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

Resource Description Format (RDF) stores have formed an essential part of many semantic web applications. Current RDF store systems have primarily focused on efficiently storing and querying large numbers of triples. Little attention has been given to how triples would be updated and maintained or how access to store can be controlled. In this paper we describe the motivation for an RDF store with complete maintenance capabilities and access control. We propose a policy based access control model providing control over the various actions possible on an RDF store. Finally, we discuss on how the Hypertext Transport Protocol (HTTP) and its extensions can be used to provide communication with the store.

General Terms

Management, Experimentation, Security.

Keywords

RDF Store, Access control, Policies, HTTP

1. INTRODUCTION

The Semantic Web is leading us to a world of information sharing, by enabling distributed knowledge aggregation and creation. Thus many semantic web applications require management of large amounts of semantic data and there have been ample number of RDF store implementations, which are capable of storing large number of RDF triples. 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. Current implementations of RDF stores such as Redland and Kowari are mostly focused on the aspect of scalability and very rarely address the issue of security and access control.

In this paper we will map 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 this 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. 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 Boolean 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 we would like to support 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.*

In the remainder of this paper we describe our preliminary design for RAP, a simple RDF access policy framework. An initial prototype, implemented using Jena [11], is under construction at the time of this writing.

2. RDF Graph

In this section we review the RDF model [8,9,10] and identify a set of primitive actions that can be performed on a RDF graph. An RDF graph is composed of three types of node, a RDF URI references node (N), a Blank node (B) and a RDF literal Node (L). The edges (E) in the graph are directional and each edge also is associated with a URI [1]. The triple in a RDF graph can be described as (subject, predicate, object) $\in (N \cup B) \times E \times (N \cup B \cup L)$.

The basic primitive manipulations on this graph can be performed by one of the following ways:

1. Add a triple (subject, predicate, object) to graph such that both subject and object node did not previously exist in the graph prior to this addition. This leads to addition of two new nodes and an edge to the graph.
2. Add a triple (subject, predicate, object) to graph such that either subject or object node did not exist in the graph prior to this addition. This leads addition of one new node and an edge to the graph.
3. Add a triple (subject, predicate, object) to graph 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 (subject, predicate, object) 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.

3. RDF store Actions

We need to identify the set of actions which are needed to maintain an RDF store. The access control policies will control permission and prohibition to these actions. Maintaining RDF store involves four basics actions: Adding, Deleting, Updating and Searching for triples.

3.1 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 *foaf:Mammal.*
- **insertModel(A, T)**: Agent A insertModels triple T If Agent A performed **Insert(A, T1)** and the inserting of T1 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 *foaf:Mammal*, addition of triple X Instance of *foaf:Person* leads to indirect addition of X *rdf:type foaf: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 Agent A inserts all the triples in {Tc} into the store together. It is possible that Agent 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 fof:mbox or foaf:mbox_sha1sum property .

3.2 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 Agent 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 Agent 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.

3.3 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.

3.4 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 Agent A can 'see' triple T, then Agent A can 'use' triple T but that is not the case. For example consider three triples T1, T2 and T3. Let us assume that you can infer T3 only by using T1 and T2. If Agent A can see T1 but cannot use it and can use T2 but cannot see it, then Agent A will not be able to see T3.

4. RDF Store Structure

An RDF store typically contains domain specific RDF schema and RDF data. In the RAP framework, the 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 will also maintain metadata about the triples in the store, like the creator of the triple

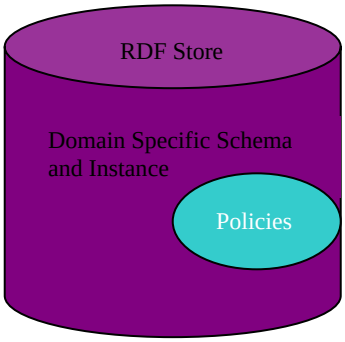

Figure 1: RDF Store

5. 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 [3,4], 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.

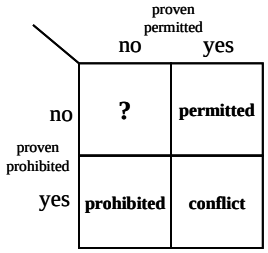

Figure 2. In reasoning about an action, four outcomes are possible. An uncertain or conflicted outcome may be resolved my meta-policy rules

The default policy, if specified, determines what happens in the upper left quadrant of the decision matrix shown in Figure 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)*).

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). 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. Conditions can be combined using Boolean AND (&), OR (|) operations.

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.

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.

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.

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.

Custom Predicates. There are certain custom predicates which might be helpful in writing access policies. Some of them have already been discussed such as createdNode(A,C), rdfs:subProperty(P,emp:salary). Another important predicate is schemaPredicate(P) which would return true if P is a predicate used to define RDF schema level information (e.g., rdfs:subClass, rdfs:domain, etc).

prohibit((insert(A,(?,P,?))) :- schemaPredicate(P).

This policy will prevent Agent A form changing the schema of the RDF store.

Delegation. As the Policies are represented in RDF and are stored in RDF store, delegation of policies can be achieved by creating Meta-polices, which are policies governing the policy triples in the store.

6. Architecture

We believe that the clients should be able to access the RDF store like any other website on Web. To enable this we propose the use of HTTP methods to access the RDF store.

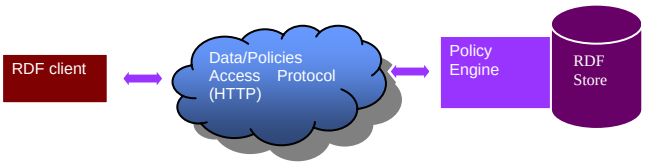

Figure 3: Proposed Architecture

HTTP seemed the optimal choice because of its synergy with current web and its wide acceptance.

We use the different HTTP Methods to access and modify the RDF store, the body of these methods would contain the XML serialized RDF.

The PUT Method is used for inserting the triples. All the triples that are to be inserted are sent in the body of the method. The store treats all these triples as one set and if that is prohibited, it then inserts each triple individually. All those triples which were prohibited from inserting are returned in the response message.

The Delete Method is used for removing the triples. The POST method would be used to query the store, the body of the POST method will contain the SPARQL query.

7. Status and conclusions

We have described a policy based framework to provide access and update control for an RDF store. Access and modifications are governed by a policy expressed as a collection of policy rules. Each rule defines a constraint on a class of actions that can depend on the actor and the content of the triples involved. The framework is currently being implemented using Jena [11].

8. REFERENCES

[1] Daniel Weitzner, Jim Hendler, Tim Berners-Lee, and Dan Connolly, Creating a policy-aware web: Discretionary, rule-based access for the World Wide Web. In Elena Ferrari and Bhavani Thuraisingham, editors, *Web and Information Security,*.

[2] Berners-Lee, T., Hendler, J., and Lassila, O. The Semantic Web, *Scientific American*, May, 2001.

[3] Kagal, L., Paoucci, M., Srinivasan, N., Denker, G., Finin, T., and Sycara, K. (2004). Authorization and Privacy for Semantic Web Services, IEEE Intelligent Systems (Special Issue on Semantic Web Services), July, 2004.

[4] Lalana Kagal (2004). A Policy-Based Approach to Governing Autonomous Behavior in Distributed Environments", Phd Thesis, Department of Computer Science and Electrical Engineering, University of Maryland Baltimore County, September 2004.

[5] J.M. Bradshaw, et al., (2003). Representation and Reasoning for DAML-Based Policy and Domain Services in KAoS and Nomads, Proceedings of the Conference on Autonomous Agents and Multiagent Systems, ACM Press, 2003.

[6] Claudio Gutierrez, Carlos Hurtado, and Alberto Mendelzon. Formal aspects of querying RDF databases, First VLDB Workshop on Semantic Web and Databases, Berlin, Germany, September 7-8, 2003

[7] Berners-Lee, T., Fielding, R. and Frystyk, H. (1996). “Hypertext Transfer Protocol” HTTP/1.0,” HTTP Working Group, Feb. 1996.

[8] Ora Lassila and Ralph Swick, Working draft, W3C, 1998. Resource description framework (RDF) model and syntax specification, Edit.

[9] Patrick Hayes, editor (2003). RDF Semantics, W3C Working Draft, 23 January 2003.

[10] Dan Brickley, R.V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema, W3C Working Draft 23 January 2003, Edit.

[11] McBride, B., Jena: a semantic Web toolkit, IEEE Internet Computing, v6n6, pp. 55-59, November 2002.