An RDF Modification Protocol, based on the Needs of Editing Tools

Fredrik Enoksson[1], Matthias Palmér[1], Ambjörn Naeve[1]

[1]Royal Institute of Technology(KTH/CSC),
Lindstedtsv. 5, 100 44 Stockholm, Sweden
{fen,matthias,amb}@csc.kth.se

Abstract. The use of RDF on the web is increasing, unfortunately the amount of editing tools suitable for end users without knowledge of technicalities of the language are not so common. We believe that a vital ingredient for the editing tools to flourish is a working remote modification protocol. This will allow editing tools to be developed separately from triple-stores and make them more flexible and reusable. Several initiatives for remote modification exists already but have not gained wide spread adoption. In this paper we will show that most of them fall short when it comes to edit arbitrary RDF construct, especially in combination with typical requirements of editing tools. We will first list these requirements, then propose a solution and finally outline an implementation. With this implementation we will also see how annotation profiles, a configuration mechanism for RDF metadata editors, has the additional feature of making modification requests very precise.

1 Introduction

RDF is increasingly used for expressing metadata on the web, consequently appropriate end-user editing tools are in demand. To become more user-friendly such editing tools need to hide the complex RDF-structure behind. This is typically achieved by displaying the metadata in forms and focusing on a single or a few central resources at a time. Beyond the user-interface layer such editing tools need to access a triple-store where the RDF is stored. A common solution is to use the triple-stores own API. Unfortunately this approach makes it harder to separate editing tools from the underlying triple-store which has a negative impact on both the variety of triple-stores as well as the success of well designed editing tools. Hence, a common protocol for remote modification of RDF is desirable. Earlier initiatives, such as [1] and [2], have not gained wide acceptance, probably due to the lack of a common query language which more or less is a prerequisite. Now, when SPARQL and the closely related SPARQL protocol for RDF endorsed by W3C has matured, the

prospect for a wide acceptance of a remote modification protocol looks brighter. A recent initiative, SPARUL [3], builds on top of SPARQL and looks promising, especially since much of the work done for supporting SPARQL in a triple-store can be reused. However, when taking a closer look, serious limitations from the perspective of an editing tool surfaced. In this paper we will first consider the requirements on a protocol that supports remote modification from the perspective of an editing tool, then list possible approaches for a remote modification protocol, and finally discuss the deficiencies in SPARUL and also outline a simple solution in accordance with the introduced requirements for editing tools.

2 Remote Modification Protocol

The following list of requirements are not complete. Rather, they represent requirements from the perspective of editing tools. Even though they might seem quite natural and simple, they are not so easy to fulfill. Especially when taking into account the common usage of blank nodes in RDF .

Resource centric – all kinds of subgraphs reachable (possible via intermediate blank nodes) from a named node (resource identified through a URI) should be modifiable.

Concise modifications – The modification requests should be concise, not inefficient by transferring to much or to little data at a time.

Without side effects – parts of the graph that are not to be modified should be left intact. More specifically, you should not be required to have knowledge of all parts of the graph to be able to modify it successfully.

Application independent – There should not be any built-in knowledge in the protocol of specific properties or resources.

2.1 Different approaches for a Remote Modification Protocol

The naive approaches for remote modification, i.e. updating one statement at a time or the entire RDF graph, are both flawed. Updating one statement at a time would yield a chatty protocol where a single update operation could require hundreds of requests. On the other side, updating the whole RDF graph could result in the transfer of large amounts of data where large parts are sent unmodified back and forth. It would also be problematic to support simultaneous updates with this approach as the entire graph would have to be locked.

A better approach could be to send only deltas (differences) between RDF graphs as described in [4]. The described strong delta is especially interesting as they can be applied to subsets of the whole RDF graph. Unfortunately, the outlined algorithm breaks down whenever there are blank nodes in the graph unless there are unique ways to identify them. (Breaking or leaving orphaned blank nodes in the graph is of course not acceptable, see the requirement 'without side effects'.) As the general problem of identifying them requires finding the largest common subgraph which is a graph isomorphism problem which in turn has been proven to be NP-complete, see discussion in [4], another path has been taken. Instead, the algorithm relies on knowledge of how the graph was constructed, i.e. according to which OWL ontology. Hence, if there are inbound functional or outbound inversely functional properties from a blank node to another identifiable node in the graph the blank node can also be identified. Unfortunately, there are a lot of real world situations when the data does not follow an OWL ontology, or even if there is an OWL ontology it may not be known by the tool or simply that there is not enough functional or inversely functional properties to uniquely identify the blank nodes. Taken together, this approach seem to be too brittle to base a remote modification protocol on.

Another approach is sending an easily identified subgraph that encompasses the modification. To avoid the problem of preserving identity of blank nodes, the subgraph should be defined in such a way that it shares no blank nodes with the rest of the graph. In the general case, this requirement could mean that the subgraph will be large, or even identical to the whole graph, for example if the graph consists solely of blank nodes. However, in nearly all real editing scenarios that we care about from the perspective of the requirement 'resource centric' listed above, there is a mixture of named nodes and blank nodes. Furthermore, the blank nodes are typically arranged into tree like data-structures that are reachable from the named nodes and not interconnected in a larger graph except via the named nodes. This is not something we postulate here, but rather a consequence of an established best practice where you name the resources that you express metadata on with URIs. From this observation we realize that a useful method of calculating a subgraph is the *anonymous closure* (or the closely related *concise bounded descriptions*), i.e. starting from a named resource and then recursively including all statements until other named resources are encountered.

It is important that the calculation of the subgraph is done deterministically, as with anonymous closure, it has to be done twice, first to be accessed for remote modification and second to be removed from the bigger graph before the modified subgraph is inserted. The alternatives, to calculating the subgraph twice, are either removing the subgraph directly or keeping a copy of the subgraph on the server side for later removal. The first alternative will yield a graph where data is missing from

time to time, this should clearly be avoided. The second alternative requires a stateful protocol and sessions to keep track of who initiated a modification on a specific subgraph. This should also be avoided both to avoid complexity and to be compatible with the principles of REST (sometimes referred to as the architecture of the web). Hence, the preferred alternative is calculating the subgraph twice.

Even though the anonymous closure subgraph approach is simple and solves the problem with graphs with blank nodes in it, it has to be complemented to allow 'concise modifications' of larger subgraphs including several named nodes. (See above for the 'concise modification' requirement.) If you know the names of all the named nodes you can simply take the anonymous closure of each and one of them. However, this is not always the case, instead you typically only know the name of one of the named nodes and how the other nodes are connected to it. In this case you have to express the relationship from the known named node to the other nodes in a query language and then require the anonymous closure of all the matches. With SPARQL this can be almost achieved with graph patterns and the DESCRIBE option. Unfortunately, DESCRIBE is formally undefined and left to the triple-store to implement, however, for general purpose triple-stores the anonymous closure algorithm or it's close counterparts seems to be frequently used.

2.2 Deficiencies in SPARUL

The main idea with SPARUL is to specify which statements to DELETE and which to INSERT into a specific model. In it simplest form the statements are simply listed, and the approach has the limitations of sending deltas without the elaborate scheme to identify blank nodes as discussed above. Hence, in this simple form, statements with blank nodes can be inserted but never removed or modified. In the more advanced case the delete and insert blocks contain templates which generate the statements to DELETE and INSERT via matching of a WHERE clause. Modifying subgraphs containing blank nodes is in this case possible but awkward as: First, it requires the WHERE clause to uniquely identify the right subgraph and capture the blank nodes in appropriate variables. Second the templates in DELETE and INSERT has to be carefully constructed to express the modification to be done. For simple and well known metadata and specific applications this is perhaps feasible. But in the general, the required algorithm would be complex and be done in every compliant client that want to use the protocol. We argue that a better approach is to allow the DELETE to be combined with DESCRIBE based on the variables matched in WHERE. In this case it will be possible to delete a series of anonymous closures

according to the principle described above and then list the modified subgraph to be inserted in INSERT.

3 Approach: calculating the subgraph twice

The solution suggested in the last part of section 2.1 will in the following subsection be described in a more detailed way. In following section 4 an implementation specific way of doing this in a dynamic way by reusing a configuration mechanism for RDF metadata editors.

3.1 Retrieving a proper subgraph to edit

In order to find the proper subgraph a resource is needed as a starting point. This the central resource from where to start the search of what related resources to be edited. Since not all the resources connected to the starting point is known a pattern has to be provided. This way arbitrary deep constructs can be retrieved by using the DESCRIBE query in SPARQL. As said before in this paper DESCRIBE is not formally defined and can therefore return different answer on different implementations. For the purpose of retrieving a proper subgraph all the direct properties of the resource, the Concise Bounded Description for the resource and all properties for the resources matched in the pattern needs to be returned. In the following an example will be given where we want to edit the dc:title, dc:creator and the dc:subject for a given resource. The RDF graph on the remote storage is depicted in figure 1

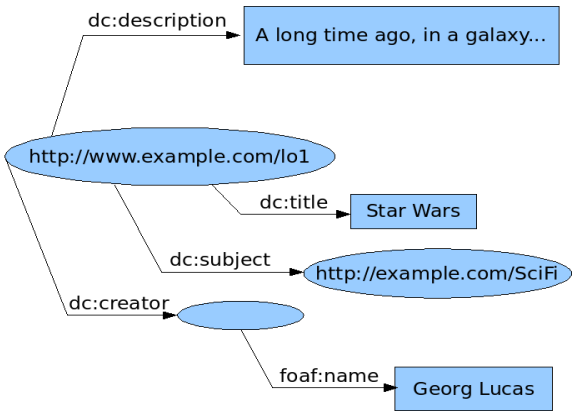

Figure 1: An example RDF graph on the remote storage

In order to retrieve the subgraph with a DESCRIBE-query the following question is enough:

```
DESCRIBE <http://www.example.com/lo1> ?creator
?subject

WHERE
{ OPTIONAL {
    <http://www.example.com/lo1>
      <http://purl.org/dc/terms/creator>  ?creator . }
  OPTIONAL  {
    <http://www.example.com/lo1>
      <http://purl.org/dc/terms/subject>  ?subject . }
  }
```

From the pattern in this example, a model with all properties and values for the resource http://www.example.com/lo1 will be returned, and, all properties and values for potential resources matching the variables *?creator* and *?subject*. The reason that we do not include a title property in the query is because it is a direct property. The reason why we include subject and creator is that they might point to non-anonymous resources. The use of OPTIONAL for every property also assures that if one property will not match, a graph will anyway be returned with all the properties if such a resource exist.

3.2 Updating the remote storage

When the retrieved model has been modified by the application the subgraph on the remote storage is to be updated with these modification. An update-request to the remote storage consists of two parts, first, an indication of which subgraph to remove and second, a subgraph to be inserted as a replacement. This subgraph have to be sent by serialising RDF into whatever serialisation is supported by the remote storage. The subgraph to remove can on the other hand be indicated by a query (that is, the same query used to retrieve it in the first place).

After the remote server have received the query to calculate the subgraph to remove and the model to insert, it first need to calculate from the query what to remove and from that remove the subgraph. After that the subgraph to be inserted can be put into the model on the remote storage.

4. Implementation

An implementation of the described approach in section 3 have been made by the authors of this paper. The remote storage used is version 3.1 of the *Joseki*[1] server and the editor application was implemented with the SHAME[2] code library. It is a code library used to built editors that can be configured with RDF metadata Annotation Profiles, as described in [5] and [6].

An RDF metadata Annotation Profile consists of a *Form Template* and a *Graph Pattern, w*here the latter can be used to fairly easy create a query used to retrieve the subgraph to edit from the remote storage. A Graph Pattern is expressed in an RDF query language like SPARQL, it defines the structure of the metadata to be edited and also act as a template when new metadata structures are to be created. The Graph Pattern is a tree structure where the root-node variable matches the resource to be edited and intermediate resources are variable nodes that matches nodes in the RDF-model. A query to send to the remote storage can be constructed from the Graph Pattern by removing the leafs from the Graph Pattern and send it as a DESCRIBE query.

As the Annotation Profile is constructed from the perspective of usability for the end users, you can expect that it encompasses a suitable subgraph to update in one operation. And as the DESCRIBE query is constructed from the graph pattern it will correspond to a *concise modification* (one of our requirements).

In the implementation this is sent to the Joseki server over HTTP, and returned is the calculated subgraph, or an empty graph if no subgraph can be calculated.

Once the model has been retrieved it is edited by applying the method that is described in [5] and [6]. The editing process, described shortly here, will match the Graph Pattern against the retrieved subgraph, and create a binding to the matching variables. The bindings are combined with the Form Template that makes it possible to create a Graphical User Interface that also hides all the complexity from the end user. This makes it possible for the end user to modify the subgraph in a rather simple form-based manner.

When the changes to the subgraph are finished, ie the end user decides to save the modifications, the modified subgraph is serialised into RDF/XML and the query to calculate the original subgraph is expressed as a DESCRIBE query in SPARQL, as said before is the same as the one to retrieve it. For the Joseki-server to perform the update we have to implement and add a service called Update that is called with these two arguments. On the Joseki server the operation of removing the calculated

1 http://www.joseki.org

2 http://kmr.nada.kth.se/shame

subgraph is performed first followed by inserting the modified subgraph. Joseki is using Jena to handle RDF and the two operations is implemented using the methods *add* and *remove* defined in the interface Model in the Jena API.

5. Conclusion

In this paper we have described several initiatives/approaches to remotely edit RDF graphs, according to the requirements of being *Resource centric***,** allowing *Concise modifications***,** being *Without side effects***,** and being *Application independent*, We specifically discussed a recent initiative, SPARUL, that are very promising but falls short regarding handling blank nodes properly in combination with the requirements. The only approach that could meet all the requirements for the needs of an editing tool was the one described in section 3. The approach relies on that most graphs consists of a mixture of blank and non blank nodes and that replacing whole subgraphs calculated from one or a few starting points corresponds well to the extent of a typical update. The update is performed by retrieving a subgraph, modifying it and then submitting a the modified subgraph back to replace the original.

In addition to the modification mechanism, the paper has shortly outlined how to use Annotation Profiles, a configuration mechanism for remote editing tools, to retrieve the subgraph needed for editing. Consequently, if such an approach is used, the modification protocol will work automatically, avoiding manual construction of queries.

A rather important issue that is out of the scope of this paper is how to handle concurrent changes of the same subgraph. The subgraph is not locked after the first retrieval of it and changes could be done inside the same subgraph by another user. Additional to the hassle of updates being overwritten falsely, this might lead to inconsistencies in the RDF, since differences in the queries used to extract the subgraph may lead to orphaned constructs. Some kind of additional restrictions on the queries may be needed for a consistent locking mechanism to be feasible.

Acknowledgement

This work has been carried out with financial support from the EU-FP6 project LUISA, which the authors gratefully acknowledge.

References

1. Seaborne, A.: An RDF NetAPI, Proceedings of the First International Semantic Web Conference on The Semantic Web, 2002
2. Nejdl, W., Siberski, W., Simon, B., Tane, J.: Towards a Modification Exchange Language for Distributed RDF Repositories. Proceedings of the First International Semantic Web Conference on The Semantic Web, 2002
3. Seaborne, A., Manjunath, G., SPARQL/Update A language for updating RDF graphs, http://jena.hpl.hp.com/~afs/SPARQL-Update.html
4. Berners-Lee, T., Connolly, D., Delta: an ontology for the distribution of differences between RDF graphs, http://www.w3.org/DesignIssues/Diff.
5. Palmér, M, Enoksson, F., Naeve, A., D3.2: Annotation Profile Specification, Retrieved, August 30, 2007, from http://www.luisa-project.eu
6. Palmér, M, Enoksson, F., Nilsson, M., Naeve, A., Annotation Profiles: Configuring forms to edit RDF, Proceedings of the Dublin Core Metadata Conference, Singapore, 2007