The case for generating URIs by hashing RDF content

Craig Sayers, Kave Eshghi
Intelligent Enterprise Technology Laboratory
HP Laboratories Palo Alto

HPL-2002-216
August 22nd , 2002*

RDF, URI, hash, semantic web, hashed URI, hashed references

In this paper we argue for using hashed URIs to represent RDF content. These URIs are generated by serializing the RDF facts which describe an Object, computing the hash of that serialization, and then using the computed hash as the Object's URI. In this way, the hashed URI serves both as a short-hand notation for all the facts which describe it and as proof that the facts have not changed. This is particularly advantageous for distributed systems since it guarantees consistency while removing any need to trust the underlying storage. If you receive a URI, and later ask for the facts about that URI, you can prove that the returned facts exactly match those which existed at the time the URI was created.

The case for generating URIs by hashing RDF content

Craig Sayers and Kave Eshghi

Hewlett Packard Labs

Technical Report HPL-2002-216

August, 2002

Abstract

In this paper we argue for using hashed URIs to represent RDF content. These URIs are generated by serializing the RDF facts which describe an Object, computing the hash of that serialization, and then using the computed hash as the Object's URI. In this way, the hashed URI serves both as a short-hand notation for all the facts which describe it and as proof that the facts have not changed. This is particularly advantageous for distributed systems since it guarantees consistency while removing any need to trust the underlying storage. If you receive a URI, and later ask for the facts about that URI, you can prove that the returned facts exactly match those which existed at the time the URI was created.

1 Introduction

In the Resource Description Framework (RDF) [10, 13], data can be represented as a set of statements with the form (*subject*, *predicate*, *object*). In this paper, we'll write such statements using the N-triples syntax [6]:

> *subject*
> *predicate*
> *object* .

For this discussion we'll use the term *Object* (with a capital 'O') to refer to a collection of statements within a single model that have the same subject. That common subject is the Object's identifier.

Using an example from the RDF Primer [11], we could describe an address with the statements:

```
_:johnaddress
<http://www.example.org/terms/street>
"1501 Grant Avenue" .

_:johnaddress
<http://www.example.org/terms/city>
"Bedford" .

_:johnaddress
<http://www.example.org/terms/state>
"Massachusetts" .
```

In this case, the Object's identifier is the node identifier "_:johnaddress".

In conventional RDF stores (for example HP's Jena [7, 12]), the identifier for an Object is assigned when the Object is created, and remains constant throughout the lifetime of the Object. Sometimes the identifier is a particular URI/URL specified by the user. Other times, as in the case of a bNode, the identifier is computer-generated. In either case, it does not change as new statements are added. This is efficient for storage/manipulation, and is desirable for cases where you always want the most recent version of an Object. In our case, we wanted each different version of an Object to be separately identifiable, we wanted to guarantee that identifiers were unique, and we wanted the identifiers to act as proof that the facts describing an Object had not been altered.

To generate these identifiers, we compute a hash of the Object's properties and use that as the Object's Uniform Resource Identifier (URI). The hash code acts as a short-hand for all the statements which describe that Object. It is guaranteed that any visible change in the Object will result in a change to the hash code. Usually the hash will be generated by serializing the Object and then computing an MD5 [15] or SHA-1 [5] hash over the resulting string. If two Objects have the same hash, then they have identical facts. Specifically, when performing any read-only operation, the results of performing that on different Objects with the same hash should be identical.

By using Hash codes as URI's we gain many of the advantages of using hash codes in file systems [3, 14] (that's where we got the idea). Most significantly, we guarantee consistency. If a statement refers to a hashed-URI then the values of the facts describing that URI are guaranteed to have been constant for the lifetime of that URI. Furthermore, you can prove that they have not changed. There is no longer any need to use locking while reading from a shared database, nor any need to trust the underlying storage in a distributed database.

2 Hashed References

The use of a computed hash as a surrogate or identifier for an Object is not new. For example, in [8] a scheme is proposed for using hash digests as content derived names (CDN) for software components such as shared libraries (DLLs). Under this scheme, shared libraries would be identified using their CDN, and the programs that need to load them would use the CDN instead of the file name. This, it is argued, would overcome the version incompatibility problems that occur with modularized software.

Hash digests have been used extensively for file comparison, for example in [1], where it is used for avoiding the duplicate storage of identical files, and in backup systems.

There are number of proposed systems for a location independent, distributed file system, that use hash digests as pointers or references to the file content. These include OceanStore [9], a low-bandwidth network filesystem [14] and CFS [2].

3 Hashed URIs

In this section we examine the benefits and costs of hashed URIs. It should be noted that the term “hashed URI” has been used before [4], but the usage is different. That prior work deals with computing the hash of a URI while we’re interested in computing the hash of the facts describing an Object.

3.1 Consistency in meaning

RDF allows any URI as the subject of a statement. It is therefore tempting to use a convenient pre-existing URI when storing information. For example, if making statements about a company, it would be easy to use the URL of their corporate web page as the subject of the RDF statements. The problem with this approach is that somone else can use exactly the same URI for an entirely different purpose. For example, statements with the subject <http://www.hp.com> might be describing the HP company in one application and the web page in another.

By re-writing such statements to use hashed URIs we can avoid any inconsistency. Two objects with the same hashed URI have exactly the same facts. The “meaning” of such URIs depends only on the facts that describe them.

For example, instead of saying:

> <http://www.hp.com>
> <http://www.example.org/onto1/contact>
> “Webmaster” .

We could store similar information by using a hashed URI:

> <hashedrdf.v1:SHA1=0AB . . . A>
> <http://www.example.org/onto2/location>
> <http://www.hp.com> .
>
> <hashedrdf.v1:SHA1=0AB . . . A>
> <http://www.example.org/onto2/contact>
> “Webmaster” .

In this case, we have the cost of an extra statement, and the need for a different ontology, but we gain the benefit of guaranteed consistency.

3.2 Consistency in verification

In a conventional system, if you wish another agent to verify your results, you must share all the statements. In particular, if you query for statements with a particular subject, make a decision based on those, and then wish to justify that to your colleague, you can't simply refer to the subject because the statements about that subject may have changed by the time your colleague attempts to reproduce your result.

With hashed URIs, you can refer to the hashed URI in your arguments with the knowledge that your colleague will later be able to exactly reproduce your results. Every query they do on that hashed URI will return exactly the same results that you received.

3.3 Untrusted storage

If you ask for all the facts describing an Object with a hashed URI, you can always take those facts and recompute the hash to verify that you got what you asked for. This is particularly advantageous in distributed systems. It removes any need to trust the underlying storage.

3.4 Caching

Hashed URIs provide a convenient means to efficiently cache local copies of Objects. For example, consider the case where you query a remote system and store a local copy of all the facts about a hashed Object. If a later query returns a statement that refers to that hashed URI then there's no need to query the remote store for additional details. Any query you would have made for that hashed URI on the remote store may be satisfied by applying the same query to your locally cached copy.

This also has applications with peer-to-peer distributed RDF datastores. Given a hashed URI, you could search your peers for the facts that describe it.

3.5 Consistency in context

Since the hashed URI is a unique, unambiguous surrogate for an Object, it can be used in inter-agent communications as a means of providing context. For example, consider two agents, one acting as a car salesman, the other as a customer. They engage in a communication to decide on the features of a particular model. This process is incremental, sometimes the salesman may suggest a feature, sometimes the customer may demand one, sometimes the features are mutually-exclusive, sometimes one feature requires another. At the end of the process, the customer wishes to purchase the car and initiates a secure purchase transaction. While the two sides have tried to build a common understanding of the car in question, it is possible that due to some failure their understandings actually differ. By using the hashed URI of the RDF description of the car as a context in the purchase, the two sides can be assured that they are in fact talking about the same set of features, because they can both independently check the correctness of the URI against their RDF description. The alternative would be to send the whole RDF description back and forth, which may be more expensive.

3.6 Costs

The obvious drawbacks are the costs from computing the hashes and the cost of storing all the facts for every version of every hashed Object.

For distributed systems, we believe computational costs will be somewhat offset by the ability to cache local copies of Objects and that the benefits of using hashed URIs will increase over time as processor speeds will likely increase more quickly than the bandwidth of the networks connecting them.

By keeping copies of previous versions of Objects we do increase the storage requirements and hashed URIs are most appropriate for use with Objects that change infrequently. A practical example would be the metadata about the author/title/publisher of a novel. Those facts are known when the novel is published and are most unlikely to change. We expect databases will hold both static information (referenced by hashed URIs) and dynamic information (referenced by conventional URIs). In some cases it will be beneficial to break a single Object up into two pieces, so the constant portion may be hashed and efficiently cached/distributed.

4 Algorithm

There are clearly a number of ways one could generate a hash for a set of RDF statements. For our application, we require that:

- it is repeatable on different platforms
- it is independent of the current subject identifier
- it is independent of the ordering of facts in the RDF store.
- it generates different hash values if, and only if, two Objects have at least one difference in their describing statements. Here "difference" means that, when written as N-triples they have at least one character that is not the same. The hash is thus a very strict test for "sameness". The Objects don't just have the same "meaning" they must also be represented in the same way.

To generate a hashed URI Object we start with a list of statements that apply to the Object, and then:

1. replace any references to the identifier for this Object with the special URI: `hashedRDF.v1:self` (These could occur in self-referential statements).
2. write the statements out to an in-memory buffer. Unfortunately there is no current canonical serialization standard for RDF (there are ways to serialize it, but there is no canonical representation) . Until such a standard exists, we use a modified form of the N-triples syntax, dropping the beginning subject and trailing period while removing any options for whitespace and end-of-line characters. Specifically, the format is:

canonicalObj	::=	*pair**
pair	::=	*predicate space object newline*
predicate	::=	*uriref*
object	::=	*uriref* | *literal*
space	::=	space character (0x20)
newline	::=	new-line character ('`\n`')
uriref	::=	definition from N-triples spec
literal	::=	definition from N-triples spec

3. sort the resulting output line by line in ascending order (so the output ordering is independent of the order in which statements were added to the model).

4. compute the hash of that in-memory buffer.
5. create a URI using that hash by pre-pending the string:

   `hashedRDF.v1:`*hashingAlgorithm*`=`

   for example, if using MD5, the string would be: “hashedRDF.v1:MD5=”. This format is similar to previous recommendations for storing hashes in URIs [4].
6. replace any occurrences of the “self” URI with the newly-computed hash
7. construct and add a statement for each (*predicate*, *object*) pair using the computed URI as the subject.

This algorithm will not work if there are any bNodes present in the Object description. In adding facts to an Object, one must take care to only add facts that refer to absolute URIs.

It should be noted that this algorithm will not cope with cyclic references between Objects with hashed-URIs (for example: Object A has a statement that refers to Object B and Object B has a statement that refers to Object A). If such references are needed, then you’d need to create an absolute URI to serve as a surrogate for the URI of at least one Object, compute the hashed-URIs, and then add additional facts to state that the surrogate URI was equivalent to the computed hashed-URI.

5 Example

First, consider an Object with an initial internal identifier of "_:self". Just to make things interesting, we'll include a self-referential statement in which the object of the statement is also the subject (this isn't very meaningful in this case, but it's just to make the point that such statements are possible).

```
_:self
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
"testType" .

_:self
<http://www.w3.org/1999/02/22-rdf-syntax-ns#value>
_:self .
```

When this Object is committed the store will contain two statements:

```
<hashedrdf.v1:SHA1=09ed2af86a1fd78e5195c734e96d614b7c192158>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
"testType" .

<hashedrdf.v1:SHA1=09ed2af86a1fd78e5195c734e96d614b7c192158>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#value>
<hashedrdf.v1:SHA1=09ed2af86a1fd78e5195c734e96d614b7c192158> .
```

We can now take the original Object and add an additional statement:

```
_:self
<http://www.w3.org/1999/02/22-rdf-syntax-ns#value>
"5" .
```

When that updated Object is committed, the store will contain five statements – two from the initial version of the Object, and three for the new version:

```
<hashedrdf.v1:SHA1=2feaa829b1caae389c419d6ce15370acc9cd76b3>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
"testType" .

<hashedrdf.v1:SHA1=2feaa829b1caae389c419d6ce15370acc9cd76b3>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#value>
<hashedrdf.v1:SHA1=2feaa829b1caae389c419d6ce15370acc9cd76b3> .

<hashedrdf.v1:SHA1=2feaa829b1caae389c419d6ce15370acc9cd76b3>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#value>
"5" .
```

If we wished to store an additional statement for our own private use, and did not require the security of the hashed URI, then we could store that additional statement in a separate private Model (so we can distinguish our local statements from those which may be verified using the hashed URI).

6 Java interface to the hash-store system

Our implementation of the hashed Object system is based on the Jena [7] RDF store. Using Jena, one constructs statements by creating *resources* and adding *properties* to them. For example, here we create a new anonymous resource and add a property to it:

```
Model m = new ModelMem();
Resource r = m.createResource();

// Add some statements to the model by adding properties to r
r.addProperty( RDF.type, "temperature");
r.addProperty( RDF.value, 451);
System.out.println("Object has anonymous ID "+r.getId() );
```

In our system, we introduce a special type of resource called a *HashedObject*. The conventional Jena resource adds statements to the model as properties are added to the resource. The HashedObject is different. You add properties to it independently of the model. Then when you writeToModel() the system computes a hash for the Object based on all its properties and then adds statements to the model using that computed hash as the subject. Here's an example:

```
Model m = new ModelMem();
// Create a hashed Object independent of the model
HashedObject ho = new HashedObject();

// Add some properties to the Object
ho.addProperty( RDF.type, "temperature");
ho.addProperty( RDF.value, 451);

// Now add statements to the model by writing the Object
Resource one = writeToModel(m, ho);
System.out.println("Object has hashed URI "+one.getURI());
```

7 Discussion

In the current system, we don't store any knowledge concerning the evolution of Objects over time. In particular, given a URI, there's no standard way to find if a more recent version of that Object exists. One option would be add an additional statement to each updated Object to indicate the original URI. For example, again using the example from Section 5, when we added the additional fact:

```
_:self
<http://www.w3.org/1999/02/22-rdf-syntax-ns#value>
"5" .
```

we'd also add a fact:

```
_:self
<derivedFrom>
<hashedrdf.v1:SHA1=09ed2af86a1fd78e5195c734e96d614b7c192158> .
```

then it would be possible to look for updated versions of an *object*, by querying for any x, where there was a statement (x, `derivedFrom`, *object*).

When new facts are added to an Object, another alternative would be to store just the new facts, and a reference to the original Object. In that case, you'd need to follow the trail of previous Objects in order to get all the facts that currently apply.

The use of hashed URIs allows the unambiguaous mixing of different statements about the same "thing" within a single model. In this way, they provide some of the benefits of storing all statments in a reified form, but with lower cost.

The algorithm presented in this paper may also useful when the Object is the definition of a Class or Property, but that application requires further study.

8 Conclusions

Hashed URIs are computed by serializing an Object, computing the hash of that serialization, and then using the computed hash as the Object's URI. In this way the hash code serves as a surrogate for the entire Object. It is guaranteed that any change to the Object would result in a different hash code.

The use of hashed URIs provides consistency both when an agent queries a database and when one agent attempts to verify the results of another. In addition, it makes it possible to create RDF stores that are physically distributed and that may rely on untrusted machines for storage.

There are additional storage and computational costs due to the use of hashed URIs, but, for Objects that change infrequently, those costs are offset by the guaranteed consistency and the ability to reliably cache local copies of Objects.

9 Acknowledgements

Thanks to Claudio Bartolini, Harumi Kuno, Brian McBride, and Andy Seaborne for reviewing an earlier draft of this report. Any remaining errors in this version are solely the responsibility of the authors.

References

[1] W. J. Bolosky, S. Corbin, D. Goebel, and J. R. Douceur. Single instance storage in windows 2000. In *Proceedings of the 4th USENIX Windows Systems Symposium*, 2000.

[2] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with cfs. In *ACM SOSP 2001, Banff*, October 2001.

[3] K. Eshghi. Intrinsic references in distributed systems. In *IEEE Workshop on Resource Sharing in Massively Distributed Systems, Vienna, Austria*, July 2002.

[4] C. Feather. Internet draft: The hashed URI. draft-feather-hashed-uri-02.txt at http://www.ietf.org, January 2002.

[5] Federal Information Processing Standards Publication 180-1, U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfield, Virginia. *FIPS 180-1: Secure hash standard*, April 1995.

[6] J. Grant and D. Beckett. Resource description framework (RDF) test cases, W3C working draft. http://w3.org/TR/2002/WD-rdf-testcases-20020429, April 2002.

[7] Hewlett Packard Laboratories, http://www.hpl.hp.com/semweb. *Jena: A Java API for RDF*, 2002.

[8] J. K. Hollingsworth and E. L. Miller. Using content-derived names for configuration management. In *ACM Symposium on Software Reusibility, Boston, MA*, May 1997.

[9] J. Kubiatowicz et al. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, November 2000.

[10] O. Lassila and R. Swick. Resource description framework (RDF) model and syntax specification, W3C recommendation. http://w3.org/TR/1999/RED-rdf-syntax-19990222, February 1999.

[11] F. Manola and E. Miller. Resource description framework (RDF) primer, W3C working draft. http://w3.org/TR/2002/WD-rdf-primer-20020319, March 2002.

[12] B. McBride. Jena: Implementing the RDF model and syntax specification. In S. Decker et al., editors, *Second International Workshop on the Semantic Web*, Hong Kong, May 2001.

[13] E. Miller et al. Resource description framework (RDF)/W3C semantic web activity. http://w3.org/RDF, April 2002.

[14] Q. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In *18th ACM Symposium on Operating System Principles*, Banff, Canada, October 2001.

[15] R. Rivest. Request for comments (RFC) 1321: The MD5 message digest algorithm. http://www.ietf.org/rfc/rfc1321.txt, April 1992.