RDFSync: Efficient Remote Synchronization of RDF Models*

Giovanni Tummarello[1], Christian Morbidoni[2], Reto Bachmann-Gmür[3], and Orri Erling[4]

[1]Digital Enterprise Research Institute, Galaway, Ireland
[2]SeMedia, DEIT, Università Politecnica delle Marche, Ancona, Italy
[3]Talis Information Limited, Birmingham, U.K.
[4]OpenLink Software
g.tummarello@gmail.com, c.morbidoni@deit.univpm.it,
reto@gmuer.ch, oerling@openlinksw.com

Abstract. In this paper we describe RDFSync, a methodology for efficient synchronization and merging of RDF models. RDFSync is based on decomposing a model into Minimum Self-Contained graphs (MSGs). After illustrating theory and deriving properties of MSGs, we show how a RDF model can be represented by a list of hashes of such information fragments. The synchronization procedure here described is based on the evaluation and remote comparison of these ordered lists. Experimental results show that the algorithm provides very significant savings on network traffic compared to the file-oriented synchronization of serialized RDF graphs. Finally, we provide the design and report the implementation of a protocol for executing the RDFSync algorithm over HTTP.

Keywords: RDF synchronization algorithm, MSG, graph decomposition.

1 Introduction and Definitions

Remote synchronization of data files is a procedure by which local information (e.g. A data file) is updated over a network in order to be made identical with a remote one (or vice versa). Synchronizing could be trivially achieved by copying the entire remote file locally and then comparing it with the local one, but this is largely undesirable due to the performance issues in comparing the entire data file and most of all due to the bandwidth cost of frequent full data transfers.

In 1998, the rsync algorithm was developed [1] to efficiently synchronize remote binary files. rsync operates under the assumption that the changes will be significantly lower in size compared to the data file itself and that these are likely to happen in "clusters", that is, in localized spots rather than distributed across the file. When this is the case, rsync can achieve synchronization by transferring data in quantity just slightly higher than the size of the changes. As such, rsync and others comparable algorithms that later followed are today the backbone of data replication across the Internet.

*The work presented in this paper was supported (in part) by the Lion project supported by Science Foundation Ireland under Grant No. SFI/02/CE1/I131 and (in part) by the European project DISCOVERY No. ECP-2005-CULT-038206.

K. Aberer et al. (Eds.): ISWC/ASWC 2007, LNCS 4825, pp. 537–551, 2007.

In this paper we provide an algorithm for the efficient synchronization of RDF models. RDF Models cannot be efficiently synchronized by the rsync or similar algorithms due to the RDF semantics itself. Serializing an RDF model could in theory result in a factorial number of ordering for the composing triples and even more when blank nodes are involved. Remote RDF synchronization has been highlighted as a very important but open problem [2].

One could think of serializing the graph into a deterministic, canonical way, e.g. by ordering the triples in lexicographical order. This is partially possible as we will see, but the results of a simple rsync synchronization will be shown to be still unsatisfactory, especially when graphs contain blank nodes (e.g. FOAF personal profile documents, OWL/RDFS ontologies, etc.).

Our proposed algorithm, which we call RDFSync, works by decomposing RDF graphs in minimal subsets of triples and then creating hashes that can be sorted and efficiently synchronized. In this way, we will show how the RDFSync algorithm can exhibit on RDF models the same behavior of rsync for general data files: as long as the number of triples involved in changes is small compared to the size of the entire model, as is the case for frequent updates, the network traffic and the computational power required for the synchronization will be very small.

With more and more data being made available on-line on the Semantic Web as HTTP retrievable data (open linked data model or RDF dumps), the RDFSync algorithm seems of great importance to enable the Semantic Web itself to scale. With respect to this, we conclude this paper by illustrating how the RDFSync algorithm can be transparently exposed by a web server providing its service over HTTP and triggered by a standard content negotiation procedure.

The paper is structured as follows: the basic theory for the decomposition of RDF models into Minimum Self-contained Graphs (MSG) is illustrated in section 2. This will be used to derive methodologies for remote RDF synchronization, illustrated in section 3. Experimental results will be illustrated and discussed in section 4, while additional issues are covered in section 5. Section 6 will illustrate the algorithm as implemented on top of HTTP.

1.1 Different Sync Modalities

Unlike rsync, which can only be used to make remote files identical, RDFSync is capable of performing different “kinds” of synchronization between a *target* and a *source* model.

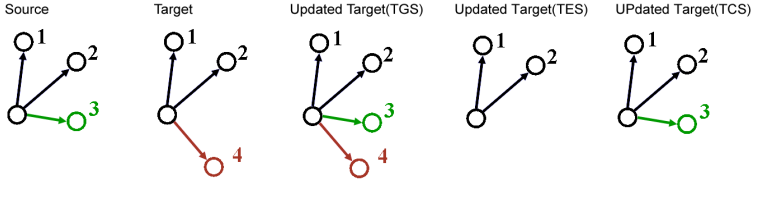

Fig. 1. Starting from two graphs and considering one of them the 'source' and the other the 'target', we show how the target will be modified by running RDFsync in different modes. Edges and nodes with the same color and number are equals, different colors and numbers represent different RDF statements.

Using the algorithms we propose, it will be possible to cause the *target*:

- to be equal to the *merge* of both graphs (Target Growth Sync, TGS);
- to delete information that is not known by the *source* (Target Erase sync, TES);
- to be equal to the *source* (Target Change Sync, TCS).

Figure 4 shows these different kinds of exchanged and the resulting RDF graphs. Here, the definition of *merge* and *equals* are strictly derived from RDF Semantics [3]. This means in practice that:

- B-nodes IDs will not be preserved;
- Sync is not required to transfer redundant information that might be contained in the graphs. This means that the only *lean* versions of two graphs (as defined in [3]) are required to be isomorphic for the graphs to be considered equal;
- Serialization format idiosyncrasies (e.g. RDF/XML comments) are ignored.

2 The Minimum Self Contained Graph Theory

In this section we will illustrate the Minimum Self-Contained Graph (MSG) theory. This and the following section expand on [4] by considering additional cases and conditions. The theory set forth in [4] is here reported in full to serve as base for the discussion of the RDFSync algorithm.

Let's first define what is the minimum "standalone" fragment of an RDF model. As blank nodes are not addressable from outside a graph, they must always be considered together with all surrounding statements, i.e. stored and transfered together with these. MSGs are the smallest components of a lossless decomposition of a graph which does not take into account inference (e.g. the OWL methods for detecting identity of nodes). Discussion about RDFsync with respect to inference is given in section 5.3.

We will here give a formal definition of MSG (Minimum Self-Contained Graph) and will prove some important properties.

Definition 1. *An RDF statement involves a name if it has that name as subject or object.*

Definition 2. *An RDF graph involves a name, if any of its statements involves that name.*

Definition 3. *Given an RDF statement* ***s****, the Minimum Self-Contained Graph (MSG) containing that statement, written* ***MSG(s)****, is the set of RDF statements comprised of the following:*

- *The statement in question;*
- *Recursively, for all the blank nodes involved by statements included in the description so far, the MSG of all the statements involving such blank nodes;*

This definition recursively builds the MSG from a particular starting statement; we now show however that the choice of the starting statement is arbitrary and this leads to a unique decomposition of any RDF graph into MSGs.

Proposition 1. The MSG of a ground statement is the statement itself.

Theorem 1. If s and t are distinct statements and t belongs to MSG(s), *then MSG(t) = MSG(s).*

Proof: *We first show that MSG(t) ⊂ MSG(s) and then MSG(s) ⊂ MSG (t). MSG (t) ⊂ MSG (s): This is straightforward from Definition 3 and knowing that t ∈ MSG(s). MSG(s)⊂MSG(t): Let t be a ground statement. Then MSG(t) = t for Proposition 1. But then t ∉ MSG(s), which contradicts the hypothesis t ∈ MSG(s).*

Hence, t is not a ground statement. Then there is either a single blank node connecting the statements t and s (e.g., writing statements in n3, t=":a :p :_id1", s="_:id1 :q :b") or there is a sequence of blank nodes connecting t and s. In the first case, there is a blank node of s involving t (by hypothesis, since t ∈ MSG(s)). So s ∈ MSG(t) and MSG(s) ⊆ MSG(t). In the second case, there is a blank node of s involving MSG(t), also in this case s ∈ MSG(t) and MSG(s) ⊆ MSG(t). In other case, where s does not involve any blank node of t or of MSG(t), the hypothesis t ∈ MSG(s) is contradicted.

Theorem 2. Each statement belongs to one and only one MSG.

Proof: *Let $s \neq t \neq u$ be distinct statements, and let s belong to different MSGs: s ∈ MSG (t) and s ∈ MSG (u). By Theorem 1, MSG (s) = MSG (t) and MSG (s) = MSG(u), hence MSG(t) = MSG(u), so the three MSGs considered are actually the same.*

Corollary 1. An RDF model has a unique decomposition in MSGs.

Proof: *This is a consequence of Theorem 2 and of the determinism of the procedure.*

3 MSG Based Graph Decomposition and Merging

As a consequence of *Corollary 1*, after a graph has been decomposed into MSGs, it can be incrementally transferred between parties with granularity down to one MSG at a time. As a consequence of theorem 2, such a transfer would be maximally network efficient since statements would never be repeated. While the above results intuitively apply to most of the graphs, there are a few special cases which require particular attention.

3.1 Non-lean Graphs

In the same way as in relational databases table entries are never duplicated, RDF graphs are defined as being set of triples rather than a collection of triples, that is, triples are never duplicated. However, even without containing duplicate triples an RDF graph may contain redundant information.

The reason of this is that the RDF semantics gives blank nodes the meaning of existentially quantified variables, so while it is legal to have a graph containing multiple isomorphic subgraphs, such a graphs is said to be 'non lean' as it expresses exactly the same knowledge as it would have after removing all but one of the isomorphic subgraphs. In other words, a non-lean graph is a graph in which some of its triples can be removed without changing the meaning of the information expressed in the graph.

In the case of a non-lean graph, the decomposition into MSGs could result in a certain amount of MSGs to be completely indistinguishable if it were not for the ids of the blank nodes they contain, so they are isomorphic. In such situations the MSG is said to be 'repeated' in the decomposition.

While such a redundant information can be kept in RDF serialization formats and most RDF triplestores currently available do not remove redundancies, implementations of RDFSync are not required to transfer redundancies: in fact the only reason for an RDFSync implementation to keep redundancies in some cases is the computational cost of complete leanification.

In the case of the decomposition into isomorphic MSGs a partial leanification can be done in the interest of saving in terms of network bandwidth and storage requirements as more convenient processing of the target graph(s).

For these reasons RDFSync considers two graphs equivalent even if the same MSG is repeated in one of the decompositions and not in another one: this might result in two graphs with different number of triples after the synchronization, but still the lean versions of the resulting graphs will be isomorphic.

3.2 Removing Redundant MSGs

There are other cases in which, with respect to the RDF semantics, an MSG can be removed from a decomposition without changing the meaning of the graph resulting from merging the MSGs. This happens when the graph resulting from removing an MSG *a* (i.e. the union of the remaining MSGs) has a subgraph which is an instance of *a*, in which case *a* can be removed without changing the content expressed by the graph. With respect to RDF semantics the MSG *b* is an instance of the MSG *a,* if they are isomorphic except for a number (zero or more) of blank nodes in *a* which correspond to grounded nodes in *b*.

In these cases, for the purpose of RDFSync, the MSG can be safely removed. Note however that other applications of MSGs, such as tracking versions and provenance, which are outside the scope of this paper, may not allow removing such MSGs. So in the general case the identity criterion of a set of MSGs is not the same as of the graph resulting from the merge of this set.

3.3 Canonical Serialization of MSGs and MSG's Hashes

MSGs are standalone RDF graphs and, as such, they can be processed with algorithms like canonical serialization in order to provide a sort of digest or hash value of the graph (as discussed in [4]). We use an implementation of the algorithm described in [5], which is part of the RDFContextTools Java library[1], to obtain a canonical string representing the MSG and then we hash it to an appropriate number of bits to reasonably avoid collisions (math and common sense say 128-160 bits will suffice). This hash acts as an unique identifier for the MSG.

There might be issues in certain situations where the canonical serialization described in [5] will behave in a non-deterministic way. It can happen for some graph structures involving several bnodes, especially where bnodes have no property or label attached. Such cases are luckily not very frequent in real-world use cases and in particular in the RDF graph we experimented with (see section 4).

As there is a finite number of possible serialization alternatives, it is always possible to compute all the hashes that an MSG, in one such very hard case, has and treat them as being equivalent in the synchronization procedure.

[1] http://semedia.deit.univpm.it/tiki-index.php?page=RdfContextTools

3.4 Canonical Serialization and RDF Graphs Synchronization

As a graph can be decomposed unequivocally into a set of MSGs, it can be canonically represented by the ordered list of the identifiers (hashes) of its composing MSGs. In the RDFSync algorithm, such lists are created independently at each end and ordered by the binary value itself. The synchronization is then performed in 2 steps:

1. A diff between the source and the target ordered lists of MSGs is performed;
2. Such diff indicates which MSGs have to be requested from the other side and which should be deleted in the local model.

To perform the diff between the source and the target ordered list of MSGs, we first need to transfer or locally reconstruct the remote list. For this purpose, two procedures can be employed: the first, trivial one is to directly transfer the list, the second is to create a copy of the remote list, using the standard rsync, from the local list.

The latter approach can be shown to be highly efficient in case of small differences between the two lists, since rsync is optimized for differences which result in shifting of data blocks within the file. This is the case when small differences exist between a source and a target model; the few added or removed MSGs in the ordered list have the effect of "shifting" the remaining MSG hashes.

In case of MSGs hashes lists, big changes (e.g. a lot of MSGs added to one model) result in a great amount of hashes to be inserted in random positions of the list, which cause almost all the file to be transferred, plus, of course, the overhead of the rsync operation (calculating hashes of file sections, transferring and conparing them). The experimental data, as shown in the next section, can be used to select the best approach to follow, given an approximate estimate (e.g an expectation) of the amount of differences between the models (e.g. one would choose the rsync-based synchronization if the RDFSync is executed very often compared to the changes in the model).

Once the two lists are available to the target host, a diff is computed in order to obtain:

- The list of MSGs to be requested from the remote model (in case of a TCS or TGS sync), which is then sent to the remote host which complies to the request.
- The list of MSGs to be deleted in the local model (in case of a TCS and TES sync).

Figure 2 illustrates this process, showing how the different RDFSync methodologies can be applied to two remote MSG lists.

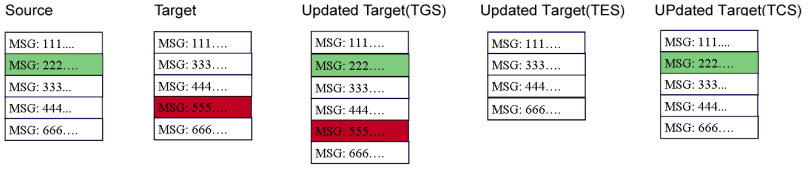

Fig. 2. Two ordered lists of MSGs are processed by RDFSync in different modes, producing different results at the target side

[2] http://wymiwyg.org/rdf-utils

4 Experimental results

In this section we show and discuss the results obtained from actual runs of RDFSync, as implemented in the RDFContextTools. Runtime results of this implementation (the synchonized graphs) have been validated with the diff utility independently developed within the 'RDF Utils' project[2]. We show the performance of the algorithm in three notable cases.

One, labeled *SyntGraph no bnodes,* deals with a synthetically generated graph composed completely by ground triples. This graph is 1.07 MB in size and is composed by 8000 triples (therefore 8000 MSGs). The performance as evaluated on this graph is completely comparable with any other made completely of ground triples such as DBPedia dataset. The second one, labeled *SyntGraph bnodes*, also deals with a synthetically generated graph, this time with a moderate number of blank nodes (approximately 600, with MSGs composed by 2, 3 and 4 triples). The graph is 1.3 MB in size and has 9000 triples in 7800 MSGs. The third one, labeled *DBWorld Graph* is a fragment of a real world graph which makes a more extensive use of blank nodes[3].

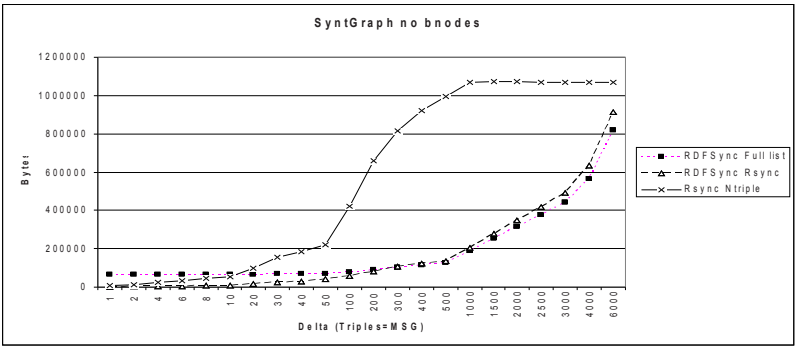

Fig. 3. Traffic vs Delta MSG for a graph that makes no use of bnodes

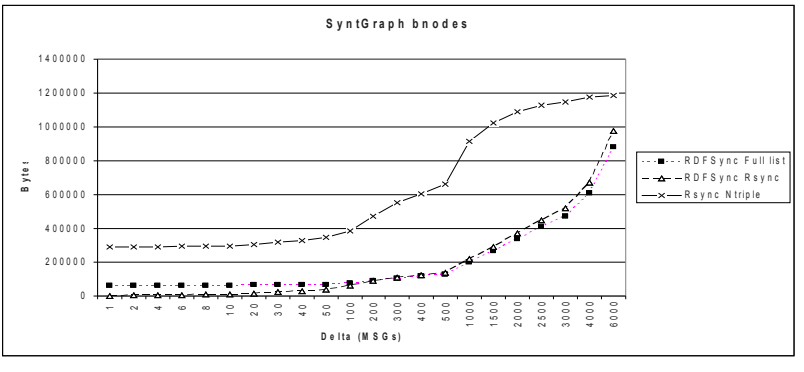

Fig. 4. Traffic VS Delta for a graph with a moderate number of bnodes

[3] http://sw.deri.org/~aharth/2005/08/dbworld/dbworld.rdf, representing calls for papers in RDF.

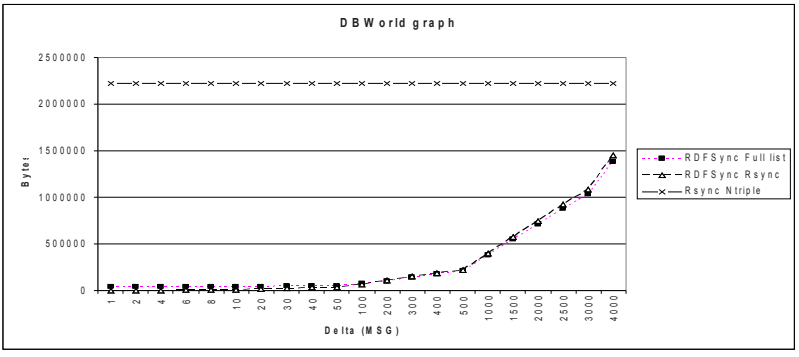

Fig. 5. Traffic vs delta MSGs for a graph that makes intense use of bnodes

This graph is 2.1 MB and contains approximately 13000 triples in 5000 MSGs. Performance on this graphs will be comparable with that on others with similar characteristics e.g. DBLP dump in RDF[4].

The algorithms that we compare are:

- *RDFSync Full list*: by graph decomposition we produce a list of 64 bits MSG hashes. This is entirely copied on the other side and then the missing ones are requested;
- *RDFSync rsync*: the list of hashes, created as above, is synchronized itself with rsync. The missing MSGs are then copied;
- *rsync:* rsync is applied on a lexicographically sorted list of triples (Ntriples).

In every case the traffic shown is inclusive of all the algorithm overhead (hashes synchronization plus the actual transport of the MSGs serialized in Ntriples).

The scale is the same for all the graphs and is pseudo logarithmic for the delta axis while linear on the traffic. As the lines are close in the SyncGraph no bnodes graph for the first part (low delta), the numeric results are reported in Table 1.

Table 1. The data from the low delta runs on the SyncGraph no Bnodes

Delta(in MSGs)	RDFSync Full list	RDFSync Rsync	Rsync Ntriple
1	64126	1895	7403
2	64250	2743	12535
4	64504	4445	22805
6	64756	5693	33073
8	65008	7841	43341
10	65258	9539	53607
20	66506	17459	99935
30	67764	25801	156273
40	69036	28667	182599
50	70290	39237	221999
100	76582	57169	420507

[4] http://lists.w3.org/Archives/Public/www-rdf-interest/2004Dec/0015

As evident from the experimental data, applying the proposed algorithm gives very high bandwidth saving as opposed to the alternative rsync Ntriple algorithm in almost all cases. When bnodes are used in the graph, the difference is as much as the entire graph size for any delta (*DBWorld Graph*). This can be explained with the blank nodes Ids (which are usually random generated by the triplestores) that have to be synchronized at the both ends. If these are too many, there will be more than one change per rsync block size thus causing the transfer of the entire dataset.

Performances are dramatically different also when a small number of blank nodes are used (*SyntGraph bnodes*) as the blank node impose an almost constant weight which is otherwise not present for the RDF sync algorithm (especially in its rsync format). The different for small updates is huge, as much as 150 to 1 for a single delta MSG (1.8 k on the RDFSync algorithm vs 290k of rsync).
The most "difficult" case is the *SyntGraph no bnodes*. Even in this case however, RDFSync outperforms rsync approximately 4 to 1 (for the single delta MSG) to reach approximately 8 to 1 past the 100 MSG mark (which is only approximately 1.2 percent of the graph size).

With respect to the difference between sending the entire list or not, these data shows that this is seldom a good strategy (if not in cases of very large expected delta).

4.1 Optimizations

The data transfer can be further optimized by using weaker hashes (a lower number of bits) with a master hash calculated at both ends list of hashes is as supposed to(no collision has detected). With this procedure in place, the MSG hashes could be of 32 bits, thus cutting the list size in half, at the sole cost of having to redo the sync with stronger hashes in very rare cases when the weak ones fail.

5 Additional issues

5.1 Complexity

Computationally speaking, evaluating the MSGs each time one wants to RDFSync is a time consuming procedure; although such algorithm is O(n) with n the number of triples, the large number of API calls to explore the graph usually determines a slow execution. The solution is to use caching techniques so that each time a new information is inserted, the MSG is calculated. MSGs are by definition immutable (mutating one can be seen as removing a statement and adding another one), so this is possible to cache them with great efficiency.

In the implementation constructed using OpenLink Virtuoso[5], the MSG hashes are calculated in a server side stored procedure, saving on client-server communication delays. For graphs involving relatively few blank nodes, the MSG calculation time is near linear since all triples of a subject are read serially and translated from internal ID to the IRI (Internationalized Resource Identifier)[6].

[5] http://www.openlinksw.com/virtuoso/
[6] http://www.w3.org/International/O-URL-and-ident

With a 2GHz Xeon processor, about 13000 MSG's can be calculated per second, provided a warm cache. An auxiliary table is used for sorting the MSG hashes, with a two-part key, consisting of a graph ID and MSG hash and a dependent part of subject, predicate and object. Keeping an up to date MSG table increases storage consumption by about 25%. Retrieval of MSG's requires only a lookup of this table and typically no extra joining.

For graphs with hundreds of millions of triples, making the MSG list is a time consuming process. This can be alleviated by keeping an update log table which records additions and deletions of triples with an associated timestamp. These changes can then be reflected on the MSG table and removed from the log table. The maintaining of a log table adds an overhead of approximately 15% to the insert or delete times and adds practically no IO since the additions are always in ascending order.

5.2 Non lean MSGs

It can happen, in situations in which an MSG is not lean, that is it has a duplicated statement where, for example, the subject and the predicate are identified by the very same URI but the objects are two distinct blank nodes. In this case, this MSG can be considered to be equivalent to a lean version (where one of the duplicated statements is missing), but the hashes of the two MSGs turn out to be different.

This can be avoided by leanifying each MSG before carrying out the hash, a procedure that will be addressed in future versions of the RDFSync implementation.

5.3 Shortcomings of the Heuristic Canonical Serialization

It is possible that, by using the canonical serialization as described in [5], two isomorphic MSGs give two different hashes. This is true in cases where many consecutive blank nodes are used and these are connected to the same URI. In this case, labeling of blank nodes becomes difficult and a 2 step algorithm can fail. It is conjectured that an N step canonical serialization algorithm could probably take care of all these cases but as it is not proven and as the complexity of such algorithm would necessarily be higher, for RDFSync we follow a different approach.

In case an MSG is in fact serialized differently at the two ends, this would show as "missing" from one side and "added" from the other. Once transferred, the "additional" one is in fact checked against all the "missing ones" for isomorphism in order to rule out such case. Such checking is usually very fast given that MSGs are usually small, furthermore a simple node count and or a simple lexicographical triple ordering can rule out trivial cases.

5.4 Inferencing

It is to be noticed that, beyond RDF-entailment, this paper does not take into account other reasoning capabilities as provided by RDFS [6] and OWL [7]. The algorithm is therefore applicable to "base" RDF models, that is, sets of triples with no consideration for automatic inferences (e.g. same-as or inverse-functional-properties).

Under a certain point of view, this is somehow "cleaner" as it does not require that the sending and the receiving party agree on ontologies. In particular scenarios, however, the decomposition into MSGs could lead to very big chunks of data, for

example for databases which contain a lot of anonymous resources identified by inverse functional properties. In this case the MSG decomposition would lead to bigger components than a decomposition taking into account (inverse) functional properties.

An example of such a store would be a graph obtained my merging a number of personal FOAF files, where bnodes are often used to represent a person. In such cases the RDFSync procedure would still give correct results but the decomposition would degrade in efficiency, generating big MSGs containing information about more than one person (e.g. sub-groups of persons connected by *foaf:knows* properties). To avoid this, inverse functional properties can be used to stop the recursion in the MSG definition, similarly to what suggested in [8].

5.5 Atomicity and Integrity

Given that the MSGs hash are content based, the RDFSync algorithm poses no real questions of atomicity and integrity. As each MSG is atomic per se, as long as one arrives entirely it can be added to the synchronizing database. If the flow of MSGs is interrupted, the synchronization can be restarted and it would only cover the missing MSGs (it would not repeat those that were previously transferred).

6 RDFSync over HTTP

RDFSync, as implemented to perform the experiments in the previous section, operates over a TCP/IP connection. In this section we will however illustrate a modification of the RDFSync algorithm to operate on top of HTTP. Such modification has very important practical consequences: it is well known in fact that the modern Internet highly favors (if not enforces) HTTP over any other protocol as a way to reach out from any local network configurations (e.g. local firewalls). So, while the TCP/IP based implementation is certainly useful for server to server scenarios, the HTTP version is practically fundamental for client to server scenarios.

The main challenge of the HTTP based synchronization protocol is comparing large graphs and efficiently detecting a usually small set of differences in the MSG checksum list by using a stateless request/response paradigm.

In the following, *receiver* refers to the party initiating the synchronization and being updated, while *sender* is the party providing the data. The protocol does a sequential merge of the sorted MSG checksum lists on either side: since the lists often have largely the same content, the protocol keeps a pointer on the receiver and another on the sender. The pointers are said to be in sync if they point to the same checksum on either side.

The lists of MSG checksums are divided into blocks of 256 checksums. This works reasonably if there is a difference between the lists every 1000-2000 checksums, but can be varied based on difference estimates (e.g. based on the time since the last update).

The synchronization step consists of the following:

- The receiver sends the checksum at its sync pointer plus a set of block checksums for checksum blocks starting at its sync pointer;
- The sender replies with the offsets of the blocks that are different;

- The receiver sends the checksum blocks for the blocks that were different;
- The sender replies with the MSG serializations that are needed to sync the differing blocks. The message also contains the highest checksum covered in the step, indicating the sync pointer value on the sender.

The client sets its sync pointer to the sender's sync pointer. If the receiver has this same checksum, we repeat the process. If the receiver has a next checksum that is higher then the sender's next checksum, the receiver requests the MSG serializations with checksums between the receiver's sync pointer and the sender's sync pointer. If the receiver has no further checksums, it asks for the remaining MSG serializations. When both sync pointers are at end the process is completed. Details of special cases are omitted for brevity.

As mentioned, such sync process is stateless. It can be repeated any number of times and will terminate rapidly if both sides are already in sync. Locking of data in the data store is usually minimal since the transaction unit is small.

6.1 Performances

With the corpus of DBPedia[7], we have about 30 million triples accumulated over several years of Wikipedia editing. These are derived from 1.6 million articles. Assuming a rate of change of 20% of articles per year and assuming a change influenced half the triples in each, we'd have 3 million changed triples per year. This would be an average of 8219 per day, or that is a checksum every 3650 checksums would change on the average. If we had a checksum block of 100 checksums, we could send units of 4000 checksums, represented as 40 block checksums of 100 checksums, equals 320 for 8 byte checksums plus overhead and expect one mismatched block in each. We would exchange the information in the mismatched block, amounting to one serialized MSG, under 100 bytes compressed. Thus, we would cover an average of 3650 MSG's with 2 round trips and about 500 bytes. Transferred. Thus, for a day's worth of changes, we'd have 16438 messages totaling approximately 8.2 megabytes. A full HTTP download of the same dataset would require 1.6 Gigabytes of traffic.

7 Specific Applicability

The Semantic Web is centered around the idea of distributed and cooperative metadata production and consumption, hence the remote RDF model synchronization is clearly an important goal to achieve. This is even more so if one considers that most of the semantic web applications currently known (client side, such as Piggy Bank[8], Tabulator[9], DBin[9]) or server side work by downloading remote RDF data locally in order to then perform inference, queries and data merging as needed. To make sure that the local data is up to date with the remote data, such applications today follow the "RSS approach" and continuously download the file to get updates. Clearly this is not a scalable approach when databases of considerable dimensions are involved, hence the use of the RDFSync procedure to enable frequent updates at low cost.

[7] http://dbpedia.org/

RDFSync can also be used to create powerful centralized RDF based services, by enabling efficient synchronization across a grid structure. Graph synchronization might be useful in the aggregation of news feeds available in RDF, such as RSS 1.0 [10] or more recently as AtomRDF[10] or AtomOWL[11]: since the same news is often published in different feeds, using RDFSync would prevent multiple downloading of the same entry. Another context of applicability, which has strong similarities with the one just described, could be within systems like URIQA [11], or in general following the linked data paradigm, where a server answers to client's requests about a URI/URL, returning a subgraph representing the knowledge of the server about the URI/URL. Using RDFSync in this case could allow clients to efficiently update their knowledge about a resource from an authoritative server.

As said in section 1.1, RDFSync-based update operations can be performed in different modalities. These modalities address distinct scenarios and social environments. The TGS mode, for example, perfectly fits a scenario like the one described in [12], where peers typically want to know everything that has been said around a topic. The TGS modality would be used to merge the news from different sources (feeds) into one target model. The TCS mode could be preferable in a URIQA like scenario, where information owned by the server are to be considered authoritative and clients might want to act as 'mirrors', thus deleting triples which are no more on the server.

Furthermore we think RDFsync might prove very valuable in mobile or wireless environments, where bandwidth efficiency issues are even more important.

8 Related Works

While the problem of finding diffs and generating patches for RDF graphs has been investigated, to the best of our knowledge, our algorithm is the first to address their efficient remote synchronization.

An approach to diffs and patches of RDF graphs is described in [2], where the authors introduce the concept of functionally ground nodes, which are blank with an inverse functional property value. Functionally ground nodes behave like named resources, once an ontology has defined which properties are inverse functional, hence the diff algorithm becomes straightforward. This approach works under some assumptions on the RDF graph it applies to. All the nodes have to be ground, or functionally ground so it is not applicable to the general case, where blank might be non functionally ground (e.g. the 'root node' of an rdf:Bag construct). Furthermore, the methodology relies on the OWL layer for identifying the inverse functional properties, limiting its scope to scenarios where an ontology has been previously agreed upon and is owned by each party involved in the exchange.

In this paper we considered the case of generic RDF graphs and we intentionally based our approach on the RDF layer as we think that ontologies and reasoning capabilities should be applied at a different level. Choosing the appropriate ontology (or ontologies, as data could be heterogeneous) to present, edit and interact with the data, should be left, in our opinion, to domain aware applications, which could make use of RDFSync, at a lower level, for import and merge 'pure' RDF.

[8] http://simile.mit.edu/wiki/Piggy_Bank
[9] http://www.w3.org/2005/ajar/tab
[10] http://djpowell.net/blog/entries/Atom-RDF.html
[11] http://atomowl.org

The idea of decomposing an RDF graph in small parts, which in our case are MSGs, has been investigated also in [8], where the concept of "RDF molecules" is introduced in order to obtain smaller components by taking functional and inverse functional properties into account. The price to pay for this approach is an higher complexity of the decomposition and of the algorithms to achieve it, as well as the requirement of a shared set of ontological beliefs. Furthermore, the original concept of molecules does not provide a deterministic decomposition so that extensions had to be defined to allow this decomposition in implementations such as the RDF diff and patch utility in WYMIWYG rdf-utils[12]. This approach has been further developed by HP laboratories in Bristol, leading to the Graph Versioning System (GVS)[13].

Many works in literature deal with the problem of synchronizing different 'versions' of the same structured, XML based knowledge. [13], [14] and [15], describe algorithms and methodologies to merge and patch XML structured documents, both in the case of two-way synchronization (where the two files are independently created) and three-way synchronization (where the two files are separately obtained refinements of the same source file, that is available to the merging algorithm).

These algorithms are based on the analysis and comparison of DOM structures of the documents or of sequence and order of XML tags and PCDATA. These methods are not efficient if applied to RDF/XML, as RDF can be serialized in different ways and two different serializations could have the same information content.

9 Conclusions

We described a methodology to perform an efficient synchronization of RDF models called RDFSync. RDFSync is based on RDF Semantics only and it is therefore a general purpose tool independent of the application domain and independent of the used ontologies. The sync acts purely on the level of the content expressed by RDF graphs as defined by RDF Semantics.

Experimental results show that the algorithm provides very significant saving on network traffic compared to a simple rsync on a ordered list of triples. Such savings are even more evident when small amounts of differences exists, in other words the more frequent are the updates, the more efficient they are. This ultimately enables scenarios where large datasets (e.g. Even those of the size of DBPedia) can be kept in sync even multiple times a day with local (personal or intranet) copies without gigabytes of traffic locally and or terabytes of traffic remotely.

Finally, the procedure has been adapted also to operate over the HTTP protocol thus enabling end user clients to use it directly.

References

1. Tridgell, A.: Efficient Algorithms for Sorting and Synchronization, PhD Thesis, Australian National University (1998)
2. Berners-Lee, T., Connolly, D.: Delta: an ontology for the distribution of differences between RDF graphs. In: MIT Computer Science and Artificial Intelligence Laboratory (2004)
3. RDF Semantics, W3C Recommendation (2004), `http://www.w3.org/TR/rdf-mt/`

[12] http://wymiwyg.org/rdf-utils
[13] http://gvs.hpl.hp.com/

4. Sayers, C., Karp, A.H.: RDF Graph Digest Techniques and Potential Applications, HP Technical Report (2004)
5. Carroll, J.: Signing RDF Graphs, TechnicalReport HPL-2003-142, HP Lab (2003)
6. RDF Vocabulary Description Language 1.0: RDF Schema (2004), http://www.w3.org/TR/rdfschema/
7. OWL Web Ontology Language Overview, http://www.w3.org/2001/sw/WebOnt/
8. Ding, L., Finin, T., Peng, Y., da Silva, P.P., McGuinness, D.L.: Tracking RDF Graph Provenance using RDF Molecules. In: Proceedings of the Fourth International Semantic Web Conference (2005)
9. Tummarello, G., Morbidoni, C., Nucci, M.: Enabling Semantic Web communities with DBin: an overview (2006)
10. RDF Site Summary (RSS) 1.0 (2000), http://web.resource.org/rss/1.0/
11. URIQA The URI Query Agent Model (2003), http://sw.nokia.com/uriqa/URIQA.html
12. Tummarello, G., Morbidoni, C., Petersson, J., Piazza, F., Puliti, P.: RDFGrowth, a P2P annotation exchange algorithm for scalable Semantic Web applications (2004)
13. Lindholm, T., Fault-tolerant, A.: Three-way Merge for XML and HTML, Internet and Multimedia Systems and Applications (EuroIMSA), Grindelwald, Switzerland (2005)
14. La Fontaine, R.: Merging XML files: a new approach providing intelligent merge of XML data sets, XMLEurope (2002)
15. Tancred Lindholm, XML Three-way Merge as a Reconciliation Engine for Mobile Data. In: Third International ACM Workshop on Data Engineering for Wireless and Mobile Access, San Diego, California (2003)