Policy-Based Access Control for an RDF Store

Pavan Reddivari
University of Maryland, Baltimore County
Baltimore MD USA
pavan2@csee.umbc.edu

Tim Finin
University of Maryland, Baltimore County
Baltimore MD USA
finin@csee.umbc.edu

Anupam Joshi
University of Maryland, Baltimore County
Baltimore MD USA
joshi@csee.umbc.edu

ABSTRACT

Specialized stores for RDF data are essential parts of many Semantic Web applications. Current RDF stores have primarily focused on efficiently storing and querying large volumes of data and little attention has been given other features common to many database systems, including how information can updated and maintained or access to data controlled. The problem is complicated by the fact that the addition or deletion of a simple fact (i.e., an RDF triple) are not atomic since they can trigger reasoning that can result in adding or deleting derived triples. Current access control mechanisms for RDF stores largely ignore this aspect. We describe a policy based mechanism to determine access control for an RDF store.

RAP is a prototype implementation of an RDF store with integrated maintenance capabilities and access control using user defined policies. All actions to the store are routed through RAP policy engine, to determine whether the action is permitted or prohibited. In the RAP framework, the same RDF store is also used to store the policy, as well as metadata about the triples, allowing greater range in policy specification.

1. INTRODUCTION

The Wed is evolving from a purely human consumption perspective to a more machine comprehensible web. This new evolution, driven by the Semantic Web vision, is leading us to a world of information sharing, by enabling distributed knowledge aggregation and creation. Most of these Semantic Web applications require management of large amounts of semantic data which is stored in knowledge stores.

Resource Description Format (RDF)[27, 23, 14] knowledge stores have become a standard component of many Semantic Web applications. We believe that for RDF stores to be more functional and widely deployed in applications they ought to provide a mechanism to specify restrictions on creation, modification and browsing of the knowledge. Current implementations such as the open source project Kowari metastore [24], Redland [11], and TAP [29] are mostly focused on scalability and rarely address issues of security and access control.

In our work we have mapped out a set of actions which are required to completely manage a store, and describe a model of access control to permit or prohibit these actions. In our model, agents make requests to perform actions against the RDF store and the decision whether or not to carry out the requested action is governed by an explicit policy described in a simple, declarative policy language.

The policies are defined by a collection of policy rules governing whether the action is permitted or prohibited. Examples of actions include inserting a set of triples into the store, deleting a triple, and querying whether or not a triple is in the store. The conditions on a policy rule are a combination of constraints on the agent requesting the action, the type of action requested, the history of previous actions, the contents of the store, and the possible effect on the store and its model.

Informal examples illustrating the range of policy rules that RAP supports include the following.

- *Only agents assigned to an editor role are allowed to insert or delete triples.*
- *An agent can only delete triples it previously inserted.*
- *An agent is only allowed to 'add properties' to classes it introduced.*
- *No agent may see any values of a 'social security number' property.*
- *No agent may insert a triple that allows any agent to infer a patient's 'HIV status'.*
- *An agent may modify any data about itself.*
- *An agent may not add an instance of a foaf:Person without providing a foaf:name property and either a fof:mbox or foaf:mbox_sha1sum property.*

The wide array of policies is possible because of the conscious decision of storing the domain ontology, the knowledge base schema, RDF data and polices together. The

agents are also represented in RDF and are part of the domain specific knowledge.

Access to the RDF store is controlled by the RDF-Store Access-Control Policies (RAP) framework through which all the transactions are routed. RAP uses the policies defined in the framework to determine whether to permit or prohibit the action requested by the agent.

We have implemented RAP using the Jena Semantic Web framework [28]. Jena provides a comprehensive environment for parsing, reasoning over and querying information expressed in the Semantic Web languages RDF, RDFS and OWL. Jena also includes a rule-based inference engine that supports forward and backward reasoning using rules whose atoms are RDF triples [9].

The RAP polices are specified as rules using the RAP ontology to the RETE inferring engine running in the forward chain mode. A Web-Service is also provided to access the store, using the Sun One web-service server [6]. The authentication of the agent is to be performed by an independent mechanism.

2. RELATED WORK

Though knowledge stores have a made a presence in many systems, access control in these system has hardly received the attention it deserves. Though there has been considerable research relating to the field of access control, these schemes can neither be applied directly to RDF stores nor do these schemes meet all the requirements. In this section, we discuss some related work in access control in general and access control in XML.

2.1 Access Control Mechanisms

In this section, we discuss a few schemes for access control. Most of these schemes have poor expressibility and do not support indirect actions, where performing one action leads to automatically perform another action. The most common access control mechanism is Role Based Access Control (RABC) [20, 22, 30, 32] in which agents are assigned roles and permissions are specified for each agent to decide which actions the agent has permission to perform. In such systems roles have to be pre-assigned and every time the access right of an entity needs to changed, it cannot be performed without changing roles. Role Based Access Control have inherit limitation because of close coupling of roles and access rights and changing one requires modification of the other. In comparison using Policy based scheme allows us to bind the access rights not to the roles, but credentials and properties of the agent. This allows the independence of the user properties and the access

The Simple Public Key Infrastructure (SPKI)[5], Simple Distributed Security Infrastructure (SDSI) are some of the other access control mechanisms widely used, but none can be successfully incorporated in to access control for RDF.

2.2 XML Access Control and RDF

In current form RDF can be represented in either N-triples or XML. XML is the most common used format to represent RDF. Access Control in XML has been researched and many mechanisms have been developed [18, 17]. These mechanisms provide an authorization model tightly tied to the XML syntax and structure. This approach is completely acceptable to XML documents but not to RDF documents.

In XML syntax of a document is fixed, each document can be represented only in one way. The limitation of using XML access control mechanism in RDF is that fact RDF is syntax independent, the same RDF statement can be represented in variety of way. The other issue is the knowledge property of RDF statements. In RDF adding one statement might lead to inferring additional statements. For example RDF describing X as a boy implies that X is male and similarly deleting a statement will lead to loss of more statements.

Therefore the mechanisms should be able to dynamically assign access control on the new inferred statements and detect unauthorized statements being inferred. Kaushik et al. [33] have done some work on access control in ontologies. They use a logic programming based policy language to determine access to partial or full ontologies. Jain and Farkas [10] propose an access control framework for RDF data that assigns Multi Level security classifications to RDF patterns. The model has been partially implemented in a prototype system that controls access to information stored in RDF. It does not deal with insertions, deletions or updates.

Rei [26, 25] uses a policy-based approach for governing autonomous behavior in distributed environments. It allows permissions to be specified as policies over actions and obligation. The policies are specified in OWL-Lite [7] extended with a simple variable mechanism. Rei has a concept of meta-policies for conflict resolution, speech acts for remote policy management. The Rei policy engine is developed in XSB [15], it uses the policies and domain knowledge to generate permissions for the actions. The philosophies and design decisions used in REI were studied and some incorporated during the design of RAP.

3. RDF GRAPH

In this section we review the RDF model [27, 23, 14] and identify a set of primitive actions that can be performed on a RDF graph. An RDF graph is composed of three types of nodes, a RDF URI references node (N), a Blank node y(B) and a RDF literal Node (L). The edges (E) in the graph are directional and each edge also is associated with a URI [31]. The triple in a RDF graph can be described as (subject, predicate, object) $\in \quad (N \cup \quad B) \times E \times (N \cup B \cup \quad L)$.

Consider the following example RDF document encoded using the RDF/XML syntax and depicted as a graph in Figure 1.

```
1:<?xml version="1.0" encoding="utf-8"?>
2:<rdf:RDF xmlns:rdf=
         "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3: xmlns:owl="http://www.w3.org/2002/07/owl#"
4:         xmlns:foaf="http://xmlns.com/foaf/0.1/" >
5: <foaf:Person>
6: <foaf:name>Li Ding</foaf:name>
7:  <foaf:mbox rdf:resource="mailto:dingli1@umbc.edu"/>
8: <owl:sameAs rdf:resource=
    "http://www.csee.umbc.edu/~dingli1/foaf.rdf#dingli"/>
9: </foaf:Person>
```

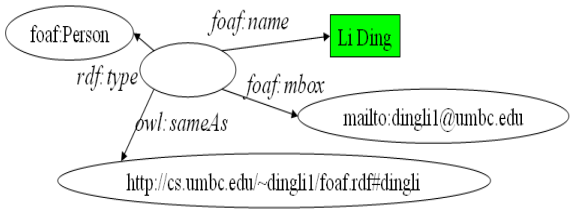

Figure 1: The RDF document described earlier has a simple representation as a graph.

```
10:</rdf:RDF>
```

Line 1 declares that this is an XML document. Lines 2-4 further define the content to be an RDF document and provide abbreviations for three common "namespaces" for RDF, OWL, and Friend of a Friend (FOAF), which defines classes and properties for describing people, their common attributes, and relations among them. The SWD's vocabulary consists of literals ('Li Ding' in line 6), URI-based resources (mailto:dingli1@umbc.edu in line 7), and anonymous resources (lines 5-9). Users assert statements using RDF triples such as the one starting at line 5, which has an anonymous resource as the subject, rdf:type as the predicate, and foaf:Person as the object. A higher level of granularity is class-instance, which RDFS's object-oriented ontology constructs offer. Lines 5-9 assert that "there is an instance of a foaf:Person having foaf:name Li Ding, foaf:mbox mailto:dingli1@umbc.edu, and this instance is owl:sameAs, identified by http://www.csee.umbc.edu/ dingli1/foaf.

The simplest manipulations on this graph are adding or deleting a single triple consisting of a *subject*, a *predicate* and an *object*. For our purposes, we consider three special cases when inserting a triple, yielding the following four actions:

1. Add a triple such that neither the subject nor object nodes were previously in the graph. This leads to addition of two new nodes and an edge to the graph.
2. Add a triple such that either subject or object (but not both) existed in the graph prior to the addition. This leads addition of one new node and an edge to the graph.
3. Add a triple such that both subject and object node exist in the graph prior to this addition. This leads addition of an edge to the graph.
4. Delete a triple from the graph. This will lead to the predicate edge being removed from the graph and the subject and object nodes may be removed or not, depending on whether they are part of any other triple or not.

In addition, we will introduce and make use of several compound actions and indirect actions. Compound actions include the action of updating or replacing one triple with another, the action of inserting a set of triples, and the action of deleting a set of triples. Indirect actions cover the introduction or removal of a triple in the model through the addition or deletion of separate tripe into the explicit store.

4. RDF STORE ACTIONS

We need to identify the set of actions which are required to maintain an RDF store. The access control policies will control permission and prohibition to these actions. Maintaining an RDF store involves four basics actions: adding, deleting, updating and searching for triples.

4.1 Additions to the store

These actions allow agents to add new information to the RDF stores.

- **insert(A, T)**. Agent A directly inserts triple T into the graph. This action is used by the agent to add minimal information into the store, such as '*foaf:Person* is a subclass of *wn:Mammal*.
- **insertModel(A, T)**: Agent A insertModels triple T if A performed **insert(A, T1)** and T1's insertion enables the store to infer that triple T is in the model. This action leads to indirect addition of knowledge by the user, such as after adding the triple *foaf:Person* is a subclass of *wn:Mammal*, addition of triple X Instance of *foaf:Person* leads to indirect addition of X *rdf:type wn:Mammal*. Constraints on this action are useful in preventing an agent from adding information indirectly.
- **insertSet(A, {Tc})**: Agent A insertSets a set of triples {Tc} if A inserts all the triples in {Tc} into the store together. It is possible that A is not allowed to add the triples in set {Tc} individually. This action can be used to ensure that the agent always inserts a set of triples which are related, for instance an agent may not add an instance of a foaf:Person without providing a foaf:name property and either a foaf:mbox or foaf:mbox_sha1sum property .

4.2 Deletions from the store

These actions allow agents to delete information from the stores.

- **remove(A, T)**: Agent A directly removes triple T from the graph. This Action would be used by the agent to remove minimal information from the store, such as ?X*emp:WorksFor* of *foaf:CompanyX*.
- **removeModel(A,T):** Agent A removeModels triple T if A performs **Remove(A,T1)** and the store cannot infer triple T after the removal of T1.
- **removeSet(A, {Tc})**: Agent A removeSets a set of triples {Tc} if A removes all the triples in {Tc} into the store together. It is possible that agent A is not allowed to remove the triples in set {Tc} individually. This action is useful when you do not want the agent to remove something unless it is removing something else too. For instance you might want to enforce a

policy that unless you are deleting the entire employee record, the social security number property can not be removed.

4.3 Updates to the store

The update action provides a mechanism to update particular triples in an RDF store. While this could me modeled as a combination of a delete and an insert, it is convenient to have an update that acts as a single transaction.

- **update(A, T1, T2):** Agent A directly replaces the triple T1 with the T2.

The update action is useful in cases when you want the user to have the modification rights without the deletion right as in the case where you want your employees to be able to modify their cell phone triple but not delete it.

4.4 Querying the store

Two actions are defined to describe an agent's actions of querying or searching an RDF store, covering both direct and indirect access.

- **see(A, T):** Agent A sees triple T if it returned in the response to one of A's queries to the store. This action will allow users to browse the knowledge in the store.
- **use(A, T):** Agent A uses triple T if it is used by the store in answering one of A's queries. This action is useful when you want the user to be able to restrict what information is being used to answer agent A's query.

Both these actions are independent of each other, even though it might appear that if A can 'see' triple T, then A can 'use' triple T. For example, consider three triples T1, T2 and T3. Let us assume that A can infer T3 only by using T1 and T2. If A can see T1 but cannot use it and can use T2 but cannot see it, then A will not be able to see T3.

5. RAP POLICIES

In the RAP framework, a policy is defined by a set of policy rules that together specify if an agent's specific requested action is permitted or prohibited. Following Rei [26, 25], a query about the status of an agent's specific action request might have any of four outcomes: unknown, proven to be permitted, proven to be forbidden, and proven to be both permitted and forbidden. Like Rei, RAP allows a policy to include meta-rules that can be used to resolve the two problematic cases. The two kinds of meta-rules that RAP allows are a *default policy* and a *modality preference*. Together, these can be thought of as implicit policy constraints.

The default policy, if specified, determines what happens in the upper left quadrant of the decision matrix shown in 2. If *default(permitted)* is true then any actions not explicitly prohibited are permitted. If *default(prohibited)* is true, than actions not expressly permitted are prohibited. One of these two default settings must be selected (typically *default(prohibited)*).

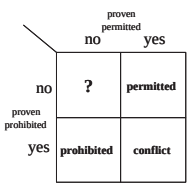

Figure 2: In reasoning about an action, four outcomes are possible. An uncertain or conflicted outcome may be resolved by meta-policy rules

The modality preference specifies what to do when we are in the lower right quadrant of the decision matrix. If *prefer(permitted)* is true, then an action that can be proven to be both permitted and prohibited is considered to be permitted. *If prefer(prohibited)* is true, then prohibitions dominate permissions. One of these two settings must be selected, typically the latter.

Explicit policy rules are used to permit or prohibit an agent from performing a class of actions on the RDF store. The general form of a policy rule is "*Modality(Action(A,T)) :- Condition*" where Modality is one of *permit* or *prohibit*, Action names an action, A identifies an agent and T identifies a triple. Condition is a Boolean combination of simple constraints expressed as RDF triples. The Triple (T) represented in the head of the policy has the form (subject, predicate, object). The wild card character "?" can be used in the triple pattern, a triple of the form (?, ?, ?) would thus hold true for all the triples.

The Specification of the agent is defined by the agent representation in the domain knowledge. This allows us to specify policies using agent specific data.

The Condition for the policy can be specified either using the metadata about the triples, the triple data itself, the agent data or by combing both Agent and triple data.

5.1 Metadata specific conditions.

The conditions in the policy can be specified based on the metadata about the triples that the store maintains. The kind of metadata to be collected is specific to the store implementation.

permit(insert(A,(?,rdfs:type,C))) :- createdNode(A,C)

The above policy will allow agents to create instances of classes only if they had created those classes. The createdNode (A, C) returns true if agent A had created triple T which created node C.

5.2 Triple specific conditions

The policies can also be specific to the kind of triples being added.

prohibit(see(A,(?,emp:salary,?))

prohibit(see(A,(?,P,?))) :-

rdfs:subProperty(P,emp:salary)

These policies will prohibit agents from seeing the value of the emp:salary property, its sub properties or any equivalent property. The rdfs:subProperty(P,emp:salary) returns True if predicate P is defined to be an rdfs:subProperty of emp:salary.

5.3 Agent specific conditions.

The attributes of the agent could also be used in the conditions of policy. The agent's representation would be specific to the domain

permit(see(A,(?,emp:salary,?)):-

existTriple(A,rdfs:type,emp:Auditor)

This policy will permit an agent A to see anyone's salary as long as the agent A is an auditor.

5.4 Agent and Triple specific conditions.

The conditions in the policy could be tied to both the agent attributes and the triple data being acted upon.

permit(update(A,(P,emp:salary,?),(P,emp:salary,?)) :-

existTriple(A,emp:Supervisor,P)

This policy will permit an agent A to update salary of P as long as A is the supervisor of P.

5.5 Custom Policy Predicates

The RAP framework defines and implements several special predicates useful for building policies. The current set of four predicates, two using the metadata about triple 'owners' and two based on RDFS predicates, be extended as needed.

- **isTripleOwner (A, T):** This predicate determines ownership of the triple. It is true if agent A created the triple T.
- **isNodeOwner(A, N):** This predicate determines ownership of the node in the RDF graph. It is true if agent A was first to create the node N in the RDF graph.
- **isSchemaPredicate (P):** This predicate is true if P is a predicate used to define RDF schema level information (e.g., rdfs:subClass, rdfs:domain, etc). This is useful in policies that specify whether or not an agent is allowed to add to or modify the schema used by the RDF store.
- **isSubProperty (P1, P2):** This predicate is true if P1 is a Sub-Property of P2.

6. RAP POLICY ONTOLOGY

In this section we explain the RAP ontology and describe how the RAP policies are converted to set of rules. These rules are then used by the RAP policy engine to determine whether the actions requested by the agents should be permitted or prohibited. Here list of permissible set of actions:

- **Insert Action:** This action adds a triple into the store explicitly.
- **Model Insert Action:** This action adds a triple as result of addition another triple into the store.
- **Set Insert Action:** This action adds a set of triples to the store.
- **Delete Action:** This action deletes a triple from the store.
- **Model Delete Action:** This actions deletes a triple as result of deletion another triple.
- **Set Delete Action:** This action deletes a set of triples from the store.
- **See Action:** This action shows a triple as a part of a result set for query requested by the agent
- **Use Action:** This action uses a triple to generate the result set for query requested by the agent
- **Update Action:** This action replaces one triple with another

Each action is associated with an actor, the agent requesting the action as shown in Figures 3. The effects of each action are also associated with it, an insert action might lead to a model insert action and a delete action might lead model delete action. The actions are also associated with an action object, it is the statements or set of statements the action is performed with. Each action is associated with two types of permissions, an explicit permission which gets dynamically generated as per the policies. An implicit permission is also associated with each type of action which is static to each type.

RAP policies are converted to Jena's RETE forward chaining rules as shown in the examples in Figures 4 and 5.

7. RAP ARCHITECTURE

RAP allows agents to access the RDF store via a web-service, the agent tries to access the RDF store via the RAP web service by providing its identity and the desired action it wants to perform. The identity of the agent is verified independently by other mechanisms. The agent's request is send to RAP policy engine, here the action is modeled as a RAP ontology action and submitted to the policy engine for approval. The policy engine first determines if according to current set of policies the agent has Permission to Perform this action. In the case the agent is allowed perform the action, RAP temporarily performs the action and records the indirect effects of this action. These effects are then modeled as RAP ontology effects of the performed action. Once the agent has permission to perform the action and its effects

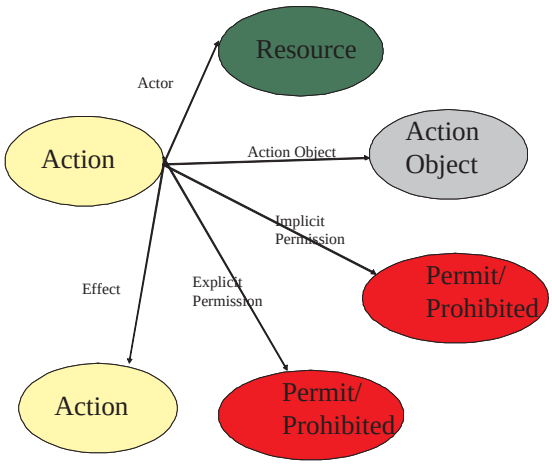

Figure 3: RAP Action description

RAP Policy : Only a Supervisor of a Person can update the Salary .

permit(update(A,(P,emp:salary,?),(P,emp:salary ,?)) :- existTriple(A,emp:Supervisor,P)

Raprule: *(?x rdf:type rap:Update_Action) (?x rap:Actor ?y (?x ap:oldTriple_object ?z1) (?x newTriple_object ?z2) (?z1 rdf:subject ?s1)(?z1 rdf:predicate emp: Salary) (?z2 rdf:subject ?s1)(?z2 rdf:predicate emp:Salary) (?y rap:Supervisor ?s1)*

->(?x rap:explicitPermission rap:Permitted)

Figure 4: RAP policy rules are expressed in a Prolog-like notation and then compiled into Jena forward chaining rules. This policy rule specifies that a person's supervisor is allowed to update his salary.

RAP Policy : Only a Creator of a class can make instances .

permit(insert(A,(?,rdfs:type,C))) :- createdNode(A,C)

Raprule: *(?x rdf:type rap:Insert_Action) (?x rap:Actor ?y (?x rap:insertTriple_object ?z1) (?z1 rdf:object ?s1)(?z1 rdf:predicate rdfs: type) (IsNodeOwner(?y , ?s1))*

->(?x rap:explicitPermission rap:Permitted)

Figure 5: RAP policy rules are expressed in a Prolog-like notation and then compiled into Jena forward chaining rules. This policy rule specifies that an agent is allowed to create an instance of a class if the agent had previously introduced the class into the schema.

RAP performs the action on the RDF store on the agent's behalf.

The framework consists of four components: RDF store, the RETE policy engine, the actions implementations, and a web-service API. In section we study them detail and show how each component was designed and implemented.

7.1 RAP RDF Store

In RAP, the RDF store contains domain specific RDF schema as well as RDF data. RDF store is also used to store the policy, represented in RDF, as well as other data and meta-data needed for the policy rules. The agents are also represented in RDF and are parts of the domain specific knowledge. This representation of agents is used in the policy specifications. The RDF store also maintains metadata about the triples in

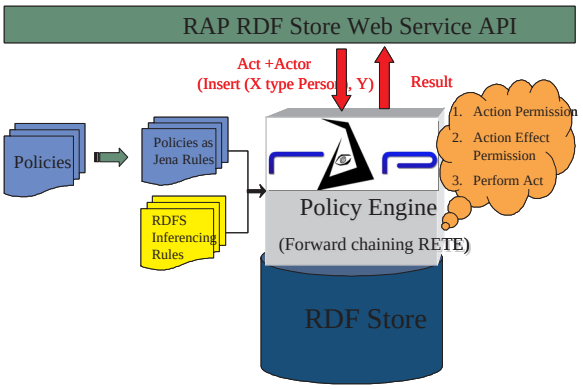

Figure 6: RAP Architecture has four main components: RDF store, the RETE policy engine, the actions implementations, and a web-service API.

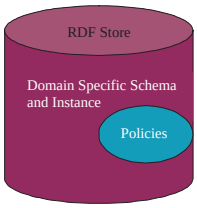

Figure 7: RAP's RDF store holds the RDF data, its schema, provenance metadata and the access policy rules. A uniform and integrated representation in RDF supports a wide and expressive range of access control policies.

the store, like the creator of the triple. The store is implemented as jena RDF store object. RAP provides persistence of the store by serializing the store and storing it on the file system. The meta-data about the triples such as creator of the triple, creator of nodes in the graph are stored using persistent hash-maps in Java.

7.2 RAP RETE Policy Engine

The policy engine is implemented using the general purpose reasoner provided in Jena2 [2] framework. Jena2 includes a general purpose rule-based reasoner which is used to implement both the RDFS and OWL reasoner but is also available for general use. This reasoner supports rule-based inference over RDF graphs and provides forward chaining, backward chaining and a hybrid execution model.

In RAP we use the policy engine in RETE forward chaining mode, which is based on the standard RETE algorithm [21] which is optimized for such incremental changes. The policies are modified to allow the re-inferencing of certain triples even if the triples are already present in the model. A new primitive *deduce* was developed to allow the engine to derive even the redundant facts. This feature allows us to determine what effects an action can cause. The RAP policies are also converted to forward chaining rules and added to the same rule-set.

7.3 RAP Action Implementations

In the RAP framework the access to store is controlled by a set API of actions. These set of actions allow addition, updating, deletion, querying for complete management of the RDF store. In section we discuss how each action was implemented.

7.3.1 Insert Actions

This action is most basic action, allowing the addition of a triple to the store. To add a triple to the store an Insert Action RAP ontology action is created. The policy engine after inferencing fills the permission status of this action. If an explicit permission is not generated the implicit permission for insert action is referred to. A permissible status of action leads to a temporary addition of the triple. Following the addition inferencing is performed on the model to determine the set of new inferred triples. These set of triples are recorded by using the model listener in Jena. The Model listener monitors the RDF graph and records the set of new triples added to the graph. These set of triples are modeled as addition to the store by Model Insert actions. Each triple which was inferred causes Model Insert RAP ontology action, which linked to the action as effects. The action would be permitted if the value of its permission property is *permitted* and the permission of all its effect is also *permitted.* If either the action or any its effect have *prohibited* the action is said to be prohibited.

7.3.2 Set Insert Actions

This action allows addition of a set of triple to the store. This action allows RAP to implement a kind of obligation speech act. The agents are obligated to insert specific set of triples at the same time. These statements may not be allowed to be inserted individually. For example a policy rule may require that an agent insert a foaf:Name property whenever it inserts a new foaf:Person.

To add a set of triples to the store a Set Insert Action RAP ontology action is created. Each statement which is a part of the set is attached to the current action as a set insert object. The action is also appended with the agent that requested the action. The policy engine after inferencing fills the permission status of each action object. If an explicit permission is not generated the implicit permission for the set insert action is referred to. Permission of each action object is verified, a *prohibition* status for even a single action object leads to Prohibition of Set-Insert action. A permissible status of action object leads to a temporary addition of the triple, following the addition inferencing is performed on the model to determine the set of new inferred triples. These set of triples are recorded by using the model listener in Jena. The Model listener monitors the RDF graph and records the set of new triples added to the graph. These set of triples are then modeled as addition to the store by Model Insert actions. Each triple which was inferred causes Model Insert RAP ontology action, which linked to the current action as effects. The set action would be permitted if the value of the action objects permission property is *permitted* and the permission all its effect is also *permitted.* If either the either action objects or any the effects are *prohibited* the Set Insert action is said to be prohibited and no triples would be added.

7.3.3 Delete Action

This allows deletion of triple to the store. To delete a triple to the store a Delete Action RAP ontology action is created. This current action is appended with the agent that requested the action. The policy engine after inferencing fills the permission status of this action. If an explicit permission is not generated the implicit permission for insert action is referred to. A permissible status of action leads to a temporary deletion of the triple, following the deletion inferencing is performed on the model to determine the set of triples were depended on the deleted triple. These set of triples are recorded by calculating the difference in the model before and after the deletion of the said triple. These set of triples are modeled as deletion to the store by Model Delete actions. Each triple which was deleted causes Model Delete RAP ontology action, which linked to the current action as effects. The current action would be permitted if the value of its permission property is *permitted* and the permission all its effect is also *permitted.* If either the current

action or any its effect have *prohibited* the current action is said to be prohibited.

7.3.4 Set Delete Actions

This action allows deletion of a set of triple to the store. This action allows RAP to implement a kind of *obligation* speech act. The agents are obligated to delete specific set of triples at the same time – these statements may not be allowed to be deleted individually.

To delete a set of triples to the store a Set Delete Action RAP ontology action is created. Each statement which is a part of the set is attached to the current action as a set delete object. The current action is also appended with the agent that requested the action. The policy engine after inferencing fills the permission status of each action object. If an explicit permission is not generated the implicit permission for the set delete action is referred to. Permission of each action object is verified, a *prohibition* status for even a single action object leads to Prohibition of Set-Delete action. A permissible status of action object leads to a temporary deletion of the triple, following the deletion inferencing is performed on the model to determine the set of indirectly deleted triples. These set of triples are recorded by calculating the difference in the model before and after the deletion of the said triple. These set of triples are then modeled as deletion to the store by Model Delete actions. Each triple which was deleted causes Model Delete RAP ontology action, which linked to the current action as effects. The set action would be permitted if the value of the action objects permission property is *permitted* and the permission all its effect is also *permitted.* If either the either action objects or any the effects are *prohibited* the Set Delete action is said to be prohibited and no triples would be deleted.

7.3.5 Update Action

This action replaces one triple with another**.** To update a triple in the store an Update Action RAP ontology action is created. This current action appended with the agent that requested the action. The policy engine after inferencing fills the permission status of this action. If an explicit permission is not generated the implicit permission for insert action is referred to. A permissible status of action leads to a temporary updating of the triple, following the update inferencing is performed on the model to determine the set of new inferred triples and the set of indirectly deleted triples. These set of triples are recorded by using the model listener in Jena. The Model listener monitors the RDF graph and records the set of new triples added to the graph. The set of newly inferred triples are modeled as addition to the store by Model Insert and the set of indirectly deleted triples are modeled as deletion to the store by Model Delete actions. Each triple which was newly inferred or indirectly deleted is linked to the current action as effects. The current action would be permitted if the value of its permission property is *permitted* and the permission all its effect is also *permitted.* If either the current action or any its effect are *prohibited* the current action is said to be prohibited.

7.3.6 Query Actions

This action allows the agent to Query the store and see the result set**.** For each triple shown as a part of the result set a See Action RAP ontology action is created. This current action is also appended with the agent that requested the action. The policy engine after inferencing fills the permission status of this action. If an explicit permission is not generated the implicit permission for see action is referred to. A permissible status of action leads to a temporary adding to the result set. This triple is then checked to determine whether it was derived using other triples. If the triple is a derived triple all the triples leading to the derivation are collected. All these triples are then modeled as Use Action RAP ontology. This action is also appended with the agent that requested the action. The policy engine after inferencing fills the permission status of this action. If an explicit permission is not generated the implicit permission for use action is referred to. If either the see action or the use action are *prohibited* the triple is not added as part of the result set.

7.4 Rap Web Service

RAP provides a web-service to access the RDF store. All modifications, querying are routed through web-service to the policy engine. The web-service is developed using Sun One web-service server. The agent while invoking the specific methods provides the identity of itself as well as RDF data it wants to add, delete or update. The RDF data is serialized either in XML or N3 [12] format. While querying the store the agent has to specify the query in only RDQL [3] format. For details of messages and operations of the web-service refer to RAP web-service WSDL file at `http:\\www.rap.umbc.edu\rap.wsdl`.

8. EVALUATION AND FUTURE WORK

Performance evaluation of Rap is somewhat subjective and non-trivial as complexity of manipulation of an RDF-entailment is dependent on complexity of the RDF graph as well as the RDF schema of the store. Depending how much the RDF-entailment effects the RDF graph the performance of manipulating the entailments varies. Performance evaluation of RAP is also determined by the fact that it uses Jena for truth maintenances, which for performance issues does not store the complete truth table.

In general, the insert action had a negligible delay and the delete action had delay depending on the size of RDF graph due to graph difference operation in the implementation. The setInsert and setDelete actions were delayed by factor two to three times with setDelete taking somewhat more time. Querying the store was delayed in general by a factor of three, but it is completely dependent on the size of the result set.

Jena has a simple justification-based truth maintenance system [16] (TMS) which was not sufficient to support the reasoning required by RAP. For example, Jena only records one proof for an inferred fact even if several are possible. This means that we can not rely on the TMS to determine whether a triple's presence in the model depends on the presence of another which a querying agent is not permitted to use. Consequently, we employed the expensive technique of temporarily removing all triples that a querying agent is not allowed to see or use from the model before running a query. Using a reasoning system with a more comprehensive TMS would make checking *use* constraints more efficient.

Whether the reasoning overhead of a complete TMS would be justified would depend, of course, on the expected mix of policies and the value of other uses of a complete TMS.

A well known issue in database access control is the *inference problem* [19] – preventing sensitive data from unauthorized disclosure by revealing non-sensitive data and metadata from which the sensitive data can be inferred. Such mechanisms are known as *inference channels* and are difficult to detect and remove, especially since they are often low-bandwidth and complex and may depend on the use of additional background knowledge known by the querying agent. While handling constraints using the insertModel predicate might seem to require solving the inference problem, they do not for two reasons. The first is that we are only handling constraints with respect to knowledge and data currently in the RDF store. Additional knowledge, perhaps of a statistical nature, that the querying agent may have is not within the scope of RAP's processing. Second, RAP's reasoning is limited rdf- and rdfs-entailments, a tractable and complete reasoning problem for which many practical approaches are possible.

One of the main issues in the current RAP framework is that the actual store is not built into the framework. The store is an independent implementation in Jena and there were no specific modification made to Jena code to integrate the inferencing needs of the policy engine. In future we would like to integrate the policy action requirements, such as notification of inferred triples as a feature of the store. This would increase performance of RAP store and will make the response time of RAP store negligibly higher to that of normal RDF store.

Delegation of policies allows remote management of polices, it allows one agent to delegate the right of changing particular policy, to another agent. We feel delegation of policies would be very useful in the management of RDF stores. In current implementation of RAP delegation of policies is not possible. But a conscious design choice of representing policies in RDF and storing policies and domain knowledge together allows delegation of policies in future.

In the current version of RAP only RDF- and RDFS-entailments are considered, all OWL entailments are also treated as RDF, no specific provisions are considered. In future we would like to integrate some reasoning over OWL-lie and OWL-DL features into RAP.

9. CONCLUSION

The Semantic Web is often described as a "web of data" in which information and knowledge is encoded in ways that are easy for machines to process and make use of. The current technology being used to explore this model is RDF, extended with the RDFS and OWL vocabulary and semantics. RDF triple stores are playing an increasingly important role in practical applications using RDF data. We believe that for RDF store to be more functional and widely deployed in applications they ought to provide a mechanism to specify restrictions on creation, modification and browsing of the knowledge. The main motivation for our work was lack of such provisions in current implementations of RDF stores.

Towards this end, our RAP framework makes the following contributions. In our work we mapped out a set of actions which are required to completely manage a store, and describe a model of access control to permit or prohibit these actions. In our model, agents make requests to perform actions against the RDF store and the decision whether or not to carry out the requested action is governed by an explicit policy. Policies are defined by a collection of policy rules governing whether the action is permitted or prohibited. The overall system also shows the effectiveness of a policy based access control mechanism for a RDF store.

10. REFERENCES

[1] The foaf project. http://www.foaf-project.org/.

[2] Jena2 inference support. http://jena.sourceforge.net/inference/.

[3] Rdql w3c specifications. http://www.w3.org/Submission/RDQL/.

[4] Redland rdf application framework. http://www.librdf.org.

[5] Simple public key infrastructure. simple public key infrastructure (spki). http://www.ietf.org/html.charters/spkicharter.html.

[6] Sun one application server. http://www.sun.com/software/products/appsrvr/index.xml.

[7] Owl web ontology language. In *W3C Candidate Recommendation*, 2003.

[8] Rdf vocabulary description language 1.0: Rdf schema. In *W3C Working Draft*, 2003.

[9] Jena: implementing the semantic web recommendations. In *Proceedgins of the 13th International World Wide Web Conference*, pages 74–83, May 2004.

[10] Secure resource description framework: an access control model. In *ACM Symposium on Access Control Models And Technologies*, June 2006.

[11] D. Beckett. The design and implementation of the redland rdf application framework. In *Proceedgins of the Tenth International World Wide Web Conference*, May 2001.

[12] T. Berners Lee. Notatation3- an rdf language for the semantic web. http://www.w3.org/DesignIssues/Notation3.

[13] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, 279(5):34–43, May 2001.

[14] D. Brickley and R. Guha. Rdf vocabulary descrip-tion language 1.0: Rdf schemardf semantic. In *W3C Working draft, W3C*, January 2003.

[15] S. Decker. Yajxb (yet another java-xsb bridge). http://www-db.stanford.edu/.

[16] J. Doyle. A truth maintenance system. pages 259–279, 1987.

[17] E.Bertino, S.Castano, E.Ferrai, and M.Mesiti. Rspecifying and enforcing access control policies for xml documents and sources. In *World Wide Web*, 2000.

[18] E.Damaiani, S. di Vimeracati, S.Paraboshi, and P.Samarati. Role-based security for distributed object systems. In *A fine grained access control system for XML documents. ACM Transaction on Information and System Security TISSEC*, 2002.

[19] C. Farkas and S. Jajodia. The inference problem: a survey. *SIGKDD Explorations Newsletter*, 4(2):6–11, 2002.

[20] D. Ferraiolo and R. Kuhn. Role-based access controls. In *15th NIST-NCSC National Computer Security Conference, pages 554563*, 1992.

[21] C. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. In *Artificial Intelligence*, 1982.

[22] L. Guiri. A new model for role-based access control. In *11th Annual Computer Security Application Conference, pages 249255, New Orleans, LA*, December 2003.

[23] P. Hayes. Rdf semantic. In *W3C Working draft, W3C*, January 2003.

[24] T. Inc. Kowari-metastore. http://www.kowari.org.

[25] L. Kagal, T. Finin, and A. Joshi. A policy based approach to security for the semantic web. In *Proceedings of 2nd International Semantic Web Conference (ISWC2003)*, September 2003.

[26] L. Kagal, T. Finin, and A. Joshi. A policy language for a pervasive computing environment. In *Proceedings of IEEE 4th International Workshop on Policies for Distributed Systems and Networks*, June 2003.

[27] O. Lassila and R. Swick. Resource description framework (rdf) model and syntax specification. In *Working draft, W3C*, 1998.

[28] B. McBride. Jena: a semantic web toolkit. In *IEEE Internet Computing, v6n6, pp. 55-59*, November 2002.

[29] R.Guha, R.McCool, A.Sundarajan, and K.Joly. Tap:building semantic web. In *http://tap.stanford.edu*.

[30] R. S. Sandhu. Role-based access control. In *In M. Zerkowitz, editor, Advances in Computers, volume 48. Academic Press*, 1998.

[31] D. Weitzner, J. Hendler, T. Berners-Lee, and D. Connolly. Creating a policy-aware web: Discretionary, rule-based access for the world wide web. In *In Elena Ferrari and Bhavani Thuraisingham, editors, Web and Information Security*.

[32] N. Yialelis, E. Lupu, and M. Sloman. Role-based security for distributed object systems. In *In Fifth IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE 96) (1996), pp. 80-85,*, 1996.

[33] N. Yialelis, E. Lupu, and M. Sloman. Policy-based dissination of partial web-ontologies. In *SwS*, 2005.