Selforganisation in a storage for semantic information

Robert Tolksdorf, Anne Augustin
Netzbasierte Informationssysteme, Institut für Informatik, Freie Universität
Email: talk@ag-nbi.de, aaugusti@inf.fu-berlin.de

Abstract—Scalable distributed semantic storage infrastructures are hard to realize. We propose the usage of principles of selforganization for the storage and retrieval of RDF triples. We use a biology-inspired algorithm for clustering of triples based on a purely syntactical similarity measure.

Index Terms—semantic web, selforganization, triple store, Linda, coordination

I. INTRODUCTION

Applications of semantic technologies need a scalable storage infrastructure to make knowledge persistent. This infrastructure has to be organized in a distributed manner both for reasons of integrating existing systems as well as for avoiding system bottlenecks.

This distribution is hard to realize if some scalability is to be guaranteed. Traditional approaches for distribution such as replication or partitioning fail in large open systems. [1] has discussed the respective problems.

In this paper we report on an approach to use principles from self organization to store and seek information in a decentralized associative storage infrastructure for RDF triples. For identifying similar triples, we use a similarity measure on the URIs of their resources.

In the following, we report on our prior work on using self organization in distributed storage services. We then extend that concept to a semantic storage service and detail out our algorithms to store and retrieve triples. We outline our similarity measure and finally evaluate the approach by several experiments.

II. LINDA, SWARMLINDA, RDFSWARMS

Linda [2] is a coordination language which establishes a ubiquitous environment – the tuplespace – in which distributed applications manage the data they exchange. The languages consists of a minimal number of primitive operations. The primitive out-operation puts a tuple – list of data from a set of primitive types like <10,"hello",15.3> – into that space.

To retrieve the data, an application can use the in-operation which searches the space for data matching a template which is an argument to the operation – as in in{<10,?string,?float>}. The above example tuple matches since the template contains the same value in the first field and the others have the datatype requested with the tuple. Values in templates are called actuals while the placeholders are named formals. The in-operation retrieves a matching tuple, the rd-operation returns a copy of it.

Linda allows for great variations of the data exchanged without enforcing additional or changed operations. One can aim at other datatypes instead of data-oriented tuples and one can extend the underlying matching-relation. Most interesting are variations that move from identity of values to similarity of information. For the field of semantic middleware, several projects have taken that approach, see [3] for a survey and [4] and [5] for examples.

Linda provides a high-level abstraction of communication and coordination. It allows for parallel and distributed implementation and for variants of the initial concept. However, scalability of Linda to open systems at a Web scale has longely remained an unsolved problem. The existing approaches like the complete or partial replication of the tuplespace to multiple machines are not scalable and dynamic enough to support truely large open systems.

SwarmLinda [1] takes a novel approach to that problem. The idea is to completely decentralize the tuplespace and to establish self organization mechanisms to make it scalable and dynamic. Centralized mechanisms of tuple-placement (eg. global hash functions for tuple-placement) and search are replaced with autonomic entities for tuple distribution and retrieval that take decisions on where to store and where to search based on local observations only. As a result, there is no local bottleneck hindering scalability and the ability to adapt to changing topologies of the distributed system. The analogy used by SwarmLinda is that of ants and the algorithms used are those found in natural ant-colonies for finding food and sorting things [6]. For example, instead of performing an out-procedure with remote access to some server, an out-ant is generated which wanders through the systems until it is on a node that it considers suited for tuple-placement.

This prior work considered tuples as typed data that are related only be data or type equality. Any decisions taken by SwarmLinda ant was based on that relation only. However, that view does not cover the while information covered in a tuple.

This work is a first step to enrich SwarmLinda with a view on information in which the original Linda matching rule on data is replaced with a comparison method that looks on similarity instead of equality. The fundamental
step is to exchange the plain data tuples of variable length with RDF triples.

So \( \text{out}(s, p, o) \) stores a triple \((s, p, o)\) with subject \(s\), predicate \(p\) and object \(o\). Triples are retrieved from the system by using the primitives in and rd with templates that have one or two forms like for example \((?s, p, o)\) or \((s, ?, o)\). Here the first template has one formal and two actual fields and matches all triples that have \(p\) as predicate and \(o\) as object. For the implementation, we adapt the swarming algorithms of SwarmLinda and use a similarity measure based purely on the URIs identifying resources in triples. As a result RDFSwarms provides a scalable, self organized storage service for RDF triples.

Some families of related work to RDFSpaces exist. There are extension to the original Linda which offer more flexibility. Most noteworthy has been the LIME system ([7]). Here, agents are mobile and exchange data with their currently hosting node. Opposed to RDFSpaces, the focus is on mobility as a characteristic of agents using the coordination middleware. In RDFSwarms ants are mobile for the sake of implementing the coordination middleware.

There are several other works in the spirit of Linda that propose self-organized coordination middlewares like TOTA ([8]) or Spray computing ([9]). RDFSpaces is different from these in that it injects semantic information into the coordination model. The named approaches deal with pure data while we aim at handling information.

Finally, some extension to Linda are similar to our approach in that they use semantic information instead of pure data-tuples ([3]). These systems, however, do not consider self-organization as a means to implement a semantic Linda. RDFSpaces aims at scalability by a novel design, namely that of self-organization instead.

III. SWARMING ALGORITHMS

Common Linda implementations use some component which decides on which machine a tuple is to be stored (using some function that optimizes later retrieval) and then forwards the tuple. In SwarmLinda, the idea is to make tuples (and templates) active and autonomous. Like ants that seek food, virtual ants carrying a tuple move from machine to machine and decide locally whether the tuple should be dropped at a specific location. Eventually, clusters of similar tuples evolve. For seeking a match, an ant carrying a template wanders along the machines to find a cluster of matches. On its way back, it leaves a trail of scent which will guide future template ants to that cluster reducing their search times. The ants here perform a certain task autonomously by making simple, fast computable decisions based on local information. The different types of ants and their algorithms are detailed in the next subsections.

A. Triple distribution

For an \(\text{out}\)-operation, three ants are generated, each carrying a copy of the triple which is to be stored. The ants consider only one resource (subject, predicate or object) – which we call cluster resource – when deciding on similarity to triples. For a given triple \((s, p, o)\) one out of three generated ants has \(s\) as cluster resource and the other two \(p\) and \(o\) respectively. The \(\text{out}\)-ants roam the network following scents that resemble the cluster resource to find a cluster of similar triples. The ants age with each hop in the network. When an \(\text{out}\)-ant decides to drop the triple it emits the scent of the cluster resource on the node and in weaker quantity on the neighbor nodes such that future ants are guided to the cluster. Algorithm 1 shows a high-level description of the algorithm used by tuple ants.

Algorithm 1: High-level description of the out ant’s algorithm

| Variables: |
| age: realizes the ant’s aging mechanism |
| triple: the RDF-triple to be stored |
| cluster-resource: the out-ant’s cluster resource |

1: Initialization: \(\text{age} \) is set to a given integer value \(> 0\)  
2: while \(\text{age} > 0\) do  
3: Compute drop probability on current node based on \(\text{cluster-resource}\) (according to one of the eq. 9, 10 or 12). On the basis of the drop probability decide if \(\text{triple}\) should be dropped  
4: if the decision is made to drop \(\text{triple}\) then  
5: Drop \(\text{triple}\) on current node, drop the scent of \(\text{cluster-resource}\) on the current node and in weaker quantity on the neighbor nodes and die \(^1\)  
6: else  
7: Select next node from neighborhood based on \(\text{cluster-resource}\) (according to eq. 15)  
8: Move to selected node  
9: end if  
10: \(\text{age} \leftarrow \text{age} - 1\)  
11: end while  
12: Drop \(\text{triple}\) on current node, drop the scent of \(\text{cluster-resource}\) on the current node and in weaker quantity on the neighbor nodes and die

The formulas that are used for the path selection and drop probability are detailed in section III-E and III-D. The algorithm leads to clusters that contain triples which are similar in respect to at least one resource. For example a resulting cluster could consist of the triples \((s_1, p_1, r_1)\) and \((r_2, p_2, o_2)\) which resemble in resource \(r_1\) and \(r_2\). An ant may drop a triple \((r_3, p_3, o_3)\) where \(r_3\) is the cluster resource if \(r_3\) is similar enough to \(r_1\) and \(r_2\).

B. Triple retrieval

Ants implementing a \(rd\)-operation carry a template instead of a triple. We write \((s, ?p, o)\) for a template with two actuals and one formal. Depending on whether the template contains one or two actual fields, an equal number of ants is generated. If two \(rd\)-ants are active, the match from the one arriving first will be returned from

\(^1\)in all algorithms \textit{die} means exiting the entire algorithm
the rd-operation and the second one is ignored. Template ants each have one resource from the template as their cluster resource guiding what scents the ant follows. For the above template one ant would have s and the other o as cluster resource. Roaming the network to look for matching triples the rd-ants, like the out-ants, age with each hop. In order to find back to its origin the rd-ants use a memory where they add all visited nodes. The rd-ants’ behaviour is detailed in algorithm 2.

Algorithm 2 High level description of the rd ant’s algorithm

Variables:

age: realizes the ant’s aging mechanism
template: the ant’s template
cluster-resource: the ant’s cluster resource
memory: memory for visited nodes to find way back

1: Initialization: age is set to a given integer value > 0
2: while age > 0 do
3: Add the current node to memory
4: Look on current node, if there is a triple matching template
5: if matching triple is found then
6: Make a copy and use memory to return to the origin. Leave scent of cluster-resource on each node on the way back with one hop in a weaker quantity. Return the copy as result and die.
7: else
8: Select next node from neighborhood based on cluster-resource (according to eq. 15)
9: Move to selected node
10: age ← age - 1
11: end if
12: end while
13: Die

The ant behaviour for the in-operation is the same as for the rd-operation concerning the search for a match. The in-operation, however, has to remove the triple before returning it to the invoking process. Since there exist three copies of the triple, the removals have to be coordinated. In addition, the removal might be in competition with another in-operation that selected the same triple.

If there is one actual field in the template, eg. (s, p, o), one in-ant is working. When it finds a match, it takes the triple from its current node and generates two lock-ants. These will lock the other two copies on remote nodes.

The algorithm differs from the rd-algorithm in steps 5 and 6. They are replaced by the steps shown in algorithm 3. The generated lock-ants carry templates with three actual fields and also have a cluster resource. For example if an in-ant with template (s, p, o) (thus its cluster resource is s) finds a matching triple (s, p, o) it creates two lock-ants carrying template (s, p, o) and having p and o respectively as cluster resource. The algorithm of the

Algorithm 3 Additional steps for in-ants with templates that have one actual field

1: if matching triple is found then
2: Pick it up
3: Create two lock-ants which lock the other two copies of the triple.
4: Increase age to twice the lifetime of the lock-ants
5: while age > 0 do
6: if both lock ants returned then
7: Order them to delete their locked triples and return with the picked up triple to the origin by use of the memory. Leave scent of cluster-resource on each node on the way back. Return the triple as result and die.
8: else
9: Wait maximum hop time
10: age ← age - 1
11: end if
12: end while
13: if one lock-ant returned then
14: Order to unlock triple
15: end if
16: Drop triple and die
17: end if

lock-ants is shown in algorithm 4.

1) in-operation for templates with two actual fields:

When there are two actual fields in the template, two template ants are generated, which follow different scents. For example if the template is (s, p, o) one ant has p as cluster resource and the other one o. Their work has to be coordinated, otherwise they might compete for locking copies of the same triple during retrieval. The probability for such a situation is high since both template-ants originate from the same node. Also it must not happen that both ants remove three copies, since only one in-operation is executed.

In the extended algorithm, both ants know each other necessarily by some id. They follow the standard algorithm but perform an extended rd-operation in which the match gets locked whereas the lock information is extended by the id of the template ant.

This makes it possible that two in-ants that were generated for one in-operation lock the same copy so that they do not hinder each other by locking all the three copies of a triple. Since eventually only one of the two ants is allowed to remove its locked triple copies, this does not result in a conflict.

When the locking phase succeeds, the in-ant returns to its origin. Only the first successful ant is allowed to remove its locked triple copies. By depositing its id it informs the second ant about the successful execution of the in-operation such that the latter unlocks its own.

Like the algorithm for in-ants that carry a template with

maximum hop time is an empirical value for the longest time that an ant needs to switch from one node to another. Yet, in individual cases, it can be exceeded

© 2009 ACADEMY PUBLISHER
Algorithm 4 High-level description of the algorithm for
lock-ants which work for in-ants that have templates with
one actual field

Variables:
age: realizes the ant’s aging mechanism
template: the ant’s template
cluster-resource: the ant’s cluster resource
memory: memory for visited nodes to find way back

1: Initialization: age is set to a given integer value > 0
2: while age > 0 and no matching triple found do
3:   Add current node to memory
4:   Look for matching triple on current node
5: if matching triple is found that is not already locked
6:   Lock it
7:   else
8: Select next node from neighborhood based on
9:   cluster-resource (eq. 15)
10: Move to selected node
11: age ← age - 1
12: end if
13: if matching triple is found then
14:   Use memory to get back to in-ant
15: if in-ant still alive
16:   while no order from in-ant do
17:     if in-ant ordered to unlock then
18:       Use memory to get back to locked triple,
19:       unlock it and die
20:     end if
21:   use memory to get back to locked triple,
22:     remove it and die
23: end if
24: else
25: Use memory to get back to locked triple, unlock
26: it and die
27: end if
28: Die
29: end if

Algorithm 5 Additional steps used by in-ants that have
templates with two actual fields

1: if matching triple is found then
2: Lock it
3: Create two lock ants which lock the other two
4: copies of the triple.
5: Increase age to twice the lifetime of the lock ants
6: While age > 0 and both lock ants have not yet
7: returned do
8: Wait maximum hop time
9: age ← age - 1
10: end while
11: if both lock-ants returned then
12: Pick up triple
13: Use memory to return to origin
14: if second in-ant left its id on the original node
15: then
16: Create 3 unlock-ants with memory information which unlock
17: the three locked copies and die
18: else
19: Leave own id, create 3 remove-ants with
20: memory information which remove the three
21: locked copies, return triple as result and die
22: end if
23: end if

remove-ants are not detailed here since they simply use
a given memory to find the locked triples and unlock or
respectively remove them.

C. Triple Movement

Because of the probabilistic behaviour of the tuple
distribution algorithm some tuples are placed in clusters
where they do not fit. Another reason for a triple being
misplaced can be that an ant died before it could find a
suitable cluster. The cleaning ant’s task is to find triples
that are misplaced, to pick them up and carry them to
to better locations. Therefore the cleaning ant roams the
network examining the triples on the nodes. On each
visited node it determines the triple that is most dissimilar
to the other triples. Depending on the similarity between
the other triples among one another the cleaning ant picks
the triple to carry it to a more suitable location. The cleaning
algorithm is described in the following.

1) The cleaning ant is born on a node in the network.
2) While the ant carries no triple it roam the network
randomly and looks for misplaced triples. For that,
the ant determines the triple that is least similar to
the other triples on the current node. Because triples
were clustered in respect to only one resource it is
sufficient that a considered triple is similar to the other triples in respect to one of their resources. The ant computes the similarity sum $Sim$ which is introduced in section III-D for each resource of the regarded triple $t_i = (s_i, p_i, o_i)$. This is done by comparing the resource with all other triples $t_1 \ldots t_m$ on the node on the basis of the similarity function $sim_{trip-res}$ which is defined in formula 7. So it gets three similarity sums called $Sim_{subj}$, $Sim_{pred}$ and $Sim_{obj}$ for the subject, predicate and the object of the triple which are shown below.

$$Sim_{subj}^i = \sum_{j=1}^{m} sim_{trip-res}(s_i, t_j)$$  \hspace{1cm} (1)$$

$$Sim_{pred}^i = \sum_{j=1}^{m} sim_{trip-res}(p_i, t_j)$$ \hspace{1cm} (2)$$

$$Sim_{obj}^i = \sum_{j=1}^{m} sim_{trip-res}(s_i, t_j)$$ \hspace{1cm} (3)

The ant determines the maximum of the three sums which is called $Sim_{max}$.

$$Sim_{max}^i = \max(Sim_{subj}^i, Sim_{pred}^i, Sim_{obj}^i)$$ \hspace{1cm} (4)$$

$Sim_{max}$ is the similarity sum for the resource of the regarded triple $t_i$ that is most similar to the other triples on the node. It is likely that this is the resource, that was originally determined as the triple’s cluster resource, because the cluster resource is expected to be more similar to the other triples on the node than the other two resources of $t_i$. But this is not necessarily the case as the ant’s decisions are stochastic. $Sim_{max}$ is calculated for all triples $t_1 \ldots t_m$ on the node. The triple which has the lowest value for $Sim_{max}$ is the triple which is least similar to the other triples on the node and is regarded as the most unsuitable triple on the node. The similarity sum of this triple is called $Sim_{pick-up}$:

$$Sim_{pick-up} = \min(Sim_{max}^1, \ldots, Sim_{max}^n)$$ \hspace{1cm} (5)$$

Having determined the triple that fits least the decision for picking it up is given as follows.

$$D_{pick-up} = \begin{cases} 1 & \text{if} \, \frac{Sim_{pick-up}}{Sim_{max}} < \gamma \\ 0 & \text{else} \end{cases}$$ \hspace{1cm} (6)$$

with $\gamma \in [0, 1]$.

Thus the ratio of $Sim_{pick-up}$ and the average $Sim_{max}$ value of the other triples on the node is calculated. If the result is lower than a specified value $\gamma$ the triple is picked up. That means that if

the concerned triple is much more dissimilar to the other triples, than the other triples are among each other, the cleaning ant picks it up. For example if the sum $Sim_{max}$ of the concerned triple is 11 and the sum $Sim_{max}$ of the other triples on the node is 11.2 on average, this triple fits well into the cluster. In this example the ratio of 11 and 11.2 is 0.982 and very close to 1. $\gamma$ determines the upper bound for the triple being picked up.

3) If the cleaning ant decides to pick up the triple it behaves henceforward like a tuple ant which is looking for a suitable location for its triple and gets the renewed lifespan of a tuple ant. The resource of the triple for which the similarity sum was highest is determined as the cluster resource.

D. Drop Probability

Because triples are clustered in respect to only one of their resources, it is not necessary that triples are similar concerning all of their resources to form a cluster. It is sufficient that the triples are similar concerning one of their resources. Thus a tuple ant, which is looking for a convenient location for its triple, need not consider all resources of the triples on a node to compute the drop probability. It takes only the resource of a regarded triple into account that is most similar to its cluster resource. The tuple ant computes the namespace similarity $sim_{res}$, that is introduced in section IV, between its cluster resource $r$ and all resources of a regarded triple $T = (s, p, o)$ and determines the maximum value of the results. The maximum value forms the similarity $sim_{trip-res}$ between the cluster resource $r$ and the concerned triple $T$ and is given by the following formula.

$$sim_{trip-res}(r, T) = \max(sim_{res}(r, t_i)) \, t_i \in \{s, p, o\}$$ \hspace{1cm} (7)$$

The ant compares all triples on the current node with its cluster resource on the basis of $sim_{trip-res}$ adding up the results of the comparisons. This results in the similarity sum $Sim$, which is shown in formula 8.

$$Sim = \sum_{t_i \in TS} sim_{trip-res}(r, t_i)$$ \hspace{1cm} (8)$$

On the basis of the similarity sum $Sim$ the tuple ant determines the drop probability. There are three different drop probabilities which will be introduced in the following. The first and the second drop probability are closely related to the drop probabilities used in the implementation of SwarmLinda, only that the concentration $C$ is replaced by the similarity sum $Sim$.

1) The probability $P_{drop}$ to drop a triple on a current node is calculated from the similarity sum $Sim$ and the number of steps $K$ that the ant still can take in the network before it dies.

$$P_{drop} = \left( \frac{Sim}{Sim + K} \right)^2$$ \hspace{1cm} (9)$$
Adding \( K \) in the denominator causes that the drop probability increases with the ant’s age. If \( K = 0 \), that means that the ant can make no further steps, the probability that it drops the triple is 1. The more similar triples there are on a node the less the drop probability is influenced by the ant’s age.

2) The second drop probability additionally considers the total number of triples on a node. This is done by dividing the similarity sum \( Sim \) by the total number of triples on a node and applying the sigmoid function to the result. So we get a modified similarity sum named \( Sim_{mod} \). Taking the original formula 9 which is used for \( P_{drop1} \) and replacing \( Sim \) by \( Sim_{mod} \) results in the second drop probability \( P_{drop2} \).

\[
P_{drop2} = \left( \frac{Sim_{mod1}}{Sim_{mod1} + K} \right)^2 \tag{10}
\]

with

\[
Sim_{mod1} = F_{sig}(\frac{Sim}{N})Sim \tag{11}
\]

and

\[
F_{sig}(x) = \frac{1}{1 + e^{-(20x-10)}}
\]

3) For the third drop probability the similarity sum \( Sim \) is divided by the total number of triples on the node and the result is raised to the power of 4. The power of 4 causes the probability for dropping the triple to decrease if the ratio is not 1 (or 0). The age of the ant has no influence on this drop probability if there are triples on the node. If the node is empty the probability for dropping the triple increases with the age of the ant. For that purpose the difference of \( lifespan \) and \( K \) is divided by \( lifespan \), in which \( K \) is the number of steps that the ant still can take in the network and \( lifespan \) is the maximal age, measured in steps, that the ant can reach.

\[
P_{drop3} = \begin{cases} 
(Sim_{mod2})^4 & \text{if node is empty} \\
(lifespan-K)^4 & \text{else}
\end{cases} \tag{12}
\]

with

\[
Sim_{mod2} = \frac{Sim}{N} \tag{13}
\]

E. Path selection

In order to decide which node to visit next, the ant examines the scents as well as the triples on all adjacent nodes. The similarity to its cluster resource as well as the strength of the scents on a certain node effect the probability to visit it. For each adjacent node the pheromone sum \( Ph \) is calculated as follows. Let \( ph_1 \ldots ph_n \) be the scents on a node and \( st_1 \ldots st_n \) the respective scent strengths. Let \( r \) be the ant’s cluster resource.

\[
Ph = \sum_{i=1}^{n} st_i \text{sim}_{res}(r, ph_i) \tag{14}
\]

Additionally the similarity sum \( Sim \) which was presented in section III-D is computed for each neighbor, so that if there are triples on a certain node which are similar to the ant’s cluster resource this also increases the probability to visit it. The probability for an ant to move from its current node \( i \) to an adjacent node \( j \) in the neighborhood \( NH(i) \) is calculated as follows.

\[
P_{ij} = \frac{Sim_j + Ph_j}{\sum_{n \in NH(i)} Sim_n + Ph_n} \tag{15}
\]

IV. A SIMILARITY MEASURE ON URIs

The swarming algorithms presented depend on the notion of a similarity between two or more RDF triples. Therefore the choice of a suited similarity measure is crucial. The RDFSwarms presented here is designed as a storage service for triples and not as a reasoning infrastructure. Therefore, we only consider triples by structure and by their concrete content without interpreting them semantically.\(^3\)

A. Similitude of Host

To compare the host-components of two URIs we consider their “.”-separated domain labels ([10], sec 3.1). Starting with the hierarchical highest label, we compare them pairwise. Let \( n_1 \ldots n_k \) be the domain labels of URI\(_1\) and \( n_1 \ldots n_l \) those of URI\(_2\). The host similarity then is defined by

\[
sim_{host} = \sum_{i=1}^{\min(k,l)} e_i edit(m_{k-i}, n_{l-i}) \tag{16}
\]

with

\[
e_i = \frac{2\max(k,l)-i}{2\max(k,l)-1}
\]

as a weighting function and \( edit \) as the normalized Levenshtein-distance of two strings. The weighting function values a domain label a level higher in the hierarchy with doubles weight.

\(^3\)A semantic interpretation would need access to shared ontologies which are global to the overall system. A decentralized reasoning approach is more complex and will be described as a further extension to RDFSwarms separately.
B. Similarity of Path

The components of the URL-path are compared pairwise. Let \( m_1 \ldots m_k \) be the path segments of URL_1 and \( n_1 \ldots n_l \) those of URL_2. The path similarity then is defined as

\[
sim_{\text{path}} = \frac{\min(k,l)}{k+l} \sum_{i=1}^{\min(k,l)} c_i \text{edit}(m_i, n_i)
\]

with

\[
c_i = \frac{2^{\max(k,l)-i} - 1}{2^{\max(k,l)} - 1}
\]

as a weighting function and \( \text{edit} \) as the normalized Levenshtein-distance of two strings. As a result of the weighting function, a path segment on level higher in the hierarchy gets twice a weight.

If one URI contains a fragment part we can apply an extended similarity function on fragments as follows

\[
sim_{\text{path}} = \frac{\min(k,l)}{k+l} \sum_{i=1}^{\min(k,l)} c_i \text{edit}(m_i, n_i) + c_{\min(k,l)+1} \text{edit}(\text{frag}_{\text{url}1}, \text{frag}_{\text{url}2})
\]

with

\[
c_i = \frac{2^{\max(k,l)+1-i} - 1}{2^{\max(k,l)+1} - 1}
\]

as a weighting function and \( \text{edit} \) as the normalized Levenshtein-distance of two strings and \( \text{frag}_{\text{url}1} \) and \( \text{frag}_{\text{url}2} \) being the respective fragments. If one URI does not contain a fragment, we compare with the empty string.

C. Similarity of User Info

To compare the user info of a mailto:-URI to the path of a hierarchical URI we consider the path segments. The user info is compared with the hierarchical highest segment for equality. The earlier die comparison succeeds, the higher the similarity gets.

Let \( n_1 \ldots n_k \) be the path segments of URL_2 with \( n_k \) as a fragment if present. Let \( u \) be the user info of URL_1, \( \text{sim}_{\text{URL1-Path}} \) is then calculated as

\[
\text{sim}_{\text{URL1-Path}} = \sum_{i=1}^{k} c_i f(u, n_i)
\]

\[
f(u, n_i) = \begin{cases} 
1, & \text{if } u = n_i \text{ und } f(u, n_1) = \ldots f(u, n_{i-1}) = 0 \\
0, & \text{else}
\end{cases}
\]

D. Overall similarity

The results of comparing the three URI components are combined into an overall similarity by weighting them. Components compared earlier are weighted higher. Let \( m_1 \ldots m_k \) be the results of each component comparison. Then the overall similarity of two URIs is given by

\[
sim_{\text{gesamt}} = \sum_{i=1}^{k} c_i n_i
\]

\[
c_i = \frac{a^i}{\sum_{j=1}^{k} a^j} = \frac{a^i}{a^{k+1} - 1} = \frac{a^i(a-1)}{a^{k+1}}
\]

\( a \) is the base for the weight. For hierarchical URIs the host component should be weighted much higher than the path component. Only if the hosts are equal, the path should differentiate the URIs. We set \( a \) to 9 to achieve this. For the two other comparison of two mailto:-URIs or of a mailto:-URI with a hierarchical URI \( a \) is set to 2 so that the host is weighted twice as much as the user-info.

V. IMPLEMENTATION

The RDFSwarms system has been implemented as a simulation with NetLogo and Eclipse. The behaviour of the ants and the GUI use the NetLogo [11] environment while the resource comparisons and data collection have been implemented in Java and integrated as NetLogo extensions. We used the SimMetrics library [12] for functions such as the computation of the Levenshtein distance.

Figure 1 shows an aperture of the resulting GUI for experimenting with the system and taking measures. The simulation allows to import RDF triples and generate matching templates from them. Out, in and rd-operations can be applied to an imported network. These are performed by their relevant ant type which carry the imported RDF-triples or respectively the generated templates by using the corresponding algorithms (see section III). During the execution the ants can be observed roaming the network. Monitors show the current state of the system and information about the ants’ success and performance. Several parameters like the ants’ drop-probability or maximal lifetime can be set. Also the cleaning ants are realized in the simulation and there are different hard-coded test-runs for evaluating the different swarm algorithms. These were used for the measurements which are described in the next section.

VI. MEASUREMENTS

A. Evaluation of the out and rd algorithms

We used RDF data from DBPedia [13] as well as OWL data from LUBM [14] which were serialized to RDF beforehand for our evaluation. Five test runs were executed on a network with 50 nodes. For each test run 300 triples were randomly selected from the test data. Tuple ants distributed these triples in three runs each time using a different drop probability (see III-D).
In the following we refer to the tuple ants as swarm1, swarm2 and swarm3, where each swarm uses its own drop probability $p_{\text{drop}1}$, $p_{\text{drop}2}$ and $p_{\text{drop}3}$ (see formulas 9, 10 and 12). In addition the triples were randomly distributed once in each test run. To make the random distribution comparable to the distribution carried out by the tuple ants in either case three copies were stored.

After each triple distribution the similarity of the triples and resources on the nodes was calculated. This was done with evaluation measures which are introduced in VI-C. Afterwards 50 rd-operations with templates that matched the triples in the network were executed. We logged how many rd-ants found a matching triple and how many steps it took them to find it.

The average results from the five test runs are shown in table I and figure 2 and in table II and figure 3. In table I average denotes the average similarity of the triples on the nodes in the network and median respectively (see formulas 27 and 28). The average similarity of the resources on the nodes average-res is calculated as in formula 30. In table II success ants denotes the number of rd-ants which found a matching triple, failed ants is the number of ants which did not find a triple. steps success ants is the average number of steps that a successful in ant took before finding a matching triple.

B. Evaluation of the Cleaning Algorithm

For the evaluation of the cleaning algorithm 300 triples were distributed in the network by swarm3. Then the similarity of the triples on the nodes was calculated. Afterwards 50 cleaning ants were sent into the network, executing 50 cleaning steps. For the pick-up decision $\gamma$ was set to 0.8 (for further details see III-C). Then again the similarity of the triples on the nodes was calculated. The average results of overall five test runs are shown in table III and figure 4.

C. Evaluation Measures

1) Similarity of Triples: Because in RDFSwarms triples are clustered with respect to only one resource it is expected that clusters of triples arise that are similar to at least one of their resources. To measure the clustering
success of the tuple distribution algorithm the similarity measure \( \text{sim}_{\text{triple}} \) is introduced which compares two triples \( T_1 = (s_1, p_1, o_1) \) and \( T_2 = (s_2, p_2, o_2) \) by computing \( \text{sim}_{\text{res}} \) for their resources and determining the maximum value (24).

\[
\text{sim}_{\text{triple}}(T_1, T_2) = \max(\text{sim}_{\text{res}}(t_i, u_j)) \tag{24}
\]

with \( t_i \in \{s_1, p_1, o_1\} \) \( u_j \in \{s_2, p_2, o_2\} \)

Successfully clustered triples are expected to have increased values for \( \text{sim}_{\text{triple}} \). For the evaluation all triples \( t_1 \ldots t_k \) on a node \( n \) are compared among one another on the basis of \( \text{sim}_{\text{triple}} \) and the average value of the comparisons is calculated. This gives the average similarity on the node:

\[
\text{avg-sim}_n(n) = \frac{\sum_{i=1}^{k-1} \sum_{j=1}^{k-1} \text{sim}_{\text{triple}}(t_j, t_j + 1)}{\text{Number of comparisons}} = \frac{\sum_{i=1}^{k-1} \sum_{j=1}^{k-1} \text{sim}_{\text{triple}}(t_j, t_j + 1)}{(k - 1)(k - 2)} \tag{25}
\]

In addition the median similarity on the node,

\[
\text{med-sim}_n(n) = \text{median} \left( \text{sim}_{\text{triple}}(t_i, t_j) \right) \tag{26}
\]

s.t. \( i, j < k, i < j \), is determined. Eventually the average similarities for all nodes \( \text{node}_1 \ldots \text{node}_m \) in the network are calculated

\[
\text{avg-sim} = \frac{\sum_{i=1}^{m} \text{avg-sim}_n(\text{node}_i)}{m} \tag{27}
\]

Also the average figure for the \( \text{med-sim}_n \) values of all nodes in the network is computed.

\[
\text{med-sim} = \frac{\sum_{i=1}^{m} \text{med-sim}_n(\text{node}_i)}{m} \tag{28}
\]

2) Similarity of Resources: It is likely that also the similarity of the resources on the nodes is slightly increased, if the triples are clustered successfully by the tuple ants. To evaluate the average similarity of the resources \( r_1 \ldots r_k \) on a node \( n \) we compute

\[
\text{avg-sim}_{\text{res}}_n(n) = \frac{\sum_{i=1}^{k-1} \sum_{j=1}^{k-1} \text{sim}_{\text{res}}(r_j, r_j + 1)}{\text{Number of comparisons}} = \frac{\sum_{i=1}^{k-1} \sum_{j=1}^{k-1} \text{sim}_{\text{res}}(r_j, r_j + 1)}{(k - 1)(k - 2)} \tag{29}
\]

Eventually we compute the average similarity of resources on the nodes \( n_1 \ldots n_m \) in the network as

\[
\text{avg-sim}_{\text{res}} = \frac{\sum_{i=1}^{m} \text{avg-sim}_{\text{res}}_n(n_i)}{m} \tag{30}
\]
D. Discussion

It can be observed that the distribution of the triples by tuple ants with all three drop probabilities effects an increase of the similarity on the nodes which is an indicator for successful cluster formation. It is peculiar that the \( rd \)-operations become much more efficient, if the triples are distributed by tuple ants in comparison with the random distribution. There are considerably more successful \( rd \)-ants and the search time after distribution with \( p_{\text{drop1}} \) and \( p_{\text{drop3}} \) is halved compared to the random distribution. Analysing the different drop probabilities it can be observed that \( p_{\text{drop2}} \) effects a higher similarity on the nodes than \( p_{\text{drop1}} \) but the \( rd \)-ants perform worse afterwards. Altogether \( p_{\text{drop2}} \) produces worse results than the two other drop probabilities. The measurement that the cleaning algorithm increases the similarity on the nodes shows that the misplaced triples are successfully carried to more suitable clusters.

VII. Conclusion and Outlook

In this paper we have reported on our approach to a scalable storage service for semantic information. At the core is the usage of mechanisms of selforganization to enable a scalable distributed service. We have presented decentralized algorithms that implement the distribution and retrieval of triples. All of these were based on purely local decisions.

For the decision on what is similar, we have developed a measure which considers the URIs of resources referenced in triples only. This has the advantage that the comparison is most local in the sense that no global ontology is used. Our assumption is that the syntactical measure reflects a semantical similarity.

The evaluation has shown that the algorithms and the selected similarity mechanism leads to less entropy in the network which enables a more effective triple retrieval. We conclude that our algorithms are parameterized with a suited similarity measure to achieve a significant advantage in triple retrieval in our infrastructure.

The next step is the extension of our algorithms with a similarity measure and suited mechanisms that do consider ontological information for determining similarity.

References


Prof. Dr.-Ing. Robert Tolksdorf graduated in Computer Science at Technische Universität Berlin and was appointed professor in 2002 at Freie Universität Berlin. His areas of work and interest are Semantic Technologies, Coordination and Selforganization.

Anne Augustin was born in Berlin, Germany. She is currently a Masters student at the Department of Computer Science at Freie Universität Berlin. She pursued her Bachelor studies in computer science at the University of Freiburg and Freie Universität Berlin.