a credential indicating that he is permitted to read from the database.

Bob is then assigned to a different operational region, and the policy protecting resources within CompuMe is changed. However, since CompuMe makes use of an eventual consistency model for propagating policy changes, this new policy is not immediately propagated to all databases. When Bob attempts to access the inventory database, he is required to either satisfy (the original) policy, or present a previously-issued "read" credential indicating that the policy was satisfied. Bob presents his read credential, and is then granted access to the database. Note that Bob's second access was granted (i) using an old version of the access control policy and (ii) under the false pretense that Bob was still assigned to a valid operational region.

In general, many such problems can be encountered due to policy and/or credential inconsistencies. If any problems of policy consistency are not alleviated, the company or individual may suffer. The company may leak information about customers and face harsh penalties and loss of credibility. The individual may lose commission or other benefits for a sale.

III. System Assumptions and Problem Definition

A. System Model

We assume a cloud infrastructure consisting of a set of $\mathcal{S}$ servers, where each server is responsible for hosting a subset D of all data items $\mathcal{D}$ belonging to a specific application domain ($D \subset \mathcal{D}$). Users interact with the system by submitting queries or update requests encapsulated in ACID transactions. Transactions submitted to the system are first forwarded to a *Transaction Manager* (TM) that distributes the queries to the involved servers and coordinates their execution. Multiple TMs could be invoked as the system workload increases for load balancing, but each transaction is handled by only one TM. We denote each transaction as $T = q_1, q_2, \ldots, q_n$, where

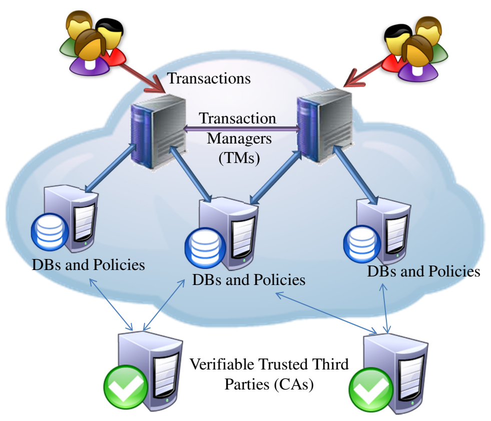
Transactions
Transaction Managers (TMs)
DBs and Policies
DBs and Policies
DBs and Policies
Verifiable Trusted Third Parties (CAs)

Fig. 2. Interaction among the system components

$q_i \in Q$ is a single query/update belonging to the set of all queries Q. The start time of each transaction is denoted by $\alpha(T)$, and the time at which the transaction finishes execution and is ready to commit is denoted by $\omega(T)$. Without loss of generality, we assume that queries belonging to a transaction execute sequentially and without any restriction on the servers. That is, two queries of the same transaction may execute on the same server at different instances in time but not concurrently. These assumptions simplify the way we present our model and definitions, but does not affect the correctness or the validity of our consistency definitions.

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

We will refer to the set of all credentials as $\mathcal{C}$. We assume that users' certified credentials are issued by an arbitrary number of *Certificate Authorities* (CAs) that exist in the system. We assume that each CA offers an online method that allows any server to check the current status of a particular credential issued by the CA [7]. Given a credential $c_k \in \mathcal{C}$, $\alpha(c_k)$ denotes the time at which the credential was issued by the CA, while $\omega(c_k)$ denotes the credential expiration time. Credentials can prematurely expire if they are revoked, and they can only be revoked by the issuing CA. Different cloud servers can also issue access credentials that act as *capabilities* allowing the user to continue submitting queries to other servers during the transaction lifetime (as was the case with Bob's read credential in Section II). Servers can verify access credentials issued by each other.

We assume that a transaction does not fork to other sub-transactions. This assumption is necessary to simplify the proof of correctness of our proposed scheme as presented later in Section V. Transactions also do not externalize any data items to the users until commit time. Figure 2 illustrates the interaction among the different system components.

We now present a formal definition of a proof of authorization. Let $f_{s_i} = \langle q_i, s_i, P_{s_i}(m(q_i)), t_i, C\rangle$, 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_{s_i} 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 $m : Q \rightarrow 2^D$, 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 querier to complete the proof of authorization such that $C \subseteq \mathcal{C}$.

Let $\mathcal{F}$ denote the set of all proofs of authorizations, and the set TS contain all possible timestamps. The validity of each proof of authorization $f \in \mathcal{F}$ at time instance t is evaluated using the predicate $eval(f, t)$ such that $eval : \mathcal{F} \times TS \rightarrow \mathbb{B}$. 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 time $\alpha(c_k)$ has passed, and (iv) the time $\omega(c_k)$ 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 $t_i \leq t' \leq t$.
2) (The inference rules are satisfiable) A policy is a set of inference rules that are encoded by policy makers to capture systems 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.

B. Problem Definition

As we mentioned earlier, a *safe transaction* is a more general term for a transaction that satisfies the correctness properties of proofs of authorizations, in which case we refer to as a *trusted transaction*, and also satisfies the data integrity constraints. Since data integrity and consistency has been extensively studied within the distributed database community, in this section we focus on defining the new concept of trusted transaction.

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 the different consistency levels that could be achieved within a transactions lifetime. These consistency models strengthen the trusted transaction definition 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 the different proofs of authorizations evaluated during the lifetime of a particular transaction.

Definition 1: (View) A transaction's *view* V^T is the set of proofs of authorizations observed during the lifetime of a transaction $[\alpha(T), \omega(T)]$ and defined as $V^T = \{f_{s_i} \mid f_{s_i} = \langle q_i, s_i, P_{s_i}(m(q_i)), t_i, C\rangle \wedge q_i \in T\}$. ◆

Following from Definition 1, a transaction's view is built incrementally as more proofs of authorizations are being evaluated by servers during the transaction execution.

We now present two increasingly more powerful definitions of consistencies within transactions.

Definition 2: (View Consistency) A view $V^T = \{\langle q_i, s_i, P_{s_i}(m(q_i)), t_i, C\rangle, \ldots, \langle q_n, s_n, P_{s_n}(m(q_n)), t_n, C\rangle\}$ 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 $\phi\text{-consistent}(V^T) \leftrightarrow \forall_{i,j} : ver(P_{s_i}) = ver(P_{s_j})$ for all policies belonging to the same administrator A, where function *ver* is defined as $ver : P \rightarrow \mathbb{N}$. ◆

With a view consistency model, the policy versions should be internally consistent across all servers involved in the transaction. That is, a snapshot of the system is what is used to evaluate the decision of a trusted transaction with the servers agreeing among themselves. 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 of the 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.

Definition 3: (Global Consistency) A view $V^T = \{\langle q_i, s_i, P_{s_i}(m(q_i)), t_i, C\rangle, \ldots, \langle q_n, s_n, P_{s_n}(m(q_n)), t_n, C\rangle\}$ *global consistent*, or ψ-consistent, if V^T satisfies a predicate ψ-consistent that places constraints on the versioning of the policies such that $\psi\text{-consistent}(V^T) \leftrightarrow \forall_i : ver(P_{s_i}) = ver(\mathcal{P})$ for all policies belonging to the same administrator A, and function *ver* follows the same aforementioned definition, while $ver(\mathcal{P})$ refers to the latest policy version. ◆

With a global consistency model, policies used to evaluate the proofs of authorizations during a transaction execution among S servers should match the latest policy version among the entire policy set $\mathcal{P}$, for all policies enforced by the same administrator A.

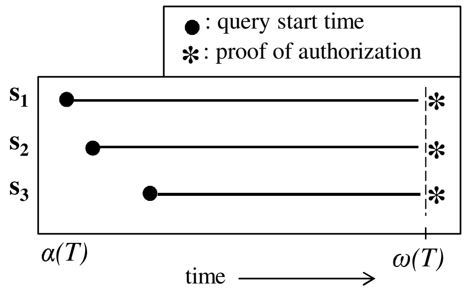

Fig. 3. Deferred proofs of authorization

Given the above definitions, we now have a precise vocabulary for defining the conditions necessary for a transaction to be asserted as "trusted".

Definition 4: (Trusted Transaction) Given a transaction $T = \{q_1, q_2, \ldots, q_n\}$ and its corresponding view V^T, T is *trusted* iff $\forall_{f_{s_i} \in V^T} : eval(f_{s_i}, t)$, at some time instance $t : \alpha(T) \leq t \leq \omega(T) \wedge$ (ϕ-consistent(V^T) $\vee\psi$-consistent(V^T)) ◆

Following from Definition 4, a safe transaction is a transaction that is both trusted and satisfies the data integrity constraints. A safe transaction is allowed to commit, while an unsafe transaction is forced to rollback.

IV. Trusted Transactions Enforcement

In this section, we present several approaches for enforcing trusted transactions. We show that each approach offers different guarantees during the course of executing transactions over cloud servers. This indicates that, like many other aspects of distributed proving and consistency guarantees, the choice of which approach to use is likely to be a strategic choice made independently by each application. We delay all discussions pertaining to the trade-offs to be considered when making the choice until section VI-B. We now present our approaches starting from the most permissive and gradually all the way to the least permissive approach.

A. Deferred Proofs of Authorization

Definition 5: (Deferred Proofs of Authorization) Given a transaction T and its corresponding view V^T, T is trusted under the deferred proofs of authorization approach, iff at commit time $\omega(T)$, $\forall_{f_{s_i} \in V^T} : eval(f_{s_i}, \omega(T)) \wedge$ (ϕ-consistent(V^T) $\vee\psi$-consistent(V^T)) ◆

Deferred proofs of authorization present an optimistic system with weaker authorization guarantees, since different portions of the transaction are allowed to execute without being validated against the access policies. It is only at commit time when the proofs of authorizations are evaluated

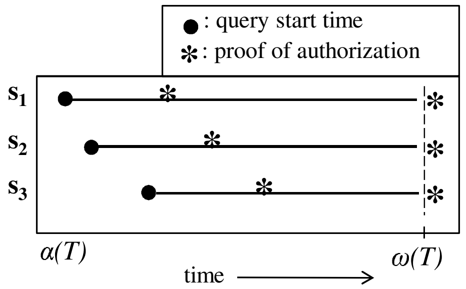

Fig. 4. Punctual proofs of authorization

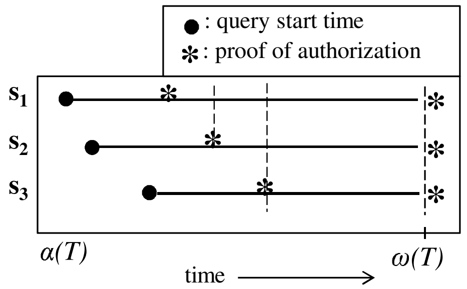

Fig. 5. Incremental Punctual proofs of authorization

simultaneously, that is, the proof trees are constructed and credentials are syntactically and semantically validated at the end point of the transaction. Accordingly, a decision is made whether the transaction is trusted or not. Note that a deferred proof of authorization has the choice of enforcing either view or global consistency from Definitions 2 and 3 at commit time. The choice of which consistency level to enforce is a choice to be made by applications based on the required trust level.

Figure 3 shows a scenario where three servers s_1, s_2, and s_3 are involved in the execution of a transaction. The horizontal lines define the transaction lifetime, and the dots represent the arrival time of a query to each server. The stars indicate the times at which each server validates a proof of authorization. The vertical dotted line represents an enforcement of either a view consistency among the three servers or a global consistency between all servers. As shown in this figure, the deferred proofs of authorizations requires only that proofs are evaluated at the transaction commit time using either view or global consistency.

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 if it violates the trusted transaction condition.

B. Punctual Proofs of Authorization

Definition 6: (Punctual Proofs of Authorization) 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 $t_i : \alpha(T) \leq t_i \leq \omega(T)$ $\forall_{f_{s_i} \in V^T} : eval(f_{s_i}, t_i) \wedge eval(f_{s_i}, \omega(T)) \wedge$ (ϕ-consistent(V^T) $\vee \psi$-consistent(V^T)) ◆

Punctual proofs of authorizations present a more proactive approach in which the proofs of authorizations are evaluated instantaneously whenever a query is being handled by a server. A re-evaluation of all the proofs of authorizations at commit time is mandatory to ensure that throughout the window of execution of the transaction, policies were not updated in a way that would invalidate a previous proof, and/or credentials were not invalidated.

Using this approach, early decisions on whether a transaction should proceed or rollback could be made based on instantaneous evaluations of the proofs. Early detections of unsafe transactions can save the system from going into expensive undo operations.

As shown in Figure 4, Punctual proofs of authorizations 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 authorizations are re-evaluated that either view consistency or global consistency are enforced. Hence, and 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 no guarantee that the decision made by that server is valid or invalid. As a consequence, servers might have false negative decisions and deny access to queries, and on the other hand, false positive decisions could also be made. Therefore, we propose two more restrictive approaches, that if combined with global consistency can avoid the false positive and false negative decisions.

C. Incremental Punctual Proofs of Authorization

Before we define the Incremental Punctual proofs of authorization approach, we define a *view instance*, which is a view snapshot at a specific time instance.

Definition 7: (View Instance) A view instance $V^T_{t_i} \subseteq V^T$ is defined as $V^T_{t_i} = \{f_{s_i} \mid f_{s_i} = \langle q_i, s_i, P^A_{s_i}(m(q_i)), t, C\rangle \in V^T \wedge t \leq t_i\}$, $\forall t, t_i : \alpha(T) \leq t \leq t_i \leq \omega(T)$. ◆

Informally, a view instance $V^T_{t_i}$ is the subset of all proofs of authorizations evaluated by servers involved in transaction T up till the time instance t_i.

Definition 8: (Incremental Punctual Proofs of Authorization) 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 $t_i : \alpha(T) \leq t_i \leq \omega(T)$, $\forall_{f_{s_i} \in V^T_{t_i}}$: $eval(f_{s_i}, t_i) \wedge$ (ϕ-consistent($V^T_{t_i}$) $\vee \psi$-consistent($V^T_{t_i}$)) ◆

Incremental Punctual proofs of authorizations develop a stronger conception of trusted transactions in such that a

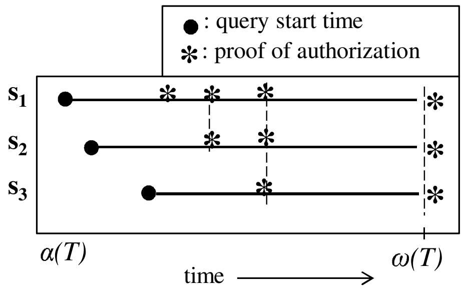

Fig. 6. Continuous proofs of authorization

transaction is not allowed to proceed unless the desired level of the policy consistency at each server is achieved. In Figure 5, at every time instance where a proof of authorization is evaluated a vertical line is drawn to indicate that some consistency level among the servers is required, this could be either a view consistency or a global consistency.

Without loss of generality in Figure 5, if the first server that starts executing the transaction has the latest policy version, in such case it is server s_1, all other servers (s_2, s_3) will be forced to have a consistent view with the first server before they can proceed with evaluating their proofs of authorization. In such a scenario, we have the guarantee that no false positive or false negative authorization decisions will be made by any of those servers. On the other hand, 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 later 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.

Finally, we present the least pervasive approach which we call Continuous proofs of authorizations. In this approach proofs of authorizations evaluated during the transaction execution are re-evaluated at each time instance when a new proof has to be evaluated. That is, the transaction is not allowed to proceed if at any time instance an inconsistency among the policies and/or credentials is captured. Following is the formal definition for this approach.

D. Continuous Proofs of Authorization

Definition 9: (Continuous Proofs of Authorization) A transaction T is declared trusted under the Continuous approach, iff $\forall_{1\leq i\leq n}\forall_{1\leq j\leq i}$: $eval(f_{s_i}, t_i) = true \wedge eval(f_{s_j}, t_i) = true$ $\wedge$ (ϕ-consistent($V^T_{t_i}$) $\vee\psi$-consistent($V^T_{t_i}$)) at any time instance $t : \alpha(T) \leq t_i \leq \omega(T)$ ◆

The stronger guarantees that this approach offers arise from the fact that view and global consistencies are not enough to guarantee that the proofs of authorizations are valid at all times. If credentials are prematurely revoked, (as was the case with Bob's OpRegion credential that was revoked between the two different queries as shown in Section II), a re-evaluation of the proofs of authorization would be necessary to capture such situations. 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. Figure 6 illustrates the Continuous proofs of authorizations.

Once again, the decision of which approach to adopt is to be handed to the policy administrators. As with any trade-off, there is no free lunch, and the stronger the safety guarantees given by an approach, the more the system has to pay in terms of communication and delay overheads. Further discussion of performance issues will be presented in Section VI-B.

V. Implementing Safe Transactions

A safe transaction is a transaction that is both trusted (i.e., satisfies the correctness properties of proofs of authorizations) and database correct (i.e., satisfies the data integrity constraints). In this section, we will first describe an algorithm that provides for trusted transactions. Then, we expand to satisfy safe transactions. Finally, we show how these algorithms can be used to implement the various approaches discussed in Section IV.

A. Two-Phase Validation Algorithm

A common characteristic of our proposed approaches to achieve trusted transactions is the need for policy consistency validation at the end of a transaction. That is, in order for a trusted transaction to commit, its TM needs to determine the consistency of the definitions among the servers participating in the transaction. Toward this, we propose a new algorithm called *Two-Phase Validation* (2PV).

As the name implies, 2PV operates in two phases: the *collection phase* and the *validation phase*. During the collection phase, the TM first sends a Prepare-to-Validate message to each participant. In response to this message, each participant (1) evaluates the proofs for each query of the transaction using the latest policies it has available and (2) sends a reply back to the TM containing the truth value (TRUE/FALSE) of those proofs along with the version number and policy identifier for each policy used in the proof evaluation. Further, each participant server keeps track of its reply (i.e., the state of each query) which include the id of the TM (TM_{id}) and the id of the transaction (T_{id}) to which the query belongs along with a set of policy versions used in the query's authorization (v_i, p_i).

Once the TM receives the replies from all of the participants, it moves on to the validation phase. During this phase, the TM notes any policy version inconsistencies. If all polices are

Algorithm 1: Two-Phase Validation (Coordinator)

```
1  Send “Prepare-to-Validate” to all participants
2  Wait for all replies (a True/False, and a set of policy
     versions for each unique policy)
3  Identify the largest version for all unique policies
4  If all participants utilize the largest version for each
     unique policy
5      If any responded False
6          ABORT
7      Otherwise
8          CONTINUE
9  Otherwise, for all participants with old versions of policies
10     Send “Update” with the largest version number of each
         policy
11     Goto 2
```

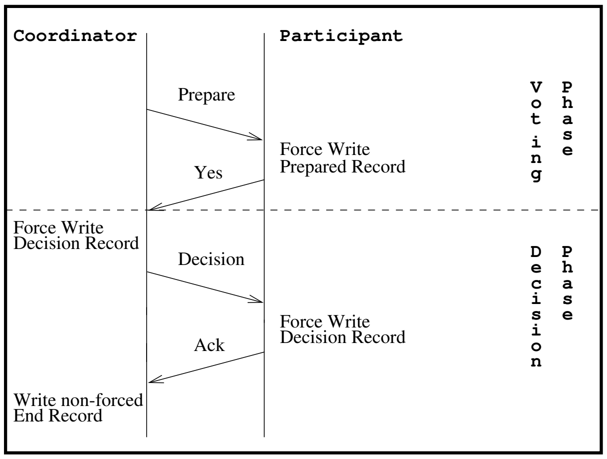

Fig. 7. The basic two-phase commit protocol

consistent, then the protocol honors the truth value where any FALSE causes an ABORT decision and all TRUE causes a CONTINUE decision. In the case of inconsistent policies, the TM identifies the latest policy and sends an Update message to each out-of-date participant with a policy identifier and goes back to the collection phase. In this case, the participants (1) update to the new policy from the server, (2) re-evaluate the proofs and (3) send a new reply to the TM. Algorithm 1 shows the process for the TM.

In the case of view consistency (Definition 2), there will be at most two rounds of the collection phase. A participant may only be asked to re-evaluate a query using a newer policy by an Update message from the TM after one collection phase.

To provide 2PV under global consistency (Definition 3), only minor changes are needed. The global consistent version of the protocol uses something akin to a master server to find the latest policy version. As such, the TM will retrieve this from some known master server in Step 2 and use it to compare against the version numbers of each participant in Step 3.

This master version may be retrieved only once or each time Step 3 is invoked. For the former case, the collection phase may only be executed twice as in the case of view consistency. In the latter case, if the TM is retrieving the latest version every round, global consistency may execute the collection phase many times. This is the case if the policy is updated during the round. While the number of rounds are theoretically infinite, in a practical setting, we assume that this will occur infrequently. The selection of which method depends on the application and the properties of the privacy polices.

Besides being used at commit time, 2PV can be used during the execution of the transaction in the case of Continuous approach. In this case, when a query is to be executed, the TM will (1) execute 2PV to validate authorizations of all queries up to this point, and (2) upon CONTINUE being the result of 2PV, submit the query to be executed at an appropriate server.

B. Two-Phase Validate Commit Algorithm

Although 2PV provides trusted transactions, it does not satisfy the definition of a safe transaction as it does not validate the satisfaction of integrity constraints. Traditionally, integrity constraints in distributed systems are enforced by the *Two-Phase Commit* atomic protocol (2PC), which is a distributed agreement algorithm with two distinct phases: *voting phase* and *decision phase* [8]. There is a central TM that collects the decisions of each participant. In the voting phase, the participants involved in the transaction are polled for their vote on the commit. A YES vote from every participant is interpreted by the TM as a global agreement for a commit. On the other hand, a single NO vote from any participant induces a global rollback. In the decision phase, the TM notifies each participant with the voting decision and waits for an acknowledgment. Figure 7 illustrates the sequence of events of the basic atomic 2PC protocol.

In its basic format, 2PC cannot be used for satisfying safe transactions by combining integrity constraint validation and policy consistency validations because a response of YES (even if it were to suggest both data and policy consistency) would not indicate the version of the policy that the participant used to determine the authorization of the commit. There exists a situation where a participant says YES, when another participant has a fresher policy that would have contradicted the decision of the first participant. However, because of their similarities, we propose to integrate 2PV and 2PC into a new protocol called *Two-Phase Validation Commit* (2PVC), which is used to ensure the data consistency and policy consistency of distributed transactions.

Specifically, 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

Algorithm 2: Two-Phase Validation Commit

```
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
```

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.

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 behavior 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.

C. Discussion

With 2PV and 2PVC, the various proofs of authorization approaches can be easily implemented. Deferred and Punctual (Definitions 5 and 6) are roughly the same. The only difference is that Punctual will return proof evaluations upon executing each query. Yet, this is done on a single server, and therefore, does not need 2PVC or 2PV to distribute the decision. To provide for trusted transactions, both require a commit time evaluation at all participants using 2PVC.

Incremental Punctual (Definition 8) acts just as in the basic Punctual case. However, as queries are executed, the TM must also check consistency within any servers that have evaluated a proof for a data item previously in the transaction. For view consistency, the TM merely needs to check the version number it receives from the server that is executing the query with all of the version numbers from previous queries. If the current version number is newer than one previously seen, it must abort the transaction. At the end of the transaction, all of the proofs will have been generated with consistent policies, and therefore do not have to be re-evaluated. That is, 2PVC does not do policy validation and acts like 2PC.

For Incremental Punctual under global consistency, however, propagations of new policies are seen by the transaction and the TM must communicate with previous servers. Again, the TM simply needs to poll each server for the latest policy versions and compare them with the known master version. The TM will then abort if it finds a server that has a policy newer than the master. It is still unnecessary, however, for validations to be evaluated by 2PVC.

Finally, Continuous proofs of authorization (Definition 9) are the most involved. Unlike the case of Incremental Punctual in a view consistency, Continuous proofs of authorization does not abort when it sees a newer policy version. Instead, it invokes 2PV at the execution of each query which will update the older policies with the new policy and re-evaluate. If the 2PV results in a CONTINUE decision, the transaction executes the next query. Upon ABORT, the transaction aborts. The same actions occur under global consistency with the exception that a global version number is used.

Recovery: In distributed environments, being able to handle failures is critical. The resilience of 2PVC to system and communication failures can be achieved in the same manner as 2PC by recording the progress of the protocol in the logs of the TM and participant (as in Figure 7). In the case of 2PVC, a participant must forcibly log the set of (v_i, p_i) tuples along with its vote and truth value.

Furthermore, the logging behavior of 2PC is agnostic to the actions taken by the voting phase as it logs strictly before and after. As such, any log-based optimizations of 2PC also apply to 2PVC. This includes the common variants *Presumed-Abort* (PrA) and *Presumed-Commit* (PrC) [5].

VI. Evaluation

A. 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 $2n + 1$ [5]. This can be improved by using a compatible optimization as discussed in the previous section.

TABLE I
THE COMPLEXITIES OF THE VARIOUS PROOF SCHEMES

	Deferred		Punctual		Incremental		Continuous	
	View	Global	View	Global	View	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	$\frac{u(u+1)}{2}$	$\frac{u(u+1)}{2} + ur$

Table I 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. Generally, 2PVC requires $2n+2nr$ messages, where there are $2n$ messages for the voting phase (which may be repeated r times) and $2n$ messages for the decision phase. With view consistency, the number of voting rounds r is at most 2 (one extra round when compared to 2PC). For the case of Deferred and Punctual under view consistency, only the 2PVC is used. As such, they both require $2n + 4n$ messages in the worst case (r is 2). For Incremental and Continuous, however, consistency is maintained throughout the transaction which fixes r to 1. In the case of Incremental Punctual, the 2PVC is invoked without validations for a total of $4n$ messages. Continuous proofs of authorization perform 2PV at each query, which requires communication with potentially one extra server for each subsequent query executed. As such, the number of messages for 2PV is given by $2\sum_{i=1}^{u} i$, which is equal to $u(u + 1)$. By adding the $4n$ messages of the 2PVC without validations, the total becomes $u(u + 1) + 4n$.

In terms of proofs, the general 2PVC will evaluate u queries each round for ur overall proof evaluations. For the case of a view consistency, 2PVC will evaluate the first round by validating all u queries. For the second round, at least one query will not have to be re-evaluated as it is the one providing the latest policy. Therefore, at most $2u-1$ proofs are required. Since Deferred uses only the 2PVC, it requires those $2u - 1$ proofs. Because Punctual also evaluates proofs during the transaction, it adds an extra u proofs totally $u + 2u - 1$. Incremental Punctual, as mentioned previously, maintains the policy consistency as the transaction executes and, as such, does not require 2PVC with validations. In this case, it simply evaluates the u proofs during the transaction execution. Since Continuous proofs of authorization perform 2PV at each query, it will require an extra proof for each subsequent query executed. Similar to the number of messages, the number of proofs during the transaction execution are given by $\sum_{i=1}^{u} i$ which is equal to $\frac{u(u+1)}{2}$ proofs. It does not require 2PVC with validations at commit time since running 2PV at the final query does the equivalent work.

In the case of global consistency, r is not bounded. The Deferred and Punctual schemes now require the general $2n + 2nr$ messages plus r messages to receive the latest policy version per round. The proofs for both must account for extra rounds during 2PVC giving ur and $u + ur$ proofs, respectively. Both Incremental Punctual for global and view consistencies have the same complexity. The global version has one difference as it retrieves the global version number every query for an extra u messages on top of the $4n$ for 2PC giving a total of $4n + u$. The number of proof evaluations is the same since 2PVC with validations are not required. Finally, Continuous uses the general 2PVC along with u messages to get the master version number for the u invocations of 2PV and r messages to get the master version number for 2PVC. The total messages become $u(u+1)+u+2n+2nr+r$. Since the 2PVC validations must now be performed, an extra ur proofs are added in the global consistency giving a total of $\frac{u(u+1)}{2} + ur$.

B. Trade-Off Discussion

Clearly the various approaches towards consistency enforcement described in this paper offer differing guarantees regarding the level of consistency that they provide and their cost of enforcement. We now briefly describe the choice of algorithm used in response to two factors: transaction length and time between policy updates.

Transaction length < update interval. If, on the average, the length of a transaction does not exceed the expected interval between policy updates, it is best to rely upon either Deferred or Punctual proofs. If the transaction duration is relatively short, Deferred proofs are preferred, as the time required to rollback a failed transaction will be very short and recovery can happen on-the-fly. For longer transactions, Punctual proofs can be used to detect inconsistencies early, update, and then finish the transaction using the updated policy.

Transaction length > update interval. If, on the other hand, policy updates are expected to happen during the course of a transaction, Incremental or Continuous proofs should be used. During the execution of long transactions, the use of Continuous proofs is best since this will prevent potentially long rollbacks from occurring. By contrast, Incremental proofs are sensible for use in relatively short transactions, as they do not require additional policy synchronizations that would prolong an otherwise short transaction.

VII. RELATED WORK

Cloud Environment: Many database solutions have been written for use within the cloud environment. For their own purposes and for their cloud infrastructure, EC2, Amazon uses their own database solution called Dynamo, which is built on top of their S3 storage layer and is motivated by a desire to provide high availability among thousands of servers [9]. Google built Bigtable, which is widely used for their own services such as Google Earth, Google Finance, and their web indexes [10]. Facebook implemented Cassandra, which is now maintained by Apache, which implements a simple key-value store model with eventual consistency [11].

The fact that these new database projects were implemented even though mature database solutions were already available suggests that the cloud environment requires a level of specialization not before seen. Apparently, the difference lies with the focus on elasticity, also known as horizontal scalability. With this in mind, these solutions make a trade-off between data consistency and availability that scales as servers are added to the system. It becomes obvious that such a consistency model adds a new dimension to the complexity of the design of large scale applications [12].

Distributed Transactions: Providing transactions in the cloud is not a new revelation. It is assumed that strict transactions are still necessary for many applications in the cloud. The work of CloudTPS shows a solution that will provide full ACID properties with a scalable transaction manager designed for a NoSQL environment [13]. However, This work is primarily concerned with providing consistency and isolation upon data without regard to considerations of authorization policies.

There has also been work on providing some guarantees about the relationship between data and policies [14]. This work proactively ensures that data stored at a particular site conforms to the policy stored at that site. If data does not conform, it is lost. If it does conform, it is stored. If the policy is updated, it will scan the data items and throw out any that would be denied. 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 within a transaction spanning multiple nodes.

Distributed Authorization: Distributed proofs of authorization have been studied as well, although not in this dynamic cloud environment. The work by Lee, et al, shows that when policy is static, a distributed proof can be determined that satisfies several different types of consistency [4]. These consistency guarantees are very similar to our definitions of safe transactions. It follows that if the policy is the same on all nodes, that is strictly consistent, during a transaction, then a distributed proof can be found. It is our motivation to ensure that policies are consistent within a transaction.

VIII. Conclusions

In this paper, we identified prospective consistency problems that can arise as transactional database systems are deployed on cloud servers and use policy-based authorization systems to protect sensitive resources. We defined the notions of *trusted* and *safe transactions*, and introduced different levels of policy consistency constraints. We presented different proofs of authorizations approaches to achieve trusted transactions, and showed that each approach offers different ”trust” guarantees during the course of executing transactions. We proposed Two-Phase Validation Commit (2PVC) protocol, an enhanced version of the widely used Two-Phase Commit (2PC) protocol, to implement our approaches and ensure safe transactions. Finally, we evaluated each approach in terms of the performance and applicability.

As an extension to this work and part of our ongoing work, we are investigating the different trade-offs of the proposed approaches by simulating their execution over a cloud infrastructure. Given a better understanding of the execution times of each approach in both short/long transactions and frequent/infrequent policy updates, we can provide quantitative measures to better guide the decision process.

Acknowledgments

This work was supported in part by the National Science Foundation under awards CCF–0916015, CNS–0964295, CNS–1017229 and IIS–1050301.

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] D. J. Abadi, “Data management in the cloud: Limitations and opportunities,” IEEE Data Engineering Bulletin, 32(1), Mar. 2009.

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

[5] 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.

[6] G. Samaras, K. Britton, A. Citron, and C. Mohan, “Two-phase commit optimizations and tradeoffs in the commercial environment,” in *IEEE ICDE*, 1993.

[7] M. Myers *et al.*, “X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP,” RFC 2560 (Proposed Standard), IETF, 1999.

[8] W. Yu, Y. Wang, and C. Pu, “A dynamic two-phase commit protocol for self-adapting services,” in *IEEE SCC*, 2004.

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

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

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

[12] W. Vogels, “Eventually consistent,” *in Commun ACM*, vol. 52, Jan. 2009.

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

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