Entity Identifiers for Lineage Preservation

Julien Gaugaz and Gianluca Demartini

L3S Research Center
Leibniz Universität Hannover
Appelstrasse 9a D-30167 Hannover, Germany
{gaugaz,demartini}@L3S.de

Abstract. The generation of entity identifiers is a key issue in the context of semantic web technologies. Entity identifiers are needed when we perform any kind of operation (e.g., reference, match, disambiguation) on entities.
In this paper, we present a methodology for generating entity identifiers which can be well integrated in the *Entity Name System* of the OKKAM infrastructure: a system for managing entity identifiers. Our approach is based on a known labelling scheme for directed acyclic graphs. Unlike other related approaches, it allows to keep track of the lineage of an entity over the operations of creation, merge of entities, and split of a single entity.
This feature is important in a identifier generation framework because it can allow both local deprecation detection between two entities, as well as the reconstruction of the origin of an entity partially described in different sources (i.e., knowing which entity attributes are coming from which sources).

1 Introduction

The main goal of the OKKAM project is to enable the Web of Entities. This will be accomplished by supporting the use of globally unique identifiers for entities with the aim that the same object will always be referred by the same identifier. It is important to clarify that the goal of OKKAM is not to create a comprehensive knowledge base of information about entities (i.e., attributes of an entity and their values, like for example `name:John surname:Doe`), but, instead, to create a collection of entity identifiers. The subpart of the entire OKKAM infrastructure that has the role of managing entity identifiers is the *Entity Name System* (ENS), presented in [4], that is:

> a service which stores and makes available for reuse URIs for any type of entity in a fully decentralized and open knowledge publication space.

The main functionalities of the ENS are: search for the identifier of an entity, generation of entity identifiers, matching entities present in the repository with external ones, and ranking entities by similarity to a given one.

Within this architecture, the current way for generating IDs is the process described in [5], which is based on hash functions. The main contribution of this paper is a novel way of generating IDs for entities, preserving their lineage. The algorithm we present can be well integrated in the ENS architecture which is being developed in the context of the OKKAM project[1]. In more detail, this paper focuses on the *OKKAMization* process. That is, "the process of assigning an OKKAM identifier to an entity that is being annotated in any kind of content, such as an OWL/RDF ontology, an XML file, or a database, to make the entity globally identifiable." (from [4]).

1.1 Settings

We now describe by way of an example the possible operations that can be applied to an entity. A knowledge-based system, at some point in time, might have a wrong representation of an entity: for example, initially, a system has the knowledge that a person is called John Doe, but, after having acquired more evidence, the system knows that another person, named John J. Doe, is, actually, the same entity. In this case the two entity representations as well as their EIDs need to be *merged*. Also, the entity itself can change over time (e.g. semantic evolution, evolution of documents [2]): for example, the attribute height of a person can change over time. This needs to be considered by description-dependent entity identification techniques. For these reasons, we need to define the possible changes that the entity identifiers might undergo. The following operations can be applied on an entity:

Creation When an entity is first encountered, a new EID is generated.

Split When the system discovers that the same entity representation describes two different real world entities, two new EIDs have to be created. For example, the system knows that a person is called "Andrea Rossi" but, at some point, it discovers that there are actually a man and a woman with the same name, in the dataset.

Merge When two entities are matched, and recognized to be the same, they are merged, and the same has to be done to their EIDs. For example, John Doe and John J. Doe are two different entities which are then mapped to the same person.

1.2 Motivation

The reason why we include lineage information directly in the EIDs is mainly for efficiency reason. Using EIDs for lineage preservation we can easily reconstruct ancestors and descendants[2] of EIDs locally, without having to query a remote ENS service provided by OKKAM. For example, when a mail is received by someone, an exchange of data and metadata happens. The mail can contain

[1] `http://www.okkam.org/`

[2] According to split/merge operations, see Fig. 1.

named entities that OKKAM tools will recognize and associate an EID to each of them. Also, when some already existing EIDs arrive from the sender they need to be matched with the local (i.e., receiver's) EID repository. At this point, some EIDs will be recognized and relevant local metadata can be shown to the user. Some other EIDs will not be recognized because they are not yet present in the local machine. At this point, using classical EIDs, the local system should query an OKKAM node in order to resolve the entities which are unknown. Using lineage preserving EIDs an intermediate step can be performed before querying the remote system. It is possible, for example, to check whether the incoming EIDs are obsolete versions of entities known by the local system. It is also possible to check if the local EIDs are obsolete versions of the incoming EIDs thus updating the local knowledge. These operation are possible just by using the EIDs without querying a remote system. The advantage is that, when the OKKAM infrastructure will be widely used, a lot of system load on the ENS can be avoided by redistributing some of it on the clients.

Another desired aspect is the possibility to decide whether two EIDs which are related by an ancestor/descendant relationship are actually the results of a split or merge operation. For example, in Fig. 1 we can see the lineage of an entity representation A which has been first created, then merged with another entity B after a mapping operation, and, finally, split into two different entities D and E after obtaining more knowledge (i.e., semantic annotations) on it. If the local system already knows A and it receives the EID C, then knowing that this is the result of a merge operation, it can safely replace all the occurrences of A with C. In the other case, if the local system already knows C and it receives the EID D, it is not able to decide autonomously if it should be replaced by D or by F, but only that C is deprecated and that the system should query a remote OKKAM node for updating it.

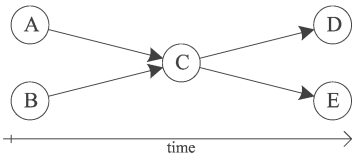

Fig. 1. An example of an entity lineage graph. Time goes from left to right.

From the described scenario we can now extract a list operations that the EIDs generation algorithms should support. We have seen that it is important to be able to:

- create EIDs for unknown entities;
- resolve whether a given EID A is an ancestor of an EID B;
- deciding whether two related EIDs are the result of a split or of a merge operation;
- retrieve the list of all the ancestors of an EID;
- retrieve the list of all the descendants of an EID.

As we will present in Section 3, using EIDs for lineage preservation, some of these operations can be handled locally, without the need to query a remote system, and others still require a remote query can be optimized selecting only some candidates thus reducing the number of requests to be sent at the remote node.

Our solution to entity identification aims at assigning a label to each node in the lineage graph (see Fig. 1), that is, assigning an identifier to an entity, so that its lineage of creation, splits, and merges, is stored in its EID. We can easily see that the generated graph is directed and acyclic. Therefore, we can adapt a known algorithm for labelling Directed Acyclic Graphs (DAGs) to our purpose, as presented in the next section. Our approach, based on DAGs labelling, is formally defined in Section 2. In Section 3, we critically discuss the advantages and the disadvantages of our approach, and, in Section 4, we conclude the paper.

2 Our Approach to Entity Identification

In this section we describe the entire framework of entity identification we propose in detail. First, we describe an algorithm for labelling DAGs, the adaptation of this algorithm to the context of entity identification, and, then, how DNS can be used for linking EIDs with user-friendly names.

2.1 PLSD: Prime Numbers Labelling Scheme for Directed Acyclic Graphs

Wu et al.[8] proposed a labelling scheme for transitive closure computation on DAGs. Computing a transitive closure in a graph is used for identifying all ancestors (or descendants) of a given node in the graph. We describe hereafter the Lite version of Prime Numbers Labelling Scheme for DAGs (PLSD) proposed by Wu et al.

Prime number factorisation. PLSD is based on the well-known fundamental theorem of arithmetic (or unique prime factorisation theorem), from number theory, which states that any natural number greater than 1 can be written as unique product of prime numbers. The idea of PLSD is then to assign a unique prime number EID to each vertex in the DAG, and label each vertex with the product of its ancestors' prime number EIDs. Thus, given a vertex label, by performing a prime number decomposition, we can retrieve the EIDs of all its ancestors. More formally this gives:

Definition 1. *Let $G = (V, E)$ be a DAG, with V a set of vertex and E a set of edges. We define the bijective function $p : V \to \mathbb{N}$ such that $p(v)$ is prime, for $v \in V$.*

Definition 2. *Let $G = (V, E)$ be a DAG. We define the label of a vertex $v \in V$ as $L(v) = (c[v])$ where*

$$c[v] = p(v) \cdot \begin{cases} \prod_{v' \in parents(v)} c[v'], & in - degree(v) > 0 \\ 1, & in - degree(v) = 0 \end{cases} \tag{1}$$

with $parents(v)$ being the set of vertex parents of v'.

Now, given a vertex v and its label $c[v]$, the fundamental theorem of arithmetic assures that there exists a unique prime factorisation of $c[v]$ such that

$$c[v] = p(v) \cdot \prod_{v' \in ancestors(v)} p(v')^{m_{v'}} \tag{2}$$

where $ancestors(v)$ is the set of all the ancestors of v, for some $m_{v'} \in \mathbb{N}$; which gives us v's ancestors' EIDs. An example is shown in Fig. 2.

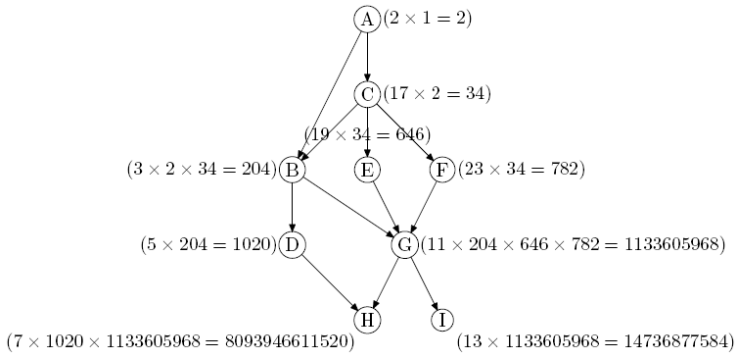

Fig. 2. PLSD-Lite on a DAG (from [8])

Optimisation. As we can see in Fig. 2, the ancestor labels $c[v]$ are growing exponentially due to the $m_{v'}$'s from Equation 2 which are in general greater than 1. This requires a storage space such that it is not much more space-efficient than simply storing the ancestors' EIDs as a list for each vertex. To avoid this, Wu et al.[8] proposed to force the $m_{v'}$'s to be 1. This can be done by modifying the computation of $c[v]$ described in Definition 2 as follows:

$$c[v] = p(v) \cdot \begin{cases} lcm\left(c[v'_1], \ldots, c[v'_n]\right) & \text{if } in - degree(v) > 0 \\ & \text{and } v'_1, \ldots, v'_2 \in parents(v) \\ 1 & \text{if } in - degree(v) = 0 \end{cases} \tag{3}$$

where $lcm(a_1, a_2, ..., a_n)$ is the *least common multiple* of the natural numbers $a_1, a_2, ..., a_n$, with the special definition of $lcm(a) = a$. Fig. 3 shows how the

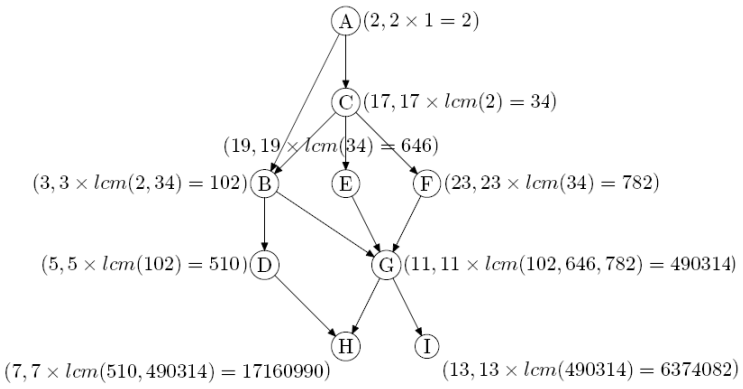

Fig. 3. PLSD-Lite on a DAG using least common multiple (modified from [8])

DAG depicted in Fig. 2 is labelled using ancestor labels as defined in Equation 3.

Wu et al. also present a topological sort before labelling a DAG, and define a PLSD-Full labelling scheme for the computation of a vertex's parents, but we omit their description here since they are not useful or applicable in our situation. The version we presented about is referred to as PLSD-Lite. See their paper[8] for more details.

2.2 Lineage Preserving Entity EIDs

Considering the entity EIDs evolution as a DAG, we can use the PLSD labelling scheme described in Section 2.1 to locally detect if an EID has been deprecated. As in Fig. 4(a), if an EID A is deprecated by another EID B, then there exists a directed edge $A \rightarrow B$. As a consequence the *merge* and *split* operations mentioned in Section 1.1 are modelled as illustrated in Fig. 4(b) and 4(c) respectively.

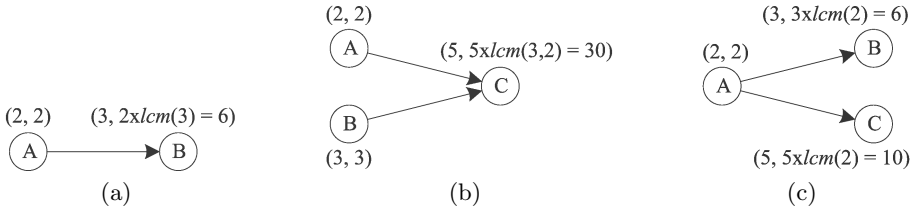

Fig. 4. (a) A is *deprecated* by B. (b) A and B are *merged* into C. (c) A is *split* into B and C.

We annotate each node of the DAG such built with a pair of natural numbers: the self-label, and the ancestors-label. The self-label is a unique prime number (i.e., from all the self-labels, a self-label appears once and only once), and the ancestors-label of a vertex is the product of its self-label with the least common multiple of the ancestors-labels of its ancestors (i.e., its in-component). Formally this gives:

Definition 3. *Let $G = (I, D)$ be a DAG with I a set of vertices representing EIDs, and D a set of edges representing deprecation. We identify a vertex $i \in I$ by a pair $(i_{self}, i_{ancestors})$ with $i_{self}, i_{ancestors} \in \mathbb{N}^+$, where*

i) i_{self} is a prime number such that there exists a bijection between I and the set $I_{self} \equiv \{j_{self} | j \in I\}$
ii) $i_{ancestors} = i_{self} \cdot lcm(a^1_{self}, \ldots, a^n_{self})$ with $a^1, \ldots, a^n$ the ancestors (in-component) of i

We call $i = (i_{self}, i_{ancestors})$ the Lineage Preserving ID (LPID), i_{self} the self-label of i, and $i_{ancestors}$ its ancestors-label.

Lemma 1 shows that, given two LPIDs as defined in Definition 3, the fact that the self-label of one LPID divides the ancestors-label of the other LPID implies that the former is deprecated.

Lemma 1. *Let $i = (i_{self}, i_{ancestors})$ and $j = (j_{self}, j_{ancestors})$ be two LPIDs. $j_{ancestors}/i_{self} \in \mathbb{N}$ implies that i is deprecated.*

We can now formalize in Algorithms 1, 2 and 3 the three operations defined in Section 1.1: *create*, *merge* and *split* respectively.

Algorithm 1 *create*()

Require: I the set of attributed LPIDs of the form $i = (i_{self}, i_{ancestors})$
1: create the LPID $j = (l, l)$ with l the lowest prime such that $l \notin \{r | r = i_{self}, i \in I\}$
2: add j to I
3: **return** j

Algorithm 2 *merge*(M)

Require: $M \subset I$, with I the set of attributed LPIDs of the form $i = (i_{self}, i_{ancestors})$
1: create the LPID $j = (l, l \cdot lcm(m_1, \ldots, m_n))$ to the lowest prime l such that $l \notin \{r | r = i_{self}, i \in I\}$ with $\{m_1, \ldots, m_n\} \equiv M$
2: add j to I
3: **return** j

Algorithm 3 $split(j, n)$

Require: $j \in I$, $n \geq 1 \in \mathbb{N}$, with I the set of attributed LPIDs of the form $i = (i_{self}, i_{ancestors})$

1: create empty set S
2: **for** $k = 1$ to n **do**
3: create the LPID $s_k = (l, l \cdot i_{ancestors})$ to the lowest prime l such that $l \notin \{r | r = i_{self}, i \in I\}$
4: add s_k to I
5: add s_k to S
6: **end for**
7: **return** S

For using the proposed PLSD-Lite version presented in Section 2.1, we must make sure that the considered directed graph is actually acyclic. The acyclic nature of the graph naturally follows from Algorithm 2 line 3, and Algorithm 3 line 4 where incoming edges are added only to newly created vertices which did not belong to the set of vertices I before the respective operation is executed. This implies that, using the *create*, *merge* and *split* operations, a $deprecated_{by}$ edge $d = (i, j)$ can only be created if $i \in I$ and $j \notin I$, i.e. $\forall i \in I$ no new incoming edge can ever be created. Thus, the graph $G = (I, D)$ is acyclic.

2.3 Using DNS for resolving EIDs

A Domain Name System (DNS) is used on the Internet to resolved user-friendly domain names (like www.L3S.de) into IP addresses usable by hosts computers and routers for transmitting data. A DNS is a hierarchical distributed map. We propose to use it for attributing user-friendly names (domain names) to EIDs.

DNS is an application layer core service widely used in today's internet. It is described in details in RFCs 1034 and 1035 [6, 7]. It offers mapping of human-readable names to various information, mainly IP addresses and email-related information. The DNS space is a tree of domains and sub-domains, and information (IP address or other) is stored at the leaves. A domain name is composed of a sequence of *labels* separated by dots, each identifying a domain. The left-most label represents the top-level domain (like .com, .org or .de). The authoritative DNS name servers of a domain knows where to locate the DNS name servers of the sub-domains. Thus the resolution of a domain name into an IP address, for example, goes down the tree from domain to sub-domain DNS name server, until it reaches the desired leaf, where the IP address is stored and returned to the querier. Actually, a DNS name server is authoritative for a so-called *zone*, which comprises at least one domain, but possibly several domains and sub-domains (and sub-sub-domains etc. . .). In practice, DNS makes use of time-bound caching (time to live or TTL) for optimization, but we will not enter those details here.

At each node of the DNS tree so-called Resource Records (RRs) can be stored. Several types of RR are described in the RFC 1035 [7], the best known

being the ARPA Internet specific RR A which stores IP addresses. We define below the LPID type and RDATA format

Definition 4. *Let $i = (i_{self}, i_{ancestors})$ be an LPID to be encoded in a DNS node. We define a new RR Type LPID. A LPID RDATA is a bit sequence composed of three parts. The first part is a 8 bits unsigned integer indicating the length of the self-label, the second part is the self-label, and the last part is the ancestors-label. This is illustrated in Fig. 5.*

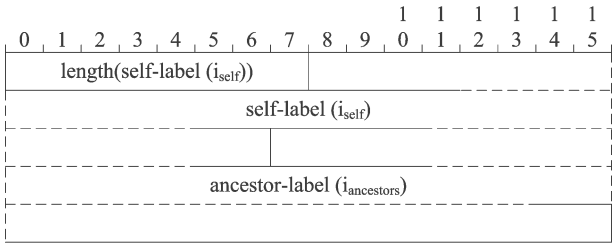

Fig. 5. LPID RDATA Format

The maximum number of entities which can be identified depends on the the maximum length of the RDATA field, and of the first part of the LPID RDATA indicating the bit length of the self label. Let us take as a reference the number of addresses that IPv6 can handle, namely 2^{128}. To give an idea, this corresponds approximately to 5000 URIs per square micrometer on earth, on the whole planet. Using the Prime Number Theorem, we can approximate the $2^{128}{}^{th}$ prime number to $2^{128} \ln 2^{128} \approx 3 \cdot 10^{40}$ which requires $\lceil log_2(10^{40}) \rceil = 135$ bits to write. We thus need the bit length of the self-label: 8 bits (and it is sufficient to can write the self-label bit length of 135), then the maximum self-label takes 135 additional bits, which makes a total of 143 bits. According to RFC1035[7], Section 3.2.1, the maximum length of the RDATA field is $2^{16} = 65536$ bits. This leaves us with $65536 - 143$ bits to write the ancestors-label. This ancestors-label is the product of the self-label with the least common multiplier of the ancestors' self-labels. Since the biggest self-label we consider requires 135 bits to write the maximum number of unique ancestors we can encode in the ancestors-label field is at least $\lfloor (\overbrace{2^{16}}^{RDLENGTH} - \overbrace{135}^{self-label})/135 \rfloor = 483$. Experiments will be performed in the future to further investigate the space requirements of the LPIDs.

3 Discussion

3.1 LPIDs vs. Ancestor Enumeration

We proposed the LPIDs for being able to determine locally if an EIDs a is an ancestor of another EID b as described in Section 1.2, i.e. to perform the oper-

ation $isAncestor(a, b)$. For this we proposed to use prime numbers as self-label, and the product of the self-label with least common multiplier of the self-labels of the ancestors as an ancestors-label, which we described in Section 2.2. This solution has to be compared to using arbitrary IDs as self-label, and storing the the enumeration of ancestor IDs as ancestors-label. The LPID approach has the advantage of time efficiency. When creating an EID, to compute the ancestor-label the LPID solution will require the computation of the least common multiplier only of the parent's ancestors-labels; whereas the enumeration solution would require to collect the list of all ancestors of the new ID (not only of its parents) with a graph-search algorithm, and then remove duplicate EIDs, requiring in turn to first sort the collected EIDs. Also, to perform the operation $isAncestor(a, b)$ the LDIP approach requires a single division, whereas the list approach requires to compare a with all ancestors of b.

Regarding the space efficiency, things are less clear. On one hand, storing the product of n integers does require less space than storing their enumeration. Let us take the example of two integers 3 and 5. Storing their product requires $log_2(3 \cdot 5) = 3.9$ bits. However we cannot store 0.9 bits, so we need 4 bits to store the product. Similarly to store 3 we need $\lceil log_2(3) \rceil = 2bits$ and $\lceil log_2(5) \rceil = 3$ bits. The required space to store the enumeration is already $2 + 3 = 5$ bits, i.e., 1 bit more than to store the product. Additionally, given the sequence of bits representing the enumeration, in our example $\overbrace{11}^{3}\overbrace{101}^{5}$, we also need to know where the first integers ends (or where the second begins). For this we can add the length of bits of the first integers, e.g., on 4 bits. This would then become $\overbrace{0010}^{2}\overbrace{11}^{3}\overbrace{101}^{5}$, which gives a total of 9 bits for the enumeration versus only 4 bits for the product solution.

On the other hand, if we take the enumeration approach, we do not need to limit ourselves to prime numbers for the self-labels. Thus the integers to be stored in this case are smaller than in the case of the LPID approach. We will perform an in-depth comparison of space and time efficiency of the two approaches experimentally as future work.

3.2 LPIDs as URIs

Our Lineage Preserving IDs defined in Section 2.2 are not URIs as defined by RFC3986[3]. For using our LPIDs as URIs in RDF and the Semantic Web we use the same strategy as the current OKKAM IDs, as we define below:

Definition 5. *Let e be an entity, and $lpid(e)$ its lineage preserving ID as defined in Section 2.2. We define the Lineage Preserving URI of e as $lpuri(e) = concat(\texttt{"http://www.okkam.org/entity/"}, lpid(e))$. Where $concat(a, b)$ is the string concatenation of a and b.*

3.3 Why Lineage and not Content in EIDs?

URIs[3] in the Semantic Web are susceptible to be transmitted (through the internet in a way or another) from one system to another system. They can and should be reused as much as possible, and thus the entity they identify may have different properties depending on their context, i.e. depending on the system they are in, depending on time, etc... This is the reason why traditionally URIs do not bear any meaning, other than possibly allowing to locate the entity (or resource) they identify–in which case they are also URLs. In our scenario, we want to be able to transmit the lineage as introduced in Section 2 along with the EID. A naive approach would be to allow to retrieve this lineage information from the ENS each time this is necessary. Note however that the lineage (ancestor) information for an EID will never change as time passes by, and depends only on one system (the ENS) and not on the local system in which the EID is used. For this reason we can include the lineage information in the EIDs. This has two more advantages:

1. Systems unaware of the lineage information will not use it, but they will further transmit it to other systems, unconsciously.
2. Since the lineage is indissociable from the EID, we don't need to query the ENS each time a system needs the lineage information. This spares a considerable amount of communications and ENS resources, as we detail hereafter.

3.4 Advantage of Local Deprecation Detection

Given that we can include the lineage into the ID of an entity, we show hereafter how we can use it to significantly reduce the number of requests to the ENS in order to insure that an entity is represented by one and only one EID in our local entity repository. Let us consider an entity repository using EIDs issued by the ENS. The EIDs can come from different sources: from the local system itself when extracting entities and querying the ENS to obtain an EID, or also from other systems when exchanging metadata on entities–and thus exchanging their EIDs as well. Given an EID in our local repository, it is quite possible that since the time it has been retrieved from the ENS, the latter deprecated it and replaced it with one or more other EIDs, depending on whether there was a split or a merge–or a sequence of them. The fact that an EID is deprecated is not annoying in itself, as long as this EID represents only one entity considered in our local repository, and this entity is represented only by this EID in the local repository. This latter condition is equivalent to ensuring that no EID in our repository is an ancestor EID in the same repository. We call this *local deprecation*. This is exactly what the LPIDs allow, using the $isAncestor(a, b)$ operation. Thus, using LPIDs, we will be able to locally detect–i.e. without querying the ENS–all suspect EIDs and query the ENS for up-to-date EIDs only for entities represented by those. In comparison, to achieve the same sound consistency with traditional (non lineage preserving) EIDs, we have to query for

the up-to-date EID for each and all entities in our local repository. This saves a huge amount of communications with the ENS, and also spares a lot of workload to the ENS.

3.5 Universally Unique IDentifiers

Universally Unique IDentifiers (UUIDs) are defined in RFC 4122 [5]. They are meant to be generable without any centralized authority to administer them. There are three different versions for generating UUIDs:

Time-Based The time and CPU clock sequence number together with one IEEE 802 MAC address of the computer generating the UUID are used to generate universally unique identifiers.

Name-Based A name supposed to be unique in a given name-space is hashed (either with MD5 of with SHA-1) to create the UUID. The uniqueness is then relayed to the uniqueness of the name in the name-space.

Random Uses purely or pseudo random numbers to generate UUIDs with low collision probability.

Our URIs are not UUIDs in the sense that we need a centralized authority ensuring the uniqueness of the self-labels. On the other hand URIs based on PLSD allow for local deprecation detection, which is a decisive advantage in a fast evolving environment.

4 Conclusion

In this paper, we presented a novel approach for identifying entities. Our approach, based on a known labelling scheme for DAGs, can be well integrated in the ENS infrastructure replacing already existing techniques for EID generation.

We now discuss the strong and weak points of the presented approach. As already said, the main novelty of this EID generation technique is the possibility of preserving the information about the history of an entity using only its ID. The possible operation that we consider are entity creation, entities merge, and entity split (i.e., creation of two different entities out of one). Such LPIDs allow to locally detect the deprecated IDs which require an update, as described in Section 3.4, thus allowing huge communication reduction and resource savings on the ENS. One possible drawback of our approach is the size of the generated EIDs. As discussed in Section 2.3, depending on the number of managed entities the size of the EID may vary. The number of entities to be managed is difficult to predict. Basically, in our approach, we would have one single (possibly not connected) graph of EIDs. A possible solution for this is to construct EIDs based on the entity types having different graphs for different kind of entities (e.g., EIDs for people, EIDs for scientific publications, and so on). A deep understanding of the efficiency and scalability of the proposed approach remains as future work.

In the future we also want to investigate the possibility to build PLSD EIDs which are also UUIDs in the sense of EIDs preserving lineage and generable

without centralised administration. We want also to investigate how we can use locality-sensitive hashing algorithms[1] to provide for similarity-aware EIDs. It is our plan to also perform a complete study of the scalability of our approach given the limitations of the standards such, for example, the RFC 1035 with a RDATA field maximum length of 16bit that will limit the number of possible ancestors that we can represent in the EIDs.

Acknowledgements. We thank the anonymous reviewers for their valuable comments. Moreover, we thank Peter Fankhauser for his help in producing the final manuscript. This work is partially supported by the FP7 EU Large-scale Integrating Project OKKAM Enabling a Web of Entities (contract no. ICT-215032). For more details, visit `http://fp7.okkam.org`.

References

1. Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, 2008.
2. Klaus Berberich, Srikanta J. Bedathur, Thomas Neumann, and Gerhard Weikum. A time machine for text search. In *SIGIR*, pages 519–526, 2007.
3. T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986 (Standard), January 2005.
4. Paolo Bouquet, Heiko Stoermer, and Barbara Bazzanella. An entity name system ('ens') for the semantic web. In *European Semantic Web Conference 2008*, 2008.
5. P. Leach, M. Mealling, and R. Salz. A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122 (Proposed Standard), July 2005.
6. P.V. Mockapetris. Domain names - concepts and facilities. RFC 1034 (Standard), November 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 2065, 2181, 2308, 2535, 4033, 4034, 4035, 4343, 4035, 4592.
7. P.V. Mockapetris. Domain names - implementation and specification. RFC 1035 (Standard), November 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2845, 3425, 3658, 4033, 4034, 4035, 4343.
8. Gang Wu, Kuo Zhang, Can Liu, and Juanzi Li. Adapting prime number labeling scheme for directed acyclic graphs. pages 787–796. 2006.